**libevent源码深度剖析六**

——初见事件处理框架  
张亮

 前面已经对libevent的事件处理框架和event结构体做了描述，现在是时候剖析libevent对事件的详细处理流程了，本节将分析libevent的事件处理框架event\_base和libevent注册、删除事件的具体流程，可结合前一节libevent对event的管理。

**1 事件处理框架-event\_base**

回想Reactor模式的几个基本组件，本节讲解的部分对应于Reactor框架组件。在libevent中，这就表现为event\_base结构体，结构体声明如下，它位于event-internal.h文件中：

**[cpp]** [view plain](http://blog.csdn.net/sparkliang/article/details/4985955) [copy](http://blog.csdn.net/sparkliang/article/details/4985955)

1. **struct** event\_base {
2. **const** **struct** eventop \*evsel;
3. **void** \*evbase;
4. **int** event\_count;  /\* counts number of total events \*/
5. **int** event\_count\_active; /\* counts number of active events \*/
6. **int** event\_gotterm;  /\* Set to terminate loop \*/
7. **int** event\_break;  /\* Set to terminate loop immediately \*/
8. /\* active event management \*/
9. **struct** event\_list \*\*activequeues;
10. **int** nactivequeues;
11. /\* signal handling info \*/
12. **struct** evsignal\_info sig;
13. **struct** event\_list eventqueue;
14. **struct** timeval event\_tv;
15. **struct** min\_heap timeheap;
16. **struct** timeval tv\_cache;
17. };

下面详细解释一下结构体中各字段的含义。  
1）evsel和evbase这两个字段的设置可能会让人有些迷惑，这里你可以把evsel和evbase看作是类和静态函数的关系，比如添加事件时的调用行为：evsel->add(evbase, ev)，实际执行操作的是evbase；这相当于class::add(instance, ev)，instance就是class的一个对象实例。  
evsel指向了全局变量static const struct eventop \*eventops[]中的一个；  
前面也说过，libevent将系统提供的I/O demultiplex机制统一封装成了eventop结构；因此eventops[]包含了select、poll、kequeue和epoll等等其中的若干个全局实例对象。  
evbase实际上是一个eventop实例对象；  
先来看看eventop结构体，它的成员是一系列的函数指针, 在event-internal.h文件中：  
struct eventop {  
 const char \*name;  
 void \*(\*init)(struct event\_base \*); // 初始化  
 int (\*add)(void \*, struct event \*); // 注册事件  
 int (\*del)(void \*, struct event \*); // 删除事件  
 int (\*dispatch)(struct event\_base \*, void \*, struct timeval \*); // 事件分发  
 void (\*dealloc)(struct event\_base \*, void \*); // 注销，释放资源  
 /\* set if we need to reinitialize the event base \*/  
 int need\_reinit;  
};  
也就是说，在libevent中，每种I/O demultiplex机制的实现都必须提供这五个函数接口，来完成自身的初始化、销毁释放；对事件的注册、注销和分发。  
比如对于epoll，libevent实现了5个对应的接口函数，并在初始化时并将eventop的5个函数指针指向这5个函数，那么程序就可以使用epoll作为I/O demultiplex机制了，这个在后面会再次提到。  
2）activequeues是一个二级指针，前面讲过libevent支持事件优先级，因此你可以把它看作是数组，其中的元素activequeues[priority]是一个链表，链表的每个节点指向一个优先级为priority的就绪事件event。   
3）eventqueue，链表，保存了所有的注册事件event的指针。  
4）sig是由来管理信号的结构体，将在后面信号处理时专门讲解；  
5）timeheap是管理定时事件的小根堆，将在后面定时事件处理时专门讲解；  
6）event\_tv和tv\_cache是libevent用于时间管理的变量，将在后面讲到；  
其它各个变量都能因名知意，就不再啰嗦了。

**2 创建和初始化event\_base**

创建一个event\_base对象也既是创建了一个新的libevent实例，程序需要通过调用event\_init()（内部调用event\_base\_new函数执行具体操作）函数来创建，该函数同时还对新生成的libevent实例进行了初始化。  
该函数首先为event\_base实例申请空间，然后初始化timer mini-heap，选择并初始化合适的系统I/O 的demultiplexer机制，初始化各事件链表；  
函数还检测了系统的时间设置，为后面的时间管理打下基础。

**3 接口函数**

前面提到Reactor框架的作用就是提供事件的注册、注销接口；根据系统提供的事件多路分发机制执行事件循环，当有事件进入“就绪”状态时，调用注册事件的回调函数来处理事件。  
Libevent中对应的接口函数主要就是：

**[cpp]** [view plain](http://blog.csdn.net/sparkliang/article/details/4985955) [copy](http://blog.csdn.net/sparkliang/article/details/4985955)

1. **int**  event\_add(**struct** event \*ev, **const** **struct** timeval \*timeout);
2. **int**  event\_del(**struct** event \*ev);
3. **int**  event\_base\_loop(**struct** event\_base \*base, **int** loops);
4. **void** event\_active(**struct** event \*event, **int** res, **short** events);
5. **void** event\_process\_active(**struct** event\_base \*base);

本节将按介绍事件注册和删除的代码流程，libevent的事件循环框架将在下一节再具体描述。  
对于定时事件，这些函数将调用timer heap管理接口执行插入和删除操作；对于I/O和Signal事件将调用eventopadd和delete接口函数执行插入和删除操作（eventop会对Signal事件调用Signal处理接口执行操作）；这些组件将在后面的内容描述。  
1）注册事件  
函数原型：  
int event\_add(struct event \*ev, const struct timeval \*tv)  
参数：ev：指向要注册的事件；  
tv：超时时间；  
函数将ev注册到ev->ev\_base上，事件类型由ev->ev\_events指明，如果注册成功，ev将被插入到已注册链表中；如果tv不是NULL，则会同时注册定时事件，将ev添加到timer堆上；  
如果其中有一步操作失败，那么函数保证没有事件会被注册，可以讲这相当于一个原子操作。这个函数也体现了libevent细节之处的巧妙设计，且仔细看程序代码，部分有省略，注释直接附在代码中。

**[cpp]** [view plain](http://blog.csdn.net/sparkliang/article/details/4985955) [copy](http://blog.csdn.net/sparkliang/article/details/4985955)

1. **int** event\_add(**struct** event \*ev, **const** **struct** timeval \*tv)
2. {
3. **struct** event\_base \*base = ev->ev\_base; // 要注册到的event\_base
4. **const** **struct** eventop \*evsel = base->evsel;
5. **void** \*evbase = base->evbase; // base使用的系统I/O策略
6. // 新的timer事件，调用timer heap接口在堆上预留一个位置
7. // 注：这样能保证该操作的原子性：
8. // 向系统I/O机制注册可能会失败，而当在堆上预留成功后，
9. // 定时事件的添加将肯定不会失败；
10. // 而预留位置的可能结果是堆扩充，但是内部元素并不会改变
11. **if** (tv != NULL && !(ev->ev\_flags & EVLIST\_TIMEOUT)) {
12. **if** (min\_heap\_reserve(&base->timeheap,
13. 1 + min\_heap\_size(&base->timeheap)) == -1)
14. **return** (-1);  /\* ENOMEM == errno \*/
15. }
16. // 如果事件ev不在已注册或者激活链表中，则调用evbase注册事件
17. **if** ((ev->ev\_events & (EV\_READ|EV\_WRITE|EV\_SIGNAL)) &&
18. !(ev->ev\_flags & (EVLIST\_INSERTED|EVLIST\_ACTIVE))) {
19. res = evsel->add(evbase, ev);
20. **if** (res != -1) // 注册成功，插入event到已注册链表中
21. event\_queue\_insert(base, ev, EVLIST\_INSERTED);
22. }
23. // 准备添加定时事件
24. **if** (res != -1 && tv != NULL) {
25. **struct** timeval now;
26. // EVLIST\_TIMEOUT表明event已经在定时器堆中了，删除旧的
27. **if** (ev->ev\_flags & EVLIST\_TIMEOUT)
28. event\_queue\_remove(base, ev, EVLIST\_TIMEOUT);
29. // 如果事件已经是就绪状态则从激活链表中删除
30. **if** ((ev->ev\_flags & EVLIST\_ACTIVE) &&
31. (ev->ev\_res & EV\_TIMEOUT)) {
32. // 将ev\_callback调用次数设置为0
33. **if** (ev->ev\_ncalls && ev->ev\_pncalls) {
34. \*ev->ev\_pncalls = 0;
35. }
36. event\_queue\_remove(base, ev, EVLIST\_ACTIVE);
37. }
38. // 计算时间，并插入到timer小根堆中
39. gettime(base, &now);
40. evutil\_timeradd(&now, tv, &ev->ev\_timeout);
41. event\_queue\_insert(base, ev, EVLIST\_TIMEOUT);
42. }
43. **return** (res);
44. }
46. event\_queue\_insert()负责将事件插入到对应的链表中，下面是程序代码；
47. event\_queue\_remove()负责将事件从对应的链表中删除，这里就不再重复贴代码了；
48. **void** event\_queue\_insert(**struct** event\_base \*base, **struct** event \*ev, **int** queue)
49. {
50. // ev可能已经在激活列表中了，避免重复插入
51. **if** (ev->ev\_flags & queue) {
52. **if** (queue & EVLIST\_ACTIVE)
53. **return**;
54. }
55. // ...
56. ev->ev\_flags |= queue; // 记录queue标记
57. **switch** (queue) {
58. **case** EVLIST\_INSERTED: // I/O或Signal事件，加入已注册事件链表
59. TAILQ\_INSERT\_TAIL(&base->eventqueue, ev, ev\_next);
60. **break**;
61. **case** EVLIST\_ACTIVE: // 就绪事件，加入激活链表
62. base->event\_count\_active++;
63. TAILQ\_INSERT\_TAIL(base->activequeues[ev->ev\_pri], ev, ev\_active\_next);
64. **break**;
65. **case** EVLIST\_TIMEOUT: // 定时事件，加入堆
66. min\_heap\_push(&base->timeheap, ev);
67. **break**;
68. }
69. }

2）删除事件：  
函数原型为：int  event\_del(struct event \*ev);  
该函数将删除事件ev，对于I/O事件，从I/O 的demultiplexer上将事件注销；对于Signal事件，将从Signal事件链表中删除；对于定时事件，将从堆上删除；  
同样删除事件的操作则不一定是原子的，比如删除时间事件之后，有可能从系统I/O机制中注销会失败。

**[cpp]** [view plain](http://blog.csdn.net/sparkliang/article/details/4985955) [copy](http://blog.csdn.net/sparkliang/article/details/4985955)

1. **int** event\_del(**struct** event \*ev)
2. {
3. **struct** event\_base \*base;
4. **const** **struct** eventop \*evsel;
5. **void** \*evbase;
6. // ev\_base为NULL，表明ev没有被注册
7. **if** (ev->ev\_base == NULL)
8. **return** (-1);
9. // 取得ev注册的event\_base和eventop指针
10. base = ev->ev\_base;
11. evsel = base->evsel;
12. evbase = base->evbase;
13. // 将ev\_callback调用次数设置为
14. **if** (ev->ev\_ncalls && ev->ev\_pncalls) {
15. \*ev->ev\_pncalls = 0;
16. }
18. // 从对应的链表中删除
19. **if** (ev->ev\_flags & EVLIST\_TIMEOUT)
20. event\_queue\_remove(base, ev, EVLIST\_TIMEOUT);
21. **if** (ev->ev\_flags & EVLIST\_ACTIVE)
22. event\_queue\_remove(base, ev, EVLIST\_ACTIVE);
23. **if** (ev->ev\_flags & EVLIST\_INSERTED) {
24. event\_queue\_remove(base, ev, EVLIST\_INSERTED);
25. // EVLIST\_INSERTED表明是I/O或者Signal事件，
26. // 需要调用I/O demultiplexer注销事件
27. **return** (evsel->del(evbase, ev));
28. }
29. **return** (0);
30. }

4 小节  
分析了event\_base这一重要结构体，初步看到了libevent对系统的I/O demultiplex机制的封装event\_op结构，并结合源代码分析了事件的注册和删除处理，下面将会接着分析事件管理框架中的主事件循环部分。