**libevent源码深度剖析七**

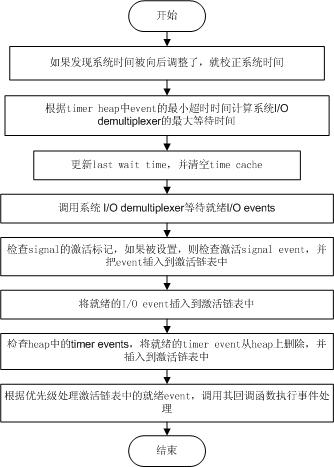
——事件主循环  
张亮

     现在我们已经初步了解了libevent的Reactor组件——event\_base和事件管理框架，接下来就是libevent事件处理的中心部分——事件主循环，根据系统提供的事件多路分发机制执行事件循环，对已注册的就绪事件，调用注册事件的回调函数来处理事件。

**1 阶段性的胜利**

    Libevent将I/O事件、定时器和信号事件处理很好的结合到了一起，本节也会介绍libevent是如何做到这一点的。  
    在看完本节的内容后，读者应该会对Libevent的基本框架：事件管理和主循环有比较清晰的认识了，并能够把libevent的事件控制流程清晰的串通起来，剩下的就是一些细节的内容了。

**2 事件处理主循环**

     Libevent的事件主循环主要是通过event\_base\_loop ()函数完成的，其主要操作如下面的流程图所示，event\_base\_loop所作的就是持续执行下面的循环。  
    
  
清楚了event\_base\_loop所作的主要操作，就可以对比源代码看个究竟了，代码结构还是相当清晰的。

**[cpp]** [view plain](http://blog.csdn.net/sparkliang/article/details/4987751) [copy](http://blog.csdn.net/sparkliang/article/details/4987751)

1. **int** event\_base\_loop(**struct** event\_base \*base, **int** flags)
2. {
3. **const** **struct** eventop \*evsel = base->evsel;
4. **void** \*evbase = base->evbase;
5. **struct** timeval tv;
6. **struct** timeval \*tv\_p;
7. **int** res, done;
8. // 清空时间缓存
9. base->tv\_cache.tv\_sec = 0;
10. // evsignal\_base是全局变量，在处理signal时，用于指名signal所属的event\_base实例
11. **if** (base->sig.ev\_signal\_added)
12. evsignal\_base = base;
13. done = 0;
14. **while** (!done) { // 事件主循环
15. // 查看是否需要跳出循环，程序可以调用event\_loopexit\_cb()设置event\_gotterm标记
16. // 调用event\_base\_loopbreak()设置event\_break标记
17. **if** (base->event\_gotterm) {
18. base->event\_gotterm = 0;
19. **break**;
20. }
21. **if** (base->event\_break) {
22. base->event\_break = 0;
23. **break**;
24. }
25. // 校正系统时间，如果系统使用的是非MONOTONIC时间，用户可能会向后调整了系统时间
26. // 在timeout\_correct函数里，比较last wait time和当前时间，如果当前时间< last wait time
27. // 表明时间有问题，这是需要更新timer\_heap中所有定时事件的超时时间。
28. timeout\_correct(base, &tv);
30. // 根据timer heap中事件的最小超时时间，计算系统I/O demultiplexer的最大等待时间
31. tv\_p = &tv;
32. **if** (!base->event\_count\_active && !(flags & EVLOOP\_NONBLOCK)) {
33. timeout\_next(base, &tv\_p);
34. } **else** {
35. // 依然有未处理的就绪时间，就让I/O demultiplexer立即返回，不必等待
36. // 下面会提到，在libevent中，低优先级的就绪事件可能不能立即被处理
37. evutil\_timerclear(&tv);
38. }
39. // 如果当前没有注册事件，就退出
40. **if** (!event\_haveevents(base)) {
41. event\_debug(("%s: no events registered.", \_\_func\_\_));
42. **return** (1);
43. }
44. // 更新last wait time，并清空time cache
45. gettime(base, &base->event\_tv);
46. base->tv\_cache.tv\_sec = 0;
47. // 调用系统I/O demultiplexer等待就绪I/O events，可能是epoll\_wait，或者select等；
48. // 在evsel->dispatch()中，会把就绪signal event、I/O event插入到激活链表中
49. res = evsel->dispatch(base, evbase, tv\_p);
50. **if** (res == -1)
51. **return** (-1);
52. // 将time cache赋值为当前系统时间
53. gettime(base, &base->tv\_cache);
54. // 检查heap中的timer events，将就绪的timer event从heap上删除，并插入到激活链表中
55. timeout\_process(base);
56. // 调用event\_process\_active()处理激活链表中的就绪event，调用其回调函数执行事件处理
57. // 该函数会寻找最高优先级（priority值越小优先级越高）的激活事件链表，
58. // 然后处理链表中的所有就绪事件；
59. // 因此低优先级的就绪事件可能得不到及时处理；
60. **if** (base->event\_count\_active) {
61. event\_process\_active(base);
62. **if** (!base->event\_count\_active && (flags & EVLOOP\_ONCE))
63. done = 1;
64. } **else** **if** (flags & EVLOOP\_NONBLOCK)
65. done = 1;
66. }
67. // 循环结束，清空时间缓存
68. base->tv\_cache.tv\_sec = 0;
69. event\_debug(("%s: asked to terminate loop.", \_\_func\_\_));
70. **return** (0);
71. }

**3 I/O和Timer事件的统一**

     Libevent将Timer和Signal事件都统一到了系统的I/O 的demultiplex机制中了，相信读者从上面的流程和代码中也能窥出一斑了，下面就再啰嗦一次了。  
     首先将Timer事件融合到系统I/O多路复用机制中，还是相当清晰的，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。  
     这是在Reactor和Proactor模式（主动器模式，比如Windows上的IOCP）中处理Timer事件的经典方法了，ACE采用的也是这种方法，大家可以参考POSA vol2书中的Reactor模式一节。  
     堆是一种经典的[**数据结构**](http://lib.csdn.net/base/datastructure)，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)；因此变成了管理Timer事件的绝佳人选（当然是非唯一的），libevent就是采用的堆结构。

**4 I/O和Signal事件的统一**

     Signal是异步事件的经典事例，将Signal事件统一到系统的I/O多路复用中就不像Timer事件那么自然了，Signal事件的出现对于进程来讲是完全随机的，进程不能只是[**测试**](http://lib.csdn.net/base/softwaretest)一个变量来判别是否发生了一个信号，而是必须告诉内核“在此信号发生时，请执行如下的操作”。  
如果当Signal发生时，并不立即调用event的callback函数处理信号，而是设法通知系统的I/O机制，让其返回，然后再统一和I/O事件以及Timer一起处理，不就可以了嘛。是的，这也是libevent中使用的方法。  
     问题的核心在于，当Signal发生时，如何通知系统的I/O多路复用机制，这里先买个小关子，放到信号处理一节再详细说明，我想读者肯定也能想出通知的方法，比如使用pipe。

**5 小节**

     介绍了libevent的事件主循环，描述了libevent是如何处理就绪的I/O事件、定时器和信号事件，以及如何将它们无缝的融合到一起。  
加油！