libmemkind探究（二）——jemalloc的内存分配规律以及size_classes.sh的修改

终于解开了libmemkind是如何利用jemalloc在NVM上分配内存的谜团了~那么就要开始考虑正事了：如何无侵入地（至多浅层侵入地）实现mallocat()功能。

mallocat()功能在之前已经阐述过很多遍了。有《面向非易失存储器（NVM）的内存分配器——libnvmalloc》中诞生的独立的分配器libnvmalloc、也有《libmallocat——让任意内存分配器支持malloc_at()操作》中诞生的低效万金油libmallocat。前者以安全为由被拒，后者则是因为效率实在太低下而被弃用。所以大半年的努力全部得重新开始。幸好这一年里我功力增进不少，如今有能力实现最初的想法了——针对jemalloc实现mallocat()。

但这一篇博客暂时还没这么快直入主题，先来探究一下jemalloc在分配内存时有什么规律，才能解释清楚mallocat()的设计思路。

使用的工具还是基于《libmemkind探究（一）——让jemalloc管理指定的空间》里面的test.c，不过稍微修改了一下，增加了交互式的功能：

test.c

```
#include <stdio.h>
#include <assert.h>
#include <jemalloc.h>
#include <sys/mman.h>

#define PAGE_SIZE 4096

// 待管理的空间大小
size_t mem_size = 1 << 30;
// 待管理的空间的首地址
void* base;
// 已经分配出去的大小
size_t allocated = 0;

//+++++++++++++++++++++++++++++HOOK BEGIN++++++++++++++++++++++++++++-

// 从这里开始定义jemalloc管理chunk的hook函数，自定义chunk管理行为
// 可以参考jemalloc/doc/jemalloc.html中arena.<i>.chunk_hooks一段

// 当jemalloc发现chunk不够用了，会callback此函数索要空间
// chunk大小在编译时配置（原版jemalloc-4.0.3默认人2M，libmemkind配置为4M）
void* my_chunk_alloc(void *chunk, size_t size, size_t alignment, bool
{
    printf("my_chunk_alloc(chunk = %p, size = %lu, alignment = %lu)\n'
```

```
    if(size % alignment)
        return NULL;
    if(allocated + size > mem_size)
        return NULL;
    if(chunk && chunk != base + allocated)
        return NULL;
    void* addr = base + allocated;
    allocated += size;
    return addr;
}

// 返回true表示该内存可以继续使用
bool my_chunk_dalloc(void *chunk, size_t size, bool commited, unsignec
{
    return true;
}

// 返回false表示内存充足
bool my_chunk_commit(void *chunk, size_t size, size_t offset, size_t l
{
    return false;
}

// 返回true表示该内存即使释放了，也是与物理内存对应的，可以重用
bool my_chunk_decommit(void *chunk, size_t size, size_t offset, size_t
{
    return true;
}

// 返回true表示该段地址空间被重用后不会清空
bool my_chunk_purge(void *chunk, size_t size, size_t offset, size_t le
{
    return true;
}

bool my_chunk_split(void *chunk, size_t size, size_t size_a, size_t si
{
    return false;
}

bool my_chunk_merge(void *chunk_a, size_t size_a, void *chunk_b, size_
{
    return false;
}

chunk_hooks_t my_chunk_hooks =
{
    .alloc = my_chunk_alloc,
    .dalloc = my_chunk_dalloc,
    .commit = my_chunk_commit,
    .decommit = my_chunk_decommit,
    .purge = my_chunk_purge,
    .split = my_chunk_split,
```

```
    .merge = my_chunk_merge,
};
//-----------------------------HOOK END-----------------------------

int main()
{
    // 模拟一段NVM空间，或者任何一段用户待管理的空间（这里有1GB)
    base = mmap(0, mem_size, PROT_READ | PROT_WRITE, MAP_ANONYMOUS | M
    if(base == MAP_FAILED)
    {
        printf("mmap() failed!\n");
        return 1;
    }
    printf("base = %p\n", base);

    // 用je_mallctl('arenas.extend')命令创建一个arean,
    // 参考jemalloc/doc/jemalloc.html中arenas.extend一段
    unsigned arena_index;
    size_t unsigned_size = sizeof(unsigned int);
    if(je_mallctl("arenas.extend", (void*)&arena_index, &unsigned_size
    {
        printf("je_mallctl('arenas.extend') failed!\n");
        return 1;
    }
    printf("arena_index = %u\n", arena_index);

    //为这个arena绑定我们自定义的chunk hook，于是该arena就会按我们的方式去
    // 参考jemalloc/doc/jemalloc.html中arena.<i>.chunk_hooks一段
    char cmd[64];
    sprintf(cmd, "arena.%u.chunk_hooks", arena_index);
    if(je_mallctl(cmd, NULL, NULL, (void*)&my_chunk_hooks, sizeof(chur
    {
        printf("je_mallctl('%s') failed!\n", cmd);
        return 1;
    }

    // 接着试试分配内存，使用je_mallocx()、je_callocx()等以x结尾的函数，指
    // 那么就会在我们创建的arena中分配内存
    while(true)
    {
        printf("input size and count to allocate (size = 0 or count =
        size_t size, count;
        if(scanf("%lu %lu", &size, &count) != 2)
            continue;
        if(!size || !count)
            break;
        for(size_t i = 0; i < count; i++)
        {
            void* ptr = je_mallocx(size, MALLOCX_ARENA(arena_index));
            if(!ptr)
            {
                printf("je_mallocx(%lu, MALLOCX_ARENA(%u)) failed\n",
                break;
```

```
            }
            assert(ptr >= base);
            // 算一算偏移量
            size_t offset = ptr - base;
            // 是待管理空间的第几个page
            size_t page_id = offset / PAGE_SIZE;
            // page内偏移多少
            size_t offset_in_page = offset % PAGE_SIZE;
            // 实际分配大小
            size_t usable_size = je_malloc_usable_size(ptr);
            printf("+++size = %lu, usable_size = %lu, offset = %lu, pa
                size, usable_size, offset, page_id, offset_in_page);
        }
    }

    if(munmap(base , mem_size) != 0)
    {
        printf("munmap() failed!\n");
        return 1;
    }
    return 0;
}
```

接着编译，运行：

```
gcc -std=gnu99 test.c -o test -I../jemalloc-4.0.3/include/jemalloc -L.
export LD_LIBRARY_PATH=$(cd ..; pwd)/jemalloc-4.0.3/lib
./test
```

先分配一个1字节的对象，然后分配10个8字节的对象，再分配15个30字节的对象，结果如下：

```
zjs@zjs:~/codelab/jemalloc_test$ ./test
base = 0x7ff7e4600000
arena_index = 16
input size and count to allocate (size = 0 or count = 0 to exit): 1 1
my_chunk_alloc(chunk = (nil), size = 2097152, alignment = 2097152)
+++size = 1, usable_size = 8, offset = 53248, page_id = 13, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit): 8 10
+++size = 8, usable_size = 8, offset = 53256, page_id = 13, offset_in_page = 8
+++size = 8, usable_size = 8, offset = 53264, page_id = 13, offset_in_page = 16
+++size = 8, usable_size = 8, offset = 53272, page_id = 13, offset_in_page = 24
+++size = 8, usable_size = 8, offset = 53280, page_id = 13, offset_in_page = 32
+++size = 8, usable_size = 8, offset = 53288, page_id = 13, offset_in_page = 40
+++size = 8, usable_size = 8, offset = 53296, page_id = 13, offset_in_page = 48
+++size = 8, usable_size = 8, offset = 53304, page_id = 13, offset_in_page = 56
+++size = 8, usable_size = 8, offset = 53312, page_id = 13, offset_in_page = 64
+++size = 8, usable_size = 8, offset = 53320, page_id = 13, offset_in_page = 72
+++size = 8, usable_size = 8, offset = 53328, page_id = 13, offset_in_page = 80
input size and count to allocate (size = 0 or count = 0 to exit): 30 15
+++size = 30, usable_size = 32, offset = 57344, page_id = 14, offset_in_page = 0
+++size = 30, usable_size = 32, offset = 57376, page_id = 14, offset_in_page = 32
+++size = 30, usable_size = 32, offset = 57408, page_id = 14, offset_in_page = 64
+++size = 30, usable_size = 32, offset = 57440, page_id = 14, offset_in_page = 96
+++size = 30, usable_size = 32, offset = 57472, page_id = 14, offset_in_page = 128
+++size = 30, usable_size = 32, offset = 57504, page_id = 14, offset_in_page = 160
+++size = 30, usable_size = 32, offset = 57536, page_id = 14, offset_in_page = 192
+++size = 30, usable_size = 32, offset = 57568, page_id = 14, offset_in_page = 224
+++size = 30, usable_size = 32, offset = 57600, page_id = 14, offset_in_page = 256
+++size = 30, usable_size = 32, offset = 57632, page_id = 14, offset_in_page = 288
+++size = 30, usable_size = 32, offset = 57664, page_id = 14, offset_in_page = 320
+++size = 30, usable_size = 32, offset = 57696, page_id = 14, offset_in_page = 352
+++size = 30, usable_size = 32, offset = 57728, page_id = 14, offset_in_page = 384
+++size = 30, usable_size = 32, offset = 57760, page_id = 14, offset_in_page = 416
+++size = 30, usable_size = 32, offset = 57792, page_id = 14, offset_in_page = 448
input size and count to allocate (size = 0 or count = 0 to exit):
```

貌似可以总结这么几条规律：

- 分配是从第13个页开始的，依次递增；
- 用户传入的大小，会被“标准化”到一个档位，称为标准化大小；
- 一个页只会分配某一种标准化大小的对象，而且这些对象紧凑排布，向后递增。

当然，这些规律可以继续深入探究验证。我这里其实已经验证过了。接着的问题是，某种标准化大小会占用多少页呢？

先分配一个size_1大小的对象，然后再分配1个size_2大小的对象（确保size_1和size_2不在同一个档位里），看看两个档位所占的第一页相距多少，那么这个差距就是size_1所在的标准化档位所占用的页数。

```
zjs@zjs:~/codelab/jemalloc_test$ ./test
base = 0x7f3d9a200000
arena_index = 16
input size and count to allocate (size = 0 or count = 0 to exit): 1 1
my_chunk_alloc(chunk = (nil), size = 2097152, alignment = 2097152)
+++size = 1, usable_size = 8, offset = 53248, page_id = 13, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit): 9 1
+++size = 9, usable_size = 16, offset = 57344, page_id = 14, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit): 17 1
+++size = 17, usable_size = 32, offset = 61440, page_id = 15, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit): 33 1
+++size = 33, usable_size = 48, offset = 65536, page_id = 16, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit): 49 1
+++size = 49, usable_size = 64, offset = 77824, page_id = 19, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit): 65 1
+++size = 65, usable_size = 80, offset = 81920, page_id = 20, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit): 81 1
+++size = 81, usable_size = 96, offset = 102400, page_id = 25, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit): 97 1
+++size = 97, usable_size = 112, offset = 114688, page_id = 28, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit): 113 1
+++size = 113, usable_size = 128, offset = 143360, page_id = 35, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit): 129 1
+++size = 129, usable_size = 160, offset = 147456, page_id = 36, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit): 161 1
+++size = 161, usable_size = 192, offset = 167936, page_id = 41, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit): 193 1
+++size = 193, usable_size = 224, offset = 180224, page_id = 44, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit): 225 1
+++size = 225, usable_size = 256, offset = 208896, page_id = 51, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit): 257 1
+++size = 257, usable_size = 320, offset = 212992, page_id = 52, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit): 321 1
+++size = 321, usable_size = 384, offset = 233472, page_id = 57, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit):
```

可以汇总一下：

标准化大小	占用页数	对象个数	是否有碎片
8	14 - 13 = 1	(4096 * 1) / 8 = 512	否
16	15 - 14 = 1	(4096 * 1) / 16 = 256	否
32	16 - 15 = 1	(4096 * 1) / 32 = 128	否
48	19 - 16 = 3	(4096 * 3) / 48 = 256	否
64	20 - 19 = 1	(4096 * 1) / 64 = 64	否
80	25 - 20 = 5	(4096 * 5) / 80 = 256	否
96	28 - 25 = 3	(4096 * 3) / 96 = 128	否
112	35 - 28 = 7	(4096 * 7) / 112 = 256	否
128	36 - 35 = 1	(4096 * 1) / 128 = 32	否
160	41 - 36 = 5	(4096 * 5) / 160 = 128	否
192	44 - 41 = 3	(4096 * 3) / 192 = 64	否
224	51 - 44 = 7	(4096 * 7) / 224 = 128	否
256	52 - 51 = 1	(4096 * 1) / 256 = 16	否
320	57 - 52 = 5	(4096 * 5) / 320 = 64	否

用这样的方法不断的扩张下去，就能得到更长的表：

标准化大小	占用页数	对象个数	是否有碎片

384	60 - 57 = 3	(4096 * 3) / 384 = 32	否
448	67 - 60 = 7	(4096 * 7) / 448 = 64	否
512	69 - 67 = 2	(4096 * 2) / 512 = 16	否
640	74 - 69 = 5	(4096 * 5) / 640 = 32	否
768	77 - 14 = 3	(4096 * 3) / 768 = 16	否
896	84 - 77 = 7	(4096 * 7) / 896 = 32	否
1024	87 - 84 = 3	(4096 * 3) / 1024 = 12	否
1280	92 - 87 = 5	(4096 * 5) / 1280 = 16	否
1536	98 - 92 = 6	(4096 * 6) / 1536 = 16	否
1792	105 - 98 = 7	(4096 * 7) / 1792 = 16	否
2048	110 - 105 = 5	(4096 * 5) / 2048 = 10	否
2560	120 - 110 = 10	(4096 * 10) / 2560 = 16	否
3072	129 - 120 = 9	(4096 * 9) / 3072 = 12	否
3584	143 - 129 = 14	(4096 * 14) / 3584 = 16	否
4096	153 - 143 = 10	(4096 * 10) / 4096 = 10	否
5120	168 - 153 = 15	(4096 * 15) / 5120 = 12	否
6144	183 - 168 = 15	(4096 * 15) / 6144 = 10	否
7168	204 - 183 = 21	(4096 * 21) / 7168 = 12	否
8192	224 - 204 = 20	(4096 * 20) / 8192 = 10	否
10240	249 - 224 = 25	(4096 * 25) / 10240 = 10	否
12288	279 - 249 = 30	(4096 * 30) / 12288 = 10	否
14336	314 - 279 = 35	(4096 * 35) / 14336 = 10	否

有没有惊奇地发现，jemalloc对于某个标准化大小std_size，会取整数n个页，使得4096 * n能够被std_size整除。但是再具体一些：

- 一般地，4096 * n会是4096与std_size的最小公倍数；
- 但是，同时，得确保4096 * n / std_size >= 10。

就在我以为我发现了世界的铁律时，打脸了：

```
+++size = 10241, usable_size = 12288, offset = 1019904, page_id = 249, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit): 12289 1
+++size = 12289, usable_size = 14336, offset = 1142784, page_id = 279, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit): 14337 1
+++size = 14337, usable_size = 16384, offset = 1288000, page_id = 314, offset_in_page = 1856
input size and count to allocate (size = 0 or count = 0 to exit): 16385 1
+++size = 16385, usable_size = 20480, offset = 1308864, page_id = 319, offset_in_page = 2240
input size and count to allocate (size = 0 or count = 0 to exit): 20481 1
+++size = 20481, usable_size = 24576, offset = 1331904, page_id = 325, offset_in_page = 704
input size and count to allocate (size = 0 or count = 0 to exit): 24577 1
+++size = 24577, usable_size = 28672, offset = 1361792, page_id = 332, offset_in_page = 1920
input size and count to allocate (size = 0 or count = 0 to exit): 16384 1
+++size = 16384, usable_size = 16384, offset = 1395200, page_id = 340, offset_in_page = 2560
input size and count to allocate (size = 0 or count = 0 to exit): 16384 1
+++size = 16384, usable_size = 16384, offset = 1416512, page_id = 345, offset_in_page = 3392
input size and count to allocate (size = 0 or count = 0 to exit):
```

16384以及之后的size，都不是以4096对齐的了！而且同是16384，页内偏移量也不同！这个问题曾经一度困扰我，直到最后才知道，原来jemalloc对于small、large以及huge对象的分配策略是不同的。这种规律性极强的“紧凑对齐分配”只适用于small对象。那么我的问题就是，如何调整small对象的个数？比如我把上限从14336提高到65536？

中间探索的艰辛就不多言了。最后发现相当容易——修改jemalloc-4.0.3/include/jemalloc/internal/size_classes.sh文件：

```
if [ ${lg_size} -lt $((${lg_p} + ${lg_g})) ] ; then
  bin="yes"
else
  bin="no"
fi
```

如果log(size) < log(page_size) + log(group)，那么就是使用bin来管理。也就是说，如果size < page_size * group，那么就是“紧凑对齐分配”的small对象。那么算一下，当前配置下，page_size = 4096，group = 4（在每次大小翻倍之间，划分为4个档位），那么当size < 4096 * 4 = 16384时，就是small对象。这与我们之前的发现一致！

我想把small对象的分界线向上提，那么想当然地：

```
if [ ${lg_size} -lt $((${lg_p} + ${lg_g} + 1)) ] ; then
  bin="yes"
else
  bin="no"
fi
```

重新configure, make之后，应该分界线从16384提高到了32768了~

```
input size and count to allocate (size = 0 or count = 0 to exit): 14336 1
+++size = 14336, usable_size = 14336, offset = 143360, page_id = 35, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit): 14337 1
+++size = 14337, usable_size = 16384, offset = 286720, page_id = 70, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit): 16384 1
+++size = 16384, usable_size = 16384, offset = 303104, page_id = 74, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit): 20480 1
+++size = 20480, usable_size = 20480, offset = 450560, page_id = 110, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit): 24576 1
+++size = 24576, usable_size = 24576, offset = 655360, page_id = 160, offset_in_page = 0
input size and count to allocate (size = 0 or count = 0 to exit): 28672 1
+++size = 28672, usable_size = 28672, offset = 901120, page_id = 220, offset_in_page = 0
```

好，那么现在再来“强行发现”一条规律：

```
#include <stdio.h>
#include <assert.h>
#include <jemalloc.h>
#include <sys/mman.h>

#define PAGE_SIZE 4096

// 待管理的空间大小
size_t mem_size = 1 << 30;
// 待管理的空间的首地址
void* base;
// 已经分配出去的大小
size_t allocated = 0;

//++++++++++++++++++++++++++++HOOK BEGIN+++++++++++++++++++++++++++-

// 从这里开始定义jemalloc管理chunk的hook函数，自定义chunk管理行为
// 可以参考jemalloc/doc/jemalloc.html中arena.<i>.chunk_hooks一段

// 当jemalloc发现chunk不够用了，会callback此函数索要空间
// chunk大小在编译时配置（原版jemalloc-4.0.3默认人2M，libmemkind配置为4M）
void* my_chunk_alloc(void *chunk, size_t size, size_t alignment, bool
{
    printf("my_chunk_alloc(chunk = %p, size = %lu, alignment = %lu)\n'
    if(size % alignment)
        return NULL;
    if(allocated + size > mem_size)
        return NULL;
    if(chunk && chunk != base + allocated)
        return NULL;
    void* addr = base + allocated;
    allocated += size;
    return addr;
}

// 返回true表示该内存可以继续使用
bool my_chunk_dalloc(void *chunk, size_t size, bool commited, unsignec
{
    return true;
}

// 返回false表示内存充足
bool my_chunk_commit(void *chunk, size_t size, size_t offset, size_t l
{
    return false;
}

// 返回true表示该内存即使释放了，也是与物理内存对应的，可以重用
bool my_chunk_decommit(void *chunk, size_t size, size_t offset, size_t
{
```

```
    return true;
}

// 返回true表示该段地址空间被重用后不会清空
bool my_chunk_purge(void *chunk, size_t size, size_t offset, size_t le
{
    return true;
}

bool my_chunk_split(void *chunk, size_t size, size_t size_a, size_t si
{
    return false;
}

bool my_chunk_merge(void *chunk_a, size_t size_a, void *chunk_b, size_
{
    return false;
}

chunk_hooks_t my_chunk_hooks =
{
    .alloc = my_chunk_alloc,
    .dalloc = my_chunk_dalloc,
    .commit = my_chunk_commit,
    .decommit = my_chunk_decommit,
    .purge = my_chunk_purge,
    .split = my_chunk_split,
    .merge = my_chunk_merge,
};
//-----------------------------HOOK END-----------------------------

int main()
{
    // 模拟一段NVM空间，或者任何一段用户待管理的空间（这里有1GB)
    base = mmap(0, mem_size, PROT_READ | PROT_WRITE, MAP_ANONYMOUS | M
    if(base == MAP_FAILED)
    {
        printf("mmap() failed!\n");
        return 1;
    }
    printf("base = %p\n", base);

    // 用je_mallctl('arenas.extend')命令创建一个arean,
    // 参考jemalloc/doc/jemalloc.html中arenas.extend一段
    unsigned arena_index;
    size_t unsigned_size = sizeof(unsigned int);
    if(je_mallctl("arenas.extend", (void*)&arena_index, &unsigned_size
    {
        printf("je_mallctl('arenas.extend') failed!\n");
        return 1;
    }
    printf("arena_index = %u\n", arena_index);
```

```
    //为这个arena绑定我们自定义的chunk hook，于是该arena就会按我们的方式去I
    // 参考jemalloc/doc/jemalloc.html中arena.<i>.chunk_hooks一段
    char cmd[64];
    sprintf(cmd, "arena.%u.chunk_hooks", arena_index);
    if(je_mallctl(cmd, NULL, NULL, (void*)&my_chunk_hooks, sizeof(chur
    {
        printf("je_mallctl('%s') failed!\n", cmd);
        return 1;
    }

    for(int i = 0; i < 4096; i++)
    {
        void* ptr = je_mallocx(4096, MALLOCX_ARENA(arena_index));
        if(!ptr)
        {
            printf("je_mallocx(4096, MALLOCX_ARENA(%u)) failed\n", are
            break;
        }
        assert(ptr >= base);
        // 算一算偏移量
        size_t offset = ptr - base;
        // 是待管理空间的第几个page
        size_t page_id = offset / PAGE_SIZE;
        // page内偏移多少
        size_t offset_in_page = offset % PAGE_SIZE;
        printf("size = 4096, offset = %lu, page_id = %lu, offset_in_pa
            offset, page_id, offset_in_page);
    }

    if(munmap(base , mem_size) != 0)
    {
        printf("munmap() failed!\n");
        return 1;
    }
    return 0;
}
```

惊奇地发现，第[512 * n, 512 * n + 13)页都是跳过的！

所以总结一下：

1. 第[512 * n, 512 * n + 13)页都是跳过的；
2. 用户传入的大小，会被“标准化”到一个档位，称为标准化大小；
3. 一个页只会分配某一种标准化大小的small对象，而且这些small对象紧凑排布，向后递增；

4. 一般地，标准化大小为std_size的small对象占用n页，4096 * n是4096与std_size的最小公倍数；
5. 但是，同时，得确保4096 * n / std_size >= 10。