VoWiFi 协议栈内存分析和优化

Document Number:	NA	**Document Version:**	1.0
Owner:	Evers Chen	**Date:**	2018/07/10
Document Type:			
NOTE:			

www.spreadtrum.com

Revision History

Revision	Date	Author	Description
1.0	2018-7-10	Evers.chen	Create

Table of Contents

Revision History 1
Table of Contents 1
1. VoWiFi 内存和缓冲区介绍 1
    1.1. 菊风内存管理和缓冲区 1
    1.2. SDK 内存一览表 2
2. 各层次 malloc 间的调用关系 3
    2.1. Zos 内存分配释放调用关系 3
    2.2. 内存的 malloc 替换的方法 4
        2.2.1. 编译时替换，在编译时取代 malloc 4
        2.2.2. 静态链接时重定位 5
        2.2.3. Load/run 时重定位 6
    2.3. Android jemalloc 的使用 7
    2.4. Chrome 内存分配器的管理 8
3. Pool - zos memory pool 11
    3.1. Pool 结构图 11
    3.2. Pool 内存介绍 12
        3.2.1. Pool 内存的分配 12
        3.2.2. Pool 内存的释放 13
        3.2.3. Pool 关键结构体定义 13
        3.2.4. Pool 的 block 和 node 15
4. Bpool - zos bitmap pool 16
    4.1. Bpool 结构图 16
    4.2. Bpool 内存介绍 17
        4.2.1. Bpool 内存的创建 17
        4.2.2. Bpool 内存的分配 17
        4.2.3. Bpool 内存的释放 18
        4.2.4. Bpool free bit 查找矩阵 18
        4.2.5. Bpool 关键结构体定义 20
        4.2.6. Bpool 的 bitmap 和内存大小的关系 22
5. Cbuf - zos convergency buffer library 22
    5.1. Cbuf 树形结构 22
    5.2. Cbuf 内存介绍 23
        5.2.1. Cbuf 内存的创建 23

5.2.2. Cbuf 内存的分配 24
5.2.3. Cbuf 内存的释放 24
5.2.4. Cbuf 关键结构体定义 24
5.2.5. Cbuf 的树形结构 25
6. Ppool - zos power pool 25
6.1. Ppool 结构图 25
6.2. Ppool 内存介绍 26
6.2.1. Ppool 内存的分配 26
6.2.2. Ppool 内存的释放 26
6.2.3. Ppool 关键结构体定义 27
6.2.4. Ppool 的地址说明 28
7. Sbuf - zos sbuf library 29
7.1. Sbuf 结构图 29
7.2. Sbuf 内存介绍 30
7.2.1. Sbuf 内存的分配 30
7.2.2. Sbuf 内存的释放 30
7.2.3. Sbuf 关键结构体定义 30
7.2.4. Sbuf 的使用 31
8. Brick- zos brick memory 32
8.1. Brick 结构图 32
8.2. Brick 内存介绍 33
8.2.1. Brick 内存的分配 33
8.2.2. Brick 内存的释放 33
8.2.3. Brick 关键结构体定义 33
8.2.4. Brick 的使用 35
9. Dbkt - zos dynamic bucket 36
9.1. Dbkt 结构图 36
9.2. Dbkt 内存介绍 37
9.2.1. Dbkt 内存的分配 37
9.2.2. Dbkt 内存的释放 37
9.2.3. Dbkt 关键结构体定义 37
9.2.4. Dbkt 的使用 39
10. Ubuf - zos unify buffer library 39
11. Dbuf - zos dbuf library 40
11.1. Dbuf 结构图 40
11.2. Dbuf 内存介绍 41

11.2.1. Dbuf 内存的创建 41
11.2.2. Dbuf 内存的分配 41
11.2.3. Dbuf 内存的释放 42
11.2.4. Dbuf 关键结构体定义 42
11.2.5. Dbuf 提供的接口 43
12. VoWiFi 协议栈内存相关的优化项 43
12.1. Dbuf ZOS_DBUF_SIZEOF_DATA（24 字节）导致的内存浪费 25.6% 43
12.1.1. 内存浪费原因分析 43
12.1.2. 解决方案 45
12.1.3. 优化前后测试对比 46
12.2. Dbuf 双重 pool 分配内存浪费问题 48% 47
12.2.1. 内存浪费原因分析 47
12.2.2. 解决方案 49
12.2.3. 优化前后测试对比 49
12.3. 用系统的 jemalloc 取代 pool 进行分配内存优化 23.4% 51
12.3.1. 用 jemalloc 取代的必要性 51
12.3.2. Pool 潜在的问题与调试缺陷 51
12.3.3. Jemalloc 取代前后的内存使用比 51
12.4. Zos_DlistRemove/Zos_PoolAlloc 的 crash 问题 60
12.5. IKE stMsgList 释放的踩内存地址问题分析 60
12.5.1. IKE stMsgList 内存分配方式 60
12.5.2. pstMsg->zMemBuf 地址在 pool 内存的位置 61
12.5.3. IKE stMsgList 修复及预防 62
12.6. Dbuf 的 dump crash 问题 63
12.7. Juphoon 修改 Dbuf dump crash 引入的 bad fix 63
12.8. Udp 接收缓冲区内存和 pstEvnt->pstMsg 内存的不释放问题 63
12.9. Brick 空指针问题 63
12.10. Dump,fsmDump 的多线程访问 wStackNum 问题 64
12.11. ERROR: SysStrFree invalid magic. 64
12.12. 几个因为提供错误内存句柄导致的释放失败问题 64
12.13. 多处 FOR_ALL_DATA_IN_DLIST 使用错误 65
12.14. 重复删除 pstSubsd->stTransLst 问题 65
12.15. VoWiFi 进程内存优化前后对比 65

1. VoWiFi 内存和缓冲区介绍

1.1. 菊风内存管理和缓冲区

尽管操作系统提供了内存管理功能，但为了保证可移植性与可管理性，而且由于不同的产品容量有不同的限制和不同的性能要求，系统平台也提供了内存管理功能。内存管理的机制是建立一个内存池(pool)，内存池被分成多个大小不同的 bucket 组，比如 32 字节、64 字节的 bucket 组。当用户申请内存的时候 ZOS 将从最适合大小的 bucket 组中把空闲的 bucket 分配给用户。比如当用户申请大小为 50 字节的内存时，ZOS 首先从 64 字节大小的 bucket 组中查找空闲 bucket，如果没有可用的空闲 bucket，就到比较大的 bucket 组中继续查找，直到找到可用的空闲 bucket。若找不到可用的空闲 bucket，则返回失败信息。

每一个 bucket 节点表明一块内存空间；所有的 bucket 都由 bucket 组管理。事实上，内存池就是从系统内存堆分配的一块足够大的内存块。当 bucket 组中的 bucket 用完的时候，ZOS 可以从系统内存堆中分配更多的 bucket，前提是 bucket 可增加数目是非零。Bucket 管理机制是一种通用而高效的内存管理方法。在 ZOS 平台中，内存管理、消息管理、数据缓冲区管理都是采用 bucket 管理机制实现的。

缓冲区主要用于 ZOS 层间任务之间的数据交换，数据可以是协议报文、层任务控制消息等等。使用缓冲区来交换数据可以减少数据复制的次数。如果系统需要花大量的 CPU 时间和内存带宽用于缓冲区之间的数据复制，则系统性能就会严重下降。例如传输层收到消息后会把协议报文放到个缓冲区中，然后把缓冲区发送给协议模块。

缓冲区也是一种高效而稳定的内存控制方式，尤其当模块需要使用大量的动态内存块时。如果这些内存是从内存池中申请，那么就必须在合适的地方释放这些内存块，稍有不慎就会产生内存泄露，从而影响系统的稳定性，而且大量的分配释放操作也影响了系统性能。但如果这些内存块是从缓冲区中申请的，那么只要该缓冲区被释放了，所有的内存块都被安全的释放了，不会存在内存泄露问题，且系统的稳定性也得到了提高。比如在协议编解码中，需要产生大量的协议消息控制块，这些控制块都是从一个数据缓冲区中申请的。当用户使用完这些协议消息控制块后，只要释放这一个数据缓冲区就可以了，因此编解码的性能和稳定性都得到了很大的提高了。

缓冲区管理提供了一种在系统内对缓冲区进行分配、操作和释放的通用机制。分配就是从全局缓冲区池获得缓冲区，包括将数据复制到缓冲区，从缓冲区中读出数据，在缓冲区的头部和尾部添加或删除数据，将两个缓冲区的数据拼接起来及复制缓冲区副本等。释放就是将缓冲区返还给全局缓冲区，以便其他任务或模块能对其进行分配。

有两种缓冲区管理方式：全局缓冲区管理和局部缓冲区管理。在全局管理方式下所有 ZOS 任务都从一个缓冲区池中分配缓冲区；而在局部管理方式下 ZOS 任务可以独自创建缓冲区池，且缓冲区的分配不会占用其他 ZOS 任务的缓冲区资源。

ZOS 的缓冲区管理称为数据缓中区（ Data Buffer）管理，数据缓冲区控制块中有所有缓冲区的数据长度链接了所有的数据块的链表。控制块中的默认数据块的大小有独特的用法，当请求的数据块小于默认大小的时候，ZOS 从内存池中分配如默认大小的内存块。但如果所请求的数据块大于默认数值时，缓冲区会首先尝试着从内存池中分配所需数据块的大小的内存块，如果能分配到，则放置于缓冲区中的数据块链表中，否则分配失败。

数据缓冲区有两种类型：字节对齐和结构对齐类型。字节对齐类型表明在分配数据内存块的时候，内存块所占用的地址空间是字节连续的，其内存块首地址不需要是 4 的倍数值；而结构对齐类型则表明每次分配的数据内存块大小都是 4 的倍数，如请求 2 字节数据内存块意味着分配 4 字节的数据内存块。

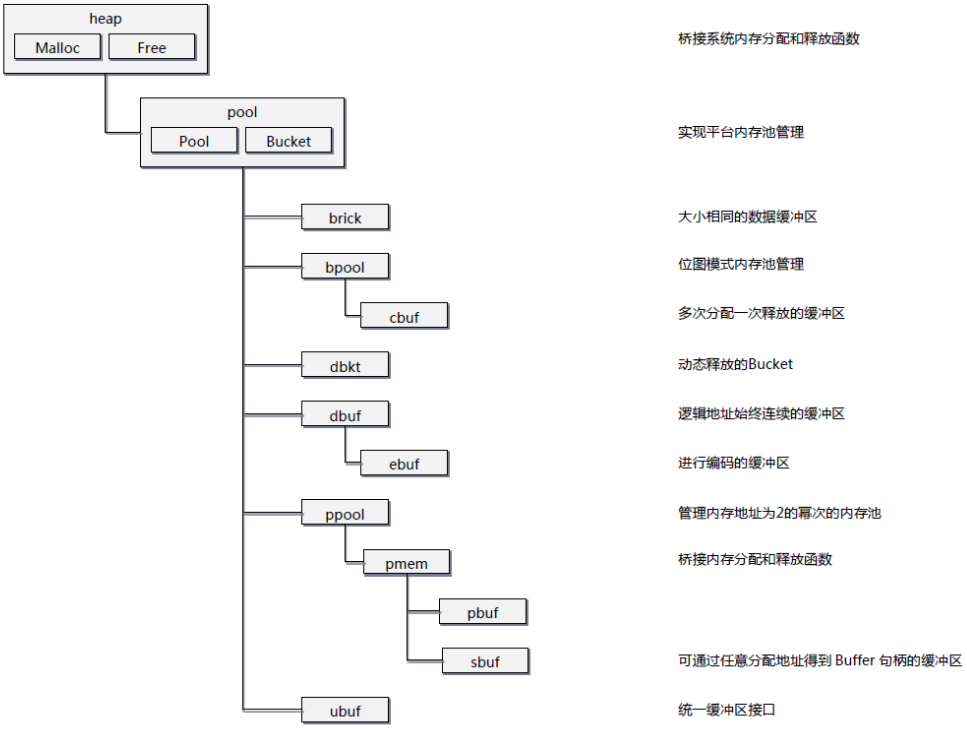
此外数据缓冲区可以被多次引用，当引用计数为 0 时，也就意味数据缓冲区被释放了。

除了可以从内存池中动态的申请缓冲区，也可以通过静态内存区来创建缓冲区。

1.2. SDK 内存一览表

下面是菊风 SDK 的内存一览表，这里列了所有使用到的内存及其关系。

一览



2. 各层次 malloc 间的调用关系

2.1. Zos 内存分配释放调用关系

Zos_Malloc/ Zos_Realloc / Zos_Free –> Zos_PoolAlloc/ Zos_PoolFree-> Zos_HeapAlloc/ Zos_HeapFree-> MALLOC/FREE

Zos_DbufAlloc/ Zos_DbufFree -> Zos_PoolAlloc/ Zos_PoolFree

Zos_PbufAlloc/ Zos_SbufAlloc ->Zos_PMemAlloc ->Zos_PPoolAlloc ->Zos_Malloc

2.2. 内存的 malloc 替换的方法

```
// File Name : hello.c
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
int main(void)
{
    // Call to user defined malloc
    void *ptr = malloc(4);

    printf("Hello World\n");
    return 0;
}
```

2.2.1. 编译时替换，在编译时取代 malloc

```
/* Compile-time interposition of malloc using C preprocessor.
A local malloc.h file defines malloc as wrapper */

// A file that contains our own malloc function
// File Name : mymalloc.c
#include <stdio.h>
#include <malloc.h>
void *mymalloc(size_t s)
{
printf("My malloc called");
return NULL;
}

// filename : malloc.h
// To replace all calls to malloc with mymalloc
#define malloc(size) mymalloc(size)
void *mymalloc(size_t size);
```


Steps to execute above on Linux:

```
// Compile the file containing user defined malloc()
:~$ gcc -c mymalloc.c

// Compile hello.c with output file name as helloc.
// -I. is used to include current folder (.) for header
// files to make sure our malloc.h is becomes available.
:~$ gcc -I. -o helloc hello.c mymalloc.o

// Run the generated executable
:~$ ./helloc
My malloc called
Hello World
```

2.2.2. 静态链接时重定位

```
// filename : mymalloc.c
/* Link-time interposition of malloc using the   static linker's (ld) "--wrap symbol" flag. */
#include <stdio.h>

// __real_malloc() is used to called actual library
// malloc()
void *__real_malloc(size_t size);

// User defined wrapper for malloc()
void *__wrap_malloc(size_t size)
{
printf("My malloc called");
return NULL;
}
```

Steps to execute above on Linux:

```
// Compile the file containing user defined malloc()
```

```
:~$ gcc -c mymalloc.c

// Compile hello.c with output name as hellol
// "-Wl,--wrap=malloc" is used tell the linker to use
// malloc() to call __wrap_malloc(). And to use
// __real_malloc() to actual library malloc()
:~$ gcc -Wl,--wrap=malloc -o hellol hello.c mymalloc.o

// Run the generated executable
:~$ ./hellol
My malloc called
Hello World
```

2.2.3. Load/run 时重定位

The environment variable LD_PRELOAD gives the loader a list of libraries to load be loaded before a command or executable.
We make a dynamic library and make sure it is loaded before our executable for hello.c.

```
/* Run-time interposition of malloc based on dynamic linker's
   (ld-linux.so) LD_PRELOAD mechanism */
#define _GNU_SOURCE
#include <stdio.h>

void *malloc(size_t s)
{
   printf("My malloc called\n");
   return NULL;
}
```

Steps to execute above on Linux:

```
// Compile hello.c with output name as helloc
:~$ gcc -o hellor hello.c
```



```
// Generate a shared library myalloc.so. Refer
// https://www.geeksforgeeks.org/working-with-shared-libraries-set-2/
// for details.
:~$ gcc -shared -fPIC -o mymalloc.so mymalloc.c

// Make sure shared library is loaded and run before .
:~$ LD_PRELOAD=./mymalloc.so ./hellor
My malloc called
Hello World
```

On Android: load-time symbol interposition (unlike the Linux/CrOS case) is not possible. This is because Android processes are `fork()`-ed from the Android zygote, which pre-loads libc.so and only later native code gets loaded via `dlopen()` (symbols from `dlopen()`-ed libraries get a different resolution scope).

2.3. Android jemalloc 的使用

/bionic/libc/bionic/malloc_common.cpp

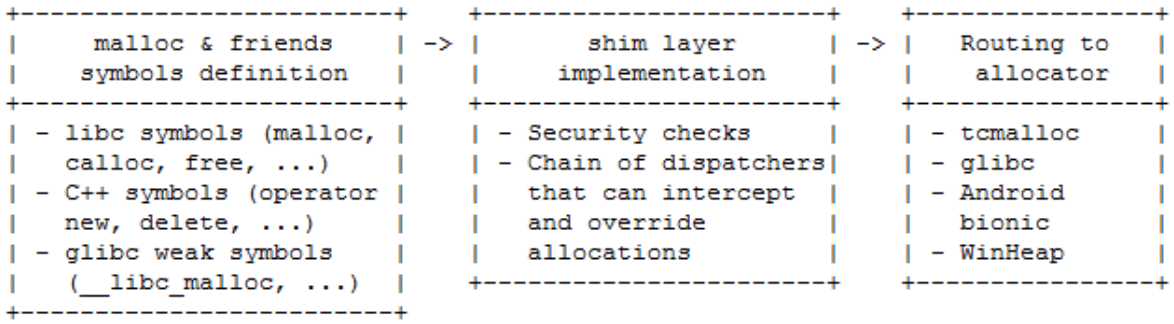
```
#define Malloc(function)  je_ ## function

extern "C" void* malloc(size_t bytes) {
  auto _malloc = __libc_globals->malloc_dispatch.malloc;
  if (__predict_false(_malloc != nullptr)) {
    return _malloc(bytes);
  }
  return Malloc(malloc)(bytes);
}
```

Jemalloc 的路径：external/jemalloc/

2.4. Chrome 内存分配器的管理

shim 分配器由下面三个阶段组成:



malloc->__wrap_malloc->ShimMalloc->[chain_head->alloc_function]->RealMalloc->__real_malloc (Android bionic)->je_malloc

1. malloc symbols definition
This stage takes care of overriding the symbols `malloc`, `free`, `operator new`, `operator delete` and friends and routing those calls inside the allocator shim (next point).
This is taken care of by the headers in `allocator_shim_override_*`.

On Linux/CrOS: the allocator symbols are defined as exported global symbols in `allocator_shim_override_libc_symbols.h` (for `malloc`, `free` and friends) and in `allocator_shim_override_cpp_symbols.h` (for `operator new`, `operator delete` and friends).
This enables proper interposition of malloc symbols referenced by the main executable and any third party libraries. Symbol resolution on Linux is a breadth first search that starts from the root link unit, that is the executable (see EXECUTABLE AND LINKABLE FORMAT (ELF) - Portable Formats Specification).
Additionally, when tcmalloc is the default allocator, some extra glibc symbols are also defined in `allocator_shim_override_glibc_weak_symbols.h`, for subtle reasons explained in that file.

The Linux/CrOS shim was introduced by
crrev.com/1675143004.

On Android: load-time symbol interposition (unlike the Linux/CrOS case) is not possible. This is because Android processes are `fork()`-ed from the Android zygote, which pre-loads libc.so and only later native code gets loaded via `dlopen()` (symbols from `dlopen()`-ed libraries get a different resolution scope).
In this case, the approach instead of wrapping symbol resolution at link time (i.e. during the build), via the `--Wl,-wrap,malloc` linker flag.
The use of this wrapping flag causes:
- All references to allocator symbols in the Chrome codebase to be rewritten as references to `__wrap_malloc` and friends. The `__wrap_malloc` symbols are defined in the `allocator_shim_override_linker_wrapped_symbols.h` and route allocator calls inside the shim layer.
- The reference to the original `malloc` symbols (which typically is defined by the system's libc.so) are accessible via the special `__real_malloc` and friends symbols (which will be relocated, at load time, against `malloc`).

In summary, this approach is transparent to the dynamic loader, which still sees undefined symbol references to malloc symbols.
These symbols will be resolved against libc.so as usual.
More details in crrev.com/1719433002.

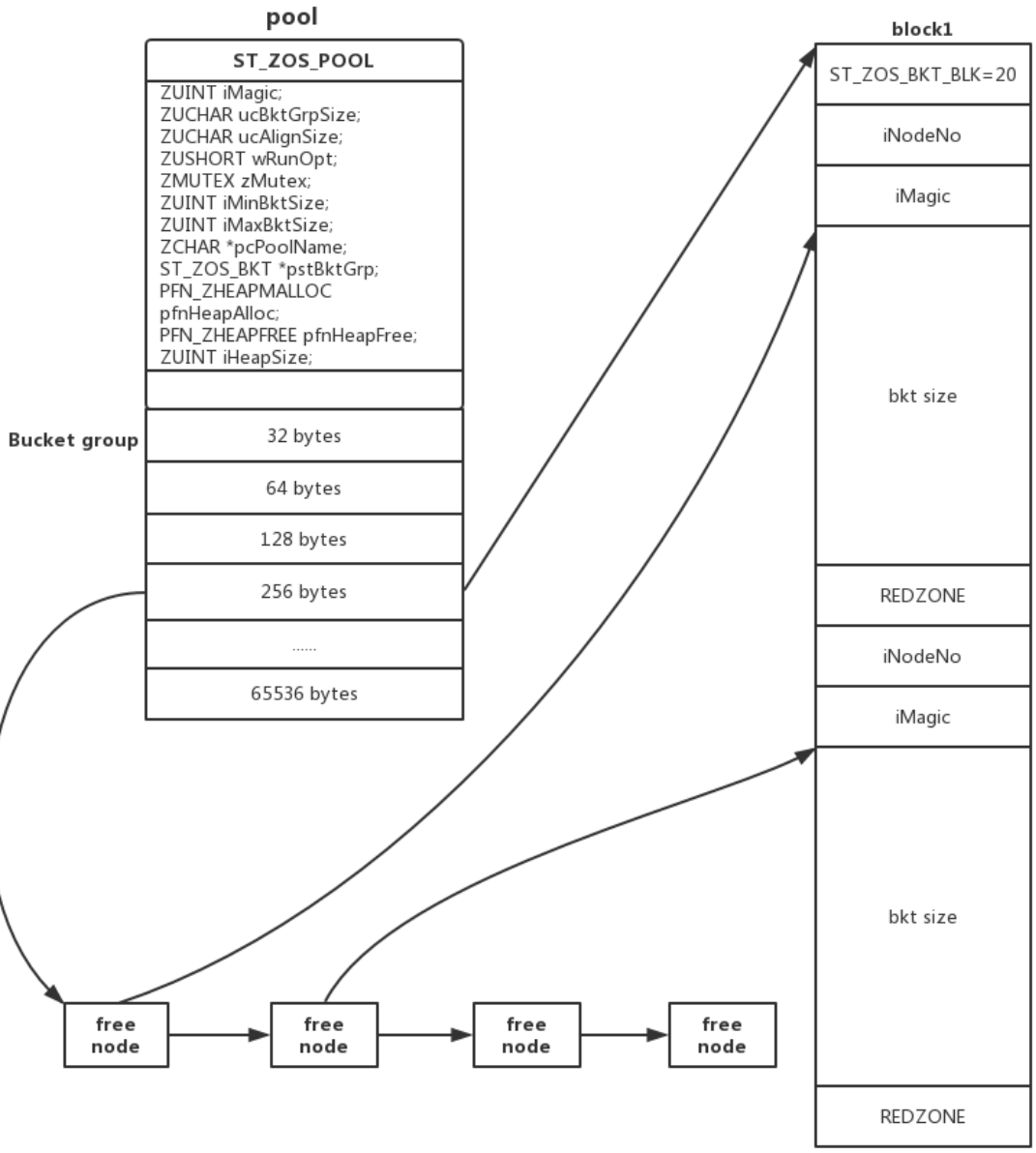
2. Shim layer implementation
This stage contains the actual shim implementation. This consists of:
- A singly linked list of dispatchers (structs with function pointers to `malloc`-like functions). Dispatchers can be dynamically inserted at runtime (using the `InsertAllocatorDispatch` API). They can intercept and override allocator calls.
- The security checks (suicide on malloc-failure via `std::new_handler`, etc).
This happens inside `allocator_shim.cc`

3. Final allocator routing
The final element of the aforementioned dispatcher chain is statically defined at build time and ultimately routes the allocator calls to the actual allocator

(as described in the *Background* section above). This is taken care of by the headers in `allocator_shim_default_dispatch_to_*` files.

3. Pool - zos memory pool

3.1. Pool 结构图



3.2. Pool 内存介绍

3.2.1. Pool 内存的分配

找到一个刚好大于申请内存大小的 bucket， 从该 bucket 分配一个 free node，如果该 bucket 的 stFreeNodeLst 为空，则申请一个 block 再继续进行分配。

如果应用申请的内存大小超过最大的 bucket size（64K），直接从系统分配内存，并且对 node 的信息做如下特别处理：

1. pstNode->iNodeNo = iSize | ZBKT_HEAP_BKTID;
2. pstNode->iMagic = ZBKT_HEAP_MAGIC;

pool 内存从系统分配是按 block 为单位，一个 block 包含的 node 数量是对应 bucket 配置信息里的 increment count。应用从 pool 申请内存是按 node 来分配，每个 node 的大小是对应 bucket 的 size，一次分配得到整个 node，有内存浪费现象。

下面是 pool 的 bucket 配置信息，对于不同的 bucket size，会有不同的 increment count--也就是每个 block 中包含 node 的数量。

```
/* zos default memory bucket config info group */
ZCONST ST_ZOS_BKT_INFO m_astZosCfgMemBktInfoGrp[] =
{
      /* size,                maximum count,          increment count */
      {32,                      0,                          64},//2k
      {64,                      0,                          32},//2k
      {128,                     0,                          16},//2k
      {256,                     0,                          8},//2k
      {512,                     0,                          4},//2k
      {1024,                    0,                          4},//4k
      {2048,                    0,                          4},//8k
      {4096,                    0,                          2},//8k
      {5120,                    0,                          2},//10k
      {8192,                    0,                          1},//8k
      {16384,                   0,                          1},//16k
      {32768,                   0,                          1},//32k
```



```
    {65536,                 0,                          1}//64k
};
```

3.2.2. Pool 内存的释放

先根据内存地址得到 node 地址，检查是否是直接从 HEAP 分配的大内存，pstNode->iMagic == ZBKT_HEAP_MAGIC，如果是则释放 heap 内存，否则，从 node id 得到 bucket id，再进行一些状态，magic 和 REDZONE 检测，通过后恢复 pstFreeInfo 信息，并把该 node 作为 free node 重新放入 bucket 的 stFreeNodeLst 中。如果当前 block 的所有的 node 都释放后，需要把整个 block 释放回系统，block 是 pool 从系统分配和释放内存的单位。

3.2.3. Pool 关键结构体定义

```
/* zos memory pool */
typedef struct tagZOS_POOL
{
    ZUINT iMagic;                   /* pool magic */
    ZUCHAR ucBktGrpSize;            /* bucket group size */
    ZUCHAR ucAlignSize;             /* alignment size */
    ZUSHORT wRunOpt;                /* run options ZPOOL_OPT_NEED_MUTEX... */
    ZMUTEX zMutex;                  /* bucket manager mutex */
    ZUINT iMinBktSize;              /* minimum size for bucket */
    ZUINT iMaxBktSize;              /* maximum size for bucket */
    ZCHAR *pcPoolName;              /* pool name name */
    ST_ZOS_BKT *pstBktGrp;          /* bucket group */
    PFN_ZHEAPMALLOC pfnHeapAlloc;   /* allocate memory from heap */
    PFN_ZHEAPFREE pfnHeapFree;      /* free memory into heap */
    ZUINT iHeapSize;                /* allocate size from heap */
    ZOS_PADX64
} ST_ZOS_POOL;
```

block 的头长度为 20 字节，ST_ZOS_BKT_BLK 如下：

```
/* zos bucket memory block */
typedef struct tagZOS_BKT_BLK
```



```
{
    struct tagZOS_BKT_BLK *pstNext;  /* next bucket memory block */
    struct tagZOS_BKT_BLK *pstPrev;  /* previous bucket memory block */
    ZCHAR *pcStart;                  /* bucket start memory after aligned */
    ZCHAR *pcEnd;                    /* bucket end memory after aligned */
    ZUSHORT wAllCount;               /* all bucket count */
    ZUSHORT wFreeCount;              /* free bucket count */
    ZOS_PADX64
} ST_ZOS_BKT_BLK;
```

bucket 头长度为 36 字节，ST_ZOS_BKT 如下：

```
/* zos bucket */
typedef struct tagZOS_BKT
{
    ZUINT iBktSize;                  /* bucket size */
    ZUINT iNodeSize;                 /* bucket node size */
    ZUSHORT wMaxCount;               /* bucket maixmum count */
    ZUSHORT wIncCount;               /* bucket memory inc count per time */
    ZUSHORT wAllCount;               /* all bucket count */
    ZUSHORT wFreeCount;              /* bucket free count */
    ST_ZOS_DLIST stAllocLst;         /* bucket allocated memory list */
    ST_ZOS_DLIST stFreeNodeLst;      /* bucket free node list */
    ZUINT iPeekAllocCount;           /* maximum allocate count for statistics
*/
    ZUINT iRequestCount;             /* request count for statistics */
    ZUINT iReleaseCount;             /* release count for statistics */
    ZOS_PADX64
} ST_ZOS_BKT;
```

node 头长度为 8 字节 ST_ZOS_BKT_NODE，结构如下：

```
/* zos bucket node */
typedef struct tagZOS_BKT_NODE
{
    ZUINT iNodeNo;                   /* bucket node no */
    ZUINT iMagic;                    /* bucket node magic */
    union
```



```
    {
        ST_ZOS_BKT_FREE_INFO astFreeInfo[1]; /* free node infomation */
        ZUCHAR aucData[4];           /* data block memory */
    } u;
#define pstNextFreeInfo u.astFreeInfo[0].pstNext
} ST_ZOS_BKT_NODE;
```

3.2.4. Pool 的 block 和 node

分配的所有 block 的内存会放入 stAllocLst，新分配的 node 如果没有使用，会把每个 node 放到 stFreeNodeLst（node 的链表）里去。

iNodeSize 比 iBktSize 大 12 字节，这 12 字节是 iNodeNo，iMagic 和 ZBKT_REDZONE。

```
/* bucket node size */
#define ZOS_BKT_NODE_SIZE(_bktsize) \
  (_bktsize) + ZOS_OFFSETOF(ST_ZOS_BKT_NODE, u) + sizeof(ZBKT_REDZONE)

/* zos bucket node */
typedef struct tagZOS_BKT_NODE
{
    ZUINT iNodeNo;                  /* bucket node no */
    ZUINT iMagic;                   /* bucket node magic */
    union
    {
        ST_ZOS_BKT_FREE_INFO astFreeInfo[1]; /* free node infomation */
        ZUCHAR aucData[4];          /* data block memory */
    } u;
#define pstNextFreeInfo u.astFreeInfo[0].pstNext
} ST_ZOS_BKT_NODE;
```

iNodeNo 包含的信息：

```
            wNodeId = wBaseIndex + i;//node id，唯一标识 node 的值。
```

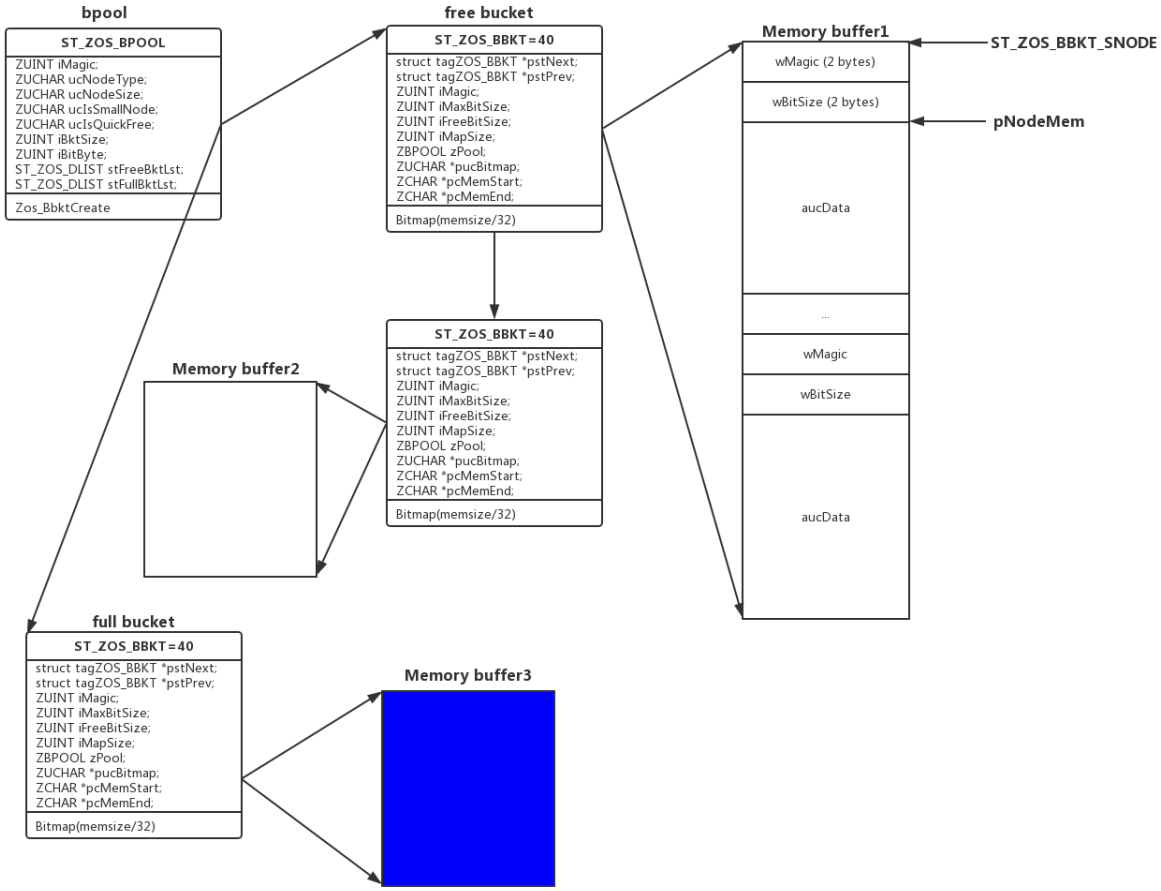
pstNode->iNodeNo = ZOS_BKT_MAKE_NODENO(wBktId, wNodeId);//把当前 node 的所在的 bucket id（wBktId<0xFF）放在高 16 位，node id 放在低 16 位；

pstNode->iNodeNo = pstNode->iNodeNo | ZBKT_BKT_NODE_FREE;//因为 wBktId 不大于 0xFFF,所以这里可以用高四位来设置 node 状态标志，

```
/* zos node id state is free */
#define ZBKT_BKT_NODE_FREE 0x40000000
```

4. Bpool - zos bitmap pool

4.1. Bpool 结构图



4.2. Bpool 内存介绍

4.2.1. Bpool 内存的创建

Bpool 内存创建的时候指定了一个 bucket size，范围为[32,4096]，和 Bpool 的类型，目前只用到 EN_ZOS_BPOOL_SMALL_SIZE。Bpool 目前只给 cbuf 使用。

4.2.2. Bpool 内存的分配

如果当前的 free bucket 里有足够的空间，会尝试进行分配。分配过程先找第一 bitmap 字节的最大可用空间，如果够找到返回。如果不够看该 bitmap 左边是否有可用空间，如果没有则继续找下一个字节的最大空间，如果有则找下一个字节的右边的可用空间，如果两者相加够则返回找到，这里因为前一个字节的左侧空间和后一个字节的右侧空间是连续的，所以可以连接起来用。如果不够但这个字节全部都可用，则继续找下一个字节的右侧。否则第一次找到的不符合条件，重新找，因为中间不连续。

例如分配 5 个 bit 的内存时：

bit no	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
	0	0	0	1	0	1	0	0	1	0	0	1	0	1	0	0
	< left >														<right>	
	3 bits														2 bits	

如果没有可用的 free bucket，则创建新的 bucket，如果当前申请内存大于 bucket size，则使用当前实际申请的内存大小作为新 bucket 的大小，否则使用 bucket size 新建 bucket。

4.2.3. Bpool 内存的释放

释放当前 bucket 的这块内存，只修改 bitmap 值。如果原来该 bucket 是处理 full 状态，那么释放后，需要把该 bucket 从 stFullBktLst 移到 stFreeBktLst 里。如果释放后这块 bucket 全部都是空的，没有被使用那么删除这块 bucket，这个时候实际释放内存。

4.2.4. Bpool free bit 查找矩阵

对应当前 bitmap 值左侧空闲的 bit 位数查找矩阵如下：

```
/* zos bitmap free bits count at the left */
ZCONST ZUCHAR m_aucZosBpoolLeftFreeBits[256] =
{
  /*  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  */
      8, 7, 6, 6, 5, 5, 5, 5, 4, 4, 4, 4, 4, 4, 4, 4, /* 0 */
      3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, /* 1 */
      2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, /* 2 */
      2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, /* 3 */
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, /* 4 */
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, /* 5 */
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, /* 6 */
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, /* 7 */
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 8 */
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 9 */
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* A */
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* B */
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* C */
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* D */
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* E */
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0   /* F */
};
```

例如下面的 bitmap，值为 20=0x14 ：

bit no	7	6	5	4	3	2	1	0
	0	0	0	1	0	1	0	0

m_aucZosBpoolLeftFreeBits[20]=3，表示左侧有 3 个 bit 是空闲的。

对应当前 bitmap 值右侧空闲的 bit 位数查找矩阵如下：

```
/* zos bitmap free bits count at the right */
ZCONST ZUCHAR m_aucZosBpoolRightFreeBits[256] =
{
  /*  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  */
      8, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0 */
      4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 1 */
      5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 2 */
      4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 3 */
      6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 4 */
      4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 5 */
      5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 6 */
      4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 7 */
      7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 8 */
      4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 9 */
      5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* A */
      4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* B */
      6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* C */
      4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* D */
      5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* E */
      4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0  /* F */
};
```

例如下面的 bitmap，值为 20=0x14 ：

bit no	7	6	5	4	3	2	1	0
	0	0	0	1	0	1	0	0

m_aucZosBpoolRightFreeBits[20]=2，表示右侧有 2 个 bit 是空闲的。

对应当前 bitmap 值最大空闲的 bit 位数查找矩阵如下：

```
/* zos bitmap maximum free bits in a byte */
```



```
ZCONST ZUCHAR m_aucZosBpoolMaxFreeBits[256] =
{
  /*  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  */
      8, 7, 6, 6, 5, 5, 5, 5, 4, 4, 4, 4, 4, 4, 4, 4, /* 0 */
      4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, /* 1 */
      5, 4, 3, 3, 2, 2, 2, 2, 3, 2, 2, 2, 2, 2, 2, 2, /* 2 */
      4, 3, 2, 2, 2, 2, 2, 2, 3, 2, 2, 2, 2, 2, 2, 2, /* 3 */
      6, 5, 4, 4, 3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 2, 2, /* 4 */
      4, 3, 2, 2, 2, 1, 1, 1, 3, 2, 1, 1, 2, 1, 1, 1, /* 5 */
      5, 4, 3, 3, 2, 2, 2, 2, 3, 2, 1, 1, 2, 1, 1, 1, /* 6 */
      4, 3, 2, 2, 2, 1, 1, 1, 3, 2, 1, 1, 2, 1, 1, 1, /* 7 */
      7, 6, 5, 5, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, /* 8 */
      4, 3, 2, 2, 2, 2, 2, 2, 3, 2, 2, 2, 2, 2, 2, 2, /* 9 */
      5, 4, 3, 3, 2, 2, 2, 2, 3, 2, 1, 1, 2, 1, 1, 1, /* A */
      4, 3, 2, 2, 2, 1, 1, 1, 3, 2, 1, 1, 2, 1, 1, 1, /* B */
      6, 5, 4, 4, 3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 2, 2, /* C */
      4, 3, 2, 2, 2, 1, 1, 1, 3, 2, 1, 1, 2, 1, 1, 1, /* D */
      5, 4, 3, 3, 2, 2, 2, 2, 3, 2, 1, 1, 2, 1, 1, 1, /* E */
      4, 3, 2, 2, 2, 1, 1, 1, 3, 2, 1, 1, 2, 1, 1, 0  /* F */
};
```

例如下面的 bitmap，值为 20=0x14 ：

bit no	7	6	5	4	3	2	1	0
	0	0	0	1	0	1	0	0

m_aucZosBpoolMaxFreeBits[20]=3，表示最大的空闲长度为 3 个 bit。

4.2.5. Bpool 关键结构体定义

```
/* zos bitmap pool */
typedef struct tagZOS_BPOOL
{
    ZUINT iMagic;                   /* magic value */
    ZUCHAR ucNodeType;              /* node type EN_ZOS_BPOOL_TYPE */
```

```
    ZUCHAR ucNodeSize;              /* node size */
    ZUCHAR ucIsSmallNode;           /* is small node */
    ZUCHAR ucIsQuickFree;           /* is quick free node */
    ZUINT iBktSize;                 /* bucket size */
    ZUINT iBitByte;                 /* bit byte size */
    ST_ZOS_DLIST stFreeBktLst;      /* free bucket list */
    ST_ZOS_DLIST stFullBktLst;      /* full bucket list */
} ST_ZOS_BPOOL;
```

ST_ZOS_BBKT 的头长为 40 字节。

```
/* zos bitmap bucket */
typedef struct tagZOS_BBKT
{
    struct tagZOS_BBKT *pstNext;    /* next bucket memory block */
    struct tagZOS_BBKT *pstPrev;    /* previous bucket memory block */
    ZUINT iMagic;                   /* magic value */
    ZUINT iMaxBitSize;              /* maximum bit size */
    ZUINT iFreeBitSize;             /* free bit size */
    ZUINT iMapSize;                 /* map size */
    ZBPOOL zPool;                  /* pool id */
    ZUCHAR *pucBitmap;              /* bitmap */
    ZCHAR *pcMemStart;              /* bucket start memory */
    ZCHAR *pcMemEnd;                /* bucket end memory */
} ST_ZOS_BBKT;
```

ST_ZOS_BBKT_SNODE 头是 4 字节：

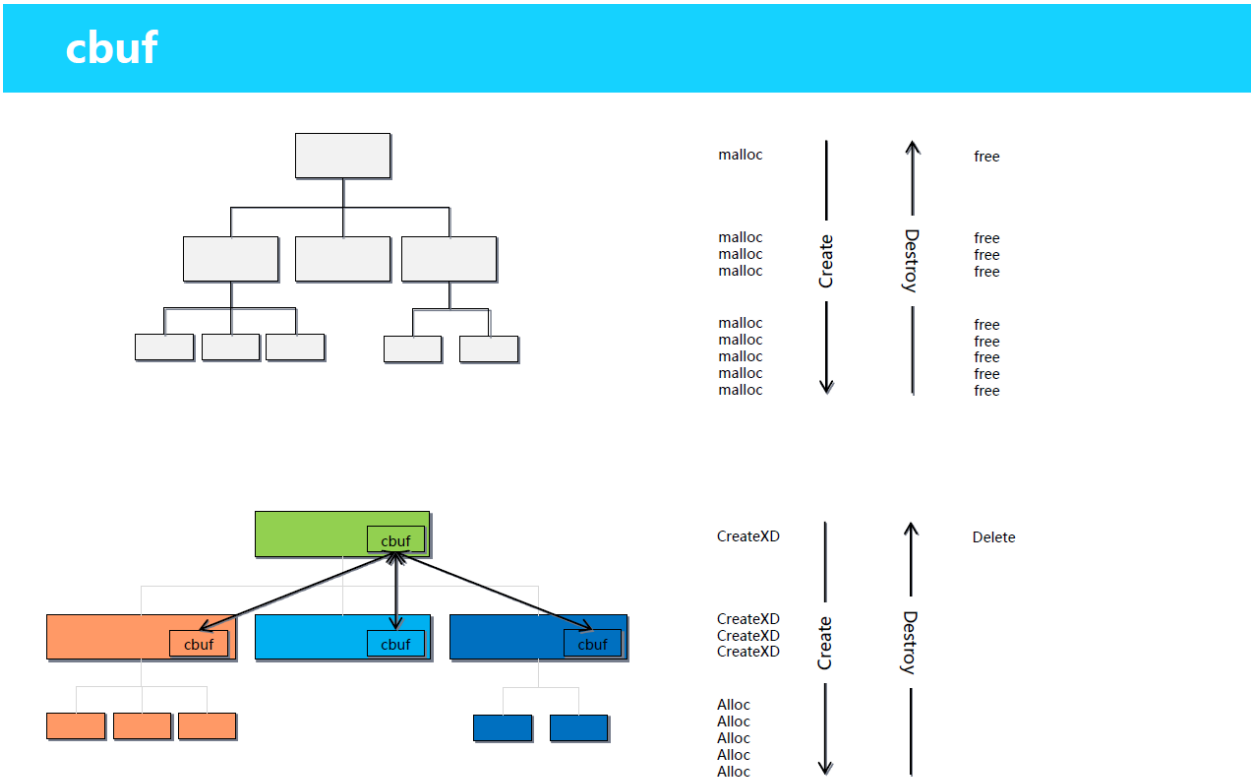
```
/* zos bitmap bucket small node */
typedef struct tagZOS_BBKT_SNODE
{
    ZUSHORT wMagic;                 /* magic value ZBPOOL_SNODE_MAGIC */
    ZUSHORT wBitSize;               /* bit size */
    ZUCHAR aucData[4];              /* data block memory */
} ST_ZOS_BBKT_SNODE;
```


4.2.6. Bpool 的 bitmap 和内存大小的关系

bitmap 和内存大小的关系？每个 bit 位可以表示 4 个字节的内存状态，所以 3 个字节最多表示内存为：3*8=24，24*4=96 字节内存，2 字节 bitmap 可以表示 64 字节内存。Bitmap 的大小也会根据当前 bucket 的内存大小来分配。

5. Cbuf - zos convergency buffer library

5.1. Cbuf 树形结构



5.2. Cbuf 内存介绍

5.2.1. Cbuf 内存的创建

每一个 cbuf 有自己的 bpool，pstBuf->zPool = Zos_BpoolCreate（...），cbuf 的数据管理在 ST_ZOS_BBKT。和 Dbuf 比较不同的是，所有的 dbuf 共享一个 pool，pstEnv->zDbufPoolId。

Cbuf 创建的时候 iBktSize 的计算问题，代码如下：

```
iBktSize = ZOS_ROUNDUP2(iBktSize, 32);
iBktSize = iBktSize >> 1;
while (iBktSize)
{
      iBktSize = iBktSize >> 1;
      iOffset++;
}
/* reset the bucket byte */
iBktSize = iOffset ? (1 << iOffset) : 0;
```

执行后取值如下，第一列是输入值，第二个是向上 32 取整后的值，最后一列是最终的值。可以看到这个值和原来的输入值相比可能变大，也可能变小。规则是 bktsize 32 向上取，然后按最高位取 bktsize 值；bktsize 值为 2 的幂次值,32,64,128,512,1024,...,4096。

```
iBktSize=33, up2 iBktSize=64, iOffset=6, iBktSize=0,end iOffset=6, iBktSize=64,
iBktSize=34, up2 iBktSize=64, iOffset=6, iBktSize=0,end iOffset=6, iBktSize=64,
iBktSize=35, up2 iBktSize=64, iOffset=6, iBktSize=0,end iOffset=6, iBktSize=64,
iBktSize=36, up2 iBktSize=64, iOffset=6, iBktSize=0,end iOffset=6, iBktSize=64,
iBktSize=37, up2 iBktSize=64, iOffset=6, iBktSize=0,end iOffset=6, iBktSize=64,
iBktSize=38, up2 iBktSize=64, iOffset=6, iBktSize=0,end iOffset=6, iBktSize=64,
...
iBktSize=95, up2 iBktSize=96, iOffset=6, iBktSize=0,end iOffset=6, iBktSize=64,
iBktSize=96, up2 iBktSize=96, iOffset=6, iBktSize=0,end iOffset=6, iBktSize=64,

iBktSize=97, up2 iBktSize=128, iOffset=7,iBktSize=0,end iOffset=7,iBktSize=128,
iBktSize=98, up2 iBktSize=128, iOffset=7,iBktSize=0,end iOffset=7,iBktSize=128,
```


iBktSize=99, up2 iBktSize=128, iOffset=7,iBktSize=0,end iOffset=7,iBktSize=128,

5.2.2. Cbuf 内存的分配

如果小于等于 4K，用 Zos_BpoolAlloc 分配内存，要不用 Zos_PoolAlloc 分配并且存在 stDBlkLst 中。

5.2.3. Cbuf 内存的释放

如果在 Zos_BpoolHoldD 里找到，用 Zos_BpoolFree 释放，要不用 Zos_PoolFree 释放 stDBlkLst 里的内存节点；

5.2.4. Cbuf 关键结构体定义

```
/* zos convergency buffer dynamic block */
typedef struct tagZOS_CBUF_DBLK
{
    struct tagZOS_CBUF_DBLK *pstNext; /* next block */
    struct tagZOS_CBUF_DBLK *pstPrev; /* previous block */
    ZUINT iSize;                      /* block size */
    ZUINT iMagic;                     /* block magic ZCBUF_DBLK_MAGIC */
} ST_ZOS_CBUF_DBLK;

/* zos convergency buffer */
typedef struct tagZOS_CBUF
{
    struct tagZOS_CBUF *pstNext;      /* next sibling */
    struct tagZOS_CBUF *pstPrev;      /* previous sibling */
    ZUINT iMagic;                     /* magic ZCBUF_MAGIC */
    ZUINT iRefCnt;                    /* reference count */
    ZCBUF zParent;                    /* parent buffer id */
    ZBPOOL zPool;                     /* bitmap pool id */
    ST_ZOS_DLIST stDBlkLst;           /* dynamic memory block list */
```



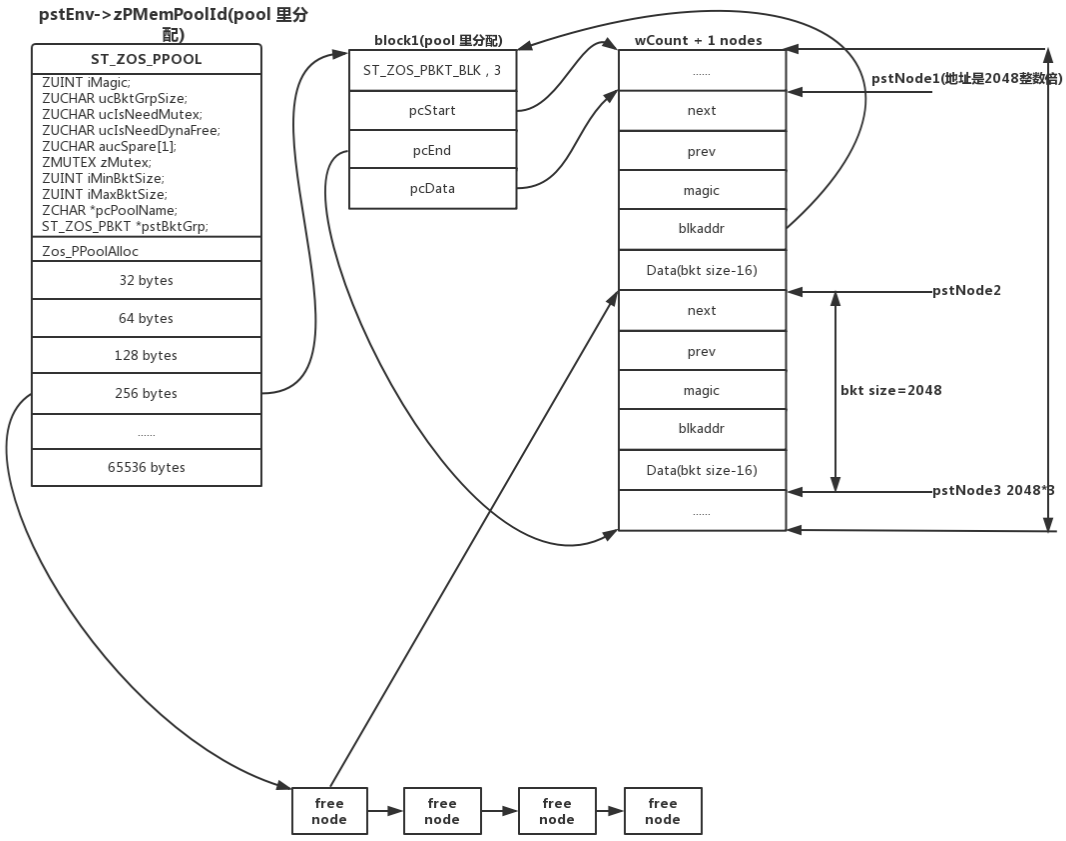
```
    ST_ZOS_DLIST stChildLst;          /* child cbuf list */
} ST_ZOS_CBUF;
```

5.2.5. Cbuf 的树形结构

cbuf 可以通过 attach 连接子 cbuf 到 stChildLst 生成树的结构；多个 cbuf 连接在一起，只在 json 里用到这个功能。

6. Ppool - zos power pool

6.1. Ppool 结构图



6.2. Ppool 内存介绍

6.2.1. Ppool 内存的分配

找到一个刚好大于申请内存大小的 bucket， 从该 bucket 分配一个 free node，如果该 bucket 的 stFreeNodeLst 为空，则申请一个 block 再继续进行分配。

如果应用申请的内存大小超过最大的 bucket size（64K），直接返回出错。

Ppool 内存从系统分配是按 block 为单位，一个 block 包含的 node 数量是对应 bucket 配置信息里的 increment count。应用从 Ppool 申请内存是按 node 来分配，每个 node 的大小是对应 bucket 的 size，一次分配得到整个 node，有内存浪费现象。

下面是 Ppool 的 bucket 配置信息，对于不同的 bucket size，会有不同的 increment count--也就是每个 block 中包含 node 的数量。

```
/* zos default power memory bucket config info group */
ZCONST ST_ZOS_BKT_INFO m_astZosCfgPMemBktInfoGrp[] =
{
      /* size,                maximum count,          increment count */
      {256,                    0,                          2},
      {512,                    0,                          4},
      {1024,                   0,                          4},
      {2048,                   0,                          2},
      {4096,                   0,                          2},
      {8192,                   0,                          1},
      {16384,                  0,                          1},
      {32768,                  0,                          1},
      {65536,                  0,                          1}
};
```

6.2.2. Ppool 内存的释放

Ppool 释放需要同时指定 bucket size 大小，这是因为 Ppool 是通过 pmem 给 pbuf 和 sbuf 提供服务的，在 sbuf 有 pagesize，pbuf 有 blksize，这些就是 Ppool 的 bucket

size 值。通过 bucket size 找到所在的 bucket。然后根据需要释放的 node 地址和 bucket 地址找到所在的 block。再把该 node 放回 stFreeNodeLst 中，并且重新保存 block 地址到 pBlkAddr。如果当前 block 所以的 node 都已经释放了，则释放当前 block 内存回系统。

6.2.3. Ppool 关键结构体定义

```
/* zos power pool pool */
typedef struct tagZOS_PPOOL
{
    ZUINT iMagic;                    /* pool magic */
    ZUCHAR ucBktGrpSize;             /* bucket group size */
    ZUCHAR ucIsNeedMutex;            /* is need mutex for bucket manager */
    ZUCHAR ucIsNeedDynaFree;         /* is need dynamic free for bucket */
    ZUCHAR aucSpare[1];              /* for 32 bit alignment */
    ZMUTEX zMutex;                   /* bucket manager mutex */
    ZUINT iMinBktSize;               /* minimum data size for bucket user */
    ZUINT iMaxBktSize;               /* maximum data size for bucket user */
    ZCHAR *pcPoolName;               /* pool name name */
    ST_ZOS_PBKT *pstBktGrp;          /* bucket group */
} ST_ZOS_PPOOL;
```

block 的头长度为 24 字节，ST_ZOS_PBKT_BLK 如下：

```
/* zos power pool bucket block */
typedef struct tagZOS_PBKT_BLK
{
    struct tagZOS_PBKT_BLK *pstNext; /* next bucket memory block */
    struct tagZOS_PBKT_BLK *pstPrev; /* previous bucket memory block */
    ZUSHORT wMaxCount;               /* block maixmum count */
    ZUSHORT wBusyCount;              /* block busy count */
    ZOS_PADX64
    ZCHAR *pcStart;                  /* bucket start memory after aligned */
    ZCHAR *pcEnd;                    /* bucket end memory after aligned */
    ZCHAR *pcData;                   /* bucket data memory */
} ST_ZOS_PBKT_BLK;
```


bucket 头长度为 32 字节，ST_ZOS_PBKT 如下：

```
/* zos power pool bucket */
typedef struct tagZOS_PBKT
{
    ZUINT iSize;                    /* bucket size */
    ZUSHORT wMaxCount;              /* bucket maixmum count */
    ZUSHORT wIncCount;              /* bucket memory inc count per time */
    ZUSHORT wFreeCount;             /* bucket free count */
    ZUCHAR aucSpare[2];             /* for 32 bit alignment */
    ZUINT iPeekAllocCount;          /* maximum allocate count for statistics
*/
    ZUINT iRequestCount;            /* request count for statistics */
    ZUINT iReleaseCount;            /* release count for statistics */
    ST_ZOS_DLIST stAllocLst;        /* bucket allocated memory list */
    ST_ZOS_DLIST stFreeNodeLst;     /* bucket free node list */
} ST_ZOS_PBKT;
```

node 头长度为 16 字节 ST_ZOS_PBKT_NODE，该头只对于 free node 的时候存在，如果 node 被使用，这些也作为数据区被使用。结构如下：

```
/* zos power pool bucket node */
typedef struct tagZOS_PBKT_NODE
{
    struct tagZOS_PBKT_FREE_INFO *pstNext; /* next bucket node */
    struct tagZOS_PBKT_FREE_INFO *pstPrev; /* previous bucket node */
    ZUINT iMagic;                   /* free info magic */
    ZOS_PADX64
    ZVOID *pBlkAddr;                /* block memory address */
} ST_ZOS_PBKT_NODE;
```

6.2.4. Ppool 的地址说明

node 地址是 bucket size 的整数倍；比如 bucket size 2048，那么两个 node 地址后 11 位都为 0，所以实际分配了三个 node，浪费了一个 node 空间来达到地址为整数倍的

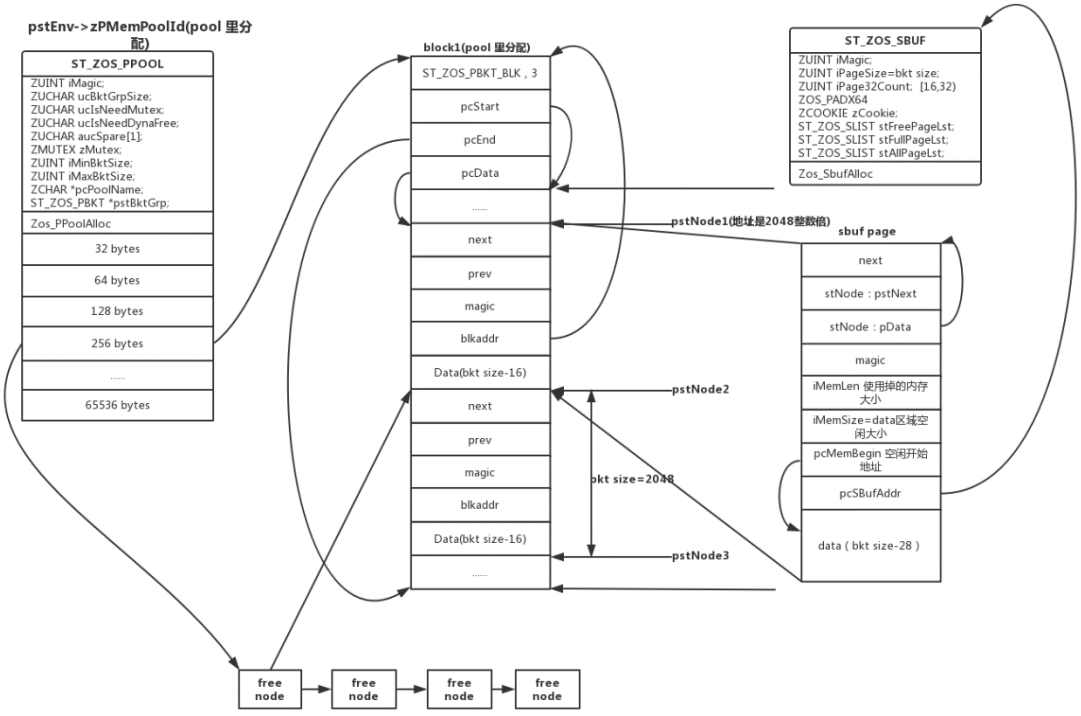
目的，这个对于通过任意 node 内部地址找 node 地址很方便。通过下面的 ZOS_PPOOL_DTOM 函数可以直接得到 node 地址。

```
/* zos power pool and the size value */
#define ZOS_PPOOL_AND_VAL(_bufsize) (~(((ZSIZE_T)_bufsize) - 1))

/* zos power pool from data to node memory */
#define ZOS_PPOOL_DTOM(_mem, _bufsize) \
    (((ZSIZE_T)(_mem) & ZOS_PPOOL_AND_VAL(_bufsize)))
```

7. Sbuf - zos sbuf library

7.1. Sbuf 结构图



7.2. Sbuf 内存介绍

7.2.1. Sbuf 内存的分配

从 stFreePageLst 的尾部找到一个 page，如果尾部没有合适的 page，找头部的 page，然后再逐个找到空闲块合适大小的 page。如果都没有合适的 page，那么分配一个新的 page，再进行内存分配。最后进行 stFreePageLst 和 stFullPageLst 的调整，然后返回内存地址。

7.2.2. Sbuf 内存的释放

一个 Sbuf 的删除，先删除 stFreePageLst 里的 page 节点，再删除 stFullPageLst 里的 page 节点，通过 Zos_PMemFree 释放内存。最后释放结构体 ST_ZOS_SBUF 变量自己的内存。

7.2.3. Sbuf 关键结构体定义

```
/* zos structure buffer page */
typedef struct tagZOS_SBUF_PAGE
{
    struct tagZOS_SBUF_PAGE *pstNext; /* next page */
    ST_ZOS_SLIST_NODE stNode;        /* node for all list */
    ZUINT iMagic;                    /* buffer magic */
    ZUINT iMemLen;                   /* memory length */
    ZUINT iMemSize;                  /* memory size */
    ZOS_PADX64
    ZCHAR *pcMemBegin;               /* memory begin */
    ZCHAR *pcSBufAddr;               /* structure buffer address */
} ST_ZOS_SBUF_PAGE;

/* zos structure buffer block */
typedef struct tagZOS_SBUF
```

```
{
    ZUINT iMagic;                   /* buffer magic */
    ZUINT iPageSize;                /* page size */
    ZUINT iPage32Count;             /* 32 bytes page count */
    ZOS_PADX64
    ZCOOKIE zCookie;                /* cookie */
    ST_ZOS_SLIST stFreePageLst;     /* free memory page list */
    ST_ZOS_SLIST stFullPageLst;     /* full memory page list */
    ST_ZOS_SLIST stAllPageLst;      /* all memory page list */
} ST_ZOS_SBUF;
```

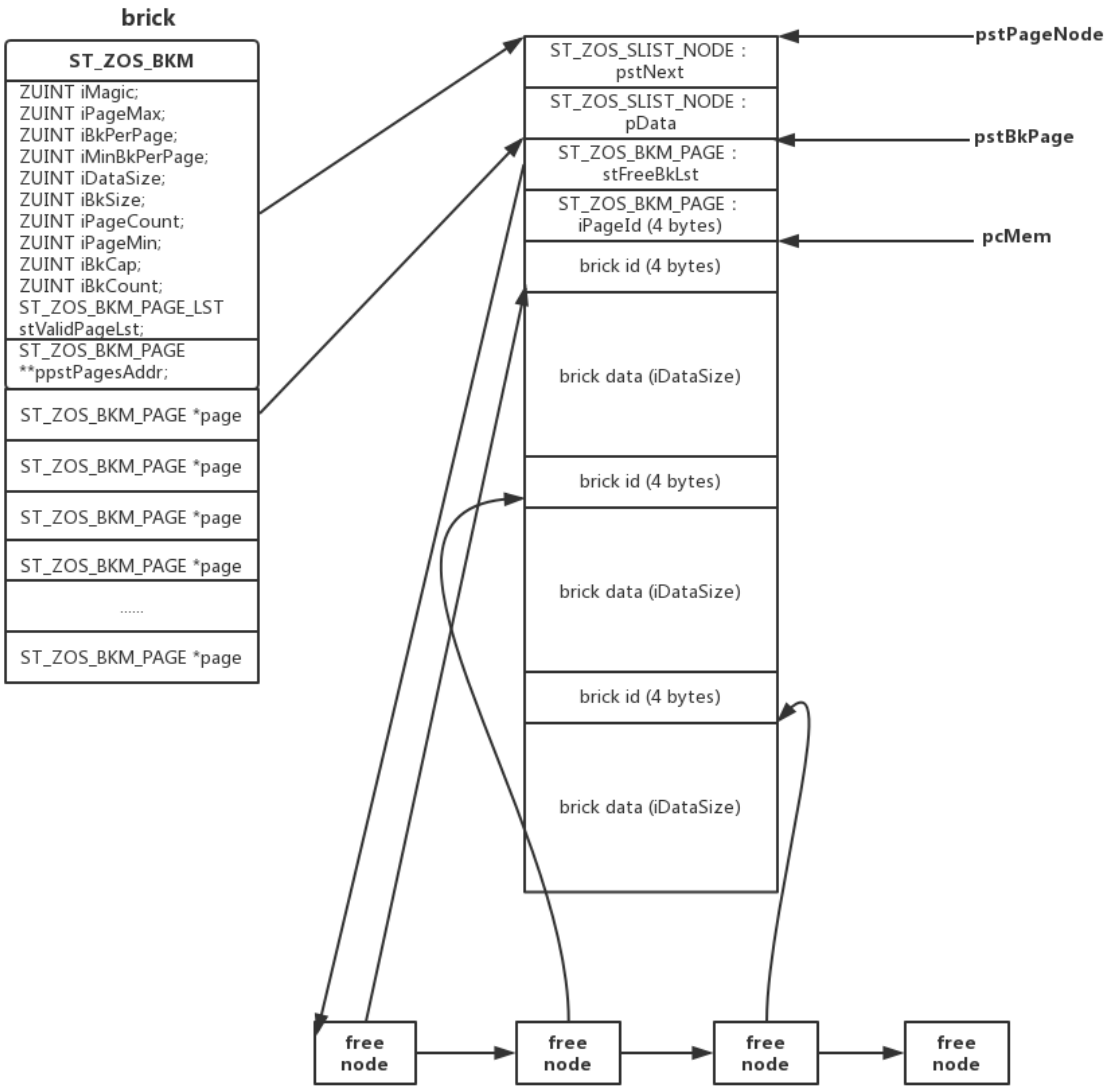
7.2.4. Sbuf 的使用

Pmem 桥接内存分配和释放函数，只是对 Ppool 做了个接口，基本上没有实际的其他动作。

Sbuf 主要在 xml DOM（ Document Object Model）（dom decdoe/encode），EAX（extension api for xml ）里用到，pbuf 没有使用的。

8. Brick- zos brick memory

8.1. Brick 结构图



8.2. Brick 内存介绍

8.2.1. Brick 内存的分配

因为 Brick 内存在创建的时候大小就定义好了，所以 Brick 内存每次分配的大小都是一样的，分配地址的同时会返回 brick id。

分配的时候先从 stValidPageLst 得到有空闲的 page 地址。然后从 stFreeBkLst 中找到当前 page 中可用的 brick。

8.2.2. Brick 内存的释放

Page id 是根据 slot id 产生的，并且置于高 16 位：

```
/* zos brick slot id to page id */
#define ZBRICK_SLOT_TO_PAGEID(_slotid) \
    (((ZUINT)(_slotid) + ZBRICK_PAGEID_START) << 16)
```

Brick id 在高 16 保存的是 page id，低 16 是当前 page 的 brick id 序号。

```
/* zos brick slot id to brick id */
#define ZBRICK_SLOT_TO_BKID(_pageid, _slotid) \
        ((_pageid) + (_slotid) + ZBRICK_BKID_START)
```

先根据内存地址得到 brick 地址，然后根据 brick id 得到 iPageSlot，在根据 iPageSlot 得到 page 的地址，把该 brick 重新放回当前 page 的 stFreeBkLst 里。 如果当前 page 原来是全部使用完的，则需要把该 page 放入 stValidPageLst，表示有当前 page 有可用的 brick。如果当前的 page 所有的 brick 都为空闲，则需要释放当前 page 的内存回系统。

8.2.3. Brick 关键结构体定义

```
/* zos brick memory manager struct definition */
```



```
typedef struct tagZOS_BKM
{
    ZUINT iMagic;                    /* brick magic */
    ZUINT iPageMax;                  /* maximum page count */
    ZUINT iBkPerPage;                /* brick count per page */
    ZUINT iMinBkPerPage;             /* minimum brick count per page */
    ZUINT iDataSize;                 /* data size */
    ZUINT iBkSize;                   /* brick size */
    ZUINT iPageCount;                /* current page count */
    ZUINT iPageMin;                  /* minimum page count */
    ZUINT iBkCap;                    /* capability of brick */
    ZUINT iBkCount;                  /* allocated brick count */
    ST_ZOS_BKM_PAGE_LST stValidPageLst; /* pages which have free brick */
    ST_ZOS_BKM_PAGE **ppstPagesAddr; /* pages address array */
} ST_ZOS_BKM;

/* zos brick page, means group of bricks */
typedef struct tagZOS_BKM_PAGE
{
    ST_ZOS_SLIST stFreeBkLst;        /* free brick list */
    ZUINT iPageId;                   /* page id */
    ZOS_PADX64
} ST_ZOS_BKM_PAGE;

/* zos brick */
typedef struct tagZOS_BKM_BRICK
{
    ZUINT iBkId;                     /* brick id */
    ZOS_PADX64
    union
    {
        ST_ZOS_BK_FREE_INFO astFreeInfo[1]; /* free brick info */
        ZUCHAR aucData[1];           /* data block memory */
    } u;
#define pstNextFreeInfo u.astFreeInfo[0].pstNext
} ST_ZOS_BKM_BRICK;
```

```
ZBRICK_TYPEDEF_LIST(ZOS_BKM_PAGE);
#define ZBRICK_TYPEDEF_LIST(_name) ZOS_TYPEDEF_SLIST(_name)

typedef struct tagZOS_BKM_PAGE_LST_NODE \
{ \
   struct tagZOS_BKM_PAGE_LST_NODE *pstNext; /**< @brief next slist node */ \
   ST_ ZOS_BKM_PAGE *pData;                  /**< @brief slist node data */ \
} ST_ZOS_BKM_PAGE_LST_NODE; \

/* single list */ \
typedef struct tagZOS_BKM_PAGE_LST \
{ \
  ZUINT iMaxNum;        /**< @brief maximum number of slist nodes */ \
  ZUINT iCount;         /**< @brief actual count of slist nodes */ \
  ST_ZOS_BKM_PAGE_LST_NODE *pstHead; /**< @brief slist node head */ \
  ST_ZOS_BKM_PAGE_LST_NODE *pstTail; /**< @brief slist node tail */ \
} ST_ZOS_BKM_PAGE_LST
```

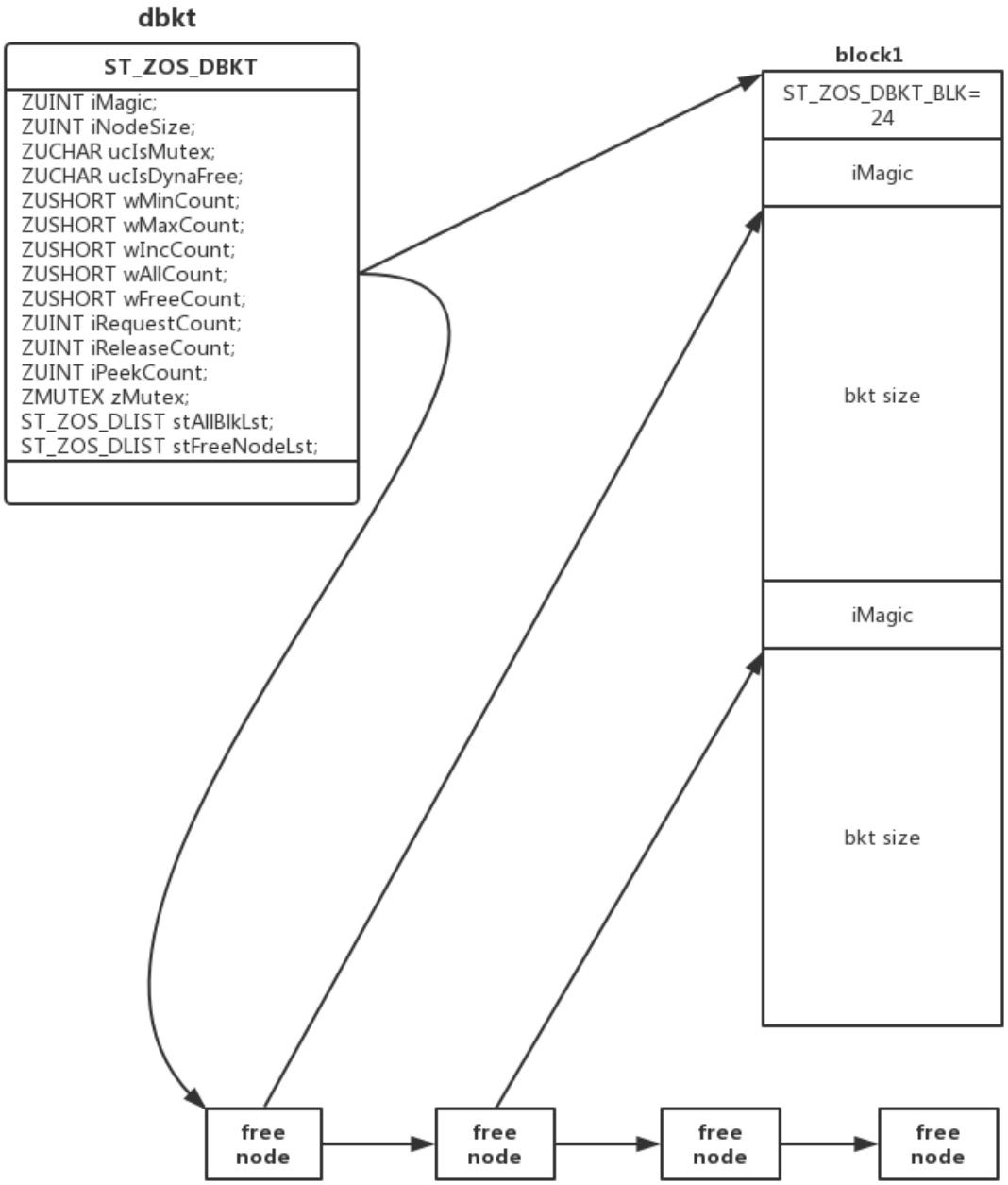
8.2.4. Brick 的使用

当前系统中使用 brick 内存有如下这些：

```
ZBKMGR zCallBk;                 /* brick memory for call */
ZBKMGR zSessBk;                 /* brick memory for session */
ZBKMGR zDlgBk;                  /* brick memory for dialog */
ZBKMGR zSubsdBk;                /* brick memory for subscription */
ZBKMGR zTransBk;                /* brick memory for transaction */
ZBKMGR zConnBk;                 /* brick memory for connection */
```

9. Dbkt - zos dynamic bucket

9.1. Dbkt 结构图



9.2. Dbkt 内存介绍

9.2.1. Dbkt 内存的分配

Zos_DbktGetBkt：从指定的 pstBkt 的 stFreeNodeLst 中得到一个 free bucket node，然后由 pstFreeInfo 得到 pstNode 地址 pstNode = ZOS_DBKT_FREE_TO_NODE(pstFreeInfo);只是减去 4 个字节的 iMagic 空间，再返回 pstNode 的内存地址 ZOS_DBKT_NODE_DATA_PTR(pstNode)给上层应用。

9.2.2. Dbkt 内存的释放

Zos_DbktPutBkt：从内存地址得到 node 地址，再从 node 地址得到 pstFreeInfo 地址。把 pstFreeInfo 重新插入 stFreeNodeLst 中。从指定的 pstBkt 的查找 node 所在的 block，恢复 node 的 iMagic 和 astFreeInfo 信息。如果当前 block 所有的 node 都释放完，释放当前 block 回系统。

9.2.3. Dbkt 关键结构体定义

```
/* zos bucket */
typedef struct tagZOS_DBKT
{
    ZUINT iMagic;                   /* magic value */
    ZUINT iNodeSize;                /* bucket node size */
    ZUCHAR ucIsMutex;               /* is mutex lock */
    ZUCHAR ucIsDynaFree;            /* is dynamic free memory */
    ZUSHORT wMinCount;              /* minimum bucket count */
    ZUSHORT wMaxCount;              /* maixmum bucket count */
    ZUSHORT wIncCount;              /* inc bucket count per time */
    ZUSHORT wAllCount;              /* all bucket count */
    ZUSHORT wFreeCount;             /* free bucket count */
    ZUINT iRequestCount;            /* requested count */
    ZUINT iReleaseCount;            /* released count */
    ZUINT iPeekCount;               /* maximum requested count */
    ZMUTEX zMutex;                  /* bucket manager mutex */
    ST_ZOS_DLIST stAllBlkLst;       /* all bucket block list */
```



```
    ST_ZOS_DLIST stFreeNodeLst;      /* bucket free node list */
} ST_ZOS_DBKT;

/* zos bucket memory block */
typedef struct tagZOS_DBKT_BLK
{
    struct tagZOS_DBKT_BLK *pstNext; /* next bucket memory block */
    struct tagZOS_DBKT_BLK *pstPrev; /* previous bucket memory block */
    ZUINT iMagic;                    /* bucket node magic */
    ZUSHORT wAllCount;               /* all bucket count */
    ZUSHORT wFreeCount;              /* free bucket count */
    ZCHAR *pcStart;                  /* bucket start memory after aligned */
    ZCHAR *pcEnd;                    /* bucket end memory after aligned */
} ST_ZOS_DBKT_BLK;

/* zos bucket free node information */
typedef struct tagZOS_DBKT_FREE_INFO
{
    struct tagZOS_DBKT_FREE_INFO *pstNext; /* next bucket node */
    struct tagZOS_DBKT_FREE_INFO *pstPrev; /* previous bucket node */
    ST_ZOS_DBKT_BLK *pstBlk;         /* bucket block */
} ST_ZOS_DBKT_FREE_INFO;

/* zos bucket node */
typedef struct tagZOS_DBKT_NODE
{
    ZUINT iMagic;                    /* bucket node magic */
    ZOS_PADX64
    union
    {
        ST_ZOS_DBKT_FREE_INFO astFreeInfo[1]; /* free node infomation */
        ZUCHAR aucData[1];           /* data block memory */
    } u;
#define pstNextFreeInfo u.astFreeInfo[0].pstNext
} ST_ZOS_DBKT_NODE;
```


9.2.4. Dbkt 的使用

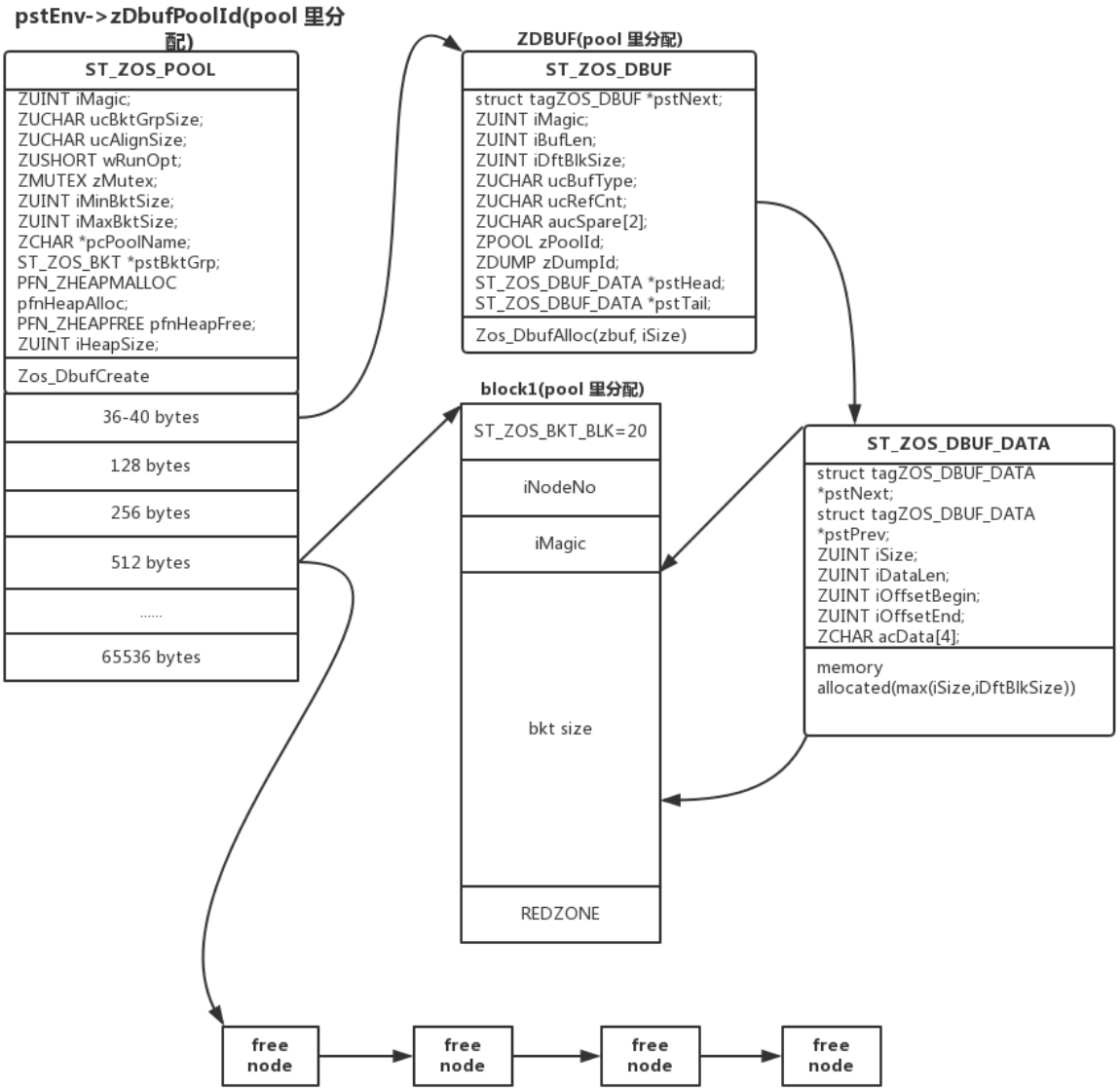
Dbkt 当前主要用在 zos priority queue 和 zos queue library 里面。

10. Ubuf - zos unify buffer library

Ubuf 是统一缓冲区接口，为多种 buf 提供一个统一接口用的，其中支持的类型包括：Cbuf，Dbuf，Pbuf，Sbuf 等。

11. Dbuf - zos dbuf library

11.1. Dbuf 结构图



11.2. Dbuf 内存介绍

11.2.1. Dbuf 内存的创建

系统给 dbuf 单独创建了一个内存池，pstEnv->zDbufPoolId，每一个 dbuf 创建的时候，会创建一个结构体 ST_ZOS_DBUF，保持该 dbuf 的 iDftBlkSize 值，为后续内存分配做准备。

11.2.2. Dbuf 内存的分配

Dbuf 内存分配需要指定新建时候的 ST_ZOS_DBUF 结构体指针和需要的内存大小，可以通过不同的接口使用 pstBuf->pstHead 或者 pstBuf-> pstTail 优先的内存分配方式。内存分配会从 ST_ZOS_DBUF_DATA 结构的后面或者前面查找可用内存区，如果找到直接修改当前 ST_ZOS_DBUF_DATA 的数据长度和边界值，返回内存地址。如果没有满足条件的数据块，则从分配一个新的 ST_ZOS_DBUF_DATA，再进行内存分配。

下面是 dbuf 的 bucket 配置信息，对于不同的 bucket size，会有不同的 increment count--也就是每个 block 中包含 node 的数量。

```
/* zos default dbuf bucket config info group */
ZCONST ST_ZOS_BKT_INFO m_astZosCfgDbufBktInfoGrp[] =
{
      /* size,                  maximum count,          increment count */
      {sizeof(ST_ZOS_DBUF),      10,                          4},
      {128,                      0,                           2},
      {256,                      0,                           4},
      {512,                      0,                           4},
      {1024,                     0,                           4},
      {2048,                     0,                           2},
      {4096,                     0,                           2},
      {5120,                     0,                           2},
      {8192,                     0,                           1},
      {16384,                    0,                           1},
      {32768,                    0,                           1},
      {65536,                    0,                           1}
};
```


11.2.3. Dbuf 内存的释放

Dbuf 是逻辑地址连续的缓冲区，一个可能多次申请的内存，但是释放的时候是一次性全部释放，中间有 ref 值来决定是不是需要真正的释放内存。释放函数是 Zos_DbufDelete。

11.2.4. Dbuf 关键结构体定义

```
/* zos data buffer block */
typedef struct tagZOS_DBUF
{
    struct tagZOS_DBUF *pstNext;     /* next data buffer */
    ZUINT iMagic;                    /* magic value */
    ZUINT iBufLen;                   /* buffer used length */
    ZUINT iDftBlkSize;               /* default data block size in
buffer */
    ZUCHAR ucBufType;                /* buffer mode ZDBUF_TYPE_BYTE...
*/
    ZUCHAR ucRefCnt;                 /* buffer reference count */
    ZUCHAR aucSpare[2];              /* for 32 bit alignment */
    ZPOOL zPoolId;                   /* memory pool id for dbuf alloc */
    ZDUMP zDumpId;                   /* stack dump */
    ST_ZOS_DBUF_DATA *pstHead;       /* the first data block in buffer
*/
    ST_ZOS_DBUF_DATA *pstTail;       /* the last data block in buffer */
} ST_ZOS_DBUF;
```

ST_ZOS_DBUF_DATA 头有 24 字节，这个 24 字节是在包含在 node 的 bucket size 里面的。

```
/* zos data buffer data block */
typedef struct tagZOS_DBUF_DATA
{
    struct tagZOS_DBUF_DATA *pstNext;/* next data block */
    struct tagZOS_DBUF_DATA *pstPrev;/* previous data block */
```

```
    ZUINT iSize;                    /* block size */
    ZUINT iDataLen;                 /* data length in block */
    ZUINT iOffsetBegin;             /* data begin offset in block */
    ZUINT iOffsetEnd;               /* data end offset in block */
    ZCHAR acData[4];                /* data memory */
    ZOS_PADX64
} ST_ZOS_DBUF_DATA;
```

11.2.5. Dbuf 提供的接口

Dbuf 提供了切分合并等非常丰富的操作接口，比如 cat，split，preADD，pstADD 等等。

12. VoWiFi 协议栈内存相关的优化项

12.1. Dbuf ZOS_DBUF_SIZEOF_DATA（24 字节）导致的内存浪费 25.6%

12.1.1. 内存浪费原因分析

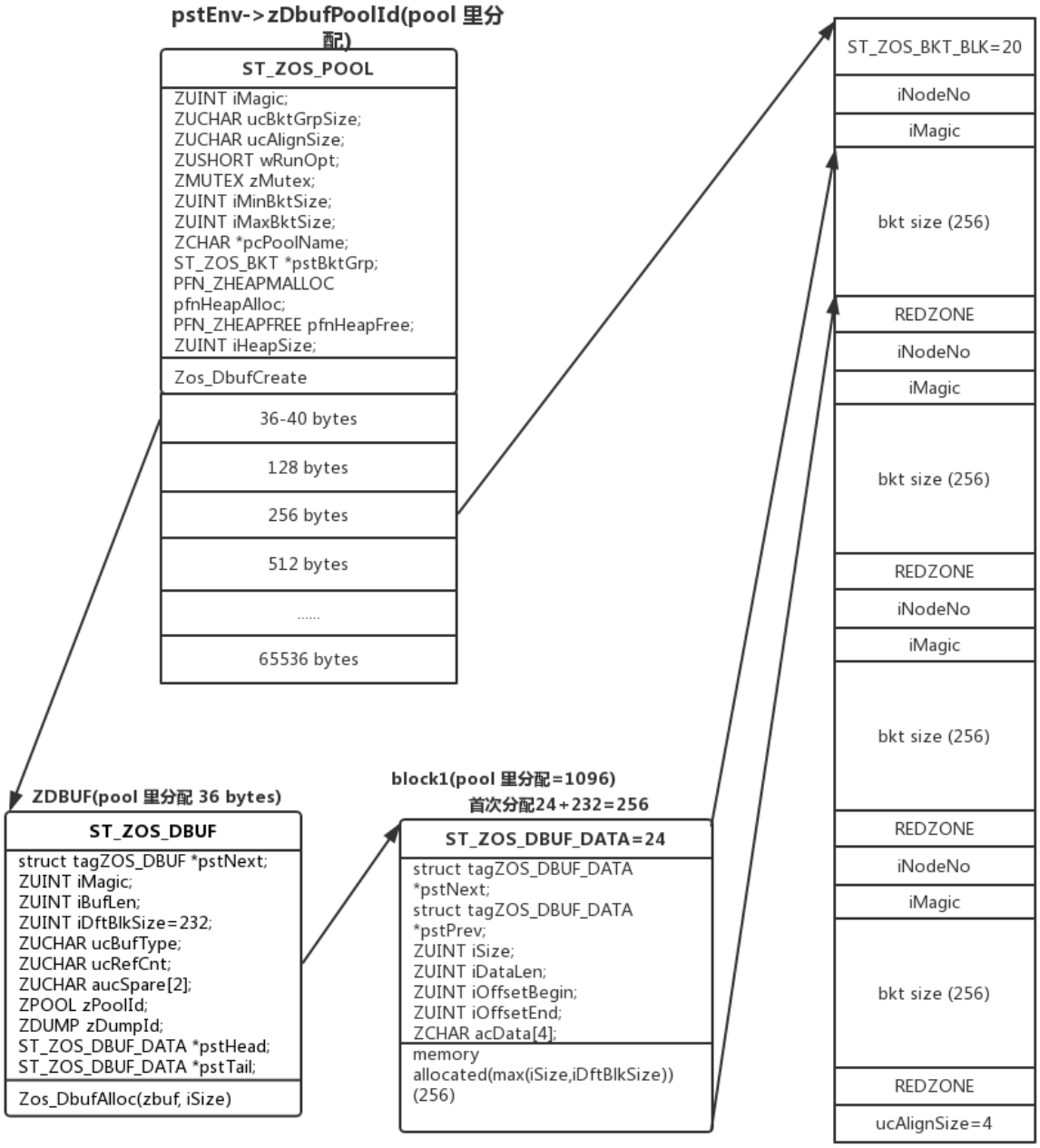
当 dbuf 的 iDftBlkSize 和 pool 的 bkt size 大小匹配时，dbuf 还有带 ST_ZOS_DBUF_DATA=24 的头字节，这样会造成请求空间很多是 bkt size+24，导致 pool 不得不分配下一个 bkt size，只是因为 24 字节的头；

但是 dbuf 请求的不一定是刚好 dbuf bkt size 的内存吗？请求数据是任意大小的，可是实际分配大小是 iAllocSize = (iSize > pstBuf->iDftBlkSize) ? iSize : pstBuf->iDftBlkSize; 而一般用户设置 iDftBlkSize 为：128,256,512,1024,2048,4096。而一般用户申请的 iSize 是结构体，如 pstBranch = SIP_DBUF_ALLOC(pstSess->zMemBuf, sizeof(ST_ZOS_SSTR)); 都是小于 iDftBlkSize，所以绝大部分实际分配的内存大小都是 iAllocSize=iDftBlkSize；

_mem = Zos_PoolAlloc((_buf)->zPoolId, ZOS_DBUF_DATA_SIZE(_size))，这里内存分配的大小为：#define ZOS_DBUF_DATA_SIZE(_size) (_size + ZOS_DBUF_SIZEOF_DATA=24)，也就是绝大部分都是 iDftBlkSize+24；

实际使用例子：
Ike_WiresharkDecodeStringDump ： zData = Zos_DbufCreate(ZNULL, ZDBUF_TYPE_BYTE, 256); //iDftBlkSize=256
Zos_DbufPstAddStrD(zData, "SECURITY IKE_KEY:"); //size=17
iAllocSize=iDftBlkSize+24=280；//这里分配的 280 字节不会浪费，虽然这次只是用了 17 个字节，但是下次的申请，我继续用剩下的 280-17 的内存；对于这种情况，pool 对于 280 内存的请求会分配 512 字节的内存，而这个 512-280=232 的内存是浪费的，永远不会被用到，直到释放；

12.1.2. 解决方案



如上图所示，在初始化 dbuf 时，把 iDftBlkSize 设为 bucket size（256bytes） - ZOS_DBUF_SIZEOF_DATA（24bytes）=232 bytes，当进行内存分配的时候，如果申请的内存很小，会直接使用 iDftBlkSize 值，然后再加上 ZOS_DBUF_SIZEOF_DATA 长度，刚好匹配 bucket size 值，不会造成没必要的内存浪费。

这里 iDftBlkSize 减小不会对分配内存产生影响，这个只是每次分配的最小值而已。

代码提交如下：
http://review.source.spreadtrum.com/gerrit/#/c/502438/

12.1.3. 优化前后测试对比

优化前 VoWiFi 注册达到稳态后 Dbuf 内存使用情况：

```
>df
   BUCKET     TOTAL      FREE  PEEK ALLOC    REQUEST    RELEASE
   ------     -----      ----  ----------    -------    -------
     104        0         0        0         0          0
     232        0         0        1         1          1
     488       12         6       12        93         87
    1000        0         0        2        38         38
    2024       28         6       49        93         71
    4072        0         0       14        97         97
    5096        4         0        4         4          0
    8168        0         0        0         0          0
   16360        0         0        0         0          0
   32744        0         0        1         2          2
   65512        0         0        0         0          0

              total      free      peek      request      release
Dbuf:          18         7        26         191          180

              total       used       free
Mem:          83968      68608      15360

return = 0x00000000
```

优化后 VoWiFi 注册达到稳态后 Dbuf 内存使用情况：

```
>df
   BUCKET     TOTAL      FREE  PEEK ALLOC    REQUEST    RELEASE
   ------     -----      ----  ----------    -------    -------
     104        0         0        1         1          1
     232        4         3        6        83         82
     488       12         7       11        47         42
```

1000	36	14	46	95	73
2024	0	0	14	101	101
4072	2	0	3	3	1
5096	2	0	2	2	0
8168	0	0	0	0	0
16360	0	0	0	0	0
32744	0	0	1	2	2
65512	0	0	0	0	0

	total	free	peek	request	release
Dbuf:	18	7	24	194	183

	total	used	free
Mem:	62464	43776	18688

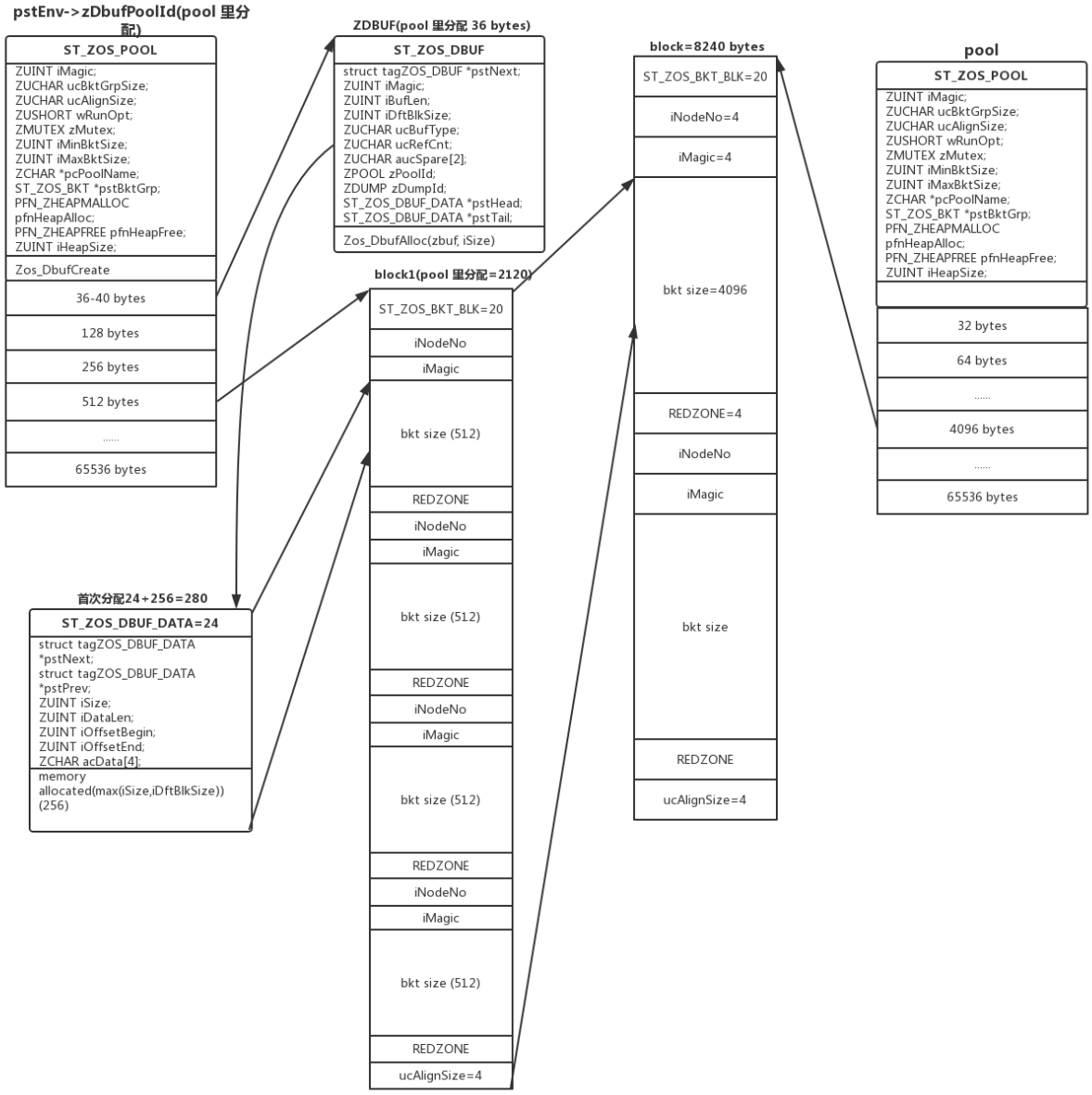
return = 0x00000000

对比优化前后，**dbuf** 对内存的需求降低了：**(83968-62464)/83968=25.6%**

12.2. Dbuf 双重 pool 分配内存浪费问题 48%

12.2.1. 内存浪费原因分析

Dbuf 双重 pool 分配结构如下图所示：



如图所示，dbuf 在分配内存时使用的是 pool 的分配方式，然后 dbuf 的配置方式为：pstCfg->stDbuf.pfnHeapAlloc = Zos_Malloc; 在从系统分配的时候，dbuf 再一次使用了 Zos_Malloc，该分配函数会调用 Zos_PoolAlloc，这个函数会再一次的使用 pool 方式，从 pstEnv->zPMemPoolId 里分配一次内存，这样会造成没有必要的双重 pool 内存分配方式。

代码提交如下：

http://review.source.spreadtrum.com/gerrit/#/c/503503/

12.2.2. 解决方案

pstCfg->stDbuf.pfnHeapAlloc = Zos_Malloc;
pstCfg->stDbuf.pfnHeapFree = Zos_Free;
把这两个修改 ZNULL，使用系统 use heap default 分配内存。

12.2.3. 优化前后测试对比

优化前 VoWiFi 注册达到稳态后 pool 内存使用量：

>mm

BUCKET	TOTAL	FREE	PEEK ALLOC	REQUEST	RELEASE
32	192	66	140	680	554
64	640	82	662	2995	2437
128	176	18	169	370	212
256	64	10	63	288	234
512	40	5	46	147	112
1024	16	2	15	33	19
2048	24	2	22	40	18
4096	70	0	72	166	96
5120	14	3	16	105	94
8192	2	0	3	4	2
16384	5	0	6	6	1
32768	0	0	0	0	0
65536	8	0	10	13	5

	total	used	free	
Mem:	1153024	1116736	36288	-1032192=120832
Heap:	352120			

优化后 VoWiFi 注册达到稳态后 pool 内存使用量：

>mm

BUCKET	TOTAL	FREE	PEEK ALLOC	REQUEST	RELEASE
32	192	68	140	636	512
64	640	84	636	2431	1875

128	160	3	160	339	182
256	56	5	57	244	193
512	36	2	40	103	69
1024	12	0	12	26	14
2048	20	0	20	33	13
4096	68	1	68	139	72
5120	2	0	2	6	4
8192	2	0	3	4	2
16384	3	0	3	3	0
32768	0	0	0	0	0
65536	8	0	10	11	3

	total	used	free
Mem:	1032192	1017856	14336
Heap:	352120		

VoWiFi 注册达到稳态后 dbuf 实际申请的内存使用量：

>df

BUCKET	TOTAL	FREE	PEEK ALLOC	REQUEST	RELEASE
104	0	0	1	1	1
232	4	3	6	84	83
488	12	7	11	45	40
1000	36	14	46	95	73
2024	0	0	14	99	99
4072	2	0	3	3	1
5096	2	0	2	2	0
8168	0	0	0	0	0
16360	0	0	0	0	0
32744	0	0	1	2	2
65512	0	0	0	0	0

	total	free	peek	request	release
Dbuf:	18	7	24	190	179

	total	used	free

Mem: 62464 43776 18688

return = 0x00000000

对比优化前后，pool 对内存的需求降低了：
1153024-1032192=120832，也就是说 dbuf 申请 62464 的内存，实际 pool 用了 120832 的内存消耗。
(120832-62464)/ 120832=48%

12.3. 用系统的 jemalloc 取代 pool 进行分配内存优化 23.4%

12.3.1. 用 jemalloc 取代的必要性

Jemalloc 是 android 默认的内存分配函数，是整合在 libc 的函数；
Jemalloc 包含了 zos pool 的功能并且更加高效，内存使用效率也更高。

12.3.2. Pool 潜在的问题与调试缺陷

Pool 发现了很多链表指针操作问题，内存被踩的问题，pool 基本没有提供相应的内存问题调试机制，这些都是 pool 的缺陷，如果取代后可以直接使用 jemalloc 的强大的内存调试工具。

12.3.3. Jemalloc 取代前后的内存使用比

Pool 存在时，jemalloc 的内存使用统计如下：

___ Begin jemalloc statistics ___
Version: 4.4.0-0-gf1f76357313e7dcad7262f17a48ff0a2e005fcdc
Assertions disabled
config.malloc_conf: ""
Run-time option settings:
  opt.abort: false
  opt.lg_chunk: 19
  opt.dss: "secondary"
  opt.narenas: 1
  opt.purge: "decay"


```
  opt.decay_time: 0 (arenas.decay_time: 0)
  opt.junk: "false"
  opt.quarantine: 0
  opt.redzone: false
  opt.zero: false
  opt.stats_print: false
Arenas: 1
Unused dirty page decay time: 0
Quantum size: 8
Page size: 4096
Allocated: 1907032, active: 2351104, metadata: 141848, resident: 2482176, mapped:
3670016, retained: 0
Current active ceiling: 2621440

arenas[0]:
assigned threads: 11
dss allocation precedence: disabled
decay time: 0
purging: dirty: 0, sweeps: 52, madvises: 52, purged: 296
                     allocated      nmalloc      ndalloc    nrequests
small:                  764248          585          356          585
large:                 1142784           20            5           20
huge:                        0            0            0            0
total:                 1907032          605          361          605
active:                2351104
mapped:                3145728
retained:                    0
metadata: mapped: 73728, allocated: 11480
bins:           size ind     allocated      nmalloc      ndalloc    nrequests       curregs
curruns regs pgs  util
  newruns       reruns
                8   0          136           25            8           25           17            1  512   1 0.033
      1            0
               16   1          112           11            4           11            7            1  256   1 0.027
      4            0
               24   2           48            5            3            5            2            1  512   3 0.003
```

```
      1        0
             32  3       192       9       3       9       6       1  128   1 0.046
      1        0
             40  4       200       100     95      100     5       1  512   5
0.009
      1        0
             48  5       144       3       0       3       3       1  256   3 0.011
      1        0
             56  6       168       4       1       4       3       1  512   7 0.005
      1        0
             64  7       64        1       0       1       1       1  64    1 0.015
      1        0
             80  8       400       5       0       5       5       1  256   5 0.019
      1        0
             96  9       480       5       0       5       5       1  128   3 0.039
      1        0
            112  10      224       2       0       2       2       1  256   7 0.007
      1        0
            128  11      1536      27      15      27      12      1   32   1
0.375
      1        0
            160  12      160       1       0       1       1       1  128   5 0.007
      1        0
            192  13      192       1       0       1       1       1  64    3 0.015
      1        0
            224  14      448       15      13      15      2       1  128   7
0.015
      1        0
             ---
            320  16      2880      21      12      21      9       1   64   5
0.140
      1        0
            384  17      384       2       1       2       1       1   32   3 0.031
      2        0
             ---
            512  19      2560      7       2       7       5       1    8   1 0.625
```



```
        1        0
              640  20       4480        7        0        7        7        1   32   5 0.218
        1        0
                ---
              896  22       2688        3        0        3        3        1   32   7 0.093
        1        0
             1024  23       1024        1        0        1        1        1    4   1 0.250
        1        0
             1280  24       7680       12        6       12        6        1   16   5
0.375
        1        0
                ---
             1792  26       5376        3        0        3        3        1   16   7 0.187
        1        0
                ---
             2560  28     140800      109       54      109       55        8    8   5
0.859
       13        6
             3072  29       9216        6        3        6        3        1    4   3 0.750
        1        0
                ---
             4096  31       4096        2        1        2        1        1    1   1 1
        2        0
             5120  32      66560      101       88      101       13        4    4   5
0.812
       17       13
             6144  33       6144        1        0        1        1        1    2   3 0.500
        1        0
                ---
            10240  36     481280       91       44       91       47       24    2   5
0.979
       45        3
            12288  37      24576        5        3        5        2        2    1   3 1
        5        0
                ---
large:       size ind    allocated      nmalloc      ndalloc    nrequests      curruns
```

```
                ---
              20480  40       61440          3          0          3          3
                ---
              40960  44           0          2          2          2          0
                ---
              81920  48      655360         11          3         11          8
              98304  49      196608          2          0          2          2
             114688  50      229376          2          0          2          2
                ---
huge:         size ind    allocated      nmalloc     ndalloc    nrequests   curhchunks

>mm

   BUCKET      TOTAL       FREE  PEEK ALLOC    REQUEST    RELEASE
   ------      -----       ---- ----------    -------    -------
     32        192          64        143        812        684
     64        704         139        694       4378       3813
    128        160           3        176        416        259
    256         64          12         66        310        258
    512         44          10         44        288        254
   1024         16           2         14         30         16
   2048         20           2         21         33         15
   4096         72           0         73        149         77
   5120          2           0          2          6          4
   8192          2           0          3          4          2
  16384          3           0          3          3          0
  32768          0           0          0          0          0
  65536          8           0         10         11          3

            total       used       free
Mem:      1062912    1037248      25664
Heap:      384136
```

Pool 被取代后，直接用 jemalloc 的统计如下：

___ Begin jemalloc statistics ___
Version: 4.4.0-0-gf1f76357313e7dcad7262f17a48ff0a2e005fcdc
Assertions disabled
config.malloc_conf: ""
Run-time option settings:
  opt.abort: false
  opt.lg_chunk: 19
  opt.dss: "secondary"
  opt.narenas: 1
  opt.purge: "decay"
  opt.decay_time: 0 (arenas.decay_time: 0)
  opt.junk: "false"
  opt.quarantine: 0
  opt.redzone: false
  opt.zero: false
  opt.stats_print: false
Arenas: 1
Unused dirty page decay time: 0
Quantum size: 8
Page size: 4096
Allocated: 1459928, active: 1916928, metadata: 130240, resident: 2035712, mapped: 3145728, retained: 0
Current active ceiling: 2097152

arenas[0]:
assigned threads: 16
dss allocation precedence: disabled
decay time: 0
purging: dirty: 0, sweeps: 88, madvises: 88, purged: 278

	allocated	nmalloc	ndalloc	nrequests
small:	476888	8045	6867	8045
large:	983040	20	5	20
huge:	0	0	0	0
total:	1459928	8065	6872	8065
active:	1916928			
mapped:	2621440			

retained: 0
metadata: mapped: 61440, allocated: 12160
bins: size ind allocated nmalloc ndalloc nrequests curregs curruns regs pgs util newruns reruns
8 0 176 50 28 50 22 1 512 1 0.042 1 0
16 1 544 291 257 291 34 1 256 1 0.132 4 0
24 2 2280 466 371 466 95 1 512 3 0.185 1 0
32 3 448 162 148 162 14 1 128 1 0.109 1 0
40 4 840 946 925 946 21 1 512 5 0.041 1 0
48 5 11856 1660 1413 1660 247 2 256 3 0.482 8 25
56 6 6776 1245 1124 1245 121 1 512 7 0.236 1 0
64 7 11840 1528 1343 1528 185 4 64 1 0.722 20 420
80 8 2320 64 35 64 29 1 256 5 0.113 1 0
96 9 2016 53 32 53 21 1 128 3 0.164 1 0
112 10 1120 25 15 25 10 1 256 7 0.039

1 0
128 11 16128 371 245 371 126 5 32 1
0.787
5 16
160 12 1440 90 81 90 9 1 128 5
0.070
1 0
192 13 1152 108 102 108 6 1 64 3
0.093
1 0
224 14 5600 57 32 57 25 1 128 7
0.195
1 0
256 15 4096 77 61 77 16 1 16 1 1
22 1
320 16 5440 46 29 46 17 1 64 5
0.265
1 0
384 17 5760 68 53 68 15 1 32 3
0.468
1 0
448 18 2688 17 11 17 6 1 64 7
0.093
2 0
512 19 5632 332 321 332 11 2 8 1
0.687
3 2
640 20 7680 13 1 13 12 1 32 5
0.375
1 0
768 21 3840 9 4 9 5 1 16 3 0.312
1 0
896 22 4480 7 2 7 5 1 32 7 0.156
1 0
1024 23 3072 12 9 12 3 1 4 1 0.750
1 0

1280 24 23040 34 16 34 18 2 16 5
0.562
2 2
1536 25 1536 3 2 3 1 1 8 3 0.125
1 0
1792 26 10752 8 2 8 6 1 16 7 0.375
1 0
2048 27 4096 3 1 3 2 1 2 1 1
1 0
2560 28 186880 168 95 168 73 10 8 5
0.912
18 1
3072 29 3072 3 2 3 1 1 4 3 0.250
2 0

4096 31 12288 11 8 11 3 3 1 1 1
11 0
5120 32 56320 106 95 106 11 4 4 5
0.687
19 10
6144 33 18432 5 2 5 3 2 2 3 0.750
3 0

10240 36 40960 6 2 6 4 2 2 5 1
4 0
12288 37 12288 1 0 1 1 1 1 3 1
1 0
---
large: size ind allocated nmalloc ndalloc nrequests curruns
16384 39 49152 3 0 3 3

40960 44 0 2 2 2 0
49152 45 49152 1 0 1 1

65536 47 458752 10 3 10 7

```
            98304  49       196608          2          0          2          2
           114688  50       229376          2          0          2          2
             ---
huge:          size ind    allocated      nmalloc    ndalloc    nrequests  curhchunks
             ---
```

(1907032-1459928)/1907032=447104/1907032=23.4%

Pool 被取代后节约 **23.4%**的内存空间。

修复如下：

http://review.source.spreadtrum.com/gerrit/#/c/525264/

12.4. Zos_DlistRemove/Zos_PoolAlloc 的 crash 问题

Zos_DlistRemove crash 有 Bug 911008/809274/572937，Zos_PoolAlloc crash 有 bug 915024/839335/863489，这两个 crash 的 root cause 都是操作 pool 的 stFreeNodeLst/ stAllocLst 导致的问题，这个原因可以在 dlist 加 mutex 来修复。

修复如下：

http://review.source.spreadtrum.com/gerrit/#/c/525099/

12.5. IKE stMsgList 释放的踩内存地址问题分析

12.5.1. IKE stMsgList 内存分配方式

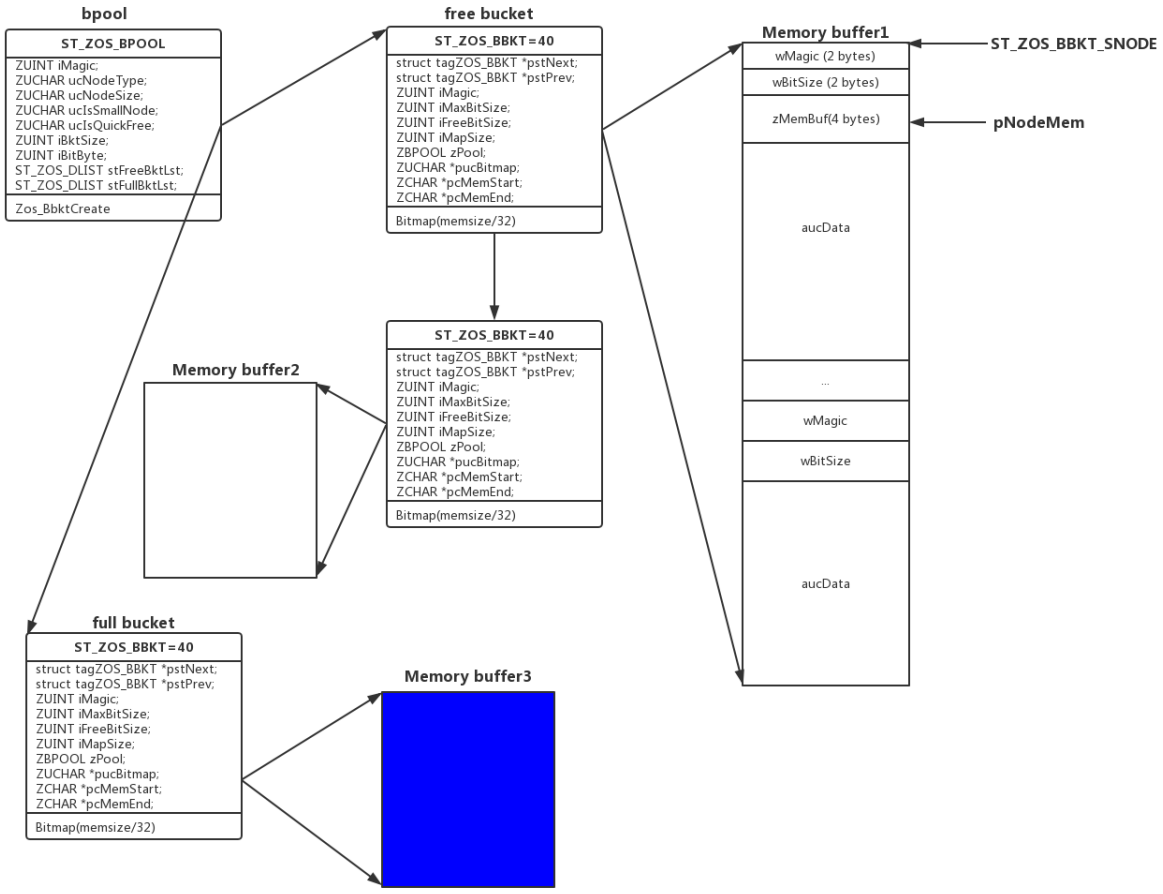
```
        zMemBuf = Zos_CbufCreateXClrd(pstSess->zMemBuf, 128,
sizeof(ST_IKE_MSG),(ZVOID **)&pstMsg);
        /* get the session */
        pstMsg->zMemBuf = zMemBuf;
        pstMsg->iMsgId = ++pstEnv->iMsgId;
        pstMsg->iSessId = pstSess->iSessId;
```

pstMsg 的结构体内存是在 zMemBuf 里分配的，所以在内存 delete 后进行的 pstMsg->zMemBuf = NULL;操作是在已释放内存区，这是非法的操作。

```
        Zos_CbufDelete(pstMsg->zMemBuf);
```

```
pstMsg->zMemBuf = NULL;
pstMsg = NULL;
```

12.5.2. pstMsg->zMemBuf 地址在 pool 内存的位置



如图所示，bpool 的 memory buffer 头两个字节是 magic，再后面两个字节是 bitsize，这四个字节对应于 pool 的 ST_ZOS_BKT_FREE_INFO 的 pstNext，zMemBuf 是 pstMsg 第一个成员，所以刚好对应到 ST_ZOS_BKT_FREE_INFO 的 pstPrev。

```
typedef struct tagZOS_BKT_FREE_INFO
{
   struct tagZOS_BKT_FREE_INFO *pstNext; /* next bucket node */
   struct tagZOS_BKT_FREE_INFO *pstPrev; /* previous bucket node */
   ST_ZOS_BKT_BLK *pstBlk;          /* bucket block */
   ZUINT iMagic;                    /* free info magic */
```

```
    ZOS_PADX64
} ST_ZOS_BKT_FREE_INFO;
```

因为在内存释放后，zos pool 会置空块连接和其他相关信息，而你对该内存的操作很可能会破坏空块链接和相关信息，所以如果做了 pstMsg->zMemBuf = NULL;就相当于把 pool 的 stFreeNodeLst 的刚释放的节点的 pstPrev 清空，会导致 dlist 双向链表断裂。

12.5.3. IKE stMsgList 修复及预防

所以对于类似结构体本身的首地址就是本次分配的地址，并且把分配的 CBUF 操作句柄存在该结构体中时，特别需要注意内存删除后不能再次操作该结构体地址。

包含自己的分配方法，pstMsg->zMemBuf 在分配区，这种情况删除后置空是极其危险的：

```
zMemBuf = Zos_CbufCreateXClrd(pstSess->zMemBuf, 128, sizeof(ST_IKE_MSG),
                              (ZVOID **)&pstMsg);
pstMsg->zMemBuf = zMemBuf;
Zos_CbufDelete(pstMsg->zMemBuf);
pstMsg->zMemBuf = NULL;
```

下面这些分配方式的地址 pstAuth->zMemBuf/ pstAuth->zCredBuf 不在分配区，所以可以放心的置 NULL；

```
pstAuth->zMemBuf = Zos_CbufCreateX(zMemBuf, 128);
pstAuth->zCredBuf = Zos_CbufCreateX(pstAuth->zMemBuf, 64);

Zos_CbufDelete(pstAuth->zMemBuf);
pstAuth->zMemBuf=NULL;

Zos_CbufDelete(pstAuth->zCredBuf);
pstAuth->zCredBuf =NULL;
```

修复如下：

http://review.source.spreadtrum.com/gerrit/#/c/525040/

12.6. Dbuf 的 dump crash 问题

Dbuf 的 dump 部分 crash，是因为多线程操作 dump list 导致的，这边没有做多线程互斥访问。
修复如下：
http://review.source.spreadtrum.com/gerrit/#/c/459797/1/src/zos/zos_dump.c

12.7. Juphoon 修改 Dbuf dump crash 引入的 bad fix

这个 bad fix 会导致很大一部分 dbuf 内存永远得不到释放，会滞留在内存。同时，这个也不能完全避免 crash，因为有部分其他分支的 dbuf 在释放的时候还是存在 crash 的可能。

修复提交如下：
http://review.source.spreadtrum.com/gerrit/#/c/459797/1/src/protocol/sip/sip_core_ua.c

12.8. Udp 接收缓冲区内存和 pstEvnt->pstMsg 内存的不释放问题

pstEvnt->pstMsg 因为多 clone 了一次，导致永远得不到释放。
Udp 接收的 buf 缓冲区的内存泄漏问题。
http://review.source.spreadtrum.com/gerrit/#/c/471550/

12.9. Brick 空指针问题

```
ZBKDATA Zos_BkEnum(ZBKMGR zBkMgr, ZUINT iIndex)
pstPage = pstBkm->ppstPagesAddr[i]; //空指针的检测
    if (pstBkm->ppstPagesAddr[iPageId] == ZNULL)
        continue;
```


12.10. Dump,fsmDump 的多线程访问 wStackNum 问题

多线程访问会导致 wStackNum 数据异常，进而使得 dump 调试数据异常，带来潜在的 crash 风险，对于调试和定位问题带来不利影响。
http://review.source.spreadtrum.com/gerrit/#/c/502759/

12.11. ERROR: SysStrFree invalid magic.

当 URI 的获取方式改变，内存不再是由 Zos_SysStrAlloc 分配时，不能由 SysStrFree 去释放。该内存没有新分配，所以这里不需要释放内存。

http://review.source.spreadtrum.com/gerrit/#/c/516063/

12.12. 几个因为提供错误内存句柄导致的释放失败问题

http://review.source.spreadtrum.com/gerrit/#/c/525100/1/src/framework/mrf/mrf_reg.c

```
Sip_CpyCallId(pstRegIpsecSA->stAuth.zMemBuf, &pstRegIpsecSA->stAuth.stCallId, &pstActRegIpsecSA->stAuth.stCallId);
Msf_StrReplaceX(pstRegIpsecSA->stAuth.zMemBuf, &pstRegIpsecSA->stAuth.stPasswd, &pstActRegIpsecSA->stAuth.stPasswd);
Msf_StrReplaceX(pstRegIpsecSA->stAuth.zMemBuf, &pstRegIpsecSA->stAuth.stNonce, &pstActRegIpsecSA->stAuth.stNonce);
Msf_StrReplaceX(pstRegIpsecSA->stAuth.zMemBuf, &pstRegIpsecSA->stAuth.stRealm, &pstActRegIpsecSA->stAuth.stRealm);
```

http://review.source.spreadtrum.com/gerrit/#/c/525040/1/src/protocol/ike/ike_sres.c

```
IKE_CPY_XLUSTR(pstDstSa->zMemBuf, &pstSrcSa->stAuth.stIdent, &pstDstSa->stAuth.stIdent);
```


12.13. 多处 FOR_ALL_DATA_IN_DLIST 使用错误

对于循环内部有删除节点的情况，FOR_ALL_DATA_IN_DLIST 是不安全的，存在 crash 风险，应该使用 FOR_ALL_DATA_IN_DLIST_NEXT，Enumerate all node and data in list, safe for remove node.

代码提交：
http://review.source.spreadtrum.com/gerrit/#/c/529239/
http://review.source.spreadtrum.com/gerrit/#/c/529238/

12.14. 重复删除 pstSubsd->stTransLst 问题

代码提交：
http://review.source.spreadtrum.com/gerrit/#/c/527303/

12.15. VoWiFi 进程内存优化前后对比

对于 12.1，12.2，12.3 三部分优化，测试内存优化前后进程占用内存情况如下：

三项优化后内存数据如下：

```
PS C:\Users\evers.chen> adb shell dumpsys meminfo com.spreadtrum.vowifi
Applications Memory Usage (in Kilobytes):
Uptime: 340402 Realtime: 340402

** MEMINFO in pid 5346 [com.spreadtrum.vowifi] **
                   Pss  Private  Private  SwapPss     Heap     Heap     Heap
                 Total    Dirty    Clean    Dirty     Size    Alloc     Free
                ------   ------   ------   ------   ------   ------   ------
  Native Heap     5285     5208        0        0     8192     4795     3396
  Dalvik Heap      465      424        0        0     2570     1034     1536
 Dalvik Other      331      328        0        0
        Stack       64       64        0        0
```

```
  Other dev        7        0        4        0
   .so mmap    10057     1108     5052        0
  .apk mmap      957        0      348        0
  .dex mmap     2615        4     1832        0
  .oat mmap      812        0        0        0
  .art mmap      875      472       80        0
 Other mmap      191        4        0        0
    Unknown     1884     1880        0        0
      TOTAL    23543     9492     7316        0    10762     5829     4932

App Summary
                       Pss(KB)
                        ------
               Java Heap:      976
             Native Heap:     5208
                    Code:     8344
                   Stack:       64
                Graphics:        0
           Private Other:     2216
                  System:     6735

                  TOTAL:    23543       TOTAL SWAP PSS:        0

Objects
                  Views:        0         ViewRootImpl:        0
            AppContexts:        7           Activities:        0
                 Assets:        3        AssetManagers:        2
          Local Binders:       10        Proxy Binders:       21
          Parcel memory:        4         Parcel count:       16
       Death Recipients:        1      OpenSSL Sockets:        0
              WebViews:        0

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


三项优化前内存数据如下：

PS C:\Users\evers.chen> adb shell dumpsys meminfo com.spreadtrum.vowifi
Applications Memory Usage (in Kilobytes):
Uptime: 379452 Realtime: 379452

** MEMINFO in pid 4306 [com.spreadtrum.vowifi] **

	Pss Total	Private Dirty	Private Clean	SwapPss Dirty	Heap Size	Heap Alloc	Heap Free
Native Heap	6547	6476	0	0	8192	4925	3266
Dalvik Heap	589	548	0	0	2573	1037	1536
Dalvik Other	307	304	0	0			
Stack	68	68	0	0			
Other dev	7	0	4	0			
.so mmap	9943	1108	4960	0			
.apk mmap	953	0	348	0			
.dex mmap	2598	4	1872	0			
.oat mmap	788	0	0	0			
.art mmap	877	480	72	0			
Other mmap	142	4	0	0			
Unknown	1880	1876	0	0			
TOTAL	24699	10868	7256	0	10765	5962	4802

App Summary

	Pss(KB)
Java Heap:	1100
Native Heap:	6476
Code:	8292
Stack:	68
Graphics:	0
Private Other:	2188
System:	6575
TOTAL:	24699

TOTAL SWAP PSS: 0

Objects

Views:	0	ViewRootImpl:	0
AppContexts:	9	Activities:	0
Assets:	3	AssetManagers:	3
Local Binders:	10	Proxy Binders:	21
Parcel memory:	4	Parcel count:	16
Death Recipients:	1	OpenSSL Sockets:	0
WebViews:	0		

SQL

MEMORY_USED:	0		
PAGECACHE_OVERFLOW:	0	MALLOC_SIZE:	0

经过 12.1，12.2，12.3 三项优化，Native Heap 内存减少 1.268M，百分比如下：
Native Heap 内存降低 6476-5208/6476=1268/6476=19.6%
Total 内存降低 24699-23543/24699=1156/24699=4.7%

