[堆](#header-n749)  
 [堆利用](#header-n750)  
 [堆概述](#header-n759)  
 [内存分配后的系统调用](#header-n760)  
 [(s)brk](#header-n765)  
 [mmap](#header-n781)  
 [堆相关的数据结构](#header-n1085)  
 [微观结构](#header-n792)  
 [malloc\_chunk概述](#header-n794)  
 [bin](#header-n871)  
 [概述](#header-n872)  
 [Fast Bin](#header-n907)  
 [Small bin](#header-n922)  
 [Large bin](#header-n962)  
 [unsorted bin](#header-n1000)  
 [top chunk](#header-n1037)  
 [宏观结构](#header-n1057)  
 [arena](#header-n1061)

# 堆

## 堆利用

目前堆的实现有很多种，具体如下：

dlmalloc – General purpose allocator

ptmalloc2 – glibc

jemalloc – FreeBSD and Firefox

tcmalloc – Google

libumem – Solaris

接下来主要以glibc中堆的实现为主进行介绍。

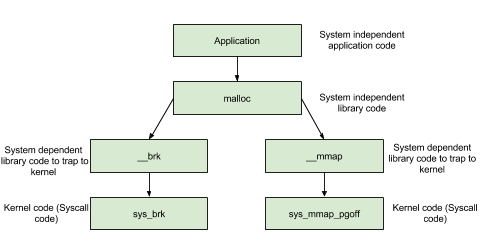
## 堆概述

### 内存分配后的系统调用

在我们动态申请和释放内存时，无论是malloc还是free函数，都不是真正与系统交互的函数

这些函数的背后系统调用主要是(s)brk和mmap、munmap函数

如下图：



### (s)brk

对于堆来说，操作系统提供了brk函数，glibc库提供了sbrk函数，我们也可通过增加brk的大小来向操作系统申请内存。

在一开始，堆的起始位置(start\_brk)和堆的当前结尾地址(brk)指向同一地址。根据是否开启ASLR，指的位置不同

* 未开启ASLR：start\_brk和brk指向data/bss段的结尾
* 开启ASLR：start\_brk和brk会指向data/bss段结尾后的随机偏移的位置

举个小例子：(来自ctf-wiki)

/\* sbrk and brk example \*/
  
#include <stdio.h>
  
#include <unistd.h>
  
#include <sys/types.h>
  
   
int main()
  
{
  
 void \*curr\_brk, \*tmp\_brk = NULL;
  
   
 printf("Welcome to sbrk example:%d\n", getpid());
  
   
 /\* sbrk(0) gives current program break location \*/
  
 tmp\_brk = curr\_brk = sbrk(0);
  
 printf("Program Break Location1:%p\n", curr\_brk);
  
 getchar();
  
   
 /\* brk(addr) increments/decrements program break location \*/
  
 brk(curr\_brk+4096);
  
   
 curr\_brk = sbrk(0);
  
 printf("Program break Location2:%p\n", curr\_brk);
  
 getchar();
  
   
 brk(tmp\_brk);
  
   
 curr\_brk = sbrk(0);
  
 printf("Program Break Location3:%p\n", curr\_brk);
  
 getchar();
  
   
 return 0;
  
}

执行后3次分别如下：

yutao@pwnbaby:~/Desktop$ ./a.out
  
Welcome to sbrk example:2552
  
Program Break Location1:0x56c4d000
  
   
Program break Location2:0x56c4e000
  
   
Program Break Location3:0x56c4d000

yutao@pwnbaby:~/Desktop$ cat /proc/2552/maps
  
...
  
56c2b000-56c4d000 rw-p 00000000 00:00 0 [heap]
  
...
  
   
yutao@pwnbaby:~/Desktop$ cat /proc/2552/maps
  
...
  
56c2b000-56c4e000 rw-p 00000000 00:00 0 [heap]
  
...
  
   
yutao@pwnbaby:~/Desktop$ cat /proc/2552/maps
  
...
  
56c2b000-56c4d000 rw-p 00000000 00:00 0 [heap]
  
...

### mmap

malloc会使用mmap来创建独立的匿名映射段。匿名映射的主要目的是可以申请以0填充的内存，并且这块内存仅被调用进程所使用。

例子：

/\* Private anonymous mapping example using mmap syscall \*/
  
#include <stdio.h>
  
#include <sys/mman.h>
  
#include <sys/types.h>
  
#include <sys/stat.h>
  
#include <fcntl.h>
  
#include <unistd.h>
  
#include <stdlib.h>
  
   
void static inline errExit(const char\* msg)
  
{
  
 printf("%s failed. Exiting the process\n", msg);
  
 exit(-1);
  
}
  
   
int main()
  
{
  
 int ret = -1;
  
 printf("Welcome to private anonymous mapping example::PID:%d\n", getpid());
  
 printf("Before mmap\n");
  
 getchar();
  
 char\* addr = NULL;
  
 addr = mmap(NULL, (size\_t)132\*1024, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE | MAP\_ANONYMOUS, -1, 0);
  
 if (addr == MAP\_FAILED)
  
 errExit("mmap");
  
 printf("After mmap\n");
  
 getchar();
  
   
 /\* Unmap mapped region. \*/
  
 ret = munmap(addr, (size\_t)132\*1024);
  
 if(ret == -1)
  
 errExit("munmap");
  
 printf("After munmap\n");
  
 getchar();
  
 return 0;
  
}

在执行mmap前：

yutao@pwnbaby:~$ cat /proc/2412/maps
  
5656c000-5656d000 r-xp 00000000 08:01 956125 /home/yutao/Desktop/a.out
  
5656d000-5656e000 r--p 00000000 08:01 956125 /home/yutao/Desktop/a.out
  
5656e000-5656f000 rw-p 00001000 08:01 956125 /home/yutao/Desktop/a.out
  
56fdf000-57001000 rw-p 00000000 00:00 0 [heap]
  
f7d90000-f7f65000 r-xp 00000000 08:01 1195832 /lib/i386-linux-gnu/libc-2.27.so
  
f7f65000-f7f66000 ---p 001d5000 08:01 1195832 /lib/i386-linux-gnu/libc-2.27.so
  
f7f66000-f7f68000 r--p 001d5000 08:01 1195832 /lib/i386-linux-gnu/libc-2.27.so
  
f7f68000-f7f69000 rw-p 001d7000 08:01 1195832 /lib/i386-linux-gnu/libc-2.27.so
  
f7f69000-f7f6c000 rw-p 00000000 00:00 0
  
f7f82000-f7f84000 rw-p 00000000 00:00 0
  
f7f84000-f7f87000 r--p 00000000 00:00 0 [vvar]
  
f7f87000-f7f88000 r-xp 00000000 00:00 0 [vdso]
  
f7f88000-f7fae000 r-xp 00000000 08:01 1195828 /lib/i386-linux-gnu/ld-2.27.so
  
f7fae000-f7faf000 r--p 00025000 08:01 1195828 /lib/i386-linux-gnu/ld-2.27.so
  
f7faf000-f7fb0000 rw-p 00026000 08:01 1195828 /lib/i386-linux-gnu/ld-2.27.so
  
ffab0000-ffad1000 rw-p 00000000 00:00 0 [stack]

执行mmap后：

yutao@pwnbaby:~$ cat /proc/2412/maps
  
5656c000-5656d000 r-xp 00000000 08:01 956125 /home/yutao/Desktop/a.out
  
5656d000-5656e000 r--p 00000000 08:01 956125 /home/yutao/Desktop/a.out
  
5656e000-5656f000 rw-p 00001000 08:01 956125 /home/yutao/Desktop/a.out
  
56fdf000-57001000 rw-p 00000000 00:00 0 [heap]
  
f7d6f000-f7d90000 rw-p 00000000 00:00 0
  
f7d90000-f7f65000 r-xp 00000000 08:01 1195832 /lib/i386-linux-gnu/libc-2.27.so
  
f7f65000-f7f66000 ---p 001d5000 08:01 1195832 /lib/i386-linux-gnu/libc-2.27.so
  
f7f66000-f7f68000 r--p 001d5000 08:01 1195832 /lib/i386-linux-gnu/libc-2.27.so
  
f7f68000-f7f69000 rw-p 001d7000 08:01 1195832 /lib/i386-linux-gnu/libc-2.27.so
  
f7f69000-f7f6c000 rw-p 00000000 00:00 0
  
f7f82000-f7f84000 rw-p 00000000 00:00 0
  
f7f84000-f7f87000 r--p 00000000 00:00 0 [vvar]
  
f7f87000-f7f88000 r-xp 00000000 00:00 0 [vdso]
  
f7f88000-f7fae000 r-xp 00000000 08:01 1195828 /lib/i386-linux-gnu/ld-2.27.so
  
f7fae000-f7faf000 r--p 00025000 08:01 1195828 /lib/i386-linux-gnu/ld-2.27.so
  
f7faf000-f7fb0000 rw-p 00026000 08:01 1195828 /lib/i386-linux-gnu/ld-2.27.so
  
ffab0000-ffad1000 rw-p 00000000 00:00 0 [stack]

munmap：

yutao@pwnbaby:~$ cat /proc/2412/maps
  
5656c000-5656d000 r-xp 00000000 08:01 956125 /home/yutao/Desktop/a.out
  
5656d000-5656e000 r--p 00000000 08:01 956125 /home/yutao/Desktop/a.out
  
5656e000-5656f000 rw-p 00001000 08:01 956125 /home/yutao/Desktop/a.out
  
56fdf000-57001000 rw-p 00000000 00:00 0 [heap]
  
f7d90000-f7f65000 r-xp 00000000 08:01 1195832 /lib/i386-linux-gnu/libc-2.27.so
  
f7f65000-f7f66000 ---p 001d5000 08:01 1195832 /lib/i386-linux-gnu/libc-2.27.so
  
f7f66000-f7f68000 r--p 001d5000 08:01 1195832 /lib/i386-linux-gnu/libc-2.27.so
  
f7f68000-f7f69000 rw-p 001d7000 08:01 1195832 /lib/i386-linux-gnu/libc-2.27.so
  
f7f69000-f7f6c000 rw-p 00000000 00:00 0
  
f7f82000-f7f84000 rw-p 00000000 00:00 0
  
f7f84000-f7f87000 r--p 00000000 00:00 0 [vvar]
  
f7f87000-f7f88000 r-xp 00000000 00:00 0 [vdso]
  
f7f88000-f7fae000 r-xp 00000000 08:01 1195828 /lib/i386-linux-gnu/ld-2.27.so
  
f7fae000-f7faf000 r--p 00025000 08:01 1195828 /lib/i386-linux-gnu/ld-2.27.so
  
f7faf000-f7fb0000 rw-p 00026000 08:01 1195828 /lib/i386-linux-gnu/ld-2.27.so
  
ffab0000-ffad1000 rw-p 00000000 00:00 0 [stack]

## 堆相关的数据结构

glibc内部有精心设计的数据结构来管理它：

* 宏观结构：包含堆的宏观信息，可通过这些数据结构索引对的基本信息
* 微观结构：用于处理堆分配与回收中的内存块

### 微观结构

这里先讲微观结构，堆的漏洞利用和这些相关

#### malloc\_chunk概述

我们成malloc申请的内存为chunk。这些内存在ptmalloc内部用malloc\_chunk结构体来表示，

当程序申请的chunk被free后，会被加入到响应的空闲管理列表中。

无论一个chunk有多大，无论处于malloc状态还是free状态，他们的结构都是一样的。

malloc\_chunk结构：

/\*  
 This struct declaration is misleading (but accurate and necessary).  
 It declares a "view" into memory allowing access to necessary  
 fields at known offsets from a given base. See explanation below.  
\*/  
struct malloc\_chunk {  
  
 INTERNAL\_SIZE\_T prev\_size; /\* Size of previous chunk (if free). \*/  
 INTERNAL\_SIZE\_T size; /\* Size in bytes, including overhead. \*/  
  
 struct malloc\_chunk\* fd; /\* double links -- used only if free. \*/  
 struct malloc\_chunk\* bk;  
  
 /\* Only used for large blocks: pointer to next larger size. \*/  
 struct malloc\_chunk\* fd\_nextsize; /\* double links -- used only if free. \*/  
 struct malloc\_chunk\* bk\_nextsize;  
};

这里有INTERNAL*SIZE*T，SIZE*SZ，MALLOC*ALIGN\_MASK的解释：

/ \* INTERNAL\_SIZE\_T是用于内部记录块大小的。  
 默认版本与size\_t相同。  
 尽管不是绝对必要，但最好将其定义为无符号类型，即使size\_t是有符号类型也是如此。这可以避免某些系统的人为大小限制。  
 在64位计算机上，通过将INTERNAL\_SIZE\_T定义为32位“ unsigned int”，您可以减少malloc开销。  
 不能处理超过2 ^ 32的已分配空间的开销。如果可以接受此限制，除非您在要求16字节对齐的平台上，否则建议您进行设置。在这种情况下，对齐要求最终抵消了减小size\_t字大小的任何潜在优势。  
 实现者：谨防以下情况的可能组合：  
 -INTERNAL\_SIZE\_T可能是有符号的或无符号的，可能是32位或64位，并且宽度可能与int相同或与long相同  
 -size\_t的宽度和签名可能与INTERNAL\_SIZE\_T不同  
 -int和long可能是32或64位，并且可能是相同的宽度  
 为了解决这个问题，INTERNAL\_SIZE\_T之间的大多数比较和差值计算都应将它们强制转换为无符号长整数，并意识到将无符号int强制转换为较宽长整数不会对符号进行扩展的事实。 （这也使检查负数变得很尴尬。）其中的某些强制转换在某些系统上会导致无害的编译器警告。 \* /  
#ifndef INTERNAL\_SIZE\_T  
#define INTERNAL\_SIZE\_T size\_t  
#endif  
  
/\* The corresponding word size. \*/  
#define SIZE\_SZ (sizeof (INTERNAL\_SIZE\_T))  
  
/\* The corresponding bit mask value. \*/  
#define MALLOC\_ALIGN\_MASK (MALLOC\_ALIGNMENT - 1)

一般来说，size\_t在64位中是64位无符号整数，32位中是32位无符号整数。

malloc\_chunk的具体解释如下：

* prev\_size：如果该chunk的 物理相邻的前一个chunk是空闲的话，那该处记录的是前一个chunk的大小(**包含chunk头**)。如果物理相邻的前一个chunk处于malloc状态，那么此处可以填写前一个chunk的数据。
* size：该chunk的大小(prev*size+size+user data)，必须是2\*SIZE*SZ的整数倍。如果申请的内存不是2\*SIZE*SZ的整数倍，那么向上取整。32位中SIZE*SZ是4；64位中是8。该字段的 低3bit位对chunk大小没有影响，依次为：
  + NON*MAIN*ARENA(A)：记录当前chunk是否属于主线程，1表示不属于，0表示属于。
  + IS\_MAPPED(M)：记录当前chunk是否是由mmap分配的。
  + PREV*INUSE(P)：记录前一个chunk是否被分配。一般来说，堆中第一个被分配的内存块的该位都会被设为1，以便防止访问前面的非法内存；当该位是0时，可以通过prev*size字段来获取上一个chunk的大小和地址。也方便空闲chunk之间的合并。
* fd，dk：chunk处于分配状态时，从 fd 字段开始是用户的数据。chunk 空闲时，会被添加到对应的空闲管理链表中，其字段的含义如下：
  + fd指向下一个(**非物理相邻**)空闲的chunk。
  + bk指向上一个(**非物理相邻**)空闲的chunk。
  + 通过bk和fd可以将空闲的chunk加入到空闲的chunk链表进行统一管理
* fd*nextsize，bk*nextsize：只有chunk空闲时候才使用，**用于较大的chunk(large chunk)**。
  + fd\_nextsize：指向前一个与当chunk大小不同的第一个空闲快，不含bin的头指针。
  + k\_nextsize 指向后一个与当前 chunk 大小不同的第一个空闲块，不包含 bin 的头指针。
  + 一般空闲的 large chunk 在 fd 的遍历顺序中，按照由大到小的顺序排列。这样做可以避免在寻找合适 chunk 时挨个遍历。

一个已经分配的 chunk 的样子如下。**我们称前两个字段称为 chunk header，后面的部分称为 user data。每次 malloc 申请得到的内存指针，其实指向 user data 的起始处。**

chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
 | Size of previous chunk, if unallocated (P clear) |  
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
 | Size of chunk, in bytes |A|M|P|  
 mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
 | User data starts here... |  
 . .  
 . (malloc\_usable\_size() bytes) .  
next . .  
chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
 | (size of chunk, but used for application data) |  
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
 | Size of next chunk, in bytes |A|0|1|  
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

被free的 chunk 被记录在链表中（可能是循环双向链表，也可能是单向链表）。具体结构如下

chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  
 | Size of previous chunk, if unallocated (P clear) |
  
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  
`head:' | Size of chunk, in bytes |A|0|P|
  
 mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  
 | Forward pointer to next chunk in list |
  
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  
 | Back pointer to previous chunk in list |
  
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  
 | Unused space (may be 0 bytes long) .
  
 . .
  
 next . |
  
chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  
`foot:' | Size of chunk, in bytes |
  
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  
 | Size of next chunk, in bytes |A|0|0|
  
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

可以发现，如果一个 chunk 处于 free 状态，那么会有两个位置记录其相应的大小

1. 本身的 size 字段会记录，
2. 它后面的 chunk 会记录。

一般情况下，物理相邻的两个空闲 chunk 会被合并为一个 chunk 。堆管理器会通过 prev\_size 字段以及 size 字段合并两个物理相邻的空闲 chunk 块。

#### bin

##### 概述

bin这个概念是与**内存回收**相关堆，也就是堆管理器会根据用户已经申请到堆内存空间大小进行释放，决定放入哪类bins当中去。

用户释放的chunk不会马上归还给系统，ptmalloc会统一管理heap和mmap映射区域中空闲的chunk。当用户再次malloc时，ptmalloc分配器会识图在空闲的chunk中挑选一块合适的给用户。这样可以避免频繁的系统调用，降低内存分配的开销。

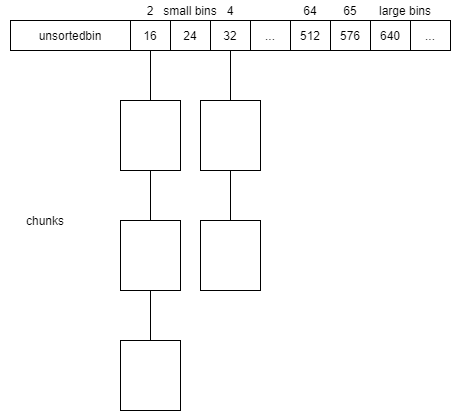
**ptmalloc将相似大小堆chunk用双向线标链接起来，这样的一个链表叫做一个bin**。

ptmalloc一共维护了128个bin，并使用一个数组来存储这些bin。

数组中**bin 1为unsorted bin**；**bin 2到63为small bin**；**bin 64到126为large bin**

具体的实现中，ptmalloc采用分箱式的方法对空闲的chunk进行管理。

首先，他会根据空闲chunk的大小及使用状态将chunk初步分为4类：fast bins，small bins，large bins，unsorted bin。每一类中仍有更详细的划分。相似大小的chunk会使用双向链表链接起来。也就是说，在每一类bin的内部仍有多个互不相关的链表来保存不同大小的chunk。



对于 small bins，large bins，unsorted bin 来说，ptmalloc 将它们维护在同一个数组中。这些 bin 对应的数据结构在 malloc\_state 中，如下：

#define NBINS 128  
/\* Normal bins packed as described above \*/  
mchunkptr bins[ NBINS \* 2 - 2 ];

一个bin相当于一个chunk链表，每个链表的头结点chunk作为bins数组，但由于这个头结点作为bin表头，prev*size与size字段是没有任何实际作用的，因此在存储头结点chunk的时候仅仅只需存储头结点chunk的fd和bk即可，而其中的prev*size与size字段被重用为另一个bin的头结点的fd和bk，这样可节省空间，并提高可用性。。因此**我们仅仅只需要 mchunkptr 类型的指针数组就足够存储这些头节点**，那 prev\_size 与 size 字段到底是怎么重用的呢？这里我们以 32 位系统为例：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 含义 | bin1 的 fd/bin2 的 prev\_size | bin1 的 bk/bin2 的 size | bin2 的 fd/bin3 的 prev\_size | bin2 的 bk/bin3 的 size |
| bin 下标 | 0 | 1 | 2 | 3 |

可以看出除了第一个 bin（unsorted bin）外，后面的每个 bin 的表头 chunk 会重用前面的 bin 表头 chunk 的 fd 与 bk 字段，将其视为其自身的 prev*size 和 size 字段。这里也说明了一个问题，\*\*bin 的下标和我们所说的第几个 bin 并不是一致的。同时，bin 表头的 chunk 头节点 的 prev*size 与 size 字段不能随便修改，因为这两个字段是其它 bin 表头 chunk 的 fd 和 bk 字段。\*\*

数组中的 bin 依次介绍如下：

1. 第1个为 unsorted bin，这里面的 chunk 没有进行排序，存储的 chunk 比较杂。
2. 从 2 到 63 的 bin 称为 small bin，同一个 small bin 链表中的 chunk 的大小相同。两个相邻索引的 small bin 链表中的 chunk 大小相差的字节数为 **2 个机器字长**，即 32 位相差 8 字节，64 位相差 16 字节。
3. small bins 后面的 bin 被称作 large bins。large bins 中的每一个 bin 都包含一定范围内的 chunk，其中的 chunk 按 fd 指针的顺序从大到小排列。相同大小的 chunk 同样按照最近使用顺序排列。

此外，上述这些 bin 的排布都会遵循一个原则：**任意两个物理相邻的空闲 chunk 不能在一起**。

需要注意的是，并不是所有的 chunk 被释放后就立即被放到 bin 中。ptmalloc 为了提高分配的速度，会把一些小的 chunk 先放到 fast bins 的容器内。**而且，fastbin 容器中的 chunk 的使用标记总是被置位的，所以不满足上面的原则。**

##### Fast Bin

fast bin所包含chunk的大小为16 Bytes, 24 Bytes, 32 Bytes, … , 80 Bytes。当分配一块较小的内存(mem<=64 Bytes)时，**会首先检查对应大小的fastbin中是否包含未被使用的chunk，如果存在则直接将其从fastbin中移除并返回；否则通过其他方式（剪切top chunk）得到一块符合大小要求的chunk并返回。**

描述：

1. x86中，当用户释放堆堆块大小小于64B时使用fast bin进行管理，即chunk空间最大为80字节。
2. fast bin只用了fd，是单链表。
3. fast bin不会对P位进行操作，即它不会主动进行合并，只有在某些特定堆情况下，堆管理器才会对fast bin进行合并。
4. fast binY为管理fast bin的数组，每个成员分别管理不同大小的fast bin链表，且均指向了当前链表的尾结点，当尾结点被分配时，通过fd指针指向前一个节点。
5. 当用户申请chunk的大小小于或等于**MAX*FAST*SIZE**时，优先从fast bin中查找相应的空闲块，且规则为LIFO。

##### Small bin

small bins 中每个 chunk 的大小与其所在的 bin 的 index 的关系为：chunk\_size = 2 \* SIZE\_SZ \*index，具体如下：

|  |  |  |
| --- | --- | --- |
| 下标 | SIZE\_SZ=4（32 位） | SIZE\_SZ=8（64 位） |
| 2 | 16 | 32 |
| 3 | 24 | 48 |
| 4 | 32 | 64 |
| 5 | 40 | 80 |
| x | 2*4*x | 2*8*x |
| 63 | 504 | 1008 |

small bins 中一共有 62 个循环双向链表，每个链表中存储的 chunk 大小都一致。

描述：

1. small bin为双向链表，且使用FIFO,所以同一个链表中先被释放的 chunk 会先被分配出去。
2. 当满足small bin条件的chunk被释放后，会优先被放入unosrted bin，只有在一定情况下，才会被分配到small bin中。

##### Large bin

large bins中一共包含63个bin，每个bin中的chunk的大小不一致，而是处于一定的区间范围没，此外，这63个bin被分为了6组，每个bin中的chunk大小之间的公差一致，如下：

|  |  |  |
| --- | --- | --- |
| 组 | 数量 | 公差 |
| 1 | 32 | 64B |
| 2 | 16 | 512B |
| 3 | 8 | 4096B |
| 4 | 4 | 32768B |
| 5 | 2 | 262144B |
| 6 | 1 | 不限制 |

这里以 32 位平台的 large bin 为例，第一个 large bin 的起始 chunk 大小为 512 字节，位于第一组，所以该 bin 可以存储的 chunk 的大小范围为 [512,512+64)

##### unsorted bin

1. unsorted bin可以视为空闲chunk回归其所属bin之前的缓冲区。
2. 当释放较小或较大的chunk的时候，为了增加分配效率，系统会先将最近释放的chunk添加到unsorted bin中
3. unsorted bin 为一个双向循环链表，对chunk的大小没有限制，即任何大小的chunk都可以放入unsorted bin链表中

#### top chunk

程序在第一次进行malloc时，heap会分为两部分，一部分给用户，另一部分就是top chunk。

top chunk是处于当前堆的物理地址的最高的chunk，这个chunk不属于任何一个bin，当所有的 bin 都无法满足用户请求的内存大小时，如果其大小不小于指定的大小，就进行分配，并将剩下的部分作为新的 top chunk。否则，就对 heap 进行扩展后再进行分配。在 main arena 中通过 sbrk 扩展 heap，而在 thread arena 中通过 mmap 分配新的 heap。

### 宏观结构

#### arena

在之前介绍的例子中，无论是主线程还是新创建的线程，在第一次申请内存时，都会有独立的 arena。

但不是每个线程都有独立的arena，对于不同的系统，arena数量的约束如下：

For 32 bit systems:
  
 Number of arena = 2 \* number of cores.
  
For 64 bit systems:
  
 Number of arena = 8 \* number of cores.