登录 | 注册

Gabby

android learner

目录视图 摘要视图 RSS 订阅

个人资料

GabbyZang

访问： 477860次

积分： 6388

等级： BLOG 6

排名： 第3887名

原创： 73篇 转载： 637篇

译文： 0篇 评论： 6条

文章搜索

文章分类

Q_CAMERA	(144)
Q_TP	(20)
Q_DISPLAY	(25)
Q_WIFI	(66)
Q_AUDIO	(22)
Q_STORAGE	(23)
Q_BOOT	(5)
Q_USB	(17)
Q_SENSOR	(10)
Q_BATTERY	(11)
Q_MODEM	(5)
Q_BT	(3)
Q_GPS	(1)
MTK	(17)
M_CAMERA	(5)
M_TP	(2)
M_DISPLAY	(11)
M_SENSOR	(3)
M_CHARGING/FG/POWER	(3)
M_MEMORY	(1)
M_AUDIO	(3)
M_MODEM	(1)
M_WIFI	(0)
Multimedia	(18)
LINUX	(22)
LINUX_TIPS	(43)
ANDROID	(93)
DRIVER	(14)
APK	(25)

ION基本概念介绍和原理分析

2014-03-07 14:22 838人阅读 评论(0) 收藏 举报

分类：

Q_CAMERA（143） ANDROID（92）

作者同类文章 X

- ./cts-tradefed run error[REASON:java config is not ri…
- Linux下获取空闲内存和内存使用率的方法
- 正确计算linux系统内存使用率
- 内存调试
- Autobracketing

更多

图是为内存在不同用户态进程之间传递和访问提供了

图是为内存在不同用户态进程之间传递和访问提供了

ev/ion设备内单独的handle空间，方便之处如下：

fget/fput从struct file级别进行kref控制；

当不需要在用户态访问时，是不需要与struct file关联的，内核结构ion_handle/ion_buffer唯一的表征了该buffer，所以与struct file关联的工作是在ioctl(ion, ION_IOC_SHARE/ION_ION_MAP, &share)中完成并输出的，用于后续的mmap调用；或者该进程不需要mmap而是仅仅向别的进程binder transfer，这就实现了用户态进行buffer流转控制，而内核态完成buffer数据流转。

转自http://blog.csdn.net/kris_fei/article/details/8588661 &http://blog.csdn.net/kris_fei/article/details/8618587

考察平台：

chipset: MSM8X25Q

codebase: Android 4.1

ION概念：

ION是Google的下一代内存管理器，用来支持不同的内存分配机制，如CARVOUT(PMEM)，物理连续内存(kmalloc), 虚拟地址连续但物理不连续内存(vmalloc)， IOMMU等。

用户空间和内核空间都可以使用ION，用户空间是通过/dev/ion来创建client的。

说到client, 顺便看下ION相关比较重要的几个概念。

Heap: 用来表示内存分配的相关信息，包括id, type, name等。用struct ion_heap表示。

Client: Ion的使用者，用户空间和内核控件要使用ION的buffer,必须先创建一个client,一个client可以有多个buffer，用struct ion_buffer表示。

Handle: 将buffer该抽象出来，可以认为ION用handle来管理buffer，一般用户直接拿到的是handle,而不是buffer。用struct ion_handle表示。

heap类型：

由于ION可以使用多种memory分配机制，例如物理连续和不连续的，所以ION使用enum ion_heap_type表示。

[cpp]

```cpp
/**
 * enum ion_heap_types - list of all possible types of heaps
 * @ION_HEAP_TYPE_SYSTEM:    memory allocated via vmalloc
 * @ION_HEAP_TYPE_SYSTEM_CONTIG: memory allocated via kmalloc
 * @ION_HEAP_TYPE_CARVEOUT:  memory allocated from a prereserved
```

LINUX_PRO (6)
OS_Install_Tips (12)
C++ (60)
HR (9)
health (1)
tv&&music&&fun (9)
compile (9)
并发处理相关 (8)
const&mutable (2)
callback_func (1)
ipc (1)
camera_framework (9)
sp&wp (3)
selinux (1)
initrc (3)
cmd (2)
virtual_func (2)
stl (1)
log (2)
mediaplayer (1)
parcel (1)
property (2)
debug (1)
busybox (1)
bionic (1)
zygote (2)
surface (2)
jni (1)
笔试题目 (3)
平台设备总线模型 (1)
iic (0)
字符驱动 (1)

文章存档

2016年04月 (32)
2016年03月 (6)
2016年02月 (13)
2015年12月 (43)
2015年11月 (9)
展开

阅读排行

请教：过量喝咖啡对心脏 (4385)
C++和JAVA的区别 -- 给扌 (3934)
svn如何取消认证缓存设] (3792)
android TP (2739)
Android LCD (2562)
高通平台android开发总纟 (2280)
Android图形合成和显示扌 (2258)
一个离职员工对中兴的回 (2237)
高通QPST Download使F (2208)
android camera (2088)

推荐文章

* CSDN日报20170828——《4个方法快速打造你的阅读清单》
* Android检查更新下载安装
* 动手打造史上最简单的 Recycleview 侧滑菜单
* TCP网络通讯如何解决分包粘包问题
* 程序员的八重境界

```
 *               carveout heap, allocations are physically
 *               contiguous
 * @ION_HEAP_TYPE_IOMMU: IOMMU memory
 * @ION_HEAP_TYPE_CP:    memory allocated from a prereserved
 *              carveout heap, allocations are physically
 *              contiguous. Used for content protection.
 * @ION_HEAP_TYPE_DMA:          memory allocated via DMA API
 * @ION_HEAP_END:       helper for iterating over heaps
 */
enum ion_heap_type {
    ION_HEAP_TYPE_SYSTEM,
    ION_HEAP_TYPE_SYSTEM_CONTIG,
    ION_HEAP_TYPE_CARVEOUT,
    ION_HEAP_TYPE_IOMMU,
    ION_HEAP_TYPE_CP,
    ION_HEAP_TYPE_DMA,
    ION_HEAP_TYPE_CUSTOM, /* must be last so device specific heaps always
             are at the end of this enum */
    ION_NUM_HEAPS,
};
```

[cpp]

```
<span xmlns="http://www.w3.org/1999/xhtml" style="">/**
 * enum ion_heap_types - list of all possible types of heaps
 * @ION_HEAP_TYPE_SYSTEM:    memory allocated via vmalloc
 * @ION_HEAP_TYPE_SYSTEM_CONTIG: memory allocated via kmalloc
 * @ION_HEAP_TYPE_CARVEOUT:  memory allocated from a prereserved
 *               carveout heap, allocations are physically
 *               contiguous
 * @ION_HEAP_TYPE_IOMMU: IOMMU memory
 * @ION_HEAP_TYPE_CP:    memory allocated from a prereserved
 *              carveout heap, allocations are physically
 *              contiguous. Used for content protection.
 * @ION_HEAP_TYPE_DMA:          memory allocated via DMA API
 * @ION_HEAP_END:       helper for iterating over heaps
 */
enum ion_heap_type {
    ION_HEAP_TYPE_SYSTEM,
    ION_HEAP_TYPE_SYSTEM_CONTIG,
    ION_HEAP_TYPE_CARVEOUT,
    ION_HEAP_TYPE_IOMMU,
    ION_HEAP_TYPE_CP,
    ION_HEAP_TYPE_DMA,
    ION_HEAP_TYPE_CUSTOM, /* must be last so device specific heaps always
             are at the end of this enum */
    ION_NUM_HEAPS,
};</span>
```

代码中的注释很明确地说明了哪种type对应的是分配哪种memory。不同type的heap需要不同的method去分配，不过都是用struct ion_heap_ops来表示的。如以下例子：

[cpp]

```
static struct ion_heap_ops carveout_heap_ops = {
    .allocate = ion_carveout_heap_allocate,
    .free = ion_carveout_heap_free,
    .phys = ion_carveout_heap_phys,
    .map_user = ion_carveout_heap_map_user,
    .map_kernel = ion_carveout_heap_map_kernel,
    .unmap_user = ion_carveout_heap_unmap_user,
    .unmap_kernel = ion_carveout_heap_unmap_kernel,
    .map_dma = ion_carveout_heap_map_dma,
    .unmap_dma = ion_carveout_heap_unmap_dma,
    .cache_op = ion_carveout_cache_ops,
    .print_debug = ion_carveout_print_debug,
    .map_iommu = ion_carveout_heap_map_iommu,
    .unmap_iommu = ion_carveout_heap_unmap_iommu,
};

static struct ion_heap_ops kmalloc_ops = {
    .allocate = ion_system_contig_heap_allocate,
    .free = ion_system_contig_heap_free,
    .phys = ion_system_contig_heap_phys,
    .map_dma = ion_system_contig_heap_map_dma,
    .unmap_dma = ion_system_heap_unmap_dma,
    .map_kernel = ion_system_heap_map_kernel,
```

关闭

* 四大线程池详解

```
    .unmap_kernel = ion_system_heap_unmap_kernel,
    .map_user = ion_system_contig_heap_map_user,
    .cache_op = ion_system_contig_heap_cache_ops,
    .print_debug = ion_system_contig_print_debug,
    .map_iommu = ion_system_contig_heap_map_iommu,
    .unmap_iommu = ion_system_heap_unmap_iommu,
};
```

[cpp]

```
<span xmlns="http://www.w3.org/1999/xhtml" style="">static struct ion_heap_ops carveout_heap_ops = {
    .allocate = ion_carveout_heap_allocate,
    .free = ion_carveout_heap_free,
    .phys = ion_carveout_heap_phys,
    .map_user = ion_carveout_heap_map_user,
    .map_kernel = ion_carveout_heap_map_kernel,
    .unmap_user = ion_carveout_heap_unmap_user,
    .unmap_kernel = ion_carveout_heap_unmap_kernel,
    .map_dma = ion_carveout_heap_map_dma,
    .unmap_dma = ion_carveout_heap_unmap_dma,
    .cache_op = ion_carveout_cache_ops,
    .print_debug = ion_carveout_print_debug,
    .map_iommu = ion_carveout_heap_map_iommu,
    .unmap_iommu = ion_carveout_heap_unmap_iommu,
};

static struct ion_heap_ops kmalloc_ops = {
    .allocate = ion_system_contig_heap_allocate,
    .free = ion_system_contig_heap_free,
    .phys = ion_system_contig_heap_phys,
    .map_dma = ion_system_contig_heap_map_dma,
    .unmap_dma = ion_system_heap_unmap_dma,
    .map_kernel = ion_system_heap_map_kernel,
    .unmap_kernel = ion_system_heap_unmap_kernel,
    .map_user = ion_system_contig_heap_map_user,
    .cache_op = ion_system_contig_heap_cache_ops,
    .print_debug = ion_system_contig_print_debug,
    .map_iommu = ion_system_contig_heap_map_iommu,
    .unmap_iommu = ion_system_heap_unmap_iommu,
};</span>
```

Heap ID：

同一种type的heap上当然可以分为若该干个chunk供用户使用，所以ION又使用ID来区分了。例如在type为ION_HEAP_TYPE_CARVEOUT的heap上，audio和display部分都需要使用，ION就用ID来区分。

Heap id用enumion_heap_ids表示。

[cpp]

```
/**
 * These are the only ids that should be used for Ion heap ids.
 * The ids listed are the order in which allocation will be attempted
 * if specified. Don't swap the order of heap ids unless you know what
 * you are doing!
 * Id's are spaced by purpose to allow new Id's to be inserted in-between (for
 * possible fallbacks)
 */

enum ion_heap_ids {
    INVALID_HEAP_ID = -1,
    ION_CP_MM_HEAP_ID = 8,
    ION_CP_MFC_HEAP_ID = 12,
    ION_CP_WB_HEAP_ID = 16, /* 8660 only */
    ION_CAMERA_HEAP_ID = 20, /* 8660 only */
    ION_SF_HEAP_ID = 24,
    ION_IOMMU_HEAP_ID = 25,
    ION_QSECOM_HEAP_ID = 26,
    ION_AUDIO_HEAP_BL_ID = 27,
    ION_AUDIO_HEAP_ID = 28,

    ION_MM_FIRMWARE_HEAP_ID = 29,
    ION_SYSTEM_HEAP_ID = 30,

    ION_HEAP_ID_RESERVED = 31 /** Bit reserved for ION_SECURE flag */
};
```

关闭

[cpp]

```cpp
<span xmlns="http://www.w3.org/1999/xhtml" style="">/**
 * These are the only ids that should be used for Ion heap ids.
 * The ids listed are the order in which allocation will be attempted
 * if specified. Don't swap the order of heap ids unless you know what
 * you are doing!
 * Id's are spaced by purpose to allow new Id's to be inserted in-between (for
 * possible fallbacks)
 */

enum ion_heap_ids {
    INVALID_HEAP_ID = -1,
    ION_CP_MM_HEAP_ID = 8,
    ION_CP_MFC_HEAP_ID = 12,
    ION_CP_WB_HEAP_ID = 16, /* 8660 only */
    ION_CAMERA_HEAP_ID = 20, /* 8660 only */
    ION_SF_HEAP_ID = 24,
    ION_IOMMU_HEAP_ID = 25,
    ION_QSECOM_HEAP_ID = 26,
    ION_AUDIO_HEAP_BL_ID = 27,
    ION_AUDIO_HEAP_ID = 28,

    ION_MM_FIRMWARE_HEAP_ID = 29,
    ION_SYSTEM_HEAP_ID = 30,

    ION_HEAP_ID_RESERVED = 31 /** Bit reserved for ION_SECURE flag */
};</span>
```

Heap 定义：

了解了heaptype和id，看看如何被用到了，本平台使用的文件为board-qrd7627a.c，有如下定义：

[cpp]

```cpp
/**
 * These heaps are listed in the order they will be allocated.
 * Don't swap the order unless you know what you are doing!
 */
struct ion_platform_heap msm7627a_heaps[] = {
        {
            .id = ION_SYSTEM_HEAP_ID,
            .type   = ION_HEAP_TYPE_SYSTEM,
            .name   = ION_VMALLOC_HEAP_NAME,
        },
#ifdef CONFIG_MSM_MULTIMEDIA_USE_ION
        /* PMEM_ADSP = CAMERA */
        {
            .id = ION_CAMERA_HEAP_ID,
            .type   = CAMERA_HEAP_TYPE,
            .name   = ION_CAMERA_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_mm_ion_pdata,
            .priv   = (void *)&ion_cma_device.dev,
        },
        /* AUDIO HEAP 1*/
        {
            .id = ION_AUDIO_HEAP_ID,
            .type   = ION_HEAP_TYPE_CARVEOUT,
            .name   = ION_AUDIO_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
        },
        /* PMEM_MDP = SF */
        {
            .id = ION_SF_HEAP_ID,
            .type   = ION_HEAP_TYPE_CARVEOUT,
            .name   = ION_SF_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
        },
        /* AUDIO HEAP 2*/
        {
            .id    = ION_AUDIO_HEAP_BL_ID,
            .type  = ION_HEAP_TYPE_CARVEOUT,
            .name  = ION_AUDIO_BL_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
```

关闭

```
            .base = BOOTLOADER_BASE_ADDR,
        },

#endif
};
```

[cpp]

```
<span xmlns="http://www.w3.org/1999/xhtml" style="">/**
 * These heaps are listed in the order they will be allocated.
 * Don't swap the order unless you know what you are doing!
 */
struct ion_platform_heap msm7627a_heaps[] = {
        {
            .id = ION_SYSTEM_HEAP_ID,
            .type   = ION_HEAP_TYPE_SYSTEM,
            .name   = ION_VMALLOC_HEAP_NAME,
        },
#ifdef CONFIG_MSM_MULTIMEDIA_USE_ION
        /* PMEM_ADSP = CAMERA */
        {
            .id = ION_CAMERA_HEAP_ID,
            .type   = CAMERA_HEAP_TYPE,
            .name   = ION_CAMERA_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_mm_ion_pdata,
            .priv   = (void *)&ion_cma_device.dev,
        },
        /* AUDIO HEAP 1*/
        {
            .id = ION_AUDIO_HEAP_ID,
            .type   = ION_HEAP_TYPE_CARVEOUT,
            .name   = ION_AUDIO_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
        },
        /* PMEM_MDP = SF */
        {
            .id = ION_SF_HEAP_ID,
            .type   = ION_HEAP_TYPE_CARVEOUT,
            .name   = ION_SF_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
        },
        /* AUDIO HEAP 2*/
        {
            .id    = ION_AUDIO_HEAP_BL_ID,
            .type  = ION_HEAP_TYPE_CARVEOUT,
            .name  = ION_AUDIO_BL_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
            .base = BOOTLOADER_BASE_ADDR,
        },

#endif
};</span>
```

ION Handle：

当Ion client分配buffer时，相应的一个唯一的handle也会被指定，当然client可以多次申请ion buffer。申请好buffer之后，返回的是一个ion handle, 不过要知道Ion buffer才和实际的内存相关，包括size, address等信息。Struct ion_handle和struct ion_buffer如下：

[cpp]

```
/**
 * ion_handle - a client local reference to a buffer
 * @ref:        reference count
 * @client:     back pointer to the client the buffer resides in
 * @buffer:     pointer to the buffer
 * @node:       node in the client's handle rbtree
 * @kmap_cnt:       count of times this client has mapped to kernel
 * @dmap_cnt:       count of times this client has mapped for dma
 *
 * Modifications to node, map_cnt or mapping should be protected by the
 * lock in the client.  Other fields are never changed after initialization.
 */
struct ion_handle {
    struct kref ref;
    struct ion_client *client;
    struct ion_buffer *buffer;
    struct rb_node node;
    unsigned int kmap_cnt;
    unsigned int iommu_map_cnt;
};

/**
 * struct ion_buffer - metadata for a particular buffer
 * @ref:        refernce count
 * @node:       node in the ion_device buffers tree
 * @dev:        back pointer to the ion_device
 * @heap:       back pointer to the heap the buffer came from
 * @flags:      buffer specific flags
 * @size:       size of the buffer
 * @priv_virt:      private data to the buffer representable as
 *          a void *
 * @priv_phys:      private data to the buffer representable as
 *          an ion_phys_addr_t (and someday a phys_addr_t)
 * @lock:       protects the buffers cnt fields
 * @kmap_cnt:          number of times the buffer is mapped to the kernel
 * @vaddr:      the kenrel mapping if kmap_cnt is not zero
 * @dmap_cnt:       number of times the buffer is mapped for dma
 * @sg_table:       the sg table for the buffer if dmap_cnt is not zero
*/
struct ion_buffer {
    struct kref ref;
    struct rb_node node;
    struct ion_device *dev;
    struct ion_heap *heap;
    unsigned long flags;
    size_t size;
    union {
        void *priv_virt;
        ion_phys_addr_t priv_phys;
    };
    struct mutex lock;
    int kmap_cnt;
    void *vaddr;
    int dmap_cnt;
    struct sg_table *sg_table;
    int umap_cnt;
    unsigned int iommu_map_cnt;
    struct rb_root iommu_maps;
    int marked;
};
```

[cpp]

```
<span xmlns="http://www.w3.org/1999/xhtml" style="">/**
 * ion_handle - a client local reference to a buffer
 * @ref:        reference count
 * @client:     back pointer to the client the buffer resides in
 * @buffer:     pointer to the buffer
 * @node:       node in the client's handle rbtree
 * @kmap_cnt:       count of times this client has mapped to kernel
 * @dmap_cnt:       count of times this client has mapped for dma
 *
 * Modifications to node, map_cnt or mapping should be protected by the
 * lock in the client.  Other fields are never changed after initialization.
 */
struct ion_handle {
    struct kref ref;
    struct ion_client *client;
    struct ion_buffer *buffer;
    struct rb_node node;
    unsigned int kmap_cnt;
    unsigned int iommu_map_cnt;
};

/**
 * struct ion_buffer - metadata for a particular buffer
 * @ref:        refernce count
 * @node:       node in the ion_device buffers tree
 * @dev:        back pointer to the ion_device
 * @heap:       back pointer to the heap the buffer came from
 * @flags:      buffer specific flags
 * @size:       size of the buffer
 * @priv_virt:      private data to the buffer representable as
```

关闭

```
 *          a void *
 * @priv_phys:      private data to the buffer representable as
 *          an ion_phys_addr_t (and someday a phys_addr_t)
 * @lock:       protects the buffers cnt fields
 * @kmap_cnt:       number of times the buffer is mapped to the kernel
 * @vaddr:      the kenrel mapping if kmap_cnt is not zero
 * @dmap_cnt:       number of times the buffer is mapped for dma
 * @sg_table:       the sg table for the buffer if dmap_cnt is not zero
*/
struct ion_buffer {
    struct kref ref;
    struct rb_node node;
    struct ion_device *dev;
    struct ion_heap *heap;
    unsigned long flags;
    size_t size;
    union {
        void *priv_virt;
        ion_phys_addr_t priv_phys;
    };
    struct mutex lock;
    int kmap_cnt;
    void *vaddr;
    int dmap_cnt;
    struct sg_table *sg_table;
    int umap_cnt;
    unsigned int iommu_map_cnt;
    struct rb_root iommu_maps;
    int marked;
};</span>
```

ION Client：

用户空间和内核空间都可以成为client，不过创建的方法稍稍有点区别，先了解下基本的操作流程吧。

内核空间:

先创建client:

[cpp]

```
struct ion_client *ion_client_create(struct ion_device *dev,
                     unsigned int heap_mask,
                     const char *name)
```

[cpp]

```
<span xmlns="http://www.w3.org/1999/xhtml" style="">struct ion_client *ion_client_create(struct ion_device *dev,
                     unsigned int heap_mask,
                     const char *name)</span>
```

heap_mask: 可以分配的heap type，如carveout,system heap, iommu等。

高通使用msm_ion_client_create函数封装了下。

有了client之后就可以分配内存：

[cpp]

```
struct ion_handle *ion_alloc(struct ion_client *client, size_t len,
                 size_t align, unsigned int flags)
```

[cpp]

```
<span xmlns="http://www.w3.org/1999/xhtml" style="">struct ion_handle *ion_alloc(st
                 size_t align, unsigned int flags)</span>
```

flags: 分配的heap id.

有了handle也就是buffer之后就准备使用了，不过还是物理地址，需要map：

[cpp]

```
void *ion_map_kernel(struct ion_client *client, struct ion_handle *handle,
            unsigned long flags)
```

[cpp]

```
<span xmlns="http://www.w3.org
```

```
/1999/xhtml" style="">void *ion_map_kernel(struct ion_client *client, struct ion_handle
            unsigned long flags)</span>
```

用户空间:

用户空间如果想使用ION，也必须先要创建client,不过它是打开/dev/ion,实际上它最终也会调用ion_client_create。

不过和内核空间创建client的一点区别是，用户空间不能选择heap type（使用预订的heap id隐含heap type），但是内核空间却可以。

另外，用户空间是通过IOCTL来分配内存的，cmd为ION_IOC_ALLOC.

[cpp]

```
ion_fd = open("/dev/ion", O_ RDONLY | O_SYNC);
ioctl(ion_fd, ION_IOC_ALLOC, alloc);
```

[cpp]

```
<span xmlns="http://www.w3.org/1999/xhtml" style="">ion_fd = open("
/dev/ion", O_ RDONLY | O_SYNC);
ioctl(ion_fd, ION_IOC_ALLOC, alloc); </span>
```

alloc为struct ion_allocation_data,len是申请buffer的长度，flags是heap id。

[cpp]

```
/**
 * struct ion_allocation_data - metadata passed from userspace for allocations
 * @len:    size of the allocation
 * @align:  required alignment of the allocation
 * @flags:  flags passed to heap
 * @handle: pointer that will be populated with a cookie to use to refer
 *      to this allocation
 *
 * Provided by userspace as an argument to the ioctl
 */
struct ion_allocation_data {
    size_t len;
    size_t align;
    unsigned int flags;
    struct ion_handle *handle;
};
```

[cpp]

```
<span xmlns="http://www.w3.org/1999/xhtml" style="">/**
 * struct ion_allocation_data - metadata passed from userspace for allocations
 * @len:    size of the allocation
 * @align:  required alignment of the allocation
 * @flags:  flags passed to heap
 * @handle: pointer that will be populated with a cookie to use to refer
 *      to this allocation
 *
 * Provided by userspace as an argument to the ioctl
 */
struct ion_allocation_data {
    size_t len;
    size_t align;
    unsigned int flags;
    struct ion_handle *handle;
};</span>
```

关闭

分配好了buffer之后，如果用户空间想使用buffer，先需要mmap. ION是通过先调用IOCTL中的ION_IOC_SHARE/ION_IOC_MAP来得到可以mmap的fd,然后再执行mmap得到bufferaddress.

然后，你也可以将此fd传给另一个进程，如通过binder传递。在另一个进程中通过ION_IOC_IMPORT这个IOCTL来得到这块共享buffer了。

来看一个例子：

[cpp]

```
进程A：
int ionfd = open("/dev/ion", O_RDONLY | O_DSYNC);
alloc_data.len = 0x1000;
alloc_data.align = 0x1000;
alloc_data.flags = ION_HEAP(ION_CP_MM_HEAP_ID);
rc = ioctl(ionfd,ION_IOC_ALLOC, &alloc_data);
fd_data.handle = alloc_data.handle;
rc = ioctl(ionfd,ION_IOC_SHARE,&fd_data);
shared_fd = fd_data.fd;

进程B：
fd_data.fd = shared_fd;
rc = ioctl(ionfd,ION_IOC_IMPORT,&fd_data);
```

[cpp]

```cpp
<span xmlns="http://www.w3.org/1999/xhtml" style="">进程A：
int ionfd = open("/dev/ion", O_RDONLY | O_DSYNC);
alloc_data.len = 0x1000;
alloc_data.align = 0x1000;
alloc_data.flags = ION_HEAP(ION_CP_MM_HEAP_ID);
rc = ioctl(ionfd,ION_IOC_ALLOC, &alloc_data);
fd_data.handle = alloc_data.handle;
rc = ioctl(ionfd,ION_IOC_SHARE,&fd_data);
shared_fd = fd_data.fd;

进程B：
fd_data.fd = shared_fd;
rc = ioctl(ionfd,ION_IOC_IMPORT,&fd_data); </span>
```

从上一篇ION基本概念中，我们了解了heaptype, heap id, client, handle以及如何使用，本篇再从原理上分析下ION的运作流程。

MSM8x25Q平台使用的是board-qrd7627.c，ION相关定义如下：

[cpp]

```cpp
/**
 * These heaps are listed in the order they will be allocated.
 * Don't swap the order unless you know what you are doing!
 */
struct ion_platform_heap msm7627a_heaps[] = {
        {
            .id = ION_SYSTEM_HEAP_ID,
            .type   = ION_HEAP_TYPE_SYSTEM,
            .name   = ION_VMALLOC_HEAP_NAME,
        },
#ifdef CONFIG_MSM_MULTIMEDIA_USE_ION
        /* PMEM_ADSP = CAMERA */
        {
            .id = ION_CAMERA_HEAP_ID,
            .type   = CAMERA_HEAP_TYPE,
            .name   = ION_CAMERA_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_mm_ion_pdata,
            .priv   = (void *)&ion_cma_device.dev,
        },
        /* AUDIO HEAP 1*/
        {
            .id = ION_AUDIO_HEAP_ID,
            .type   = ION_HEAP_TYPE_CARVEOUT,
            .name   = ION_AUDIO_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
        },
        /* PMEM_MDP = SF */
        {
            .id = ION_SF_HEAP_ID,
            .type   = ION_HEAP_TYPE_CARVEOUT,
            .name   = ION_SF_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
        },
        /* AUDIO HEAP 2*/
        {
            .id    = ION_AUDIO_HEAP_BL_ID,
            .type  = ION_HEAP_TYPE_CARVEOUT,
            .name  = ION_AUDIO_BL_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
            .base = BOOTLOADER_BASE_ADDR,
```

关闭

```
        },

#endif
};

static struct ion_co_heap_pdata co_ion_pdata = {
    .adjacent_mem_id = INVALID_HEAP_ID,
    .align = PAGE_SIZE,
};

static struct ion_co_heap_pdata co_mm_ion_pdata = {
    .adjacent_mem_id = INVALID_HEAP_ID,
    .align = PAGE_SIZE,
};

static u64 msm_dmamask = DMA_BIT_MASK(32);

static struct platform_device ion_cma_device = {
    .name = "ion-cma-device",
    .id = -1,
    .dev = {
        .dma_mask = &msm_dmamask,
        .coherent_dma_mask = DMA_BIT_MASK(32),
    }
};
```

[cpp]

```
<span xmlns="http://www.w3.org/1999/xhtml" style="">/**
 * These heaps are listed in the order they will be allocated.
 * Don't swap the order unless you know what you are doing!
 */
struct ion_platform_heap msm7627a_heaps[] = {
        {
            .id = ION_SYSTEM_HEAP_ID,
            .type   = ION_HEAP_TYPE_SYSTEM,
            .name   = ION_VMALLOC_HEAP_NAME,
        },
#ifdef CONFIG_MSM_MULTIMEDIA_USE_ION
        /* PMEM_ADSP = CAMERA */
        {
            .id = ION_CAMERA_HEAP_ID,
            .type   = CAMERA_HEAP_TYPE,
            .name   = ION_CAMERA_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_mm_ion_pdata,
            .priv   = (void *)&ion_cma_device.dev,
        },
        /* AUDIO HEAP 1*/
        {
            .id = ION_AUDIO_HEAP_ID,
            .type   = ION_HEAP_TYPE_CARVEOUT,
            .name   = ION_AUDIO_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
        },
        /* PMEM_MDP = SF */
        {
            .id = ION_SF_HEAP_ID,
            .type   = ION_HEAP_TYPE_CARVEOUT,
            .name   = ION_SF_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
        },
        /* AUDIO HEAP 2*/
        {
            .id    = ION_AUDIO_HEAP_BL_ID,
            .type  = ION_HEAP_TYPE_CARVEOUT,
            .name  = ION_AUDIO_BL_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
            .base = BOOTLOADER_BASE_ADDR,
        },

#endif
};

static struct ion_co_heap_pdata co_ion_pdata = {
    .adjacent_mem_id = INVALID_HEAP_ID,
    .align = PAGE_SIZE,
```

关闭

```
};

static struct ion_co_heap_pdata co_mm_ion_pdata = {
    .adjacent_mem_id = INVALID_HEAP_ID,
    .align = PAGE_SIZE,
};

static u64 msm_dmamask = DMA_BIT_MASK(32);

static struct platform_device ion_cma_device = {
    .name = "ion-cma-device",
    .id = -1,
    .dev = {
        .dma_mask = &msm_dmamask,
        .coherent_dma_mask = DMA_BIT_MASK(32),
    }
};</span>
```

Qualcomm提示了不要轻易调换顺序，因为后面代码处理是将顺序定死了的，一旦你调换了，代码就无法正常运行了。
另外， 本系统中只使用了ION_HEAP_TYPE_CARVEOUT和 ION_HEAP_TYPE_SYSTEM这两种h
对于ION_HEAP_TYPE_CARVEOUT的内存分配，后面将会发现，其实就是之前讲述过的使用mem pool来分配的。
Platform device如下,在msm_ion.c中用到。

[cpp]

```
static struct ion_platform_data ion_pdata = {
    .nr = MSM_ION_HEAP_NUM,
    .has_outer_cache = 1,
    .heaps = msm7627a_heaps,
};

static struct platform_device ion_dev = {
    .name = "ion-msm",
    .id = 1,
    .dev = { .platform_data = &ion_pdata },
};
```

[cpp]

```
<span xmlns="http://www.w3.org/1999/xhtml" style="">static struct ion_platform_data ion_pdata = {
    .nr = MSM_ION_HEAP_NUM,
    .has_outer_cache = 1,
    .heaps = msm7627a_heaps,
};

static struct platform_device ion_dev = {
    .name = "ion-msm",
    .id = 1,
    .dev = { .platform_data = &ion_pdata },
};</span>
```

ION初始化

转到msm_ion.c，ion.c的某些函数也被重新封装了下.万事都从设备匹配开始：

[cpp]

```
static struct platform_driver msm_ion_driver = {
    .probe = msm_ion_probe,
    .remove = msm_ion_remove,
    .driver = { .name = "ion-msm" }
};
static int __init msm_ion_init(void)
{
    /*调用msm_ion_probe */
    return platform_driver_register(&msm_ion_driver);
}

static int msm_ion_probe(struct platform_device *pdev)
{
    /*即board-qrd7627a.c中的ion_pdata */
    struct ion_platform_data *pdata = pdev->dev.platform_data;
    int err;
```

关闭

```
    int i;

    /*heap数量*/
    num_heaps = pdata->nr;
    /*分配struct ion_heap */
    heaps = kcalloc(pdata->nr, sizeof(struct ion_heap *), GFP_KERNEL);

    if (!heaps) {
        err = -ENOMEM;
        goto out;
    }
    /*创建节点，最终是/dev/ion,供用户空间操作。*/
    idev = ion_device_create(NULL);
    if (IS_ERR_OR_NULL(idev)) {
        err = PTR_ERR(idev);
        goto freeheaps;
    }
    /*最终是根据adjacent_mem_id 是否定义了来分配相邻内存，
我们没用到，忽略此函数。*/
    msm_ion_heap_fixup(pdata->heaps, num_heaps);

    /* create the heaps as specified in the board file */
    for (i = 0; i < num_heaps; i++) {
        struct ion_platform_heap *heap_data = &pdata->heaps[i];
        /*分配ion*/
        msm_ion_allocate(heap_data);

        heap_data->has_outer_cache = pdata->has_outer_cache;
        /*创建ion heap。*/
        heaps[i] = ion_heap_create(heap_data);
        if (IS_ERR_OR_NULL(heaps[i])) {
            heaps[i] = 0;
            continue;
        } else {
            if (heap_data->size)
                pr_info("ION heap %s created at %lx "
                    "with size %x\n", heap_data->name,
                              heap_data->base,
                              heap_data->size);
            else
                pr_info("ION heap %s created\n",
                              heap_data->name);
        }
        /*创建的heap添加到idev中，以便后续使用。*/
        ion_device_add_heap(idev, heaps[i]);
    }
    /*检查heap之间是否有重叠部分*/
    check_for_heap_overlap(pdata->heaps, num_heaps);
    platform_set_drvdata(pdev, idev);
    return 0;

freeheaps:
    kfree(heaps);
out:
    return err;
}

通过ion_device_create创建/dev/ion节点：
struct ion_device *ion_device_create(long (*custom_ioctl)
                     (struct ion_client *client,
                      unsigned int cmd,
                      unsigned long arg))
{
    struct ion_device *idev;
    int ret;

    idev = kzalloc(sizeof(struct ion_device), GFP_KERNEL);
    if (!idev)
        return ERR_PTR(-ENOMEM);
    /*是个misc设备*/
    idev->dev.minor = MISC_DYNAMIC_MINOR;
    /*节点名字为ion*/
    idev->dev.name = "ion";
    /*fops为ion_fops,所以对应ion的操作都会调用ion_fops的函数指针。*/
    idev->dev.fops = &ion_fops;
    idev->dev.parent = NULL;
    ret = misc_register(&idev->dev);
    if (ret) {
        pr_err("ion: failed to register misc device.\n");
        return ERR_PTR(ret);
```

关闭

```
    }
    /*创建debugfs目录，路径为/sys/kernel/debug/ion/*/
    idev->debug_root = debugfs_create_dir("ion", NULL);
    if (IS_ERR_OR_NULL(idev->debug_root))
        pr_err("ion: failed to create debug files.\n");

    idev->custom_ioctl = custom_ioctl;
    idev->buffers = RB_ROOT;
    mutex_init(&idev->lock);
    idev->heaps = RB_ROOT;
    idev->clients = RB_ROOT;
    /*在ion目录下创建一个check_leaked_fds文件，用来检查Ion的使用是否有内存泄漏。如果申请了ion之后
不需要使用却没有释放，就会导致memory leak.*/
    debugfs_create_file("check_leaked_fds", 0664, idev->debug_root, idev,
                &debug_leak_fops);
    return idev;
}

msm_ion_allocate：
static void msm_ion_allocate(struct ion_platform_heap *heap)
{

    if (!heap->base && heap->extra_data) {
        unsigned int align = 0;
        switch (heap->type) {
        /*获取align参数*/
        case ION_HEAP_TYPE_CARVEOUT:
            align =
            ((struct ion_co_heap_pdata *) heap->extra_data)->align;
            break;
        /*此type我们没使用到。*/
        case ION_HEAP_TYPE_CP:
        {
            struct ion_cp_heap_pdata *data =
                (struct ion_cp_heap_pdata *)
                heap->extra_data;
            if (data->reusable) {
                const struct fmem_data *fmem_info =
                    fmem_get_info();
                heap->base = fmem_info->phys;
                data->virt_addr = fmem_info->virt;
                pr_info("ION heap %s using FMEM\n", heap->name);
            } else if (data->mem_is_fmem) {
                const struct fmem_data *fmem_info =
                    fmem_get_info();
                heap->base = fmem_info->phys + fmem_info->size;
            }
            align = data->align;
            break;
        }
        default:
            break;
        }
        if (align && !heap->base) {
            /*获取heap的base address。*/
            heap->base = msm_ion_get_base(heap->size,
                              heap->memory_type,
                              align);
            if (!heap->base)
                pr_err("%s: could not get memory for heap %s "
                   "(id %x)\n", __func__, heap->name, heap->id);
        }
    }
}

static unsigned long msm_ion_get_base(unsigned long s
                    unsigned int align)
{
    switch (memory_type) {
    /*我们定义的是ebi type，看见没，此函数在mem pool中分析过了。
原理就是使用Mempool 来管理分配内存。*/
    case ION_EBI_TYPE:
        return allocate_contiguous_ebi_nomap(size, align);
        break;
    case ION_SMI_TYPE:
        return allocate_contiguous_memory_nomap(size, MEMTYPE_SMI,
                            align);
        break;
    default:
        pr_err("%s: Unknown memory type %d\n", __func__, memory_type);
```

关闭

```
        return 0;
    }
}
ion_heap_create：
struct ion_heap *ion_heap_create(struct ion_platform_heap *heap_data)
{
    struct ion_heap *heap = NULL;
    /*根据Heap type调用相应的创建函数。*/
    switch (heap_data->type) {
    case ION_HEAP_TYPE_SYSTEM_CONTIG:
        heap = ion_system_contig_heap_create(heap_data);
        break;
    case ION_HEAP_TYPE_SYSTEM:
        heap = ion_system_heap_create(heap_data);
        break;
    case ION_HEAP_TYPE_CARVEOUT:
        heap = ion_carveout_heap_create(heap_data);
        break;
    case ION_HEAP_TYPE_IOMMU:
        heap = ion_iommu_heap_create(heap_data);
        break;
    case ION_HEAP_TYPE_CP:
        heap = ion_cp_heap_create(heap_data);
        break;
#ifdef CONFIG_CMA
    case ION_HEAP_TYPE_DMA:
        heap = ion_cma_heap_create(heap_data);
        break;
#endif
    default:
        pr_err("%s: Invalid heap type %d\n", __func__,
               heap_data->type);
        return ERR_PTR(-EINVAL);
    }

    if (IS_ERR_OR_NULL(heap)) {
        pr_err("%s: error creating heap %s type %d base %lu size %u\n",
               __func__, heap_data->name, heap_data->type,
               heap_data->base, heap_data->size);
        return ERR_PTR(-EINVAL);
    }
    /*保存Heap的name,id和私有数据。*/
    heap->name = heap_data->name;
    heap->id = heap_data->id;
    heap->priv = heap_data->priv;
    return heap;
}
```

[cpp]

```
<span xmlns="http://www.w3.org/1999/xhtml" style="">static struct platform_driver msm_ion_driver = {
    .probe = msm_ion_probe,
    .remove = msm_ion_remove,
    .driver = { .name = "ion-msm" }
};
static int __init msm_ion_init(void)
{
    /*调用msm_ion_probe */
    return platform_driver_register(&msm_ion_driver);
}

static int msm_ion_probe(struct platform_device *pdev)
{
    /*即board-qrd7627a.c中的ion_pdata */
    struct ion_platform_data *pdata = pdev->dev.plat
    int err;
    int i;

    /*heap数量*/
    num_heaps = pdata->nr;
    /*分配struct ion_heap */
    heaps = kcalloc(pdata->nr, sizeof(struct ion_heap *), GFP_KERNEL);

    if (!heaps) {
        err = -ENOMEM;
        goto out;
    }
    /*创建节点，最终是/dev/ion,供用户空间操作。*/
    idev = ion_device_create(NULL);
    if (IS_ERR_OR_NULL(idev)) {
        err = PTR_ERR(idev);
        goto freeheaps;
    }
    /*最终是根据adjacent_mem_id 是否定义了来分配相邻内存，
我们没用到，忽略此函数。*/
    msm_ion_heap_fixup(pdata->heaps, num_heaps);

    /* create the heaps as specified in the board file */
    for (i = 0; i < num_heaps; i++) {
        struct ion_platform_heap *heap_data = &pdata->heaps[i];
        /*分配ion*/
        msm_ion_allocate(heap_data);

        heap_data->has_outer_cache = pdata->has_outer_cache;
        /*创建ion heap。*/
        heaps[i] = ion_heap_create(heap_data);
        if (IS_ERR_OR_NULL(heaps[i])) {
            heaps[i] = 0;
            continue;
        } else {
            if (heap_data->size)
                pr_info("ION heap %s created at %lx "
                    "with size %x\n", heap_data->name,
                              heap_data->base,
                              heap_data->size);
            else
                pr_info("ION heap %s created\n",
                              heap_data->name);
        }
        /*创建的heap添加到idev中，以便后续使用。*/
        ion_device_add_heap(idev, heaps[i]);
    }
    /*检查heap之间是否有重叠部分*/
    check_for_heap_overlap(pdata->heaps, num_heaps);
    platform_set_drvdata(pdev, idev);
    return 0;

freeheaps:
    kfree(heaps);
out:
    return err;
}

通过ion_device_create创建/dev/ion节点：
struct ion_device *ion_device_create(long (*custom_ioctl)
                     (struct ion_client *client,
                      unsigned int cmd,
                      unsigned long arg))
{
    struct ion_device *idev;
    int ret;

    idev = kzalloc(sizeof(struct ion_device), GFP_KERNEL);
    if (!idev)
        return ERR_PTR(-ENOMEM);
    /*是个misc设备*/
    idev->dev.minor = MISC_DYNAMIC_MINOR;
    /*节点名字为ion*/
    idev->dev.name = "ion";
    /*fops为ion_fops,所以对应ion的操作都会调用ion_fops的函数指针。*/
    idev->dev.fops = &ion_fops;
    idev->dev.parent = NULL;
    ret = misc_register(&idev->dev);
    if (ret) {
        pr_err("ion: failed to register misc device.\
        return ERR_PTR(ret);
    }
    /*创建debugfs目录，路径为/sys/kernel/debug/ion/*/
    idev->debug_root = debugfs_create_dir("ion", NULL);
    if (IS_ERR_OR_NULL(idev->debug_root))
        pr_err("ion: failed to create debug files.\n");

    idev->custom_ioctl = custom_ioctl;
    idev->buffers = RB_ROOT;
    mutex_init(&idev->lock);
    idev->heaps = RB_ROOT;
    idev->clients = RB_ROOT;
    /*在ion目录下创建一个check_leaked_fds文件，用来检查Ion的使用是否有内存泄漏。如果申请了ion之后
不需要使用却没有释放，就会导致memory leak.*/
```

关闭

```
    debugfs_create_file("check_leaked_fds", 0664, idev->debug_root, idev,
                &debug_leak_fops);
    return idev;
}

msm_ion_allocate：
static void msm_ion_allocate(struct ion_platform_heap *heap)
{

    if (!heap->base && heap->extra_data) {
        unsigned int align = 0;
        switch (heap->type) {
        /*获取align参数*/
        case ION_HEAP_TYPE_CARVEOUT:
            align =
            ((struct ion_co_heap_pdata *) heap->extra_data)->align;
            break;
        /*此type我们没使用到。*/
        case ION_HEAP_TYPE_CP:
        {
            struct ion_cp_heap_pdata *data =
                (struct ion_cp_heap_pdata *)
                heap->extra_data;
            if (data->reusable) {
                const struct fmem_data *fmem_info =
                    fmem_get_info();
                heap->base = fmem_info->phys;
                data->virt_addr = fmem_info->virt;
                pr_info("ION heap %s using FMEM\n", heap->name);
            } else if (data->mem_is_fmem) {
                const struct fmem_data *fmem_info =
                    fmem_get_info();
                heap->base = fmem_info->phys + fmem_info->size;
            }
            align = data->align;
            break;
        }
        default:
            break;
        }
        if (align && !heap->base) {
            /*获取heap的base address。*/
            heap->base = msm_ion_get_base(heap->size,
                              heap->memory_type,
                              align);
            if (!heap->base)
                pr_err("%s: could not get memory for heap %s "
                   "(id %x)\n", __func__, heap->name, heap->id);
        }
    }
}

static unsigned long msm_ion_get_base(unsigned long size, int memory_type,
                    unsigned int align)
{
    switch (memory_type) {
    /*我们定义的是ebi type，看见没，此函数在mem pool中分析过了。
原理就是使用Mempool 来管理分配内存。*/
    case ION_EBI_TYPE:
        return allocate_contiguous_ebi_nomap(size, align);
        break;
    case ION_SMI_TYPE:
        return allocate_contiguous_memory_nomap(size, MEMTYPE_SMI,
                            align);
        break;
    default:
        pr_err("%s: Unknown memory type %d\n", __func__, memory_type);
        return 0;
    }
}
ion_heap_create：
struct ion_heap *ion_heap_create(struct ion_platform_heap *heap_data)
{
    struct ion_heap *heap = NULL;
    /*根据Heap type调用相应的创建函数。*/
    switch (heap_data->type) {
    case ION_HEAP_TYPE_SYSTEM_CONTIG:
        heap = ion_system_contig_heap_create(heap_data);
        break;
    case ION_HEAP_TYPE_SYSTEM:
```

关闭

```cpp
        heap = ion_system_heap_create(heap_data);
        break;
    case ION_HEAP_TYPE_CARVEOUT:
        heap = ion_carveout_heap_create(heap_data);
        break;
    case ION_HEAP_TYPE_IOMMU:
        heap = ion_iommu_heap_create(heap_data);
        break;
    case ION_HEAP_TYPE_CP:
        heap = ion_cp_heap_create(heap_data);
        break;
#ifdef CONFIG_CMA
    case ION_HEAP_TYPE_DMA:
        heap = ion_cma_heap_create(heap_data);
        break;
#endif
    default:
        pr_err("%s: Invalid heap type %d\n", __func__,
               heap_data->type);
        return ERR_PTR(-EINVAL);
    }

    if (IS_ERR_OR_NULL(heap)) {
        pr_err("%s: error creating heap %s type %d base %lu size %u\n",
               __func__, heap_data->name, heap_data->type,
               heap_data->base, heap_data->size);
        return ERR_PTR(-EINVAL);
    }
    /*保存Heap的name,id和私有数据。*/
    heap->name = heap_data->name;
    heap->id = heap_data->id;
    heap->priv = heap_data->priv;
    return heap;
}</span>
```

从下面的代码可以得知，ION_HEAP_TYPE_SYSTEM_CONTIG使用kmalloc创建的，ION_HEAP_TYPE_SYSTEM使用的是vmalloc,而ion_carveout_heap_create就是系统预分配了一片内存区域供其使用。Ion在申请使用的时候，会根据当前的type来操作各自的heap->ops。分别看下三个函数：

[cpp]

```cpp
struct ion_heap *ion_system_contig_heap_create(struct ion_platform_heap *pheap)
{
    struct ion_heap *heap;

    heap = kzalloc(sizeof(struct ion_heap), GFP_KERNEL);
    if (!heap)
        return ERR_PTR(-ENOMEM);
    /*使用的是kmalloc_ops，上篇有提到哦*/
    heap->ops = &kmalloc_ops;
    heap->type = ION_HEAP_TYPE_SYSTEM_CONTIG;
    system_heap_contig_has_outer_cache = pheap->has_outer_cache;
    return heap;
}
struct ion_heap *ion_system_heap_create(struct ion_platform_heap *pheap)
{
    struct ion_heap *heap;

    heap = kzalloc(sizeof(struct ion_heap), GFP_KERNEL);
    if (!heap)
        return ERR_PTR(-ENOMEM);
    /*和上面函数的区别仅在于ops*/
    heap->ops = &vmalloc_ops;
    heap->type = ION_HEAP_TYPE_SYSTEM;
    system_heap_has_outer_cache = pheap->has_outer_cache;
    return heap;
}
struct ion_heap *ion_carveout_heap_create(struct ion_platform_heap *heap_data)
{
    struct ion_carveout_heap *carveout_heap;
    int ret;

    carveout_heap = kzalloc(sizeof(struct ion_carveout_heap), GFP_KERNEL);
    if (!carveout_heap)
        return ERR_PTR(-ENOMEM);
    /* 重新创建一个新的pool，这里有点想不通的是为什么不直接使用全局的mempools呢？*/
    carveout_heap->pool = gen_pool_create(12, -1);
    if (!carveout_heap->pool) {
```

关闭

```
        kfree(carveout_heap);
        return ERR_PTR(-ENOMEM);
    }
    carveout_heap->base = heap_data->base;
    ret = gen_pool_add(carveout_heap->pool, carveout_heap->base,
            heap_data->size, -1);
    if (ret < 0) {
        gen_pool_destroy(carveout_heap->pool);
        kfree(carveout_heap);
        return ERR_PTR(-EINVAL);
    }
    carveout_heap->heap.ops = &carveout_heap_ops;
    carveout_heap->heap.type = ION_HEAP_TYPE_CARVEOUT;
    carveout_heap->allocated_bytes = 0;
    carveout_heap->total_size = heap_data->size;
    carveout_heap->has_outer_cache = heap_data->has_outer_cache;

    if (heap_data->extra_data) {
        struct ion_co_heap_pdata *extra_data =
                heap_data->extra_data;

        if (extra_data->setup_region)
            carveout_heap->bus_id = extra_data->setup_region();
        if (extra_data->request_region)
            carveout_heap->request_region =
                    extra_data->request_region;
        if (extra_data->release_region)
            carveout_heap->release_region =
                    extra_data->release_region;
    }
    return &carveout_heap->heap;
}

Heap创建完成，然后保存到idev中：
void ion_device_add_heap(struct ion_device *dev, struct ion_heap *heap)
{
    struct rb_node **p = &dev->heaps.rb_node;
    struct rb_node *parent = NULL;
    struct ion_heap *entry;

    if (!heap->ops->allocate || !heap->ops->free || !heap->ops->map_dma ||
        !heap->ops->unmap_dma)
        pr_err("%s: can not add heap with invalid ops struct.\n",
               __func__);

    heap->dev = dev;
    mutex_lock(&dev->lock);
    while (*p) {
        parent = *p;
        entry = rb_entry(parent, struct ion_heap, node);

        if (heap->id < entry->id) {
            p = &(*p)->rb_left;
        } else if (heap->id > entry->id ) {
            p = &(*p)->rb_right;
        } else {
            pr_err("%s: can not insert multiple heaps with "
                "id %d\n", __func__, heap->id);
            goto end;
        }
    }
    /*使用红黑树保存*/
    rb_link_node(&heap->node, parent, p);
    rb_insert_color(&heap->node, &dev->heaps);
    /*以heap name创建fs,位于ion目录下。如vamlloc, camera preview , audio 等*/
    debugfs_create_file(heap->name, 0664, dev->debug_
                &debug_heap_fops);
end:
    mutex_unlock(&dev->lock);
}
```

关闭

[cpp]

```
<span xmlns="http://www.w3.org/1999/xhtml" style="">struct ion_heap *ion_system_contig_heap_create(struct ion_platfor
{
    struct ion_heap *heap;

    heap = kzalloc(sizeof(struct ion_heap), GFP_KERNEL);
    if (!heap)
```

```
        return ERR_PTR(-ENOMEM);
    /*使用的是kmalloc_ops，上篇有提到哦*/
    heap->ops = &kmalloc_ops;
    heap->type = ION_HEAP_TYPE_SYSTEM_CONTIG;
    system_heap_contig_has_outer_cache = pheap->has_outer_cache;
    return heap;
}
struct ion_heap *ion_system_heap_create(struct ion_platform_heap *pheap)
{
    struct ion_heap *heap;

    heap = kzalloc(sizeof(struct ion_heap), GFP_KERNEL);
    if (!heap)
        return ERR_PTR(-ENOMEM);
    /*和上面函数的区别仅在于ops*/
    heap->ops = &vmalloc_ops;
    heap->type = ION_HEAP_TYPE_SYSTEM;
    system_heap_has_outer_cache = pheap->has_outer_cache;
    return heap;
}
struct ion_heap *ion_carveout_heap_create(struct ion_platform_heap *heap_data)
{
    struct ion_carveout_heap *carveout_heap;
    int ret;

    carveout_heap = kzalloc(sizeof(struct ion_carveout_heap), GFP_KERNEL);
    if (!carveout_heap)
        return ERR_PTR(-ENOMEM);
    /* 重新创建一个新的pool，这里有点想不通的是为什么不直接使用全局的mempools呢？*/
    carveout_heap->pool = gen_pool_create(12, -1);
    if (!carveout_heap->pool) {
        kfree(carveout_heap);
        return ERR_PTR(-ENOMEM);
    }
    carveout_heap->base = heap_data->base;
    ret = gen_pool_add(carveout_heap->pool, carveout_heap->base,
            heap_data->size, -1);
    if (ret < 0) {
        gen_pool_destroy(carveout_heap->pool);
        kfree(carveout_heap);
        return ERR_PTR(-EINVAL);
    }
    carveout_heap->heap.ops = &carveout_heap_ops;
    carveout_heap->heap.type = ION_HEAP_TYPE_CARVEOUT;
    carveout_heap->allocated_bytes = 0;
    carveout_heap->total_size = heap_data->size;
    carveout_heap->has_outer_cache = heap_data->has_outer_cache;

    if (heap_data->extra_data) {
        struct ion_co_heap_pdata *extra_data =
                heap_data->extra_data;

        if (extra_data->setup_region)
            carveout_heap->bus_id = extra_data->setup_region();
        if (extra_data->request_region)
            carveout_heap->request_region =
                    extra_data->request_region;
        if (extra_data->release_region)
            carveout_heap->release_region =
                    extra_data->release_region;
    }
    return &carveout_heap->heap;
}

Heap创建完成，然后保存到idev中：
void ion_device_add_heap(struct ion_device *dev, stru
{
    struct rb_node **p = &dev->heaps.rb_node;
    struct rb_node *parent = NULL;
    struct ion_heap *entry;

    if (!heap->ops->allocate || !heap->ops->free || !heap->ops->map_dma ||
        !heap->ops->unmap_dma)
        pr_err("%s: can not add heap with invalid ops struct.\n",
               __func__);

    heap->dev = dev;
    mutex_lock(&dev->lock);
    while (*p) {
        parent = *p;
        entry = rb_entry(parent, struct ion_heap, node);

        if (heap->id < entry->id) {
            p = &(*p)->rb_left;
        } else if (heap->id > entry->id ) {
            p = &(*p)->rb_right;
        } else {
            pr_err("%s: can not insert multiple heaps with "
                "id %d\n", __func__, heap->id);
            goto end;
        }
    }
    /*使用红黑树保存*/
    rb_link_node(&heap->node, parent, p);
    rb_insert_color(&heap->node, &dev->heaps);
    /*以heap name创建fs,位于ion目录下。如vamlloc, camera_preview , audio 等*/
    debugfs_create_file(heap->name, 0664, dev->debug_root, heap,
                &debug_heap_fops);
end:
    mutex_unlock(&dev->lock);
}
</span>
```

到此，ION初始化已经完成了。接下来该如何使用呢？嗯，通过前面创建的misc设备也就是idev了！

有个fops为ion_fops吗？先来看下用户空间如何使用ION，最后看内核空间如何使用。

ION用户空间使用

[cpp]

```
Ion_fops结构如下：
static const struct file_operations ion_fops = {
    .owner          = THIS_MODULE,
    .open           = ion_open,
    .release        = ion_release,
    .unlocked_ioctl = ion_ioctl,
};

用户空间都是通过ioctl来控制。先看ion_open.

static int ion_open(struct inode *inode, struct file *file)
{
    struct miscdevice *miscdev = file->private_data;
    struct ion_device *dev = container_of(miscdev, struct ion_device, dev);
    struct ion_client *client;
    char debug_name[64];

    pr_debug("%s: %d\n", __func__, __LINE__);
    snprintf(debug_name, 64, "%u", task_pid_nr(current->group_leader));
    /*根据idev和task pid为name创建ion client*/
    client = ion_client_create(dev, -1, debug_name);
    if (IS_ERR_OR_NULL(client))
        return PTR_ERR(client);
    file->private_data = client;

    return 0;
}
```

[cpp]

```
<span xmlns="http://www.w3.org/1999/xhtml" style="">Ion_fops结构如下：
static const struct file_operations ion_fops = {
    .owner          = THIS_MODULE,
    .open           = ion_open,
    .release        = ion_release,
    .unlocked_ioctl = ion_ioctl,
};

用户空间都是通过ioctl来控制。先看ion_open.

static int ion_open(struct inode *inode, struct file *file)
{
    struct miscdevice *miscdev = file->private_data;
    struct ion_device *dev = container_of(miscdev, struct ion_device, dev);
    struct ion_client *client;
    char debug_name[64];

    pr_debug("%s: %d\n", __func__, __LINE__);
    snprintf(debug_name, 64, "%u", task_pid_nr(current->group_leader));
```

关闭

```
    /*根据idev和task pid为name创建ion client*/
    client = ion_client_create(dev, -1, debug_name);
    if (IS_ERR_OR_NULL(client))
        return PTR_ERR(client);
    file->private_data = client;

    return 0;
}</span>
```

前一篇文章有说到，要使用ION, 必须要先创建ion_client, 因此用户空间在open ion的时候创建了client.

[cpp]

```
struct ion_client *ion_client_create(struct ion_device *dev,
                     unsigned int heap_mask,
                     const char *name)
{
    struct ion_client *client;
    struct task_struct *task;
    struct rb_node **p;
    struct rb_node *parent = NULL;
    struct ion_client *entry;
    pid_t pid;
    unsigned int name_len;

    if (!name) {
        pr_err("%s: Name cannot be null\n", __func__);
        return ERR_PTR(-EINVAL);
    }
    name_len = strnlen(name, 64);

    get_task_struct(current->group_leader);
    task_lock(current->group_leader);
    pid = task_pid_nr(current->group_leader);
    /* don't bother to store task struct for kernel threads,
       they can't be killed anyway */
    if (current->group_leader->flags & PF_KTHREAD) {
        put_task_struct(current->group_leader);
        task = NULL;
    } else {
        task = current->group_leader;
    }
    task_unlock(current->group_leader);
    /*分配ion client struct.*/
    client = kzalloc(sizeof(struct ion_client), GFP_KERNEL);
    if (!client) {
        if (task)
            put_task_struct(current->group_leader);
        return ERR_PTR(-ENOMEM);
    }
    /*下面就是保存一系列参数了。*/
    client->dev = dev;
    client->handles = RB_ROOT;
    mutex_init(&client->lock);

    client->name = kzalloc(name_len+1, GFP_KERNEL);
    if (!client->name) {
        put_task_struct(current->group_leader);
        kfree(client);
        return ERR_PTR(-ENOMEM);
    } else {
        strlcpy(client->name, name, name_len+1);
    }

    client->heap_mask = heap_mask;
    client->task = task;
    client->pid = pid;

    mutex_lock(&dev->lock);
    p = &dev->clients.rb_node;
    while (*p) {
        parent = *p;
        entry = rb_entry(parent, struct ion_client, node);

        if (client < entry)
            p = &(*p)->rb_left;
        else if (client > entry)
            p = &(*p)->rb_right;
    }
```

关闭

```
    /*当前client添加到idev的clients根树上去。*/
    rb_link_node(&client->node, parent, p);
    rb_insert_color(&client->node, &dev->clients);

    /*在ION先创建的文件名字是以pid命名的。*/
    client->debug_root = debugfs_create_file(name, 0664,
                         dev->debug_root, client,
                         &debug_client_fops);
    mutex_unlock(&dev->lock);

    return client;
}
```

[cpp]

```cpp
<span xmlns="http://www.w3.org/1999/xhtml" style="">struct ion_client *ion_client_create(struct ion_devic
                     unsigned int heap_mask,
                     const char *name)
{
    struct ion_client *client;
    struct task_struct *task;
    struct rb_node **p;
    struct rb_node *parent = NULL;
    struct ion_client *entry;
    pid_t pid;
    unsigned int name_len;

    if (!name) {
        pr_err("%s: Name cannot be null\n", __func__);
        return ERR_PTR(-EINVAL);
    }
    name_len = strnlen(name, 64);

    get_task_struct(current->group_leader);
    task_lock(current->group_leader);
    pid = task_pid_nr(current->group_leader);
    /* don't bother to store task struct for kernel threads,
       they can't be killed anyway */
    if (current->group_leader->flags & PF_KTHREAD) {
        put_task_struct(current->group_leader);
        task = NULL;
    } else {
        task = current->group_leader;
    }
    task_unlock(current->group_leader);
    /*分配ion client struct.*/
    client = kzalloc(sizeof(struct ion_client), GFP_KERNEL);
    if (!client) {
        if (task)
            put_task_struct(current->group_leader);
        return ERR_PTR(-ENOMEM);
    }
    /*下面就是保存一系列参数了。*/
    client->dev = dev;
    client->handles = RB_ROOT;
    mutex_init(&client->lock);

    client->name = kzalloc(name_len+1, GFP_KERNEL);
    if (!client->name) {
        put_task_struct(current->group_leader);
        kfree(client);
        return ERR_PTR(-ENOMEM);
    } else {
        strlcpy(client->name, name, name_len+1);
    }

    client->heap_mask = heap_mask;
    client->task = task;
    client->pid = pid;

    mutex_lock(&dev->lock);
    p = &dev->clients.rb_node;
    while (*p) {
        parent = *p;
        entry = rb_entry(parent, struct ion_client, node);

        if (client < entry)
            p = &(*p)->rb_left;
        else if (client > entry)
```

关闭

```
            p = &(*p)->rb_right;
    }
    /*当前client添加到idev的clients根树上去。*/
    rb_link_node(&client->node, parent, p);
    rb_insert_color(&client->node, &dev->clients);

    /*在ION先创建的文件名字是以pid命名的。*/
    client->debug_root = debugfs_create_file(name, 0664,
                         dev->debug_root, client,
                         &debug_client_fops);
    mutex_unlock(&dev->lock);

    return client;
}</span>
```

有了client之后，用户程序就可以开始申请分配ION buffer了！通过ioctl命令实现。ion_ioct函数有若干个cmd，ION_IOC_ALLOC和ION_IOC_FREE相对应，表示申请和释放buffer。使用前先要调用ION_IOC_MAP才能得到buffer address，而ION_IOC_IMPORT是为了将这块内存间另一个进程。

[cpp]

```
static long ion_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    struct ion_client *client = filp->private_data;

    switch (cmd) {
    case ION_IOC_ALLOC:
    {
        struct ion_allocation_data data;

        if (copy_from_user(&data, (void __user *)arg, sizeof(data)))
            return -EFAULT;
        /*分配buffer.*/
        data.handle = ion_alloc(client, data.len, data.align,
                         data.flags);

        if (IS_ERR(data.handle))
            return PTR_ERR(data.handle);

        if (copy_to_user((void __user *)arg, &data, sizeof(data))) {
            ion_free(client, data.handle);
            return -EFAULT;
        }
        break;
    }
    case ION_IOC_FREE:
    {
        struct ion_handle_data data;
        bool valid;

        if (copy_from_user(&data, (void __user *)arg,
                   sizeof(struct ion_handle_data)))
            return -EFAULT;
        mutex_lock(&client->lock);
        valid = ion_handle_validate(client, data.handle);
        mutex_unlock(&client->lock);
        if (!valid)
            return -EINVAL;
        ion_free(client, data.handle);
        break;
    }
    case ION_IOC_MAP:
    case ION_IOC_SHARE:
    {
        struct ion_fd_data data;
        int ret;
        if (copy_from_user(&data, (void __user *)arg, sizeof(data)))
            return -EFAULT;
        /*判断当前cmd是否被调用过了，调用过就返回，否则设置flags.*/
        ret = ion_share_set_flags(client, data.handle, filp->f_flags);
        if (ret)
            return ret;

        data.fd = ion_share_dma_buf(client, data.handle);
        if (copy_to_user((void __user *)arg, &data, sizeof(data)))
            return -EFAULT;
        if (data.fd < 0)
```

关闭

```
            return data.fd;
        break;
    }
    case ION_IOC_IMPORT:
    {
        struct ion_fd_data data;
        int ret = 0;
        if (copy_from_user(&data, (void __user *)arg,
                   sizeof(struct ion_fd_data)))
            return -EFAULT;
        data.handle = ion_import_dma_buf(client, data.fd);
        if (IS_ERR(data.handle))
            data.handle = NULL;
        if (copy_to_user((void __user *)arg, &data,
                 sizeof(struct ion_fd_data)))
            return -EFAULT;
        if (ret < 0)
            return ret;
        break;
    }
    case ION_IOC_CUSTOM:
~~snip
    case ION_IOC_CLEAN_CACHES:
    case ION_IOC_INV_CACHES:
    case ION_IOC_CLEAN_INV_CACHES:
    ~~snip
    case ION_IOC_GET_FLAGS:
~~snip
    default:
        return -ENOTTY;
    }
    return 0;
}
```

[cpp]

```
<span xmlns="http://www.w3.org/1999/xhtml" style="">static long ion_ioctl(struct file *filp, unsigned int cmd, unsigne
{
    struct ion_client *client = filp->private_data;

    switch (cmd) {
    case ION_IOC_ALLOC:
    {
        struct ion_allocation_data data;

        if (copy_from_user(&data, (void __user *)arg, sizeof(data)))
            return -EFAULT;
        /*分配buffer.*/
        data.handle = ion_alloc(client, data.len, data.align,
                         data.flags);

        if (IS_ERR(data.handle))
            return PTR_ERR(data.handle);

        if (copy_to_user((void __user *)arg, &data, sizeof(data))) {
            ion_free(client, data.handle);
            return -EFAULT;
        }
        break;
    }
    case ION_IOC_FREE:
    {
        struct ion_handle_data data;
        bool valid;

        if (copy_from_user(&data, (void __user *)arg,
                   sizeof(struct ion_handle_data)))
            return -EFAULT;
        mutex_lock(&client->lock);
        valid = ion_handle_validate(client, data.handle);
        mutex_unlock(&client->lock);
        if (!valid)
            return -EINVAL;
        ion_free(client, data.handle);
        break;
    }
    case ION_IOC_MAP:
    case ION_IOC_SHARE:
    {
```

关闭

```
        struct ion_fd_data data;
        int ret;
        if (copy_from_user(&data, (void __user *)arg, sizeof(data)))
            return -EFAULT;
        /*判断当前cmd是否被调用过了，调用过就返回，否则设置flags.*/
        ret = ion_share_set_flags(client, data.handle, filp->f_flags);
        if (ret)
            return ret;

        data.fd = ion_share_dma_buf(client, data.handle);
        if (copy_to_user((void __user *)arg, &data, sizeof(data)))
            return -EFAULT;
        if (data.fd < 0)
            return data.fd;
        break;
    }
    case ION_IOC_IMPORT:
    {
        struct ion_fd_data data;
        int ret = 0;
        if (copy_from_user(&data, (void __user *)arg,
                   sizeof(struct ion_fd_data)))
            return -EFAULT;
        data.handle = ion_import_dma_buf(client, data.fd);
        if (IS_ERR(data.handle))
            data.handle = NULL;
        if (copy_to_user((void __user *)arg, &data,
                 sizeof(struct ion_fd_data)))
            return -EFAULT;
        if (ret < 0)
            return ret;
        break;
    }
    case ION_IOC_CUSTOM:
~~snip
    case ION_IOC_CLEAN_CACHES:
    case ION_IOC_INV_CACHES:
    case ION_IOC_CLEAN_INV_CACHES:
    ~~snip
    case ION_IOC_GET_FLAGS:
~~snip
    default:
        return -ENOTTY;
    }
    return 0;
}</span>
```

下面分小节说明分配和共享的原理。

ION_IOC_ALLOC

[cpp]

```
struct ion_handle *ion_alloc(struct ion_client *client, size_t len,
                 size_t align, unsigned int flags)
{
~~snip

    mutex_lock(&dev->lock);
    /*循环遍历当前Heap链表。*/
    for (n = rb_first(&dev->heaps); n != NULL; n = rb_next(n)) {
        struct ion_heap *heap = rb_entry(n, struct ion_heap, node);
/*只有heap type和id都符合才去创建buffer.*/
        /* if the client doesn't support this heap type */
        if (!((1 << heap->type) & client->heap_mask))
            continue;
        /* if the caller didn't specify this heap type */
        if (!((1 << heap->id) & flags))
            continue;
        /* Do not allow un-secure heap if secure is specified */
        if (secure_allocation && (heap->type != ION_HEAP_TYPE_CP))
            continue;
        buffer = ion_buffer_create(heap, dev, len, align, flags);
~~snip
    }
    mutex_unlock(&dev->lock);

~~snip
    /*创建了buffer之后，就相应地创建handle来管理buffer.*/
```

关闭

```
    handle = ion_handle_create(client, buffer);

~~snip
}

找到Heap之后调用ion_buffer_create：
static struct ion_buffer *ion_buffer_create(struct ion_heap *heap,
                     struct ion_device *dev,
                     unsigned long len,
                     unsigned long align,
                     unsigned long flags)
{
    struct ion_buffer *buffer;
    struct sg_table *table;
    int ret;
    /*分配struct ion buffer,用来管理buffer.*/
    buffer = kzalloc(sizeof(struct ion_buffer), GFP_KERNEL);
    if (!buffer)
        return ERR_PTR(-ENOMEM);

    buffer->heap = heap;
    kref_init(&buffer->ref);
    /*调用相应heap type的ops allocate。还记得前面有提到过不同种类的ops吗，
如carveout_heap_ops ，vmalloc_ops 。*/
    ret = heap->ops->allocate(heap, buffer, len, align, flags);
    if (ret) {
        kfree(buffer);
        return ERR_PTR(ret);
    }

    buffer->dev = dev;
    buffer->size = len;
    /*http://lwn.net/Articles/263343/*/
    table = buffer->heap->ops->map_dma(buffer->heap, buffer);
    if (IS_ERR_OR_NULL(table)) {
        heap->ops->free(buffer);
        kfree(buffer);
        return ERR_PTR(PTR_ERR(table));
    }
    buffer->sg_table = table;

    mutex_init(&buffer->lock);
    /*将当前ion buffer添加到idev 的buffers 树上统一管理。*/
    ion_buffer_add(dev, buffer);
    return buffer;
}
```

[cpp]

```
<span xmlns="http://www.w3.org/1999/xhtml" style="">struct ion_handle *ion_alloc(struct ion_client *client, size_t le
                 size_t align, unsigned int flags)
{
~~snip

    mutex_lock(&dev->lock);
    /*循环遍历当前Heap链表。*/
    for (n = rb_first(&dev->heaps); n != NULL; n = rb_next(n)) {
        struct ion_heap *heap = rb_entry(n, struct ion_heap, node);
/*只有heap type和id都符合才去创建buffer.*/
        /* if the client doesn't support this heap type */
        if (!((1 << heap->type) & client->heap_mask))
            continue;
        /* if the caller didn't specify this heap type */
        if (!((1 << heap->id) & flags))
            continue;
        /* Do not allow un-secure heap if secure is specified */
        if (secure_allocation && (heap->type != ION_HEAP_TYPE_CP))
            continue;
        buffer = ion_buffer_create(heap, dev, len, align, flags);
~~snip
    }
    mutex_unlock(&dev->lock);

~~snip
    /*创建了buffer之后，就相应地创建handle来管理buffer.*/
    handle = ion_handle_create(client, buffer);

~~snip
}
```

关闭

```

找到Heap之后调用ion_buffer_create：
static struct ion_buffer *ion_buffer_create(struct ion_heap *heap,
                     struct ion_device *dev,
                     unsigned long len,
                     unsigned long align,
                     unsigned long flags)
{
    struct ion_buffer *buffer;
    struct sg_table *table;
    int ret;
    /*分配struct ion buffer,用来管理buffer.*/
    buffer = kzalloc(sizeof(struct ion_buffer), GFP_KERNEL);
    if (!buffer)
        return ERR_PTR(-ENOMEM);

    buffer->heap = heap;
    kref_init(&buffer->ref);
    /*调用相应heap type的ops allocate。还记得前面有提到过不同种类的ops吗，
如carveout_heap_ops ，vmalloc_ops 。*/
    ret = heap->ops->allocate(heap, buffer, len, align, flags);
    if (ret) {
        kfree(buffer);
        return ERR_PTR(ret);
    }

    buffer->dev = dev;
    buffer->size = len;
    /*http://lwn.net/Articles/263343/*/
    table = buffer->heap->ops->map_dma(buffer->heap, buffer);
    if (IS_ERR_OR_NULL(table)) {
        heap->ops->free(buffer);
        kfree(buffer);
        return ERR_PTR(PTR_ERR(table));
    }
    buffer->sg_table = table;

    mutex_init(&buffer->lock);
    /*将当前ion buffer添加到idev 的buffers 树上统一管理。*/
    ion_buffer_add(dev, buffer);
    return buffer;
}</span>
```

[cpp]

```
<p>static struct ion_handle *ion_handle_create(struct ion_client *client,
         struct ion_buffer *buffer)
{
 struct ion_handle *handle;
 /*分配struct ion_handle.*/
 handle = kzalloc(sizeof(struct ion_handle), GFP_KERNEL);
 if (!handle)
  return ERR_PTR(-ENOMEM);
 kref_init(&handle->ref);
 rb_init_node(&handle->node);
 handle->client = client; //client放入handle中
 ion_buffer_get(buffer); //引用计数加1
 handle->buffer = buffer; //buffer也放入handle中</p><p> return handle;
}
</p>
```

[cpp]

```
<p><span xmlns="http://www.w3.org
/1999/xhtml" style="">static struct ion_handle *ion_handle_create(struct ion_client *cl
         struct ion_buffer *buffer)
{
 struct ion_handle *handle;
 /*分配struct ion_handle.*/
 handle = kzalloc(sizeof(struct ion_handle), GFP_KERNEL);
 if (!handle)
  return ERR_PTR(-ENOMEM);
 kref_init(&handle->ref);
 rb_init_node(&handle->node);
 handle->client = client; //client放入handle中
 ion_buffer_get(buffer); //引用计数加1
 handle->buffer = buffer; //buffer也放入handle中</span></p><p><span xmlns="http:
//www.w3.org/1999/xhtml" style=""> return handle;
}
</span></p>
```

关闭

先拿heap type为ION_HEAP_TYPE_CARVEOUT为例，看下它是如何分配buffer的。

allocate对应ion_carveout_heap_allocate。

[cpp]

```cpp
static int ion_carveout_heap_allocate(struct ion_heap *heap,
                      struct ion_buffer *buffer,
                      unsigned long size, unsigned long align,
                      unsigned long flags)
{
    buffer->priv_phys = ion_carveout_allocate(heap, size, align);
    return buffer->priv_phys == ION_CARVEOUT_ALLOCATE_FAIL ? -ENOMEM : 0;
}
ion_phys_addr_t ion_carveout_allocate(struct ion_heap *heap,
                      unsigned long size,
                      unsigned long align)
{
    struct ion_carveout_heap *carveout_heap =
        container_of(heap, struct ion_carveout_heap, heap);
    /*通过创建的mem pool来管理buffer,由于这块buffer在初始化的
时候就预留了，现在只要从上面拿一块区域就可以了。*/
    unsigned long offset = gen_pool_alloc_aligned(carveout_heap->pool,
                            size, ilog2(align));
    /*分配不成功可能是没有内存空间可供分配了或者是有碎片导致的。*/
    if (!offset) {
        if ((carveout_heap->total_size -
              carveout_heap->allocated_bytes) >= size)
            pr_debug("%s: heap %s has enough memory (%lx) but"
                " the allocation of size %lx still failed."
                " Memory is probably fragmented.",
                __func__, heap->name,
                carveout_heap->total_size -
                carveout_heap->allocated_bytes, size);
        return ION_CARVEOUT_ALLOCATE_FAIL;
    }
    /*已经分配掉的内存字节。*/
    carveout_heap->allocated_bytes += size;
    return offset;
}
```

[cpp]

```cpp
<span xmlns="http://www.w3.org/1999/xhtml" style="">static int ion_carveout_heap_allocate(struct ion_heap *heap,
                      struct ion_buffer *buffer,
                      unsigned long size, unsigned long align,
                      unsigned long flags)
{
    buffer->priv_phys = ion_carveout_allocate(heap, size, align);
    return buffer->priv_phys == ION_CARVEOUT_ALLOCATE_FAIL ? -ENOMEM : 0;
}
ion_phys_addr_t ion_carveout_allocate(struct ion_heap *heap,
                      unsigned long size,
                      unsigned long align)
{
    struct ion_carveout_heap *carveout_heap =
        container_of(heap, struct ion_carveout_heap, heap);
    /*通过创建的mem pool来管理buffer,由于这块buffer在初始化的
时候就预留了，现在只要从上面拿一块区域就可以了。*/
    unsigned long offset = gen_pool_alloc_aligned(carveout_heap->pool,
                            size, ilog2(align));
    /*分配不成功可能是没有内存空间可供分配了或者是有碎片导致的。*/
    if (!offset) {
        if ((carveout_heap->total_size -
              carveout_heap->allocated_bytes) >= size
            pr_debug("%s: heap %s has enough memory (%lx) but"
                " the allocation of size %lx still failed."
                " Memory is probably fragmented.",
                __func__, heap->name,
                carveout_heap->total_size -
                carveout_heap->allocated_bytes, size);
        return ION_CARVEOUT_ALLOCATE_FAIL;
    }
    /*已经分配掉的内存字节。*/
    carveout_heap->allocated_bytes += size;
    return offset;
}</span>
```

关闭

同样地，对于heap type为ION_HEAP_TYPE_SYSTEM的分配函数是ion_system_heap_allocate。

[cpp]

```cpp
static int ion_system_contig_heap_allocate(struct ion_heap *heap,
                       struct ion_buffer *buffer,
                       unsigned long len,
                       unsigned long align,
                       unsigned long flags)
{
    /*通过kzalloc分配。*/
    buffer->priv_virt = kzalloc(len, GFP_KERNEL);
    if (!buffer->priv_virt)
        return -ENOMEM;
    atomic_add(len, &system_contig_heap_allocated);
    return 0;
}
```

[cpp]

```cpp
<span xmlns="http://www.w3.org/1999/xhtml" style="">static int ion_system_contig_heap_allocate(struct ion_heap *heap,
                       struct ion_buffer *buffer,
                       unsigned long len,
                       unsigned long align,
                       unsigned long flags)
{
    /*通过kzalloc分配。*/
    buffer->priv_virt = kzalloc(len, GFP_KERNEL);
    if (!buffer->priv_virt)
        return -ENOMEM;
    atomic_add(len, &system_contig_heap_allocated);
    return 0;
}</span>
```

其他的几种Heap type可自行研究，接着调用ion_buffer_add将buffer添加到dev的buffers树上去

[cpp]

```cpp
static void ion_buffer_add(struct ion_device *dev,
               struct ion_buffer *buffer)
{
    struct rb_node **p = &dev->buffers.rb_node;
    struct rb_node *parent = NULL;
    struct ion_buffer *entry;

    while (*p) {
        parent = *p;
        entry = rb_entry(parent, struct ion_buffer, node);

        if (buffer < entry) {
            p = &(*p)->rb_left;
        } else if (buffer > entry) {
            p = &(*p)->rb_right;
        } else {
            pr_err("%s: buffer already found.", __func__);
            BUG();
        }
    }
/*又是使用红黑树哦！*/
    rb_link_node(&buffer->node, parent, p);
    rb_insert_color(&buffer->node, &dev->buffers);
}
```

关闭

[cpp]

```cpp
<span xmlns="http://www.w3.org/1999/xhtml" style="">static void ion_buffer_add(struct ion_device *dev,
               struct ion_buffer *buffer)
{
    struct rb_node **p = &dev->buffers.rb_node;
    struct rb_node *parent = NULL;
    struct ion_buffer *entry;

    while (*p) {
        parent = *p;
        entry = rb_entry(parent, struct ion_buffer, node);

        if (buffer < entry) {
            p = &(*p)->rb_left;
        } else if (buffer > entry) {
            p = &(*p)->rb_right;
        } else {
            pr_err("%s: buffer already found.", __func__);
            BUG();
        }
    }
/*又是使用红黑树哦！*/
    rb_link_node(&buffer->node, parent, p);
    rb_insert_color(&buffer->node, &dev->buffers);
}</span>
```

至此，已经得到client和handle，buffer分配完成！

ION_IOC_MAP/ ION_IOC_SHARE

[cpp]

```
int ion_share_dma_buf(struct ion_client *client, struct ion_handle *handle)
{
    struct ion_buffer *buffer;
    struct dma_buf *dmabuf;
    bool valid_handle;
    int fd;

    mutex_lock(&client->lock);
    valid_handle = ion_handle_validate(client, handle);
    mutex_unlock(&client->lock);
    if (!valid_handle) {
        WARN(1, "%s: invalid handle passed to share.\n", __func__);
        return -EINVAL;
    }

    buffer = handle->buffer;
    ion_buffer_get(buffer);
    /*生成一个新的file描述符*/
    dmabuf = dma_buf_export(buffer, &dma_buf_ops, buffer->size, O_RDWR);
    if (IS_ERR(dmabuf)) {
        ion_buffer_put(buffer);
        return PTR_ERR(dmabuf);
    }
    /*将file转换用户空间识别的fd描述符。*/
    fd = dma_buf_fd(dmabuf, O_CLOEXEC);
    if (fd < 0)
        dma_buf_put(dmabuf);

    return fd;
}
struct dma_buf *dma_buf_export(void *priv, const struct dma_buf_ops *ops,
                size_t size, int flags)
{
    struct dma_buf *dmabuf;
    struct file *file;
~~snip
    /*分配struct dma_buf.*/
    dmabuf = kzalloc(sizeof(struct dma_buf), GFP_KERNEL);
    if (dmabuf == NULL)
        return ERR_PTR(-ENOMEM);
    /*保存信息到dmabuf，注意ops为dma_buf_ops，后面mmap为调用到。*/
    dmabuf->priv = priv;
    dmabuf->ops = ops;
    dmabuf->size = size;
    /*产生新的file*/
    file = anon_inode_getfile("dmabuf", &dma_buf_fops, dmabuf, flags);

    dmabuf->file = file;

    mutex_init(&dmabuf->lock);
    INIT_LIST_HEAD(&dmabuf->attachments);

    return dmabuf;
}
```

关闭

[cpp]

```
<span xmlns="http://www.w3.org
/1999/xhtml" style="">int ion_share_dma_buf(struct ion_client *client, struct ion_handl
{
    struct ion_buffer *buffer;
    struct dma_buf *dmabuf;
    bool valid_handle;
    int fd;

    mutex_lock(&client->lock);
    valid_handle = ion_handle_validate(client, handle);
    mutex_unlock(&client->lock);
    if (!valid_handle) {
        WARN(1, "%s: invalid handle passed to share.\n", __func__);
        return -EINVAL;
    }

    buffer = handle->buffer;
    ion_buffer_get(buffer);
    /*生成一个新的file描述符*/
    dmabuf = dma_buf_export(buffer, &dma_buf_ops, buffer->size, O_RDWR);
    if (IS_ERR(dmabuf)) {
        ion_buffer_put(buffer);
        return PTR_ERR(dmabuf);
    }
    /*将file转换用户空间识别的fd描述符。*/
    fd = dma_buf_fd(dmabuf, O_CLOEXEC);
    if (fd < 0)
        dma_buf_put(dmabuf);

    return fd;
}
struct dma_buf *dma_buf_export(void *priv, const struct dma_buf_ops *ops,
                size_t size, int flags)
{
    struct dma_buf *dmabuf;
    struct file *file;
~~snip
    /*分配struct dma_buf.*/
    dmabuf = kzalloc(sizeof(struct dma_buf), GFP_KERNEL);
    if (dmabuf == NULL)
        return ERR_PTR(-ENOMEM);
    /*保存信息到dmabuf，注意ops为dma_buf_ops，后面mmap为调用到。*/
    dmabuf->priv = priv;
    dmabuf->ops = ops;
    dmabuf->size = size;
    /*产生新的file*/
    file = anon_inode_getfile("dmabuf", &dma_buf_fops, dmabuf, flags);

    dmabuf->file = file;

    mutex_init(&dmabuf->lock);
    INIT_LIST_HEAD(&dmabuf->attachments);

    return dmabuf;
}</span>
```

通过上述过程，用户空间就得到了新的fd,重新生成一个新的fd的目的是考虑了两个用户空间进程想共享这块heap内存的情况。然后再对fd作mmap，相应地kernel空间就调用到了file 的dma_buf_fops中的dma_buf_mmap_internal。

[cpp]

```
static const struct file_operations dma_buf_fops = {
    .release    = dma_buf_release,
    .mmap       = dma_buf_mmap_internal,
};
static int dma_buf_mmap_internal(struct file *file, struct vm_area_struct *vma)
 {
    struct dma_buf *dmabuf;

    if (!is_dma_buf_file(file))
        return -EINVAL;

    dmabuf = file->private_data;
    /*检查用户空间要映射的size是否比目前dmabuf也就是当前heap的size
还要大，如果是就返回无效。*/
    /* check for overflowing the buffer's size */
    if (vma->vm_pgoff + ((vma->vm_end - vma->vm_start) >> PAGE_SHIFT) >
        dmabuf->size >> PAGE_SHIFT)
        return -EINVAL;
    /*调用的是dma_buf_ops 的mmap函数*/
    return dmabuf->ops->mmap(dmabuf, vma);
```

关闭

```
 }

struct dma_buf_ops dma_buf_ops = {
    .map_dma_buf = ion_map_dma_buf,
    .unmap_dma_buf = ion_unmap_dma_buf,
    .mmap = ion_mmap,
    .release = ion_dma_buf_release,
    .begin_cpu_access = ion_dma_buf_begin_cpu_access,
    .end_cpu_access = ion_dma_buf_end_cpu_access,
    .kmap_atomic = ion_dma_buf_kmap,
    .kunmap_atomic = ion_dma_buf_kunmap,
    .kmap = ion_dma_buf_kmap,
    .kunmap = ion_dma_buf_kunmap,
};
static int ion_mmap(struct dma_buf *dmabuf, struct vm_area_struct *vma)
{
    struct ion_buffer *buffer = dmabuf->priv;
    int ret;

    if (!buffer->heap->ops->map_user) {
        pr_err("%s: this heap does not define a method for mapping "
               "to userspace\n", __func__);
        return -EINVAL;
    }

    mutex_lock(&buffer->lock);
    /* now map it to userspace */
    /*调用的是相应heap的map_user，如carveout_heap_ops 调用的是
ion_carveout_heap_map_user ，此函数就是一般的mmap实现，不追下去了。*/
    ret = buffer->heap->ops->map_user(buffer->heap, buffer, vma);

    if (ret) {
        mutex_unlock(&buffer->lock);
        pr_err("%s: failure mapping buffer to userspace\n",
               __func__);
    } else {
        buffer->umap_cnt++;
        mutex_unlock(&buffer->lock);

        vma->vm_ops = &ion_vm_ops;
        /*
         * move the buffer into the vm_private_data so we can access it
         * from vma_open/close
         */
        vma->vm_private_data = buffer;
    }
    return ret;
}
```

[cpp]

```
<span xmlns="http://www.w3.org/1999/xhtml" style="">static const struct file_operations dma_buf_fops = {
    .release    = dma_buf_release,
    .mmap       = dma_buf_mmap_internal,
};
static int dma_buf_mmap_internal(struct file *file, struct vm_area_struct *vma)
 {
    struct dma_buf *dmabuf;

    if (!is_dma_buf_file(file))
        return -EINVAL;

    dmabuf = file->private_data;
    /*检查用户空间要映射的size是否比目前dmabuf也就是当前heap的size
还要大，如果是就返回无效。*/
    /* check for overflowing the buffer's size */
    if (vma->vm_pgoff + ((vma->vm_end - vma->vm_start) >> PAGE_SHIFT) >
        dmabuf->size >> PAGE_SHIFT)
        return -EINVAL;
    /*调用的是dma_buf_ops 的mmap函数*/
    return dmabuf->ops->mmap(dmabuf, vma);
 }

struct dma_buf_ops dma_buf_ops = {
    .map_dma_buf = ion_map_dma_buf,
    .unmap_dma_buf = ion_unmap_dma_buf,
    .mmap = ion_mmap,
    .release = ion_dma_buf_release,
    .begin_cpu_access = ion_dma_buf_begin_cpu_access,
```

关闭

```
    .end_cpu_access = ion_dma_buf_end_cpu_access,
    .kmap_atomic = ion_dma_buf_kmap,
    .kunmap_atomic = ion_dma_buf_kunmap,
    .kmap = ion_dma_buf_kmap,
    .kunmap = ion_dma_buf_kunmap,
};
static int ion_mmap(struct dma_buf *dmabuf, struct vm_area_struct *vma)
{
    struct ion_buffer *buffer = dmabuf->priv;
    int ret;

    if (!buffer->heap->ops->map_user) {
        pr_err("%s: this heap does not define a method for mapping "
               "to userspace\n", __func__);
        return -EINVAL;
    }

    mutex_lock(&buffer->lock);
    /* now map it to userspace */
    /*调用的是相应heap的map_user，如carveout_heap_ops 调用的是
ion_carveout_heap_map_user ，此函数就是一般的mmap实现，不追下去了。*/
    ret = buffer->heap->ops->map_user(buffer->heap, buffer, vma);

    if (ret) {
        mutex_unlock(&buffer->lock);
        pr_err("%s: failure mapping buffer to userspace\n",
               __func__);
    } else {
        buffer->umap_cnt++;
        mutex_unlock(&buffer->lock);

        vma->vm_ops = &ion_vm_ops;
        /*
         * move the buffer into the vm_private_data so we can access it
         * from vma_open/close
         */
        vma->vm_private_data = buffer;
    }
    return ret;
}</span>
```

至此，用户空间就得到了bufferaddress，然后可以使用了！

ION_IOC_IMPORT

当用户空间另一个进程需要这块heap的时候，ION_IOC_IMPORT就派上用处了！注意，传进去的fd为在ION_IOC_SHARE中得到的。

[cpp]

```
struct ion_handle *ion_import_dma_buf(struct ion_client *client, int fd)
{

    struct dma_buf *dmabuf;
    struct ion_buffer *buffer;
    struct ion_handle *handle;

    dmabuf = dma_buf_get(fd);
    if (IS_ERR_OR_NULL(dmabuf))
        return ERR_PTR(PTR_ERR(dmabuf));
    /* if this memory came from ion */
~~snip
    buffer = dmabuf->priv;

    mutex_lock(&client->lock);
    /* if a handle exists for this buffer just take a
/*查找是否已经存在对应的handle了，没有则创建。因为另外一个进程只是
调用了open 接口，对应的只创建了client，并没有handle。
*/
    handle = ion_handle_lookup(client, buffer);
    if (!IS_ERR_OR_NULL(handle)) {
        ion_handle_get(handle);
        goto end;
    }
    handle = ion_handle_create(client, buffer);
    if (IS_ERR_OR_NULL(handle))
        goto end;
    ion_handle_add(client, handle);
end:
    mutex_unlock(&client->lock);
```

关闭

```
    dma_buf_put(dmabuf);
    return handle;
}
```

[cpp]

```
<span xmlns="http://www.w3.org/1999/xhtml" style="">struct ion_handle *ion_import_dma_buf(struct ion_client *client, :
{

    struct dma_buf *dmabuf;
    struct ion_buffer *buffer;
    struct ion_handle *handle;

    dmabuf = dma_buf_get(fd);
    if (IS_ERR_OR_NULL(dmabuf))
        return ERR_PTR(PTR_ERR(dmabuf));
    /* if this memory came from ion */
~~snip
    buffer = dmabuf->priv;

    mutex_lock(&client->lock);
    /* if a handle exists for this buffer just take a reference to it */
/*查找是否已经存在对应的handle了，没有则创建。因为另外一个进程只是
调用了open 接口，对应的只创建了client，并没有handle。
*/
    handle = ion_handle_lookup(client, buffer);
    if (!IS_ERR_OR_NULL(handle)) {
        ion_handle_get(handle);
        goto end;
    }
    handle = ion_handle_create(client, buffer);
    if (IS_ERR_OR_NULL(handle))
        goto end;
    ion_handle_add(client, handle);
end:
    mutex_unlock(&client->lock);
    dma_buf_put(dmabuf);
    return handle;
}</span>
```

这样，用户空间另一个进程也得到了对应的bufferHandle，client/buffer/handle之间连接起来了！然后另一个一个进程就也可以使用mmap来操作这块heap buffer了。

和一般的进程使用ION区别就是共享的进程之间struction_buffer是共享的，而struct ion_handle是各自的。

可见，ION的使用流程还是比较清晰的。不过要记得的是，使用好了ION，一定要释放掉，否则会导致内存泄露。

ION内核空间使用

内核空间使用ION也是大同小异，按照创建client,buffer,handle的流程，只是它的使用对用户空间来说是透明的罢了！

ion_client_create在kernel空间被Qualcomm给封装了下。

[cpp]

```
struct ion_client *msm_ion_client_create(unsigned int heap_mask,
                    const char *name)
{
    return ion_client_create(idev, heap_mask, name);
}
```

[cpp]

```
<span xmlns="http://www.w3.org/1999/xhtml" style="">struct ion_client *msm_ion_client_create(unsigned int heap_mask,
                    const char *name)
{
    return ion_client_create(idev, heap_mask, name);
}</span>
```

关闭

调用的流程也类似，不过map的时候调用的是heap对应的map_kernel()而不是map_user().

msm_ion_client_create -> ion_alloc ->ion_map_kernel

参考文档：

http://lwn.net/Articles/480055/

《ARM体系结构与编程》存储系统章节。

其handle纳入进程file desc空间而不是/dev/ion设备内单独的handle空间，方便之处如下：

每个buffer一个handle，便于更灵活地细粒度地控制每个buffer的使用周期；

向用户进程输出fd，细粒度地对每个buffer进行mmap；

使用struct file可以重用已有struct file_operations进行mmap；

在binder driver中以BINDER_TYPE_FD类型为不同进程传递提供支撑，并借助fget/fput从struct file级别进行kref控制；

当不需要在用户态访问时，是不需要与struct file关联的，内核结构ion_handle/ion_buffer唯一的表征了该buffer，所以与struct file关联的工作是在ioctl(ion, ION_IOC_SHARE/ION_IOC_MAP, &share)中完成并输出的，用于后续的mmap调用；或者该进程不需要mmap而是仅仅向别的进程binder transfer，这就实现了用户态进行buffer流转控制，而内核态完成buffer数据流转。

转自http://blog.csdn.net/kris_fei/article/details/8588661 &http://blog.csdn.net/kris_fei/article/detai

考察平台：

chipset: MSM8X25Q

codebase: Android 4.1

ION概念：

ION是Google的下一代内存管理器，用来支持不同的内存分配机制，如CARVOUT(PMEM)，物理连续内存(kmalloc), 虚拟地址连续但物理不连续内存(vmalloc)， IOMMU等。

用户空间和内核空间都可以使用ION，用户空间是通过/dev/ion来创建client的。

说到client, 顺便看下ION相关比较重要的几个概念。

Heap: 用来表示内存分配的相关信息，包括id, type, name等。用struct ion_heap表示。

Client: Ion的使用者，用户空间和内核控件要使用ION的buffer,必须先创建一个client,一个client可以有多个buffer，用struct ion_buffer表示。

Handle: 将buffer该抽象出来，可以认为ION用handle来管理buffer，一般用户直接拿到的是handle,而不是buffer。用struct ion_handle表示。

heap类型：

由于ION可以使用多种memory分配机制，例如物理连续和不连续的，所以ION使用enum ion_heap_type表示。

[cpp]

```
/**
 * enum ion_heap_types - list of all possible types of heaps
 * @ION_HEAP_TYPE_SYSTEM:    memory allocated via vmalloc
 * @ION_HEAP_TYPE_SYSTEM_CONTIG: memory allocated via kmalloc
 * @ION_HEAP_TYPE_CARVEOUT:  memory allocated from a prereserved
 *               carveout heap, allocations are physically
 *               contiguous
 * @ION_HEAP_TYPE_IOMMU: IOMMU memory
 * @ION_HEAP_TYPE_CP:    memory allocated from a prereserved
 *              carveout heap, allocations are physically
 *              contiguous. Used for content protection.
 * @ION_HEAP_TYPE_DMA:          memory allocated via DMA API
 * @ION_HEAP_END:       helper for iterating over heaps
 */
enum ion_heap_type {
    ION_HEAP_TYPE_SYSTEM,
    ION_HEAP_TYPE_SYSTEM_CONTIG,
    ION_HEAP_TYPE_CARVEOUT,
    ION_HEAP_TYPE_IOMMU,
    ION_HEAP_TYPE_CP,
    ION_HEAP_TYPE_DMA,
    ION_HEAP_TYPE_CUSTOM, /* must be last so device specific heaps always
                 are at the end of this enum */
    ION_NUM_HEAPS,
};
```

关闭

[cpp]

```
/**
 * enum ion_heap_types - list of all possible types of heaps
 * @ION_HEAP_TYPE_SYSTEM:    memory allocated via vmalloc
 * @ION_HEAP_TYPE_SYSTEM_CONTIG: memory allocated via kmalloc
 * @ION_HEAP_TYPE_CARVEOUT:  memory allocated from a prereserved
 *               carveout heap, allocations are physically
 *               contiguous
 * @ION_HEAP_TYPE_IOMMU: IOMMU memory
 * @ION_HEAP_TYPE_CP:    memory allocated from a prereserved
 *              carveout heap, allocations are physically
 *              contiguous. Used for content protection.
 * @ION_HEAP_TYPE_DMA:          memory allocated via DMA API
 * @ION_HEAP_END:       helper for iterating over heaps
 */
enum ion_heap_type {
    ION_HEAP_TYPE_SYSTEM,
    ION_HEAP_TYPE_SYSTEM_CONTIG,
    ION_HEAP_TYPE_CARVEOUT,
    ION_HEAP_TYPE_IOMMU,
    ION_HEAP_TYPE_CP,
    ION_HEAP_TYPE_DMA,
    ION_HEAP_TYPE_CUSTOM, /* must be last so device specific heaps always
                 are at the end of this enum */
    ION_NUM_HEAPS,
};
```

代码中的注释很明确地说明了哪种type对应的是分配哪种memory。不同type的heap需要不同的me

不过都是用struct ion_heap_ops来表示的。如以下例子：

[cpp]

```
static struct ion_heap_ops carveout_heap_ops = {
    .allocate = ion_carveout_heap_allocate,
    .free = ion_carveout_heap_free,
    .phys = ion_carveout_heap_phys,
    .map_user = ion_carveout_heap_map_user,
    .map_kernel = ion_carveout_heap_map_kernel,
    .unmap_user = ion_carveout_heap_unmap_user,
    .unmap_kernel = ion_carveout_heap_unmap_kernel,
    .map_dma = ion_carveout_heap_map_dma,
    .unmap_dma = ion_carveout_heap_unmap_dma,
    .cache_op = ion_carveout_cache_ops,
    .print_debug = ion_carveout_print_debug,
    .map_iommu = ion_carveout_heap_map_iommu,
    .unmap_iommu = ion_carveout_heap_unmap_iommu,
};

static struct ion_heap_ops kmalloc_ops = {
    .allocate = ion_system_contig_heap_allocate,
    .free = ion_system_contig_heap_free,
    .phys = ion_system_contig_heap_phys,
    .map_dma = ion_system_contig_heap_map_dma,
    .unmap_dma = ion_system_heap_unmap_dma,
    .map_kernel = ion_system_heap_map_kernel,
    .unmap_kernel = ion_system_heap_unmap_kernel,
    .map_user = ion_system_contig_heap_map_user,
    .cache_op = ion_system_contig_heap_cache_ops,
    .print_debug = ion_system_contig_print_debug,
    .map_iommu = ion_system_contig_heap_map_iommu,
    .unmap_iommu = ion_system_heap_unmap_iommu,
};
```

[cpp]

```
static struct ion_heap_ops carveout_heap_ops = {
    .allocate = ion_carveout_heap_allocate,
    .free = ion_carveout_heap_free,
    .phys = ion_carveout_heap_phys,
    .map_user = ion_carveout_heap_map_user,
    .map_kernel = ion_carveout_heap_map_kernel,
    .unmap_user = ion_carveout_heap_unmap_user,
    .unmap_kernel = ion_carveout_heap_unmap_kernel,
    .map_dma = ion_carveout_heap_map_dma,
    .unmap_dma = ion_carveout_heap_unmap_dma,
    .cache_op = ion_carveout_cache_ops,
    .print_debug = ion_carveout_print_debug,
    .map_iommu = ion_carveout_heap_map_iommu,
    .unmap_iommu = ion_carveout_heap_unmap_iommu,
};

static struct ion_heap_ops kmalloc_ops = {
    .allocate = ion_system_contig_heap_allocate,
    .free = ion_system_contig_heap_free,
    .phys = ion_system_contig_heap_phys,
    .map_dma = ion_system_contig_heap_map_dma,
    .unmap_dma = ion_system_heap_unmap_dma,
    .map_kernel = ion_system_heap_map_kernel,
    .unmap_kernel = ion_system_heap_unmap_kernel,
```

关闭

```
    .map_user = ion_system_contig_heap_map_user,
    .cache_op = ion_system_contig_heap_cache_ops,
    .print_debug = ion_system_contig_print_debug,
    .map_iommu = ion_system_contig_heap_map_iommu,
    .unmap_iommu = ion_system_heap_unmap_iommu,
};
```

Heap ID：

同一种type的heap上当然可以分为若该干个chunk供用户使用，所以ION又使用ID来区分了。例如在type为ION_HEAP_TYPE_CARVEOUT的heap上，audio和display部分都需要使用，ION就用ID来区分。

Heap id用enumion_heap_ids表示。

[cpp]

```cpp
/**
 * These are the only ids that should be used for Ion heap ids.
 * The ids listed are the order in which allocation will be attempted
 * if specified. Don't swap the order of heap ids unless you know what
 * you are doing!
 * Id's are spaced by purpose to allow new Id's to be inserted in-between (f
 * possible fallbacks)
 */

enum ion_heap_ids {
    INVALID_HEAP_ID = -1,
    ION_CP_MM_HEAP_ID = 8,
    ION_CP_MFC_HEAP_ID = 12,
    ION_CP_WB_HEAP_ID = 16, /* 8660 only */
    ION_CAMERA_HEAP_ID = 20, /* 8660 only */
    ION_SF_HEAP_ID = 24,
    ION_IOMMU_HEAP_ID = 25,
    ION_QSECOM_HEAP_ID = 26,
    ION_AUDIO_HEAP_BL_ID = 27,
    ION_AUDIO_HEAP_ID = 28,

    ION_MM_FIRMWARE_HEAP_ID = 29,
    ION_SYSTEM_HEAP_ID = 30,

    ION_HEAP_ID_RESERVED = 31 /** Bit reserved for ION_SECURE flag */
};
```

[cpp]

```cpp
/**
 * These are the only ids that should be used for Ion heap ids.
 * The ids listed are the order in which allocation will be attempted
 * if specified. Don't swap the order of heap ids unless you know what
 * you are doing!
 * Id's are spaced by purpose to allow new Id's to be inserted in-between (for
 * possible fallbacks)
 */

enum ion_heap_ids {
    INVALID_HEAP_ID = -1,
    ION_CP_MM_HEAP_ID = 8,
    ION_CP_MFC_HEAP_ID = 12,
    ION_CP_WB_HEAP_ID = 16, /* 8660 only */
    ION_CAMERA_HEAP_ID = 20, /* 8660 only */
    ION_SF_HEAP_ID = 24,
    ION_IOMMU_HEAP_ID = 25,
    ION_QSECOM_HEAP_ID = 26,
    ION_AUDIO_HEAP_BL_ID = 27,
    ION_AUDIO_HEAP_ID = 28,

    ION_MM_FIRMWARE_HEAP_ID = 29,
    ION_SYSTEM_HEAP_ID = 30,

    ION_HEAP_ID_RESERVED = 31 /** Bit reserved for ION_SECURE flag */
};
```

关闭

Heap 定义：

了解了heaptype和id，看看如何被用到了，本平台使用的文件为board-qrd7627a.c，有如下定义：

[cpp]

```cpp
/**
 * These heaps are listed in the order they will be allocated.
 * Don't swap the order unless you know what you are doing!
 */
struct ion_platform_heap msm7627a_heaps[] = {
        {
            .id = ION_SYSTEM_HEAP_ID,
            .type   = ION_HEAP_TYPE_SYSTEM,
            .name   = ION_VMALLOC_HEAP_NAME,
        },
#ifdef CONFIG_MSM_MULTIMEDIA_USE_ION
        /* PMEM_ADSP = CAMERA */
        {
            .id = ION_CAMERA_HEAP_ID,
            .type   = CAMERA_HEAP_TYPE,
            .name   = ION_CAMERA_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_mm_ion_pdata,
            .priv   = (void *)&ion_cma_device.dev,
        },
        /* AUDIO HEAP 1*/
        {
            .id = ION_AUDIO_HEAP_ID,
            .type   = ION_HEAP_TYPE_CARVEOUT,
            .name   = ION_AUDIO_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
        },
        /* PMEM_MDP = SF */
        {
            .id = ION_SF_HEAP_ID,
            .type   = ION_HEAP_TYPE_CARVEOUT,
            .name   = ION_SF_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
        },
        /* AUDIO HEAP 2*/
        {
            .id    = ION_AUDIO_HEAP_BL_ID,
            .type  = ION_HEAP_TYPE_CARVEOUT,
            .name  = ION_AUDIO_BL_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
            .base = BOOTLOADER_BASE_ADDR,
        },

#endif
};
```

[cpp]

```cpp
/**
 * These heaps are listed in the order they will be allocated.
 * Don't swap the order unless you know what you are doing!
 */
struct ion_platform_heap msm7627a_heaps[] = {
        {
            .id = ION_SYSTEM_HEAP_ID,
            .type   = ION_HEAP_TYPE_SYSTEM,
            .name   = ION_VMALLOC_HEAP_NAME,
        },
#ifdef CONFIG_MSM_MULTIMEDIA_USE_ION
        /* PMEM_ADSP = CAMERA */
        {
            .id = ION_CAMERA_HEAP_ID,
            .type   = CAMERA_HEAP_TYPE,
            .name   = ION_CAMERA_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_mm_ion_pdata,
            .priv   = (void *)&ion_cma_device.dev,
        },
        /* AUDIO HEAP 1*/
        {
            .id = ION_AUDIO_HEAP_ID,
            .type   = ION_HEAP_TYPE_CARVEOUT,
            .name   = ION_AUDIO_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
```

关闭

```
        },
        /* PMEM_MDP = SF */
        {
            .id = ION_SF_HEAP_ID,
            .type   = ION_HEAP_TYPE_CARVEOUT,
            .name   = ION_SF_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
        },
        /* AUDIO HEAP 2*/
        {
            .id    = ION_AUDIO_HEAP_BL_ID,
            .type  = ION_HEAP_TYPE_CARVEOUT,
            .name  = ION_AUDIO_BL_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
            .base = BOOTLOADER_BASE_ADDR,
        },

#endif
};
```

ION Handle：

当Ion client分配buffer时，相应的一个唯一的handle也会被指定，当然client可以多次申请ion buffer。申请好buffer之后，返回的是一个ion handle, 不过要知道Ion buffer才和实际的内存相关，包括size, address等信息。Struct ion_handle和struct ion_buffer如下：

[cpp]

```cpp
/**
 * ion_handle - a client local reference to a buffer
 * @ref:        reference count
 * @client:     back pointer to the client the buffer resides in
 * @buffer:     pointer to the buffer
 * @node:       node in the client's handle rbtree
 * @kmap_cnt:       count of times this client has mapped to kernel
 * @dmap_cnt:       count of times this client has mapped for dma
 *
 * Modifications to node, map_cnt or mapping should be protected by the
 * lock in the client.  Other fields are never changed after initialization.
 */
struct ion_handle {
    struct kref ref;
    struct ion_client *client;
    struct ion_buffer *buffer;
    struct rb_node node;
    unsigned int kmap_cnt;
    unsigned int iommu_map_cnt;
};

/**
 * struct ion_buffer - metadata for a particular buffer
 * @ref:        refernce count
 * @node:       node in the ion_device buffers tree
 * @dev:        back pointer to the ion_device
 * @heap:       back pointer to the heap the buffer came from
 * @flags:      buffer specific flags
 * @size:       size of the buffer
 * @priv_virt:      private data to the buffer representable as
 *          a void *
 * @priv_phys:      private data to the buffer representable as
 *          an ion_phys_addr_t (and someday a phys_addr_t)
 * @lock:       protects the buffers cnt fields
 * @kmap_cnt:       number of times the buffer is mapped to the kernel
 * @vaddr:      the kenrel mapping if kmap_cnt is not zero
 * @dmap_cnt:       number of times the buffer is mapped for dma
 * @sg_table:       the sg table for the buffer if dmap_cnt is not zero
*/
struct ion_buffer {
    struct kref ref;
    struct rb_node node;
    struct ion_device *dev;
    struct ion_heap *heap;
    unsigned long flags;
    size_t size;
    union {
```

关闭

```
        void *priv_virt;
        ion_phys_addr_t priv_phys;
    };
    struct mutex lock;
    int kmap_cnt;
    void *vaddr;
    int dmap_cnt;
    struct sg_table *sg_table;
    int umap_cnt;
    unsigned int iommu_map_cnt;
    struct rb_root iommu_maps;
    int marked;
};
```

[cpp]

```cpp
/**
 * ion_handle - a client local reference to a buffer
 * @ref:        reference count
 * @client:     back pointer to the client the buffer resides in
 * @buffer:     pointer to the buffer
 * @node:       node in the client's handle rbtree
 * @kmap_cnt:       count of times this client has mapped to kernel
 * @dmap_cnt:       count of times this client has mapped for dma
 *
 * Modifications to node, map_cnt or mapping should be protected by the
 * lock in the client.  Other fields are never changed after initialization.
 */
struct ion_handle {
    struct kref ref;
    struct ion_client *client;
    struct ion_buffer *buffer;
    struct rb_node node;
    unsigned int kmap_cnt;
    unsigned int iommu_map_cnt;
};

/**
 * struct ion_buffer - metadata for a particular buffer
 * @ref:        refernce count
 * @node:       node in the ion_device buffers tree
 * @dev:        back pointer to the ion_device
 * @heap:       back pointer to the heap the buffer came from
 * @flags:      buffer specific flags
 * @size:       size of the buffer
 * @priv_virt:      private data to the buffer representable as
 *          a void *
 * @priv_phys:      private data to the buffer representable as
 *          an ion_phys_addr_t (and someday a phys_addr_t)
 * @lock:       protects the buffers cnt fields
 * @kmap_cnt:       number of times the buffer is mapped to the kernel
 * @vaddr:      the kenrel mapping if kmap_cnt is not zero
 * @dmap_cnt:       number of times the buffer is mapped for dma
 * @sg_table:       the sg table for the buffer if dmap_cnt is not zero
*/
struct ion_buffer {
    struct kref ref;
    struct rb_node node;
    struct ion_device *dev;
    struct ion_heap *heap;
    unsigned long flags;
    size_t size;
    union {
        void *priv_virt;
        ion_phys_addr_t priv_phys;
    };
    struct mutex lock;
    int kmap_cnt;
    void *vaddr;
    int dmap_cnt;
    struct sg_table *sg_table;
    int umap_cnt;
    unsigned int iommu_map_cnt;
    struct rb_root iommu_maps;
    int marked;
};
```

关闭

ION Client：

用户空间和内核空间都可以成为client，不过创建的方法稍稍有点区别，先了解下基本的操作流程吧。

内核空间:

先创建client:

[cpp]

```cpp
struct ion_client *ion_client_create(struct ion_device *dev,
                     unsigned int heap_mask,
                     const char *name)
```

[cpp]

```cpp
struct ion_client *ion_client_create(struct ion_device *dev,
                     unsigned int heap_mask,
                     const char *name)
```

heap_mask: 可以分配的heap type，如carveout,system heap, iommu等。

高通使用msm_ion_client_create函数封装了下。

有了client之后就可以分配内存：

[cpp]

```cpp
struct ion_handle *ion_alloc(struct ion_client *client, size_t len,
                 size_t align, unsigned int flags)
```

[cpp]

```cpp
struct ion_handle *ion_alloc(struct ion_client *client, size_t len,
                 size_t align, unsigned int flags)
```

flags: 分配的heap id.

有了handle也就是buffer之后就准备使用了，不过还是物理地址，需要map：

[cpp]

```cpp
void *ion_map_kernel(struct ion_client *client, struct ion_handle *handle,
            unsigned long flags)
```

[cpp]

```cpp
void *ion_map_kernel(struct ion_client *client, struct ion_handle *handle,
            unsigned long flags)
```

用户空间:

用户空间如果想使用ION，也必须先要创建client,不过它是打开/dev/ion,实际上它最终也会调用ion_client_create。

不过和内核空间创建client的一点区别是，用户空间不能选择heap type（使用预订的heap id隐含heap type），但是内核空间却可以。

另外，用户空间是通过IOCTL来分配内存的，cmd为ION_IOC_ALLOC.

[cpp]

```cpp
ion_fd = open("/dev/ion", O_ RDONLY | O_SYNC);
ioctl(ion_fd, ION_IOC_ALLOC, alloc);
```

[cpp]

```cpp
ion_fd = open("/dev/ion", O_ RDONLY | O_SYNC);
ioctl(ion_fd, ION_IOC_ALLOC, alloc);
```

关闭

alloc为struct ion_allocation_data,len是申请buffer的长度，flags是heap id。

[cpp]

```cpp
/**
 * struct ion_allocation_data - metadata passed from userspace for allocations
 * @len:    size of the allocation
 * @align:  required alignment of the allocation
 * @flags:  flags passed to heap
 * @handle: pointer that will be populated with a cookie to use to refer
 *      to this allocation
 *
 * Provided by userspace as an argument to the ioctl
 */
struct ion_allocation_data {
    size_t len;
    size_t align;
    unsigned int flags;
    struct ion_handle *handle;
};
```

[cpp]

```
/**
 * struct ion_allocation_data - metadata passed from userspace for allocations
 * @len:    size of the allocation
 * @align:  required alignment of the allocation
 * @flags:  flags passed to heap
 * @handle: pointer that will be populated with a cookie to use to refer
 *      to this allocation
 *
 * Provided by userspace as an argument to the ioctl
 */
struct ion_allocation_data {
    size_t len;
    size_t align;
    unsigned int flags;
    struct ion_handle *handle;
};
```

分配好了buffer之后，如果用户空间想使用buffer，先需要mmap. ION是通过先调用IOCTL中的ION_IOC_SHARE/ION_IOC_MAP来得到可以mmap的fd,然后再执行mmap得到bufferaddress.
然后，你也可以将此fd传给另一个进程，如通过binder传递。在另一个进程中通过ION_IOC_IMPORT这个IOCTL来得到这块共享buffer了。
来看一个例子：

[cpp]

```
进程A：
int ionfd = open("/dev/ion", O_RDONLY | O_DSYNC);
alloc_data.len = 0x1000;
alloc_data.align = 0x1000;
alloc_data.flags = ION_HEAP(ION_CP_MM_HEAP_ID);
rc = ioctl(ionfd,ION_IOC_ALLOC, &alloc_data);
fd_data.handle = alloc_data.handle;
rc = ioctl(ionfd,ION_IOC_SHARE,&fd_data);
shared_fd = fd_data.fd;

进程B：
fd_data.fd = shared_fd;
rc = ioctl(ionfd,ION_IOC_IMPORT,&fd_data);
```

[cpp]

```
进程A：
int ionfd = open("/dev/ion", O_RDONLY | O_DSYNC);
alloc_data.len = 0x1000;
alloc_data.align = 0x1000;
alloc_data.flags = ION_HEAP(ION_CP_MM_HEAP_ID);
rc = ioctl(ionfd,ION_IOC_ALLOC, &alloc_data);
fd_data.handle = alloc_data.handle;
rc = ioctl(ionfd,ION_IOC_SHARE,&fd_data);
shared_fd = fd_data.fd;

进程B：
fd_data.fd = shared_fd;
rc = ioctl(ionfd,ION_IOC_IMPORT,&fd_data);
```

关闭

从上一篇ION基本概念中，我们了解了heaptype, heap id, client, handle以及如何使用，本篇再从原理上分析下ION的运作流程。
MSM8x25Q平台使用的是board-qrd7627.c，ION相关定义如下：

[cpp]

```
/**
```

```
 * These heaps are listed in the order they will be allocated.
 * Don't swap the order unless you know what you are doing!
 */
struct ion_platform_heap msm7627a_heaps[] = {
        {
            .id = ION_SYSTEM_HEAP_ID,
            .type   = ION_HEAP_TYPE_SYSTEM,
            .name   = ION_VMALLOC_HEAP_NAME,
        },
#ifdef CONFIG_MSM_MULTIMEDIA_USE_ION
        /* PMEM_ADSP = CAMERA */
        {
            .id = ION_CAMERA_HEAP_ID,
            .type   = CAMERA_HEAP_TYPE,
            .name   = ION_CAMERA_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_mm_ion_pdata,
            .priv   = (void *)&ion_cma_device.dev,
        },
        /* AUDIO HEAP 1*/
        {
            .id = ION_AUDIO_HEAP_ID,
            .type   = ION_HEAP_TYPE_CARVEOUT,
            .name   = ION_AUDIO_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
        },
        /* PMEM_MDP = SF */
        {
            .id = ION_SF_HEAP_ID,
            .type   = ION_HEAP_TYPE_CARVEOUT,
            .name   = ION_SF_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
        },
        /* AUDIO HEAP 2*/
        {
            .id    = ION_AUDIO_HEAP_BL_ID,
            .type  = ION_HEAP_TYPE_CARVEOUT,
            .name  = ION_AUDIO_BL_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
            .base = BOOTLOADER_BASE_ADDR,
        },

#endif
};

static struct ion_co_heap_pdata co_ion_pdata = {
    .adjacent_mem_id = INVALID_HEAP_ID,
    .align = PAGE_SIZE,
};

static struct ion_co_heap_pdata co_mm_ion_pdata = {
    .adjacent_mem_id = INVALID_HEAP_ID,
    .align = PAGE_SIZE,
};

static u64 msm_dmamask = DMA_BIT_MASK(32);

static struct platform_device ion_cma_device = {
    .name = "ion-cma-device",
    .id = -1,
    .dev = {
        .dma_mask = &msm_dmamask,
        .coherent_dma_mask = DMA_BIT_MASK(32),
    }
};
```

关闭

[cpp]

```
/**
 * These heaps are listed in the order they will be allocated.
 * Don't swap the order unless you know what you are doing!
 */
struct ion_platform_heap msm7627a_heaps[] = {
        {
            .id = ION_SYSTEM_HEAP_ID,
            .type   = ION_HEAP_TYPE_SYSTEM,
            .name   = ION_VMALLOC_HEAP_NAME,
        },
#ifdef CONFIG_MSM_MULTIMEDIA_USE_ION
        /* PMEM_ADSP = CAMERA */
        {
            .id = ION_CAMERA_HEAP_ID,
            .type   = CAMERA_HEAP_TYPE,
            .name   = ION_CAMERA_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_mm_ion_pdata,
            .priv   = (void *)&ion_cma_device.dev,
        },
        /* AUDIO HEAP 1*/
        {
            .id = ION_AUDIO_HEAP_ID,
            .type   = ION_HEAP_TYPE_CARVEOUT,
            .name   = ION_AUDIO_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
        },
        /* PMEM_MDP = SF */
        {
            .id = ION_SF_HEAP_ID,
            .type   = ION_HEAP_TYPE_CARVEOUT,
            .name   = ION_SF_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
        },
        /* AUDIO HEAP 2*/
        {
            .id    = ION_AUDIO_HEAP_BL_ID,
            .type  = ION_HEAP_TYPE_CARVEOUT,
            .name  = ION_AUDIO_BL_HEAP_NAME,
            .memory_type = ION_EBI_TYPE,
            .extra_data = (void *)&co_ion_pdata,
            .base = BOOTLOADER_BASE_ADDR,
        },

#endif
};

static struct ion_co_heap_pdata co_ion_pdata = {
    .adjacent_mem_id = INVALID_HEAP_ID,
    .align = PAGE_SIZE,
};

static struct ion_co_heap_pdata co_mm_ion_pdata = {
    .adjacent_mem_id = INVALID_HEAP_ID,
    .align = PAGE_SIZE,
};

static u64 msm_dmamask = DMA_BIT_MASK(32);

static struct platform_device ion_cma_device = {
    .name = "ion-cma-device",
    .id = -1,
    .dev = {
        .dma_mask = &msm_dmamask,
        .coherent_dma_mask = DMA_BIT_MASK(32),
    }
};
```

Qualcomm提示了不要轻易调换顺序，因为后面代码处理是将顺序定死了的，一旦你调换了，代码就无法正常运行了。

另外， 本系统中只使用了ION_HEAP_TYPE_CARVEOUT和 ION_HEAP_TYPE_SYSTEM这两种heap type.

关闭

对于ION_HEAP_TYPE_CARVEOUT的内存分配，后面将会发现，其实就是之前讲述过的使用mem pool来分配的。

Platform device如下,在msm_ion.c中用到。

[cpp]

```
static struct ion_platform_data ion_pdata = {
    .nr = MSM_ION_HEAP_NUM,
    .has_outer_cache = 1,
    .heaps = msm7627a_heaps,
};
```

```

static struct platform_device ion_dev = {
    .name = "ion-msm",
    .id = 1,
    .dev = { .platform_data = &ion_pdata },
};
```

[cpp]

```
static struct ion_platform_data ion_pdata = {
    .nr = MSM_ION_HEAP_NUM,
    .has_outer_cache = 1,
    .heaps = msm7627a_heaps,
};

static struct platform_device ion_dev = {
    .name = "ion-msm",
    .id = 1,
    .dev = { .platform_data = &ion_pdata },
};
```

ION初始化

转到msm_ion.c，ion.c的某些函数也被重新封装了下.万事都从设备匹配开始：

[cpp]

```
static struct platform_driver msm_ion_driver = {
    .probe = msm_ion_probe,
    .remove = msm_ion_remove,
    .driver = { .name = "ion-msm" }
};
static int __init msm_ion_init(void)
{
    /*调用msm_ion_probe */
    return platform_driver_register(&msm_ion_driver);
}

static int msm_ion_probe(struct platform_device *pdev)
{
    /*即board-qrd7627a.c中的ion_pdata */
    struct ion_platform_data *pdata = pdev->dev.platform_data;
    int err;
    int i;

    /*heap数量*/
    num_heaps = pdata->nr;
    /*分配struct ion_heap */
    heaps = kcalloc(pdata->nr, sizeof(struct ion_heap *), GFP_KERNEL);

    if (!heaps) {
        err = -ENOMEM;
        goto out;
    }
    /*创建节点，最终是/dev/ion,供用户空间操作。*/
    idev = ion_device_create(NULL);
    if (IS_ERR_OR_NULL(idev)) {
        err = PTR_ERR(idev);
        goto freeheaps;
    }
    /*最终是根据adjacent_mem_id 是否定义了来分配相邻内存，
我们没用到，忽略此函数。*/
    msm_ion_heap_fixup(pdata->heaps, num_heaps);

    /* create the heaps as specified in the board file */
    for (i = 0; i < num_heaps; i++) {
        struct ion_platform_heap *heap_data = &pdata->heaps[i];
        /*分配ion*/
        msm_ion_allocate(heap_data);

        heap_data->has_outer_cache = pdata->has_outer_cache;
        /*创建ion heap。*/
        heaps[i] = ion_heap_create(heap_data);
        if (IS_ERR_OR_NULL(heaps[i])) {
            heaps[i] = 0;
            continue;
        } else {
            if (heap_data->size)
                pr_info("ION heap %s created at %lx "
                    "with size %x\n", heap_data->name,
```

关闭

```
                              heap_data->base,
                              heap_data->size);
            else
                pr_info("ION heap %s created\n",
                              heap_data->name);
        }
        /*创建的heap添加到idev中，以便后续使用。*/
        ion_device_add_heap(idev, heaps[i]);
    }
    /*检查heap之间是否有重叠部分*/
    check_for_heap_overlap(pdata->heaps, num_heaps);
    platform_set_drvdata(pdev, idev);
    return 0;

freeheaps:
    kfree(heaps);
out:
    return err;
}

通过ion_device_create创建/dev/ion节点：
struct ion_device *ion_device_create(long (*custom_ioctl)
                     (struct ion_client *client,
                      unsigned int cmd,
                      unsigned long arg))
{
    struct ion_device *idev;
    int ret;

    idev = kzalloc(sizeof(struct ion_device), GFP_KERNEL);
    if (!idev)
        return ERR_PTR(-ENOMEM);
    /*是个misc设备*/
    idev->dev.minor = MISC_DYNAMIC_MINOR;
    /*节点名字为ion*/
    idev->dev.name = "ion";
    /*fops为ion_fops,所以对应ion的操作都会调用ion_fops的函数指针。*/
    idev->dev.fops = &ion_fops;
    idev->dev.parent = NULL;
    ret = misc_register(&idev->dev);
    if (ret) {
        pr_err("ion: failed to register misc device.\n");
        return ERR_PTR(ret);
    }
    /*创建debugfs目录，路径为/sys/kernel/debug/ion/*/
    idev->debug_root = debugfs_create_dir("ion", NULL);
    if (IS_ERR_OR_NULL(idev->debug_root))
        pr_err("ion: failed to create debug files.\n");

    idev->custom_ioctl = custom_ioctl;
    idev->buffers = RB_ROOT;
    mutex_init(&idev->lock);
    idev->heaps = RB_ROOT;
    idev->clients = RB_ROOT;
    /*在ion目录下创建一个check_leaked_fds文件，用来检查Ion的使用是否有内存泄漏。如果申请了ion之后不需要使用却没有释放，就会导致memory leak.*/
    debugfs_create_file("check_leaked_fds", 0664, idev->debug_root, idev,
                &debug_leak_fops);
    return idev;
}

msm_ion_allocate：
static void msm_ion_allocate(struct ion_platform_heap *heap)
{

    if (!heap->base && heap->extra_data) {
        unsigned int align = 0;
        switch (heap->type) {
        /*获取align参数*/
        case ION_HEAP_TYPE_CARVEOUT:
            align =
            ((struct ion_co_heap_pdata *) heap->extra_data)->align;
            break;
        /*此type我们没使用到。*/
        case ION_HEAP_TYPE_CP:
        {
            struct ion_cp_heap_pdata *data =
                (struct ion_cp_heap_pdata *)
                heap->extra_data;
            if (data->reusable) {
```

关闭

```
                const struct fmem_data *fmem_info =
                    fmem_get_info();
                heap->base = fmem_info->phys;
                data->virt_addr = fmem_info->virt;
                pr_info("ION heap %s using FMEM\n", heap->name);
            } else if (data->mem_is_fmem) {
                const struct fmem_data *fmem_info =
                    fmem_get_info();
                heap->base = fmem_info->phys + fmem_info->size;
            }
            align = data->align;
            break;
        }
        default:
            break;
        }
        if (align && !heap->base) {
            /*获取heap的base address。*/
            heap->base = msm_ion_get_base(heap->size,
                              heap->memory_type,
                              align);
            if (!heap->base)
                pr_err("%s: could not get memory for heap %s "
                   "(id %x)\n", __func__, heap->name, heap->id);
        }
    }
}

static unsigned long msm_ion_get_base(unsigned long size, int memory_type,
                    unsigned int align)
{
    switch (memory_type) {
    /*我们定义的是ebi type，看见没，此函数在mem pool中分析过了。
原理就是使用Mempool 来管理分配内存。*/
    case ION_EBI_TYPE:
        return allocate_contiguous_ebi_nomap(size, align);
        break;
    case ION_SMI_TYPE:
        return allocate_contiguous_memory_nomap(size, MEMTYPE_SMI,
                            align);
        break;
    default:
        pr_err("%s: Unknown memory type %d\n", __func__, memory_type);
        return 0;
    }
}
ion_heap_create：
struct ion_heap *ion_heap_create(struct ion_platform_heap *heap_data)
{
    struct ion_heap *heap = NULL;
    /*根据Heap type调用相应的创建函数。*/
    switch (heap_data->type) {
    case ION_HEAP_TYPE_SYSTEM_CONTIG:
        heap = ion_system_contig_heap_create(heap_data);
        break;
    case ION_HEAP_TYPE_SYSTEM:
        heap = ion_system_heap_create(heap_data);
        break;
    case ION_HEAP_TYPE_CARVEOUT:
        heap = ion_carveout_heap_create(heap_data);
        break;
    case ION_HEAP_TYPE_IOMMU:
        heap = ion_iommu_heap_create(heap_data);
        break;
    case ION_HEAP_TYPE_CP:
        heap = ion_cp_heap_create(heap_data);
        break;
#ifdef CONFIG_CMA
    case ION_HEAP_TYPE_DMA:
        heap = ion_cma_heap_create(heap_data);
        break;
#endif
    default:
        pr_err("%s: Invalid heap type %d\n", __func__,
               heap_data->type);
        return ERR_PTR(-EINVAL);
    }

    if (IS_ERR_OR_NULL(heap)) {
        pr_err("%s: error creating heap %s type %d base %lu size %u\n",
```

关闭

```
               __func__, heap_data->name, heap_data->type,
               heap_data->base, heap_data->size);
        return ERR_PTR(-EINVAL);
    }
    /*保存Heap的name,id和私有数据。*/
    heap->name = heap_data->name;
    heap->id = heap_data->id;
    heap->priv = heap_data->priv;
    return heap;
}
```

[cpp]

```
static struct platform_driver msm_ion_driver = {
    .probe = msm_ion_probe,
    .remove = msm_ion_remove,
    .driver = { .name = "ion-msm" }
};
static int __init msm_ion_init(void)
{
    /*调用msm_ion_probe */
    return platform_driver_register(&msm_ion_driver);
}

static int msm_ion_probe(struct platform_device *pdev)
{
    /*即board-qrd7627a.c中的ion_pdata */
    struct ion_platform_data *pdata = pdev->dev.platform_data;
    int err;
    int i;

    /*heap数量*/
    num_heaps = pdata->nr;
    /*分配struct ion_heap */
    heaps = kcalloc(pdata->nr, sizeof(struct ion_heap *), GFP_KERNEL);

    if (!heaps) {
        err = -ENOMEM;
        goto out;
    }
    /*创建节点，最终是/dev/ion,供用户空间操作。*/
    idev = ion_device_create(NULL);
    if (IS_ERR_OR_NULL(idev)) {
        err = PTR_ERR(idev);
        goto freeheaps;
    }
    /*最终是根据adjacent_mem_id 是否定义了来分配相邻内存，
我们没用到，忽略此函数。*/
    msm_ion_heap_fixup(pdata->heaps, num_heaps);

    /* create the heaps as specified in the board file */
    for (i = 0; i < num_heaps; i++) {
        struct ion_platform_heap *heap_data = &pdata->heaps[i];
        /*分配ion*/
        msm_ion_allocate(heap_data);

        heap_data->has_outer_cache = pdata->has_outer_cache;
        /*创建ion heap。*/
        heaps[i] = ion_heap_create(heap_data);
        if (IS_ERR_OR_NULL(heaps[i])) {
            heaps[i] = 0;
            continue;
        } else {
            if (heap_data->size)
                pr_info("ION heap %s created at %lx "
                    "with size %x\n", heap_data->name
                              heap_data->base,
                              heap_data->size);
            else
                pr_info("ION heap %s created\n",
                              heap_data->name);
        }
        /*创建的heap添加到idev中，以便后续使用。*/
        ion_device_add_heap(idev, heaps[i]);
    }
    /*检查heap之间是否有重叠部分*/
    check_for_heap_overlap(pdata->heaps, num_heaps);
    platform_set_drvdata(pdev, idev);
    return 0;

```

关闭

```
freeheaps:
    kfree(heaps);
out:
    return err;
}

通过ion_device_create创建/dev/ion节点：
struct ion_device *ion_device_create(long (*custom_ioctl)
                     (struct ion_client *client,
                      unsigned int cmd,
                      unsigned long arg))
{
    struct ion_device *idev;
    int ret;

    idev = kzalloc(sizeof(struct ion_device), GFP_KERNEL);
    if (!idev)
        return ERR_PTR(-ENOMEM);
    /*是个misc设备*/
    idev->dev.minor = MISC_DYNAMIC_MINOR;
    /*节点名字为ion*/
    idev->dev.name = "ion";
    /*fops为ion_fops,所以对应ion的操作都会调用ion_fops的函数指针。*/
    idev->dev.fops = &ion_fops;
    idev->dev.parent = NULL;
    ret = misc_register(&idev->dev);
    if (ret) {
        pr_err("ion: failed to register misc device.\n");
        return ERR_PTR(ret);
    }
    /*创建debugfs目录，路径为/sys/kernel/debug/ion/*/
    idev->debug_root = debugfs_create_dir("ion", NULL);
    if (IS_ERR_OR_NULL(idev->debug_root))
        pr_err("ion: failed to create debug files.\n");

    idev->custom_ioctl = custom_ioctl;
    idev->buffers = RB_ROOT;
    mutex_init(&idev->lock);
    idev->heaps = RB_ROOT;
    idev->clients = RB_ROOT;
    /*在ion目录下创建一个check_leaked_fds文件，用来检查Ion的使用是否有内存泄漏。如果申请了ion之后
不需要使用却没有释放，就会导致memory leak.*/
    debugfs_create_file("check_leaked_fds", 0664, idev->debug_root, idev,
                &debug_leak_fops);
    return idev;
}

msm_ion_allocate：
static void msm_ion_allocate(struct ion_platform_heap *heap)
{

    if (!heap->base && heap->extra_data) {
        unsigned int align = 0;
        switch (heap->type) {
        /*获取align参数*/
        case ION_HEAP_TYPE_CARVEOUT:
            align =
            ((struct ion_co_heap_pdata *) heap->extra_data)->align;
            break;
        /*此type我们没使用到。*/
        case ION_HEAP_TYPE_CP:
        {
            struct ion_cp_heap_pdata *data =
                (struct ion_cp_heap_pdata *)
                heap->extra_data;
            if (data->reusable) {
                const struct fmem_data *fmem_info =
                    fmem_get_info();
                heap->base = fmem_info->phys;
                data->virt_addr = fmem_info->virt;
                pr_info("ION heap %s using FMEM\n", heap->name);
            } else if (data->mem_is_fmem) {
                const struct fmem_data *fmem_info =
                    fmem_get_info();
                heap->base = fmem_info->phys + fmem_info->size;
            }
            align = data->align;
            break;
        }
        default:
            break;
        }
        if (align && !heap->base) {
            /*获取heap的base address。*/
            heap->base = msm_ion_get_base(heap->size,
                              heap->memory_type,
                              align);
            if (!heap->base)
                pr_err("%s: could not get memory for heap %s "
                   "(id %x)\n", __func__, heap->name, heap->id);
        }
    }
}

static unsigned long msm_ion_get_base(unsigned long size, int memory_type,
                    unsigned int align)
{
    switch (memory_type) {
    /*我们定义的是ebi type，看见没，此函数在mem pool中分析过了。
原理就是使用Mempool 来管理分配内存。*/
    case ION_EBI_TYPE:
        return allocate_contiguous_ebi_nomap(size, align);
        break;
    case ION_SMI_TYPE:
        return allocate_contiguous_memory_nomap(size, MEMTYPE_SMI,
                            align);
        break;
    default:
        pr_err("%s: Unknown memory type %d\n", __func__, memory_type);
        return 0;
    }
}
ion_heap_create：
struct ion_heap *ion_heap_create(struct ion_platform_heap *heap_data)
{
    struct ion_heap *heap = NULL;
    /*根据Heap type调用相应的创建函数。*/
    switch (heap_data->type) {
    case ION_HEAP_TYPE_SYSTEM_CONTIG:
        heap = ion_system_contig_heap_create(heap_data);
        break;
    case ION_HEAP_TYPE_SYSTEM:
        heap = ion_system_heap_create(heap_data);
        break;
    case ION_HEAP_TYPE_CARVEOUT:
        heap = ion_carveout_heap_create(heap_data);
        break;
    case ION_HEAP_TYPE_IOMMU:
        heap = ion_iommu_heap_create(heap_data);
        break;
    case ION_HEAP_TYPE_CP:
        heap = ion_cp_heap_create(heap_data);
        break;
#ifdef CONFIG_CMA
    case ION_HEAP_TYPE_DMA:
        heap = ion_cma_heap_create(heap_data);
        break;
#endif
    default:
        pr_err("%s: Invalid heap type %d\n", __func__,
               heap_data->type);
        return ERR_PTR(-EINVAL);
    }

    if (IS_ERR_OR_NULL(heap)) {
        pr_err("%s: error creating heap %s type %d ba
               __func__, heap_data->name, heap_data->type,
               heap_data->base, heap_data->size);
        return ERR_PTR(-EINVAL);
    }
    /*保存Heap的name,id和私有数据。*/
    heap->name = heap_data->name;
    heap->id = heap_data->id;
    heap->priv = heap_data->priv;
    return heap;
}
```

关闭

从下面的代码可以得知，ION_HEAP_TYPE_SYSTEM_CONTIG使用kmalloc创建的，ION_HEAP_TYPE_SYSTEM使用的是vmalloc,而ion_carveout_heap_create就是系统预分配了一片内存区域

供其使用。Ion在申请使用的时候，会根据当前的type来操作各自的heap->ops。分别看下三个函数：

[cpp]

```cpp
struct ion_heap *ion_system_contig_heap_create(struct ion_platform_heap *pheap)
{
    struct ion_heap *heap;

    heap = kzalloc(sizeof(struct ion_heap), GFP_KERNEL);
    if (!heap)
        return ERR_PTR(-ENOMEM);
    /*使用的是kmalloc_ops，上篇有提到哦*/
    heap->ops = &kmalloc_ops;
    heap->type = ION_HEAP_TYPE_SYSTEM_CONTIG;
    system_heap_contig_has_outer_cache = pheap->has_outer_cache;
    return heap;
}
struct ion_heap *ion_system_heap_create(struct ion_platform_heap *pheap)
{
    struct ion_heap *heap;

    heap = kzalloc(sizeof(struct ion_heap), GFP_KERNEL);
    if (!heap)
        return ERR_PTR(-ENOMEM);
    /*和上面函数的区别仅在于ops*/
    heap->ops = &vmalloc_ops;
    heap->type = ION_HEAP_TYPE_SYSTEM;
    system_heap_has_outer_cache = pheap->has_outer_cache;
    return heap;
}
struct ion_heap *ion_carveout_heap_create(struct ion_platform_heap *heap_data)
{
    struct ion_carveout_heap *carveout_heap;
    int ret;

    carveout_heap = kzalloc(sizeof(struct ion_carveout_heap), GFP_KERNEL);
    if (!carveout_heap)
        return ERR_PTR(-ENOMEM);
    /* 重新创建一个新的pool，这里有点想不通的是为什么不直接使用全局的mempools呢？*/
    carveout_heap->pool = gen_pool_create(12, -1);
    if (!carveout_heap->pool) {
        kfree(carveout_heap);
        return ERR_PTR(-ENOMEM);
    }
    carveout_heap->base = heap_data->base;
    ret = gen_pool_add(carveout_heap->pool, carveout_heap->base,
            heap_data->size, -1);
    if (ret < 0) {
        gen_pool_destroy(carveout_heap->pool);
        kfree(carveout_heap);
        return ERR_PTR(-EINVAL);
    }
    carveout_heap->heap.ops = &carveout_heap_ops;
    carveout_heap->heap.type = ION_HEAP_TYPE_CARVEOUT;
    carveout_heap->allocated_bytes = 0;
    carveout_heap->total_size = heap_data->size;
    carveout_heap->has_outer_cache = heap_data->has_outer_cache;

    if (heap_data->extra_data) {
        struct ion_co_heap_pdata *extra_data =
                heap_data->extra_data;

        if (extra_data->setup_region)
            carveout_heap->bus_id = extra_data->setup_region();
        if (extra_data->request_region)
            carveout_heap->request_region =
                    extra_data->request_region;
        if (extra_data->release_region)
            carveout_heap->release_region =
                    extra_data->release_region;
    }
    return &carveout_heap->heap;
}

Heap创建完成，然后保存到idev中：
void ion_device_add_heap(struct ion_device *dev, struct ion_heap *heap)
{
    struct rb_node **p = &dev->heaps.rb_node;
    struct rb_node *parent = NULL;
```

关闭

```
    struct ion_heap *entry;

    if (!heap->ops->allocate || !heap->ops->free || !heap->ops->map_dma ||
        !heap->ops->unmap_dma)
        pr_err("%s: can not add heap with invalid ops struct.\n",
               __func__);

    heap->dev = dev;
    mutex_lock(&dev->lock);
    while (*p) {
        parent = *p;
        entry = rb_entry(parent, struct ion_heap, node);

        if (heap->id < entry->id) {
            p = &(*p)->rb_left;
        } else if (heap->id > entry->id ) {
            p = &(*p)->rb_right;
        } else {
            pr_err("%s: can not insert multiple heaps with "
                "id %d\n", __func__, heap->id);
            goto end;
        }
    }
    /*使用红黑树保存*/
    rb_link_node(&heap->node, parent, p);
    rb_insert_color(&heap->node, &dev->heaps);
    /*以heap name创建fs,位于ion目录下。如vamlloc, camera_preview , audio 等*/
    debugfs_create_file(heap->name, 0664, dev->debug_root, heap,
                &debug_heap_fops);
end:
    mutex_unlock(&dev->lock);
}
```

[cpp]

```
struct ion_heap *ion_system_contig_heap_create(struct ion_platform_heap *pheap)
{
    struct ion_heap *heap;

    heap = kzalloc(sizeof(struct ion_heap), GFP_KERNEL);
    if (!heap)
        return ERR_PTR(-ENOMEM);
    /*使用的是kmalloc_ops，上篇有提到哦*/
    heap->ops = &kmalloc_ops;
    heap->type = ION_HEAP_TYPE_SYSTEM_CONTIG;
    system_heap_contig_has_outer_cache = pheap->has_outer_cache;
    return heap;
}
struct ion_heap *ion_system_heap_create(struct ion_platform_heap *pheap)
{
    struct ion_heap *heap;

    heap = kzalloc(sizeof(struct ion_heap), GFP_KERNEL);
    if (!heap)
        return ERR_PTR(-ENOMEM);
    /*和上面函数的区别仅在于ops*/
    heap->ops = &vmalloc_ops;
    heap->type = ION_HEAP_TYPE_SYSTEM;
    system_heap_has_outer_cache = pheap->has_outer_cache;
    return heap;
}
struct ion_heap *ion_carveout_heap_create(struct ion_platform_heap *heap_data)
{
    struct ion_carveout_heap *carveout_heap;
    int ret;

    carveout_heap = kzalloc(sizeof(struct ion_carveout_heap), GFP_KERNEL);
    if (!carveout_heap)
        return ERR_PTR(-ENOMEM);
    /* 重新创建一个新的pool，这里有点想不通的是为什么不直接使用全局的mempools呢？*/
    carveout_heap->pool = gen_pool_create(12, -1);
    if (!carveout_heap->pool) {
        kfree(carveout_heap);
        return ERR_PTR(-ENOMEM);
    }
    carveout_heap->base = heap_data->base;
    ret = gen_pool_add(carveout_heap->pool, carveout_heap->base,
            heap_data->size, -1);
    if (ret < 0) {
        gen_pool_destroy(carveout_heap->pool);
```

关闭

```
        kfree(carveout_heap);
        return ERR_PTR(-EINVAL);
    }
    carveout_heap->heap.ops = &carveout_heap_ops;
    carveout_heap->heap.type = ION_HEAP_TYPE_CARVEOUT;
    carveout_heap->allocated_bytes = 0;
    carveout_heap->total_size = heap_data->size;
    carveout_heap->has_outer_cache = heap_data->has_outer_cache;

    if (heap_data->extra_data) {
        struct ion_co_heap_pdata *extra_data =
                heap_data->extra_data;

        if (extra_data->setup_region)
            carveout_heap->bus_id = extra_data->setup_region();
        if (extra_data->request_region)
            carveout_heap->request_region =
                    extra_data->request_region;
        if (extra_data->release_region)
            carveout_heap->release_region =
                    extra_data->release_region;
    }
    return &carveout_heap->heap;
}

Heap创建完成，然后保存到idev中：
void ion_device_add_heap(struct ion_device *dev, struct ion_heap *heap)
{
    struct rb_node **p = &dev->heaps.rb_node;
    struct rb_node *parent = NULL;
    struct ion_heap *entry;

    if (!heap->ops->allocate || !heap->ops->free || !heap->ops->map_dma ||
        !heap->ops->unmap_dma)
        pr_err("%s: can not add heap with invalid ops struct.\n",
               __func__);

    heap->dev = dev;
    mutex_lock(&dev->lock);
    while (*p) {
        parent = *p;
        entry = rb_entry(parent, struct ion_heap, node);

        if (heap->id < entry->id) {
            p = &(*p)->rb_left;
        } else if (heap->id > entry->id ) {
            p = &(*p)->rb_right;
        } else {
            pr_err("%s: can not insert multiple heaps with "
                "id %d\n", __func__, heap->id);
            goto end;
        }
    }
    /*使用红黑树保存*/
    rb_link_node(&heap->node, parent, p);
    rb_insert_color(&heap->node, &dev->heaps);
    /*以heap name创建fs,位于ion目录下。如vamlloc, camera_preview , audio 等*/
    debugfs_create_file(heap->name, 0664, dev->debug_root, heap,
                &debug_heap_fops);
end:
    mutex_unlock(&dev->lock);
}
```

到此，ION初始化已经完成了。接下来该如何使用呢？嗯，通过前面创建的misc设备也就是idev了！还记得里面有个fops为ion_fops吗？先来看下用户空间如何使用ION，最后看内核空

ION用户空间使用

[cpp]

```
Ion_fops结构如下：
static const struct file_operations ion_fops = {
    .owner          = THIS_MODULE,
    .open           = ion_open,
    .release        = ion_release,
    .unlocked_ioctl = ion_ioctl,
};

用户空间都是通过ioctl来控制。先看ion_open.
```

```cpp

static int ion_open(struct inode *inode, struct file *file)
{
    struct miscdevice *miscdev = file->private_data;
    struct ion_device *dev = container_of(miscdev, struct ion_device, dev);
    struct ion_client *client;
    char debug_name[64];

    pr_debug("%s: %d\n", __func__, __LINE__);
    snprintf(debug_name, 64, "%u", task_pid_nr(current->group_leader));
    /*根据idev和task pid为name创建ion client*/
    client = ion_client_create(dev, -1, debug_name);
    if (IS_ERR_OR_NULL(client))
        return PTR_ERR(client);
    file->private_data = client;

    return 0;
}
```

[cpp]

```cpp
Ion_fops结构如下：
static const struct file_operations ion_fops = {
    .owner          = THIS_MODULE,
    .open           = ion_open,
    .release        = ion_release,
    .unlocked_ioctl = ion_ioctl,
};

用户空间都是通过ioctl来控制。先看ion_open.

static int ion_open(struct inode *inode, struct file *file)
{
    struct miscdevice *miscdev = file->private_data;
    struct ion_device *dev = container_of(miscdev, struct ion_device, dev);
    struct ion_client *client;
    char debug_name[64];

    pr_debug("%s: %d\n", __func__, __LINE__);
    snprintf(debug_name, 64, "%u", task_pid_nr(current->group_leader));
    /*根据idev和task pid为name创建ion client*/
    client = ion_client_create(dev, -1, debug_name);
    if (IS_ERR_OR_NULL(client))
        return PTR_ERR(client);
    file->private_data = client;

    return 0;
}
```

前一篇文章有说到，要使用ION, 必须要先创建ion_client, 因此用户空间在open ion的时候创建了client.

[cpp]

```cpp
struct ion_client *ion_client_create(struct ion_device *dev,
                     unsigned int heap_mask,
                     const char *name)
{
    struct ion_client *client;
    struct task_struct *task;
    struct rb_node **p;
    struct rb_node *parent = NULL;
    struct ion_client *entry;
    pid_t pid;
    unsigned int name_len;

    if (!name) {
        pr_err("%s: Name cannot be null\n", __func__);
        return ERR_PTR(-EINVAL);
    }
    name_len = strnlen(name, 64);

    get_task_struct(current->group_leader);
    task_lock(current->group_leader);
    pid = task_pid_nr(current->group_leader);
    /* don't bother to store task struct for kernel threads,
       they can't be killed anyway */
    if (current->group_leader->flags & PF_KTHREAD) {
        put_task_struct(current->group_leader);
        task = NULL;
```

关闭

```
    } else {
        task = current->group_leader;
    }
    task_unlock(current->group_leader);
    /*分配ion client struct.*/
    client = kzalloc(sizeof(struct ion_client), GFP_KERNEL);
    if (!client) {
        if (task)
            put_task_struct(current->group_leader);
        return ERR_PTR(-ENOMEM);
    }
    /*下面就是保存一系列参数了。*/
    client->dev = dev;
    client->handles = RB_ROOT;
    mutex_init(&client->lock);

    client->name = kzalloc(name_len+1, GFP_KERNEL);
    if (!client->name) {
        put_task_struct(current->group_leader);
        kfree(client);
        return ERR_PTR(-ENOMEM);
    } else {
        strlcpy(client->name, name, name_len+1);
    }

    client->heap_mask = heap_mask;
    client->task = task;
    client->pid = pid;

    mutex_lock(&dev->lock);
    p = &dev->clients.rb_node;
    while (*p) {
        parent = *p;
        entry = rb_entry(parent, struct ion_client, node);

        if (client < entry)
            p = &(*p)->rb_left;
        else if (client > entry)
            p = &(*p)->rb_right;
    }
    /*当前client添加到idev的clients根树上去。*/
    rb_link_node(&client->node, parent, p);
    rb_insert_color(&client->node, &dev->clients);

    /*在ION先创建的文件名字是以pid命名的。*/
    client->debug_root = debugfs_create_file(name, 0664,
                         dev->debug_root, client,
                         &debug_client_fops);
    mutex_unlock(&dev->lock);

    return client;
}
```

[cpp]

```
struct ion_client *ion_client_create(struct ion_device *dev,
                     unsigned int heap_mask,
                     const char *name)
{
    struct ion_client *client;
    struct task_struct *task;
    struct rb_node **p;
    struct rb_node *parent = NULL;
    struct ion_client *entry;
    pid_t pid;
    unsigned int name_len;

    if (!name) {
        pr_err("%s: Name cannot be null\n", __func__);
        return ERR_PTR(-EINVAL);
    }
    name_len = strnlen(name, 64);

    get_task_struct(current->group_leader);
    task_lock(current->group_leader);
    pid = task_pid_nr(current->group_leader);
    /* don't bother to store task struct for kernel threads,
       they can't be killed anyway */
    if (current->group_leader->flags & PF_KTHREAD) {
        put_task_struct(current->group_leader);
```

关闭

```
            task = NULL;
        } else {
            task = current->group_leader;
        }
        task_unlock(current->group_leader);
        /*分配ion client struct.*/
        client = kzalloc(sizeof(struct ion_client), GFP_KERNEL);
        if (!client) {
            if (task)
                put_task_struct(current->group_leader);
            return ERR_PTR(-ENOMEM);
        }
        /*下面就是保存一系列参数了。*/
        client->dev = dev;
        client->handles = RB_ROOT;
        mutex_init(&client->lock);

        client->name = kzalloc(name_len+1, GFP_KERNEL);
        if (!client->name) {
            put_task_struct(current->group_leader);
            kfree(client);
            return ERR_PTR(-ENOMEM);
        } else {
            strlcpy(client->name, name, name_len+1);
        }

        client->heap_mask = heap_mask;
        client->task = task;
        client->pid = pid;

        mutex_lock(&dev->lock);
        p = &dev->clients.rb_node;
        while (*p) {
            parent = *p;
            entry = rb_entry(parent, struct ion_client, node);

            if (client < entry)
                p = &(*p)->rb_left;
            else if (client > entry)
                p = &(*p)->rb_right;
        }
        /*当前client添加到idev的clients根树上去。*/
        rb_link_node(&client->node, parent, p);
        rb_insert_color(&client->node, &dev->clients);

        /*在ION先创建的文件名字是以pid命名的。*/
        client->debug_root = debugfs_create_file(name, 0664,
                             dev->debug_root, client,
                             &debug_client_fops);
        mutex_unlock(&dev->lock);

        return client;
    }
```

有了client之后，用户程序就可以开始申请分配ION buffer了！通过ioctl命令实现。

ion_ioct函数有若干个cmd，ION_IOC_ALLOC和ION_IOC_FREE相对应，表示申请和释放buffer。用户空间程序使用前先要调用ION_IOC_MAP才能得到buffer address，而ION_IOC_IMPORT是为了将这块内存共享给用户空间另一个进程。

[cpp]

```
static long ion_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    struct ion_client *client = filp->private_data;

    switch (cmd) {
    case ION_IOC_ALLOC:
    {
        struct ion_allocation_data data;

        if (copy_from_user(&data, (void __user *)arg, sizeof(data)))
            return -EFAULT;
        /*分配buffer.*/
        data.handle = ion_alloc(client, data.len, data.align,
                         data.flags);

        if (IS_ERR(data.handle))
            return PTR_ERR(data.handle);
```

关闭

```

        if (copy_to_user((void __user *)arg, &data, sizeof(data))) {
            ion_free(client, data.handle);
            return -EFAULT;
        }
        break;
    }
    case ION_IOC_FREE:
    {
        struct ion_handle_data data;
        bool valid;

        if (copy_from_user(&data, (void __user *)arg,
                   sizeof(struct ion_handle_data)))
            return -EFAULT;
        mutex_lock(&client->lock);
        valid = ion_handle_validate(client, data.handle);
        mutex_unlock(&client->lock);
        if (!valid)
            return -EINVAL;
        ion_free(client, data.handle);
        break;
    }
    case ION_IOC_MAP:
    case ION_IOC_SHARE:
    {
        struct ion_fd_data data;
        int ret;
        if (copy_from_user(&data, (void __user *)arg, sizeof(data)))
            return -EFAULT;
        /*判断当前cmd是否被调用过了，调用过就返回，否则设置flags.*/
        ret = ion_share_set_flags(client, data.handle, filp->f_flags);
        if (ret)
            return ret;

        data.fd = ion_share_dma_buf(client, data.handle);
        if (copy_to_user((void __user *)arg, &data, sizeof(data)))
            return -EFAULT;
        if (data.fd < 0)
            return data.fd;
        break;
    }
    case ION_IOC_IMPORT:
    {
        struct ion_fd_data data;
        int ret = 0;
        if (copy_from_user(&data, (void __user *)arg,
                   sizeof(struct ion_fd_data)))
            return -EFAULT;
        data.handle = ion_import_dma_buf(client, data.fd);
        if (IS_ERR(data.handle))
            data.handle = NULL;
        if (copy_to_user((void __user *)arg, &data,
                 sizeof(struct ion_fd_data)))
            return -EFAULT;
        if (ret < 0)
            return ret;
        break;
    }
    case ION_IOC_CUSTOM:
~~snip
    case ION_IOC_CLEAN_CACHES:
    case ION_IOC_INV_CACHES:
    case ION_IOC_CLEAN_INV_CACHES:
    ~~snip
    case ION_IOC_GET_FLAGS:
~~snip
    default:
        return -ENOTTY;
    }
    return 0;
}
```

关闭

[cpp]

```
static long ion_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    struct ion_client *client = filp->private_data;

    switch (cmd) {
```

```
    case ION_IOC_ALLOC:
    {
        struct ion_allocation_data data;

        if (copy_from_user(&data, (void __user *)arg, sizeof(data)))
            return -EFAULT;
        /*分配buffer.*/
        data.handle = ion_alloc(client, data.len, data.align,
                         data.flags);

        if (IS_ERR(data.handle))
            return PTR_ERR(data.handle);

        if (copy_to_user((void __user *)arg, &data, sizeof(data))) {
            ion_free(client, data.handle);
            return -EFAULT;
        }
        break;
    }
    case ION_IOC_FREE:
    {
        struct ion_handle_data data;
        bool valid;

        if (copy_from_user(&data, (void __user *)arg,
                   sizeof(struct ion_handle_data)))
            return -EFAULT;
        mutex_lock(&client->lock);
        valid = ion_handle_validate(client, data.handle);
        mutex_unlock(&client->lock);
        if (!valid)
            return -EINVAL;
        ion_free(client, data.handle);
        break;
    }
    case ION_IOC_MAP:
    case ION_IOC_SHARE:
    {
        struct ion_fd_data data;
        int ret;
        if (copy_from_user(&data, (void __user *)arg, sizeof(data)))
            return -EFAULT;
        /*判断当前cmd是否被调用过了，调用过就返回，否则设置flags.*/
        ret = ion_share_set_flags(client, data.handle, filp->f_flags);
        if (ret)
            return ret;

        data.fd = ion_share_dma_buf(client, data.handle);
        if (copy_to_user((void __user *)arg, &data, sizeof(data)))
            return -EFAULT;
        if (data.fd < 0)
            return data.fd;
        break;
    }
    case ION_IOC_IMPORT:
    {
        struct ion_fd_data data;
        int ret = 0;
        if (copy_from_user(&data, (void __user *)arg,
                   sizeof(struct ion_fd_data)))
            return -EFAULT;
        data.handle = ion_import_dma_buf(client, data.fd);
        if (IS_ERR(data.handle))
            data.handle = NULL;
        if (copy_to_user((void __user *)arg, &data,
                 sizeof(struct ion_fd_data)))
            return -EFAULT;
        if (ret < 0)
            return ret;
        break;
    }
    case ION_IOC_CUSTOM:
~~snip
    case ION_IOC_CLEAN_CACHES:
    case ION_IOC_INV_CACHES:
    case ION_IOC_CLEAN_INV_CACHES:
    ~~snip
    case ION_IOC_GET_FLAGS:
~~snip
    default:
```

关闭

```
        return -ENOTTY;
    }
    return 0;
}
```

下面分小节说明分配和共享的原理。

ION_IOC_ALLOC

[cpp]

```
struct ion_handle *ion_alloc(struct ion_client *client, size_t len,
                 size_t align, unsigned int flags)
{
~~snip

    mutex_lock(&dev->lock);
    /*循环遍历当前Heap链表。*/
    for (n = rb_first(&dev->heaps); n != NULL; n = rb_next(n)) {
        struct ion_heap *heap = rb_entry(n, struct ion_heap, node);
/*只有heap type和id都符合才去创建buffer.*/
        /* if the client doesn't support this heap type */
        if (!((1 << heap->type) & client->heap_mask))
            continue;
        /* if the caller didn't specify this heap type */
        if (!((1 << heap->id) & flags))
            continue;
        /* Do not allow un-secure heap if secure is specified */
        if (secure_allocation && (heap->type != ION_HEAP_TYPE_CP))
            continue;
        buffer = ion_buffer_create(heap, dev, len, align, flags);
~~snip
    }
    mutex_unlock(&dev->lock);

~~snip
    /*创建了buffer之后，就相应地创建handle来管理buffer.*/
    handle = ion_handle_create(client, buffer);

~~snip
}

找到Heap之后调用ion_buffer_create：
static struct ion_buffer *ion_buffer_create(struct ion_heap *heap,
                     struct ion_device *dev,
                     unsigned long len,
                     unsigned long align,
                     unsigned long flags)
{
    struct ion_buffer *buffer;
    struct sg_table *table;
    int ret;
    /*分配struct ion buffer,用来管理buffer.*/
    buffer = kzalloc(sizeof(struct ion_buffer), GFP_KERNEL);
    if (!buffer)
        return ERR_PTR(-ENOMEM);

    buffer->heap = heap;
    kref_init(&buffer->ref);
    /*调用相应heap type的ops allocate。还记得前面有提到过不同种类的ops吗，
如carveout_heap_ops ，vmalloc_ops 。*/
    ret = heap->ops->allocate(heap, buffer, len, align, flags);
    if (ret) {
        kfree(buffer);
        return ERR_PTR(ret);
    }

    buffer->dev = dev;
    buffer->size = len;
    /*http://lwn.net/Articles/263343/*/
    table = buffer->heap->ops->map_dma(buffer->heap, buffer);
    if (IS_ERR_OR_NULL(table)) {
        heap->ops->free(buffer);
        kfree(buffer);
        return ERR_PTR(PTR_ERR(table));
    }
    buffer->sg_table = table;

    mutex_init(&buffer->lock);
```

关闭

```
    /*将当前ion buffer添加到idev 的buffers 树上统一管理。*/
    ion_buffer_add(dev, buffer);
    return buffer;
}
```

[cpp]

```cpp
struct ion_handle *ion_alloc(struct ion_client *client, size_t len,
                 size_t align, unsigned int flags)
{
~~snip

    mutex_lock(&dev->lock);
    /*循环遍历当前Heap链表。*/
    for (n = rb_first(&dev->heaps); n != NULL; n = rb_next(n)) {
        struct ion_heap *heap = rb_entry(n, struct ion_heap, node);
/*只有heap type和id都符合才去创建buffer.*/
        /* if the client doesn't support this heap type */
        if (!((1 << heap->type) & client->heap_mask))
            continue;
        /* if the caller didn't specify this heap type */
        if (!((1 << heap->id) & flags))
            continue;
        /* Do not allow un-secure heap if secure is specified */
        if (secure_allocation && (heap->type != ION_HEAP_TYPE_CP))
            continue;
        buffer = ion_buffer_create(heap, dev, len, align, flags);
~~snip
    }
    mutex_unlock(&dev->lock);

~~snip
    /*创建了buffer之后，就相应地创建handle来管理buffer.*/
    handle = ion_handle_create(client, buffer);

~~snip
}

找到Heap之后调用ion_buffer_create：
static struct ion_buffer *ion_buffer_create(struct ion_heap *heap,
                     struct ion_device *dev,
                     unsigned long len,
                     unsigned long align,
                     unsigned long flags)
{
    struct ion_buffer *buffer;
    struct sg_table *table;
    int ret;
    /*分配struct ion buffer,用来管理buffer.*/
    buffer = kzalloc(sizeof(struct ion_buffer), GFP_KERNEL);
    if (!buffer)
        return ERR_PTR(-ENOMEM);

    buffer->heap = heap;
    kref_init(&buffer->ref);
    /*调用相应heap type的ops allocate。还记得前面有提到过不同种类的ops吗，
如carveout_heap_ops ，vmalloc_ops 。*/
    ret = heap->ops->allocate(heap, buffer, len, align, flags);
    if (ret) {
        kfree(buffer);
        return ERR_PTR(ret);
    }

    buffer->dev = dev;
    buffer->size = len;
    /*http://lwn.net/Articles/263343/*/
    table = buffer->heap->ops->map_dma(buffer->heap, buffer);
    if (IS_ERR_OR_NULL(table)) {
        heap->ops->free(buffer);
        kfree(buffer);
        return ERR_PTR(PTR_ERR(table));
    }
    buffer->sg_table = table;

    mutex_init(&buffer->lock);
    /*将当前ion buffer添加到idev 的buffers 树上统一管理。*/
    ion_buffer_add(dev, buffer);
    return buffer;
}
```

关闭

[cpp]

```cpp
<p>static struct ion_handle *ion_handle_create(struct ion_client *client,
         struct ion_buffer *buffer)
{
 struct ion_handle *handle;
 /*分配struct ion_handle.*/
 handle = kzalloc(sizeof(struct ion_handle), GFP_KERNEL);
 if (!handle)
  return ERR_PTR(-ENOMEM);
 kref_init(&handle->ref);
 rb_init_node(&handle->node);
 handle->client = client; //client放入handle中
 ion_buffer_get(buffer); //引用计数加1
 handle->buffer = buffer; //buffer也放入handle中</p><p> return handle;
}
</p>
```

[cpp]

```cpp
<p>static struct ion_handle *ion_handle_create(struct ion_client *client,
         struct ion_buffer *buffer)
{
 struct ion_handle *handle;
 /*分配struct ion_handle.*/
 handle = kzalloc(sizeof(struct ion_handle), GFP_KERNEL);
 if (!handle)
  return ERR_PTR(-ENOMEM);
 kref_init(&handle->ref);
 rb_init_node(&handle->node);
 handle->client = client; //client放入handle中
 ion_buffer_get(buffer); //引用计数加1
 handle->buffer = buffer; //buffer也放入handle中</p><p> return handle;
}
</p>
```

先拿heap type为ION_HEAP_TYPE_CARVEOUT为例，看下它是如何分配buffer的。

allocate对应ion_carveout_heap_allocate。

[cpp]

```cpp
static int ion_carveout_heap_allocate(struct ion_heap *heap,
                      struct ion_buffer *buffer,
                      unsigned long size, unsigned long align,
                      unsigned long flags)
{
    buffer->priv_phys = ion_carveout_allocate(heap, size, align);
    return buffer->priv_phys == ION_CARVEOUT_ALLOCATE_FAIL ? -ENOMEM : 0;
}
ion_phys_addr_t ion_carveout_allocate(struct ion_heap *heap,
                      unsigned long size,
                      unsigned long align)
{
    struct ion_carveout_heap *carveout_heap =
        container_of(heap, struct ion_carveout_heap, heap);
    /*通过创建的mem pool来管理buffer,由于这块buffer在初始化的
时候就预留了，现在只要从上面拿一块区域就可以了。*/
    unsigned long offset = gen_pool_alloc_aligned(carveout_heap->pool,
                            size, ilog2(align));
    /*分配不成功可能是没有内存空间可供分配了或者是有碎片导致的。*/
    if (!offset) {
        if ((carveout_heap->total_size -
              carveout_heap->allocated_bytes) >= size)
            pr_debug("%s: heap %s has enough memory (%lx) but"
                " the allocation of size %lx still fa
                " Memory is probably fragmented.",
                __func__, heap->name,
                carveout_heap->total_size -
                carveout_heap->allocated_bytes, size);
        return ION_CARVEOUT_ALLOCATE_FAIL;
    }
    /*已经分配掉的内存字节。*/
    carveout_heap->allocated_bytes += size;
    return offset;
}
```

关闭

[cpp]

```cpp
static int ion_carveout_heap_allocate(struct ion_heap *heap,
                      struct ion_buffer *buffer,
                      unsigned long size, unsigned long align,
                      unsigned long flags)
{
    buffer->priv_phys = ion_carveout_allocate(heap, size, align);
    return buffer->priv_phys == ION_CARVEOUT_ALLOCATE_FAIL ? -ENOMEM : 0;
}
ion_phys_addr_t ion_carveout_allocate(struct ion_heap *heap,
                      unsigned long size,
                      unsigned long align)
{
    struct ion_carveout_heap *carveout_heap =
        container_of(heap, struct ion_carveout_heap, heap);
    /*通过创建的mem pool来管理buffer,由于这块buffer在初始化的
时候就预留了，现在只要从上面拿一块区域就可以了。*/
    unsigned long offset = gen_pool_alloc_aligned(carveout_heap->pool,
                            size, ilog2(align));
    /*分配不成功可能是没有内存空间可供分配了或者是有碎片导致的。*/
    if (!offset) {
        if ((carveout_heap->total_size -
              carveout_heap->allocated_bytes) >= size)
            pr_debug("%s: heap %s has enough memory (%lx) but"
                " the allocation of size %lx still failed."
                " Memory is probably fragmented.",
                __func__, heap->name,
                carveout_heap->total_size -
                carveout_heap->allocated_bytes, size);
        return ION_CARVEOUT_ALLOCATE_FAIL;
    }
    /*已经分配掉的内存字节。*/
    carveout_heap->allocated_bytes += size;
    return offset;
}
```

同样地，对于heap type为ION_HEAP_TYPE_SYSTEM的分配函数是ion_system_heap_allocate。

[cpp]

```
static int ion_system_contig_heap_allocate(struct ion_heap *heap,
                       struct ion_buffer *buffer,
                       unsigned long len,
                       unsigned long align,
                       unsigned long flags)
{
    /*通过kzalloc分配。*/
    buffer->priv_virt = kzalloc(len, GFP_KERNEL);
    if (!buffer->priv_virt)
        return -ENOMEM;
    atomic_add(len, &system_contig_heap_allocated);
    return 0;
}
```

[cpp]

```
static int ion_system_contig_heap_allocate(struct ion_heap *heap,
                       struct ion_buffer *buffer,
                       unsigned long len,
                       unsigned long align,
                       unsigned long flags)
{
    /*通过kzalloc分配。*/
    buffer->priv_virt = kzalloc(len, GFP_KERNEL);
    if (!buffer->priv_virt)
        return -ENOMEM;
    atomic_add(len, &system_contig_heap_allocated);
    return 0;
}
```

关闭

其他的几种Heap type可自行研究，接着调用ion_buffer_add将buffer添加到dev的buffers树上去

[cpp]

```
static void ion_buffer_add(struct ion_device *dev,
               struct ion_buffer *buffer)
{
    struct rb_node **p = &dev->buffers.rb_node;
    struct rb_node *parent = NULL;
    struct ion_buffer *entry;

    while (*p) {
        parent = *p;
        entry = rb_entry(parent, struct ion_buffer, node);

        if (buffer < entry) {
            p = &(*p)->rb_left;
        } else if (buffer > entry) {
            p = &(*p)->rb_right;
        } else {
            pr_err("%s: buffer already found.", __func__);
            BUG();
        }
    }
/*又是使用红黑树哦！*/
    rb_link_node(&buffer->node, parent, p);
    rb_insert_color(&buffer->node, &dev->buffers);
}
```

[cpp]

```
static void ion_buffer_add(struct ion_device *dev,
               struct ion_buffer *buffer)
{
    struct rb_node **p = &dev->buffers.rb_node;
    struct rb_node *parent = NULL;
    struct ion_buffer *entry;

    while (*p) {
        parent = *p;
        entry = rb_entry(parent, struct ion_buffer, node);

        if (buffer < entry) {
            p = &(*p)->rb_left;
        } else if (buffer > entry) {
            p = &(*p)->rb_right;
        } else {
            pr_err("%s: buffer already found.", __func__);
            BUG();
        }
    }
/*又是使用红黑树哦！*/
    rb_link_node(&buffer->node, parent, p);
    rb_insert_color(&buffer->node, &dev->buffers);
}
```

至此，已经得到client和handle，buffer分配完成！

ION_IOC_MAP/ ION_IOC_SHARE

[cpp]

```
int ion_share_dma_buf(struct ion_client *client, struct ion_handle *handle)
{
    struct ion_buffer *buffer;
    struct dma_buf *dmabuf;
    bool valid_handle;
    int fd;

    mutex_lock(&client->lock);
    valid_handle = ion_handle_validate(client, handle);
    mutex_unlock(&client->lock);
    if (!valid_handle) {
        WARN(1, "%s: invalid handle passed to share.\n", __func__);
        return -EINVAL;
    }

    buffer = handle->buffer;
    ion_buffer_get(buffer);
    /*生成一个新的file描述符*/
    dmabuf = dma_buf_export(buffer, &dma_buf_ops, buffer->size, O_RDWR);
    if (IS_ERR(dmabuf)) {
        ion_buffer_put(buffer);
        return PTR_ERR(dmabuf);
    }
    /*将file转换用户空间识别的fd描述符。*/
    fd = dma_buf_fd(dmabuf, O_CLOEXEC);
    if (fd < 0)
        dma_buf_put(dmabuf);

    return fd;
```

关闭

```
}
struct dma_buf *dma_buf_export(void *priv, const struct dma_buf_ops *ops,
                size_t size, int flags)
{
    struct dma_buf *dmabuf;
    struct file *file;
~~snip
    /*分配struct dma_buf.*/
    dmabuf = kzalloc(sizeof(struct dma_buf), GFP_KERNEL);
    if (dmabuf == NULL)
        return ERR_PTR(-ENOMEM);
    /*保存信息到dmabuf，注意ops为dma_buf_ops，后面mmap为调用到。*/
    dmabuf->priv = priv;
    dmabuf->ops = ops;
    dmabuf->size = size;
    /*产生新的file*/
    file = anon_inode_getfile("dmabuf", &dma_buf_fops, dmabuf, flags);

    dmabuf->file = file;

    mutex_init(&dmabuf->lock);
    INIT_LIST_HEAD(&dmabuf->attachments);

    return dmabuf;
}
```

[cpp]

```
int ion_share_dma_buf(struct ion_client *client, struct ion_handle *handle)
{
    struct ion_buffer *buffer;
    struct dma_buf *dmabuf;
    bool valid_handle;
    int fd;

    mutex_lock(&client->lock);
    valid_handle = ion_handle_validate(client, handle);
    mutex_unlock(&client->lock);
    if (!valid_handle) {
        WARN(1, "%s: invalid handle passed to share.\n", __func__);
        return -EINVAL;
    }

    buffer = handle->buffer;
    ion_buffer_get(buffer);
    /*生成一个新的file描述符*/
    dmabuf = dma_buf_export(buffer, &dma_buf_ops, buffer->size, O_RDWR);
    if (IS_ERR(dmabuf)) {
        ion_buffer_put(buffer);
        return PTR_ERR(dmabuf);
    }
    /*将file转换用户空间识别的fd描述符。*/
    fd = dma_buf_fd(dmabuf, O_CLOEXEC);
    if (fd < 0)
        dma_buf_put(dmabuf);

    return fd;
}
struct dma_buf *dma_buf_export(void *priv, const struct dma_buf_ops *ops,
                size_t size, int flags)
{
    struct dma_buf *dmabuf;
    struct file *file;
~~snip
    /*分配struct dma_buf.*/
    dmabuf = kzalloc(sizeof(struct dma_buf), GFP_KER
    if (dmabuf == NULL)
        return ERR_PTR(-ENOMEM);
    /*保存信息到dmabuf，注意ops为dma_buf_ops，后面mmap为调用到。*/
    dmabuf->priv = priv;
    dmabuf->ops = ops;
    dmabuf->size = size;
    /*产生新的file*/
    file = anon_inode_getfile("dmabuf", &dma_buf_fops, dmabuf, flags);

    dmabuf->file = file;

    mutex_init(&dmabuf->lock);
    INIT_LIST_HEAD(&dmabuf->attachments);

    return dmabuf;
}
```

通过上述过程，用户空间就得到了新的fd,重新生成一个新的fd的目的是考虑了两个用户空间进程想共享这块heap内存的情况。然后再对fd作mmap，相应地kernel空间就调用到了file 的dma_buf_fops中的dma_buf_mmap_internal。

[cpp]

```cpp
static const struct file_operations dma_buf_fops = {
    .release    = dma_buf_release,
    .mmap       = dma_buf_mmap_internal,
};
static int dma_buf_mmap_internal(struct file *file, struct vm_area_struct *
 {
    struct dma_buf *dmabuf;

    if (!is_dma_buf_file(file))
        return -EINVAL;

    dmabuf = file->private_data;
    /*检查用户空间要映射的size是否比目前dmabuf也就是当前heap的size
还要大，如果是就返回无效。*/
    /* check for overflowing the buffer's size */
    if (vma->vm_pgoff + ((vma->vm_end - vma->vm_start) >> PAGE_SHIFT) >
        dmabuf->size >> PAGE_SHIFT)
        return -EINVAL;
    /*调用的是dma_buf_ops 的mmap函数*/
    return dmabuf->ops->mmap(dmabuf, vma);
 }

struct dma_buf_ops dma_buf_ops = {
    .map_dma_buf = ion_map_dma_buf,
    .unmap_dma_buf = ion_unmap_dma_buf,
    .mmap = ion_mmap,
    .release = ion_dma_buf_release,
    .begin_cpu_access = ion_dma_buf_begin_cpu_access,
    .end_cpu_access = ion_dma_buf_end_cpu_access,
    .kmap_atomic = ion_dma_buf_kmap,
    .kunmap_atomic = ion_dma_buf_kunmap,
    .kmap = ion_dma_buf_kmap,
    .kunmap = ion_dma_buf_kunmap,
};
static int ion_mmap(struct dma_buf *dmabuf, struct vm_area_struct *vma)
{
    struct ion_buffer *buffer = dmabuf->priv;
    int ret;

    if (!buffer->heap->ops->map_user) {
        pr_err("%s: this heap does not define a method for mapping "
               "to userspace\n", __func__);
        return -EINVAL;
    }

    mutex_lock(&buffer->lock);
    /* now map it to userspace */
    /*调用的是相应heap的map_user，如carveout_heap_ops 调用的是
ion_carveout_heap_map_user ，此函数就是一般的mmap实现，不追下去了。*/
    ret = buffer->heap->ops->map_user(buffer->heap, buffer, vma);

    if (ret) {
        mutex_unlock(&buffer->lock);
        pr_err("%s: failure mapping buffer to userspace\n",
               __func__);
    } else {
        buffer->umap_cnt++;
        mutex_unlock(&buffer->lock);

        vma->vm_ops = &ion_vm_ops;
        /*
         * move the buffer into the vm_private_data so we can access it
         * from vma_open/close
         */
        vma->vm_private_data = buffer;
    }
    return ret;
}
```

关闭

[cpp]

```cpp
static const struct file_operations dma_buf_fops = {
    .release    = dma_buf_release,
    .mmap       = dma_buf_mmap_internal,
};
static int dma_buf_mmap_internal(struct file *file, struct vm_area_struct *vma)
 {
    struct dma_buf *dmabuf;

    if (!is_dma_buf_file(file))
        return -EINVAL;

    dmabuf = file->private_data;
    /*检查用户空间要映射的size是否比目前dmabuf也就是当前heap的size
还要大，如果是就返回无效。*/
    /* check for overflowing the buffer's size */
    if (vma->vm_pgoff + ((vma->vm_end - vma->vm_start) >> PAGE_SHIFT) >
        dmabuf->size >> PAGE_SHIFT)
        return -EINVAL;
    /*调用的是dma_buf_ops 的mmap函数*/
    return dmabuf->ops->mmap(dmabuf, vma);
 }

struct dma_buf_ops dma_buf_ops = {
    .map_dma_buf = ion_map_dma_buf,
    .unmap_dma_buf = ion_unmap_dma_buf,
    .mmap = ion_mmap,
    .release = ion_dma_buf_release,
    .begin_cpu_access = ion_dma_buf_begin_cpu_access,
    .end_cpu_access = ion_dma_buf_end_cpu_access,
    .kmap_atomic = ion_dma_buf_kmap,
    .kunmap_atomic = ion_dma_buf_kunmap,
    .kmap = ion_dma_buf_kmap,
    .kunmap = ion_dma_buf_kunmap,
};
static int ion_mmap(struct dma_buf *dmabuf, struct vm_area_struct *vma)
{
    struct ion_buffer *buffer = dmabuf->priv;
    int ret;

    if (!buffer->heap->ops->map_user) {
        pr_err("%s: this heap does not define a method for mapping "
               "to userspace\n", __func__);
        return -EINVAL;
    }

    mutex_lock(&buffer->lock);
    /* now map it to userspace */
    /*调用的是相应heap的map_user，如carveout_heap_ops 调用的是
ion_carveout_heap_map_user ，此函数就是一般的mmap实现，不追下去了。*/
    ret = buffer->heap->ops->map_user(buffer->heap, buffer, vma);

    if (ret) {
        mutex_unlock(&buffer->lock);
        pr_err("%s: failure mapping buffer to userspace\n",
               __func__);
    } else {
        buffer->umap_cnt++;
        mutex_unlock(&buffer->lock);

        vma->vm_ops = &ion_vm_ops;
        /*
         * move the buffer into the vm_private_data so we can access it
         * from vma_open/close
         */
        vma->vm_private_data = buffer;
    }
    return ret;
}
```

关闭

至此，用户空间就得到了bufferaddress，然后可以使用了！

ION_IOC_IMPORT

当用户空间另一个进程需要这块heap的时候，ION_IOC_IMPORT就派上用处了！注意，传进去的fd为在ION_IOC_SHARE中得到的。

[cpp]

```cpp
struct ion_handle *ion_import_dma_buf(struct ion_client *client, int fd)
{

    struct dma_buf *dmabuf;
    struct ion_buffer *buffer;
    struct ion_handle *handle;

    dmabuf = dma_buf_get(fd);
    if (IS_ERR_OR_NULL(dmabuf))
        return ERR_PTR(PTR_ERR(dmabuf));
    /* if this memory came from ion */
~~snip
    buffer = dmabuf->priv;

    mutex_lock(&client->lock);
    /* if a handle exists for this buffer just take a reference to it */
/*查找是否已经存在对应的handle了，没有则创建。因为另外一个进程只是
调用了open 接口，对应的只创建了client，并没有handle。
*/
    handle = ion_handle_lookup(client, buffer);
    if (!IS_ERR_OR_NULL(handle)) {
        ion_handle_get(handle);
        goto end;
    }
    handle = ion_handle_create(client, buffer);
    if (IS_ERR_OR_NULL(handle))
        goto end;
    ion_handle_add(client, handle);
end:
    mutex_unlock(&client->lock);
    dma_buf_put(dmabuf);
    return handle;
}
```

[cpp]

```cpp
struct ion_handle *ion_import_dma_buf(struct ion_client *client, int fd)
{

    struct dma_buf *dmabuf;
    struct ion_buffer *buffer;
    struct ion_handle *handle;

    dmabuf = dma_buf_get(fd);
    if (IS_ERR_OR_NULL(dmabuf))
        return ERR_PTR(PTR_ERR(dmabuf));
    /* if this memory came from ion */
~~snip
    buffer = dmabuf->priv;

    mutex_lock(&client->lock);
    /* if a handle exists for this buffer just take a reference to it */
/*查找是否已经存在对应的handle了，没有则创建。因为另外一个进程只是
调用了open 接口，对应的只创建了client，并没有handle。
*/
    handle = ion_handle_lookup(client, buffer);
    if (!IS_ERR_OR_NULL(handle)) {
        ion_handle_get(handle);
        goto end;
    }
    handle = ion_handle_create(client, buffer);
    if (IS_ERR_OR_NULL(handle))
        goto end;
    ion_handle_add(client, handle);
end:
    mutex_unlock(&client->lock);
    dma_buf_put(dmabuf);
    return handle;
}
```

关闭

这样，用户空间另一个进程也得到了对应的bufferHandle，client/buffer/handle之间连接起来了！然后另一个一个进程就也可以使用mmap来操作这块heap buffer了。

和一般的进程使用ION区别就是共享的进程之间struction_buffer是共享的，而struct ion_handle是各自的。

可见，ION的使用流程还是比较清晰的。不过要记得的是，使用好了ION，一定要释放掉，否则会导致内存泄露。

ION内核空间使用

内核空间使用ION也是大同小异，按照创建client,buffer,handle的流程，只是它的使用对用户空间来说是透明的罢

了！

ion_client_create在kernel空间被Qualcomm给封装了下。

[cpp]

```cpp
struct ion_client *msm_ion_client_create(unsigned int heap_mask,
                    const char *name)
{
    return ion_client_create(idev, heap_mask, name);
}
```

[cpp]

```cpp
struct ion_client *msm_ion_client_create(unsigned int heap_mask,
                    const char *name)
{
    return ion_client_create(idev, heap_mask, name);
}
```

调用的流程也类似，不过map的时候调用的是heap对应的map_kernel()而不是map_user().

msm_ion_client_create -> ion_alloc ->ion_map_kernel

参考文档：

http://lwn.net/Articles/480055/

《ARM体系结构与编程》存储系统章节。

顶 0　　踩 0

上一篇　Camera服务之--Service

下一篇　Android学习之ION memory manager

相关文章推荐

- ION基本概念介绍和原理分析
- 轻松拿下Linux进程、线程和调度
- Bayesian Network 基本概念和原理
- 30天掌握机器学习升级版
- 2016计算机组成原理研究生入学统考基本概念及...
- Python网络爬虫快速入门实战
- 深入研究Windows内部原理系列之四：Windows...
- 最适合自学的C++基础知识
- 嵌入式资料---嵌入式基本概念原理和入门知识
- 一招学会Android自定义控件
- gprs基本原理GPRS基本概念问答
- 从零练就iOS高手
- RF射频知识基本概念及DTD无线产品介绍
- ION基本概念介绍
- ATM技术 基本概念 协议原理 业务类型
- SAP：销售与分销(介绍SAP的SD模块的基本概念...

查看评论

关闭

暂无评论

该文章已被禁止评论！

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

关闭