# 2. 标记结构

本章节将介绍基本的内存标记结构,包括chunk, tree chunk, sbin, tbin, segment, mstate等.这些重要的机构组成了dlmalloc分配算法的基础.

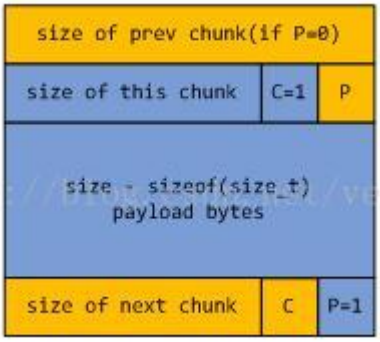
## 2.1 chunk

chunk是dlmalloc中最基本的一种结构,它代表了一块经过划分后被管理的内存单元. dlmalloc所有对内存的操作几乎都聚焦在chunk上.需要注意的是, chunk虽然看似基础,但不代表它能管理的内存小.事实上chunk被划分为两种类型,小于256字节的称为small chunk,而大于等于256字节的被称为tree chunk.

# 2.1.1 chunk布局

在dlmalloc的源码中有一幅用字符画成的示意图,套用Doug Lea本人的话说 “is misleading but accurate and necessary”.初次接触这种结构划分会觉得既古怪又别扭. 因此,接下来的几节内容需要认真体会,让大脑适应这种错位的思考.

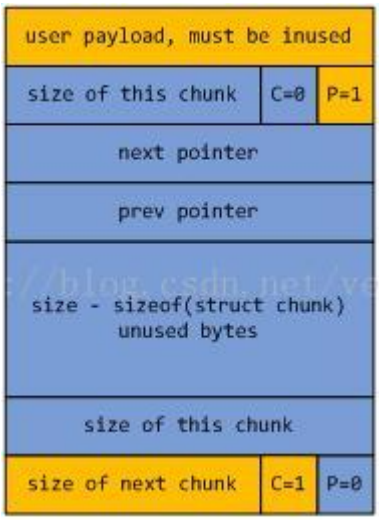
对于一个已分配chunk, 它在内存中可能是这个样子,




上图为了便于理解特意使用了两种颜色来描述,蓝色区域的部分代表当前的已分配chunk,而黄色区域代表上一个和下一个chunk的部分区域.

解释一下其中的含义, 从蓝色区域开始看, 在已分配chunk的开始记录当前chunk的size信息,同时在最后两个bit位分别记录当前chunk和前一个chunk是否被使用,简称为C和P两个bit位.之所以可以做到这一点,是因为在dlmalloc中所有的chunk size至少对齐到大于8并以2为底的指数边界上.这样,至少最后3位都是0,因此这些多余的bit位就可以拿来记录更多的信息. size信息后面是payload部分,显然,当前chunk的使用信息也会记录在下一个chunk开始的P位上.

而对于一个空闲chunk, 其内存布局应该是这样的,




与之前的一致, 蓝色区域代表当前空闲chunk, 黄色区域代表相邻的前后chunk.可以看到空闲chunk与之前的区别在于开始的size后多了next和prev指针,它们把具有相同大小的空闲chunk链接到一起.另一个区别是,在空闲chunk最后同样记录该chunk的大小信息.那么为什么同样的信息要记录两次呢?在下一个小节中会解释这个问题.

另外, 还有一点需要说明的是,空闲chunk的相邻chunk必然是已分配的chunk.因为如果存在相邻两个chunk都是空闲的,那么dlmalloc会把它们合并为一个更大的chunk.

## 2.1.2 边界标记法(Boundary Tag)

上一节中介绍的chunk布局其实只是理论上的东西,而实现它使用了被称为边界标记法 (boundary tag)的技术.据作者说,此方法最早是大神Knuth提出的.实现边界标记法使用了名为malloc_chunk的结构体,它的定义如下,

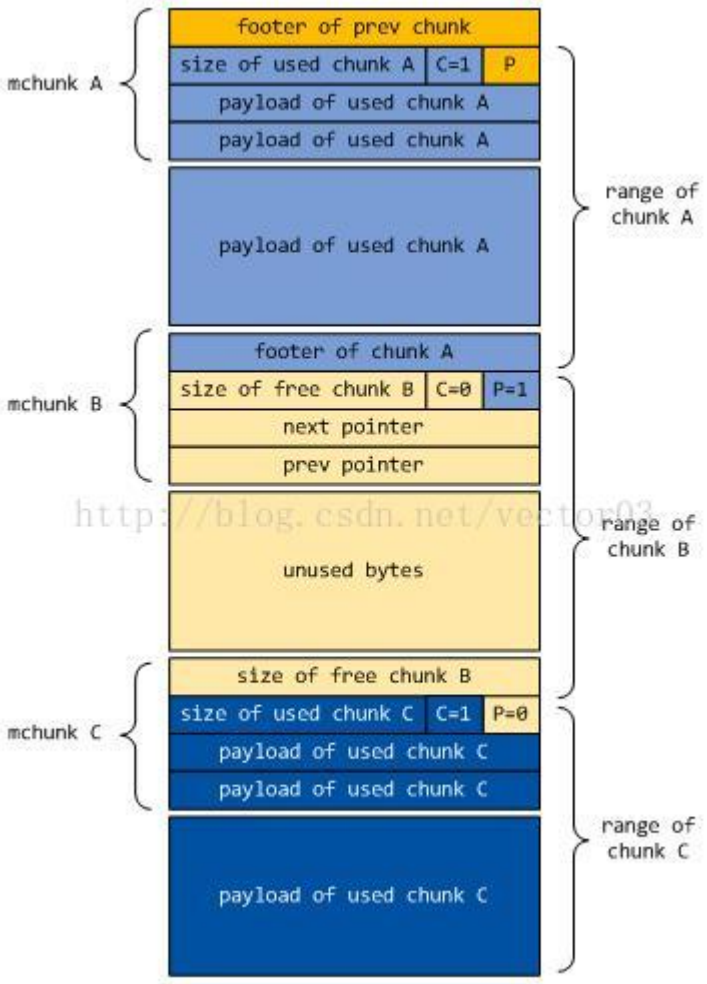
```
struct malloc_chunk {
  size_t               prev_foot;  /* Size of previous chunk (if free).  */
  size_t               head;       /* Size and inuse bits. */
  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;
};

typedef struct malloc_chunk  mchunk;
typedef struct malloc_chunk* mchunkptr;
```



该结构体(以后简称mchunk)由四个field组成.最开始是prev_foot,记录了上一个邻接chunk的最后4个字节.接下来是head,记录当前chunk的size以及C和P位.最后两个是fd, bk指针,只对空闲chunk起作用,用于链接相同大小的空闲chunk.

为了避免读者感到困惑, 在上一节的图中并没有画出对应的mchunk, 现在补充完整如下,




上图用不同颜色画了几个连续交错的chunk,并故意在中间断开,标注出mchunk以及chunk的实际范围,由此得到如下的结论,

1. mchunk的作用是将连续内存划分为一段段小的区块, 并在内部保留这些区块的信息.

2. mchunk最后的fd, bk是可以复用的,对于空闲chunk它们代表链接指针,而已使用chunk中这两个field实际上存放的是payload数据.

3. mchunk与chunk不是一个等同的概念.这一点是容易让人混淆和困惑的. mchunk只是一个边界信息,它实际上横跨了两个相邻chunk.尽管一般认为mchunk可以指代当前的chunk,因为你可以从它推算出想要的地址.但从逻辑上,它既不代表当前chunk也不代表prev chunk.

4. prev_foot字段同样是一个可复用的字段. 一般情况下它有三种含义,如果前一个相邻chunk是空闲状态,它记录该chunk的大小(图中mchunk C).其目的就是你可以很方便的从当前chunk获得上一个相邻chunk的首地址,从而快速对它们进行合并.这也就是上一小节介绍的空闲chunk会在head和footer存在两个chunk size的原因,位于footer的size实际上保存在下一个mchunk中.如果前一个相邻chunk处于in used状态,那么该字段可能有两种情况.一种情况是FOOTERS宏为1,这时它保存一个交叉检查值,用于在free()的时候进行校验.如果该宏为0,则它没有多余的含义,单纯只是前面chunk的payload数据.

5. 最后还有一点需要注意的是, 尽管前面提到所有chunk size都是对齐到至少等于8的2的指数.但这不意味着mchunk就对齐到这一边界上,因为如前所述mchunk和chunk是两码事.我本人最早在看这部分代码时曾天真的以为mchunk也是对齐的,结果到后面居然完全看不懂,后来才发现这家伙根本没有对齐.

到目前为止, 已经了解了dlmalloc的边界标记法是如何实现的.可能有人会对此感到不以为然,作者为什么要把代码写成这个样子呢?其实,要理解这个意图请换位思考一下,如果实现一个类似的东西,你会怎么写呢?人们很快会发现,不按照这种方式设计,想要达到同样目的几乎是很困难的.因为一个chunk在头和尾存在head和footer(也可能不存在footer),中间是一片长度无法确定的区域.假设按照正常从头到尾的顺序写,这个结构体中间填充多长的内容是没法确定的.因此, Doug Lea巧妙的把除了payload之外的overhead部分合并在一起,作为boundary tag来分割整个内存区域,既能做到很方便的计算,又准确地表达了chunk的结构.

# 2.1.3 chunk操作

针对mchunk的设计,源码中定义了大量的宏来对其进行操作.由于数量很多加上嵌套,对于阅读源码会造成不小的障碍,因此这里会对一些常用宏进行梳理和简单讲解.

```
#define SIZE_T_SIZE         (sizeof(size_t))
#define SIZE_T_BITSIZE      (sizeof(size_t) << 3)
```

定义size_t的大小和bit位数.注意, dlmalloc中对size_t规定为必须为无符号类型,且与指针类型(void*)等宽度.因此如果目标平台还在使用有符号的size_t,只能使用较旧版本的dlmalloc.

```
#define SIZE_T_ZERO         ((size_t)0)
#define SIZE_T_ONE          ((size_t)1)
#define SIZE_T_TWO          ((size_t)2)
#define SIZE_T_FOUR         ((size_t)4)
#define TWO_SIZE_T_SIZES    (SIZE_T_SIZE<<1)
#define FOUR_SIZE_T_SIZES   (SIZE_T_SIZE<<2)
#define SIX_SIZE_T_SIZES    (FOUR_SIZE_T_SIZES+TWO_SIZE_T_SIZES)
#define HALF_MAX_SIZE_T     (MAX_SIZE_T / 2U)
```

常量定义, 这里使用size_t作为强制类型是为了避免在某些平台上出现错误.

```
#define MALLOC_ALIGNMENT ((size_t)(2 * sizeof(void *)))

#define CHUNK_ALIGN_MASK    (MALLOC_ALIGNMENT - SIZE_T_ONE)
```

对齐边界和对齐掩码, 边界默认定义为两倍指针宽度, 当然还可以修改的更大,但必须以2为底的指数.有关对齐掩码相关知识请参考1.4.1节.

```
#define is_aligned(A)       (((size_t)((A)) & (CHUNK_ALIGN_MASK)) == 0)
```

判断给定地址是否对齐在边界上,原理同样在1.4.1节中解释,不再赘述.

```
#define align_offset(A)\
 ((((size_t)(A) & CHUNK_ALIGN_MASK) == 0)? 0 :\
  ((MALLOC_ALIGNMENT - ((size_t)(A) & CHUNK_ALIGN_MASK)) & CHUNK_ALIGN_MASK))
```

计算给定地址需要对齐的偏移量,请参考1.4.3节中的介绍.

```
#define MCHUNK_SIZE         (sizeof(mchunk))

#if FOOTERS
#define CHUNK_OVERHEAD      (TWO_SIZE_T_SIZES)
#else /* FOOTERS */
#define CHUNK_OVERHEAD      (SIZE_T_SIZE)
#endif /* FOOTERS */
```

chunk size(实际上是mchunk size,后面不再特意说明)以及overhead大小.注意,如果FOOTERS等于1, overhead将多出4个字节,其实就是在chunk最后放置交叉检查项多出的负载.另外,不管是否有FOOTERS, chunk size本身是不变的.

```
#define MIN_CHUNK_SIZE\
  ((MCHUNK_SIZE + CHUNK_ALIGN_MASK) & ~CHUNK_ALIGN_MASK)
```

最小\chunk大小. dlmalloc中即使调用malloc(0)也是会分配内存的.这种情况下完全不考虑用户payload了,仅仅将chunk size做了对齐.该值的意义在于,这是dlmalloc中除了mmap chunk以外的最坏负载损耗.

```
#define chunk2mem(p)        ((void*)((char*)(p)       + TWO_SIZE_T_SIZES))
#define mem2chunk(mem)      ((mchunkptr)((char*)(mem) - TWO_SIZE_T_SIZES))
```

这是两个最常用的宏之一, 在chunk和用户地址之前切换.

```
#define align_as_chunk(A)   (mchunkptr)((A) + align_offset(chunk2mem(A)))
```

该宏的意思是, 将指定chunk的用户地址对齐,返回对齐后的新chunk地址.理解很容易,从chunk获取用户地址,计算对齐偏移,再移动chunk指针.从这里可以看到, chunk中对齐的并非是mchunk指针,而是用户地址.

```
#define pad_request(req) \
   (((req) + CHUNK_OVERHEAD + CHUNK_ALIGN_MASK) & ~CHUNK_ALIGN_MASK)
```

将用户请求的内存值转化成实际的内部大小.该宏也是常用宏之一,请求大小先加上overhead(根据FOOTERS而有所不同),再对齐到边界上.

```
#define request2size(req) \
  (((req) < MIN_REQUEST)? MIN_CHUNK_SIZE : pad_request(req))
```

该宏是对上面一系列宏的封装, 如果用户请求小于最小请求, 直接使用最小\chunk大小,否则认为正常,转化成内部可用大小.

```
#define PINUSE_BIT          (SIZE_T_ONE)
#define CINUSE_BIT          (SIZE_T_TWO)
#define FLAG4_BIT           (SIZE_T_FOUR)
#define INUSE_BITS          (PINUSE_BIT|CINUSE_BIT)
#define FLAG_BITS           (PINUSE_BIT|CINUSE_BIT|FLAG4_BIT)
```

上面这一组定义了C位和P位,其中FLAG4_BIT并没有实际用途,只是作为未来的扩展.

```
#define cinuse(p)           ((p)->head & CINUSE_BIT)
#define pinuse(p)           ((p)->head & PINUSE_BIT)
#define flag4inuse(p)       ((p)->head & FLAG4_BIT)
#define is_inuse(p)         (((p)->head & INUSE_BITS) != PINUSE_BIT)
#define is_mmapped(p)       (((p)->head & INUSE_BITS) == 0)
```

测试C和P位,分别用掩码做与运算.着重说一下最后两个判断. is_inuse是用C和P的掩码分别测试,然后与P的掩码对比.有人会问这个直接用前面的cinuse不就好了吗?因为存在一种特殊情况,就是被mmap出来的chunk是不见得有邻接chunk的,所以其C和P都被置0. is_inuse就是为了判断包括mmap chunk在内的所有类型chunk的使用情况.理解了这个, is_mmapped就很容易理解了.

```
#define chunksize(p)        ((p)->head & ~(FLAG_BITS))
```

计算指定chunk的大小,清除最后3bit后得到的就是size.

```
#define clear_pinuse(p)     ((p)->head &= ~PINUSE_BIT)
#define set_flag4(p)        ((p)->head |= FLAG4_BIT)
#define clear_flag4(p)      ((p)->head &= ~FLAG4_BIT)
```

对P位的操作.

```
#define chunk_plus_offset(p, s)  ((mchunkptr)(((char*)(p)) + (s)))
#define chunk_minus_offset(p, s) ((mchunkptr)(((char*)(p)) - (s)))
```

这是两个计算偏移量的宏, 返回结果被认为是mchunk, 一般用作chunk切割.

```
#define next_chunk(p) ((mchunkptr)( ((char*)(p)) + ((p)->head & ~FLAG_BITS)))
#define prev_chunk(p) ((mchunkptr)( ((char*)(p)) - ((p)->prev_foot) ))
```

这两个也是非常重要的宏. 分别用来计算前一个或后一个邻接chunk的首地址. next_chunk先获取当前chunk的size作为偏移量,再移动chunk指针. prev_chunk则是直接加上前一个chunk的footer作为偏移量.需要注意的是, prev_chunk仅仅适用于前一个邻接chunk为空闲块的情况.如前所述,非空闲chunk这里放置的可能是用户数据或者交叉检查项.实际上这个宏就是用于free时试图与前一个free chunk合并时的判断.

```
#define next_pinuse(p)  ((next_chunk(p)->head) & PINUSE_BIT)
```

获取下一个chunk的P位,应该与当前chunk的C是一致的.该宏一般用作交叉检查项.

```
#define get_foot(p, s)  (((mchunkptr)((char*)(p) + (s)))->prev_foot)
#define set_foot(p, s)  (((mchunkptr)((char*)(p) + (s)))->prev_foot = (s))
```

获取和设置当前chunk的footer,因为footer被放在下一个邻接chunk的prev_foot中,因此需要加上size作为偏移量.

```
#define set_size_and_pinuse_of_free_chunk(p, s)\
  ((p)->head = (s|PINUSE_BIT), set_foot(p, s))
```

该宏是个复合操作, 用于对free chunk进行设置.因为free chunk在head和footer都保存其size,所以首先将其head中写入size.又因为free chunk的P一定是1(否则会与前面合并),因此还需要与P掩码进行一次位与.接着,利用前面的set_foot宏在footer

中同样写入size.

```
#define set_free_with_pinuse(p, s, n)\
  (clear_pinuse(n), set_size_and_pinuse_of_free_chunk(p, s))
```

这个宏与上面的区别在于多了一个next chunk的P位维护.因为是针对free chunk, next chunk的P需要置0.

# 2.1.4 tree chunk

之前介绍的chunk其实都属于小型chunk,而较大型的chunk是通过名为malloc_tree_chunk的结构体来描述的.

```
struct malloc_tree_chunk {
  /* The first four fields must be compatible with malloc_chunk */
  size_t                    prev_foot;
  size_t                    head;
  struct malloc_tree_chunk* fd;
  struct malloc_tree_chunk* bk;

  struct malloc_tree_chunk* child[2];
  struct malloc_tree_chunk* parent;
  bindex_t                  index;
};

typedef struct malloc_tree_chunk  tchunk;
typedef struct malloc_tree_chunk* tchunkptr;
typedef struct malloc_tree_chunk* tbinptr; /* The type of bins of trees */
```

同mchunk类似tree chunk(以后简称tchunk)开始的field是一模一样的,这就保证了两种类型在基础结构上具有较好的兼容性,降低了代码编写难度. tchunk与前者的区别在于管理上使用不同的结构. mchunk是通过双链表组织起来的,而tchunk使用了一种树形结构,在介绍分箱时会详细说明.因此tchunk中child分别代表了左右子树节点, parent代表父节点, index代表该chunk所在分箱的编号.而fd, bk与mchunk一致,都是用于链接相同size的chunk.另外,需要说明的是,上述结构体字段同样只适用于free chunk,在used chunk上它们都是可复用的.