2.2 分箱(bins)

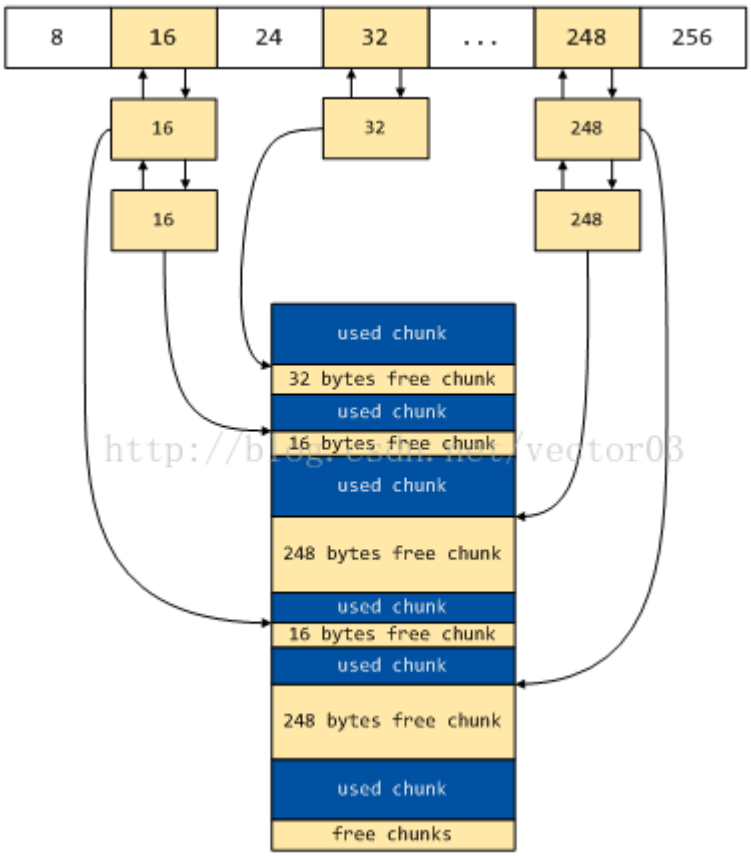
内存分配器设计中需要解决的两个重要问题就是空间和时间的矛盾.所谓空间矛盾是指要减少两方面的内存浪费,一是来自分配器本身overhead信息的占用,另外则来自分配的chunk由于对齐或碎片化造成的利用率降低.而时间矛盾是指在最短时间内,以最小的时间复杂度,计算出应该返回给用户的内存量以及内存地址. dlmalloc应对这两个矛盾使用的核心算法就是分箱机制.

所谓的分箱就是在内部划定一些chunk集合,每个集合中记录的都是固定大小或区间的free chunk,当分配时可以直接从中找到最贴近用户要求的那一个.显然,越是高效的分配,就越要将分箱划分的更细致,相应的也就浪费越多的内存.因此,规划分箱机制是一门涉及中庸之道的学问,既不能太粗放而影响效率,也不能太细致而降低利用率.

前面介绍过, dlmalloc将小于256字节的内存划分为small chunk, 256字节以上的划分为tree chunk.对这两种chunk的管理也采用不同方式. small chunk使用small bins管理,采用更精确的标记方法,而tree chunk用tree bins管理,较之使用相对粗放的管理方式.这种划分方法基于这样一种经验性的认知,即小块内存总是连续而频繁的分配,因此需要更加精确匹配,否则将产生时间和空间上的缺失,相反大块内存则往往一次性的分配,由此产生的代价和浪费相对较低.

2.2.1 small bins

dlmalloc对small bins的规定是这样的,以每8字节为一个分割,划分bins.也就是按照8, 16, 24, 32, ... , 256这样的排列,一共32个分箱.每个分箱中存放相同大小的free chunk,由一个双向环形链表管理.为了编程上的便捷,每个链表都附加了一个头节点(可以避免边界条件下繁冗的判断).我们可以通过下图来直观地理解这种分箱机制,



上图中描述了一段连续内存的分配情况,假定当前small bins中16, 32, 248字节的分箱下挂载了free chunk节点,这些节点交错分散在连续内存空间中.可以看到,每个分箱中头节点作为空闲链(FIFO)的入口,其本身并不包含实际数据.

当有分配请求时, 首先找到合适的分箱, 如果该分箱存在空闲chunk,则取出最前面的free chunk,再将剩余的chunk链接好即可.这种算法的优势在于,在大小适合的情况下, small chunk的分配是一个O(1)的过程.这对于前面所述的连续局部性的small chunk分配是一种非常合适的情况.

需要额外说明的一点是, 由于在dlmalloc中存在最小可分配chunk(见2.1.3小节的描述).因此目前8byte的分箱实际上是无效的(因为其小于最小可分配大小).

2.2.2 small bins索引寻址(Indexing)

dlmalloc记录small bins数据通过名为malloc_state的结构体实现,该结构为dlmalloc中的核心数据结构,后面会详细介绍.这里只关注在malloc_state中是如何记录small bins的.

```
#define NSMALLBINS        (32U)

mchunkptr  smallbins[(NSMALLBINS+1)*2];
```

可以看到, dlmalloc通过名为smallbins的mchunk指针数组来记录所有small bins分箱,该数组的长度为66,大小为264字节.这里可能会有人产生疑问,为什么数组长是66,不是说从8到256只有32个分箱吗?而且为什么又要建立指针数组,直接用mchunk数组不行吗?

其实在这里Doug Lea使用了一个巧妙而有趣的技巧,用作者的话说这是一个“reposition trick”.要进一步理解先来熟悉几个宏,

```
#define SMALLBIN_SHIFT    (3U)

#define is_small(s)         (((s) >> SMALLBIN_SHIFT) < NSMALLBINS)
#define small_index(s)      (bindex_t)((s)  >> SMALLBIN_SHIFT)
#define small_index2size(i) ((i)  << SMALLBIN_SHIFT)
#define MIN_SMALL_INDEX     (small_index(MIN_CHUNK_SIZE))
```

这几个比较好理解, is_small用于判断给定大小的chunk是否为small_chunk,也就是是否小于256字节.后面两个是在chunk size和分箱index之间转换,最后一个对应最小可用chunk的分箱索引,也就是最小可用分箱索引.

```
#define smallbin_at(M, i)   ((sbinptr)((void*)&((M)->smallbins[(i)<<1])))
```

这个宏是一个关键宏, 从字面上的意思是说给定一个malloc_state以及一个分箱索引,返回该分箱的指针,也就是头节点指针.该宏返回的sbinptr也就是mchunkptr,

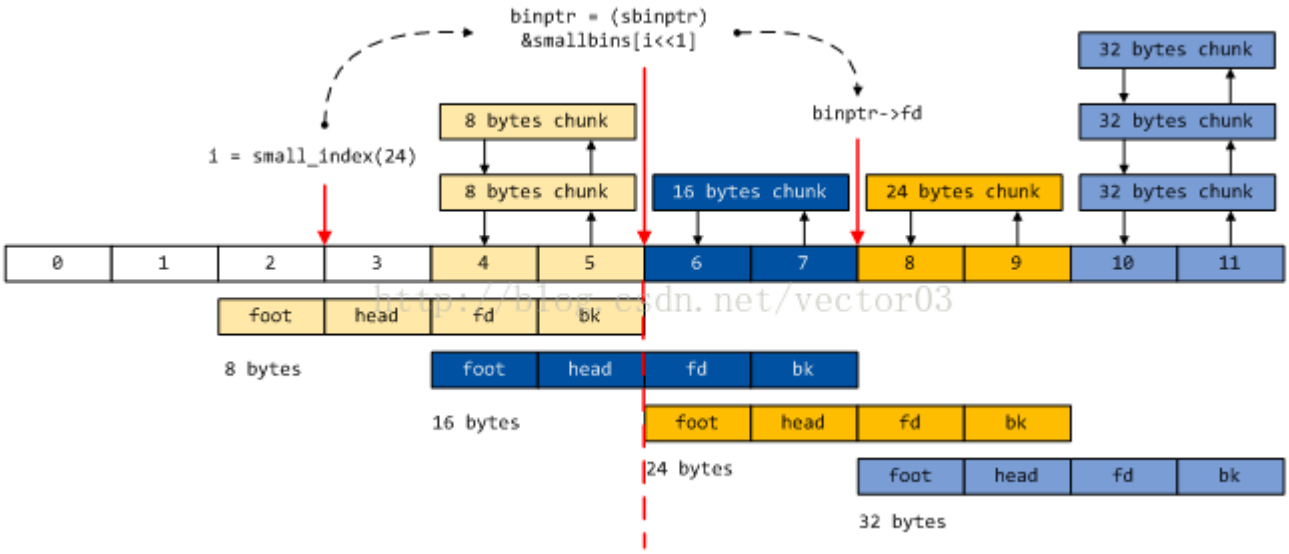
```
typedef struct malloc_chunk  mchunk;
typedef struct malloc_chunk* mchunkptr;
typedef struct malloc_chunk* sbinptr;  /* The type of bins of chunks */
```

按照一般的思路, 要表达small bins我们可能会建立一个长度为32的mchunk数组,在其中保存头节点,因此总共需要

16*32=512个字节.反观dlmalloc仅仅使用了264个字节就达到了同样的目的.

秘密在于, dlmalloc对这个长度为66的指针数组做了特殊处理.我们知道指针数组是一个外挂结构,每个元素会指向数组外的对象.而smallbins虽然名义上是一个指针数组,但它逻辑上的结构却是扁平的,其内部存放的就是mchunk本身.那有人会问,既然如此,数组长度也不够啊.关键在于,该数组中存放的mchunk不是按正常方式摆放,而是互相叠加存放的.即每一个mchunk的后半部分同后一个mchunk的前半部分是共用内存地址的.之所以可以做到这样,正是因为头节点只有fd和bk两个field有用,因此无形中省掉了那些用不到的结构体区域.

我们用一张图来形象的说明这种技巧,



从这张图中可以清晰的看到smallbins数组从形式到逻辑上一语双关的妙处.形式上看,除了前4个元素是被浪费的外,其余元素实际上都链接着相应分箱中的free chunk.因此你们可以理解为什么smallbins数组一定要使用mchunkptr类型了吧.至于长度31*2+4=66恰好就是从1到31分箱号(256归在tree chunk中)的二倍再加上额外的4个元素.从逻辑上看,图中使用不同颜色以区分不同大小的分箱.从这个角度分析,可以认为该数组的类型实际上是mchunk,只不过从1号分箱开始,每相邻两个分箱是叠加存放的,因为前两个field没有被使用.

我们以一个例子来说明通过索引寻址的过程.如上图,假设要查找大小为24的分箱地址,首先通过small_index宏计算出其所在的分箱号3,然后左移1位获得在smallbins数组中实际的索引6.图中索引6所在的地址实际上是2号分箱的成员fd,但在逻辑上也是3号分箱的首地址.取址并强制转化为sbinptr类型,就可以通过访问结构体成员fd和bk获得3号分箱中free chunk的指针.这里需要仔细体会这种强制类型转换的意义.

2.2.3 tree bins

与small bins简单的按照8字节划分分箱的方式相比, tree bins的机制更为复杂,因为其管理的范围高达256字节至理论上无穷大.为了在如此庞大的数据范围内快速查找可用的空闲chunk, dlmalloc引入了一种树型结构,这也是tree bins得名的原因(早期版本的dlmalloc对大型chunk同样使用双向链表管理,因为效率问题被抛弃了).

tree bins同样保存在核心结构体malloc_state中,是一个长度为32的指针数组,

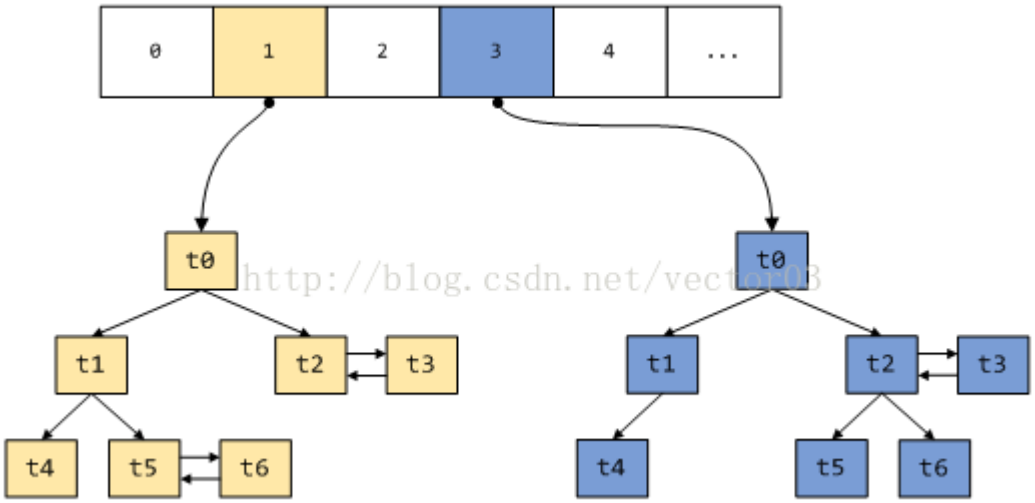
```
#define NTREEBINS       (32U)

tbinptr   treebins[NTREEBINS];

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

由于tree bins的每个分箱都要管理比small bins大的多的范围,所以使用树管理内部的free chunk.因此,与small bins不同, tree bins分箱不需要头节点,而仅仅保存一个指向该分箱树根节点的指针.

结合上面章节介绍的tree chunk结构,可以得到如下的示意图,



对于分箱范围划分, dlmalloc使用了一种规律性非常强的划分方式.相邻两个分箱,例如0号和1号, 2号和3号,被称为同一层级(tree level).下一层的分箱所覆盖的范围是上一层的2倍,同一层级间的两个分箱则平分该范围.譬如说, 0号分箱和1号分箱覆盖[2^8, 2^9)的范围,其中0号分箱负责[2^8, 2^8+2^7),而1号分箱则负责[2^8+2^7, 2^9),下一个level的2号和3号分箱则分别负责[2^9, 2^9+2^8)和[2^9+2^8, 2^10),依此类推.具体请参考下表,

箱号	区间范围	区间长度
0	[256, 384)	128
1	[384, 512)	128
2	[512, 768)	256
3	[768, 1024)	256
4	[1024, 1536)	512
5	[1536, 2048)	512
6	[2048, 3072)	1024

7	[3072, 4096)	1024
8	[4096, 6144)	2048
9	[6144, 8192)	2048
10	[8192, 12288)	4096
11	[12288, 16384)	4096
12	[16384, 24576)	8192
13	[24576, 32768)	8192
14	[32768, 49152)	16384
15	[49152, 65536)	16384
16	[65536, 98304)	32768
17	[98304, 131072)	32768
18	[131072, 196608)	65536
19	[196608, 262144)	65536
20	[262144, 393216)	131072
21	[393216, 524288)	131072
22	[524288, 786432)	262144
23	[786432, 1048576)	262144
24	[1048576, 1572864)	524288
25	[1572864, 2097152)	524288
26	[2097152, 3145728)	1048576
27	[3145728, 4194304)	1048576
28	[4194304, 6291456)	2097152
29	[6291456, 8388608)	2097152
30	[8388608, 12582912)	4194304
31	[12582912, ∞)	理论无穷大

2.2.4 tree bins索引寻址

同small bins一样, tree bins同样需要在给定chunk size的情况下,以O(1)快速计算出其所在的箱号.这里使用名为compute_tree_index(S, I)的宏.其字面意思是给定大小为S的chunk,返回值为I的索引.在源码中这个宏有四种实现,分别针对IA32/64体系下的GNUC, Intel C compiler, MS C compiler以及其他体系的C编译器.之所以有这些区分,是因为这里需要借助不同编译器提供的内联库函数,以加快计算速度.我们以IA32架构下的GNUC为例说明,

```
#define TREEBIN_SHIFT     (8U)

#define compute_tree_index(S, I)\
{\
  unsigned int X = S >> TREEBIN_SHIFT;\
  if (X == 0)\
    I = 0;\
  else if (X > 0xFFFF)\
    I = NTREEBINS-1;\
  else {\
    unsigned int K = (unsigned) sizeof(X)*__CHAR_BIT__ - 1 - (unsigned) __builtin_clz(X); \
    I =  (bindex_t)((K << 1) + ((S >> (K + (TREEBIN_SHIFT-1)) & 1)));\
  }\
}
```

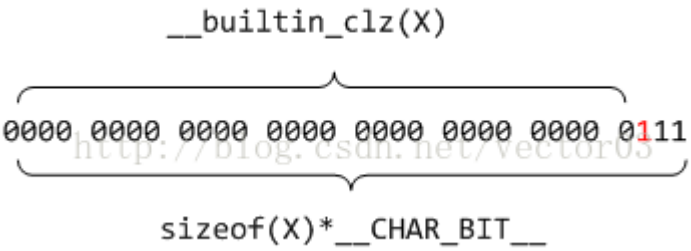

假设我们需要寻址的S = 1920.

首先, 将S右移TREEBIN_SHIFT位,也就是右移8位,得到X = 7. 如下,

之后根据结果X判断,前两个条件很简单,分别针对两种边界条件返回0号和31号分箱.重点在最后一个分支,

```
unsigned int K = (unsigned) sizeof(X)*__CHAR_BIT__ - 1 - (unsigned) __builtin_clz(X);
```

这一句看似很长, 实际的意思是得到数值X的最高有效位的位号.如下,

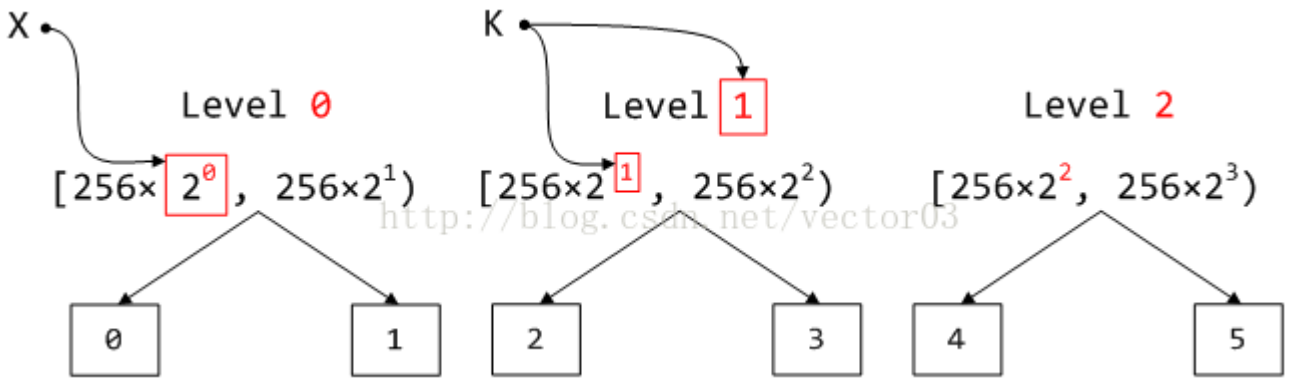


其中__builtin_clz()是glibc的一个内联库函数,实际上它是一条处理器指令CLZ (Count Leading Zeros)的封装.该指令可以统计从MSB到LSB遇到的第一个1之前所有0的个数.这里用32bit减去clz的结果,无非就是想知道最高有效位的位号.减1是因为位

号是从0开始计的,因此,该处返回值K = 2.

要说明的一点是, 该宏的四个版本区别就在这里. 如果使用intel或MS的编译器,将直接使用另外一条处理器指令BSR(Bit Scan Reverse), 即反向比特位扫描. 该指令可以更快速的直接返回最高有效位位号.

那么, 到这里求得的K值到底有什么用呢?如果我们把上一个小节介绍的tree bins范围重新变换一下书写方式,可能诸位就能看的比较明白了.



上图中,我们把范围改写为256字节与2的指数乘积的形式.显然,索引计算的第一步,右移8位得到的X就是后面的指数部分.

接着, 所谓的X最高有效位位号其实指的就是以2为底的指数幂.这很容易理解,二进制每左移一位就是乘以2,最高有效位位号可以认为从1开始左移的次数,这与乘以2的次数是一致的.

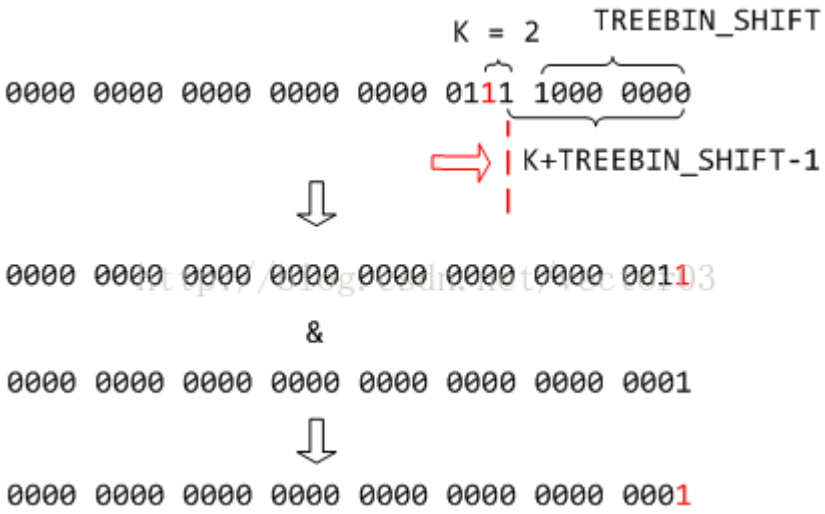
另外, 上一节提到的,相邻两个分箱分管一个level,每个level又是上个level范围的2倍,因此, K值就恰好与level值相同.这一点将会为后续计算带来极大的方便.

下面我们来看该如何计算I值,

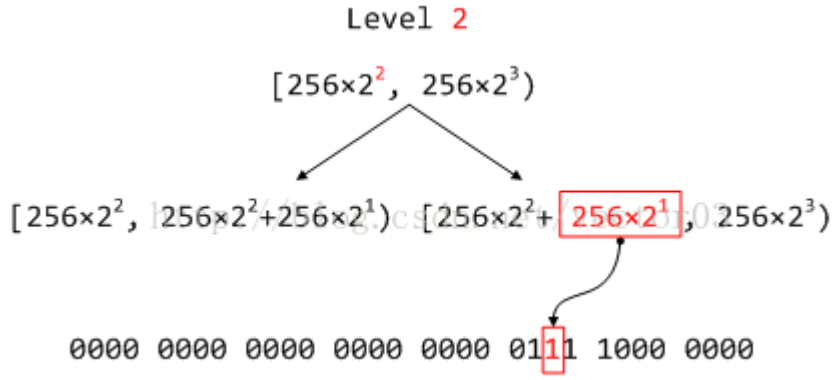
```
I =  (bindex_t)((K << 1) + ((S >> (K + (TREEBIN_SHIFT-1)) & 1)));
```

I值由两部分相加组成,前半部分K << 1是首先计算chunk落在哪两个分箱范围内.因为每level有两个分箱,需要乘以2.

而后半部分一连串的计算其实就是判断是落在左边还是右边的分箱,



可以看到s经过右移并与掩码计算的目的其实就是为了获得最高有效位的下一位,而这一位决定了落在哪一个分箱中,如图所示,



因为同一level的左右分箱平分该level的范围,对于两个连续区间的分界点来说,区别就在次最高有效位上.因此只要检查目标chunk的次最高有效位是否置1就可以断定它到底落在哪个分箱里.通过这种计算方法,我们可以快速获得I = 2 << 1 + 1 = 5.

用一句话总结就是, 计算tree bins索引,最重要的就是目标size的最高和次高有效位,这两bit决定了落在哪个分箱中.换个角度说,同一个分箱中的所有chunk头两位的位号和值是一致的.

2.2.5 数字搜索树(Digital Search Tree)

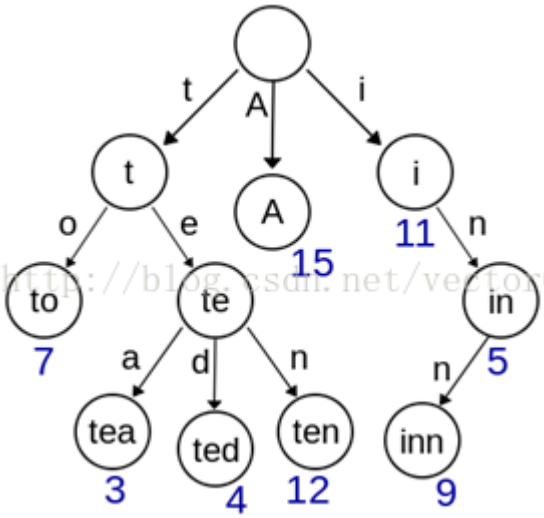
Tree bins的每个分箱下都挂载了一棵树, Doug Lea在原文中称之为"bitwise digital tree".事实上,这里Doug Lea表述的有些含糊,导致起初我认为这是一种二值字典树(binary trie tree),以致于后面各种看不懂.准确地说,这里使用的技术应该叫做数字搜索树(digital search tree),后面我们简称为DST.

首先介绍一下全值检索(whole-key search)和基值检索(radix search).

对于普通的查询结构, 如链表, 各种类型的BST等,检索过程主要是通过给定key值,与节点中储存的key作比较.这种检索方式就属于whole-key检索,即无论key是整型,浮点,或者字符串等,都需要从头至尾匹配后才能决定是否满足条件.

另外还存在一种非whole-key形式的检索,即将key值分割为若干的片段(segment),每个片段被称作一个数字(digital),而可选的数字集合被称作基值(radix).举例来说,对于一个32bit的整型数,我们可以将之看作由32个1bit片段组成,那么该key值的radix等于2,因为digital只能选择1或0.而如果将该数看作由8个4bit片段组成,则该key值的radix就等于16,依此类推.如果在检索时,每次仅需要比较key值的某一个digital,则称这种检索方式为radix search.

所谓的字典树(trie tree)就是基于radix search原理的一种算法.如图,



在trie树上,从根节点到叶子节点的path组成了一个完整的key值.比如上图中,给定key值"tea",可以分别查找到t节点, te节点以及tea节点.搜索时仅需要比较key值的当前字符与当前节点下各个子树保存的key值即可.这种算法对于BST的优势在于最大限度减少了无谓的比较,并且在词频和词缀统计方面具有先天结构上的优势.

dlmalloc中使用的DST实际上是一种特殊的trie树.它具有如下的性质,

1. 首先它是一棵二叉树, 且同普通BST树一样,它的左子树小于其右子树.

2. 不同于普通BST树, DST左右子树的root节点分别代表数字0和1.因此,每次在branching时仅需检测key值的1bit位,即它采用radix等于2的检索方式.但即便如此,它仍需要whole-key比较.

3. 不同于普通的trie树, DST在根节点,内部节点以及叶子节点上都保存键值信息(trie树在根节点和内部节点有可能没有键值,但叶子节点一定存在键值).

4. DST的任意子树中根节点与其左右子树内的节点都没有明确的排序关系.

按照描述的DST树性质,其搜索算法大致如下,

1. 设置当前节点指针指向root节点,并设i = 0.

2. 若当前节点指针为空, 则返回not found.

3. 比较当前节点储存的key, 如果相等,则返回找到节点.

4. 比较key值的第i bit,如果为0,将当前节点指针指向其左子树根节点;若为1,则将其指向右子树根节点.

5. 设i = i + 1, 并返回第2步.

与搜索算法类似, 其插入算法仅在第2步上存在区别,

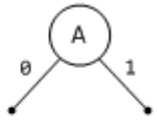
2. 若当前节点指针为空, 则建立新的节点,并在节点内储存key值,将其挂载到该子树所在的位置.

为了更详细地说明DST树的构造过程,我们以tree bins中范围最小的0号分箱来举例.出于简单,将省略多余的前导0.假设在当前的0号分箱下,管理如下大小的free chunk,

```
A  100000000 256
B  100110000 304
C  100001000 264
D  101000000 320
E  101001000 328
F  101010000 336
```

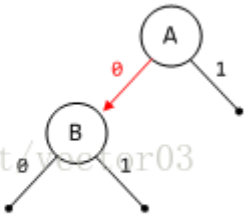

上一个小节介绍tree bins索引寻址时曾经说过,给定chunk size的头两个bit用以确定箱号,因此在dlmalloc中利用基值检索时是不包含这两bit的.另外,对于bit位的检测是按照从MSB到LSB的顺序执行的.

```
A  100000000 256
B  100110000 304
C  100001000 264
D  101000000 320
E  101001000 328
F  101010000 336
```



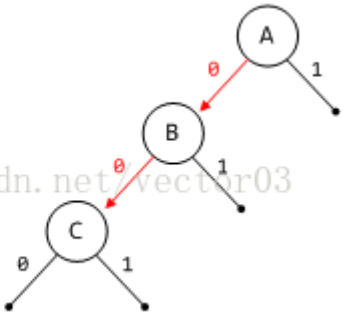
首先将A插入作为root节点.

```
A  100000000 256
B  100110000 304
C  100001000 264
D  101000000 320
E  101001000 328
F  101010000 336
```



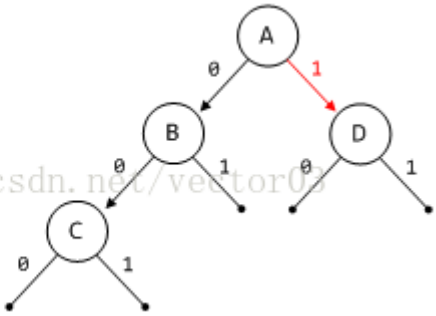
根据B节点的第6bit判断应该位于A节点的左子树.

```
A  100000000 256
B  100110000 304
C  100001000 264
D  101000000 320
E  101001000 328
F  101010000 336
```



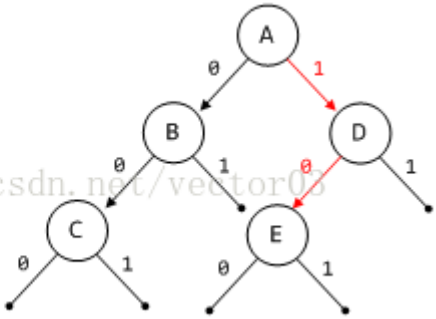
根据C节点的第5, 6bit判断应位于B节点的左子树.

```
A  100000000 256
B  100110000 304
C  100001000 264
D  101000000 320
E  101001000 328
F  101010000 336
```



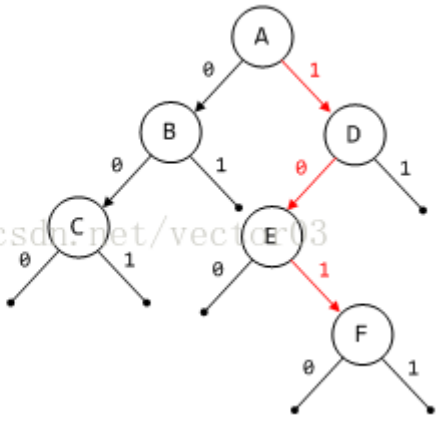
根据D节点的第6bit判断应位于A节点的右子树.

```
A  100000000 256
B  100110000 304
C  100001000 264
D  101000000 320
E  101001000 328
F  101010000 336
```



根据E节点的第5, 6bit判断应位于D节点的左子树.

```
A  100000000 256
B  100110000 304
C  100001000 264
D  101000000 320
E  101001000 328
F  101010000 336
```



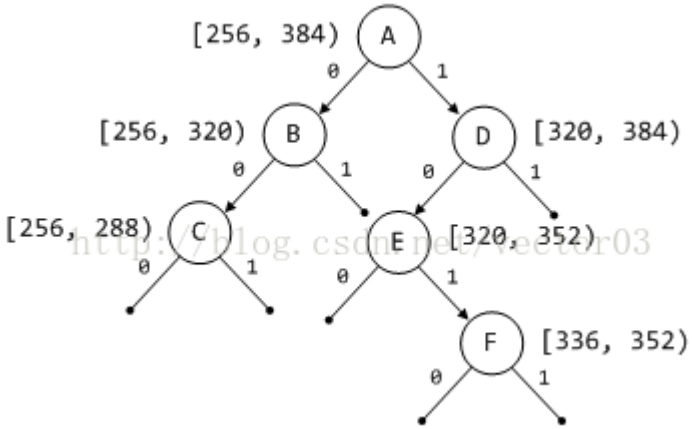
最后, 根据F节点的4, 5, 6bit判断应位于E节点的右子树.

从上面的例子中, 还可以发现DST的几个特点,

1. DST不是一棵排序树, 这一点从它的性质4可以得到验证.图中A节点虽然位于树根,但却是所有节点中数值最小的.而排序树的最小节点应该是其left-most节点.

2. DST的平衡度介乎于BST和AVL树之间.假设给定key的bit位数量不超过B,那么对于一棵具有N个节点的DST来说,其最坏情况下树高等于B.相比之下,同样具有N个节点的普通BST最坏情况下树高为N,也即完全退化为链表形态.若N <= B,两者的最坏情况应该是一致的.而当N > B时, DST由于存在公共前缀,性能会逐渐优于BST.且随着N不断增大,这种差距会越来越明显,直到N增加到Log_2N与B的数量级相接近时, DST将会近似成为一棵AVL树.

3. DST另一重要特性在于, 其具有天然的区间划分性质.在储存键值的同时,左右子树无形中划分了数值区间.显然,从根节点开始,同级level各个子树平分整个数值区间.且随着层数加深,子树数量增加,对数值的定位就越加精确.因此,尽管DST中各节点不存在严格的排序关系,但同级level各子树间却是严格从左至右的递增关系,如图,



关于dlmalloc在这里为什么选用DST的原因,我想可能有如下几点,

首先, DST具有的区间划分性质对于dlmalloc搜索最适(best-fitting)区间具有很大帮助.因为在搜索空闲chunk尤其是tree chunk时,并不总是能找到与目标size大小一致的结果,这时dlmalloc就会退而求其次,去寻找最接近目标size的chunk.而随着逐级向下搜索,理论上获得的chunk将会更接近目标size.

其次, 尽管DST的性能比不上平衡二叉树.但在样本逐渐增加的情况下(一般地,随着应用程序运行时间增加,内部节点数量递增会更加明显),能够提供比较接近的性能,同时在代码实现上却比后者大大简化,这一点提供了较好的性价比.

最后还有一点也非常重要. 平衡二叉树对DST的性能优势在算法上是说的通的,但实际执行时,某些情况下却未必能符合预期结果.尤其是现代处理器大多都支持超标量(superscalar)和乱序执行(out-of-order execution)等指令并发技术.以rb树为例,由于在节点增删时算法的复杂性(需要旋转子树及修改颜色token),导致内存读写的频度上升,其后果是引起CPU的TLB页面缓冲miss增加,以及多处理器下D-cache line同步等问题,无形中拉低了算法的执行效率.相比之下, DST更简单的算法产生更少的内存读写,尤其是写入操作更少,这有助于提升CPU多管线指令发射效率,从而以更差的算法反而得到了更快的实际执行速度.因此,在诸如像内存分配器这种随时会对内部节点进行增删操作的使用情况, DST成为一种较为理想的算法.