

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

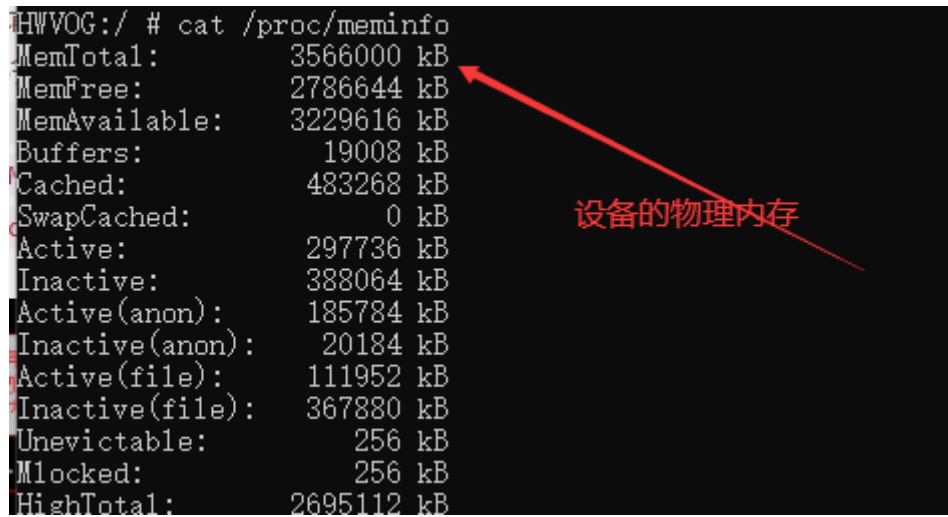
- Java申请的内存存在 vm heap 上，所以如果 java 申请的内存大小超过 VM 的逻辑内存限制,就会出现内存溢出的异常。
- native层内存申请不受其限制,native 层受 native process 对内存大小的限制

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
```

```
命令提示符 - adb shell
#
# ADDITIONAL_BUILD_PROPERTIES
#
ro.com.android.dateformat=MM-dd-yyyy
ro ril.hsxpa=1
ro ril.gprsclass=10
keyguard.no_require_sim=true
ro.com.android.dataroaming=true
media.sf.hwaccel=1
media.sf.omx-plugin=libffmpeg_omx.so
media.sf.extractor-plugin=libffmpeg_extractor.so
dalvik.vm.heapstartsize=16m
dalvik.vm.heapgrowthlimit=128m
dalvik.vm.heapsize=192m
dalvik.vm.heaptargetutilization=0.75
dalvik.vm.heapminfree=512k
dalvik.vm.heapmaxfree=8m
ro.dalvik.vm.isa.arm=x86
ro.enable.native.bridge.exec=1
wifi.interface=eth1
ro.carrier=unknown
ro.config.notification_sound=OnTheHunt.ogg
ro.config.alarm_alert=Alarm_Classic.ogg
persist.sys.dalvik.vm.lib.2=libart.so
dalvik.vm.isa.x86.variant=x86
dalvik.vm.isa.x86.features=default
dalvik.vm.lockprof.threshold=500
net.bt.name=Android
dalvik.vm.stack-trace-file=/data/anr/traces.txt
HWVOG:/ #
```



```

HWVOG:/ # cat /proc/meminfo
MemTotal:       3566000 kB
MemFree:        2786644 kB
MemAvailable:   3229616 kB
Buffers:        19008 kB
Cached:         483268 kB
SwapCached:      0 kB
Active:         297736 kB
Inactive:       388064 kB
Active(anon):   185784 kB
Inactive(anon): 20184 kB
Active(file):   111952 kB
Inactive(file): 367880 kB
Unevictable:    256 kB
Mlocked:        256 kB
HighTotal:      2695112 kB

```

设备的物理内存

2. 通过代码获取

```

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

```

3. 可以修改吗?

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

```

1 | int AndroidRuntime::startVm(JavaVM** pJavaVM, JNIEnv** pEnv, bool zygote)
2 | {
3 |     /*
4 |
5 |     * The default starting and maximum size of the heap. Larger
6 |     * values should be specified in a product property override.
7 |     */
8 |     parseRuntimeOption("dalvik.vm.heapstartsize", heapstartsizeOptsBuf,
   | "-Xms", "4m");
9 |     parseRuntimeOption("dalvik.vm.heapsize", heapsizeOptsBuf, "-Xmx",
   | "16m"); //修改这里
10 |     * }

```

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

```

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

```

内存指标概念

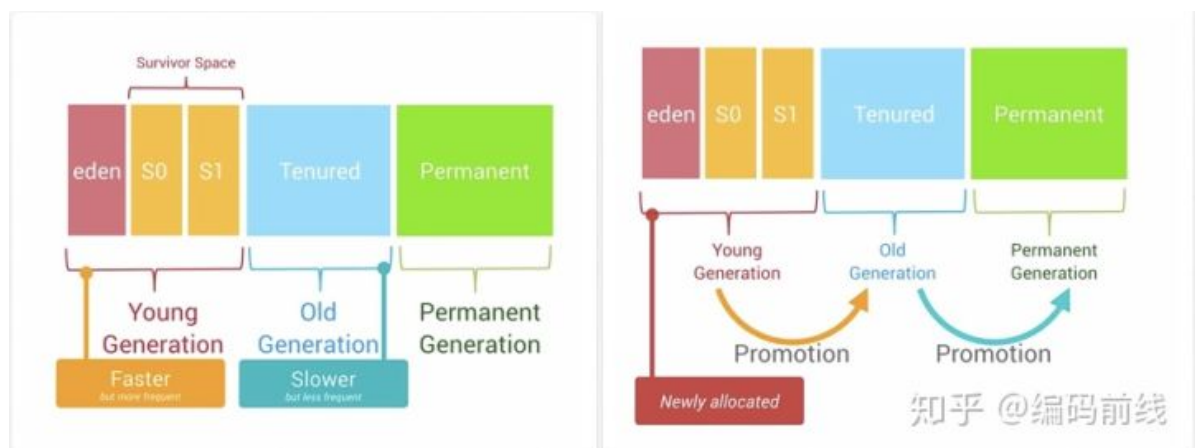
Item	全称	含义	等价
USS	Unique Set Size	物理内存	进程独占的内存
PSS	Proportional Set Size	物理内存	PSS= USS+ 按比例包含共享库
RSS	Resident Set Size	物理内存	RSS= USS+ 包含共享库
VSS	Virtual Set Size	虚拟内存	VSS= RSS+ 未分配实际物理内存

总结:VSS >= RSS >= PSS >= USS,但/dev/kgs1-3d0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

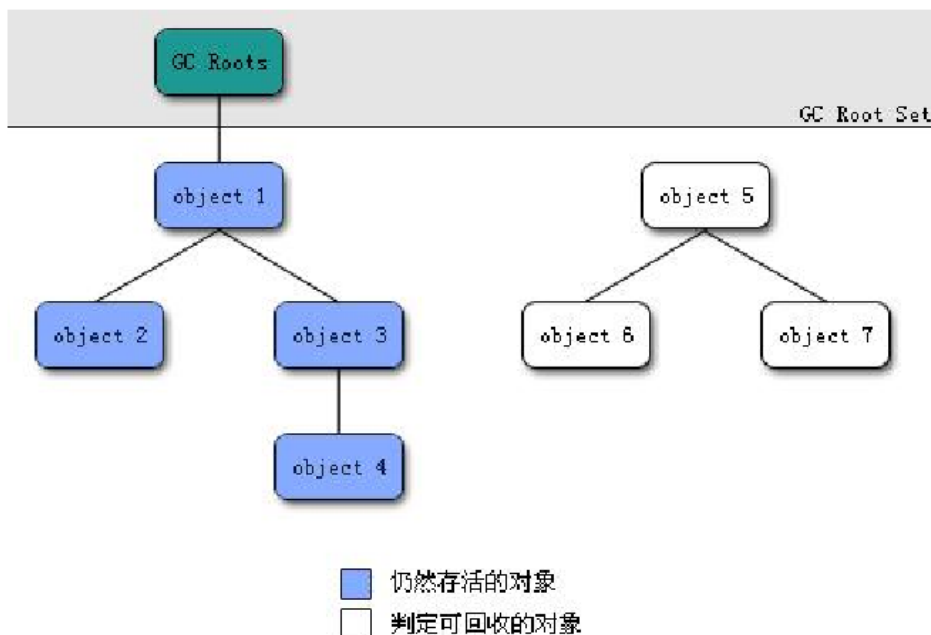
由于其对象存活时间较长，比较稳定，因此采用Mark（标记）算法（扫描出存活的对象，然后再回收未被标记的对象，回收后对空出的空间要么合并，要么标记出来便于下次分配，以减少内存碎片带来的效率损耗）来回收。

GC类型

在Android系统中，GC有三种类型：

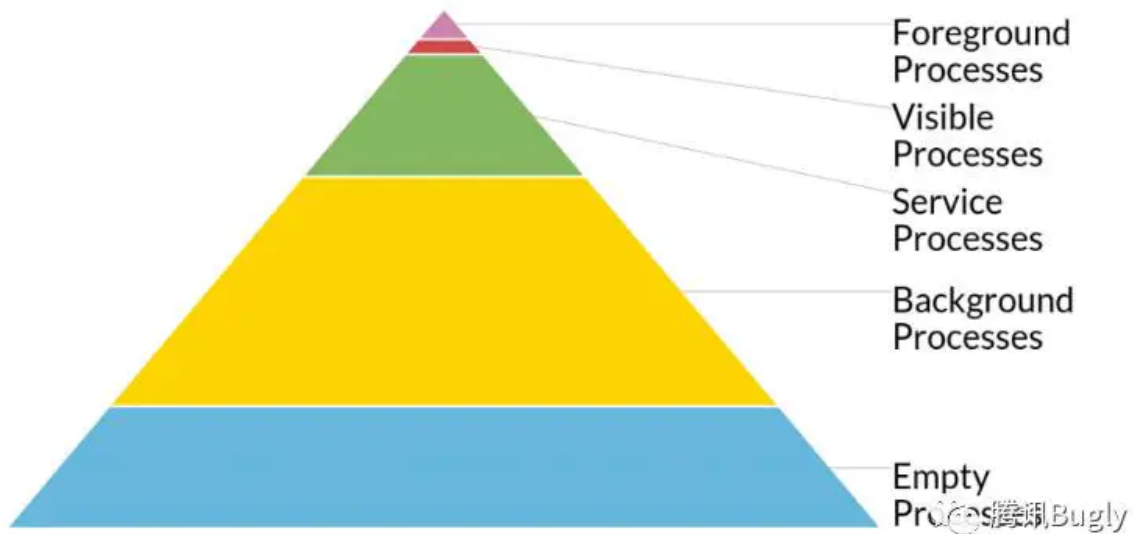
- kGcCauseForAlloc：分配内存不够引起的GC，会Stop World。由于是并发GC，其它线程都会停止，直到GC完成。
- kGcCauseBackground：内存达到一定阈值触发的GC，由于是一个后台GC，所以不会引起Stop World。
- kGcCauseExplicit：显示调用时进行的GC，当ART打开这个选项时，使用System.gc时会进行GC。

可达性分析与GCRoots



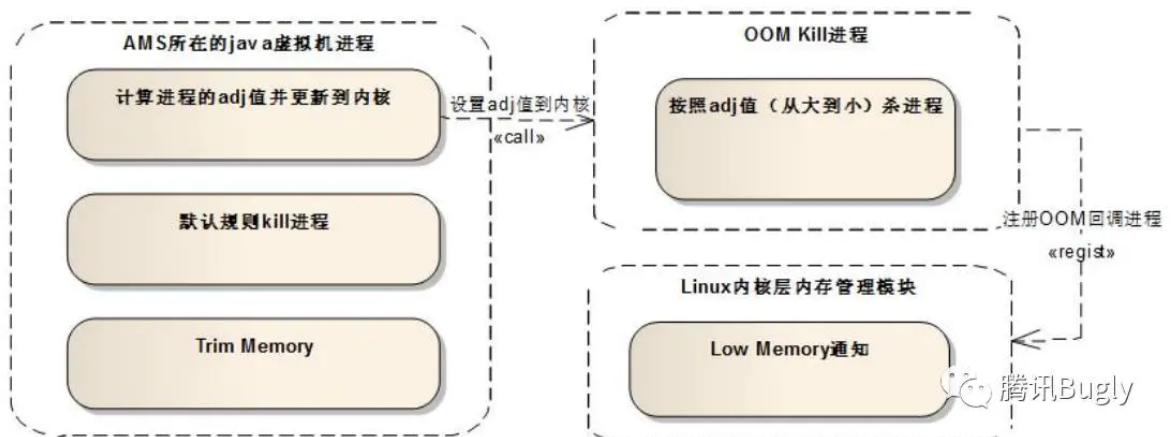
Android低内存杀进程机制

Android基于进程中运行的组件及其状态规定了默认五个回收优先级：



- 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

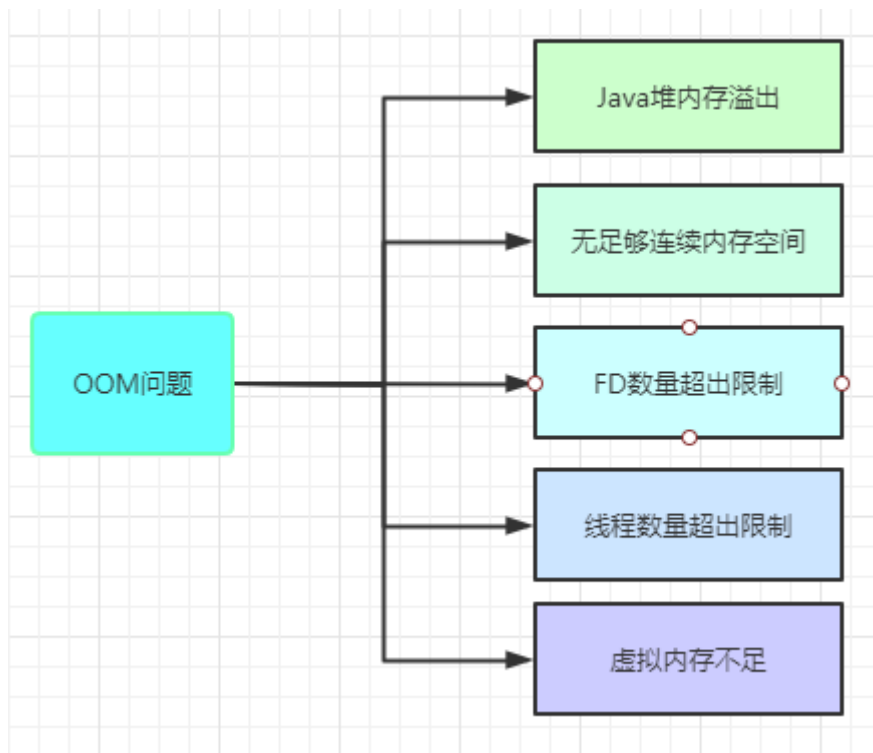
OOM (OutOfMemoryError) 内存溢出错误，在常见的Crash疑难排行榜上，OOM绝对可以名列前茅并且经久不衰。因为它发生时的Crash堆栈信息往往不是导致问题的根本原因，而只是压死骆驼的最后一根稻草

```
Process: com.enjoy.memory, PID: 64/2
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$500(ActivityThread.java:173)
at android.app.ActivityThread$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 \geq 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 $32586+8989+(\text{新分配}23975)=65550>64\text{M}$ 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 \geq 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

```

1 void Thread::ThrowOutOfMemoryError(const char* msg) {
2     LOG(WARNING) << StringPrintf("Throwing OutOfMemoryError \"%s\" \"%s\"",
3         msg, (tls32_.throwing_OutOfMemoryError ? " (recursive case)" : ""));
4     if (!tls32_.throwing_OutOfMemoryError) {
5         tls32_.throwing_OutOfMemoryError = true;
6         ThrowNewException("Ljava/lang/OutOfMemoryError;", msg);
7         tls32_.throwing_OutOfMemoryError = false;
8     } else {
9         Dump(LOG_STREAM(WARNING)); // The pre-allocated OOME has no stack, so
        help out and log one.
10        SetException(Runtime::Current()->GetPreAllocatedOutOfMemoryError());
11    }
12 }

```

堆内存分配失败

/art/runtime/gc/heap.cc

```

1 void Heap::ThrowOutOfMemoryError(Thread* self, size_t byte_count,
2     AllocatorType allocator_type) {
3     // If we're in a stack overflow, do not create a new exception. It would
4     // require running the
5     // constructor, which will of course still be in a stack overflow.
6     if (self->IsHandlingStackOverflow()) {
7         self->SetException(
8             Runtime::Current()-
9             >GetPreAllocatedOutOfMemoryErrorWhenHandlingStackOverflow());
10        return;
11    }
12
13    std::ostringstream oss;
14    size_t total_bytes_free = GetFreeMemory();
15    //为对象分配内存时达到进程的内存上限
16    oss << "Failed to allocate a " << byte_count << " byte allocation with "
17    << total_bytes_free
18    << " free bytes and " << PrettySize(GetFreeMemoryUntilOOM()) << "
19    until OOM,"
20    << " target footprint " <<
21    target_footprint_.load(std::memory_order_relaxed)
22    << ", growth limit "
23    << growth_limit_;
24
25    //没有足够大小的连续地址空间
26    // There is no fragmentation info to log for large-object space.
27    if (allocator_type != kAllocatorTypeLOS) {
28        CHECK(space != nullptr) << "allocator_type:" << allocator_type
29        << " byte_count:" << byte_count
30        << " total_bytes_free:" << total_bytes_free;
31        space->LogFragmentationAllocFailure(oss, byte_count);
32    }
33 }

```

创建线程失败

/art/runtime/thread.cc

```

1 void Thread::CreateNativeThread(JNIEnv* env, jobject java_peer, size_t
  stack_size, bool is_daemon) {
2     CHECK(java_peer != nullptr);
3     Thread* self = static_cast<JNIEnvExt*>(env)->GetSelf();
4
5     // TODO: remove from thread group?
6     env->SetLongField(java_peer,
  wellKnownClasses::java_lang_Thread_nativePeer, 0);
7     {
8         std::string msg(child_jni_env_ext.get() == nullptr ?
9             StringPrintf("Could not allocate JNI Env: %s", error_msg.c_str()) :
10             StringPrintf("pthread_create (%s stack) failed: %s",
11                 PrettySize(stack_size).c_str(),
  strerror(pthread_create_result)));
12         ScopedObjectAccess soa(env);
13         soa.Self()->ThrowOutOfMemoryError(msg.c_str());
14     }

```

Android 内存分析命令介绍

常用的内存调优分析命令：

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

dumphys 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，该统计方法包括按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，则分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: $\text{swappable_pss} = (\text{sharing_proportion} * \text{shared_clean}) + \text{private_clean}$

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Native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 \approx 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

Objects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：

```

1 root@Phone:/# procrank
2   PID      Vss      Rss      PSS      Uss  cmdline
3   4395    2270020K  202312K  136099K  121964K  com.android.systemui
4   1192    2280404K  147048K   89883K   84144K  system_server
5  29256    2145676K  97880K   44328K   40676K  com.android.settings
6   501     1458332K  61876K   23609K    9736K  zygote
7   4239    2105784K  68056K   21665K   19592K  com.android.phone
8   479     164392K  24068K   17970K   15364K  /system/bin/mediaserver
9   391     200892K  27272K   15930K   11664K  /system/bin/surfaceflinger
10  ...
11 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

```
1 | 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)：

```

1 root@phone:/ # cat /proc/meminfo
2 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）
3 MemFree:       1020.708 kB //RAM未使用的大小
4 Buffers:       75.104 kB //用于文件缓冲
5 Cached:        448.244 kB //用于高速缓存
6 SwapCached:    0 kB //用于swap缓存
7
8 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
9 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
10 Active(anon):  700.744 kB //Active = Active(anon) + Active(file)
11 Inactive(anon): 228 kB //Inactive = Inactive(anon) + Inactive(file)
12 Active(file):  132.156 kB
13 Inactive(file): 390.900 kB
14
15 Unevictable:   0 kB
16 Mlocked:      0 kB
17
18 SwapTotal:     524.284 kB //swap总大小
19 SwapFree:      524.284 kB //swap可用大小
20 Dirty:         0 kB //等待往磁盘回写的大小
21 Writeback:     0 kB //正在往磁盘回写的大小
22
23 AnonPages:     700.700 kB //匿名页，用户空间的页表，没有对应的文件
24 Mapped:        187.096 kB //文件通过mmap分配的内存，用于map设备、文件或者库
25 Shmem:         .312 kB
26
27 Slab:          91.276 kB //kernel数据结构的缓存大小，
    Slab=SReclaimable+SUnreclaim
28 SReclaimable:  32.484 kB //可回收的slab的大小
29 SUnreclaim:    58.792 kB //不可回收slab的大小
30
31 KernelStack:   25.024 kB

```

```

32 PageTables:      23.752 kB //以最低的页表级
33 NFS_Unstable:    0 kB //不稳定页表的大小
34 Bounce:          0 kB
35 WritebackTmp:    0 kB
36 CommitLimit:     1952.800 kB
37 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
38
39 VmallocTotal:    251658.176 kB //总分配的虚拟地址空间
40 VmallocUsed:     166.648 kB //已使用的虚拟地址空间
41 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：

```

1 root@phone:/proc/sys/vm # free
2              total        used        free      shared    buffers
3 Mem:      2857032      1836040      1020992          0       75104
4 -/+ buffers:          1760936      1096096
5 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

```
1 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  RSS  PSS  shared  shared  private  private  swap  object
addr   addr   size                                     clean  dirty  clean  dirty
-----
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：

```
1 Usage: vmstat [ -n iterations ] [ -d delay ] [ -r header_repeat ]
2     -n iterations      数据循环输出的次数
3     -d delay           两次数据间的延迟时长(单位: s)
4     -r header_repeat   循环多少次，再输出一头信息行
```

输入结果：

```
1 root@phone:/ # vmstat
2 procs  memory                      system                      cpu
3  r  b   free mapped   anon   slab   in  cs  flt  us ni sy id wa ir
4  2  0 663436 232836 915192 113960 196 274  0  8  0  2 99  0  0
5  0  0 663444 232836 915108 113960 180 260  0  7  0  3 99  0  0
6  0  0 663476 232836 915216 113960 154 224  0  2  0  5 99  0  0
7  1  0 663132 232836 915304 113960 179 259  0 11  0  3 99  0  0
8  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>

GC Log

GC Log分为Dalvik和ART的GC日志

ART的日志与Dalvik的日志差距非常大，除了格式不同之外，打印的时间也不同，非要在慢GC时才打印除了。下面我们看看这条ART GC Log：

Explicit	(full)	concurrent mark sweep GC	freed 104710 (7MB) AllocSpace objects,	21 (416KB) LOS objects,	33% free,25MB/38MB	paused 1.230ms total 67.216ms	
GC产生的原因	GC类型	采集方法	释放的数量和占用的空间	释放的大对象数量和所占用的空间	堆中空闲空间的百分比和（对象的个数）/（堆的总空间）	暂停耗时	

GC产生的原因如下：

- Concurrent、Alloc、Explicit跟Dalvik的基本一样，这里就不重复介绍了。
- NativeAlloc：Native内存分配时，比如为Bitmaps或者RenderScript分配对象，这会导致Native内存压力，从而触发GC。
- Background:后台GC，触发是为了给后面的内存申请预留更多空间。
- CollectorTransition：由堆转换引起的回收，这是运行时切换GC而引起的。收集器转换包括将所有对象从空闲列表空间复制到碰撞指针空间（反之亦然）。当前，收集器转换仅在以下情况下出现：在内存较小的设备上，App将进程状态从可察觉的暂停状态变更为可察觉的非暂停状态（反之亦然）。
- HomogeneousSpaceCompact：齐性空间压缩是指空闲列表到压缩的空闲列表空间，通常发生在当App已经移动到可察觉的暂停进程状态。这样做的主要原因是减少了内存使用并对堆内存进行碎片整理。
- DisableMovingGc：不是真正的触发GC原因，发生并发堆压缩时，由于使用了
- GetPrimitiveArrayCritical，收集会被阻塞。一般情况下，强烈建议不要使用
- GetPrimitiveArrayCritical，因为它在移动收集器方面具有限制。
- HeapTrim：不是触发GC原因，但是请注意，收集会一直被阻塞，直到堆内存整理完毕。

GC类型如下：

- Full：与Dalvik的FULL GC差不多。
- Partial：跟Dalvik的局部GC差不多，策略时不包含Zygote Heap。
- Sticky：另外一种局部中的局部GC，选择局部的策略是上次垃圾回收后新分配的对象。

GC采集的方法如下：

- mark sweep：先记录全部对象，然后从GC ROOT开始找出间接和直接的对象并标注。利用之前记录的全部对象和标注的对象对比，其余的对象就应该需要垃圾回收了。
- concurrent mark sweep：使用mark sweep采集器的并发GC。
- mark compact：在标记存活对象的时候，所有的存活对象压缩到内存的一端，而另一端可以更加高效地被回收。
- semispace：在做垃圾扫描的时候，把所有引用的对象从一个空间移到另外一个空间，然后直接GC剩余在旧空间中的对象即可。

通过GC日志，我们可以知道GC的量和它对卡顿的影响，也可以初步定位一些如主动调用GC、可分配的内存不足、过多使用Weak Reference等问题。

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BroadcastReceiver、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。

Bitmap

Bitmap相关方法总结

Bitmap

- `public void recycle()` // 回收位图占用的内存空间，把位图标记为Dead
- `public final boolean isRecycled()` //判断位图内存是否已释放
- `public final int getWidth()` //获取位图的宽度
- `public final int getHeight()` //获取位图的高度
- `public final boolean isMutable()` //图片是否可修改
- `public int getScaledWidth(Canvas canvas)` //获取指定密度转换后的图像的宽度
- `public int getScaledHeight(Canvas canvas)` //获取指定密度转换后的图像的高度
- `public boolean compress(CompressFormat format, int quality, OutputStream stream)` //按指定的图片格式以及画质，将图片转换为输出流
- `public static Bitmap createBitmap(Bitmap src)` //以src为原图生成不可变得新图像
- `public static Bitmap createScaledBitmap(Bitmap src, int dstwidth, int dstheight, boolean filter)` //以src为原图，创建新的图像，指定新图像的高宽以及是否可变。
- `public static Bitmap createBitmap(int width, int height, Config config)` //创建指定格式、大小的位图
- `public static Bitmap createBitmap(Bitmap source, int x, int y, int width, int height)` //以source为原图，创建新的图片，指定起始坐标以及新图像的高宽。

BitmapFactory工厂类

Option 参数类

- `public boolean inJustDecodeBounds` //如果设置为true，不获取图片，不分配内存，但会返回图片的高度宽度信息。如果将这个值置为true，那么在解码的时候将不会返回bitmap，只会返回这个bitmap的尺寸。这个属性的目的是，如果你只想知道一个bitmap的尺寸，但又不想将其加载到内存时。这是一个非常有用的属性。
- `public int inSampleSize` //图片缩放的倍数，这个值是一个int，当它小于1的时候，将会被当做1处理，如果大于1，那么就会按照比例 $(1 / \text{inSampleSize})$ 缩小bitmap的宽和高、降

低分辨率，大于1时这个值将会被处置为2的倍数。例如，`width=100, height=100, inSampleSize=2`，那么就会将 `bitmap` 处理为，`width=50, height=50`，宽高降为1 / 2，像素数降为1 / 4。

- `public int outwidth` //获取图片的宽度值
- `public int outHeight` //获取图片的高度值，表示这个 `Bitmap` 的宽和高，一般和 `inJustDecodeBounds` 一起使用来获得 `Bitmap` 的宽高，但是不加载到内存。
- `public int inDensity` //用于位图的像素压缩比
- `public int inTargetDensity` //用于目标位图的像素压缩比（要生成的位图）
- `public byte[] inTempStorage` //创建临时文件，将图片存储
- `public boolean inScaled` //设置为 `true` 时进行图片压缩，从 `inDensity` 到 `inTargetDensity`
- `public boolean inDither` //如果为 `true` ,解码器尝试抖动解码
- `public Bitmap.Config inPreferredConfig` //设置解码器，这个值是设置色彩模式，默认值是 `ARGB_8888`，在这个模式下，一个像素点占用4bytes空间，一般对透明度不做要求的话，一般采用 `RGB_565` 模式，这个模式下一个像素点占用2bytes。
- `public String outMimeType` //设置解码图像
- `public boolean inPurgeable` //当存储 `Pixel` 的内存空间在系统内存不足时是否可以被回收
- `public boolean inInputShareable` // `inPurgeable` 为 `true` 情况下才生效，是否可以共享一个 `InputStream`
- `public boolean inPreferQualityOverSpeed` //为 `true` 则优先保证 `Bitmap` 质量其次是解码速度
- `public boolean inMutable` //配置 `Bitmap` 是否可以更改，比如：在 `Bitmap` 上隔几个像素加一条线段
- `public int inScreenDensity` //当前屏幕的像素密度

工厂方法

- `public static Bitmap decodeFile(String pathName, Options opts)` //从文件读取图片
- `public static Bitmap decodeFile(String pathName)`
- `public static Bitmap decodeStream(InputStream is)` //从输入流读取图片
- `public static Bitmap decodeStream(InputStream is, Rect outPadding, Options opts)`
- `public static Bitmap decodeResource(Resources res, int id)` //从资源文件读取图片
- `public static Bitmap decodeResource(Resources res, int id, Options opts)`
- `public static Bitmap decodeByteArray(byte[] data, int offset, int length)` //从数组读取图片
- `public static Bitmap decodeByteArray(byte[] data, int offset, int length, Options opts)`
- `public static Bitmap decodeFileDescriptor(FileDescriptor fd)` //从文件读取文件 与 `decodeFile` 不同的是这个直接调用JNI函数进行读取 效率比较高
- `public static Bitmap decodeFileDescriptor(FileDescriptor fd, Rect outPadding, Options opts)`

单个像素的字节大小

单个像素的字节大小由 `Bitmap` 的一个可配置参数 `Config` 来决定。

`Bitmap` 中，存在一个枚举类 `Config`，定义了 `Android` 中支持的 `Bitmap` 配置：

Config	占用字节大小 (byte)	说明
ALPHA_8 (1)	1	单透明通道
RGB_565 (3)	2	简易RGB色调
ARGB_4444 (4)	4	已废弃
ARGB_8888 (5)	4	24位真彩色
RGBA_F16 (6)	8	Android 8.0 新增 (更丰富的色彩表现HDR)
HARDWARE (7)	Special	Android 8.0 新增 (Bitmap直接存储在graphic memory) 注1 https://blog.csdn.net/Simon_Crystin

Bitmap加载方式

Bitmap 的加载方式有 Resource 资源加载、本地 (SDcard) 加载、网络加载等加载方式。

1. 从本地 (SDcard) 文件读取

方式一

```

1  /**
2   * 获取缩放后的本地图片
3   *
4   * @param filePath 文件路径
5   * @param width 宽
6   * @param height 高
7   * @return
8   */
9  public static Bitmap readBitmapFromFile(String filePath, int width, int
height) {
10     BitmapFactory.Options options = new BitmapFactory.Options();
11     options.inJustDecodeBounds = true;
12     BitmapFactory.decodeFile(filePath, options);
13     float srcwidth = options.outWidth;
14     float srcHeight = options.outHeight;
15     int inSampleSize = 1;
16
17     if (srcHeight > height || srcwidth > width) {
18         if (srcwidth > srcHeight) {
19             inSampleSize = Math.round(srcHeight / height);
20         } else {
21             inSampleSize = Math.round(srcwidth / width);
22         }
23     }
24
25     options.inJustDecodeBounds = false;
26     options.inSampleSize = inSampleSize;
27

```

```
28     return BitmapFactory.decodeFile(filePath, options);
29 }
```

方式二 (效率高于方式一)

```
1  /**
2   * 获取缩放后的本地图片
3   *
4   * @param filePath 文件路径
5   * @param width 宽
6   * @param height 高
7   * @return
8   */
9  public static Bitmap readBitmapFromFileDescriptor(String filePath, int
width, int height) {
10     try {
11         FileInputStream fis = new FileInputStream(filePath);
12         BitmapFactory.Options options = new BitmapFactory.Options();
13         options.inJustDecodeBounds = true;
14         BitmapFactory.decodeFileDescriptor(fis.getFD(), null, options);
15         float srcwidth = options.outwidth;
16         float srcHeight = options.outHeight;
17         int inSampleSize = 1;
18
19         if (srcHeight > height || srcwidth > width) {
20             if (srcwidth > srcHeight) {
21                 inSampleSize = Math.round(srcHeight / height);
22             } else {
23                 inSampleSize = Math.round(srcwidth / width);
24             }
25         }
26
27         options.inJustDecodeBounds = false;
28         options.inSampleSize = inSampleSize;
29
30         return BitmapFactory.decodeFileDescriptor(fis.getFD(), null,
options);
31     } catch (Exception ex) {
32     }
33     return null;
34 }
```

2. 从输入流中读取文件 (网络加载)

```
1  /**
2   * 获取缩放后的本地图片
3   *
4   * @param ins 输入流
5   * @param width 宽
6   * @param height 高
7   * @return
8   */
9  public static Bitmap readBitmapFromInputStream(InputStream ins, int width,
int height) {
10     BitmapFactory.Options options = new BitmapFactory.Options();
11     options.inJustDecodeBounds = true;
```

```

12     BitmapFactory.decodeStream(ins, null, options);
13     float srcwidth = options.outWidth;
14     float srcHeight = options.outHeight;
15     int inSampleSize = 1;
16
17     if (srcHeight > height || srcwidth > width) {
18         if (srcwidth > srcHeight) {
19             inSampleSize = Math.round(srcHeight / height);
20         } else {
21             inSampleSize = Math.round(srcwidth / width);
22         }
23     }
24
25     options.inJustDecodeBounds = false;
26     options.inSampleSize = inSampleSize;
27
28     return BitmapFactory.decodeStream(ins, null, options);
29 }

```

3.Resource资源加载

Res资源加载方式:

```

1  public static Bitmap readBitmapFromResource(Resources resources, int
resourcesId, int width, int height) {
2      BitmapFactory.Options options = new BitmapFactory.Options();
3      options.inJustDecodeBounds = true;
4      BitmapFactory.decodeResource(resources, resourcesId, options);
5      float srcwidth = options.outWidth;
6      float srcHeight = options.outHeight;
7      int inSampleSize = 1;
8
9      if (srcHeight > height || srcwidth > width) {
10         if (srcwidth > srcHeight) {
11             inSampleSize = Math.round(srcHeight / height);
12         } else {
13             inSampleSize = Math.round(srcwidth / width);
14         }
15     }
16
17     options.inJustDecodeBounds = false;
18     options.inSampleSize = inSampleSize;
19
20     return BitmapFactory.decodeResource(resources, resourcesId, options);
21 }

```

此种方式相当的耗费内存 建议采用 `decodeStream` 代替 `decodeResource` 可以如下形式:

```

1  public static Bitmap readBitmapFromResource(Resources resources, int
resourcesId, int width, int height) {
2      InputStream ins = resources.openRawResource(resourcesId);
3      BitmapFactory.Options options = new BitmapFactory.Options();
4      options.inJustDecodeBounds = true;
5      BitmapFactory.decodeStream(ins, null, options);
6      float srcwidth = options.outWidth;
7      float srcHeight = options.outHeight;
8      int inSampleSize = 1;

```

```

9
10     if (srcHeight > height || srcwidth > width) {
11         if (srcwidth > srcHeight) {
12             inSampleSize = Math.round(srcHeight / height);
13         } else {
14             inSampleSize = Math.round(srcwidth / width);
15         }
16     }
17
18     options.inJustDecodeBounds = false;
19     options.inSampleSize = inSampleSize;
20
21     return BitmapFactory.decodeStream(ins, null, options);
22 }

```

`BitmapFactory.decodeResource` 加载的图片可能会经过缩放，该缩放目前是放在 java 层做的，效率比较低，而且需要消耗 java 层的内存。因此，如果大量使用该接口加载图片，容易导致OOM错误。
`BitmapFactory.decodeStream` 不会对所加载的图片进行缩放，相比之下占用内存少，效率更高。
这两个接口各有用处，如果对性能要求较高，则应该使用 `decodeStream`；如果对性能要求不高，且需要 Android 自带的图片自适应缩放功能，则可以使用 `decodeResource`。

4.Assets资源加载方式:

```

1  /**
2   * 获取缩放后的本地图片
3   *
4   * @param filePath 文件路径,即文件名称
5   * @return
6   */
7  public static Bitmap readBitmapFromAssetsFile(Context context, String
filePath) {
8      Bitmap image = null;
9      AssetManager am = context.getResources().getAssets();
10     try {
11         InputStream is = am.open(filePath);
12         image = BitmapFactory.decodeStream(is);
13         is.close();
14     } catch (IOException e) {
15         e.printStackTrace();
16     }
17     return image;
18 }

```

5.从二进制数据读取图片

```

1  public static Bitmap readBitmapFromByteArray(byte[] data, int width, int
height) {
2      BitmapFactory.Options options = new BitmapFactory.Options();
3      options.inJustDecodeBounds = true;
4      BitmapFactory.decodeByteArray(data, 0, data.length, options);
5      float srcwidth = options.outwidth;
6      float srcHeight = options.outHeight;
7      int inSampleSize = 1;
8
9      if (srcHeight > height || srcwidth > width) {
10         if (srcwidth > srcHeight) {

```

```

11         inSampleSize = Math.round(srcHeight / height);
12     } else {
13         inSampleSize = Math.round(srcWidth / width);
14     }
15 }
16
17 options.inJustDecodeBounds = false;
18 options.inSampleSize = inSampleSize;
19
20 return BitmapFactory.decodeByteArray(data, 0, data.length, options);
21 }

```

Bitmap | Drawable | InputStream | Byte[] 之间进行转换

1. Drawable 转化成 Bitmap

```

1 public static Bitmap drawableToBitmap(Drawable drawable) {
2     Bitmap bitmap = Bitmap.createBitmap(drawable.getIntrinsicWidth(),
3     drawable.getIntrinsicHeight(),
4     drawable.getOpacity() != PixelFormat.OPAQUE ?
5     Bitmap.Config.ARGB_8888 : Bitmap.Config.RGB_565);
6     Canvas canvas = new Canvas(bitmap);
7     drawable.setBounds(0, 0, drawable.getIntrinsicWidth(),
8     drawable.getIntrinsicHeight());
9     drawable.draw(canvas);
10    return bitmap;
11 }

```

drawable 的获取方式: `Drawable drawable = getResources().getDrawable(R.drawable.ic_launcher);`

2. Bitmap 转换成 Drawable

```

1 public static Drawable bitmapToDrawable(Resources resources, Bitmap bm) {
2     Drawable drawable = new BitmapDrawable(resources, bm);
3     return drawable;
4 }

```

3. Bitmap 转换成 byte[]

```

1 public byte[] bitmap2Bytes(Bitmap bm) {
2     ByteArrayOutputStream baos = new ByteArrayOutputStream();
3     bm.compress(Bitmap.CompressFormat.PNG, 100, baos);
4     return baos.toByteArray();
5 }

```

4. byte[] 转换成 Bitmap

```

1 Bitmap bitmap = BitmapFactory.decodeByteArray(byte, 0, b.length);

```

5.InputStream转换成Bitmap

```
1 InputStream is = getResources().openRawResource(id);
2 Bitmap bitmap = BitmapFactory.decodeStream(is);
```

6.InputStream转换成byte[]

```
1 InputStream is = getResources().openRawResource(id); //也可以通过其他方式接收一个
InputStream对象
2 ByteArrayOutputStream baos = new ByteArrayOutputStream();
3 byte[] b = new byte[1024*2];
4 int len = 0;
5 while ((len = is.read(b, 0, b.length)) != -1)
6 {
7     baos.write(b, 0, len);
8     baos.flush();
9 }
10 byte[] bytes = baos.toByteArray();
```

Bitmap常用操作

1.将Bitmap保存为本地文件:

```
1 public static void writeBitmapToFile(String filePath, Bitmap b, int quality)
2 {
3     try {
4         File desFile = new File(filePath);
5         FileOutputStream fos = new FileOutputStream(desFile);
6         BufferedOutputStream bos = new BufferedOutputStream(fos);
7         b.compress(Bitmap.CompressFormat.JPEG, quality, bos);
8         bos.flush();
9         bos.close();
10    } catch (IOException e) {
11        e.printStackTrace();
12    }
13 }
```

2.图片压缩:

```
1 private static Bitmap compressImage(Bitmap image) {
2     if (image == null) {
3         return null;
4     }
5     ByteArrayOutputStream baos = null;
6     try {
7         baos = new ByteArrayOutputStream();
8         image.compress(Bitmap.CompressFormat.JPEG, 50, baos);
9         byte[] bytes = baos.toByteArray();
10        ByteArrayInputStream isBm = new ByteArrayInputStream(bytes);
11        Bitmap bitmap = BitmapFactory.decodeStream(isBm);
12        return bitmap;
13    } catch (OutOfMemoryError e) {
14    } finally {
15        try {
16            if (baos != null) {
```



```

17         baos.close();
18     }
19     } catch (IOException e) {
20     }
21 }
22 return null;
23 }

```

3.图片缩放:

```

1  /**
2   * 根据scale生成一张图片
3   *
4   * @param bitmap
5   * @param scale 等比缩放值
6   * @return
7   */
8  public static Bitmap bitmapScale(Bitmap bitmap, float scale) {
9      Matrix matrix = new Matrix();
10     matrix.postScale(scale, scale); // 长和宽放大缩小的比例
11     Bitmap resizeBmp = Bitmap.createBitmap(bitmap, 0, 0, bitmap.getWidth(),
12         bitmap.getHeight(),
13         matrix, true);
14     return resizeBmp;
15 }

```

4.获取图片旋转角度:

```

1  /**
2   * 读取照片exif信息中的旋转角度
3   *
4   * @param path 照片路径
5   * @return角度
6   */
7  private static int readPictureDegree(String path) {
8      if (TextUtils.isEmpty(path)) {
9          return 0;
10     }
11     int degree = 0;
12     try {
13         ExifInterface exifInterface = new ExifInterface(path);
14         int orientation =
15             exifInterface.getAttributeInt(ExifInterface.TAG_ORIENTATION,
16                 ExifInterface.ORIENTATION_NORMAL);
17         switch (orientation) {
18             case ExifInterface.ORIENTATION_ROTATE_90:
19                 degree = 90;
20                 break;
21             case ExifInterface.ORIENTATION_ROTATE_180:
22                 degree = 180;
23                 break;
24             case ExifInterface.ORIENTATION_ROTATE_270:
25                 degree = 270;
26                 break;
27         }
28     } catch (Exception e) {
29     }
30 }

```

```
29     return degree;
30 }
```

5.设置图片旋转角度

```
1 private static Bitmap rotateBitmap(Bitmap b, float rotateDegree) {
2     if (b == null) {
3         return null;
4     }
5     Matrix matrix = new Matrix();
6     matrix.postRotate(rotateDegree);
7     Bitmap rotaBitmap = Bitmap.createBitmap(b, 0, 0, b.getWidth(),
8     b.getHeight(), matrix, true);
9     return rotaBitmap;
10 }
```

6.通过图片id获得Bitmap:

```
1 Bitmap bitmap=BitmapFactory.decodeResource(getResources(),
2 R.drawable.ic_launcher);
```

7.通过 assest 获取 获得Drawable bitmap:

```
1 InputStream in = this.getAssets().open("ic_launcher");
2 Drawable da = Drawable.createFromStream(in, null);
3 Bitmap mm = BitmapFactory.decodeStream(in);
```

8.通过 sdcard 获得 bitmap

```
1 Bitmap bit = BitmapFactory.decodeFile("/sdcard/android.jpg");
```

9.view转Bitmap

```
1 public static Bitmap convertViewToBitmap(View view, int bitmapwidth, int
2 bitmapheight){
3     Bitmap bitmap = Bitmap.createBitmap(bitmapwidth, bitmapheight,
4     Bitmap.Config.ARGB_8888);
5     view.draw(new Canvas(bitmap));
6     return bitmap;
7 }
```

10.将控件转换为bitmap

```
1 public static Bitmap convertViewToBitMap(View view){
2     // 打开图像缓存
3     view.setDrawingCacheEnabled(true);
4     // 必须调用measure和layout方法才能成功保存可视组件的截图到png图像文件
5     // 测量View大小
6     view.measure(MeasureSpec.makeMeasureSpec(0, MeasureSpec.UNSPECIFIED),
7     MeasureSpec.makeMeasureSpec(0, MeasureSpec.UNSPECIFIED));
8     // 发送位置和尺寸到View及其所有的子View
9     view.layout(0, 0, view.getMeasuredWidth(), view.getMeasuredHeight());
10    // 获得可视组件的截图
11    Bitmap bitmap = view.getDrawingCache();
```

```

12     return bitmap;
13 }
14
15 public static Bitmap getBitmapFromView(View view){
16     Bitmap returnedBitmap = Bitmap.createBitmap(view.getWidth(),
17     view.getHeight(), Bitmap.Config.ARGB_8888);
18     Canvas canvas = new Canvas(returnedBitmap);
19     Drawable bgDrawable = view.getBackground();
20     if (bgDrawable != null)
21         bgDrawable.draw(canvas);
22     else
23         canvas.drawColor(Color.WHITE);
24     view.draw(canvas);
25     return returnedBitmap;
26 }

```

11.放大缩小图片

```

1 public static Bitmap zoomBitmap(Bitmap bitmap,int w,int h){
2     int width = bitmap.getWidth();
3     int height = bitmap.getHeight();
4     Matrix matrix = new Matrix();
5     float scalewidht = ((float)w / width);
6     float scaleHeight = ((float)h / height);
7     matrix.postScale(scalewidht, scaleHeight);
8     Bitmap newbmp = Bitmap.createBitmap(bitmap, 0, 0, width, height, matrix,
9     true);
10    return newbmp;
11 }

```

12.获得圆角图片的方法

```

1 public static Bitmap getRoundedCornerBitmap(Bitmap bitmap,float roundPx){
2
3     Bitmap output = Bitmap.createBitmap(bitmap.getWidth(), bitmap
4     .getHeight(), Config.ARGB_8888);
5     Canvas canvas = new Canvas(output);
6
7     final int color = 0xff424242;
8     final Paint paint = new Paint();
9     final Rect rect = new Rect(0, 0, bitmap.getWidth(), bitmap.getHeight());
10    final RectF rectF = new RectF(rect);
11
12    paint.setAntiAlias(true);
13    canvas.drawARGB(0, 0, 0, 0);
14    paint.setColor(color);
15    canvas.drawRoundRect(rectF, roundPx, roundPx, paint);
16
17    paint.setXfermode(new PorterDuffXfermode(Mode.SRC_IN));
18    canvas.drawBitmap(bitmap, rect, rect, paint);
19
20    return output;
21 }

```

13.对 bitmap 进行裁剪

```
1 public Bitmap bitmapClip(Context context , int id , int x , int y){
2     Bitmap map = BitmapFactory.decodeResource(context.getResources(), id);
3     map = Bitmap.createBitmap(map, x, y, 120, 120);
4     return map;
5 }
```

Bitmap内存模型

2.3-	3.0-4.4	5.0-7.1	8.0
Bitmap对象	java Heap	java Heap	java Heap
像素数据	Native Heap	java Heap	Native Heap
迁移原因	-	解决Native Bitmap内存泄露	共享整个系统的内存减少OOM

Bitmap的内存回收

1. 在Android2.3.3之前推荐使用Bitmap.recycle()方法进行Bitmap的内存回收。

备注：只有当确定这个Bitmap不被引用的时候才能调用此方法，否则会有“Canvas: trying to use a recycled bitmap”这个错误。

2. Android3.0之后

Android3.0之后，并没有强调Bitmap.recycle(); 而是强调Bitmap的复用

- Save a bitmap for later use

使用LruCache对Bitmap进行缓存**，当再次使用到这个Bitmap的时候直接获取，而不用重走编码流程。

- Use an existing bitmap

Android3.0(API 11之后)引入了BitmapFactory.Options.inBitmap字段，设置此字段之后解码方法会尝试复用一张存在的Bitmap。这意味着Bitmap的内存被复用，避免了内存的回收及申请过程，显然性能表现更佳。不过，使用这个字段有几点限制：

- 声明可被复用的Bitmap必须设置inMutable为true;
- Android4.4(API 19)之前只有格式为jpg、png，同等宽高（要求苛刻），inSampleSize为1的Bitmap才可以复用;
- Android4.4(API 19)之前被复用的Bitmap的inPreferredConfig会覆盖待分配内存的Bitmap设置的inPreferredConfig;
- Android4.4(API 19)之后被复用的Bitmap的内存必须大于需要申请内存的Bitmap的内存;
- Android4.4(API 19)之前待加载Bitmap的Options.inSampleSize必须明确指定为1

获取Bitmap的大小

1. getByteCount()

getByteCount()方法是在API12加入的，代表存储Bitmap的像素需要的最少内存。API19开始getAllocationByteCount()方法代替了getByteCount()。

2. getAllocationByteCount()

API19之后, Bitmap加了一个Api: getAllocationByteCount(); 代表在内存中为Bitmap分配的内存大小。

```
1 public final int getAllocationByteCount() {
2     if (mBuffer == null) {
3         //mBuffer代表存储Bitmap像素数据的字节数组。
4         return getByteCount();
5     }
6     return mBuffer.length;
7 }
```

3. getByteCount()与getAllocationByteCount()的区别

- 一般情况下两者是相等的
- 通过复用Bitmap来解码图片, 如果被复用的Bitmap的内存比待分配内存的Bitmap大,那么 getByteCount()表示新解码图片占用内存的大小 (并非实际内存大小,实际大小是复用的那个Bitmap的大小), getAllocationByteCount()表示被复用Bitmap真实占用的内存大小 (即mBuffer的长度)

Bitmap占用内存大小计算

Bitmap作为位图, 需要读入一张图片每一个像素点的数据, 其主要占用内存的地方也正是这些像素数据。对于像素数据总大小, 我们可以猜想为: 像素总数量 × 每个像素的字节大小, 而像素总数量在矩形屏幕表现下, 应该是: 横向像素数量 × 纵向像素数量, 结合得到:

Bitmap内存占用 ≈ 像素数据总大小 = 横向像素数量 × 纵向像素数量 × 每个像素的字节大小

但真是如此吗?

我们来看下源码, Bitmap的decode过程实际上是在native层完成的, 为此, 需要从 [BitmapFactory.cpp#nativeDecodeXXX](#)方法开始跟踪, 最终在doDecode方法里面

```
1 static jobject doDecode(JNIEnv* env, std::unique_ptr<SkStreamRewindable>
   stream,
2                               jobject padding, jobject options) {
3     // Set default values for the options parameters.
4     int sampleSize = 1;
5     bool onlyDecodeSize = false;
6     skColorType prefColorType = kN32_SkColorType;
7     bool isHardware = false;
8     bool isMutable = false;
9     float scale = 1.0f;
10    bool requireUnpremultiplied = false;
11    jobject javaBitmap = NULL;
12    sk_sp<SkColorSpace> prefColorSpace = nullptr;
13
14    // Update with options supplied by the client.
15    if (options != NULL) {
16        sampleSize = env->GetIntField(options, gOptions_sampleSizeFieldID);
17        // Correct a non-positive sampleSize. sampleSize defaults to zero
   within the
18        // options object, which is strange.
19        if (sampleSize <= 0) {
```

```

20         sampleSize = 1;
21     }
22
23     if (env->GetBooleanField(options, gOptions_justBoundsFieldID)) {
24         onlyDecodeSize = true;
25     }
26
27     // initialize these, in case we fail later on
28     env->SetIntField(options, gOptions_widthFieldID, -1);
29     env->SetIntField(options, gOptions_heightFieldID, -1);
30     env->SetObjectField(options, gOptions_mimeFieldID, 0);
31     env->SetObjectField(options, gOptions_outConfigFieldID, 0);
32     env->SetObjectField(options, gOptions_outColorSpaceFieldID, 0);
33
34     jobject jconfig = env->GetObjectField(options,
gOptions_configFieldID);
35     prefColorType = GraphicsJNI::getNativeBitmapColorType(env,
jconfig);
36     jobject jcolorSpace = env->GetObjectField(options,
gOptions_colorSpaceFieldID);
37     prefColorSpace = GraphicsJNI::getNativeColorSpace(env,
jcolorSpace);
38     isHardware = GraphicsJNI::isHardwareConfig(env, jconfig);
39     isMutable = env->GetBooleanField(options, gOptions_mutableFieldID);
40     requireUnpremultiplied = !env->GetBooleanField(options,
gOptions_premultipliedFieldID);
41     javaBitmap = env->GetObjectField(options, gOptions_bitmapFieldID);
42
43     if (env->GetBooleanField(options, gOptions_scaledFieldID)) {
44         const int density = env->GetIntField(options,
gOptions_densityFieldID); //TODO: 1
45         const int targetDensity = env->GetIntField(options,
gOptions_targetDensityFieldID);
46         const int screenDensity = env->GetIntField(options,
gOptions_screenDensityFieldID);
47         if (density != 0 && targetDensity != 0 && density !=
screenDensity) {
48             scale = (float) targetDensity / density;
49         }
50     }
51 }
52
53 if (isMutable && isHardware) {
54     doThrowIAE(env, "Bitmaps with Config.HARWARE are always
immutable");
55     return nullObjectReturn("Cannot create mutable hardware bitmap");
56 }
57
58 // Create the codec.
59 NinePatchPeeker peeker;
60 std::unique_ptr<SkAndroidCodec> codec;
61 {
62     SkCodec::Result result;
63     std::unique_ptr<SkCodec> c =
SkCodec::MakeFromStream(std::move(stream), &result,
64                                     &peeker);
65     if (!c) {
66         SkString msg;

```

```

67         msg.printf("Failed to create image decoder with message '%s'",
68                     SkCodec::ResultToString(result));
69         return nullobjectReturn(msg.c_str());
70     }
71
72     codec = SkAndroidCodec::MakeFromCodec(std::move(c));
73     if (!codec) {
74         return nullobjectReturn("SkAndroidCodec::MakeFromCodec returned
75 null");
76     }
77
78     // Do not allow ninepatch decodes to 565. In the past, decodes to 565
79     // would dither, and we do not want to pre-dither ninepatches, since we
80     // know that they will be stretched. We no longer dither 565 decodes,
81     // but we continue to prevent ninepatches from decoding to 565, in
82     order
83     // to maintain the old behavior.
84     if (peeker.mPatch && kRGB_565_SkColorType == prefColorType) {
85         prefColorType = kN32_SkColorType;
86     }
87
88     // Determine the output size.
89     SkISize size = codec->getSampledDimensions(sampleSize);
90     //TODO: 2
91     int scaledWidth = size.width();
92     int scaledHeight = size.height();
93     bool willScale = false;
94
95     // Apply a fine scaling step if necessary.
96     if (needsFineScale(codec->getInfo().dimensions(), size, sampleSize)) {
97         willScale = true;
98         scaledWidth = codec->getInfo().width() / sampleSize;
99         scaledHeight = codec->getInfo().height() / sampleSize;
100     }
101
102     // Set the decode colorType
103     SkColorType decodeColorType = codec-
104     >computeOutputColorType(prefColorType);
105     sk_sp<SkColorSpace> decodeColorSpace = codec->computeOutputColorSpace(
106         decodeColorType, prefColorSpace);
107
108     // Set the options and return if the client only wants the size.
109     if (options != NULL) {
110         jstring mimeType = encodedFormatToString(
111             env, (SkEncodedImageFormat)codec->getEncodedFormat());
112         if (env->ExceptionCheck()) {
113             return nullobjectReturn("OOM in encodedFormatToString()");
114         }
115         env->SetIntField(options, gOptions_widthFieldID, scaledWidth);
116         env->SetIntField(options, gOptions_heightFieldID, scaledHeight);
117         env->SetObjectField(options, gOptions_mimeFieldID, mimeType);
118
119         jint configID =
120         GraphicsJNI::colorTypeToLegacyBitmapConfig(decodeColorType);
121         if (isHardware) {
122             configID = GraphicsJNI::kHardware_LegacyBitmapConfig;
123         }

```

```

121     jobject config = env->CallStaticObjectMethod(gBitmapConfig_class,
122         gBitmapConfig_nativeToConfigMethodID, configID);
123     env->SetObjectField(options, gOptions_outConfigFieldID, config);
124
125     env->SetObjectField(options, gOptions_outColorSpaceFieldID,
126         GraphicsJNI::getColorSpace(env, decodeColorSpace,
decodeColorType));
127
128     if (onlyDecodeSize) {
129         return nullptr;
130     }
131 }
132
133 // Scale is necessary due to density differences.
134 if (scale != 1.0f) {
135     willScale = true;
136     scaledWidth = static_cast<int>(scaledWidth * scale + 0.5f);
137     scaledHeight = static_cast<int>(scaledHeight * scale + 0.5f);
138 }
139
140 android::Bitmap* reuseBitmap = nullptr;
141 unsigned int existingBufferSize = 0;
142 if (javaBitmap != NULL) {
143     reuseBitmap = &bitmap::toBitmap(env, javaBitmap);
144     if (reuseBitmap->isImmutable()) {
145         ALOGW("Unable to reuse an immutable bitmap as an image decoder
target.");
146         javaBitmap = NULL;
147         reuseBitmap = nullptr;
148     } else {
149         existingBufferSize = bitmap::getBitmapAllocationByteCount(env,
javaBitmap);
150     }
151 }
152
153 HeapAllocator defaultAllocator;
154 RecyclingPixelAllocator recyclingAllocator(reuseBitmap,
existingBufferSize);
155 ScaleCheckingAllocator scaleCheckingAllocator(scale,
existingBufferSize);
156 SkBitmap::HeapAllocator heapAllocator;
157 SkBitmap::Allocator* decodeAllocator;
158 if (javaBitmap != nullptr && willScale) {
159     // This will allocate pixels using a HeapAllocator, since there
will be an extra
160     // scaling step that copies these pixels into Java memory. This
allocator
161     // also checks that the recycled javaBitmap is large enough.
162     decodeAllocator = &scaleCheckingAllocator;
163 } else if (javaBitmap != nullptr) {
164     decodeAllocator = &recyclingAllocator;
165 } else if (willScale || isHardware) {
166     // This will allocate pixels using a HeapAllocator,
167     // for scale case: there will be an extra scaling step.
168     // for hardware case: there will be extra swizzling & upload to
gralloc step.
169     decodeAllocator = &heapAllocator;
170 } else {

```



```

171         decodeAllocator = &defaultAllocator;
172     }
173
174     SkAlphaType alphaType = codec-
175     >computeOutputAlphaType(requireUnpremultiplied);
176
177     const SkImageInfo decodeInfo = SkImageInfo::Make(size.width(),
178     size.height(),
179     decodeColorType, alphaType, decodeColorSpace);
180
181     // For wide gamut images, we will leave the color space on the
182     SkBitmap. Otherwise,
183     // use the default.
184     SkImageInfo bitmapInfo = decodeInfo;
185     if (decodeInfo.colorSpace() && decodeInfo.colorSpace()->isSRGB()) {
186         bitmapInfo =
187         bitmapInfo.makeColorSpace(GraphicsJNI::colorSpaceForType(decodeColorType));
188     }
189
190     if (decodeColorType == kGray_8_SkColorType) {
191         // The legacy implementation of BitmapFactory used kAlpha8 for
192         // grayscale images (before kGray8 existed). While the codec
193         // recognizes kGray8, we need to decode into a kAlpha8 bitmap
194         // in order to avoid a behavior change.
195         bitmapInfo =
196         bitmapInfo.makeColorType(kAlpha_8_SkColorType).makeAlphaType(kPremul_SkAlp
197         haType);
198     }
199     SkBitmap decodingBitmap;
200     if (!decodingBitmap.setInfo(bitmapInfo) ||
201         !decodingBitmap.tryAllocPixels(decodeAllocator)) {
202         // SkAndroidCodec should recommend a valid SkImageInfo, so
203         setInfo()
204         // should only fail if the calculated value for rowBytes is
205         too
206         // large.
207         // tryAllocPixels() can fail due to OOM on the Java heap, OOM on
208         the
209         // native heap, or the recycled javaBitmap being too small to
210         reuse.
211         return nullptr;
212     }
213
214     // Use SkAndroidCodec to perform the decode.
215     SkAndroidCodec::AndroidOptions codecOptions;
216     codecOptions.fZeroInitialized = decodeAllocator == &defaultAllocator ?
217     SkCodec::kYes_ZeroInitialized : SkCodec::kNo_ZeroInitialized;
218     codecOptions.fSampleSize = sampleSize;
219     SkCodec::Result result = codec->getAndroidPixels(decodeInfo,
220     decodingBitmap.getPixels(),
221     decodingBitmap.rowBytes(), &codecOptions);
222     switch (result) {
223         case SkCodec::kSuccess:
224         case SkCodec::kIncompleteInput:
225             break;
226         default:
227             return nullobjectReturn("codec->getAndroidPixels() failed.");
228     }

```

```

218     }
219
220     // This is weird so let me explain: we could use the scale parameter
221     // directly, but for historical reasons this is how the corresponding
222     // Dalvik code has always behaved. We simply recreate the behavior
223     here.
224     // The result is slightly different from simply using scale because of
225     // the 0.5f rounding bias applied when computing the target image size
226     const float scaleX = scaledWidth / float(decodingBitmap.width());
227     const float scaleY = scaledHeight / float(decodingBitmap.height());
228
229     jbyteArray ninePatchChunk = NULL;
230     if (peeker.mPatch != NULL) {
231         if (willScale) {
232             peeker.scale(scaleX, scaleY, scaledWidth, scaledHeight);
233         }
234
235         size_t ninePatchArraySize = peeker.mPatch->serializedSize();
236         ninePatchChunk = env->NewByteArray(ninePatchArraySize);
237         if (ninePatchChunk == NULL) {
238             return nullObjectReturn("ninePatchChunk == null");
239         }
240
241         jbyte* array = (jbyte*) env->GetPrimitiveArrayCritical(ninePatchChunk, NULL);
242         if (array == NULL) {
243             return nullObjectReturn("primitive array == null");
244         }
245
246         memcpy(array, peeker.mPatch, peeker.mPatchSize);
247         env->ReleasePrimitiveArrayCritical(ninePatchChunk, array, 0);
248     }
249
250     jobject ninePatchInsets = NULL;
251     if (peeker.mHasInsets) {
252         ninePatchInsets = peeker.createNinePatchInsets(env, scale);
253         if (ninePatchInsets == NULL) {
254             return nullObjectReturn("nine patch insets == null");
255         }
256         if (javaBitmap != NULL) {
257             env->SetObjectField(javaBitmap, gBitmap_ninePatchInsetsFieldID,
258                                 ninePatchInsets);
259         }
260     }
261
262     SkBitmap outputBitmap;
263     if (willScale) {
264         // Set the allocator for the outputBitmap.
265         SkBitmap::Allocator* outputAllocator;
266         if (javaBitmap != nullptr) {
267             outputAllocator = &recyclingAllocator;
268         } else {
269             outputAllocator = &defaultAllocator;
270         }
271
272         SkColorType scaledColorType = decodingBitmap.colorType();
273         // FIXME: If the alphaType is kUnpremul and the image has alpha,
274         the

```

```

272         // colors may not be correct, since Skia does not yet support
drawing
273         // to/from unpremultiplied bitmaps.
274         outputBitmap.setInfo(
275             bitmapInfo.makewh(scaledwidth,
scaledHeight).makeColorType(scaledColorType));
276         if (!outputBitmap.tryAllocPixels(outputAllocator)) {
277             // This should only fail on OOM. The recyclingAllocator should
have
278             // enough memory since we check this before decoding using the
279             // scaleCheckingAllocator.
280             return nullObjectReturn("allocation failed for scaled bitmap");
281         }
282
283         SkPaint paint;
284         // kSrc_Mode instructs us to overwrite the uninitialized pixels in
285         // outputBitmap. Otherwise we would blend by default, which is not
286         // what we want.
287         paint.setBlendMode(SkBlendMode::kSrc);
288         paint.setFilterQuality(kLow_SkFilterQuality); // bilinear filtering
289
290         SkCanvas canvas(outputBitmap, SkCanvas::ColorBehavior::kLegacy);
291         canvas.scale(scaleX, scaleY);
292         canvas.drawBitmap(decodingBitmap, 0.0f, 0.0f, &paint);
293     } else {
294         outputBitmap.swap(decodingBitmap);
295     }
296
297     if (padding) {
298         peeker.getPadding(env, padding);
299     }
300
301     // If we get here, the outputBitmap should have an installed pixelref.
302     if (outputBitmap.pixelRef() == NULL) {
303         return nullObjectReturn("Got null SkPixelRef");
304     }
305
306     if (!isMutable && javaBitmap == NULL) {
307         // promise we will never change our pixels (great for sharing and
pictures)
308         outputBitmap.setImmutable();
309     }
310
311     bool isPremultiplied = !requireUnpremultiplied;
312     if (javaBitmap != nullptr) {
313         bitmap::reinitBitmap(env, javaBitmap, outputBitmap.info(),
isPremultiplied);
314         outputBitmap.notifyPixelsChanged();
315         // If a java bitmap was passed in for reuse, pass it back
316         return javaBitmap;
317     }
318
319     int bitmapCreateFlags = 0x0;
320     if (isMutable) bitmapCreateFlags |=
android::bitmap::kBitmapCreateFlag_Mutable;
321     if (isPremultiplied) bitmapCreateFlags |=
android::bitmap::kBitmapCreateFlag_Premultiplied;
322

```

```

323     if (isHardware) {
324         sk_sp<Bitmap> hardwareBitmap =
Bitmap::allocateHardwareBitmap(outputBitmap);
325         if (!hardwareBitmap.get()) {
326             return nullobjectReturn("Failed to allocate a hardware
bitmap");
327         }
328         return bitmap::createBitmap(env, hardwareBitmap.release(),
bitmapCreateFlags,
329             ninePatchChunk, ninePatchInsets, -1);
330     }
331
332     // now create the java bitmap
333     return bitmap::createBitmap(env,
defaultAllocator.getStorageObjAndReset(),
334         bitmapCreateFlags, ninePatchChunk, ninePatchInsets, -1);
335 }
336
337 static jobject nativeDecodeStream(JNIEnv* env, jobject clazz, jobject is,
jbyteArray storage,
338     jobject padding, jobject options) {
339
340     jobject bitmap = NULL;
341     std::unique_ptr<SkStream> stream(CreateJavaInputStreamAdaptor(env, is,
storage));
342
343     if (stream.get()) {
344         std::unique_ptr<SkStreamRewindable> bufferedStream(
SkFrontBufferedStream::Make(std::move(stream),
345             SkCodec::MinBufferedBytesNeeded()));
346         SKASSERT(bufferedStream.get() != NULL);
347         bitmap = doDecode(env, std::move(bufferedStream), padding,
options);
348     }
349     return bitmap;
350 }

```

省略多余代码

```

1  if (env->GetBooleanField(options, goptions_scaledFieldID)) {
2      const int density = env->GetIntField(options, goptions_densityFieldID);
3      const int targetDensity = env->GetIntField(options,
gOptions_targetDensityFieldID);
4      const int screenDensity = env->GetIntField(options,
gOptions_screenDensityFieldID);
5      if (density != 0 && targetDensity != 0 && density != screenDensity) {
6          scale = (float) targetDensity / density;
7      }
8  }
9  ...
10 int scaledwidth = decoded->width();
11 int scaledHeight = decoded->height();
12
13 if (willScale && mode != SkImageDecoder::kDecodeBounds_Mode) {
14     scaledwidth = int(scaledwidth * scale + 0.5f);
15     scaledHeight = int(scaledHeight * scale + 0.5f);
16 }

```

```

17 ...
18 if (willScale) {
19     const float sx = scaledWidth / float(decoded->width());
20     const float sy = scaledHeight / float(decoded->height());
21     bitmap->setConfig(decoded->getConfig(), scaledWidth, scaledHeight);
22     bitmap->allocPixels(&javaAllocator, NULL);
23     bitmap->eraseColor(0);
24     SkPaint paint;
25     paint.setFilterBitmap(true);
26     SkCanvas canvas(*bitmap);
27     canvas.scale(sx, sy);
28     canvas.drawBitmap(*decoded, 0.0f, 0.0f, &paint);
29 }

```

从上述代码中，我们看到bitmap最终通过canvas绘制出来，而canvas在绘制之前，有一个scale的操作，scale的值由

```

1 scale = (float) targetDensity / density;

```

这一行代码决定，即缩放的倍率和targetDensity和density相关，而这两个参数都是从传入的options中获取到的

- inDensity: Bitmap位图自身的密度、分辨率
- inTargetDensity: Bitmap最终绘制的目标位置的分辨率
- inScreenDensity: 设备屏幕分辨率

其中inDensity和图片存放的资源文件的目录有关，同一张图片放置在不同目录下会有不同的值：

density	0.75	1	1.5	2	3	3.5	4
densityDpi	120	160	240	320	480	560	640
DpiFolder	ldpi	mdpi	hdpi	xhdpi	xxhdpi	xxxhdpi	xxxxhdpi

可以验证几个结论：

1. 图片放在drawable中，等同于放在drawable-mdpi中，原因为：drawable目录不具有屏幕密度特性，所以采用基准值，即mdpi
2. 图片放在某个特定drawable中，比如drawable-hdpi，如果设备的屏幕密度高于当前drawable目录所代表的密度，则图片会被放大，否则会被缩小

放大或缩小比例 = 设备屏幕密度 / drawable目录所代表的屏幕密度

因此，关于Bitmap占用内存大小的公式，从之前：

Bitmap内存占用 ≈ 像素数据总大小 = 横向像素数量 × 纵向像素数量 × 每个像素的字节大小

可以更细化为：

Bitmap内存占用 ≈ 像素数据总大小 = 图片宽 × 图片高 × (设备分辨率/资源目录分辨率)^2 × 每个像素的字节大小

