Libev中在管理定时器时，使用了堆这种结构，而且除了常见的最小2叉堆之外，它还实现了更高效的4叉堆。

之所以要实现4叉堆，是因为普通2叉堆的缓存效率较低，所谓缓存效率低，也就是说对CPU缓存的利用率比较低，说白了，就是违背了局部性原理。这是因为在2叉堆中，对元素的操作通常在N和N/2之间进行，所以对于含有大量元素的堆来说，两个操作数之间间隔比较远，对CPU缓存利用不太好。Libev中的注释说明，对于元素个数为50000+的堆来说，4叉堆的效率要提高5%所有。

在看Libev中堆的实现代码之前，先来看一个基本定理：对于n叉堆来说，使用数组进行存储时，下标为x的元素，其孩子节点的下标范围是[nx+1, nx+n]。比如2叉堆，下标为x的元素，其孩子节点的下标为2x+1和2x+2.

网上的大牛们觉得这个定理无需证明，只是规定。我还是钻了下牛角尖，试着证明如下，下面的证明都是以满n叉树为前提的：

对于满n叉树而言，每层的元素个数分别是：、、、、…，符合等比数列；

根据等比数列的求和公式，前m层元素总数为;

如果下标为x的元素处于第m层，则在第m层中，它前面有x - 个元素，因此在m+1层中，它的第一个孩子节点前面有n(x - )个元素，加上之前m层元素总和，在整个树中，它的第一个孩子节点前面共有元素个数为：

n(x - ) + =nx+1。所以，它的第一个孩子节点的下标为nx+1，因而得证其孩子节点的下标范围是[nx+1, nx+n]。

这个定理还可以用数学归纳法证明，非常简单，不再赘述。

根据定理，对于4叉堆而言，下标为x的元素，其孩子节点的下标范围是[4x+1, 4x+n]。还可以得出，其父节点的下标是(x-1)/4。然而在Libev的代码中，使用数组a存储堆时，4叉堆的第一个元素存放在a[3]，2叉堆的第一个元素存放在a[1]。

所以，对于Libev中的4叉堆实现而言，下标为k的元素（对应在正常实现中的下标是k-3），其孩子节点的下标范围是[4(k-3)+1+3, 4(k-3)+4+3]；其父节点的下标是((k-3-1)/4)+3。

对于Libev中的2叉堆实现而言，下标为k的元素（对应在正常实现中，其下标是k-1），其孩子节点的下标范围是[2(k-1)+1+1, 2(k-1)+2+1]，也就是[2k, 2k+1]；其父节点的下标是((k-1-1)/2)+1，也就是k/2。

下面来看Libev中的代码：

1：堆元素

#if EV\_HEAP\_CACHE\_AT

/\* a heap element \*/

**typedef** struct **{**

ev\_tstamp at**;**

WT w**;**

**}** ANHE**;**

#define ANHE\_w(he) (he).w /\* access watcher, read-write \*/

#define ANHE\_at(he) (he).at /\* access cached at, read-only \*/

#define ANHE\_at\_cache(he) (he).at = (he).w->at /\* update at from watcher \*/

#else

*/\* a heap element \*/*

**typedef** WT ANHE**;**

#define ANHE\_w(he) (he)

#define ANHE\_at(he) (he)->at

#define ANHE\_at\_cache(he)

#endif

ANHE就是堆元素，它要么就是一个指向时间监视器结构ev\_watcher\_time的指针（WT），要么除了包含该指针之外，还缓存了ev\_watcher\_time中的成员at。堆中元素就是根据at的值进行组织的，具有最小at值得节点就是根节点。

在Libev中，为了提高缓存命中率，在堆中缓存了元素at，文档中的原文是：

Heaps are not very cache-efficient. To improve the cache-efficiency of the timer and periodics heaps, libev can cache the timestamp (at) within the heap structure (selected by defining EV\_HEAP\_CACHE\_AT to 1), which uses 8-12 bytes more per watcher and a few hundred bytes more code, but avoids random read accesses on heap changes. This improves performance noticeably with many (hundreds) of watchers.

2：宏定义

#if EV\_USE\_4HEAP

#define DHEAP 4

#define HEAP0 (DHEAP - 1) /\* index of first element in heap \*/

#define HPARENT(k) ((((k) - HEAP0 - 1) / DHEAP) + HEAP0)

#define UPHEAP\_DONE(p,k) ((p) == (k))

**...**

#else

#define HEAP0 1

#define HPARENT(k) ((k) >> 1)

#define UPHEAP\_DONE(p,k) (!(p))

**...**

其中的宏HEAP0表示堆中第一个元素的下标；HPARENT是求下标为k的节点的父节点下标；UPHEAP\_DONE宏用于向上调整堆时，判断是否已经到达了根节点，对于4叉堆而言，根节点下标为3，其父节点的下标根据公式得出，也是3，所以结束的条件((p) == (k))，对于2叉堆而言，根节点下标为1，其父节点根据公式得出下标为0，所以结束的条件是(!(p))

3：向下调整堆

首先是4叉堆：

void downheap **(**ANHE **\***heap**,** int N**,** int k**)**

**{**

ANHE he **=** heap **[**k**];**

ANHE **\***E **=** heap **+** N **+** HEAP0**;**

**for** **(;;)**

**{**

ev\_tstamp minat**;**

ANHE **\***minpos**;**

ANHE **\***pos **=** heap **+** DHEAP **\*** **(**k **-** HEAP0**)** **+** HEAP0 **+** 1**;**

*/\* find minimum child \*/*

**if** **(**expect\_true **(**pos **+** DHEAP **-** 1 **<** E**))**

**{**

*/\* fast path \*/*

**(**minpos **=** pos **+** 0**),** **(**minat **=** ANHE\_at **(\***minpos**));**

**if** **(**ANHE\_at **(**pos **[**1**])** **<** minat**)**

**(**minpos **=** pos **+** 1**),** **(**minat **=** ANHE\_at **(\***minpos**));**

**if** **(**ANHE\_at **(**pos **[**2**])** **<** minat**)**

**(**minpos **=** pos **+** 2**),** **(**minat **=** ANHE\_at **(\***minpos**));**

**if** **(**ANHE\_at **(**pos **[**3**])** **<** minat**)**

**(**minpos **=** pos **+** 3**),** **(**minat **=** ANHE\_at **(\***minpos**));**

**}**

**else** **if** **(**pos **<** E**)**

**{**

*/\* slow path \*/*

**(**minpos **=** pos **+** 0**),** **(**minat **=** ANHE\_at **(\***minpos**));**

**if** **(**pos **+** 1 **<** E **&&** ANHE\_at **(**pos **[**1**])** **<** minat**)**

**(**minpos **=** pos **+** 1**),** **(**minat **=** ANHE\_at **(\***minpos**));**

**if** **(**pos **+** 2 **<** E **&&** ANHE\_at **(**pos **[**2**])** **<** minat**)**

**(**minpos **=** pos **+** 2**),** **(**minat **=** ANHE\_at **(\***minpos**));**

**if** **(**pos **+** 3 **<** E **&&** ANHE\_at **(**pos **[**3**])** **<** minat**)**

**(**minpos **=** pos **+** 3**),** **(**minat **=** ANHE\_at **(\***minpos**));**

**}**

**else**

**break;**

**if** **(**ANHE\_at **(**he**)** **<=** minat**)**

**break;**

heap **[**k**]** **=** **\***minpos**;**

ev\_active **(**ANHE\_w **(\***minpos**))** **=** k**;**

k **=** minpos **-** heap**;**

**}**

heap **[**k**]** **=** he**;**

ev\_active **(**ANHE\_w **(**he**))** **=** k**;**

**}**

如果理解普通二叉堆的向下调整算法的话，上面的代码还是很容易理解的。参数heap表示堆的起始地址，N表示堆中实际元素的总数，k表示需要调整元素的下标。

E表示堆中最后一个元素的下一个元素，用于判断是否已经到达了末尾。在foo循环中，首先得到节点heap [k]的第一个子节点的指针pos，pos + DHEAP – 1表示最后一个子节点的指针。

依次比较4个子节点，找到heap [k]所有子节点中的最小元素minpos。如果heap [k]的at值比minpos的at值还小，说明已经符合堆结构了，直接退出循环即可。否则的话，将minpos上移，依次循环下去。

ev\_active (ANHE\_w (\*minpos)) = k，将时间监视器的active成员置为其在堆中的下标。

然后是2叉堆：

void downheap **(**ANHE **\***heap**,** int N**,** int k**)**

**{**

ANHE he **=** heap **[**k**];**

**for** **(;;)**

**{**

int c **=** k **<<** 1**;**

**if** **(**c **>=** N **+** HEAP0**)**

**break;**

c **+=** c **+** 1 **<** N **+** HEAP0 **&&** ANHE\_at **(**heap **[**c**])** **>** ANHE\_at **(**heap **[**c **+** 1**])** **?** 1 **:** 0**;**

**if** **(**ANHE\_at **(**he**)** **<=** ANHE\_at **(**heap **[**c**]))**

**break;**

heap **[**k**]** **=** heap **[**c**];**

ev\_active **(**ANHE\_w **(**heap **[**k**]))** **=** k**;**

k **=** c**;**

**}**

heap **[**k**]** **=** he**;**

ev\_active **(**ANHE\_w **(**he**))** **=** k**;**

**}**

2叉堆的实现原理与4叉堆一样，不再赘述。

4：向上调整堆

void upheap **(**ANHE **\***heap**,** int k**)**

**{**

ANHE he **=** heap **[**k**];**

**for** **(;;)**

**{**

int p **=** HPARENT **(**k**);**

**if** **(**UPHEAP\_DONE **(**p**,** k**)** **||** ANHE\_at **(**heap **[**p**])** **<=** ANHE\_at **(**he**))**

**break;**

heap **[**k**]** **=** heap **[**p**];**

ev\_active **(**ANHE\_w **(**heap **[**k**]))** **=** k**;**

k **=** p**;**

**}**

heap **[**k**]** **=** he**;**

ev\_active **(**ANHE\_w **(**he**))** **=** k**;**

**}**

代码较简单，要调整的节点下标为k，首先得到其父节点下标p，然后判断heap[k]和heap[p]的关系作出调整。

5：其余代码

void adjustheap **(**ANHE **\***heap**,** int N**,** int k**)**

**{**

**if** **(**k **>** HEAP0 **&&** ANHE\_at **(**heap **[**k**])** **<=** ANHE\_at **(**heap **[**HPARENT **(**k**)]))**

upheap **(**heap**,** k**);**

**else**

downheap **(**heap**,** N**,** k**);**

**}**

*/\* rebuild the heap: this function is used only once and executed rarely \*/*

void reheap **(**ANHE **\***heap**,** int N**)**

**{**

int i**;**

*/\* we don't use floyds algorithm, upheap is simpler and is more cache-efficient \*/*

*/\* also, this is easy to implement and correct for both 2-heaps and 4-heaps \*/*

**for** **(**i **=** 0**;** i **<** N**;** **++**i**)**

upheap **(**heap**,** i **+** HEAP0**);**

**}**