<https://blog.csdn.net/luotuo44/article/details/38458469>

**转载请注明出处:**[**http://blog.csdn.net/luotuo44/article/details/38458469**](http://blog.csdn.net/luotuo44/article/details/38458469)

        之前的博文讲了怎么实现线程、锁、内存分配、日志等功能的跨平台。Libevent最重要的跨平台功能还是实现了多路IO接口的跨平台(即Reactor模式)。这使得用户可以在不同的平台使用统一的接口。这篇博文就是来讲解Libevent是怎么实现这一点的。

        Libevent在实现线程、内存分配、日志时，都是使用了函数指针和全局变量。在实现多路IO接口上时，Libevent也采用了这种方式，不过还是有点差别的。

# 相关结构体：

        现在来看一下event\_base结构体，下面代码只列出了本文要讲的内容：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38458469) [copy](https://blog.csdn.net/luotuo44/article/details/38458469)

1. //event-internal.h文件
2. **struct** event\_base {
3. **const** **struct** eventop \*evsel;
4. **void** \*evbase;
6. …
7. };
8. **struct** eventop {
9. **const** **char** \*name; //多路IO复用函数的名字
11. **void** \*(\*init)(**struct** event\_base \*);
13. **int** (\*add)(**struct** event\_base \*, evutil\_socket\_t fd, **short** old, **short** events, **void** \*fdinfo);
14. **int** (\*del)(**struct** event\_base \*, evutil\_socket\_t fd, **short** old, **short** events, **void** \*fdinfo);
15. **int** (\*dispatch)(**struct** event\_base \*, **struct** timeval \*);
16. **void** (\*dealloc)(**struct** event\_base \*);
18. **int** need\_reinit; //是否要重新初始化
19. //多路IO复用的特征。参考http://blog.csdn.net/luotuo44/article/details/38443569
20. **enum** event\_method\_feature features;
21. **size\_t** fdinfo\_len; //额外信息的长度。有些多路IO复用函数需要额外的信息
22. };

        可以看到event\_base结构体中有一个struct eventop类型指针。而这个struct eventop结构体的成员就是一些函数指针。名称也像一个多路IO复用函数应该有的操作：add可以添加fd，del可以删除一个fd，dispatch可以进入监听。明显只要给event\_base的evsel成员赋值就能使用对应的多路IO复用函数了。

# 选择后端：

## 可供选择的后端：

        现在来看一下有哪些可以用的多路IO复用函数。其实在Libevent的源码目录中，已经为每一个多路IO复用函数专门创建了一个文件，如select.c、poll.c、epoll.c、kqueue.c等。

        打开这些文件就可以发现在文件的前面都会声明一些多路IO复用的操作函数，而且还会定义一个struct eventop类型的全局变量。如下面代码所示：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38458469) [copy](https://blog.csdn.net/luotuo44/article/details/38458469)

1. //select.c文件
2. **static** **void** \*select\_init(**struct** event\_base \*);
3. **static** **int** select\_add(**struct** event\_base \*, **int**, **short** old, **short** events, **void**\*);
4. **static** **int** select\_del(**struct** event\_base \*, **int**, **short** old, **short** events, **void**\*);
5. **static** **int** select\_dispatch(**struct** event\_base \*, **struct** timeval \*);
6. **static** **void** select\_dealloc(**struct** event\_base \*);
8. **const** **struct** eventop selectops = {
9. "select",
10. select\_init,
11. select\_add,
12. select\_del,
13. select\_dispatch,
14. select\_dealloc,
15. 0, /\* doesn't need reinit. \*/
16. EV\_FEATURE\_FDS,
17. 0,
18. };
20. //poll.c文件
21. **static** **void** \*poll\_init(**struct** event\_base \*);
22. **static** **int** poll\_add(**struct** event\_base \*, **int**, **short** old, **short** events, **void** \*\_idx);
23. **static** **int** poll\_del(**struct** event\_base \*, **int**, **short** old, **short** events, **void** \*\_idx);
24. **static** **int** poll\_dispatch(**struct** event\_base \*, **struct** timeval \*);
25. **static** **void** poll\_dealloc(**struct** event\_base \*);
27. **const** **struct** eventop pollops = {
28. "poll",
29. poll\_init,
30. poll\_add,
31. poll\_del,
32. poll\_dispatch,
33. poll\_dealloc,
34. 0, /\* doesn't need\_reinit \*/
35. EV\_FEATURE\_FDS,
36. **sizeof**(**struct** pollidx),
37. };

## 如何选定后端：

        看到这里，读者想必已经知道，只需将对应平台的多路IO复用函数的全局变量赋值给event\_base的evsel变量即可。可是怎么让Libevent根据不同的平台选择不同的多路IO复用函数呢？另外像大部分OS都会实现select、poll和一个自己的高效多路IO复用函数。怎么从多个中选择一个呢？下面看一下Libevent的解决方案吧：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38458469) [copy](https://blog.csdn.net/luotuo44/article/details/38458469)

1. //event.c文件
2. #ifdef \_EVENT\_HAVE\_EVENT\_PORTS
3. **extern** **const** **struct** eventop evportops;
4. #endif
5. #ifdef \_EVENT\_HAVE\_SELECT
6. **extern** **const** **struct** eventop selectops;
7. #endif
8. #ifdef \_EVENT\_HAVE\_POLL
9. **extern** **const** **struct** eventop pollops;
10. #endif
11. #ifdef \_EVENT\_HAVE\_EPOLL
12. **extern** **const** **struct** eventop epollops;
13. #endif
14. #ifdef \_EVENT\_HAVE\_WORKING\_KQUEUE
15. **extern** **const** **struct** eventop kqops;
16. #endif
17. #ifdef \_EVENT\_HAVE\_DEVPOLL
18. **extern** **const** **struct** eventop devpollops;
19. #endif
20. #ifdef WIN32
21. **extern** **const** **struct** eventop win32ops;
22. #endif
24. /\* Array of backends in order of preference. \*/
25. **static** **const** **struct** eventop \*eventops[] = {
26. #ifdef \_EVENT\_HAVE\_EVENT\_PORTS
27. &evportops,
28. #endif
29. #ifdef \_EVENT\_HAVE\_WORKING\_KQUEUE
30. &kqops,
31. #endif
32. #ifdef \_EVENT\_HAVE\_EPOLL
33. &epollops,
34. #endif
35. #ifdef \_EVENT\_HAVE\_DEVPOLL
36. &devpollops,
37. #endif
38. #ifdef \_EVENT\_HAVE\_POLL
39. &pollops,
40. #endif
41. #ifdef \_EVENT\_HAVE\_SELECT
42. &selectops,
43. #endif
44. #ifdef WIN32
45. &win32ops,
46. #endif
47. NULL
48. };

        它根据宏定义判断当前的OS环境是否有某个多路IO复用函数。如果有，那么就把与之对应的struct eventop结构体指针放到一个全局数组中。有了这个数组，现在只需将数组的某个元素赋值给evsel变量即可。因为是条件宏，在编译器编译代码之前完成宏的替换，所以是可以这样定义一个数组的。关于这些检测当前OS环境的宏，可以参考《[event-config.h指明所在系统的环境](http://blog.csdn.net/luotuo44/article/details/38300965)》。

        从数组的元素可以看到，低下标存的是高效多路IO复用函数。如果从低到高下标选取一个多路IO复用函数，那么将优先选择高效的。

## 具体实现：

        现在看一下Libevent是怎么选取一个多路IO复用函数的：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38458469) [copy](https://blog.csdn.net/luotuo44/article/details/38458469)

1. //event.c文件
2. **struct** event\_base \*
3. event\_base\_new\_with\_config(**const** **struct** event\_config \*cfg)
4. {
5. **int** i;
6. **struct** event\_base \*base;
7. **int** should\_check\_environment;
9. //分配并清零event\_base内存. event\_base里的所有成员都会为0
10. **if** ((base = mm\_calloc(1, **sizeof**(**struct** event\_base))) == NULL) {
11. event\_warn("%s: calloc", \_\_func\_\_);
12. **return** NULL;
13. }
15. ...
16. should\_check\_environment =
17. !(cfg && (cfg->flags & EVENT\_BASE\_FLAG\_IGNORE\_ENV));
18. //遍历数组的元素
19. **for** (i = 0; eventops[i] && !base->evbase; i++) {
20. **if** (cfg != NULL) {
21. /\* determine if this backend should be avoided \*/
22. **if** (event\_config\_is\_avoided\_method(cfg,
23. eventops[i]->name))
24. **continue**;
25. **if** ((eventops[i]->features & cfg->require\_features)
26. != cfg->require\_features)
27. **continue**;
28. }
30. /\* also obey the environment variables \*/
31. **if** (should\_check\_environment &&
32. event\_is\_method\_disabled(eventops[i]->name))
33. **continue**;
35. //找到了一个满足条件的多路IO复用函数
36. base->evsel = eventops[i];
38. //初始化evbase，后面会说到
39. base->evbase = base->evsel->init(base);
40. }
42. **if** (base->evbase == NULL) {
43. event\_warnx("%s: no event mechanism available",
44. \_\_func\_\_);
45. base->evsel = NULL;
46. event\_base\_free(base);
47. **return** NULL;
48. }
50. ....
52. **return** (base);
53. }

        可以看到，首先从eventops数组中选出一个元素。如果设置了event\_config，那么就对这个元素(即多路IO复用函数)特征进行检测，看其是否满足event\_config所描述的特征。关于event\_config，可以查看《[多路IO复用函数的选择配置](http://blog.csdn.net/luotuo44/article/details/38443569#t2)》。

# 后端数据存储结构体：

        在本文最前面列出的event\_base结构体中，除了evsel变量外，还有一个evbase变量。这也是一个很重要的变量，而且也是用于跨平台的。

        像select、poll、epoll之类多路IO复用函数在调用时要传入一些数据，比如监听的文件描述符fd，监听的事件有哪些。在Libevent中，这些数据都不是保存在event\_base这个结构体中的，而是存放在evbase这个指针指向的一个结构体中。

## IO复用结构体：

        由于不同的多路IO复用函数需要使用不同格式的数据，所以Libevent为每一个多路IO复用函数都定义了专门的结构体(即结构体是不同的)，本文姑且称之为**IO复用结构体。**evbase指向的就是这些结构体。由于这些结构体是不同的，所以要用一个void类型指针。

        在select.c、poll.c这类文件中都定义了属于自己的IO复用结构体，如下面代码所示：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38458469) [copy](https://blog.csdn.net/luotuo44/article/details/38458469)

1. //select.c文件
2. **struct** selectop {
3. **int** event\_fds;      /\* Highest fd in fd set \*/
4. **int** event\_fdsz;
5. **int** resize\_out\_sets;
6. fd\_set \*event\_readset\_in;
7. fd\_set \*event\_writeset\_in;
8. fd\_set \*event\_readset\_out;
9. fd\_set \*event\_writeset\_out;
10. };
12. //poll.c文件
13. **struct** pollop {
14. **int** event\_count;        /\* Highest number alloc \*/
15. **int** nfds;           /\* Highest number used \*/
16. **int** realloc\_copy;       /\* True iff we must realloc
17. \* event\_set\_copy \*/
18. **struct** pollfd \*event\_set;
19. **struct** pollfd \*event\_set\_copy;
20. };

        前面event\_base\_new\_with\_config的代码中，有下面一行代码：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38458469) [copy](https://blog.csdn.net/luotuo44/article/details/38458469)

1. base->evbase = base->evsel->init(base);

        明显这行代码就是用来赋值evbase的。下面是poll对应的init函数：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38458469) [copy](https://blog.csdn.net/luotuo44/article/details/38458469)

1. //poll.c文件
2. **static** **void** \*
3. poll\_init(**struct** event\_base \*base)
4. {
5. **struct** pollop \*pollop;
7. **if** (!(pollop = mm\_calloc(1, **sizeof**(**struct** pollop))))
8. **return** (NULL);
10. evsig\_init(base);//其他的一些初始化
12. **return** (pollop);
13. }

        经过上面的一些处理后，Libevent在特定的OS下能使用到特定的多路IO复用函数。在之前博文中说到的evmap\_io\_add和evmap\_signal\_add函数中都会调用evsel->add。由于在新建event\_base时就选定了对应的多路IO复用函数，给evsel、evbase变量赋值了，所以evsel->add能把对应的fd和监听事件加到对应的IO复用结构体保存。比如poll的add函数在一开始就有下面一行代码：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38458469) [copy](https://blog.csdn.net/luotuo44/article/details/38458469)

1. **struct** pollop\*pop = base->evbase;

        当然，poll的其他函数在一开始时也是会有这行代码的，因为要使用到fd和对应的监听事件等数据，就必须要获取那个IO复用结构体。

        由于有evsel和evbase这个两个指针变量，当初始化完成之后，再也不用担心具体使用的多路IO复用函数是哪个了。evsel结构体的函数指针提供了统一的接口，上层的代码要使用到多路IO复用函数的一些操作函数时，直接调用evsel结构体提供的函数指针即可。也正是如此，Libevent实现了统一的跨平台Reactor接口。

版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/luotuo44/article/details/38458469