<https://blog.csdn.net/luotuo44/article/details/38443569>

**转载请注明出处:**[**http://blog.csdn.net/luotuo44/article/details/38443569**](http://blog.csdn.net/luotuo44/article/details/38443569)

        前面的博文都是讲一些Libevent的一些辅助结构，现在来讲一下关键结构体：event\_base。

        这里作一个提醒，在阅读Libevent源码时，会经常看到backend这个单词。其直译是“后端”。实际上其指的是Libevent内部使用的多路IO复用函数，多路IO复用函数就是select、poll、epoll这类函数。本系列博文中，为了叙述方便，“多路IO复用函数”与“后端”这两种说法都会采用。

# 配置结构体：

        通常我们获取event\_base都是通过event\_base\_new()这个无参函数。使用这个无参函数，只能得到一个默认配置的event\_base结构体。本文主要是讲一些怎么获取一个非默认配置的event\_base以及可以对event\_base进行哪些配置。

        还是先看一下event\_base\_new函数吧。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38443569) [copy](https://blog.csdn.net/luotuo44/article/details/38443569)

1. //event.c文件
2. **struct** event\_base \*
3. event\_base\_new(**void**)
4. {
5. **struct** event\_base \*base = NULL;
6. **struct** event\_config \*cfg = event\_config\_new();
7. **if** (cfg) {
8. base = event\_base\_new\_with\_config(cfg);
9. event\_config\_free(cfg);
10. }
11. **return** base;
12. }

        可以看到，其先创建了一个event\_config结构体，并用cfg指针指向之，然后再用这个变量作为参数调用event\_base\_new\_with\_config。因为并没有对cfg进行任何的设置，所以得到的是默认配置的event\_base。

        从这里也可以知道，如果要对event\_base进行配置，那么对cfg变量进行配置即可。现在我们的目光从event\_base结构体转到event\_config结构体。

        先来看看event\_config结构体的定义。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38443569) [copy](https://blog.csdn.net/luotuo44/article/details/38443569)

1. **struct** event\_config {
2. TAILQ\_HEAD(event\_configq, event\_config\_entry) entries;
4. **int** n\_cpus\_hint;
5. **enum** event\_method\_feature require\_features;
6. **enum** event\_base\_config\_flag flags;
7. };
9. **struct** event\_config\_entry {
10. TAILQ\_ENTRY(event\_config\_entry) next;
12. **const** **char** \*avoid\_method;
13. };

        我们要做的就是对event\_config结构体的那四个成员变量进行配置。

# 具体的配置内容：

## 拒绝使用某个后端：

        第一个成员entries，其结构就不展开了，关于TAILQ\_HEAD，可以参考《[TAILQ\_QUEUE队列](http://blog.csdn.net/luotuo44/article/details/38374009)》。这里知道它是表示一个队列即可，队列元素的类型就是event\_config\_entry，可以用来存储一个字符串指针。它对应的设置函数为event\_config\_avoid\_method。

        Libevent是跨平台的Reactor，对于事件监听，其内部是使用多路IO复用函数。比较通用的多路IO复用函数是select和poll。而很多平台都提出了自己的高效多路IO复用函数，比如：epoll、devpoll、kqueue。Libevent对于这些多路IO复用函数都进行包装，供自己使用。event\_config\_avoid\_method函数就是指出，避免使用指定的多路IO复用函数。其是通过字符串的方式指定的，即参数method。这个字符串将由队列元素event\_config\_entry的avoid\_method成员变量存储(由于是指针，所以更准确来说是指向)。

        查看Libevent源码包里的文件，可以发现有诸如epoll.c、select.c、poll.c、devpoll.c、kqueue.c这些文件。打开这些文件就可以发现在文件内容的前面都会定义一个struct eventop类型变量。该结构体的第一个成员必然是一个字符串。这个字符串就描述了对应的多路IO复用函数的名称。所以是可以通过名称来禁用某种多路IO复用函数的。

        下面是event\_config\_avoid\_method函数的实现。其作用是把method指明的各个名称记录到entries成员变量中。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38443569) [copy](https://blog.csdn.net/luotuo44/article/details/38443569)

1. **int**
2. event\_config\_avoid\_method(**struct** event\_config \*cfg, **const** **char** \*method)
3. {
4. **struct** event\_config\_entry \*entry = mm\_malloc(**sizeof**(\*entry));
5. **if** (entry == NULL)
6. **return** (-1);
8. //复制字符串
9. **if** ((entry->avoid\_method = mm\_strdup(method)) == NULL) {
10. mm\_free(entry);
11. **return** (-1);
12. }
14. //插入到队列中
15. TAILQ\_INSERT\_TAIL(&cfg->entries, entry, next);
17. **return** (0);
18. }

上面的代码是设置拒绝使用某一个多路IO复用函数，在创建一个event\_base时怎么进行选择的可以参考[这一个链接](http://blog.csdn.net/luotuo44/article/details/38458469#t4)。

## 智能调整CPU个数：

        第二个成员变量n\_cpus\_hint。从名字来看是指明CPU的数量。是通过函数event\_config\_set\_num\_cpus\_hint来设置的。其作用是告诉event\_config，系统中有多少个CPU，以便作一些对线程池作一些调整来获取更高的效率。目前，仅仅Window系统的IOCP(Windows的IOCP能够根据CPU的个数智能调整)，该函数的设置才有用。在以后，Libevent可能会将之应用于其他系统。

        正如其名字中的hint，这仅仅是一个提示。就如同C++中的inline。event\_base实际使用的CPU个数不一定等于提示的个数。

## 规定所选后端需满足的特征：

        第三个成员变量require\_features。从其名称来看是要求的特征。不错，这个变量指定 多路IO复用函数应该满足哪些特征。所有的特征定义在一个枚举类型中。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38443569) [copy](https://blog.csdn.net/luotuo44/article/details/38443569)

1. //event.h文件
2. **enum** event\_method\_feature {
3. //支持边沿触发
4. EV\_FEATURE\_ET = 0x01,
5. //添加、删除、或者确定哪个事件激活这些动作的时间复杂度都为O(1)
6. //select、poll是不能满足这个特征的.epoll则满足
7. EV\_FEATURE\_O1 = 0x02,
8. //支持任意的文件描述符，而不能仅仅支持套接字
9. EV\_FEATURE\_FDS = 0x04
10. };

        这个成员变量是通过event\_config\_require\_features函数设置的。该函数的内部还是挺简单的。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38443569) [copy](https://blog.csdn.net/luotuo44/article/details/38443569)

1. **int**
2. event\_config\_require\_features(**struct** event\_config \*cfg,
3. **int** features)
4. {
5. **if** (!cfg)
6. **return** (-1);
7. cfg->require\_features = features;
8. **return** (0);
9. }

        从函数的实现可以看到，如果要设置多个特征，不能调用该函数多次，而应该使用位操作。比如: EV\_FEATURE\_O1 | EV\_FEATURE\_FDS作为参数。

        值得注意的是，对于某些系统，可能其提供的多路IO复用函数不能满足event\_config\_require\_features函数要求的特征，此时event\_base\_new\_with\_config函数将返回NULL，即得不到一个满足条件的event\_base。所以在设置这个特征时，那么就要检查event\_base\_new\_with\_config的返回值是否为NULL，像下面代码那样。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38443569) [copy](https://blog.csdn.net/luotuo44/article/details/38443569)

1. #include<event.h>
2. #include<stdio.h>
4. **int** main()
5. {
6. event\_config \*cfg = event\_config\_new();
7. event\_config\_require\_features(cfg,  EV\_FEATURE\_O1 | EV\_FEATURE\_FDS);
9. event\_base \*base = event\_base\_new\_with\_config(cfg);
10. **if**( base == NULL )
11. {
12. printf("don't support this features\n");
13. base = event\_base\_new(); //使用默认的。
14. }
16. …..
17. **return** 0;
18. }

        上面代码中，如果是在Linux运行，也是返回NULL。即epoll都不能同时满足那个两个特征。

        那么怎么知道多路IO复用函数支持哪些特征呢？前面说到的一个机构体struct eventop中有一个成员正是enum event\_method\_feature features。在Libevent-2.0.21-stable中是倒数第二个成员。打开epoll.c、select.c、poll.c、devpoll.c、kqueue.c这些文件，查看里面定义的struct eventop类型变量，就可以看到各个多路IO复用函数都支持哪些特征。在epoll.c文件可以看到，epoll支持EV\_FEATURE\_ET|EV\_FEATURE\_O1。所以前面的代码中，返回NULL。

## 其他一些设置：

        第四个变量flags是通过函数event\_config\_set\_flag设置的。函数的实现很简单。注意，函数的内部是进行 |= 运算的。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38443569) [copy](https://blog.csdn.net/luotuo44/article/details/38443569)

1. //event.c文件
2. **int**
3. event\_config\_set\_flag(**struct** event\_config \*cfg, **int** flag)
4. {
5. **if** (!cfg)
6. **return** -1;
7. cfg->flags |= flag;
8. **return** 0;
9. }

        现在来看一下参数flag可以取哪些值。

* EVENT\_BASE\_FLAG\_NOLOCK：不要为event\_base分配锁。设置这个选项可以为event\_base节省一点加锁和解锁的时间，但是当多个线程访问event\_base会变得不安全
* EVENT\_BASE\_FLAG\_IGNORE\_ENV：选择多路IO复用函数时，不检测EVENT\_\*环境变量。使用这个标志要考虑清楚：因为这会使得用户更难调试程序与Libevent之间的交互
* EVENT\_BASE\_FLAG\_STARTUP\_IOCP：仅用于Windows。这使得Libevent在启动时就启用任何必需的IOCP分发逻辑，而不是按需启用。如果设置了这个宏，那么evconn\_listener\_new和bufferevent\_socket\_new函数的内部将使用IOCP
* EVENT\_BASE\_FLAG\_NO\_CACHE\_TIME：在执行event\_base\_loop的时候没有cache时间。该函数的while循环会经常取系统时间，如果cache时间，那么就取cache的。如果没有的话，就只能通过系统提供的函数来获取系统时间。这将更耗时
* EVENT\_BASE\_FLAG\_EPOLL\_USE\_CHANGELIST：告知Libevent，如果决定使用epoll这个多路IO复用函数，可以安全地使用更快的基于changelist 的多路IO复用函数：epoll-changelist多路IO复用可以在多路IO复用函数调用之间，同样的fd 多次修改其状态的情况下，避免不必要的系统调用。但是如果传递任何使用dup()或者其变体克隆的fd给Libevent，epoll-changelist多路IO复用函数会触发一个内核bug，导致不正确的结果。在不使用epoll这个多路IO复用函数的情况下，这个标志是没有效果的。也可以通过设置EVENT\_EPOLL\_USE\_CHANGELIST 环境变量来打开epoll-changelist选项

        综观上面4个变量的设置，特征设置event\_config\_require\_features和CPU数目设置event\_config\_set\_num\_cpus\_hint两者的函数调用会覆盖之前的设置。如果要同时设置多个，那么需要在参数中使用位运算中的 | 。而另外两个变量的设置可以通过多次调用函数的方式同时设置多个值。

# 获取当前配置：

        前面的介绍的都是设置，现在来讲一下获取。主要有下面几个。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38443569) [copy](https://blog.csdn.net/luotuo44/article/details/38443569)

1. **const** **char** \*\*event\_get\_supported\_methods(**void**);
2. **const** **char** \*event\_base\_get\_method(**const** **struct** event\_base \*);
3. **int** event\_base\_get\_features(**const** **struct** event\_base \*base);
4. **static** **int**  event\_config\_is\_avoided\_method(**const** **struct** event\_config \*cfg, **const** **char** \*method)

        第一个函数是获取当前系统所支持的多路IO复用函数有哪些。第二个函数需要一个event\_base结构体作为参数，说明是在new到一个event\_base之后才能调用的。该函数返回值是对应event\_base\* 当前所采用的多路IO复用函数是哪个。第三个函数则是获取参数event\_base当前所采用的特征是什么。第四个函数则说明参数method指明的多路IO复用函数是不是被参数cfg所禁用了。如果是禁用了，返回非0值。不禁用就返回0。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38443569) [copy](https://blog.csdn.net/luotuo44/article/details/38443569)

1. #include<event2/event.h>
2. #include<stdio.h>
4. #ifdef WIN32
5. #include<WinSock2.h>
6. #endif
8. **int** main()
9. {
10. #ifdef WIN32
11. WSADATA wsa\_data;
12. WSAStartup(0x0201, &wsa\_data);
13. #endif
15. **const** **char**\*\* all\_methods = event\_get\_supported\_methods();
17. **while**( all\_methods && \*all\_methods )
18. {
19. printf("%s\t", \*all\_methods++);
20. }
22. printf("\n");
24. event\_base \*base = event\_base\_new();
25. **if**( base )
26. printf("current method:\t %s\n", event\_base\_get\_method(base) );
27. **else**
28. printf("base == NULL\n");
30. #ifdef WIN32
31. WSACleanup();
32. #endif
34. **return** 0;
35. }

        上面代码在我的Ubuntu10.04上运行，其结果为：

        epoll        poll        select

        currentmethod:        epoll

        在Win7 + VS2010的运行结果为

        win32

        currentmethod:        win32

参考：

<http://www.wangafu.net/~nickm/libevent-book/Ref2_eventbase.html>

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/38443569