第二十期：Android 内存管理

eoe 特刊

Android 内存管理

eoe 开发者门户

优亿市场 Android应用发布与分享平台

目录

【序言】

刘峻辉

80 后的末班车，多次撰写 eoeAndroid 特刊。

大一学生会，班干部重要成员，大二开始创业，大三进入软件公司做开发，大四还在期待中。

勤奋好学，自学能力强，自我约束能力强，责任感很强。很喜欢听别人给我的建议，极易与人相处。爱最求新技术，喜欢挑战自己的极限。为人脚踏实地，不爱投机取巧。

个人 5 大经典录语：

1， 今天的努力是明天成功的基础。
2， 男人必须对自己狠点，不然别人对你狠点你就会很不适应。
3， 我对技术从来不敢说精通，因为技术上精通的定义我根本不知道。
4， 决定是今天的事情，结果是明天的事情，后悔是将来的事情。
5， 我很少赞成太过于绝对性的话语，唯独态度决定一切。

----------采访者：QUMIN

Android 内存管理基本介绍

1.1 Android 的内存管理简介

1.1.1 Android 和 Android 内存管理基本介绍

Android 内核是基于 Linux 2.6 内核的（目前最新开发版本是 2.6.31），它是一个增强内核版本，除了修改部分 Bug 外，它提供了用于支持 Android 平台的设备驱动，其核心驱动主要包括：

Android Binder ，基于 OpenBinder 框架的一个驱动，用于提供 Android 平台的进程间通讯（IPC，inter-process communication）。 源代码位于 drivers/staging/Android/binder.c

Android 电源管理（PM） ，一个基于标准 Linux 电源管理系统的轻量级的 Android 电源管理驱动，针对嵌入式设备做了很多优化。源代码位于 kernel/power/earlysuspend.c

kernel/power/consoleearlysuspend.c kernel/power/fbearlysuspend.c

kernel/power/wakelock.c kernel/power/userwakelock.c

低内存管理器（Low Memory Killer） ，相对于 Linux 标准 OOM（Out Of Memory）机制更加灵活，它可以根据需要杀死进程来释放需要的内存。 源代码位于 drivers/staging/Android/lowmemorykiller.c

匿名共享内存（ashmem） ，为进程间提供大块共享内存，同时为内核提供回收和管理这个内存的机制。 源代码位于 mm/ashmem.c

Android PMEM（Physical） ，PMEM 用于向用户空间提供连续的物理内存区域，DSP 和某些设备只能工作在连续的物理内存上。 源代码位于 drivers/misc/pmem.c

Android Logger ，一个轻量级的日志设备，用于抓取 Android 系统的各种日志。 源代码位于 drivers/staging/Android/logger.c

Android Alarm ，提供了一个定时器用于把设备从睡眠状态唤醒，同时它也提供了一个即使在设备睡眠时也会运行的时钟基准， 源代码位于 drivers/rtc/alarm.c

USB Gadget 驱动 ，一个基于标准 Linux USB gadget 驱动框架的设备驱动，Android 的 USB 驱动是基于 gaeget 框架的， 源代码位于 drivers/usb/gadget/

Android Ram Console ，为了提供调试功能，Android 允许将调试日志信息写入一个被称为 RAM Console 的设备里，它是一个基于 RAM 的 Buffer。 源代码位于 drivers/staging/Android/ram_console.c。

每个 Android 应用都运行在一个单独的进程在它自己的 Dalvik 的实例，负责所有的 Android 运行时的内存和进程管理的责任，停止和死亡过程的必要的管理资源。

Dalvik 和 Android 运行时间在一个 Linux 内核，处理低层次的硬件交互，包括驱动程序和内存管理，其 API 提供下层的服务，功能，和硬件的访问。

Dalvik 虚拟机是基于寄存器的虚拟机的优化，以确保设备可以有效地运行多个实例。它依赖于 Linux 内核线程和低级的内存管理。

1.1.2 Dalvik 虚拟机

Android 的关键要素之一是 Dalvik 虚拟机。 而不是使用传统的 Java 虚拟机（VM）中，如 Java ME（Java 移动版），Android 使用自己的定制设计，以确保有效地在单个设备上运行多个实例的虚拟机。

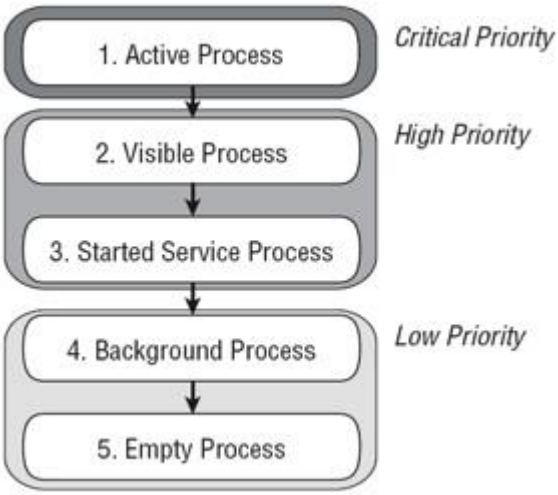
Dalvik 虚拟机，使用 Linux 内核处理低级别的功能，包括安全，线程，进程和内存管理。

所有的 Android 硬件和系统服务的访问的管理使用作为中间层的 Dalvik。 通过使用一个虚拟机，主机应用程序的执行，开发人员有一个抽象层，以确保他们从来没有担心一个特定的硬件实现。

Dalvik 虚拟机执行的 Dalvik 可执行文件格式，优化，以确保最小的内存足迹。 改造 Java 语言编译后的类的使用在 SDK 提供的工具创建.dex 可执行文件。

1.1.3 理解应用优先级和过程状态

下面的列表详细介绍每一个如图所示的应用程序状态，并解释如何取决于应用程序组件，包括它的状态是：



活动进程的活动（前台）进程是那些目前与用户交互的组件宿主应用程序。 这些都是 Android 是试图保持响应回收资源的过程。 通常有极少数的这些过程，他们将只作为最后的手段杀害。

活动进程包括：

- 活动中的一个“活跃”状态，也就是说，他们在前台和响应用户事件。 您将探索在更详细的活动状态，在本章后面。
- 活动，服务或广播接收机，当前正在执行的 onReceive 事件处理程序。
- 服务正在执行一个缓动的 onCreate，或 OnDestroy 事件处理程序。

可见进程可见，但处于非活动状态的进程是那些主办的“看得见”的活动。 顾名思义，可见活动是可见的，但他们不是在前台或响应用户事件。 这发生在一个活动仅是部分遮蔽（非全屏或透明的活动）。 通常有很少的可见的进程，他们将只在极端情况下被杀害，让活动的进程继续。

启动的服务进程的进程已经启动的托管服务。 服务支持正在进行的处理，应继续没有明显的界面。 因为服务不直接与用户交互，他们收到一个稍微较低的优先级比可见光活动。 他们仍然被认为是前台进程，并不会被杀死，除非活动或可见的进程所需要的资源。

后台进程的进程举办活动是不可见的，并没有任何被认为是后台进程已经启动的服务。一般将有大量的后台进程，Android 将杀死最后一个看到的第一个死亡的图案获得前台进程的资源。

空流程提高了整体系统性能的 Android 经常在内存中保留的申请后，他们达成的结束其一生。 Android 维护这个缓存的提高应用程序的启动时间时，他们再次推出。 这些过程是柔 tinely 杀害。

1.1.4 How to use memory efficiently 如何有效地使用内存

Android 的管理打开的应用程序是在后台运行，所以你不应该关闭这个。同时这意味着，关闭应用程序时，系统需要更多的内存。 然而，大多数 Android 用户不是很满意，它的东西它是如何，因为有时留下太多的进程运行，从而导致在日常生活中表现呆滞。 我们可以利用先进的任务杀手/任务管理器和它的工作非常好。

1.2 Low Memory Killer相关介绍

1.2.1 基本原理

Android的Low Memory Killer是在标准linux kernel的OOM基础上修改而来的一种内存管理机制，当系统内存不足时，杀死Bad进程释放其内存。Bad进程的选择标准有两个：oom_adj和占用内存的大小。oom_adj代表进程的优先级，数值越大，优先级越高，对应每个oom_adj都有一个空闲内存的阈值。Android Kernel每隔一段时间会检查当前空闲内存是否低于某个阈值，如果是，则杀死oom_adj最大的Bad进程，如果有两个以上Bad进程oom_adj相同，则杀死其中占用内存最多的进程。

1.2.2 Low Memory Killer与OOM的区别

OOM即Out of Memory是标准linux Kernel的一种内存管理机制，Low Memory Killer在它基础上作了改进：

. OOM基于多个标准给每个进程打分，分最高的进程将被杀死；Low Memory Killer则用oom_adj和占用内存的大小来选择Bad进程

. OOM在内存分配不足时调用，而Low Memory Killer每隔一段时间就会检查，一旦发现空闲内存低于某个阈值，则杀死Bad进程。

1.2.3 Low Memory Killer的实现

Low Memory Killer的源代码在drivers/staging/android/lowmemorykiller.c中，它是通过注册Cache Shrinker来实现的。Cache Shrinker是标准linux kernel回收内存页面的一种机制，它由内核线程kswapd监控，当空闲内存页面不足时，kswapd会调用注册的Shrinker回调函数，来回收内存页面。

Low Memory Killer 是在模块初始化时注册 Cache Shrinker 的，代码如下：

```
static int __init lowmem_init(void){
register_shrinker(&lowmem_shrinker); //    注册  Cache Shrinker
return 0;
}
```

lowmem_shrinker 的定义如下：

```
static struct shrinker lowmem_shrinker = {
.shrink = lowmem_shrink,
.seeks = DEFAULT_SEEKS * 16
};
```

register_shrinker 会将lowmem_shrink 加入Shrinker List 中，被kswapd 在遍历Shrinker List 时调用，而Low Memory Killer 的功能就是在lowmem_shrink 中实现的。

lowmem_shrink 用两个数组作为选择 Bad 进程的依据，这两个数组的定义如下：

```
static int lowmem_adj[6] = {
0,
1,
6,
12,
};

static int lowmem_adj_size = 4;

static size_t lowmem_minfree[6] = {
3*512, // 6MB
2*1024, // 8MB
4*1024, // 16MB
16*1024, // 64MB
};
```

lowmem_minfree 保存空闲内存的阈值，单位是一个页面4K，lowmem_adj 保存每个阈值对应的优先级。

lowmem_shrink 首先计算当前空闲内存的大小，如果小于某个阈值，则以该阈值对应的优先级为基准，遍历各个进程，计算每个进程占用内存的大小，找出优先级大于基准优先级的进程，在这些进程中选择优先级最大的杀死，如果优先级相同，则选择占用内存最多的进程。

lowmem_shrink 杀死进程的方法是向进程发送一个不可以忽略或阻塞的 SIGKILL 信号：

```
force_sig(SIGKILL, selected);
```

1.2.4 用户接口

设置空闲内存阈值的接口：/sys/module/lowmemorykiller/parameters/minfree，设置对应优先级的接口：/sys/module/lowmemorykiller/parameters/adj，设置各个进程优先级的接口： /proc/<进程pid>/oom_adj。

Android 启动时读取的配置文件/init.rc 中定义了相应的属性供 AP 使用并有设置这些参数：

```
# killed by the kernel. These are used in
ActivityManagerService.
setprop ro.FOREGROUND_APP_ADJ 0
setprop ro.VISIBLE_APP_ADJ 1
setprop ro.SECONDARY_SERVER_ADJ 2
setprop ro.BACKUP_APP_ADJ 2
setprop ro.HOME_APP_ADJ 4
setprop ro.HIDDEN_APP_MIN_ADJ 7
setprop ro.CONTENT_PROVIDER_ADJ 14
setprop ro.EMPTY_APP_ADJ 15
# Define the memory thresholds at which the above process
classes will
# be killed. These numbers are in pages (4k).
setprop ro.FOREGROUND_APP_MEM 1536
setprop ro.VISIBLE_APP_MEM 2048
setprop ro.SECONDARY_SERVER_MEM 4096
setprop ro.BACKUP_APP_MEM 4096
setprop ro.HOME_APP_MEM 4096
setprop ro.HIDDEN_APP_MEM 5120
setprop ro.CONTENT_PROVIDER_MEM 5632
setprop ro.EMPTY_APP_MEM 6144
# Write value must be consistent with the above properties.
# Note that the driver only supports 6 slots, so we have HOME_APP
at the
# same memory level as services.
write /sys/module/lowmemorykiller/parameters/adj
0,1,2,7,14,15
write /sys/module/lowmemorykiller/parameters/minfree
1536,2048,4096,5120,5632,6144
# Set init its forked children's oom_adj.
```

```
write /proc/1/oom_adj -16
```

从以上设置可以看出，将init进程oom_adj设置为-16，从而保证init进程永远不会被杀掉。

1.3 Ashmem相关介绍

1.3.1 基本原理

Android 的Ashmem是一种共享内存的机制，它基于mmap系统调用，不同进程可以将同一段物理内存映射到各自的虚拟地址控制，从而实现共享。

1.3.2 Ashmem与mmap的区别

mmap通过映射同一个普通文件实现进程间共享内存，普通文件被映射到进程地址空间后，进程可以像访问普通内存一样对文件进行访问，不必再调用read，write等操作。进程在映射空间对共享内存的改变并不直接写回到磁盘文件中，在调用munmap后才执行此操作。可以通过调用msync实现磁盘上文件内存与共享内存区的内容一致。

Ashmem与mmap的区别在于Ashmem与cache shrinker关联起来，可以控制cache shrinker在适当时机回收这些共享内存。

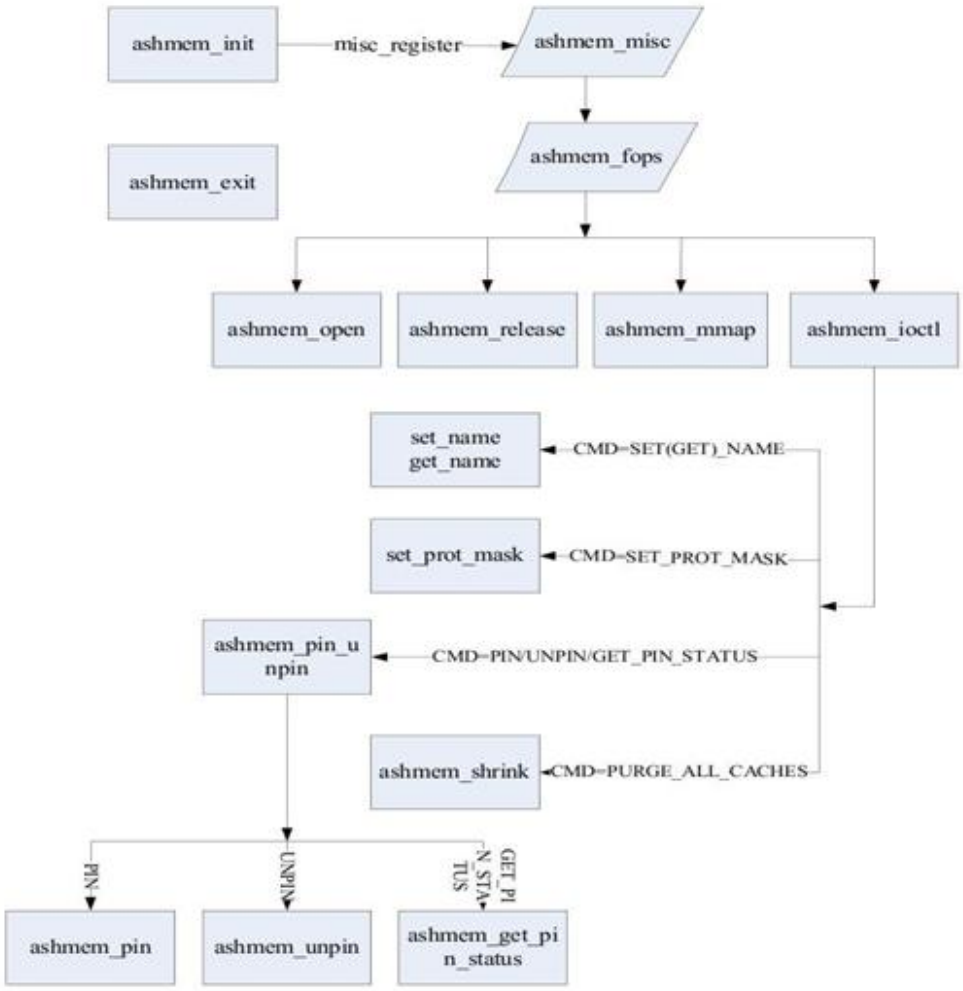
1.3.3 Ashmem的实现

Ashmem的源代码在mm/ashmem.c中，它通过注册Cache Shrinker回收内存，通过注册misc设备提供open，mmap等接口，mmap则通过tmpfs创建文件来分配内存，tmpfs将一块内存虚拟为一个文件，这样操作共享内存就相当于操作一个文件。

Ashmem用两个结构体ashmem_area和ashmem_range来维护分配的内存，ashmem_area代表共享的内存区域，ashmem_range则将这段区域以页为单位分为多个range。

ashmem_area有个unpinned_list成员，挂在这个list上的range可以被回收。ashmem_range有一个LRU链表，在cache shrink回收一个ashmem_area的某段内存时候，是根据LRU的原则来选择哪些页面优先被回收的。

Ashmem 的基本结构如下图所示。



下面依次简单分析主要函数功能：

- ashmem_init

这是module初始化函数，Ashmem是作为一个模块实现的。该函数主要功能：

- 调用kmem_cache_create分别创建struct ashmem_area和struct ashmem_range 的slab cache
- 调用misc_register注册 ahsmem driver
- 调用register_shrinker注册Ashmem的 Cache Shrinker

- ashmem_open

标准misc设备的open函数。它调用kmem_cache_zalloc分配一个 ashmem_area，并初始化各成员变量。

- ashmem_release

做与ashmem_open相反工作，释放tmpfs文件，ashmem_area及其ashmem_range。

- ashmem_mmap

mmap操作，主要就是调用shmem_file_setup从tmpfs文件系统中创建一个文件（实际上就

是一段RAM）给ashmem_area用，该文件代表着这段被共享的内存。Ashmem真正实现进程共享内存的机制是靠shmem这个linux标准机制提供的。

- ashmem_shrink

即Ashmem的cache shrink函数。它被mm/vmscan.c：：shrink_slab调用，或者被用户的 ioctl 命令调用。这个函数从LRU链表上回收指定数目的 unpinned ashmem_range。

- ashmem_ioctl

这个函数提供ioctl 接口，它实现了如下命令：

序号	命令	功能	调用的内部函数
1	ASHMEM_SET_NAME	设置 ashmem_area->name	set_name
2	ASHMEM_GET_NAME	获取 ashmem_area->name	get_name
3	ASHMEM_SET_SIZE	设置 ashmem_area->size	
4	ASHMEM_GET_SIZE	获取 ashmem_area->size	
5	ASHMEM_SET_PROT_MASK	设置 ashmem_area->prot_mask	set_prot_mask
6	ASHMEM_GET_PROT_MASK	获取 ashmem_area->prot_mask	
7	ASHMEM_PIN	Pin 一段 range	ashmem_pin_unpin→ashmem_pin
8	ASHMEM_UNPIN	unpin 一段 range	ashmem_pin_unpin→ashmem_unpin
9	ASHMEM_GET_PIN_STATUS	获取一个 range 是否被 pin	ashmem_pin_unpin→ashmem_get_pin_status
10	ASHMEM_PURGE_ALL_CACHES	Purge 一个 ashmem_area 里的所有 ashmem_range	ashmem_shrink

- ashmem_unpin

unpin一段内存。实现的方法很简单，就是分配一个ashmem_range，把它挂到ashmem_area -> unpinned_list上，并加到 LRU链表上。

ashmem_pin

pin一段内存，从 ashmem_area->unpinned_list上拿下这个 ashmem_range，由此可知，被unpin的range才能被回收，pin的range则不能回收。

1.3.4 用户接口

Ashmem 驱动创建了/dev/ashmem 设备文件，进程 A 可通过 open 打开该文件，用 ioctl 命令 ASHMEM_SET_NAME 和 ASHMEM_SET_SIZE 设置共享内存块的名字和大小，并将得到的 handle 传给 mmap，来获得共享的内存区域，进程 B 通过将相同的 handle 传给 mmap，获得同一块内存，handle 在进程间的传递可通过 Binder 来实现。

1.4 Pmem相关介绍

1.4.1 基本原理

Android Pmem是为了实现共享大尺寸连续物理内存而开发的一种机制，该机制对dsp，gpu等部件非常有用。Pmem相当于把系统内存划分出一部分单独管理，即不被linux mm管理，实际上linux mm根本看不到这段内存。

1.4.2 Pmem与Ashmem的区别

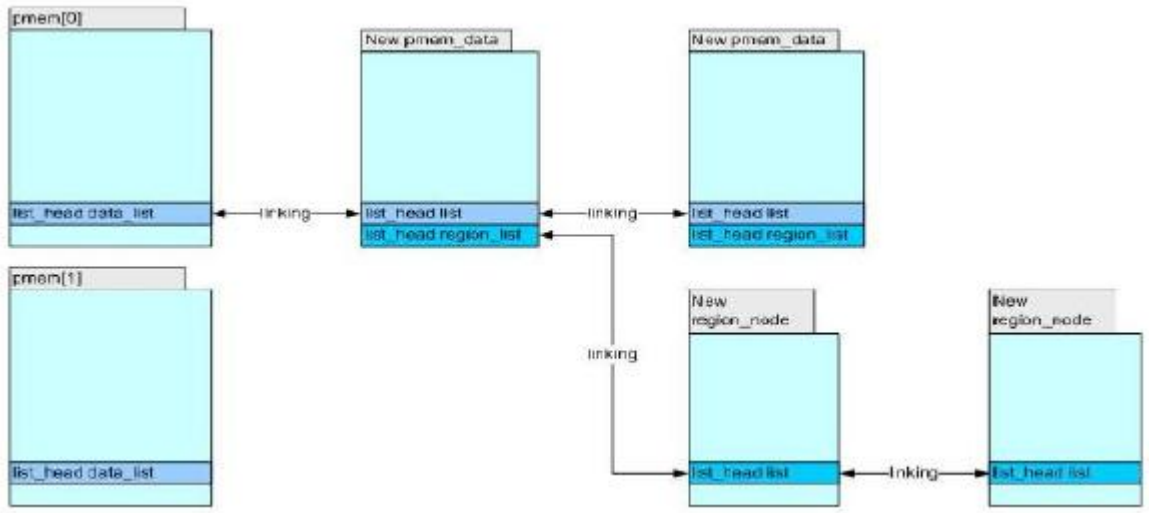
Pmem和Ashmem都通过mmap来实现共享内存，其区别在于Pmem的共享区域是一段连续的物理内存，而Ashmem的共享区域在虚拟空间是连续的，物理内存却不一定连续。dsp和某些设备只能工作在连续的物理内存上，这样cpu与dsp之间的通信就需要通过Pmem来实现。

1.4.3 Pmem的实现

Pmem的源代码在drivers/misc/pmem.c中，Pmem驱动依赖于linux的misc device和platform driver框架，一个系统可以有多个Pmem，默认的是最多10个。Pmem暴露4组操作，分别是platform driver的probe和remove操作； misc device的fops接口和vm_ops操作。模块初始化时会注册一个platform driver，在之后probe时，创建misc设备文件，分配内存，完成初始化工作。

Pmem通过pmem_info，pmem_data，pmem_region三个结构体维护分配的共享内存，其中pmem_info代表一个Pmem设备分配的内存块，pmem_data代表该内存块的一个子块，pmem_region则把每个子块分成多个区域。 pmem_data是分配的基本单位，即每次应用层要分配一块Pmem内存，就会有一个pmem_data来表示这个被分配的内存块，实际上在open的时候，并不是open一个pmem_info表示的整个Pmem内存块，而是创建一个pmem_data以备使用。一个应用可以通过ioctl来分配 pmem_data中的一个区域，并可以把它map到进程空间；并不一定每次都要分配和map整个pmem_data内存块。

上面三个数据结构的关系可以用下面的图来表示



Pmem驱动会创建/dev/pme、/dev/adsp，实现了pmem_open，pmem_mmap，pmem_release和pmem_ioctl，应用层可以通过open，mmap，close，ioctl来操作Pmem设备文件。其中ioctl支持的命令如下：

- PMEM_GET_PHYS 获取物理地址
- PMEM_MAP 映射一段内存
- PMEM_GET_SIZE 返回pmem分配的内存大小
- PMEM_UNMAP unmap一段内存
- PMEM_ALLOCATE 分配pmem空间，len 是参数，如果已分配则失败
- PMEM_CONNECT 将一个pmem file与其他相连接
- PMEM_GET_TOTAL_SIZE 返回pmem device内存的大小

1.4.4 用户接口

一个进程首先打开 Pmem 设备，通过 ioctl(PMEM_ALLOCATE)分配内存，它 mmap 这段内存到自己的进程空间后，该进程成为 master 进程。其他进程可以重新打开这个 pmem 设备，通过调用 ioctl(PMEM_CONNECT)将自己的 pmem_data 与 master 进程的 pmem_data 建立连接关系，这个进程就成为 client 进程。Client 进程可以通过 mmap 将 master Pmem 中的一段或全部重新映射到自己的进程空间，这样就实现了共享 Pmem 内存。如果是 GPU 或 DSP 则可以通过 ioctl(PMEM_GET_PHYS)获取物理地址进行操作。

1.5 Android 内存管理-SoftReference 的使用

很多时候我们需要考虑 Android 平台上的内存管理问题，Dalvik VM 给每个进程都分配了一定量的可用堆内存，当我们处理一些耗费资源的操作时可能会产生 OOM 错误(OutOfMemoryError)这样的异常。

在 Java 中内存管理，引用分为四大类，强引用 HardReference、弱引用 WeakReference、软引用 SoftReference 和虚引 用 PhantomReference。它们的区别也很明显，HardReference 对象是即使虚拟机内存吃紧抛出 OOM 也不会导致这一引用的对象被回 收，而 WeakReference 等更适合于一些数量不多，但体积稍微庞大的对象，在这四个引用中，它是最容易被垃圾回收的，而我们对于显示类似 Android Market 中每个应用的 App Icon

时可以考虑使用 SoftReference 来解决内存不至于快速回收，同时当内存短缺面临 Java VM 崩溃抛出 OOM 前时，软引用将会强制回收内存，最后的虚引用一般没有实际意义，仅仅观察 GC 的活动状态，对于测试比较实用同时必须和 ReferenceQueue 一起使用。

对于一组数据，我们可以通过 HashMap 的方式来添加一组 SoftReference 对象来临时保留一些数据，同时对于需要反复通过网络获取的不经常改变的内容，可以通过本地的文件系统或数据库来存储缓存，希望给国内做 App Store 这样的客户端一些改进建议。客户端载入时，每个列表项的图标是异步刷新显示的，但当我们快速的往下滚动到一定数量比如 50 个，再往回滚动时可能 我们看到了部分 App 的图标又重新开始加载，当然这一过程可能是从 SQLite 数据库中缓存的，但是在内存中已经通过类似 SoftReference 的方 式管理内存。

1.6 Android 垃圾回收实质内容解析

Android 手机操作系统中的代码编写方式对于有基础的编程人员来说是比较容易的。因为它是基于 Linux 平台的操作系统。我们在这里为大家介绍的是 Android 垃圾回收这一机制，以加深大家对这一系统的了解。

个人觉得 sp 和 wp 实际上就是 Android 为其 c++实现的自动垃圾 回收机制 ，具体到内部实现，sp 和 wp 实际上只是一个实现垃圾回收功能的接口而已，比如说对＊，—>的重载，是为了其看起来跟真正的指针一样，而真正实现垃 圾回收的是 refbase 这个基类。这部分代码的目录在：/frameworks/base/include/utils/RefBase.h

首先所有的类都会虚继承 refbase 类，因为它实现了达到 Android 垃圾回收所需要的所有 function，因此实际上所有的对象声明出来以后都具备了自动释放自己的能力，也就是说实际上智能指针就是我们的对象本身，它会维持一个对本身强引用和弱引用的计数，一旦强引用计数为 0 它就会释放 掉自己。

首先我们看 sp，sp 实际上不是 smart pointer 的缩写，而是 strong pointer，它实际上内部就包含了一个指向对象的指针而已。我们可以简单看看 sp 的一个构造函数：

```
template< typename T>

sp< T>::sp(T* other)

: m_ptr(other)

{

if (other) other->incStrong(this);

}
```

比如说我们声明一个对象：

```
1. sp< CameraHardwareInterface> hardware(new CameraHal());
```

实际上 sp 指针对本身没有进行什么操作，就是一个指针的赋值，包含了一个指向对象的指针，但是对象会对对象本身增加一个强引用计数，这个 incStrong 的实现就在 refbase 类里面。新 new 出来一个 CameraHal 对象，将它的值给 sp< CameraHardwareInterface>的时候，它的强引用计数就会从 0 变为 1。因此每次将对象赋值给一个 sp 指针的时候，对象的强引用 计数都会加 1，下面我们再看看 sp 的析构函数：

```
template< typename T>

sp< T>::~sp()

{

if (m_ptr) m_ptr->decStrong(this);

}
```

实际上每次 delete 一个 sp 对象的时候，sp 指针指向的对象的强引用计数就会减一，当对象的强引用技术 为 0 的时候这个对象就会被自动释放掉。

我们再看 wp，wp 就是 weak pointer 的缩写，弱引用指针的原理 ，就是为了应用 Android 垃圾回收来减少对那些胖子对象对内存的占用，我们首先来看 wp 的一个构造函数：

```
wp< T>::wp(T* other)

: m_ptr(other)

{

if (other) m_refs = other->createWeak(this);

}
```

它和 sp 一样实际上也就是仅仅对指针进行了赋值而已，对象本身会增加一个对自身的弱引用计数，同时 wp 还包含一个 m_ref 指针，这个指针主要是用来将 wp 升级为 sp 时候使用的：

```
template< typename T>

sp< T> wp< T>::promote() const

{

return sp< T>(m_ptr, m_refs);
```

```
    }

    template< typename T>

    sp< T>::sp(T* p, weakref_type* refs)

    : m_ptr((p && refs->attemptIncStrong(this)) ? p : 0)

    {

    }
```

实际上我们对 wp 指针唯一能做的就是将 wp 指针升级为一个 sp 指针，然后判断是否升级成功，如果成功说明对象依旧存在，如果失败说明对象已经被释放掉了。wp 指针我现在看到的是在单例中使用很多，确保 mhardware 对象只有一个，比如：

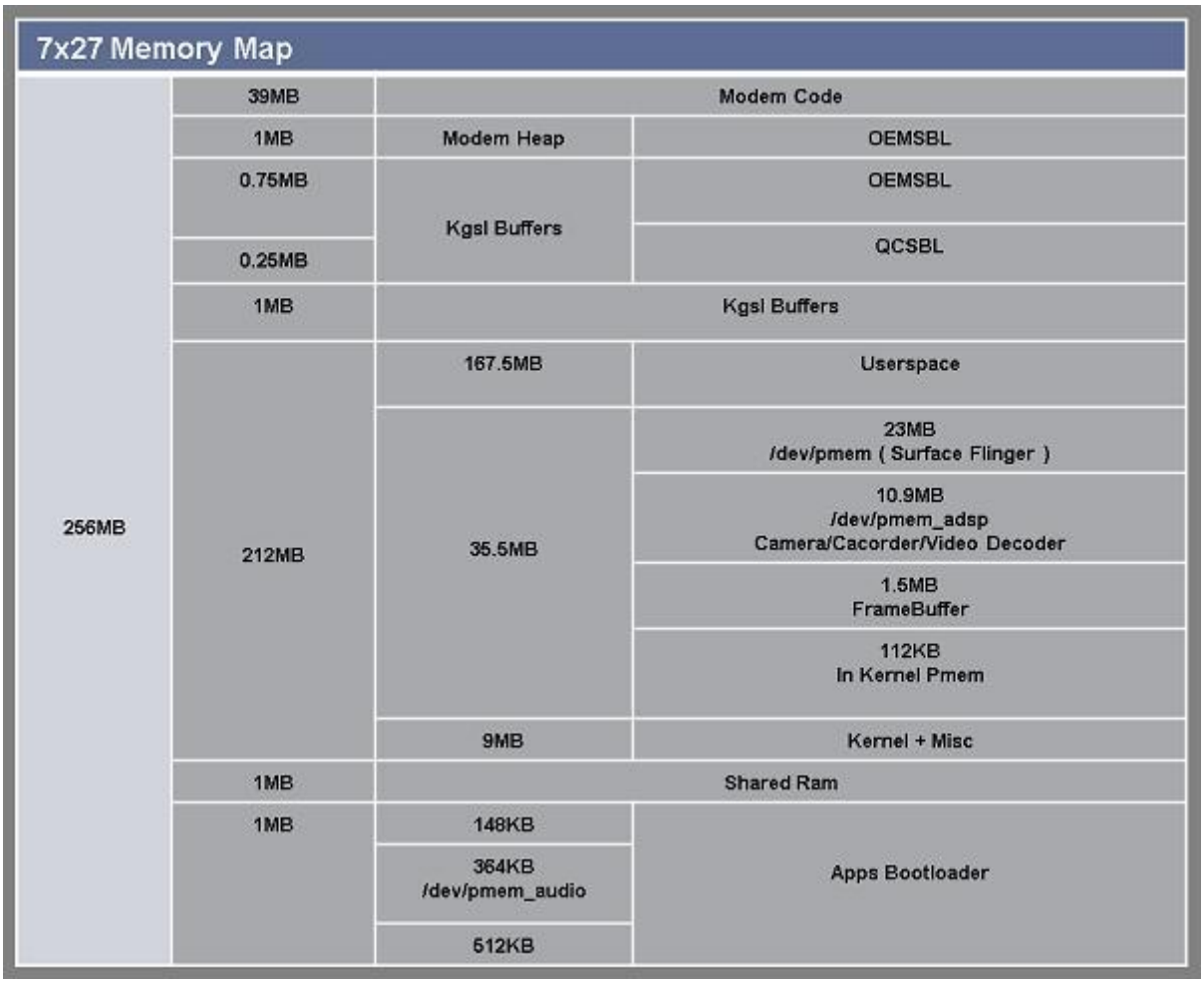
```
    wp< CameraHardwareInterface>
CameraHardwareStub::singleton;

    sp< CameraHardwareInterface>
CameraHal::createInstance()

    {

    LOG_FUNCTION_NAME

    if (singleton != 0) {

    sp< CameraHardwareInterface> hardware =
singleton.promote();

    if (hardware != 0) {

    return hardware;

    }

    }

    sp< CameraHardwareInterface> hardware(new
CameraHal()); //强引用加 1

    singleton = hardware;//弱引用加 1

    return hardware;//赋值构造函数，强引用加 1

    }
```

```
    //hardware 被删除，强引用减 1
```

1.7 Android 内存分配小结

有 Android 手机的童鞋们可能经常会有这样的疑问，为什么我的 G2 手机明明是 256M 的内存，可用任务管理器或者 free 之类的命令，看到的实际值会远远小于 256。看到网上的很多误导言论，这里我给大家澄清一下吧：

无图无真相，贴张 MSM7627 的内存分布图，一目了然：



由上图可以看到，内存主要分给 modem/bootloader/SMEM/pmem/Android 几个部分使用：

1、modem/bootloader/SMEM 部分一般会占用 40M~50M 的内存，上图占用了 43M。这部分主要用于跑 AMSS、 bootloader 及 RPC。如果你有源码的话，可以在 device\qcom\msmxxx/Boardconfig.mk 看到。还有一种方法，可以 用 adb pull /proc/config.gz .将 config.gz 文件 dump 下来，里面可以看到分配给 linux 的内存大小，用物理内存总大小减去这部分，就可以得到这部分的内存总开销。

2、Pmem 一般会占用 40M 的内存，上图占用了 38M 左右。系统的 framebuffer\mdp\video 等都会用到这部分 memory。如果你有 源码的话，可以在 kernel/arch/arm/mach-msm/Board-xxx.c 文件中看到 Pmem 的分配情况。譬如：

```
#define MSM_PMEM_MDP_SIZE 0x1B76000

#define MSM_PMEM_ADSP_SIZE 0xAE4000

#define MSM_PMEM_AUDIO_SIZE 0x5B000

#define MSM_FB_SIZE 0x177000

#define MSM_GPU_PHYS_SIZE 0x177000

#define PMEM_KERNEL_EBI1_SIZE 0x1C000
```

3、最后是给用户空间使用的 memory，用 free 或者 cat /proc/meminfo 等命令看到的是这部分的 memory 大小。大小是之前第一步传给 linux kernel 的 memory 大小减去 pmem 的占用部分再减去 linux kernel 和 ramdisk 大小。

Android 内存管理实例

2.1 Android 内存泄漏简介

不少人认为 JAVA 程序，因为有垃圾回收机制，应该没有内存泄露。这样认为就大错特错了。

其实如果我们一个程序中，已经不再使用某个对象，但是因为仍然有引用指向它，垃圾回收器就无法回收它，当然该对象占用的内存就无法被使用，这就造成了内存泄露。如果我们的 java 运行很久,而这种内存泄露不断的发生,最后就没内存可用了。当然 java 的，内存泄漏和 C/C++是不一样的。如果 java 程序完全结束后，它所有的对象就都不可达了，系统就可以对他们进行垃圾回收，它的内存泄露仅仅限于它本身，而不会影响整个系统的。C/C++的内存泄露就比较糟糕了，它的内存泄露是系统级，即使该 C/C++程序退出，它的泄露的内存也无法被系统回收，永远不可用了，除非重启机器。

Android 的一个应用程序的内存泄露对别的应用程序影响不大。为了能够使得 Android 应用程序安全且快速的运行，Android 的每个应用程序都会使用一个专有的 Dalvik 虚拟机实例来运行，它是由 Zygote 服务进程孵化出来的，也就是说每个应用程序都是在属于自己的进程中运行的。Android 为不同类型的进程分配了不同的内存使用上限，如果程序在运行过程中出现了内存泄漏的而造成应用进程使用的内存超过了这个上限，则会被系统视为内存泄漏，从而被 kill 掉，这使得仅仅自己的进程被 kill 掉，而不会影响其他进程（如果是 system_process 等系统进程出问题的话，则会引起系统重启）。

2.1.1 引用没释放造成的内存泄露

⑴注册没取消造成的内存泄露

这种 Android 的内存泄露比纯 java 的内存泄露还要严重，因为其他一些 Android 程序可能引用我们的 Anroid 程序的对象（比如注册机制）。即使我们的 Android 程序已经结束了，但是别的引用程序仍然还有对我们的 Android 程序的某个对象的引用，泄露的内存依然不能被垃圾回收。

示例:

假设我们希望在锁屏界面(LockScreen)中，监听系统中的电话服务以获取一些信息(如信号强度等)，则可以在 LockScreen 中定义一个 PhoneStateListener 的对象，同时将它注册到 TelephonyManager 服务中。对于 LockScreen 对象，当需要显示锁屏界面的时候就会创建一个 LockScreen 对象，而当锁屏界面消失的时候 LockScreen 对象就会被释放掉。

但是如果在释放 LockScreen 对象的时候忘记取消我们之前注册的 PhoneStateListener 对象，则会导致 LockScreen 无法被垃圾回收。如果不断的使锁屏界面显示和消失，则最终会由于大量的 LockScreen 对象没有办法被回收而引起 OutOfMemory,使得 system_process 进程挂掉。

虽然有些系统程序，它本身好像是可以自动取消注册的（当然不及时），但是我们还是应该在我们的程序中明确的取消注册，程序结束时应该把所有的注册都取消掉。

⑵集合中对象没清理造成的内存泄露

我们通常把一些对象的引用加入到了集合中，当我们不需要该对象时，并没有把它的引用从集合中清理掉，这样这个集合就会越来越大。如果这个集合是 static 的话，那情况就更严重了。

2.1.2 资源对象没关闭造成的内存泄露

资源性对象比如（Cursor，File 文件等）往往都用了一些缓冲，我们在不使用的时候，应该及时关闭它们，以便它们的缓冲及时回收内存。它们的缓冲不仅存在于 java 虚拟机内，还存在于 java 虚拟机外。如果我们仅仅是把它的引用设置为 null,而不关闭它们,往往会造成内存泄露。因为有些资源性对象,比如 SQLiteCursor(在析构函数 finalize（）,如果我们没有关闭它，它自己会调 close()关闭），如果我们没有关闭它，系统在回收它时也会关闭它，但是这样的效率太低了。因此对于资源性对象在不使用的时候，应该调用它的 close()函数，将其关闭掉，然后才置为 null.在我们的程序退出时一定要确保我们的资源性对象已经关闭。

程序中经常会进行查询数据库的操作，但是经常会有使用完毕 Cursor 后没有关闭的情况。如果我们的查询结果集比较小，对内存的消耗不容易被发现，只有在常时间大量操作的情况下才会复现内存问题，这样就会给以后的测试和问题排查带来困难和风险。

2.1.3 一些不良代码成内存压力

有些代码并不造成内存泄露，但是它们，或是对没使用的内存没进行有效及时的释放，或是没有有效的利用已有的对象而是频繁的申请新内存，对内存的回收和分配造成很大影响的，容易迫使虚拟机不得不给该应用进程分配更多的内存，造成不必要的内存开支。

⑴ Bitmap 没调用 recycle()

Bitmap 对象在不使用时,我们应该先调用 recycle()释放内存,然后才它设置为 null.

虽然 recycle()从源码上看，调用它应该能立即释放 Bitmap 的主要内存，但是测试结果显示它并没能立即释放内存。但是我它应该还是能大大的加速 Bitmap 的主要内存的释放。

⑵ 构造 Adapter 时，没有使用缓存的 convertView

以构造 ListView 的 BaseAdapter 为例，在 BaseAdapter 中提共了方法：

public View getView(int position, View convertView, ViewGroup parent)

来向 ListView 提供每一个 item 所需要的 view 对象。初始时 ListView 会从 BaseAdapter 中根据当前的屏幕布局实例化一定数量的 view 对象，同时 ListView 会将这些 view 对象缓存起来。当向上滚动 ListView 时，原先位于最上面的 list item 的 view 对象会被回收，然后被用来构造新出现的最下面的 list item。这个构造过程就是由 getView()方法完成的，getView()的第二个形参 View convertView 就是被缓存起来的 list item 的 view 对象(初始化时缓存中没有 view 对象则 convertView 是 null)。

由此可以看出，如果我们不去使用 convertView，而是每次都在 getView()中重新实例化一个 View 对象的话，即浪费时间，也造成内存垃圾，给垃圾回收增加压力，如果垃圾回收来不及的话，虚拟机将不得不给该应用进程分配更多的内存，造成不必要的内存开支。ListView 回收 list item 的 view 对象的过程可以查看:

android.widget.AbsListView.java --> void addScrapView(View scrap) 方法。

示例代码：

```
public View getView(int position, View convertView, ViewGroup parent) {
    View view = new Xxx(...);
    ... ...
    return view;
}
```

修正示例代码：

```
public View getView(int position, View convertView, ViewGroup parent)
{

    View view = null;
    if (convertView != null) {
        view = convertView;
        populate(view, getItem(position));
        ...
    } else {
        view = new Xxx(...);
        ...
    }
    return view;
}
```

2.2 Android 内存泄漏调试经验分享

2.2.1 概述

Java 编程中经常容易被忽视，但本身又十分重要的一个问题就是内存使用的问题。Android 应用主要使用 Java 语言编写，因此这个问题也同样会在 Android 开发中出现。本文不对 Java 编程问题做探讨，而是对于在 Android 中，特别是应用开发中的此类问题进行整理。

2.2.2 Android(Java)中常见的容易引起内存泄漏的不良代码

Android 主要应用在嵌入式设备当中，而嵌入式设备由于一些众所周知的条件限制，通常都不会有很高的配置，特别是内存是比较有限的。如果我们编写的代码当中有太多的对内存使用不当的地方，难免会使得我们的设备运行缓慢，甚至是死机。为了能够使得 Android 应用程序安全且快速的运行，Android 的每个应用程序都会使用一个专有的 Dalvik 虚拟机实例来运行，它是由 Zygote 服务进程孵化出来的，也就是说每个应用程序都是在属于自己的进程中运行的。一方面，如果程序在运行过程中出现了内存泄漏的问题，仅仅会使得自己的进程被 kill 掉，而不会影响其他进程（如果是 system_process 等系统进程出问题的话，则会引起系统重启）。另一方面 Android 为不同类型的进程分配了不同的内存使用上限，如果应用进程使用的内存超过了这个上限，则会被系统视为内存泄漏，从而被 kill 掉。Android 为应用进程分配的内存上限如下所示：

位置： /ANDROID_SOURCE/system/core/rootdir/init.rc 部分脚本

```
# Define the oom_adj values for the classes of processes that can be

# killed by the kernel.  These are used in ActivityManagerService.

    setprop ro.FOREGROUND_APP_ADJ 0

    setprop ro.VISIBLE_APP_ADJ 1

    setprop ro.SECONDARY_SERVER_ADJ 2

    setprop ro.BACKUP_APP_ADJ 2

    setprop ro.HOME_APP_ADJ 4

    setprop ro.HIDDEN_APP_MIN_ADJ 7

    setprop ro.CONTENT_PROVIDER_ADJ 14

    setprop ro.EMPTY_APP_ADJ 15
```

```
# Define the memory thresholds at which the above process classes will
# be killed.  These numbers are in pages (4k).
    setprop ro.FOREGROUND_APP_MEM 1536
    setprop ro.VISIBLE_APP_MEM 2048
    setprop ro.SECONDARY_SERVER_MEM 4096
    setprop ro.BACKUP_APP_MEM 4096
    setprop ro.HOME_APP_MEM 4096
    setprop ro.HIDDEN_APP_MEM 5120
    setprop ro.CONTENT_PROVIDER_MEM 5632
    setprop ro.EMPTY_APP_MEM 6144

# Write value must be consistent with the above properties.
# Note that the driver only supports 6 slots, so we have HOME_APP at the
# same memory level as services.
    write /sys/module/lowmemorykiller/parameters/adj 0,1,2,7,14,15

    write /proc/sys/vm/overcommit_memory 1
    write /proc/sys/vm/min_free_order_shift 4
    write /sys/module/lowmemorykiller/parameters/minfree 1536,2048,4096,5120,5632,6144
```

```
    # Set init its forked children's oom_adj.

    write /proc/1/oom_adj -16
```

正因为我们的应用程序能够使用的内存有限，所以在编写代码的时候需要特别注意内存使用问题。如下是一些常见的内存使用不当的情况。

⑴查询数据库没有关闭游标

描述：

程序中经常会进行查询数据库的操作，但是经常会有使用完毕 Cursor 后没有关闭的情况。如果我们的查询结果集比较小，对内存的消耗不容易被发现，只有在常时间大量操作的情况下才会复现内存问题，这样就会给以后的测试和问题排查带来困难和风险。

示例代码：

```
Cursor cursor = getContentResolver().query(uri ...);

if (cursor.moveToNext()) {

    ... ...

}
```

修正示例代码：

```
Cursor cursor = null;

try {

    cursor = getContentResolver().query(uri ...);

    if (cursor != null && cursor.moveToNext()) {

        ... ...

    }
```

```
} finally {
    if (cursor != null) {
        try {
            cursor.close();
        } catch (Exception e) {
            //ignore this
        }
    }
}
```

⑵ 构造 Adapter 时，没有使用缓存的 convertView

描述：

以构造 ListView 的 BaseAdapter 为例，在 BaseAdapter 中提高了方法：

public View getView(int position, View convertView, ViewGroup parent)

来向 ListView 提供每一个 item 所需要的 view 对象。初始时 ListView 会从 BaseAdapter 中根据当前的屏幕布局实例化一定数量的 view 对象，同时 ListView 会将这些 view 对象缓存起来。当向上滚动 ListView 时，原先位于最上面的 list item 的 view 对象会被回收，然后被用来构造新出现的最下面的 list item。这个构造过程就是由 getView()方法完成的，getView()的第二个形参 View convertView 就是被缓存起来的 list item 的 view 对象(初始化时缓存中没有 view 对象则 convertView 是 null)。

由此可以看出，如果我们不去使用 convertView，而是每次都在 getView()中重新实例化一个 View 对象的话，即浪费资源也浪费时间，也会使得内存占用越来越大。ListView 回收 list item 的 view 对象的过程可以查看:

android.widget.AbsListView.java --> void addScrapView(View scrap) 方法。

示例代码：

```
public View getView(int position, View convertView, ViewGroup parent) {
    View view = new Xxx(...);
    ... ...
    return view;
}
```

修正示例代码：

```
public View getView(int position, View convertView, ViewGroup parent) {
    View view = null;
    if (convertView != null) {
        view = convertView;
        populate(view, getItem(position));
        ...
    } else {
        view = new Xxx(...);
        ...
    }
    return view;
}
```

⑶ Bitmap 对象不在使用时调用 recycle()释放内存

描述：

有时我们会手工的操作 Bitmap 对象，如果一个 Bitmap 对象比较占内存，当它不在被使用的时候，可以调用 Bitmap.recycle()方法回收此对象的像素所占用的内存，但这不是必须的，视情况而定。可以看一下代码中的注释：

```
/**
 * Free up the memory associated with this bitmap's pixels, and mark the
 * bitmap as "dead", meaning it will throw an exception if getPixels() or
 * setPixels() is called, and will draw nothing. This operation cannot be
 * reversed, so it should only be called if you are sure there are no
 * further uses for the bitmap. This is an advanced call, and normally need
 * not be called, since the normal GC process will free up this memory when
 * there are no more references to this bitmap.
 */
```

⑷ 释放对象的引用

描述：

这种情况描述起来比较麻烦，举两个例子进行说明。

示例 A：

假设有如下操作

```
public class DemoActivity extends Activity {
    ... ...
    private Handler mHandler = ...
    private Object obj;
    public void operation() {
```

```
    obj = initObj();
    ...
    [Mark]
    mHandler.post(new Runnable() {
          public void run() {
           useObj(obj);
          }
    });
   }
}
```

我们有一个成员变量 obj，在 operation()中我们希望能够将处理 obj 实例的操作 post 到某个线程的 MessageQueue 中。在以上的代码中，即便是 mHandler 所在的线程使用完了 obj 所引用的对象，但这个对象仍然不会被垃圾回收掉，因为 DemoActivity.obj 还保有这个对象的引用。所以如果在 DemoActivity 中不再使用这个对象了，可以在[Mark]的位置释放对象的引用，而代码可以修改为：

```
... ...
public void operation() {
   obj = initObj();
   ...
   final Object o = obj;
   obj = null;
   mHandler.post(new Runnable() {
      public void run() {
```

```
            useObj(o);

        }

    }

}

... ...
```

示例 B:

假设我们希望在锁屏界面(LockScreen)中，监听系统中的电话服务以获取一些信息(如信号强度等)，则可以在 LockScreen 中定义一个 PhoneStateListener 的对象，同时将它注册到 TelephonyManager 服务中。对于 LockScreen 对象，当需要显示锁屏界面的时候就会创建一个 LockScreen 对象，而当锁屏界面消失的时候 LockScreen 对象就会被释放掉。

但是如果在释放 LockScreen 对象的时候忘记取消我们之前注册的 PhoneStateListener 对象，则会导致 LockScreen 无法被垃圾回收。如果不断的使锁屏界面显示和消失，则最终会由于大量的 LockScreen 对象没有办法被回收而引起 OutOfMemory,使得 system_process 进程挂掉。

总之当一个生命周期较短的对象 A，被一个生命周期较长的对象 B 保有其引用的情况下，在 A 的生命周期结束时，要在 B 中清除掉对 A 的引用。

⑸其他

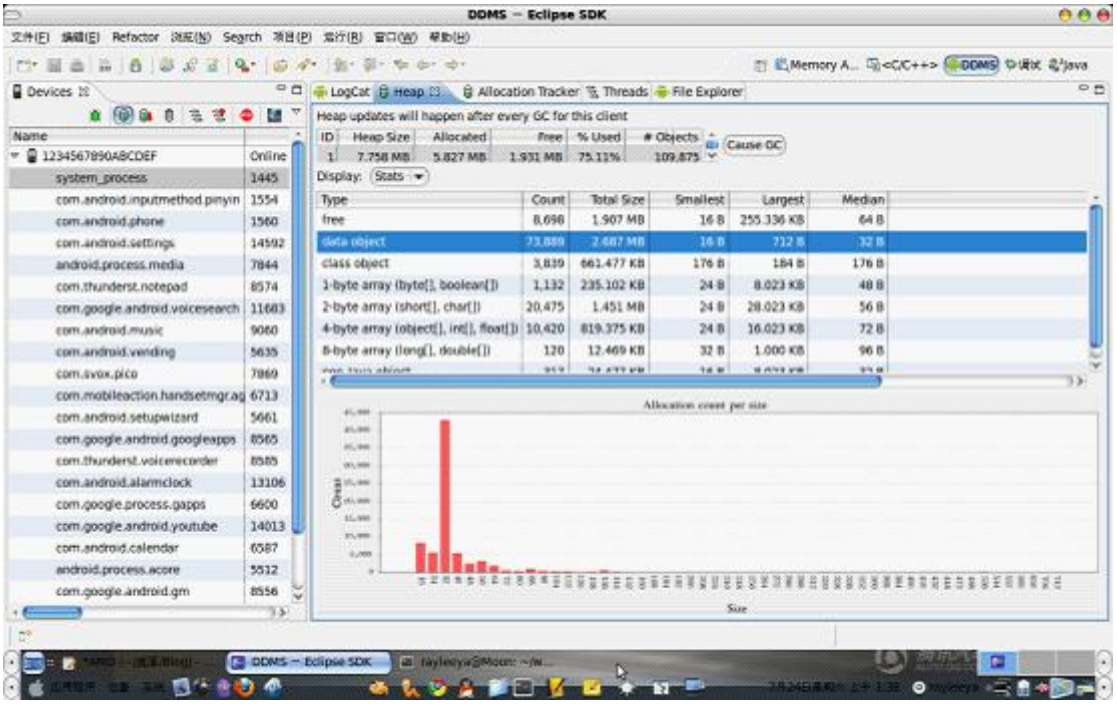
Android 应用程序中最典型的需要注意释放资源的情况是在 Activity 的生命周期中，在 onPause()、onStop()、onDestroy()方法中需要适当的释放资源的情况。由于此情况很基础，在此不详细说明，具体可以查看官方文档对 Activity 生命周期的介绍，以明确何时应该释放哪些资源。

2.2.3 内存监测工具 DDMS --> Heap

无论怎么小心，想完全避免 bad code 是不可能的，此时就需要一些工具来帮助我们检查代码中是否存在会造成内存泄漏的地方。Android tools 中的 DDMS 就带有一个很不错的内存监测工具 Heap(这里我使用 eclipse 的 ADT 插件，并以真机为例，在模拟器中的情况类似)。用 Heap 监测应用进程使用内存情况的步骤如下：

1. 启动 eclipse 后，切换到 DDMS 透视图，并确认 Devices 视图、Heap 视图都是打开的；
2. 将手机通过 USB 链接至电脑，链接时需要确认手机是处于“USB 调试”模式，而不是作为“Mass Storage”；

3. 链接成功后，在 DDMS 的 Devices 视图中将会显示手机设备的序列号，以及设备中正在运行的部分进程信息；
4. 点击选中想要监测的进程，比如 system_process 进程；
5. 点击选中 Devices 视图界面中最上方一排图标中的“Update Heap”图标；
6. 点击 Heap 视图中的“Cause GC”按钮；
7. 此时在 Heap 视图中就会看到当前选中的进程的内存使用量的详细情况[如图所示]。



说明：
a) 点击“Cause GC”按钮相当于向虚拟机请求了一次 gc 操作；
b) 当内存使用信息第一次显示以后，无须再不断的点击“Cause GC”，Heap 视图界面会定时刷新，在对应用的不断的操作过程中就可以看到内存使用的变化；
c) 内存使用信息的各项参数根据名称即可知道其意思，在此不再赘述。

如何才能知道我们的程序是否有内存泄漏的可能性呢。这里需要注意一个值：Heap 视图中部有一个 Type 叫做 data object，即数据对象，也就是我们的程序中大量存在的类类型的对象。在 data object 一行中有一列是“Total Size”，其值就是当前进程中所有 Java 数据对象的内存总量，一般情况下，这个值的大小决定了是否会有内存泄漏。可以这样判断：
a) 不断的操作当前应用，同时注意观察 data object 的 Total Size 值；
b) 正常情况下 Total Size 值都会稳定在一个有限的范围内，也就是说由于程序中的的代码良好，没有造成对象不被垃圾回收的情况，所以说虽然我们不断的操作会不断的生成很多对象，而在虚拟机不断的进行 GC 的过程中，这些对象都被回收了，内存占用量会会落到一个稳定的水平；
c) 反之如果代码中存在没有释放对象引用的情况，则 data object 的 Total Size 值在每次 GC 后不会有明显的回落，随着操作次数的增多 Total Size 的值会越来越大，
直到到达一个上限后导致进程被 kill 掉。

d）此处已 system_process 进程为例，在我的测试环境中 system_process 进程所占用的内存的 data object 的 Total Size 正常情况下会稳定在 2.2~2.8 之间，而当其值超过 3.55 后进程就会被 kill。

总之，使用 DDMS 的 Heap 视图工具可以很方便的确认我们的程序是否存在内存泄漏的可能性。

2.2.4 内存分析工具 MAT(Memory Analyzer Tool)

如果使用 DDMS 确实发现了我们的程序中存在内存泄漏，那又如何定位到具体出现问题的代码片段，最终找到问题所在呢？如果从头到尾的分析代码逻辑，那肯定会把人逼疯，特别是在维护别人写的代码的时候。这里介绍一个极好的内存分析工具 -- Memory Analyzer Tool(MAT)。

MAT 是一个 Eclipse 插件，同时也有单独的 RCP 客户端。官方下载地址、MAT 介绍和详细的使用教程请参见：www.eclipse.org/mat，在此不进行说明了。另外在 MAT 安装后的帮助文档里也有完备的使用教程。在此仅举例说明其使用方法。我自己使用的是 MAT 的 eclipse 插件，使用插件要比 RCP 稍微方便一些。

使用 MAT 进行内存分析需要几个步骤，包括：生成.hprof 文件、打开 MAT 并导入.hprof 文件、使用 MAT 的视图工具分析内存。以下详细介绍。

⑴生成.hprof 文件

生成.hprof 文件的方法有很多，而且 Android 的不同版本中生成.hprof 的方式也稍有差别，我使用的版本的是 2.1，各个版本中生成.prof 文件的方法请参考：
http://android.git.kernel.org/?p=platform/dalvik.git;a=blob_plain;f=docs/heap-profiling.html;hb=HEAD。

1. 打开 eclipse 并切换到 DDMS 透视图，同时确认 Devices、Heap 和 logcat 视图已经打开了；
2. 将手机设备链接到电脑，并确保使用“USB 调试”模式链接，而不是“Mass Storage“模式；
3. 链接成功后在 Devices 视图中就会看到设备的序列号，和设备中正在运行的部分进程；
4. 点击选中想要分析的应用的进程，在 Devices 视图上方的一行图标按钮中，同时选中“Update Heap”和“Dump HPROF file”两个按钮；
5. 这是 DDMS 工具将会自动生成当前选中进程的.hprof 文件，并将其进行转换后存放在 sdcard 当中，如果你已经安装了 MAT 插件，那么此时 MAT 将会自动被启用，并开始对.hprof 文件进行分析；

注意：第 4 步和第 5 步能够正常使用前提是我们需要有 sdcard，并且当前进程有向 sdcard 中写入的权限(WRITE_EXTERNAL_STORAGE)，否则.hprof 文件不会被生成，在 logcat 中会显示诸如

ERROR/dalvikvm(8574): hprof: can't open /sdcard/com.xxx.hprof-hptemp: Permission denied.

的信息。

如果我们没有 sdcard，或者当前进程没有向 sdcard 写入的权限（如 system_process），那我们可以这样做：

6. 在当前程序中，例如 framework 中某些代码中，可以使用 android.os.Debug 中的：

```
public static void dumpHprofData(String fileName) throws IOException
```

方法，手动的指定.hprof 文件的生成位置。例如：

```
xxxButton.setOnClickListener(new View.OnClickListener() {

    public void onClick(View view) {

        android.os.Debug.dumpHprofData("/data/temp/myapp.hprof");

        ... ...

    }

}
```

上述代码意图是希望在 xxxButton 被点击的时候开始抓取内存使用信息，并保存在我们指定的位置：/data/temp/myapp.hprof，这样就没有权限的限制了，而且也无须用 sdcard。但要保证/data/temp 目录是存在的。这个路径可以自己定义，当然也可以写成 sdcard 当中的某个路径。

⑵使用 MAT 导入.hprof 文件

1. 如果是 eclipse 自动生成的.hprof 文件，可以使用 MAT 插件直接打开（可能是比较新的 ADT 才支持）；
2. 如果 eclipse 自动生成的.hprof 文件不能被 MAT 直接打开，或者是使用 android.os.Debug.dumpHprofData()方法手动生成的.hprof 文件，则需要将.hprof 文件进行转换，转换的方法：

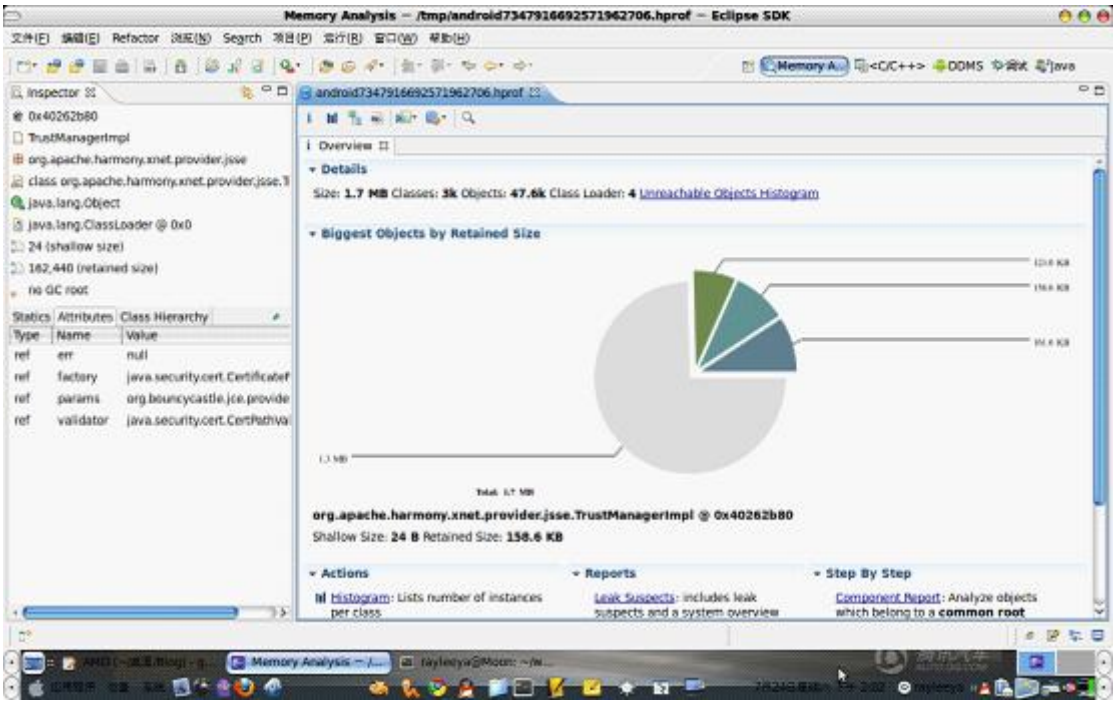
例如我将.hprof 文件拷贝到 PC 上的/ANDROID_SDK/tools 目录下，并输入命令 hprof-conv xxx.hprof yyy.hprof，其中 xxx.hprof 为原始文件，yyy.hprof 为转换过后的文件。转换过后的文件自动放在/ANDROID_SDK/tools 目录下。OK，到此为止，.hprof 文件处理完毕，可以用来分析内存泄露情况了。

3. 在 Eclipse 中点击 Windows->Open Perspective->Other->Memory Analyzer，或者打 Memory Analyzer Tool 的 RCP。在 MAT 中点击 File->Open File，浏览并导入刚刚转换而得到的.hprof 文件。

⑶使用 MAT 的视图工具分析内存

导入.hprof 文件以后，MAT 会自动解析并生成报告，点击 Dominator Tree，并按 Package 分组，选择自己所定义的 Package 类点右键，在弹出菜单中选择 List objects->With incoming references。这时会列出所有可疑类，右键点击某一项，并选择 Path to GC Roots -> exclude weak/soft references，会进一步筛选出跟程序相关的所有有内存泄露的类。据此，可以追踪到代码中的某一个产生泄露的类。

MAT 的界面如下图所示。



具体的分析方法在此不做说明了，因为在 MAT 的官方网站和客户端的帮助文档中有十分详尽的介绍。

了解 MAT 中各个视图的作用很重要，例如 www.eclipse.org/mat/about/screenshots.php 中介绍的。

总之使用 MAT 分析内存查找内存泄漏的根本思路，就是找到哪个类的对象的引用没有被释放，找到没有被释放的原因，也就可以很容易定位代码中的哪些片段的逻辑有问题了。

2.3 避免 Android 内存泄露(译)

Android 的应用被限制为最多占用 16m 的内存，至少在 T-Mobile G1 上是这样的（当然现在已经有几百兆的内存可以用了——译者注）。它包括电话本身占用的和开发者可以使用的两部分。即使你没有占用全部内存的打算，你也 应该尽量少的使用内存，以免别的应用在运行的时候关闭你的应用。Android 能在内存中保持的应用越多，用户在切换应用的时候就越快。作为我的一项工 作，我仔细研究了 Android 应用的内存泄露问题，大多数情况下它们是由同一个错误引起的，那就是对一个上下文（Context）保持了长时间的引用。

在 Android 中，上下文（Context）被用作很多操作中，但是大部分是载入和访问资源。这就是所有的 widget 都会在它们的构造函数中接 受一个上下文（Context）参数。在一个合格的 Android 应用中，你通常能够用到两种上下文(Context)：活动(Activity)和应用（Application）。活动（Activity）通常被传递给需要上下文（Context）参数的类或者方法：

```
@Override
```

```
protected void onCreate(Bundle state) {
  super.onCreate(state);

  TextView label = new TextView(this);
  label.setText("Leaks are bad");

  setContentView(label);
}
```

这就意味着那个 View 有一个对整个活动(Activity)的引用并且对这个活动(Activity)中保持的所有对象有保持了引用；通常它们包 括整个 View 的层次和它的所有资源。因此，如果你“泄露”了上下文（Context）（这里“泄露”的意思是你保持了一个引用并且组织 GC 收集它），你 将造成大量的内存泄露。如果你不够小心的话，“泄露”一整个活动（Activity）是件非常简单的事情。

当屏幕的方向改变时系统会默认的销毁当前的活动（Activity）并且创建一个新的并且保持了它的状态。这样的结果就是 Android 会从资源中 重新载入应用的 UI。现在想象一下，你写了一个应用，有一个非常大的位图，并且你并不想在每次旋转时都重新载入。保留它并且每次旋转不重新加载的最简单的 办法就是把它保存在一个静态字段上：

```
private static Drawable sBackground;

@Override
protected void onCreate(Bundle state) {
  super.onCreate(state);

  TextView label = new TextView(this);
  label.setText("Leaks are bad");

  if (sBackground == null) {
    sBackground = getDrawable(R.drawable.large_bitmap);
  }
  label.setBackgroundDrawable(sBackground);

  setContentView(label);
}
```

这段代码非常快，同时也错的够离谱。它泄露了当第一次屏幕角度改变时创建的第一个活动（Activity）。当一个 Drawable 被附加到一个 View，这个 View 被设置为 drawable 的一个回调。在上面的代码片断中，这意味着这个 Drawable 对 TextView 有一个引用，同时这个 TextView 对 Activity（Context 对象）保持着引用，同时这个 Activity 对很多对象又有引用（这个多少还要看你的代码了）。

这个例子是造成 Context 泄露的最简单的一个原因，你可以看一下我们在主屏幕源码(查看 unbindDrawables()方法)中是通过在 Activity 销毁时设置保存过的 Drawable 的回调为空来解决这个问题的。更为有趣的是，你可以创建一个 context 泄露的链，当然这非常的糟糕。它们可以让你飞快的用光所有的内存。

有两种简单的方法可以避免与 context 相关的内存泄露。最明显的一个就是避免在 context 的自身的范围外使用它。上面的例子展示了在类内部 的一个静态的引用和它们对外部类的间接引用是非常危险的。第二个解决方案就是使用 Application Context。这个 context 会伴随你的应用而存在，并且不依赖 Activity 的的生命周期。如果你计划保持一个需要 context 的长生命周期 的对象，请记得考虑 Application 对象。你可以非常方便的通过调用 Context.getApplicationContext() 或者 Activity.getApplication() 获取它。

总之，为了避免涉及到 context 的内存泄露，请记住如下几点：

1. 不要对一个 Activity Context 保持长生命周期的引用（一个对 Activity 的引用应该与 Activity 自身的生命周期相同）
2. 尝试使用应用上下文（context-application）代替活动上下文（context-activity）
3. 如果你不能控制它们的生命周期，在活动（Activity）中避免使用不是静态的内部类，使用静态类并且使用弱引用到活动（Activity）的 内部。对于这个问题的解决方法是使用静态的内部类与一个弱引用（WeakReference）的外部类。就像 ViewRoot 和它的 W 内部类那么实现的。
4. 垃圾回收器对于内存泄露来说并不是百分百保险的。

2.4 Android-避免出现 bitmap 内存限制 OUT OF MEMORY 的一种方法

在编写 Android 程序的时候，我们总是难免会碰到 OOM（OUT OF MEMORY）的错误，那么这个错误究竟是怎么来的呢，可以先看一下这篇文章 ANDROID BITMAP 内存限制 OOM,OUT OF MEMORY。

这里，我使用 Gallery 来举例，在模拟器中，不会出现 OOM 错误，但是，一旦把程序运行到真机里，图片文件一多，必然会出现 OOM,我们通过做一些额外的处理来避免。

1.创建一个图片缓存对象 HashMap dataCache，integer 对应 Adapter 中的位置 position，我们只用缓存处在显示中的图片，对于之外的位置，如果 dataCache 中有对应的图片，我们需要进行回收内存。在这个例子中，Adapter 对象的 getView 方法首先判断该位置是否有缓存的 bitmap，如果没有，则解码图片（bitmapDecoder.getPhotoItem，BitmapDecoder 类见后面）并返回 bitmap 对象，设置 dataCache 在该位置上的 bitmap 缓存以便之后使用；若是该位置存在缓存，则直接取出来使用，避免了再一次调用底层的解码图像需要的内存开销。有时为了提高 Gallery 的更新速度，我们还可以预存储一些位置上的 bitmap，比如存储显示区域位置外向上 3 个向下 3 个位置的 bitmap，这样上或下滚动 Gallery 时可以加快 getView 的获取。

```
public View getView(int position, View convertView, ViewGroup parent)
{



                                    if(convertView==null){

                                                LayoutInflater inflater  =
```

```
LayoutInflater.from(context);

                                        convertView =
inflater.inflate(R.layout.photo_item, null);



                        holder = new ViewHolder();

                        holder.photo = (ImageView)
convertView.findViewById(R.id.photo_item_image);

                        holder.photoTitle = (TextView)
convertView.findViewById(R.id.photo_item_title);

                        holder.photoDate = (TextView)
convertView.findViewById(R.id.photo_item_date);

                        convertView.setTag(holder);

                            }else {

                           holder = (ViewHolder)
convertView.getTag();

                        }

                            cursor.moveToPosition(position);



                            Bitmap current =
dateCache.get(position);

                            if(current != null){//如果缓存中已解码
该图片，则直接返回缓存中的图片


holder.photo.setImageBitmap(current);

                            }else {

                                        current =
bitmapDecoder.getPhotoItem(cursor.getString(1), 2) ;
```

```
holder.photo.setImageBitmap(current);
                                        dateCache.put(position,
current);
                                }

holder.photoTitle.setText(cursor.getString(2));

holder.photoDate.setText(cursor.getString(4));
                                return convertView;
                        }


                }
```

BitmapDecoder.class

```
package com.wuyi.bestjoy;


    import java.io.FileNotFoundException;
    import java.io.FileOutputStream;


    import android.content.Context;
    import android.graphics.Bitmap;
    import android.graphics.BitmapFactory;
    import android.graphics.Matrix;


    public class BitmapDecoder {
            private static final String TAG = "BitmapDecoder";
```

```
        private Context context;

        public BitmapDecoder(Context context) {

                this.context = context;

        }



        public Bitmap getPhotoItem(String filepath,int size) {

                BitmapFactory.Options options = new
BitmapFactory.Options();

                options.inSampleSize=size;

                Bitmap bitmap =
BitmapFactory.decodeFile(filepath,options);

                bitmap=Bitmap.createScaledBitmap(bitmap, 180,
251, true);//预先缩放，避免实时缩放，可以提高更新率

                return bitmap;



        }

    }
```

2.由于Gallery控件的特点，总有一个item处于当前选择状态，我们利用此时进行dataCache中额外不用的bitmap的清理，来释放内存。

```
    @Override

        public void onItemSelected(AdapterView<?> parent, View
view, int position,long id) {



                releaseBitmap();

                Log.v(TAG, "select id:"+ id);
```

```
            }


    private void releaseBitmap(){

        //在这，我们分别预存储了第一个和最后一个可见位置之外的3个位置的bitmap

        //即dataCache中始终只缓存了（M＝6＋Gallery当前可见view的个数）M个bitmap

                    int start = mGallery.getFirstVisiblePosition()-3;

                    int end = mGallery.getLastVisiblePosition()+3;

                    Log.v(TAG, "start:"+ start);

                    Log.v(TAG, "end:"+ end);

                    //释放position<start之外的bitmap资源

                    Bitmap delBitmap;

                    for(int del=0;del<start;del++){

                        delBitmap = dateCache.get(del);

                        if(delBitmap != null){

                            //如果非空则表示有缓存的bitmap，需要清理

                            Log.v(TAG, "release position:"+ del);

                            //从缓存中移除该del->bitmap的映射

                            dateCache.remove(del);

                            delBitmap.recycle();

                        }
```

```
                }

                freeBitmapFromIndex(end);

        }

        /**
         * 从某一位置开始释放bitmap资源
         * @param index
         */
        private void freeBitmapFromIndex(int end) {
                //释放之外的bitmap资源
                Bitmap delBitmap;
                for(int del =end+1;del<dateCache.size();del++){
                        delBitmap = dateCache.get(del);
                        if(delBitmap != null){
                                dateCache.remove(del);
                                delBitmap.recycle();
                                Log.v(TAG, "release position:"+
del);
                        }

                }
        }
```

经过这些额外的操作，有效的避免了OOM的问题。

2.5 内存溢出的解决办法

在模拟器上给 gallery 放入图片的时候，出现 java.lang.OutOfMemoryError: bitmap size exceeds VM budget 异常，图像大小超过了 RAM 内存。

模拟器 RAM 比较小，只有 8M 内存，当我放入的大量的图片（每个 100 多 K 左右），就出现上面的原因。由于每张图片先前是压缩的情况。放入到 Bitmap 的时候，大小会变大，导致超出 RAM 内存，具体解决办法如下：

```
//解决加载图片 内存溢出的问题
            //Options 只保存图片尺寸大小，不保存图片到内存
        BitmapFactory.Options opts = new BitmapFactory.Options();
        //缩放的比例，缩放是很难按准备的比例进行缩放的，其值表明缩放的倍数，
SDK 中建议其值是 2 的指数值,值越大会导致图片不清晰
        opts.inSampleSize = 4;
        Bitmap bmp = null;
            bmp = BitmapFactory.decodeResource(getResources(),
mImageIds[position],opts);

        ...

       //回收
        bmp.recycle();
```

通过上面的方式解决了，但是这并不是最完美的解决方式。

通过一些了解，得知如下：

2.5.1 优化 Dalvik 虚拟机的堆内存分配

对于 Android 平台来说，其托管层使用的 Dalvik Java VM 从目前的表现来看还有很多地方可以优化处理，比如我们在开发一些大型游戏或耗资源的应用中可能考虑手动干涉 GC 处理，使用 dalvik.system.VMRuntime 类提供的 setTargetHeapUtilization 方法可以增强程序堆内存的处理效率。当然具体 原理我们可以参考开源工程，这里我们仅说下使用方法： private final static float TARGET_HEAP_UTILIZATION = 0.75f; 在程序 onCreate 时就可以调用 VMRuntime.getRuntime().setTargetHeapUtilization(TARGET_HEAP_UTILIZATION); 即可。

2.5.2 Android 堆内存也可自己定义大小

对于一些 Android 项目，影响性能瓶颈的主要是 Android 自己内存管理机制问题，目前手机厂商对 RAM 都比较吝啬，对于软件的流畅性来说 RAM 对 性能的影响十分敏感，除了 优化 Dalvik 虚拟机的堆内存分配外，我们还可以强制定义自己软件的对内存大小，

我们使用 Dalvik 提供的 dalvik.system.VMRuntime 类来设置最小堆内存为例:

```
private final static int CWJ_HEAP_SIZE = 6* 1024* 1024 ;

VMRuntime.getRuntime().setMinimumHeapSize(CWJ_HEAP_SIZE); //设置最小 heap 内存为 6MB 大小。当然对于内存吃紧来说还可以通过手动干涉 GC 去处理
```

bitmap 设置图片尺寸，避免 内存溢出 OutOfMemoryError 的优化方法
★android 中用 bitmap 时很容易内存溢出，报如下错误：Java.lang.OutOfMemoryError : bitmap size exceeds VM budget

- 主要是加上这段：

```
BitmapFactory.Options options = new BitmapFactory.Options();
        options.inSampleSize = 2;
```

- eg1：(通过 Uri 取图片)

```
private ImageView preview;
BitmapFactory.Options options = new BitmapFactory.Options();
        options.inSampleSize = 2;//图片宽高都为原来的二分之一，即图片为原来的四分之一
        Bitmap bitmap = BitmapFactory.decodeStream(cr
            .openInputStream(uri), null, options);
        preview.setImageBitmap(bitmap);
```

以上代码可以优化内存溢出，但它只是改变图片大小，并不能彻底解决内存溢出。

- eg2:（通过路径去图片）

```
private ImageView preview;
private String fileName= "/sdcard/DCIM/Camera/2010-05-14 16.01.44.jpg";
BitmapFactory.Options options = new BitmapFactory.Options();
        options.inSampleSize = 2;//图片宽高都为原来的二分之一，即图片为原来的四分之一
            Bitmap b = BitmapFactory.decodeFile(fileName, options);
            preview.setImageBitmap(b);
            filePath.setText(fileName);
```

2.5.3 Android 还有一些性能优化的方法

- 首先内存方面，可以参考 Android 堆内存也可自己定义大小 和 优化 Dalvik 虚拟机的堆内存分配

- 基础类型上，因为 Java 没有实际的指针，在敏感运算方面还是要借助 NDK 来完成。Android123 提示游戏开发者，这点比较有意思的是 Google 推出 NDK 可能是帮助游戏开发人员，比如 OpenGL ES 的支持有明显的改观，本地代码操作图形界面是很必要的。

- 图形对象优化，这里要说的是 Android 上的 Bitmap 对象销毁，可以借助 recycle()方

法显示让 GC 回收一个 Bitmap 对象，通常对一个不用的 Bitmap 可以使用下面的方式，如

```
if(bitmapObject.isRecycled()==false) //如果没有回收
        bitmapObject.recycle();
```

- 目前系统对动画支持比较弱智对于常规应用的补间过渡效果可以，但是对于游戏而言一般的美工可能习惯了 GIF 方式的统一处理，目前 Android 系统仅能预览 GIF 的第一帧，可以借助 J2ME 中通过线程和自己写解析器的方式来读取 GIF89 格式的资源。

- 对于大多数 Android 手机没有过多的物理按键可能我们需要想象下了做好手势识别 GestureDetector 和重力感应来实现操控。通常我们还要考虑误操作问题的降噪处理。

2.6 Java 系统中内存泄漏测试方法的研究

2.6.1 问题的提出

笔者曾经参与开发的网管系统，系统规模庞大，涉及上百万行代码。系统主要采用 Java 语言开发，大体上分为客户端、服务器和数据库三个层次。在版本进入测试和试用的过程中，现场人员和测试部人员纷纷反映:系统的稳定性比较差，经常会出现服务器端运行一昼夜就死机的现象，客户端跑死的现象也比较频繁地发生。对于网管系统来讲，经常性的服务器死机是个比较严重的问题，因为频繁的死机不仅可能导致前后台数据不一致，发生错误，更会引起用户的不满，降低客户的信任度。因此，服务器端的稳定性问题必须尽快解决。

解决思路

通过察看服务器端日志，发现死机前服务器端频繁抛出 OutOfMemoryException 内存溢出错误，因此初步把死机的原因定位为内存泄漏引起内存不足，进而引起内存溢出错误。如何查找引起内存泄漏的原因呢?有两种思路:第一种，安排有经验的编程人员对代码进行走查和分析，找出内存泄漏发生的位置;第二种，使用专门的内存泄漏测试工具 Optimizeit 进行测试。这两种方法都是解决系统稳定性问题的有效手段，使用内存测试工具对于已经暴露出来的内存泄漏问题的定位和解决非常有效;但是软件测试的理论也告诉我们，系统中永远存在一些没有暴露出来的问题，而且，系统的稳定性问题也不仅仅只是内存泄漏的问题，代码走查是提高系统的整体代码质量乃至解决潜在问题的有效手段。基于这样的考虑，我们的内存稳定性工作决定采用代码走查结合测试工具的使用，双管齐下，争取比较彻底地解决系统的稳定性问题。

在代码走查的工作中，安排了对系统业务和开发语言工具比较熟悉的开发人员对应用的代码进行了交叉走查，找出代码中存在的数据库连接声明和结果集未关闭、代码冗余和低效等故障若干，取得了良好的效果，文中主要讲述结合工具的使用对已经出现的内存泄漏问题的定位方法。

2.6.2 内存泄漏的基本原理

在 C++语言程序中，使用 new 操作符创建的对象，在使用完毕后应该通过 delete 操作符显示地释放，否则，这些对象将占用堆空间，永远没有办法得到回收，从而引起内存空间的泄漏。如下的简单代码就可以引起内存的泄漏:

```
void function(){
   Int[] vec = new int[5];
}
```

在 function()方法执行完毕后，vec 数组已经是不可达对象，在 C++语言中，这样的对象永远也得不到释放，称这种现象为内存泄漏。

而 Java 是通过垃圾收集器(Garbage Collection，GC)自动管理内存的回收，程序员不需要通过调用函数来释放内存，但它只能回收无用并且不再被其它对象引用的那些对象所占用的空间。在下面的代码中，循环申请 Object 对象，并将所申请的对象放入一个 Vector 中，如果仅仅释放对象本身，但是因为 Vector 仍然引用该对象，所以这个对象对 GC 来说是不可回收的。因此，如果对象加入到 Vector 后，还必须从 Vector 中删除，最简单的方法就是将 Vector 对象设置为 null。

```
Vector v = new Vector(10);
for (int i = 1; i < 100; i++)
{
   Object o = new Object();
   v.add(o);
   o = null;
}//此时，所有的 Object 对象都没有被释放，因为变量 v 引用这些对象。
```

实际上无用，而还被引用的对象，GC 就无能为力了(事实上 GC 认为它还有用)，这一点是导致内存泄漏最重要的原因。

Java 的内存回收机制可以形象地理解为在堆空间中引入了重力场，已经加载的类的静态变量和处于活动线程的堆栈空间的变量是这个空间的牵引对象。这里牵引对象是指按照 Java 语言规范，即便没有其它对象保持对它的引用也不能够被回收的对象，即 Java 内存空间中的本原对象。当然类可能被去加载，活动线程的堆栈也是不断变化的，牵引对象的集合也是不断变化的。对于堆空间中的任何一个对象，如果存在一条或者多条从某个或者某几个牵引对象到该对象的引用链，则就是可达对象，可以形象地理解为从牵引对象伸出的引用链将其拉住，避免掉到回收池中;而其它的不可达对象由于不存在牵引对象的拉力，在重力的作用下将掉入回收池。在图 1 中，A、B、C、D、E、F 六个对象都被牵引对象所直接或者间接地“牵引”，使得它们避免在重力的作用下掉入回收池。如果 TR1-A 链和 TR2-D 链断开，则 A、B、C 三个对象由于失去牵引，在重力的作用下掉入回收池(被回收)，D 对象也是同样的原因掉入回收池，而 F 对象仍然存在一个牵引链(TR3-E-F)，所以不会被回收，如图 2、3 所示。

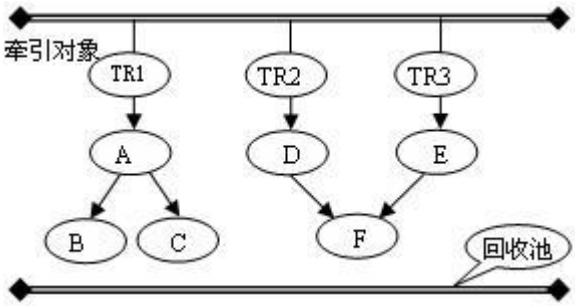


图 1 初始状态

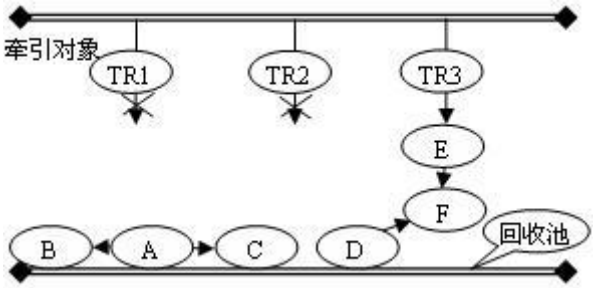


图 2 TR1-A 链和 TR2-D 链断开，A、B、C、D 掉入回收池

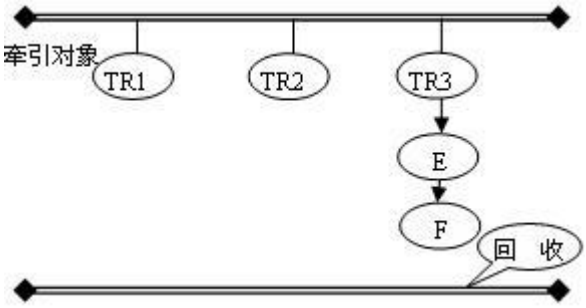


图 3 A、B、C、D 四个对象被回收

通过前面的介绍可以看到，由于采用了垃圾回收机制，任何不可达对象都可以由垃圾收集线程回收。因此通常说的 Java 内存泄漏其实是指无意识的、非故意的对象引用，或者无意识的对象保持。无意识的对象引用是指代码的开发人员本来已经对对象使用完毕，却因为编码的错误而意外地保存了对该对象的引用(这个引用的存在并不是编码人员的主观意愿)，从而使得该对象一直无法被垃圾回收器回收掉，这种本来以为可以释放掉的却最终未能被释放的空间可以认为是被"泄漏了"。

这里通过一个例子来演示 Java 的内存泄漏。假设有一个日志类 Logger，其提供一个静态的 log(String msg)方法，任何其它类都可以调用 Logger.Log(message)来将 message 的内容记录到系统的日志文件中。Logger 类有一个类型为 HashMap 的静态变量 temp，每次在执行 log(message)方法的时候，都首先将 message 的值丢入 temp 中(以当前线程+当前时间为键)，在方法退出之前再从 temp 中将以当前线程和当前时间为键的条目删除。注意，这里当前时间是不断变化的，所以 log 方法在退出之前执行删除条目的操作并不能删除方法执行之初丢入的条目。这样，任何一个作为参数传给 log 方法的字符串最终由于被 Logger 的静态变量 temp 引用，而无法得到回收，这种违背实现者主观意图的无意识的对象保持就是我们所说的 Java 内存泄漏。

2.6.3 鉴别泄漏对象的方法

一般说来，一个正常的系统在其运行稳定后其内存的占用量是基本稳定的，不应该是无限制的增长的，同样，对任何一个类的对象的使用个数也有一个相对稳定的上限，不应该是持续增长的。根据这样的基本假设，我们可以持续地观察系统运行时使用的内存的大小和各实例的个数，如果内存的大小持续地增长，则说明系统存在内存泄漏，如果某个类的实例的个数持续地增长，则说明这个类的实例可能存在泄漏情况。

Optimizeit 是 Borland 公司的产品，主要用于协助对软件系统进行代码优化和故障诊断，其功能众多，使用方便，其中的 OptimizeIt Profiler 主要用于内存泄漏的分析。Profiler 的堆视图(如图 4)就是用来观察系统运行使用的内存大小和各个类的实例分配的个数的，其界面如图四所示，各列自左至右分别为类名称、当前实例个数、自上个标记点开始增长的实例个数、占用的内存空间的大小、自上次标记点开始增长的内存的大小、被释放的实例的个数信息、自上次标记点开始增长的内存的大小被释放的实例的个数信息，表的最后一行是汇总数据，分别表示目前 JVM 中的对象实例总数、实例增长总数、内存使用总数、内存使用增长总数等。

在实践中，可以分别在系统运行四个小时、八个小时、十二个小时和二十四个小时时间点记录当时的内存状态(即抓取当时的内存快照，是工具提供的功能，这个快照也是供下一步分析使用)，找出实例个数增长的前十位的类，记录下这十个类的名称和当前实例的个数。在记录完数据后，点击 Profiler 中右上角的 Mark 按钮，将该点的状态作为下一次记录数据时的比较点。

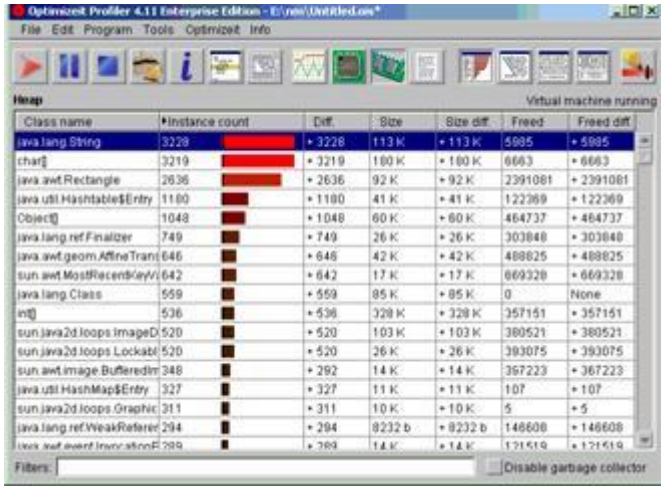


图 4 Profiler 堆视图

系统运行二十四小时以后可以得到四个内存快照。对这四个内存快照进行综合分析，如果每一次快照的内存使用都比上一次有增长，可以认定系统存在内存泄漏，找出在四个快照中实例个数都保持增长的类，这些类可以初步被认定为存在泄漏。

2.6.4 分析与定位

通过上面的数据收集和初步分析，可以得出初步结论:系统是否存在内存泄漏和哪些对象存在泄漏(被泄漏)，如果结论是存在泄漏，就可以进入分析和定位阶段了。

前面已经谈到 Java 中的内存泄漏就是无意识的对象保持，简单地讲就是因为编码的错误导致了一条本来不应该存在的引用链的存在(从而导致了被引用的对象无法释放)，因此内存泄漏分析的任务就是找出这条多余的引用链，并找到其形成的原因。前面还讲到过牵引对象，包括已经加载的类的静态变量和处于活动线程的堆栈空间的变量。由于活动线程的堆栈空间是迅速变化的，处于堆栈空间内的牵引对象集合是迅速变化的，而作为类的静态变量的牵引对象的集合在系统运行期间是相对稳定的。

对每个被泄漏的实例对象，必然存在一条从某个牵引对象出发到达该对象的引用链。处于堆栈空间的牵引对象在被从栈中弹出后就失去其牵引的能力，变为非牵引对象，因此，在长时间的运行后，被泄露的对象基本上都是被作为类的静态变量的牵引对象牵引。

Profiler 的内存视图除了堆视图以外，还包括实例分配视图(图 5)和实例引用图(图 6)。

Profiler 的实例引用图为找出从牵引对象到泄漏对象的引用链提供了非常直接的方法，其界面的第二个栏目中显示的就是从泄漏对象出发的逆向引用链。需要注意的是，当一个类的实例存在泄漏时，并非其所有的实例都是被泄漏的，往往只有一部分是被泄漏对象，其它则是正常使用的对象，要判断哪些是正常的引用链，哪些是不正常的引用链(引起泄漏的引用链)。通过抽取多个实例进行引用图的分析统计以后，可以找出一条或者多条从牵引对象出发的引用链，下面的任务就是找出这条引用链形成的原因。

实例分配图提供的功能是对每个类的实例的分配位置进行统计，查看实例分配的统计结果对于分析引用链的形成具有一定的作用，因为找到分配链与引用链的交点往往就可以找到了引用链形成的原因，下面将具体介绍。

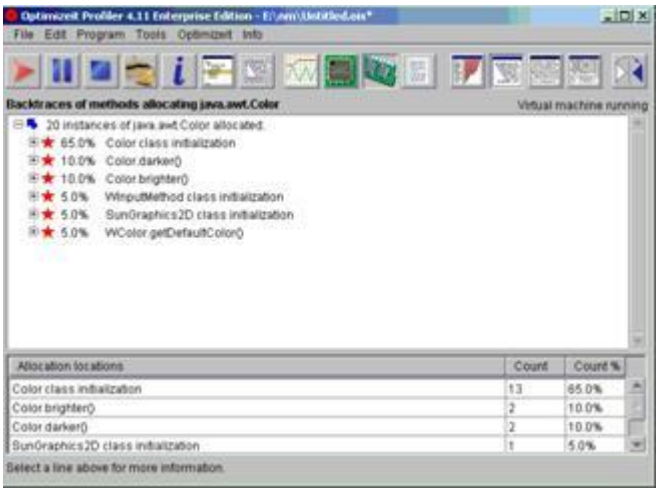


图 5 实例分配图

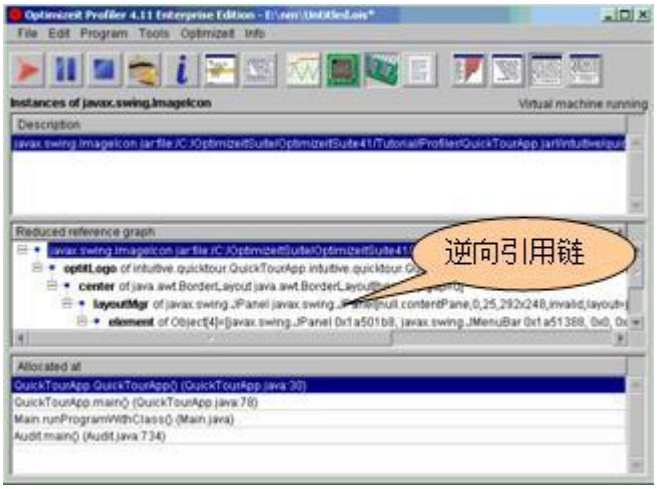


图 6 实例引用图

设想一个实例对象 a 在方法 f 中被分配，最终被实例对象 b 所引用，下面来分析从 b 到 a 的引用链可能的形成原因。方法 f 在创建对象 a 后，对它的使用分为四种情况:1、将 a 作为返回值返回;2、将 a 作为参数调用其它方法;3、在方法内部将 a 的引用传递给其它对象;4、其它情况。其中情况 4 不会造成由 b 到 a 的引用链的生成，不用考虑。下面考虑其它三种情况:对于 1、2 两种情况，其造成的结果都是在另一个方法内部获得了对象 a 的引用，它的分析与方法 f 的分析完全一样(递归分析);考虑第 3 种情况:1、假设方法 f 直接将对象 a 的引用加入到对象 b，则对象 b 到 a 的引用链就找到了，分析结束;2、假设方法 f 将对象 a 的引用加入到对象 c，则接下来就需要跟踪对象 c 的使用，对象 c 的分析比对象 a 的分析步骤更多一些，但大体原理都是一样的，就是跟踪对象从创建后被使用的历程，最终找到其被牵引对象引用的原因。

现在将泄漏对象的引用链以及引用链形成的原因找到了，内存泄漏测试与分析的工作就到此结束，接下来的工作就是修改相应的设计或者实现中的错误了。

2.6.5 总结

使用上述的测试和分析方法，在实践中先后进行了三次测试，找出了好几处内存泄漏错误。系统的稳定性得到很大程度的提高，最初运行 1~2 天就抛出内存溢出异常，修改完成后，系统从未出现过内存溢出异常。此方法适用于任何使用 Java 语言开发的、对稳定性有比较高要求的软件系统。

【其他】

3.1 提交 BUG

如果你发现文档中有不妥的地方，请发邮件至eoeandroid@eoemobile.com 进行反馈，我们会定期更新、发布更新后的版本。

3.2 关于 eoeAndroid

eoeAndroid 是国内成立最早，规模最大的Android 开发者社区，拥有海量的Android 学习资料。分享、互助的氛围，让Android 开发者迅速成长，逐步从懵懂到了解，从入门到开发出属于自己的应用，eoeAndroid 为广大Android 开发者奠定坚实的技术基础。从初级到高级，从环境搭建到底层架构，eoeAndroid 社区为开发者精挑细选了丰富的学习资料和实例代码。让Android 开发者在社区中迅速的成长，并在此基础上开发出更多优秀的Android 应用。

北京易联致远无线技术有限公司

责任编辑： QUMIN

美术支持： 金明根　　**技术支持：** gaotong

中国最大的Android 开发者社区：www.eoeandroid.com

中国本土的Android 软件下载平台：www.eoemarket.com