**libevent源码深度剖析九**

——集成定时器事件  
张亮

     现在再来详细分析libevent中I/O事件和Timer事件的集成，与Signal相比，Timer事件的集成会直观和简单很多。Libevent对堆的调整操作做了一些优化，本节还会描述这些优化方法。

**1 集成到事件主循环**

     因为系统的I/O机制像select()和epoll\_wait()都允许程序制定一个最大等待时间（也称为最大超时时间）timeout，即使没有I/O事件发生，它们也保证能在timeout时间内返回。  
那么根据所有Timer事件的最小超时时间来设置系统I/O的timeout时间；当系统I/O返回时，再激活所有就绪的Timer事件就可以了，这样就能将Timer事件完美的融合到系统的I/O机制中了。  
具体的代码在源文件event.c的event\_base\_loop()中，现在就对比代码来看看这一处理方法：

**[cpp]** [view plain](http://blog.csdn.net/sparkliang/article/details/5054473) [copy](http://blog.csdn.net/sparkliang/article/details/5054473)

1. **if** (!base->event\_count\_active && !(flags & EVLOOP\_NONBLOCK)) {
2. // 根据Timer事件计算evsel->dispatch的最大等待时间
3. timeout\_next(base, &tv\_p);
4. } **else** {
5. // 如果还有活动事件，就不要等待，让evsel->dispatch立即返回
6. evutil\_timerclear(&tv);
7. }
8. // ...
9. // 调用select() or epoll\_wait() 等待就绪I/O事件
10. res = evsel->dispatch(base, evbase, tv\_p);
11. // ...
12. // 处理超时事件，将超时事件插入到激活链表中
13. timeout\_process(base);

    timeout\_next()函数根据堆中具有最小超时值的事件和当前时间来计算等待时间，下面看看代码：

**[cpp]** [view plain](http://blog.csdn.net/sparkliang/article/details/5054473) [copy](http://blog.csdn.net/sparkliang/article/details/5054473)

1. **static** **int** timeout\_next(**struct** event\_base \*base, **struct** timeval \*\*tv\_p)
2. {
3. **struct** timeval now;
4. **struct** event \*ev;
5. **struct** timeval \*tv = \*tv\_p;
6. // 堆的首元素具有最小的超时值
7. **if** ((ev = min\_heap\_top(&base->timeheap)) == NULL) {
8. // 如果没有定时事件，将等待时间设置为NULL,表示一直阻塞直到有I/O事件发生
9. \*tv\_p = NULL;
10. **return** (0);
11. }
12. // 取得当前时间
13. gettime(base, &now);
14. // 如果超时时间<=当前值，不能等待，需要立即返回
15. **if** (evutil\_timercmp(&ev->ev\_timeout, &now, <=)) {
16. evutil\_timerclear(tv);
17. **return** (0);
18. }
19. // 计算等待的时间=当前时间-最小的超时时间
20. evutil\_timersub(&ev->ev\_timeout, &now, tv);
21. **return** (0);
22. }

**2 Timer小根堆**

      Libevent使用堆来管理Timer事件，其key值就是事件的超时时间，源代码位于文件min\_heap.h中。  
所有的[**数据结构**](http://lib.csdn.net/base/datastructure)书中都有关于堆的详细介绍，向堆中插入、删除元素时间复杂度都是O(lgN)，N为堆中元素的个数，而获取最小key值（小根堆）的复杂度为O(1)。堆是一个完全二叉树，基本存储方式是一个数组。  
      Libevent实现的堆还是比较轻巧的，虽然我不喜欢这种编码方式（搞一些复杂的表达式）。轻巧到什么地方呢，就以插入元素为例，来对比说明，下面伪代码中的size表示当前堆的元素个数：  
典型的代码逻辑如下：

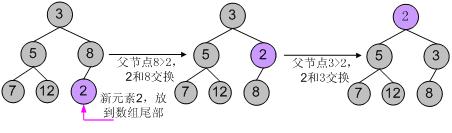
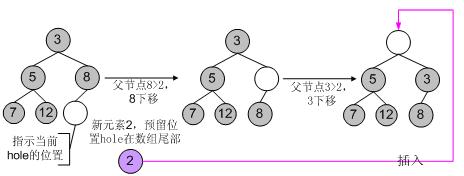
**[c-sharp]** [view plain](http://blog.csdn.net/sparkliang/article/details/5054473) [copy](http://blog.csdn.net/sparkliang/article/details/5054473)

1. Heap[size++] = **new**; // 先放到数组末尾，元素个数+1
2. // 下面就是shift\_up()的代码逻辑，不断的将new向上调整
3. \_child = size;
4. **while**(\_child>0) // 循环
5. {
6. \_parent = (\_child-1)/2; // 计算parent
7. **if**(Heap[\_parent].key < Heap[\_child].key)
8. **break**; // 调整结束，跳出循环
9. swap(\_parent, \_child); // 交换parent和child
10. }

     而libevent的heap代码对这一过程做了优化，在插入新元素时，只是为新元素预留了一个位置hole（初始时hole位于数组尾部），但并不立刻将新元素插入到hole上，而是不断向上调整hole的值，将父节点向下调整，最后确认hole就是新元素的所在位置时，才会真正的将新元素插入到hole上，因此在调整过程中就比上面的代码少了一次赋值的操作，代码逻辑是：  
     下面就是shift\_up()的代码逻辑，不断的将new的“预留位置”向上调整

**[cpp]** [view plain](http://blog.csdn.net/sparkliang/article/details/5054473) [copy](http://blog.csdn.net/sparkliang/article/details/5054473)

1. // 下面就是shift\_up()的代码逻辑，不断的将new的“预留位置”向上调整
2. \_hole = size; // \_hole就是为new预留的位置，但并不立刻将new放上
3. **while**(\_hole>0) // 循环
4. {
5. \_parent = (\_hole-1)/2; // 计算parent
6. **if**(Heap[\_parent].key < **new**.key)
7. **break**; // 调整结束，跳出循环
8. Heap[\_hole] = Heap[\_parent]; // 将parent向下调整
9. \_hole = \_parent; // 将\_hole调整到\_parent
10. }
11. Heap[\_hole] = **new**; // 调整结束，将new插入到\_hole指示的位置
12. size++; // 元素个数+1

     由于每次调整都少做一次赋值操作，在调整路径比较长时，调整效率会比第一种有所提高。libevent中的min\_heap\_shift\_up\_()函数就是上面逻辑的具体实现，对应的向下调整函数是min\_heap\_shift\_down\_()。  
举个例子，向一个小根堆3, 5, 8, 7, 12中插入新元素2，使用第一中典型的代码逻辑，其调整过程如下图所示：  
    
  
使用libevent中的堆调整逻辑，调整过程如下图所示：  
    
  
对于删除和元素修改操作，也遵从相同的逻辑，就不再罗嗦了。

**3 小节**

通过设置系统I/O机制的wait时间，从而简捷的集成Timer事件；主要分析了libevent对堆调整操作的优化。