libevent源码分析

1.开篇

Source Insight

Doxgen工具

do{}while(0)宏

函数调用关系图

2.event-config.h指明所在系统的环境

3.日志和错误处理

日志处理

定制日志回调函数

日志API以及日志消息处理流程

错误处理

4.内存分配

5.多线程、锁、条件变量(一)

开启多线程

锁和条件变量结构体

Libevent封装的多线程

定制的顺序

6.多线程、锁、条件变量(二)

Debug锁操作

开启调试功能

debug递归锁

debug解锁

定制线程锁、条件变量

锁的使用

加锁和解锁

断言已加锁

7. TAILQ_QUEUE队列

队列结构体

队列操作宏函数以及使用例子

展开宏函数

特殊指针操作

队尾节点

前一个节点

8.event_io_map哈希表

哈希结构体

什么情况会使用哈希表

哈希函数

哈希表操作函数

哈希表在Libevent的使用

9.event_signal_map

相关结构体

libevent源码分析

1.开篇

原文地址

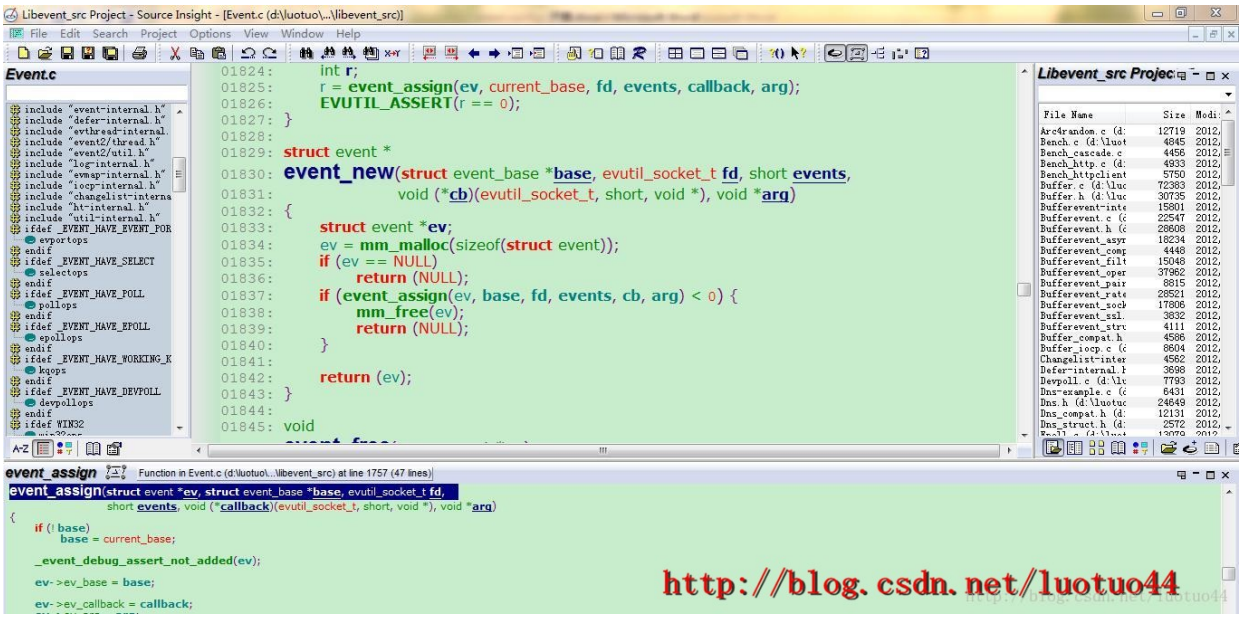
我所分析的Libevent版本是2.0.21版本，是目前最新的稳定版本。看这系列博文中，需要读者有Linux编程的一些基础。因为像POSIX、文件描述符、多线程等等这些概念，我并不会去解释，我默认读者已经熟悉这些概念了。如果读者读过《UNIX环境高级编程》，那就完全没问题了。

因为Libevent是跨平台的，所以它使用了很多它自己定义的通用跨平台类型，比如evutil_socket_t。此外，Libevent也定义了一些跨平台通用的API，这些都可以在《通用类型和函数》一文中找到。

相信来看本系列的文章的读者，都不会是刚刚接触Libevent的用户。这里就不说Libevent的优点和怎么安装使用Libevent了。我是想介绍其他东西。

Source Insight

这个工具是阅读代码的神器。下面是一个截图。



正中央是代码窗口。在代码中，如果你想看一下event_assign函数的内部是怎么实现的，那么你不用找到event_assign函数实现文件，然后打开，再Ctrl + F查找。在Source Insight中，你只需用鼠标单击一下event_assign函数。那么就会在下面的那个窗口显示event_assign函数的具体实现。是不是很厉害的功能？

右边的窗口是文件列表，和其他IDE的功能差不多。

左边那个窗口功能也是很强大的，特殊是当代码中出现了很多条件宏。在这样的条件下，这个宏会被定义成这样。在那样的条件下，又会定义成那样。左边的窗口可以清楚地看到。

Doxgen工具

这是一个可以制作chm文件的工具。

在Source Insight中，虽然是很容易追踪到某个函数的具体实现(实现都在c文件中，非头文件)。但Libevent的源文件中很少有注释，也没有这个函数的解释。而且Source Insight无法追踪到函数的声明，只能追踪到定义(就是函数的实现)（或许是我对Source Insight还不熟悉，如果有这样的功能，还望大家指出）。

而由Doxgen工具生成的chm文件是离线把的帮助文档，它会列出函数的说明、参数和返回值。

怎么用Doxgen工具制作Libevent的离线版帮助文档，可以参考这里。

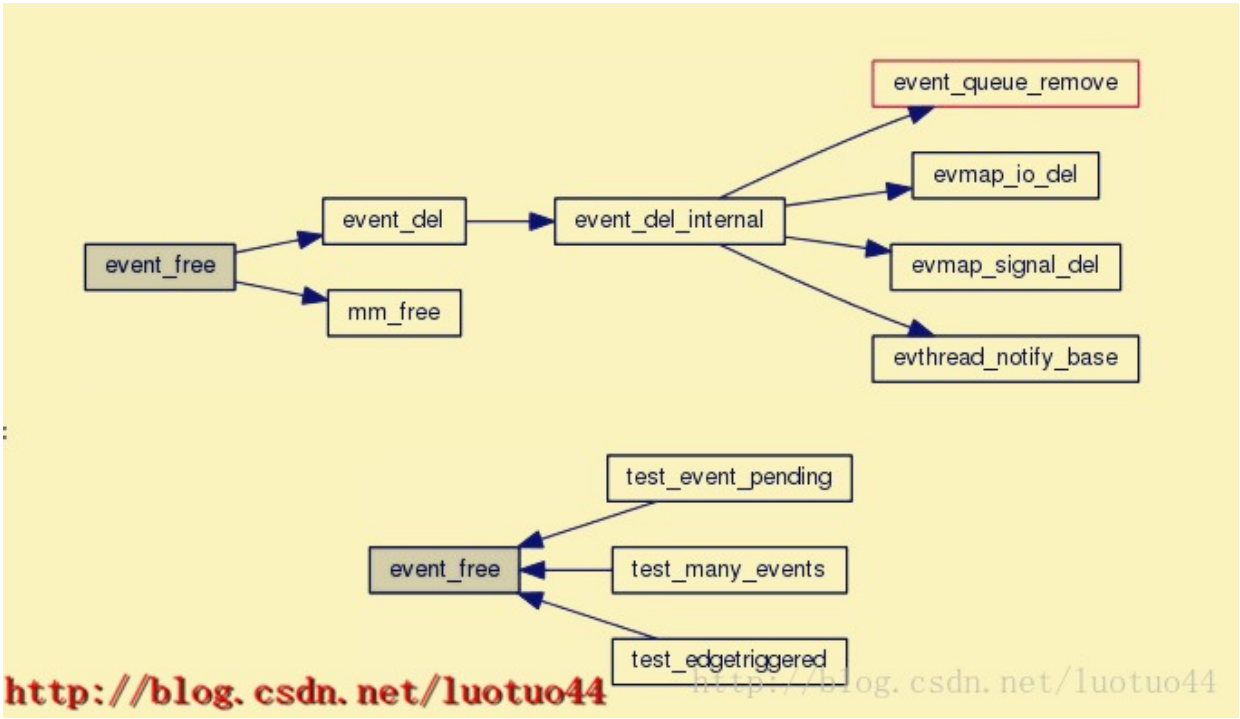
do{}while(0)宏

在Libevent的源代码中，经常能看到do{}while(0)宏的使用。如果是第一次碰到这种写法，估计都会比较不解。可以参考这里来解惑。

函数调用关系图

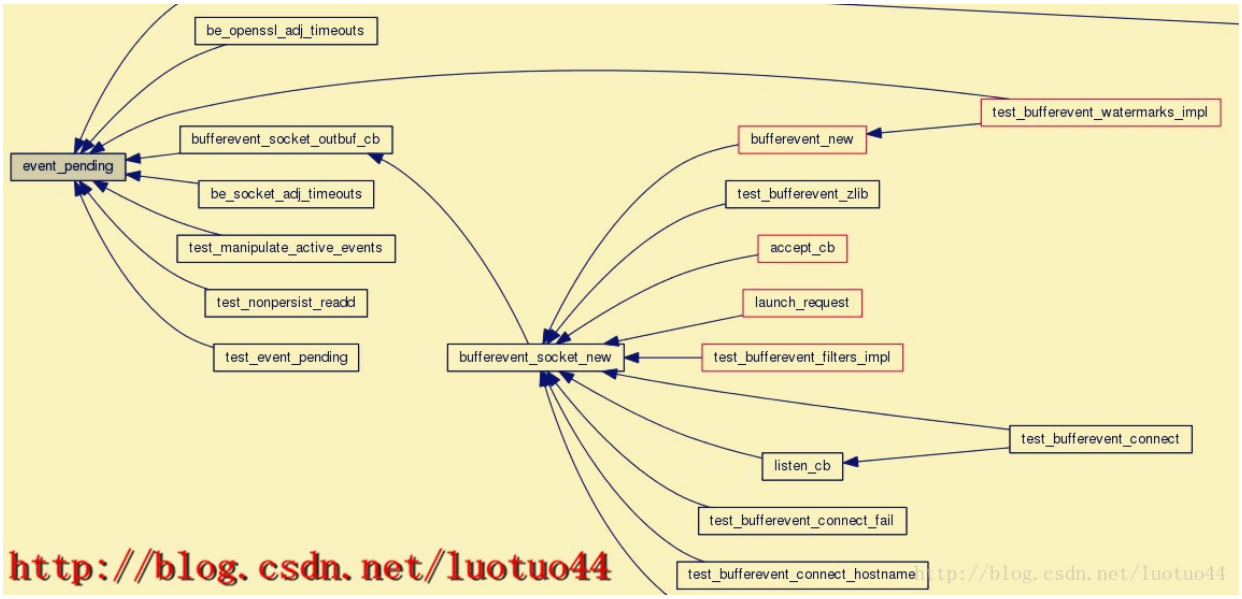
有时候追踪一个函数，想知道该函数的调用关系图。有一个网站提供了这个关系图。

下面举两个例子。



图中，上面的是event_free函数会调用哪些函数，一颗主调用树。下面的是哪些函数会调用event_free函数，是被调用关系。其中test_event_pending这些是Libevent提供的测试例子的测试函数。

下面再给另外一个被调用关系的图：



有一个不足之处，这个网站并没有和Libevent同步更新，目前提供的最高Libevent版本是2.0.3-alpha。

2.event-config.h指明所在系统的环境

原文地址

如果你打开Libevent的一些文件，比如util.h文件。就会发现使用了很多宏定义，并根据一些宏定义而进行条件编译。这些宏定义往往来自event-config.h文件中。

如util.h文件的代码开始处：

```
#ifdef _EVENT_HAVE_SYS_TIME_H
#include <sys/time.h>
#endif
#ifdef _EVENT_HAVE_STDINT_H
#include <stdint.h>
#elif defined(_EVENT_HAVE_INTTYPES_H)
#include <inttypes.h>
#endif
```

其会根据是否定义了某个宏，而决定是否包含某个头文件。从宏的名字来看，其指明了是否有这个头文件。有时还会指明是否有某个函数。这样做的原因很简单，因为Libevent是跨平台的，必须得考虑到某些系统可能没有一些头文件或者函数。

event-config.h文件是一个很基础和重要的文件。在文件的一开始有这样一句”This file was generated by autoconf when libevent was built”。这说明这个文件是在Libevent配置的时候生成的，即在编译Libevent之前就应该要生成该文件了。当然也早于我们在Libevent基础上编写应用程序。

其在编译之前就检查所在的系统的一些情况。比如是否含有某个文件或者函数。其对这些进行检测，然后把结果写入到event-config.h文件中。等到编译Libevent和编译我们的APP时，会include该头文件。

PS：上面两段的说法有点错误。待修改。

该文件大部分内容是根据config.h.in文件生成的。比如，config.h.in文件里面有下面的代码：

```
/* Define to 1if you have the <arpa/inet.h> header file. */
#undefHAVE_ARPA_INET_H
.....
/* Define ifyour system supports the epoll system calls */
#undefHAVE_EPOLL
```

对应地，Linux内核版本在2.6以上的Linux对应生成的event-config.h文件会定义这两个宏，如下：

```
/* Define to 1if you have the <arpa/inet.h> header file. */
#define_EVENT_HAVE_ARPA_INET_H 1
......
/* Define to 1if you have the <sys/epoll.h> header file. */
#define_EVENT_HAVE_SYS_EPOLL_H 1
```

而在Windows系统下生成的event-config.h文件就没有定义这两个宏。

可以说，event-config.h这个文件定义的宏指明了所在的系统有哪些可用的头文件、函数和一些配置。

又比如对于gcc来说，是支持**func**这个宏的，但对于VS编译器就不支持，VS对应功能的宏为**FUNCTION**。此时在Windows系统的event-config.h文件中，就会定义：

```
/* Define toappropriate substitue if compiler doesnt have __func__ */
#define_EVENT___func__ __FUNCTION__
```

而在util-internal.h文件中，有这样的定义：

```
#ifdef_EVENT___func__
#define __func___EVENT___func__
#endif
```

这样就可以在其他文件中通用**func**宏了，无需关注是什么系统了。

event-config.h文件的有些内容是根据编译Libevent时的配置选项生成的。比如是否支持多线程这个选项。如果配置Libevent的时，加入了这样一句：

```
./configure --disable-thread-support
```

那么，在event-config.h文件将定义DISABLE_THREAD_SUPPORT这个宏，此时得到的Libevent是不支持多线程的。

3.日志和错误处理

原文地址

日志处理

在Libevent的源码中，经常会见到形如event_warn、event_msgx、event_err之类的函数。这通常出现在代码中一些值是不合理时。这些函数就是Libevent的日志函数。它能把这些不合理的情况打印出来，告知用户。

定制日志回调函数

Libevent在默认情况下，会将这些日志信息输出到终端上。这当然就不利于日后的观察。为此，Libevent允许用户定制自己的日志回调函数。所有的日志函数在最后输出信息时，都会调用日志回调函数的。所以用户可以通过定制自己的日志回调函数（在回调函数中把信息输出到一个文件上），方便日后的查看。定制回调函数就像设置自己信号处理函数那样，设置一个日志回调函数。当有日志时，Libevent库就会调用这个日志回调函数。

回调函数的格式和日志定制函数如下所示：

```
typedef void (*event_log_cb)(int severity, const char *msg);
void event_set_log_callback(event_log_cb cb);
```

回调函数中的第一个参数severity是日志级别类型，有下面这些：

```
#define EVENT_LOG_DEBUG 0
#define EVENT_LOG_MSG   1
#define EVENT_LOG_WARN  2
#define EVENT_LOG_ERR   3

/* Obsolete names: these are deprecated, but older programs might use the
m.
 * They violate the reserved-identifier namespace. */
#define _EVENT_LOG_DEBUG EVENT_LOG_DEBUG
#define _EVENT_LOG_MSG EVENT_LOG_MSG
#define _EVENT_LOG_WARN EVENT_LOG_WARN
#define _EVENT_LOG_ERR EVENT_LOG_ERR
```

值得注意的是，不能在你的日志回调函数里面调用任何Libevent提供的API函数，否则将发生未定义行为。

在实现上，Libevent是通过定义一个全局函数指针变量来保存用户在日志定制函数中传入的参数cb。日志定制函数在实现上也是很简单的:

```
static event_log_cb log_fn = NULL;
void
event_set_log_callback(event_log_cb cb)
{
	log_fn = cb;
}
```

从event_set_log_callback的实现代码可以看到，并没有对这个参数cb做任何检查。

Libevent的默认日志处理函数event_log函数还是很简陋的，只是简单地根据参数判断日志记录的级别，然后把级别和日志内容输出。复杂一点的日志功能，可以参考muduo日志功能。当然也有log4pp、log4xx这些把一个重量级的日志库。

```
static void
event_log(int severity, const char *msg)
{
    if (log_fn)
        log_fn(severity, msg);//调用用户的日志回调函数
    else {
        const char *severity_str;
        switch (severity) {
        case _EVENT_LOG_DEBUG:
            severity_str = "debug";
            break;
        case _EVENT_LOG_MSG:
            severity_str = "msg";
            break;
        case _EVENT_LOG_WARN:
            severity_str = "warn";
            break;
        case _EVENT_LOG_ERR:
            severity_str = "err";
            break;
        default:
            severity_str = "???";
            break;
        }
        (void)fprintf(stderr, "[%s] %s\n", severity_str, msg);//输出到标准
错误终端上
    }
}
```

从event_log函数中可以看到，当函数指针log_fn不用NULL时，就调用log_fn指向的函数。否则就直接向stderr输出日志信息。所以，设置自己的日志回调函数后，如果想恢复Libevent默认的日志回调函数，只需再次调用event_set_log_callback函数，参数设置为NULL即可。

日志API以及日志消息处理流程

Libevent的日志API的使用也是挺简单的。首先，使用者确定要记录的日志的级别和错误类型，然后调用对应的日志函数。有下面这些可供选择的日志函数。

```
void event_err(int eval, const char *fmt, ...);
void event_warn(const char *fmt, ...);
void event_sock_err(int eval, evutil_socket_t sock, const char *fmt,
...);
void event_sock_warn(evutil_socket_t sock, const char *fmt, ...);

void event_errx(int eval, const char *fmt, ...);
void event_warnx(const char *fmt, ...);
void event_msgx(const char *fmt, ...);
void _event_debugx(const char *fmt, ...);
```

这些函数都是声明在log-internal.h文件中。所以用户并不能使用之，这些函数都是Libevent内部使用的。

这些函数内部实现都差不多。下面是其中几个实现：

```
void
event_warn(const char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    _warn_helper(_EVENT_LOG_WARN, strerror(errno), fmt, ap);
    va_end(ap);
}

void
event_sock_err(int eval, evutil_socket_t sock, const char *fmt, ...)
{
    va_list ap;
    int err = evutil_socket_geterror(sock);

    va_start(ap, fmt);
    _warn_helper(_EVENT_LOG_ERR, evutil_socket_error_to_string(err), fmt,
 ap);
    va_end(ap);
    event_exit(eval);
}
```

可以看到，主要是设置调用的日记级别，把可变参数用va_list变量记录，然后调用_warn_helper这个辅助函数。其中，event_warn和event_warnx的区别是_warn_helper函数的第二个参数分别是strerror(errno)和NULL。

而_warn_helper这个辅助函数也是蛮简单的。

```
static void
_warn_helper(int severity, const char *errstr, const char *fmt, va_list a
p)
{
    char buf[1024];
    size_t len;

    //如果有可变参数，就把可变参数格式化到一个缓存区buf中。
    if (fmt != NULL)
        evutil_vsnprintf(buf, sizeof(buf), fmt, ap);
    else
        buf[0] = '\0';

    //如果有额外的信息描述，把这些信息追加到可变参数的后面。
    if (errstr) {
        len = strlen(buf);
       //-3是因为还有另外三个字符，冒号、空格和\0。
        if (len < sizeof(buf) - 3) {
            evutil_snprintf(buf + len, sizeof(buf) - len, ": %s", errst
r);
        }
    }

    //把缓存区的数据作为一条日志记录，调用Libevent的日志函数。
    event_log(severity, buf);//这个函数也是最前面的说到的那个日志处理函数
}
```

这些日志函数的使用，如同printf函数那样，支持%s, %d之类的格式化输出。比如：

```
event_warnx("Far too many %s (%d)", "wombats", 99);
```

值得注意的是，也如同printf那样，可变参数中的参数要和第一个参数中的格式要求相匹配。在这些日志函数的声明中，如果是GNU 的编译器，将会检查是否匹配。其声明如下：

```
#ifdef __GNUC__
#define EV_CHECK_FMT(a,b) __attribute__((format(printf, a, b)))
#define EV_NORETURN __attribute__((noreturn))
#else
#define EV_CHECK_FMT(a,b)
#define EV_NORETURN
#endif

#define _EVENT_ERR_ABORT ((int)0xdeaddead)

void event_err(int eval, const char *fmt, ...) EV_CHECK_FMT(2,3) EV_NORET
URN;
void event_warn(const char *fmt, ...) EV_CHECK_FMT(1,2);
void event_sock_err(int eval, evutil_socket_t sock, const char *fmt, ...)
 EV_CHECK_FMT(3,4) EV_NORETURN;
void event_sock_warn(evutil_socket_t sock, const char *fmt, ...) EV_CHECK
_FMT(2,3);
void event_errx(int eval, const char *fmt, ...) EV_CHECK_FMT(2,3) EV_NORE
TURN;
void event_warnx(const char *fmt, ...) EV_CHECK_FMT(1,2);
void event_msgx(const char *fmt, ...) EV_CHECK_FMT(1,2);
void _event_debugx(const char *fmt, ...) EV_CHECK_FMT(1,2);
```

错误处理

Libevent库运行的时候有可能会致命发生错误，此时，Libevent的默认行为是终止程序。同日志处理一样，用户也是可以用Libevent定制自己的错误处理函数。错误处理函数的格式和定制函数如下：

```
typedef void (*event_fatal_cb)(int err);

void event_set_fatal_callback(event_fatal_cb cb);
```

定制函数的内部实现原理和前面的日志处理函数一样，都是通过一个全局函数指针变量存储用户的错误处理函数。

在默认情况下，Libevent处理这些致命错误时会粗暴地杀死程序，但大多数情况下，Libevent在调用这个致命处理函数前都会调用前面的日志记录函数，其级别是_EVENT_LOG_ERR。此时，虽然程序突然死了，但还是可以在日志中找到一些信息。用户的错误处理函数也应该杀死程序。

如果要定制自己的日志处理函数和错误处理函数，那么应该在程序的一开始位置就进行定制。

4.内存分配

原文地址

Libevent的内存分配函数还是比较简单的，并没有定义内存池之类的东西。如同前一篇博客那样，给予Libevent库的使用者充分的设置权(定制)，即可以设置用户(Libevent库的使用者)自己的内存分配函数。至于怎么分配，主动权在于用户。但在设置(定制)的时候要注意一些地方，下面会说到。

首先，如果要定制自己的内存分配函数，就得在一开始配置编译Libevent库是，不能加入–disable-malloc-replacement选项。默认情况下，是没有这个选项的。如果加入了这个选项，那么将会在生成的event-config.h中，定义_EVENT_DISABLE_MM_REPLACEMENT这个宏。关于event-config.h文件，可以参考博文。下面是Libevent内存分配函数的声明(在mm-internal.h文件)：

```
//mm-internal.h文件
#ifndef _EVENT_DISABLE_MM_REPLACEMENT
void *event_mm_malloc_(size_t sz);
void *event_mm_calloc_(size_t count, size_t size);
char *event_mm_strdup_(const char *s);
void *event_mm_realloc_(void *p, size_t sz);
void event_mm_free_(void *p);
#define mm_malloc(sz) event_mm_malloc_(sz)
#define mm_calloc(count, size) event_mm_calloc_((count), (size))
#define mm_strdup(s) event_mm_strdup_(s)
#define mm_realloc(p, sz) event_mm_realloc_((p), (sz))
#define mm_free(p) event_mm_free_(p)
#else
#define mm_malloc(sz) malloc(sz)
#define mm_calloc(n, sz) calloc((n), (sz))
#define mm_strdup(s) strdup(s)
#define mm_realloc(p, sz) realloc((p), (sz))
#define mm_free(p) free(p)
#endif
```

这些内存分配函数是给Libevent使用的，而非用户(从这些接口声明在mm-internal.h文件中就可以看到这一点)。Libevent的其他函数要申请内存就调用mm_malloc之类的宏定义。如果一开始在配置的时候(event-config.h)就禁止用户定制自己的内存分配函数，那么就把这些宏定义为C语言标准内存分配函数。

当然，即使没有禁止，如果用户没有定制自己的内存分配函数，最终还是调用C语言的标准内存分配函数。这一点在event_mm_xxxx这些函数的实现上可以看到。

这些函数的实现是在event.c文件中的。定制功能的实现原理和前一篇博客中说到的定制实现原理是一样的。如下：

```
#ifndef _EVENT_DISABLE_MM_REPLACEMENT
static void *(*_mm_malloc_fn)(size_t sz) = NULL;
static void *(*_mm_realloc_fn)(void *p, size_t sz) = NULL;
static void (*_mm_free_fn)(void *p) = NULL;

void
event_set_mem_functions(void *(*malloc_fn)(size_t sz),
            void *(*realloc_fn)(void *ptr, size_t sz),
            void (*free_fn)(void *ptr))
{
    _mm_malloc_fn = malloc_fn;
    _mm_realloc_fn = realloc_fn;
    _mm_free_fn = free_fn;
}
```

用户就是通过调用event_set_mem_functions函数来定制自己的内存分配函数。虽然这个函数不做任何的检查，但还是有一点要注意。这个三个指针，要么全设为NULL(恢复默认状态)，要么全部都非NULL。原因后面会说到。

这些内存分配函数的实现是相当简单。看看event_mm_malloc_：

```
void *
event_mm_malloc_(size_t sz)
{
    if (_mm_malloc_fn)
        return _mm_malloc_fn(sz);
    else
        return malloc(sz);
}
```

如果用户定制了内存分配函数(_mm_malloc_fn不为NULL)，那么就直接调用用户定制的内存分配函数。否则使用C语言标准库提供的。其他几个内存分配函数也是这样实现的。这里就不贴代码了。

定制自己的内存分配函数需要注意的一些地方：

- 替换内存管理函数影响libevent 随后的所有分配、调整大小和释放内存操作。所以必须保证在调用任何其他libevent函数之前进行定制。否则，Libevent可能用定制的free函数释放C语言 库的malloc函数分配的内存
- malloc和realloc函数返回的内存块应该具有和C库返回的内存块一样的地址对齐
- realloc函数应该正确处理realloc(NULL, sz)（也就是当作malloc(sz)处理）
- realloc函数应该正确处理realloc(ptr, 0)（也就是当作free(ptr)处理）
- 如果在多个线程中使用libevent，替代的内存管理函数需要是线程安全的
- 如果要释放由Libevent函数分配的内存，并且已经定制了malloc和realloc函数，那么就应该使用定制的free函数释放。否则将会C语言标准库的free函数释放定制内存分配函数分配的内存，这将发生错误。所以三者要么全部不定制，要么全部定制。

参考：http://www.wangafu.net/~nickm/libevent-book/Ref1_libsetup.html

5.多线程、锁、条件变量(一)

原文地址

Libevent提供给用户的可见多线程API都在thread.h文件中。在这个文件提供的API并不多。基本上都是一些定制函数，像前面几篇博文说到的，可以为Libevent定制用户自己的多线程函数。

开启多线程

Libevent默认是不开启多线程的，也没有锁、条件变量这些东西。这点和前面博客说到的"没有定制就用Libevent默认提供"，有所不同。只有当你调用了evthread_use_windows_threads()或者evthread_use_pthreads()或者调用evthread_set_lock_callbacks函数定制自己的多线程、锁、条件变量才会开启多线程功能。其实，前面的那两个函数其内部实现也是定制，在函数的内部，Libevent封装的一套Win32线程、pthreads线程。然后调用evthread_set_lock_callbacks函数，进行定制。

thread.h文件只提供了定制线程的接口，并没有提供使用线程接口。这点很像前面说到的Libevent日志和内存分配。其实这也很好理解。因为都是你提供定制的线程函数。你都能提供了，你肯定有办法使用，没必要要Libevent提供一些API给你使用。

如果用户为libevent开启了多线程，那么libevent里面的函数就会变成线程安全的。此时主线程在使用event_base_dispatch，别的线程是可以线程安全地使用event_add把一个event添加到主线程的event_base中。具体的工作原理可以参考《evthread_notify_base通知主线程》。

锁和条件变量结构体

Libevent允许用户定制自己的锁和条件变量。其实现原理和前面说到的日志和内存分配一样，都是内部有一个全局变量。定制自己的锁和条件变量，就是对这个全局变量进行赋值。

锁结构：

```
//thread.h文件
struct evthread_lock_callbacks {
    //版本号，设置为宏EVTHREAD_LOCK_API_VERSION
    int lock_api_version;
    //支持的锁类型，有普通锁，递归锁，读写锁三种
    unsigned supported_locktypes;

    //分配一个锁变量(指针类型)，因为不同的平台锁变量是不同的类型
    //所以用这个通用的void*类型
    void *(*alloc)(unsigned locktype);
    void (*free)(void *lock, unsigned locktype);
    int (*lock)(unsigned mode, void *lock);
    int (*unlock)(unsigned mode, void *lock);
};
```

目前Libevent支持的locktype(锁类型)有三种：

- 普通锁， 值为0
- 递归锁， 值为EVTHREAD_LOCKTYPE_RECURSIVE
- 读写锁， 值为EVTHREAD_LOCKTYPE_READWRITE

当用户定制了自己的线程锁后，就可以用alloc这个函数指针调用函数，获取一个锁变量指针（在支持pthreads的系统获得的是pthread_mutex_t类型指针）。参数locktype就是用户指定的锁类型。

参数mode(锁模式)则取下面的值：

- EVTHREAD_READ：仅用于读写锁：为读操作请求或者释放锁
- EVTHREAD_WRITE：仅用于读写锁：为写操作请求或者释放锁
- EVTHREAD_TRY：仅用于锁定：仅在可以立刻锁定的时候才请求锁定

虽然Libevent提供了这些锁类型和mode类型，但实际上是否支持这些类型完全是由所定制的线程锁决定的。Libevent提供的pthreads线程锁和WIN32线程锁就只支持其中的一部分。具体是哪些下面会说到。

条件变量结构：

```
//thread.h文件
struct evthread_condition_callbacks {
    //版本号，设置为EVTHREAD_CONDITION_API_VERSION宏
    int condition_api_version;

    void *(*alloc_condition)(unsigned condtype);

    void (*free_condition)(void *cond);
    int (*signal_condition)(void *cond, int broadcast);
    int (*wait_condition)(void *cond, void *lock,
        const struct timeval *timeout);
};
```

条件变量的版本为EVTHREAD_CONDITION_API_VERSION时，alloc_condition的参数取0。奇怪的是，Libevent并没有提供其他的版本号。前面的线程锁也是只提供给了一个版本号。

signal_condition的第一个参数为alloc_condition的返回值，第二个参数指明唤醒多少个等待的线程。当broadcast取1时，唤醒所有的线程。取其他值时，只唤醒其中一个线程。

wait_condition的第二个参数为前面线程锁evthread_lock_callbacks结构中的alloc指针函数的返回值。熟悉条件变量的读者，这点还是比较容易懂的。对于第三个参数，和pthread_cond_timedwait有所不同。pthread_cond_timedwait的时间是绝对时间，**这里的timeout则是等待的时间，所以千万不要用一个绝对时间作为参数值，不然等到老都等不到超时。**如果该参数为NULL，那么就没有超时，将死等下去，直到另外的线程调用了signal_condition。

Libevent封装的多线程

说了这么多，其实对于用户来说，如果想让Libevent支持多线程。Windows用户直接调用evthread_use_windows_threads()，遵循pthreads线程的系统直接调用evthread_use_pthreads()就可以了。其他什么东西都不需要做了。**还有一点要注意的是，这两个函数要在代码的一开始就调用，必须在event_base_new函数之前调用**。好了，现在还是研究代码吧。

下面看一下evthread_use_pthreads()函数。

```
//evthread_pthreads.c文件
int
evthread_use_pthreads(void)
{

    //结构体中做一些函数指针作为参数。这些函数都是定义在evthread_pthread.c文件中
    struct evthread_lock_callbacks cbs = {
        EVTHREAD_LOCK_API_VERSION,
        EVTHREAD_LOCKTYPE_RECURSIVE,
        evthread_posix_lock_alloc,//函数指针
        evthread_posix_lock_free,//函数指针
        evthread_posix_lock,//函数指针
        evthread_posix_unlock//函数指针
    };
    struct evthread_condition_callbacks cond_cbs = {
        EVTHREAD_CONDITION_API_VERSION,
        evthread_posix_cond_alloc,
        evthread_posix_cond_free,
        evthread_posix_cond_signal,
        evthread_posix_cond_wait
    };
    /* Set ourselves up to get recursive locks. */
    if (pthread_mutexattr_init(&attr_recursive))
        return -1;
    if (pthread_mutexattr_settype(&attr_recursive, PTHREAD_MUTEX_RECURSIV
E))
        return -1;

    evthread_set_lock_callbacks(&cbs); //定制锁操作
    evthread_set_condition_callbacks(&cond_cbs); //定制条件变量操作
    evthread_set_id_callback(evthread_posix_get_id); //设置可以获取线程ID的回
调函数
    return 0;
}
```

函数一开始就定义并初始化了一个evthread_lock_callbacks结构和一个evthread_condition_callbacks结构。然后用之去进行定制。

代码中的attr_recursive是一个锁属性pthread_mutexattr_t类型的全局变量。在这个函数中，它被设置成具有递归属性。在申请锁时，可以看到其作用。

```
static void *
evthread_posix_lock_alloc(unsigned locktype)
{
    pthread_mutexattr_t *attr = NULL;
    pthread_mutex_t *lock = mm_malloc(sizeof(pthread_mutex_t));
    if (!lock)
        return NULL;
    if (locktype & EVTHREAD_LOCKTYPE_RECURSIVE)
        attr = &attr_recursive;
    if (pthread_mutex_init(lock, attr)) {
        mm_free(lock);
        return NULL;
    }
    return lock;
}
```

可以看到这个全局变量是用于设置递归锁的。从这个函数可以看到，Libevent提供的pthreads版本锁只支持递归锁和普通非递归锁，并不支持读写锁。当然你可以提供一套支持读写锁的锁操作。阅读Libevent提供的WIN32版本锁代码，也可以看到并不支持读写锁。值得注意的是，WIN32的锁默认是具有递归功能的，无需用EVTHREAD_LOCKTYPE_RECURSIVE作参数值。

前面还说到Libevent提供了EVTHREAD_READ、EVTHREAD_READ、EVTHREAD_TRY三种锁模式(mode)。但在Libevent提供的pthreads版本锁中，只在evthread_posix_lock函数中使用到这些宏。

```
static int
evthread_posix_lock(unsigned mode, void *_lock)
{
    pthread_mutex_t *lock = _lock;
    if (mode & EVTHREAD_TRY)
        return pthread_mutex_trylock(lock);
    else
        return pthread_mutex_lock(lock);
}
```

可以看到，它仅仅支持EVTHREAD_TRY这个锁模式。WIN32版本也是如此。

条件变量也简单地对系统native的条件进行一些简单的封装。这里就不多说了。在Windows中，因为在Windows Vista之前的Windows 操作系统并不支持提供条件变量，此时Libevent就使用Windows提供的EVENT进行一些封装来实现条件变量的功能。如果所在的Windows系统支持条件变量，Libevent将优先使用Windows本身提供的条件变量。这点可以在evthread_use_windows_threads函数看到。

```
//evthread_win32.c文件
int
evthread_use_windows_threads(void)
{
    struct evthread_lock_callbacks cbs = {
        EVTHREAD_LOCK_API_VERSION,
        EVTHREAD_LOCKTYPE_RECURSIVE,
        evthread_win32_lock_create,
        evthread_win32_lock_free,
        evthread_win32_lock,
        evthread_win32_unlock
    };


    struct evthread_condition_callbacks cond_cbs = {
        EVTHREAD_CONDITION_API_VERSION,
        evthread_win32_cond_alloc,
        evthread_win32_cond_free,
        evthread_win32_cond_signal,
        evthread_win32_cond_wait
    };
#ifdef WIN32_HAVE_CONDITION_VARIABLES //有内置的条件变量功能
    struct evthread_condition_callbacks condvar_cbs = {
        EVTHREAD_CONDITION_API_VERSION,
        evthread_win32_condvar_alloc,
        evthread_win32_condvar_free,
        evthread_win32_condvar_signal,
        evthread_win32_condvar_wait
    };
#endif

    evthread_set_lock_callbacks(&cbs);
    evthread_set_id_callback(evthread_win32_get_id);

//优先使用Windows自身提供的条件变量
#ifdef WIN32_HAVE_CONDITION_VARIABLES
    if (evthread_win32_condvar_init()) {
        evthread_set_condition_callbacks(&condvar_cbs);
        return 0;
    }
#endif
    evthread_set_condition_callbacks(&cond_cbs);

    return 0;
}
```

一旦用户调用evthread_use_windows_threads()或者evthread_use_pthreads()函数，那么用户就为Libevent定制了自己的线程锁操作。Libevent的其他代码中，如果需要用到锁，就会去调用这些线程锁操作。在实现上，当调用evthread_use_windows_threads()或者evthread_use_pthreads()函数时，两个函数的内部都会调用evthread_set_lock_callbacks函数。而这个设置函数会把前面两个evthread_use_xxx函数中定义的cbs变量值复制到一个evthread_lock_callbacks类型的_evthread_lock_fns全局变量保存起来。以后，Libevent需要用到多线程锁操作，直接访问这个_evthread_lock_fn变量即可。对于条件变量，也是用这样方式实现的。

定制的顺序

前面的一些博文和这篇都说到了用户可以定制自己的操作，比如内存分配、日志记录、线程锁。这些定制都应该放在代码的最前面，即不能在使用Libevent的event、event_base这些结构体之后。因为这些结构体会使用到内存分配、日志记录、线程锁的。而这三者的定制顺序应该是：**内存分配->日志记录->线程锁**。

参考：http://www.wangafu.net/~nickm/libevent-book/Ref1_libsetup.html

6.多线程、锁、条件变量(二)

原文地址

Debug锁操作

Libevent还支持对锁操作的一些检测，进而捕抓一些典型的锁错误。Libevent检查：

- 解锁自己(线程)没有持有的锁
- 在未解锁前，自己(线程)再次锁定一个非递归锁。

Libevent通过一些变量记录锁的使用情况，当检查到这些锁的错误使用时，就调用abort，退出运行。

开启调试功能

用户只需在调用evthread_use_pthreads或者evthread_use_windows_threads之后，调用evthread_enable_lock_debuging()函数即可开启调试锁的功能。该函数有一个拼写错误。在2.1.2-alpha版本中会改正为evthread_enable_lock_debugging，为了后向兼容，两者都会支持的。

现在看一下Libevent是锁调试功能。

```
//evthread.c文件
void
evthread_enable_lock_debuging(void)
{
    struct evthread_lock_callbacks cbs = {
        EVTHREAD_LOCK_API_VERSION,
        EVTHREAD_LOCKTYPE_RECURSIVE,
        debug_lock_alloc,
        debug_lock_free,
        debug_lock_lock,
        debug_lock_unlock
    };
    if (_evthread_lock_debugging_enabled)
        return;

    //把当前用户定制的锁操作复制到_original_lock_fns结构体变量中。
    memcpy(&_original_lock_fns, &_evthread_lock_fns,
        sizeof(struct evthread_lock_callbacks));

    //将当前的锁操作设置成调试锁操作。但调试锁操作函数内部
    //还是使用_original_lock_fns的锁操作函数
    memcpy(&_evthread_lock_fns, &cbs,
        sizeof(struct evthread_lock_callbacks));

    memcpy(&_original_cond_fns, &_evthread_cond_fns,
        sizeof(struct evthread_condition_callbacks));
    _evthread_cond_fns.wait_condition = debug_cond_wait;
    _evthread_lock_debugging_enabled = 1;

    /* XXX return value should get checked. */
    event_global_setup_locks_(0);
}
```

在上面代码的注释可以知道，虽然evthread_lock_fns的值被更新为debug_lock_alloc、debug_lock_lock和debug_lock_unlock。但实际上，使用的还是之前用户定制的线程锁操作函数，只是加多了一层抽象而已。如果看不懂这段话，可以看下面的代码，看完已经就会懂的了。

```
//evthread.c文件
static void *
debug_lock_alloc(unsigned locktype)
{
    struct debug_lock *result = mm_malloc(sizeof(struct debug_lock));
    if (!result)
        return NULL;

    //用户设置过自己的线程锁函数
    if (_original_lock_fns.alloc) {
        //用用户定制的线程锁函数分配一个线程锁
        if (!(result->lock = _original_lock_fns.alloc(
                locktype|EVTHREAD_LOCKTYPE_RECURSIVE))) {
            mm_free(result);
            return NULL;
        }
    } else {
        result->lock = NULL;
    }
    result->locktype = locktype;
    result->count = 0;
    result->held_by = 0;
    return result;
}
```

现在看看Libevent是怎么调试（更准确来说，应该是检测）锁的。锁的检测，需要用到debug_lock结构体，它对锁的一些使用状态进行了记录。

debug递归锁

```c
//evthread.c文件
struct debug_lock {
    unsigned locktype; //锁的类型
    unsigned long held_by; //这个锁是被哪个线程所拥有
    /* XXXX if we ever use read-write locks, we will need a separate
     * lock to protect count. */
    int count; //这个锁的加锁次数
    void *lock; //锁类型，在pthreads下为pthread_mutex_t*类型
};

static int
debug_lock_lock(unsigned mode, void *lock_)
{
    struct debug_lock *lock = lock_;
    int res = 0;
    if (lock->locktype & EVTHREAD_LOCKTYPE_READWRITE)
        EVUTIL_ASSERT(mode & (EVTHREAD_READ|EVTHREAD_WRITE));
    else
        EVUTIL_ASSERT((mode & (EVTHREAD_READ|EVTHREAD_WRITE)) == 0);
    if (_original_lock_fns.lock)
        res = _original_lock_fns.lock(mode, lock->lock);
    //lock 成功返回0，失败返回非0
    if (!res) {
        //记录这个锁的使用情况。
        evthread_debug_lock_mark_locked(mode, lock);
    }
    return res;
}

static void
evthread_debug_lock_mark_locked(unsigned mode, struct debug_lock *lock)
{
    ++lock->count; //增加锁的加锁次数.解锁时会减一
    if (!(lock->locktype & EVTHREAD_LOCKTYPE_RECURSIVE))
        EVUTIL_ASSERT(lock->count == 1);
    if (_evthread_id_fn) {
        unsigned long me;
        me = _evthread_id_fn(); //获取线程ID
        if (lock->count > 1)
            EVUTIL_ASSERT(lock->held_by == me);
        lock->held_by = me; //记录这个锁是被哪个线程所拥有
    }
}
```

这里主要是测试一个锁类型(如pthread_mutex_t)同时被加锁的次数。如果是一个非递归锁，那么将不允许多次锁定。对于锁的实现没有bug的话，如果是非递归锁，那么会在第二次锁住同一个锁时，卡死在debug_lock_lock 函数的original_lock_fns.lock上(即发生了死锁)。此时evthread_debug_lock_mark_locked是不会被调用的。但是，对于一个有bug的锁实现，那么就有可能发生这种情况。即对于非递归锁，其还是可以多次锁住同一个锁，并且不会发生死锁。此时，evthread_debug_lock_mark_locked函数将会被执行，在这个函数内部将会检测这种情况。Libevent的锁调试(检测)就是调试(检测)这种有bug的锁实现。

debug解锁

现在看一下解锁时的检测。这主要是检测解锁一个自己没有锁定的锁，比如锁是由线程A锁定的，但线程B却去解锁。

```
//evthread.c文件。
static int
debug_lock_unlock(unsigned mode, void *lock_)
{
    struct debug_lock *lock = lock_;
    int res = 0;
    //先检测
    evthread_debug_lock_mark_unlocked(mode, lock);
    if (_original_lock_fns.unlock)
        res = _original_lock_fns.unlock(mode, lock->lock);
    return res;
}

static void
evthread_debug_lock_mark_unlocked(unsigned mode, struct debug_lock *lock)

{
    if (lock->locktype & EVTHREAD_LOCKTYPE_READWRITE)
        EVUTIL_ASSERT(mode & (EVTHREAD_READ|EVTHREAD_WRITE));
    else
        EVUTIL_ASSERT((mode & (EVTHREAD_READ|EVTHREAD_WRITE)) == 0);
    if (_evthread_id_fn) {
        //检测锁的拥有者是否为要解锁的线程
        EVUTIL_ASSERT(lock->held_by == _evthread_id_fn());
        if (lock->count == 1)
            lock->held_by = 0;
    }
    --lock->count;//减少被加锁次数
    EVUTIL_ASSERT(lock->count >= 0);
}
```

从代码中可以看到，这里主要是检测解锁的线程是否为锁的实际拥有者。即检测是否解锁一个自己不拥有的锁。这里不是为了检测锁的实现是否有bug，而是检测锁在使用的时候是否有bug。

当然Libevent提供的检测能力还是很有限的。特别是对于前一个检测，如果是使用Windows线程锁或者pthreads线程锁，这个检测并没有什么用。毕竟这些锁的实现已经经过了千锤百炼。

定制线程锁、条件变量

现在来看一下线程锁定制函数evthread_set_lock_callbacks。本来这个定制应该放在前一篇博客讲的。但由于其实现用到了调试锁的一些内容，所以就放到这里讲。

```
//evthread.h文件
int
evthread_set_lock_callbacks(const struct evthread_lock_callbacks *cbs)
{
    struct evthread_lock_callbacks *target =
        _evthread_lock_debugging_enabled //默认为0
        ? &_original_lock_fns : &_evthread_lock_fns;

    if (!cbs) {//参数为NULL，取消线程锁功能
        if (target->alloc)
            event_warnx("Trying to disable lock functions after "
                "they have been set up will probaby not work.");
        memset(target, 0, sizeof(_evthread_lock_fns));
        return 0;
    }

    //一旦设置就不能修改
    if (target->alloc) {
        /* Uh oh; we already had locking callbacks set up.*/
        if (target->lock_api_version == cbs->lock_api_version &&
            target->supported_locktypes == cbs->supported_locktypes &&
            target->alloc == cbs->alloc &&
            target->free == cbs->free &&
            target->lock == cbs->lock &&
            target->unlock == cbs->unlock) {
            /* no change -- allow this. */
            return 0;
        }
        event_warnx("Can't change lock callbacks once they have been "
            "initialized.");
        return -1;
    }

    //这个四个函数指针都不为NULL时才能成功定制。因为这四个函数是配套使用的
    if (cbs->alloc && cbs->free && cbs->lock && cbs->unlock) {
        memcpy(target, cbs, sizeof(_evthread_lock_fns));
        return event_global_setup_locks_(1);
    } else {
        return -1;
    }
}
```

全局变量_evthread_lock_debugging_enabled的初始化值为0，当调用evthread_enable_lock_debuging函数后其值为1。于是，无论是在调试锁还是非调试的情况下，target变量都能够修改实际使用的evthread_lock_callbacks结构体（线程锁操作函数指针结构体）。前面已经说到了，在非调试情况下，实际使用的是_evthread_lock_fns变量的线程锁函数指针成员。在调试情况下实际使用的是_original_lock_fns变量的。

从上面的代码中也可以看到：当参数为NULL时，就等于取消了线程锁功能。此后，Libevent的代码将运行在没有线程锁的无线程安全状态下。

上面的第二个if语句则说明，在已经定制了线程锁之后，是无法再次定制的。我觉得这主要是怕：这个修改线程锁的动作刚好发生在另外一个线程获取获取锁的之后，即调用lock函数之后。并且是在另外的线程释放锁之前，即调用unlock函数之前。如果允许修改锁定制的线程锁，那么将可能发生，加锁和解锁操作是完全不同的两套线程锁。

前面说到参数cbs可以为NULL，其实这就给了我们一个修改定制线程锁的方法。我们可以先用NULL作为参数调用一次evthread_set_lock_callbacks函数，然后用真正的线程锁方案作为参数，再次调用evthread_set_lock_callbacks函数。当然这相当容易发生bug。后面也会给出一个例子。

锁的使用

加锁和解锁

Libevent中，一些函数支持多线程。一般都是使用锁进行线程同步。在Libevent的代码中，一般是使用EVTHREAD_ALLOC_LOCK宏获取一个锁变量，EVBASE_ACQUIRE_LOCK宏进行加锁，EVBASE_RELEASE_LOCK宏进行解锁。在阅读Libevent源代码中，一般都只会看到EVBASE_ACQUIRE_LOCK和EVBASE_RELEASE_LOCK。锁的内部实现是看不见的。

现在对EVBASE_ACQUIRE_LOCK进行深究，看其是怎么一层层地封装的。先看event_add函数的实现：

```c
//event.c文件
int
event_add(struct event *ev, const struct timeval *tv)
{
    int res;

    if (EVUTIL_FAILURE_CHECK(!ev->ev_base)) {
        event_warnx("%s: event has no event_base set.", __func__);
        return -1;
    }
    //加锁
    EVBASE_ACQUIRE_LOCK(ev->ev_base, th_base_lock);

    res = event_add_internal(ev, tv, 0);
    //解锁
    EVBASE_RELEASE_LOCK(ev->ev_base, th_base_lock);

    return (res);
}
```

其中，EVBASE_ACQUIRE_LOCK是一个条件宏。

```
//evthread-internal.h文件
#ifndef WIN32
#define EVTHREAD_EXPOSE_STRUCTS
#endif


#if ! defined(_EVENT_DISABLE_THREAD_SUPPORT) && defined(EVTHREAD_EXPOSE_S
TRUCTS)

#define EVLOCK_LOCK(lockvar,mode)                    \
    do {                                    \
        if (lockvar)                            \
            _evthread_lock_fns.lock(mode, lockvar);     \
    } while (0)

#define EVBASE_ACQUIRE_LOCK(base, lockvar) do {             \
        EVLOCK_LOCK((base)->lockvar, 0);         \
    } while (0)


#elif ! defined(_EVENT_DISABLE_THREAD_SUPPORT)

int _evthreadimpl_lock_lock(unsigned mode, void *lock);


#define EVLOCK_LOCK(lockvar,mode)                    \
    do {                                    \
        if (lockvar)                            \
            _evthreadimpl_lock_lock(mode, lockvar);     \
    } while (0)

#define EVBASE_ACQUIRE_LOCK(base, lockvar) do {             \
        EVLOCK_LOCK((base)->lockvar, 0);         \
    } while (0)


#else //不支持多线程
#define EVBASE_ACQUIRE_LOCK(base, lock)  ((void)0)

#endif


//evthread.c文件
int
_evthreadimpl_lock_lock(unsigned mode, void *lock)
{
    if (_evthread_lock_fns.lock)
        return _evthread_lock_fns.lock(mode, lock);
    else
        return 0;
}
```

虽然是条件宏，但最终都是调用了_evthread_lock_fns结构体中的lock指针指向的函数，即调用了定制锁的锁函数，进行了锁定。但不同的是，在第一种宏中，并没有对_evthread_lock_fns.lock这个指针作是否为NULL判断，而第二种宏，会在_evthreadimpl_lock_lock对这个指针进行判断，当这个指针不为NULL时才进行函数调用。

在非Windows系统上会把EVBASE_ACQUIRE_LOCK宏定义成第一种情况。但在Linux上调用event_add时，即使_evthread_lock_fns.lock为NULL也没有出现段错误。

实际上，虽然第一种情况没有对_evthread_lock_fns.lock进行判断，但它对lockvar进行了判断。但Lockvar为何物？顺藤摸瓜，lockvar为event_base结构体中的th_base_lock成员，类型为viod*。实际上，lockvar就是申请得到的锁变量。下面代码将看到如何申请。如果th_base_lock为NULL，那么就不会对_evthread_lock_fns.lock这个函数指针进行函数调用了。

在event_base_new_with_config函数可以看到th_base_lock成员的赋值情况。

```
struct event_base *
event_base_new_with_config(const struct event_config *cfg)
{
    struct event_base *base;

    //之所以不用mm_malloc是因为mm_malloc并不会清零该内存区域。
    //而这个函数是会清零申请到的内存区域。这相当于给base初始化
    if ((base = mm_calloc(1, sizeof(struct event_base))) == NULL) {
        event_warn("%s: calloc", __func__);
        return NULL;
    }

        …….//其他成员的初始化

#ifndef _EVENT_DISABLE_THREAD_SUPPORT

    //对于th_base_lock变量，目前的值为NULL.
   //EVTHREAD_LOCKING_ENABLED宏是测试_evthread_lock_fns.lock
   //是否不为NULL
    if (EVTHREAD_LOCKING_ENABLED() &&
        (!cfg || !(cfg->flags & EVENT_BASE_FLAG_NOLOCK))) {
        int r;
        EVTHREAD_ALLOC_LOCK(base->th_base_lock, //申请锁变量
            EVTHREAD_LOCKTYPE_RECURSIVE);
    }
#endif

    …..
    return (base);
}
```

从这里可以看到，如果_evthread_lock_fns.lock为NULL，那么th_base_lock成员肯定为NULL，那么后面就不会调用_evthread_lock_fns.lock()函数。从而避过段错误。

会不会th_base_lock不为NULL，而_evthread_lock_fns.lock为NULL呢？

th_base_lock是由要_evthread_lock_fns.lock非NULL，才会被赋值为非NULL。如果_evthread_lock_fns.lock为NULL，那么th_base_lock就肯定为NULL了。此外，结构体event_base是定义是event_internal.h文件的。所以，正常情况下，该结构体的成员是不可见的。所以你是无法直接访问并修改其成员。

其实，有一种可能达到目标。就是先把_evthread_lock_fns.lock赋值成非NULL，然后用来把th_base_lock赋值成非NULL，之后把_evthread_lock_fns.lock修改为NULL。下面是Libevent提供的定制线程锁的函数evthread_set_lock_callbacks。

```
int
evthread_set_lock_callbacks(const struct evthread_lock_callbacks *cbs)
{
    …..
    if (!cbs) {//参数为NULL，取消线程锁功能
        if (target->alloc)
            event_warnx("Trying to disable lock functions after "
                "they have been set up will probaby not work.");
        memset(target, 0, sizeof(_evthread_lock_fns));
        return 0;
    }
    …..
}
```

从代码中可以看到，当参数cbs为NULL时，是可以取消线程锁功能的。可以尝试编译运行下面的代码。代码一运行就可以看到段错误了。

```
#include  <event.h>
#include  <thread.h>
#include  <unistd.h>

void cmd_cb(int fd, short event, void *arg)
{

}

int main()
{
    evthread_use_pthreads();

    event_base *base = event_base_new();

    evthread_set_lock_callbacks(NULL);

    event *cmd_event = event_new(base, STDIN_FILENO, EV_READ | EV_PERSIS
T,
                                 cmd_cb, base);
    event_add(cmd_event, NULL);

    event_base_dispatch(base);

    return 0;
}
```

从这里可以看到，一旦设置了线程、锁函数，那么就不应该对其进行修改。

值得注意的是，在Libevent中，像EVBASE_ACQUIRE_LOCK这个宏是专门给event_base用的。

断言已加锁

在Libevent中，很多线程安全的函数都会调用一个已加锁断言。确保在进入这函数的时候，已经获得了一个锁。一般是调用EVENT_BASE_ASSERT_LOCKED(base);完成这个断言。要注意的是：**这个已锁断言要在开启了调试锁的前提下，才能使用的。**

下面代码可以看到断言锁是怎么实现的：

```
EVENT_BASE_ASSERT_LOCKED(base);

#define EVENT_BASE_ASSERT_LOCKED(base)      \
    EVLOCK_ASSERT_LOCKED((base)->th_base_lock)


#define EVLOCK_ASSERT_LOCKED(lock)                  \
    do {                                    \
        if ((lock) && _evthread_lock_debugging_enabled) {   \
            EVUTIL_ASSERT(_evthread_is_debug_lock_held(lock)); \
        }                               \
    } while (0)


int
_evthread_is_debug_lock_held(void *lock_)
{
    struct debug_lock *lock = lock_;
    if (! lock->count)
        return 0;
    if (_evthread_id_fn) {
        unsigned long me = _evthread_id_fn();
        if (lock->held_by != me)
            return 0;
    }
    return 1;
}
```

从EVLOCK_ASSERT_LOCKED宏的判断可以知道，_evthread_lock_debugging_enabled要不为0。而它的赋值是由evthread_enable_lock_debuging()完成的，这个函数的作用就是开启锁调试功能。

前面在讲调试锁的时候，有说到evthread_debug_lock_mark_locked函数，这个函数在加锁的时候会被调用。该函数会记录锁是由哪个线程加的，具体实现是通过记录线程ID。面的_evthread_is_debug_lock_held函数的功能就是测试本线程ID是否等于之前加锁的线程ID。这样就完成了已加锁断言。

参考：http://www.wangafu.net/~nickm/libevent-book/Ref1_libsetup.html

7. TAILQ_QUEUE队列

原文地址

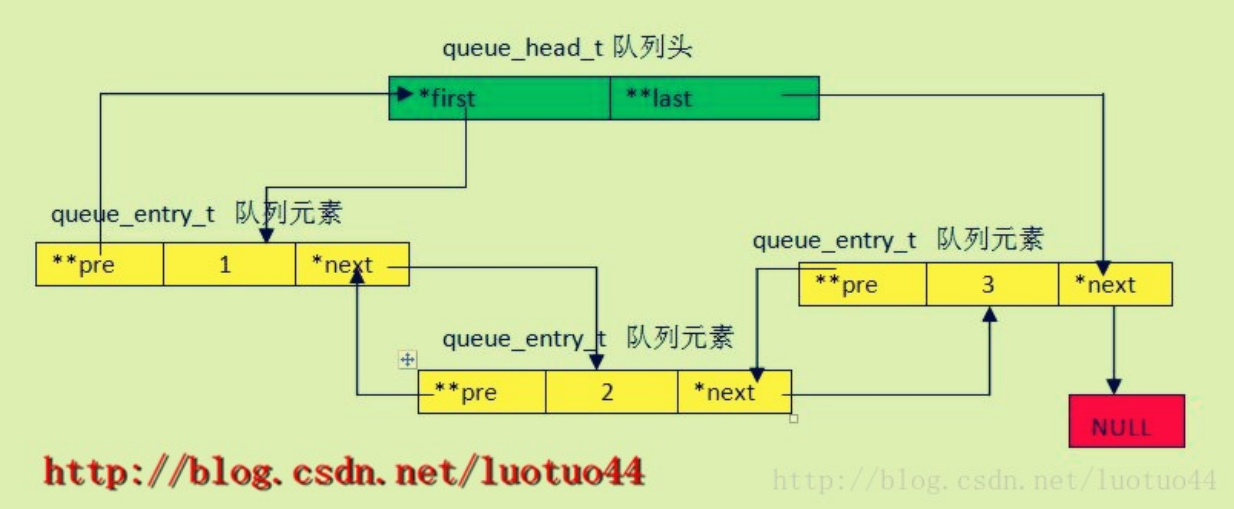
Libevent源码中有一个queue.h文件，位于compat/sys目录下。该文件里面定义了5个数据结构，其中TAILQ_QUEUE是使得最广泛的。本文就说一下这个数据结构。

队列结构体

TAILQ_QUEUE由下面两个结构体一起配合工作。

```
#define TAILQ_HEAD(name, type)                      \
struct name {                                       \
    struct type *tqh_first; /* first element */         \
    struct type **tqh_last; /* addr of last next element */     \
}

//和前面的TAILQ_HEAD不同，这里的结构体并没有name.即没有结构体名。
//所以该结构体只能作为一个匿名结构体。所以，它一般都是另外一个结构体
//或者共用体的成员
#define TAILQ_ENTRY(type)                           \
struct {                                        \
    struct type *tqe_next;  /* next element */          \
    struct type **tqe_prev; /* address of previous next element */  \
}
```

由这两个结构体配合构造出来的队列一般如下图所示：



图中，一级指针指向的是queue_entry_t这个结构体，即存储queue_entry_t这个结构体的地址值。二级指针存储的是一级地址变量的地址值。所以二级指针指向的是图中的一级指针，而非结构体。图中的1,2, 3为队列元素保存的一些值。

队列操作宏函数以及使用例子

除了这两个结构体，在queue.h文件中，还为TAILQ_QUEUE定义了一系列的访问和操作函数。很不幸，它们是一些宏定义。这里就简单贴几个函数（准确来说，不是函数）的代码。

```
#define TAILQ_FIRST(head)       ((head)->tqh_first)

#define TAILQ_NEXT(elm, field)      ((elm)->field.tqe_next)

#define TAILQ_INIT(head) do {                       \
    (head)->tqh_first = NULL;                   \
    (head)->tqh_last = &(head)->tqh_first;              \
} while (0)

#define TAILQ_INSERT_TAIL(head, elm, field) do {            \
    (elm)->field.tqe_next = NULL;                   \
    (elm)->field.tqe_prev = (head)->tqh_last;           \
    *(head)->tqh_last = (elm);                  \
    (head)->tqh_last = &(elm)->field.tqe_next;          \
} while (0)

#define TAILQ_REMOVE(head, elm, field) do {             \
    if (((elm)->field.tqe_next) != NULL)            \
        (elm)->field.tqe_next->field.tqe_prev =         \
            (elm)->field.tqe_prev;              \
    else                                \
        (head)->tqh_last = (elm)->field.tqe_prev;       \
    *(elm)->field.tqe_prev = (elm)->field.tqe_next;         \
} while (0)
```

这些宏是很难看的，也没必要直接去看这些宏。下面来看一个使用例子。有例子更容易理解。

```c
//队列中的元素结构体。它有一个值，并且有前向指针和后向指针
//通过前后像指针，把队列中的节点(元素)连接起来
struct queue_entry_t
{
    int value;

    //从TAILQ_ENTRY的定义可知，它只能是结构体或者共用体的成员变量
    TAILQ_ENTRY(queue_entry_t)entry;
};

//定义一个结构体，结构体名为queue_head_t，成员变量类型为queue_entry_t
//就像有头节点的链表那样，这个是队列头。它有两个指针，分别指向队列的头和尾
TAILQ_HEAD(queue_head_t, queue_entry_t);

int main(int argc, char **argv)
{
    struct queue_head_t queue_head;
    struct queue_entry_t *q, *p, *s, *new_item;
    int i;

    TAILQ_INIT(&queue_head);

    for(i = 0; i < 3; ++i)
    {
        p = (struct queue_entry_t*)malloc(sizeof(struct queue_entry_t));

        p->value = i;

        //第三个参数entry的写法很怪，居然是一个成员变量名(field)
        TAILQ_INSERT_TAIL(&queue_head, p, entry);//在队尾插入数据
    }

    q = (struct queue_entry_t*)malloc(sizeof(struct queue_entry_t));
    q->value = 10;
    TAILQ_INSERT_HEAD(&queue_head, q, entry);//在队头插入数据

    //现在q指向队头元素、p指向队尾元素

    s = (struct queue_entry_t*)malloc(sizeof(struct queue_entry_t));
    s->value = 20;
    //在队头元素q的后面插入元素
    TAILQ_INSERT_AFTER(&queue_head, q, s, entry);


    s = (struct queue_entry_t*)malloc(sizeof(struct queue_entry_t));
    s->value = 30;
    //在队尾元素p的前面插入元素
    TAILQ_INSERT_BEFORE(p, s, entry);

    //现在进行输出
    //获取第一个元素
```

```
    s = TAILQ_FIRST(&queue_head);
    printf("the first entry is %d\n", s->value);

    //获取下一个元素
    s = TAILQ_NEXT(s, entry);
    printf("the second entry is %d\n\n", s->value);

    //删除第二个元素，但并没有释放s指向元素的内存
    TAILQ_REMOVE(&queue_head, s, entry);
    free(s);


    new_item = (struct queue_entry_t*)malloc(sizeof(struct
queue_entry_t));
    new_item->value = 100;

    s = TAILQ_FIRST(&queue_head);
    //用new_iten替换第一个元素
    TAILQ_REPLACE(&queue_head, s, new_item, entry);


    printf("now, print again\n");
    i = 0;
    TAILQ_FOREACH(p, &queue_head, entry)//用foreach遍历所有元素
    {
        printf("the %dth entry is %d\n", i, p->value);
    }

    p = TAILQ_LAST(&queue_head, queue_head_t);
    printf("last is %d\n", p->value);

    p = TAILQ_PREV(p, queue_head_t, entry);
    printf("the entry before last is %d\n", p->value);
}
```

例子并不难看懂。这里就不多讲了。

展开宏函数

下面把这些宏翻译一下(即展开)，显示出它们的本来面貌。这当然不是用人工方式去翻译。而是用gcc 的-E选项。

阅读代码时要注意，tqe_prev和tqh_last都是二级指针，行为会有点难理解。平常我们接触到的双向链表，next和prev成员都是一级指针。对于像链表A->B->C（把它们想象成双向链表），通常B的prev指向A这个结构体本身。此时，B->prev->next指向了本身。但队列Libevent的TAILQ_QUEUE，B的prev是一个二级指向，它指向的是A结构体的next成员。此时，*B->prev就指向了本身。当然，这并不能说用二级指针就方便。我觉得用二级指针理解起来更难，编写代码更容易出错。

```c
//队列中的元素结构体。它有一个值，并且有前向指针和后向指针
//通过前后像指针，把队列中的节点连接起来
struct queue_entry_t
{
    int value;

    struct
    {
        struct queue_entry_t *tqe_next;
        struct queue_entry_t **tqe_prev;
    }entry;
};

//就像有头节点的链表那样，这个是队列头。它有两个指针，分别指向队列的头和尾
struct queue_head_t
{
    struct queue_entry_t *tqh_first;
    struct queue_entry_t **tqh_last;
};

int main(int argc, char **argv)
{
    struct queue_head_t queue_head;
    struct queue_entry_t *q, *p, *s, *new_item;
    int i;

    //TAILQ_INIT(&queue_head);
    do
    {
        (&queue_head)->tqh_first = 0;
        //tqh_last是二级指针，这里指向一级指针
        (&queue_head)->tqh_last = &(&queue_head)->tqh_first;
    }while(0);

    for(i = 0; i < 3; ++i)
    {
        p = (struct queue_entry_t*)malloc(sizeof(struct queue_entry_t));

        p->value = i;

        //TAILQ_INSERT_TAIL(&queue_head, p, entry);在队尾插入数据
        do
        {
            (p)->entry.tqe_next = 0;
            //tqh_last存储的是最后一个元素(队列节点)tqe_next成员
            //的地址。所以，tqe_prev指向了tqe_next。
            (p)->entry.tqe_prev = (&queue_head)->tqh_last;

            //tqh_last存储的是最后一个元素(队列节点)tqe_next成员
            //的地址，所以*(&queue_head)->tqh_last修改的是最后一个
            //元素的tqe_next成员的值，使得tqe_next指向*p(新的队列
```

```
            //节点)。
            *(&queue_head)->tqh_last = (p);
            //队头结构体（queue_head）的tqh_last成员保存新队列节点的
            //tqe_next成员的地址值。即让tqh_last指向tqe_next。
            (&queue_head)->tqh_last = &(p)->entry.tqe_next;
        }while(0);
    }

    q = (struct queue_entry_t*)malloc(sizeof(struct queue_entry_t));
    q->value = 10;

    //TAILQ_INSERT_HEAD(&queue_head, q, entry);在队头插入数据
    do {
        //queue_head队列中已经有节点(元素了)。要对第一个元素进行修改
        if(((q)->entry.tqe_next = (&queue_head)->tqh_first) != 0)
            (&queue_head)->tqh_first->entry.tqe_prev = &(q)->entry.tqe_ne
xt;
        else//queue_head队列目前是空的，还没有任何节点（元素）。修改queue_head即
可
            (&queue_head)->tqh_last = &(q)->entry.tqe_next;

        //queue_head的first指针指向要插入的节点*q
        (&queue_head)->tqh_first = (q);
        (q)->entry.tqe_prev = &(&queue_head)->tqh_first;
    }while(0);

    //现在q指向队头元素、p指向队尾元素

    s = (struct queue_entry_t*)malloc(sizeof(struct queue_entry_t));
    s->value = 20;

    //TAILQ_INSERT_AFTER(&queue_head, q, s, entry);在队头元素q的后面插入元素

    do
    {
        //q不是最后队列中最后一个节点。要对q后面的元素进行修改
        if (((s)->entry.tqe_next = (q)->entry.tqe_next) != 0)
            (s)->entry.tqe_next->entry.tqe_prev = &(s)->entry.tqe_next;
        else//q是最后一个元素。对queue_head修改即可
            (&queue_head)->tqh_last = &(s)->entry.tqe_next;

        (q)->entry.tqe_next = (s);
        (s)->entry.tqe_prev = &(q)->entry.tqe_next;
    }while(0);



    s = (struct queue_entry_t*)malloc(sizeof(struct queue_entry_t));
    s->value = 30;

    //TAILQ_INSERT_BEFORE(p, s, entry); 在队尾元素p的前面插入元素
    do
```

```
    {
        //无需判断节点p前面是否还有元素。因为即使没有元素，queue_head的两个
        //指针从功能上也相当于一个元素。这点是采用二级指针的一大好处。
        (s)->entry.tqe_prev = (p)->entry.tqe_prev;
        (s)->entry.tqe_next = (p);
        *(p)->entry.tqe_prev = (s);
        (p)->entry.tqe_prev = &(s)->entry.tqe_next;
    }while(0);


    //现在进行输出

    // s = TAILQ_FIRST(&queue_head);
    s = ((&queue_head)->tqh_first);
    printf("the first entry is %d\n", s->value);

    // s = TAILQ_NEXT(s, entry);
    s = ((s)->entry.tqe_next);
    printf("the second entry is %d\n\n", s->value);

    //删除第二个元素，但并没有释放s指向元素的内存
    //TAILQ_REMOVE(&queue_head, s, entry);
    do
    {
        if (((s)->entry.tqe_next) != 0)
            (s)->entry.tqe_next->entry.tqe_prev = (s)->entry.tqe_prev;
        else (&queue_head)->tqh_last = (s)->entry.tqe_prev;

        *(s)->entry.tqe_prev = (s)->entry.tqe_next;
    }while(0);

    free(s);


    new_item = (struct queue_entry_t*)malloc(sizeof(struct
queue_entry_t));
    new_item->value = 100;

    //s = TAILQ_FIRST(&queue_head);
    s = ((&queue_head)->tqh_first);

    //用new_iten替换第一个元素
    //TAILQ_REPLACE(&queue_head, s, new_item, entry);
    do
    {
        if (((new_item)->entry.tqe_next = (s)->entry.tqe_next) != 0)
            (new_item)->entry.tqe_next->entry.tqe_prev = &(new_item)->ent
ry.tqe_next;
        else
            (&queue_head)->tqh_last = &(new_item)->entry.tqe_next;

        (new_item)->entry.tqe_prev = (s)->entry.tqe_prev;
```

```
        *(new_item)->entry.tqe_prev = (new_item);
    }while(0);


    printf("now, print again\n");
    i = 0;
    //TAILQ_FOREACH(p, &queue_head, entry)//用foreach遍历所有元素
    for((p) = ((&queue_head)->tqh_first); (p) != 0;
        (p) = ((p)->entry.tqe_next))
    {
        printf("the %dth entry is %d\n", i, p->value);
    }

    //p = TAILQ_LAST(&queue_head, queue_head_t);
    p = (*(((struct queue_head_t *)((&queue_head)->tqh_last))->tqh_las
t));
    printf("last is %d\n", p->value);


    //p = TAILQ_PREV(p, queue_head_t, entry);
    p = (*(((struct queue_head_t *)((p)->entry.tqe_prev))->tqh_last));
    printf("the entry before last is %d\n", p->value);
}
```

代码中有一些注释，不懂的可以看看。其实对于链表操作，别人用文字说再多都对自己理解帮助不大。只有自己动手一步步把链表操作都画出来，这样才能完全理解。

特殊指针操作

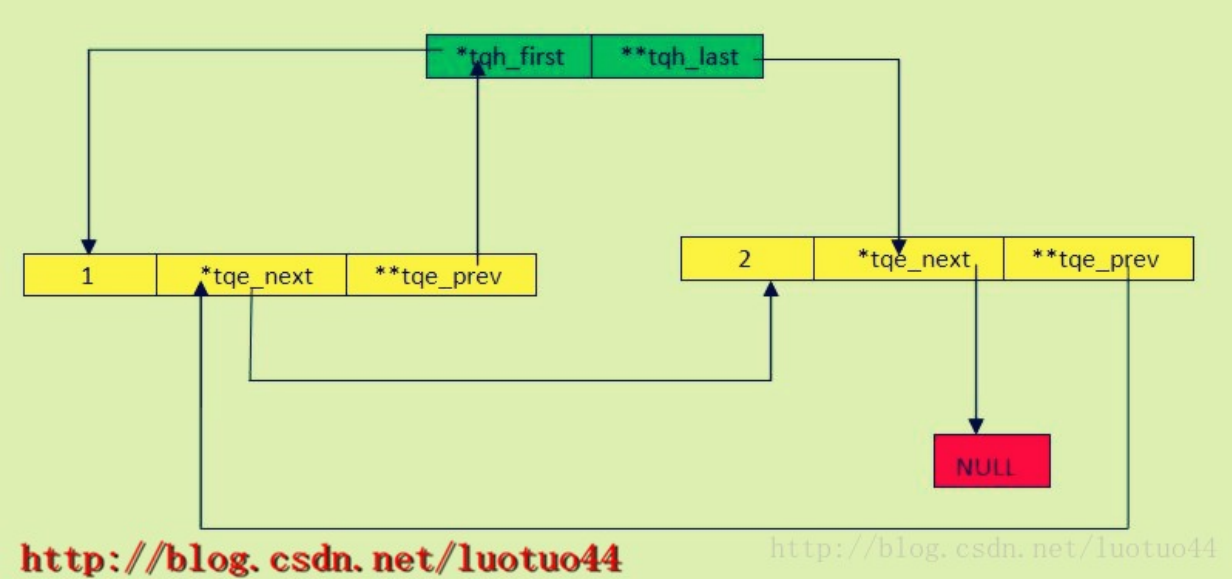
最后那两个操作宏函数有点难理解，现在来讲一下。在讲之前，先看一个关于C语言指针的例子。

```c
#include<stdio.h>

struct item_t
{
    int a;
    int b;
    int c;
};

struct entry_t
{
    int a;
    int b;
};


int main()
{
    struct item_t item = { 1, 2, 3};

    entry_t *p = (entry_t*)(&item.b);
    printf("a = %d, b = %d\n", p->a, p->b);

    return 0;
}
```

代码输出的结果是：a = 2, b = 3

对于entry_t *p， 指针p指向的内存地址为&item.b。此时对于编译器来说，它认为从&item.b这个地址开始，是一个entry_t结构体的内存区域。并且把前4个字节当作entry_t成员变量a的值，后4个字节当作entry_t成员变量b的值。所以就有了a = 2, b = 3这个输出。

好了，现在开始讲解那两个难看懂的宏。先看一张图。



虽然本文最前面的图布局更好看一点，但这张图才更能反映文中这两个结构体的内存布局。不错，tqe_next是在tqe_prev的前面。这使得tqe_next、tqe_prev于tqh_first、tqh_last的内存布局一样。一级指针在前，二级指针在后。

现在来解析代码中最后两个宏函数。

队尾节点

```
//p = TAILQ_LAST(&queue_head, queue_head_t);
p = (*(((struct queue_head_t *)((&queue_head)->tqh_last))->tqh_last));
```

首先是(&queue_head)->tqh_last，它的值是最后一个元素的tqe_next这个成员变量的地址。然后把这个值强制转换成struct queue_head_t *指针。此时，相当于有一个匿名的struct queue_head_t类型指针q。它指向的地址为队列的最后一个节点的tqe_next成员变量的地址。无论一级还是二级指针，其都是指向另外一个地址。只是二级指针只能指向一个一级指针的地址。

此时，在编译器看来，从tqe_next这个变量的地址开始，是一个struct queue_head_t结构体的内存区域。并且可以将代码简写成：

```
p = (*(q->tqh_last));
```

回想一下刚才的那个例子。q->tqh_last的值就是上图中最后一个节点的tqe_prev成员变量的值。所以*(q->tqh_last))就相当于*tqe_prev。注意，变量tqe_prev是一个二级指针，它指向倒数第二个节点的tqe_next成员。所以*tqe_prev获取了倒数第二个节点的tqe_next成员的值。它的值就是最后一个节点的地址。最后，将这个地址赋值给p，此时p指向最后一个节点。完成了任务。好复杂的过程。

前一个节点

现在来看一下最后那个宏函数，代码如下：

```
//p = TAILQ_PREV(p, queue_head_t, entry);
p = (*(((struct queue_head_t *)((p)->entry.tqe_prev))->tqh_last));
```

注意，右边的p此时是指向最后一个节点(元素)的。所以(p)->entry.tqe_prev就是倒数第二个节点tqe_next成员的地址。然后又强制转换成struct queue_head_t指针。同样，假设一个匿名的struct queue_head_t *q;此时，宏函数可以转换成：

```
p = (*((q)->tqh_last));
```

同样，在编译器看来，从倒数第二个参数节点tqe_next的地址开始，是一个structqueue_head_t结构体的内存区域。所以tqh_last实际值是tqe_prev变量上的值，即tqe_prev指向的地址。*((q)->tqh_last)就是*tqe_prev，即获取tqe_prev指向的倒数第三个节点的tqe_next的值。而该值正是倒数第二个节点的地址。将这个地址赋值给p，此时，p就指向了倒数第二个节点。完成了TAILQ_PREV函数名的功能。

这个过程确实有点复杂。而且还涉及到强制类型转换。

其实，在TAILQ_LAST(&queue_head, queue_head_t);中，既然都可以获取最后一个节点的tqe_next的地址值，那么直接将该值 + 4就可以得到tqe_precv的地址值了（假设为pp）。有了该地址值pp，那么直接**pp就可以得到最后一个节点的地址了。代码如下：

```
struct queue_entry_t **pp = (&queue_head)->tqh_last;
pp += 1; //加1个指针的偏移量，在32位的系统中，就等于+4

//因为这里得到的是二级指针的地址值，所以按理来说，得到的是一个
//三级指针。故要用强制转换成三级指针。
struct queue_entry_t ***ppp = (struct queue_entry_t ***)pp;

s = **ppp;
printf("the last is %d\n", s->value);
```

该代码虽然能得到正确的结果，但总感觉直接加上一个偏移量的方式太粗暴了。

有一点要提出，+1那里并不会因为在64位的系统就不能运行，一样能正确运行的。因为1不是表示一个字节，而是一个指针的偏移量。在64位的系统上一个指针的偏移量为8字节。这种”指针 + 数值”，实际其增加的值为:数值 + sizeof(*指针)。不信的话，可以试一下char指针、int指针、结构体指针(结构体要有多个成员)。

好了，还是回到最开始的问题上吧。这个TAILQ_QUEUE队列是由两部分组成：队列头和队列节点。在Libevent中，队列头一般是event_base结构体的一个成员变量，而队列节点则是event结构体。比如event_base结构体里面有一个struct event_list eventqueue;其中，结构体struct event_list如下定义：

```
//event_struct.h
TAILQ_HEAD (event_list, event);

//所以event_list的定义展开后如下：
struct event_list
{
    struct event *tqh_first;
    struct event **tqh_last;
};
```

在event结构体中，则有几个TAILQ_ENTRY(event)类型的成员变量。这是因为根据不同的条件，采用不同的队列把这些event结构体连在一起，放到一条队列中。

8.event_io_map哈希表

原文地址

上面说到了TAILQ_QUEUE队列，它可以把多个event结构体连在一起。是一种归类方式。本文也将讲解一种将event归类、连在一起的结构：哈希结构。

哈希结构体

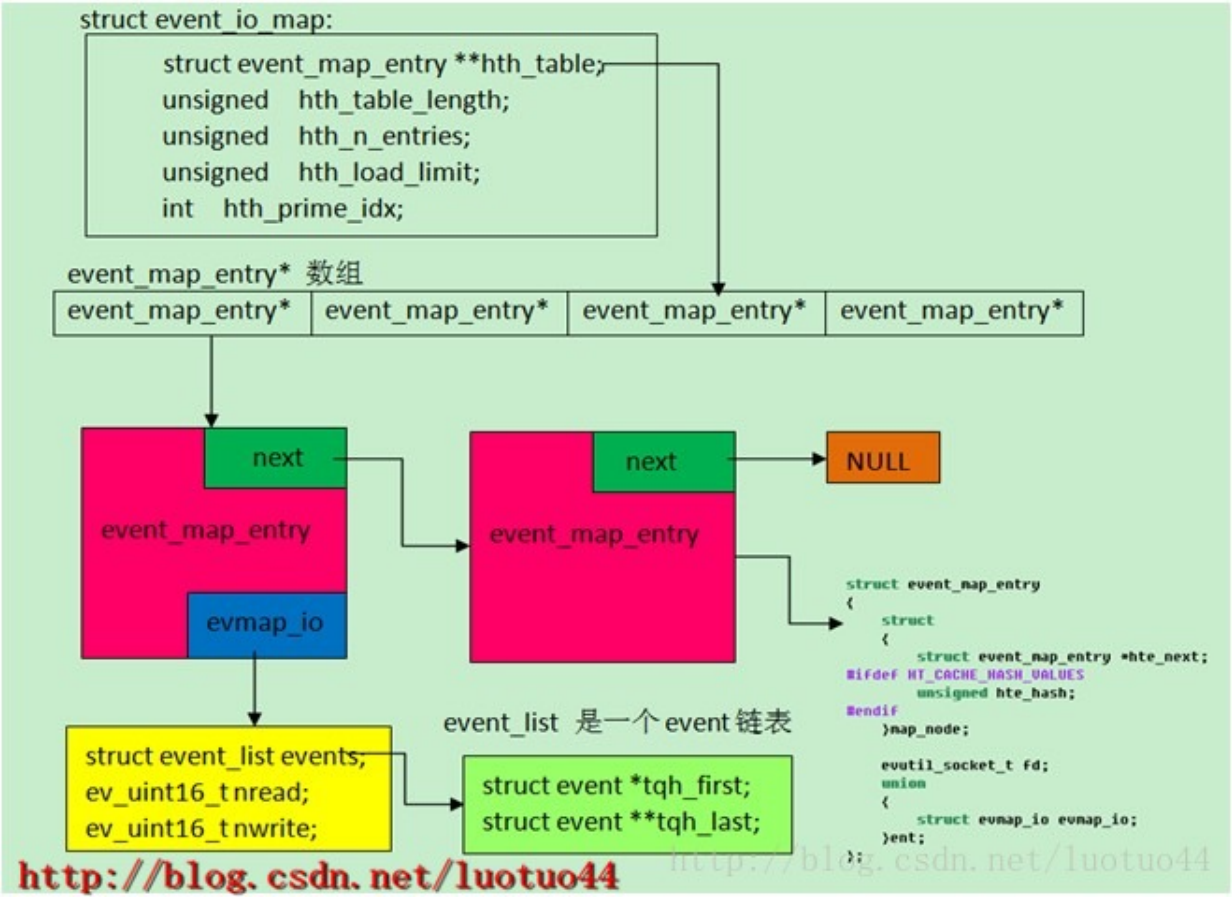
哈希结构由下面几个结构体一起配合工作：

```
struct event_list
{
    struct event *tqh_first;
    struct event **tqh_last;
};

struct evmap_io {
    //TAILQ_HEAD (event_list, event);
    struct event_list events;
    ev_uint16_t nread;
    ev_uint16_t nwrite;
};

struct event_map_entry {
    HT_ENTRY(event_map_entry) map_node; //next指针
    evutil_socket_t fd;
    union { /* This is a union in case we need to make more things that c
an
                be in the hashtable. */
        struct evmap_io evmap_io;
    } ent;
};

struct event_io_map
{
    //哈希表
    struct event_map_entry **hth_table;
    //哈希表的长度
    unsigned hth_table_length;
    //哈希的元素个数
    unsigned hth_n_entries;
    //resize 之前可以存多少个元素
    //在event_io_map_HT_GROW函数中可以看到其值为hth_table_length的
    //一半。但hth_n_entries>=hth_load_limit时，就会发生增长哈希表的长度
    unsigned hth_load_limit;
    //后面素数表中的下标值。主要是指明用到了哪个素数
    int hth_prime_idx;
};
```

结构体event_io_map指明了哈希表的存储位置、哈希表的长度、元素个数等数据。**该哈希表是使用链地址法解决冲突问题的**，这一点可以从hth_talbe成员变量看到。它是一个二级指针，因为哈希表的元素是event_map_entry指针。

除了怎么解决哈希冲突外，哈希表还有一个问题要解决，那就是哈希函数。这里的哈希函数就是模(%)。用event_map_entry结构体中的fd成员值模 event_io_map结构体中的hth_table_length。

由上面那些结构体配合得到的哈希表结构如下图所示：



从上图可以看到，两个发生了冲突的哈希表元素event_map_entry用一个next指向连在一起了(链地址解决冲突)。

另外，从图或者从前面关于event_map_entry结构体的定义可以看到，它有一个evmap_io结构体。而这个evmap_io结构体又有一个struct event_list 类型的成员，而struct event_list类型就是一个TAILQ_HEAD。这正是上文中说到的TAILQ_QUEUE队列的队列头。从这一点可以看到，这个哈希结构还是比较复杂的。

为什么在哈希表的元素里面，还会有一个TAILQ_QUEUE队列呢？这得由Libevent的一个特征说起。Libevent允许用同一个文件描述符fd或者信号值，调用event_new、event_add多次。所以，同一个fd或者信号值就可以对应多个event结构体了。所以这个TAILQ_QUEUE队列就是将这些具有相同fd或者信号值的多个event结构体连一起。

什么情况会使用哈希表

有一点需要说明，那就是Libevent中的哈希表只会用于Windows系统，像遵循POSIX标准的OS是不会用到哈希表的。从下面的定义可以看到这一点。

```
//event-internal.h文件
#ifdef WIN32
/* If we're on win32, then file descriptors are not nice low densely pack
ed
   integers.  Instead, they are pointer-like windows handles, and we want
 to
   use a hashtable instead of an array to map fds to events.
*/
#define EVMAP_USE_HT
#endif

#ifdef EVMAP_USE_HT
#include "ht-internal.h"
struct event_map_entry;
HT_HEAD(event_io_map, event_map_entry);
#else
#define event_io_map event_signal_map
#endif
```

可以看到，如果是非Windows系统，那么event_io_map就会被定义成event_signal_map(这是一个很简单的结构)。而在Windows系统，那么就由HT_HEAD这个宏定义event_io_map。最后得到的event_io_map就是本文最前面所示的那样。

为什么只有在Windows系统才会使用这个哈希表呢？代码里面的注释给出了一些解释。因为在Windows系统里面，文件描述符是一个比较大的值，不适合放到event_signal_map结构中。而通过哈希(模上一个小的值)，就可以变得比较小，这样就可以放到哈希表的数组中了。而遵循POSIX标准的文件描述符是从0开始递增的，一般都不会太大，适用于event_signal_map。

哈希函数

前面说到哈希函数是 用文件描述符fd 模 哈希表的长度。实际上，并不是直接用fd，而是用一个叫hashsocket的函数将这个fd进行一些处理后，才去模 哈希表的长度。下面是hashsocket函数的实现：

```
//evmap.c文件
/* Helper used by the event_io_map hashtable code; tries to return a good
 hash
 * of the fd in e->fd. */
static inline unsigned
hashsocket(struct event_map_entry *e)
{
    /* On win32, in practice, the low 2-3 bits of a SOCKET seem not to
     * matter.  Our hashtable implementation really likes low-order bits,

     * though, so let's do the rotate-and-add trick. */
    unsigned h = (unsigned) e->fd;
    h += (h >> 2) | (h << 30);
    return h;
}
```

前面的event_map_entry结构体中，还有一个HT_ENTRY的宏。从名字来看，它是一个哈希表的表项。它是一个条件宏，定义如下：

```
//ht-internal.h文件
#ifdef HT_CACHE_HASH_VALUES
#define HT_ENTRY(type)                          \
  struct {                                      \
    struct type *hte_next;                      \
    unsigned hte_hash;                          \
  }
#else
#define HT_ENTRY(type)                          \
  struct {                                      \
    struct type *hte_next;                      \
  }
#endif
```

可以看到，如果定义了HT_CACHE_HASH_VALUES宏，那么就会多一个hte_hash变量。从宏的名字来看，这是一个cache。不错，变量hte_hash就是用来存储前面的hashsocket的返回值。当第一次计算得到后，就存放到hte_hash变量中。以后需要用到(会经常用到)，就直接向这个变量要即可，无需再次计算hashsocket函数。如果没有这个变量，那么需要用到这个值，都要调用hashsocket函数计算一次。这一点从后面的代码可以看到。

哈希表操作函数

```
struct event_list
{
    struct event *tqh_first;
    struct event **tqh_last;
};

struct evmap_io
{
    struct event_list events;
    ev_uint16_t nread;
    ev_uint16_t nwrite;
};


struct event_map_entry
{
    struct
    {
        struct event_map_entry *hte_next;
#ifdef HT_CACHE_HASH_VALUES
        unsigned hte_hash;
#endif
    }map_node;

    evutil_socket_t fd;
    union
    {
        struct evmap_io evmap_io;
    }ent;
};

struct event_io_map
{
    //哈希表
    struct event_map_entry **hth_table;
    //哈希表的长度
    unsigned hth_table_length;
    //哈希的元素个数
    unsigned hth_n_entries;
    //resize 之前可以存多少个元素
    //在event_io_map_HT_GROW函数中可以看到其值为hth_table_length的
    //一半。但hth_n_entries>=hth_load_limit时，就会发生增长哈希表的长度
    unsigned hth_load_limit;
    //后面素数表中的下标值。主要是指明用到了哪个素数
    int hth_prime_idx;
};



int event_io_map_HT_GROW(struct event_io_map *ht, unsigned min_capacity);
void event_io_map_HT_CLEAR(struct event_io_map *ht);
```

```
int _event_io_map_HT_REP_IS_BAD(const struct event_io_map *ht);

//初始化event_io_map
static inline void event_io_map_HT_INIT(struct event_io_map *head)
{
    head->hth_table_length = 0;
    head->hth_table = NULL;
    head->hth_n_entries = 0;
    head->hth_load_limit = 0;
    head->hth_prime_idx = -1;
}


//在event_io_map这个哈希表中，找个表项elm
//在下面还有一个相似的函数，函数名少了最后的_P。那个函数的返回值
//是event_map_entry *。从查找来说，后面那个函数更适合。之所以
//有这个函数，是因为哈希表还有replace、remove这些操作。对于
//A->B->C这样的链表。此时，要replace或者remove节点B。
//如果只有后面那个查找函数，那么只能查找并返回一个指向B的指针。
//此时将无法修改A的next指针了。所以本函数有存在的必要。
//在本文件中，很多函数都使用了event_map_entry **。
//因为event_map_entry **类型变量，既可以修改本元素的hte_next变量
//也能指向下一个元素。

//该函数返回的是查找节点的前驱节点的hte_next成员变量的地址。
//所以返回值肯定不会为NULL,但是对返回值取*就可能为NULL
static inline struct event_map_entry **
_event_io_map_HT_FIND_P(struct event_io_map *head,
                        struct event_map_entry *elm)
{
    struct event_map_entry **p;
    if (!head->hth_table)
        return NULL;

#ifdef HT_CACHE_HASH_VALUES
    p = &((head)->hth_table[((elm)->map_node.hte_hash)
            % head->hth_table_length]);
#else
    p = &((head)->hth_table[(hashsocket(*elm))%head->hth_table_length]);
#endif

    //这里的哈希表是用链地址法解决哈希冲突的。
    //上面的 % 只是找到了冲突链的头。现在是在冲突链中查找。
    while (*p)
    {
        //判断是否相等。在实现上，只是简单地根据fd来判断是否相等
        if (eqsocket(*p, elm))
            return p;

        //p存放的是hte_next成员变量的地址
        p = &(*p)->map_node.hte_next;
```

```
    }

    return p;
}

/* Return a pointer to the element in the table 'head' matching 'elm',
 * or NULL if no such element exists */
//在event_io_map这个哈希表中，找个表项elm
static inline struct event_map_entry *
event_io_map_HT_FIND(const struct event_io_map *head,
                     struct event_map_entry *elm)
{
    struct event_map_entry **p;
    struct event_io_map *h = (struct event_io_map *) head;

#ifdef HT_CACHE_HASH_VALUES
    do
    {   //计算哈希值
        (elm)->map_node.hte_hash = hashsocket(elm);
    } while(0);
#endif

    p = _event_io_map_HT_FIND_P(h, elm);
    return p ? *p : NULL;
}


/* Insert the element 'elm' into the table 'head'.  Do not call this
 * function if the table might already contain a matching element. */
static inline void
event_io_map_HT_INSERT(struct event_io_map *head,
                       struct event_map_entry *elm)
{
    struct event_map_entry **p;
    if (!head->hth_table || head->hth_n_entries >= head->hth_load_limit)
        event_io_map_HT_GROW(head, head->hth_n_entries+1);

    ++head->hth_n_entries;

#ifdef HT_CACHE_HASH_VALUES
    do
    {   //计算哈希值.此哈希不同于用%计算的简单哈希。
        //存放到hte_hash变量中，作为cache
        (elm)->map_node.hte_hash = hashsocket(elm);
    } while (0);

    p = &((head)->hth_table[((elm)->map_node.hte_hash)
            % head->hth_table_length]);
#else
    p = &((head)->hth_table[(hashsocket(*elm))%head->hth_table_length]);
#endif
```

```
    //使用头插法，即后面才插入的链表，反而会在链表头。
    elm->map_node.hte_next = *p;
    *p = elm;
}


/* Insert the element 'elm' into the table 'head'. If there already
 * a matching element in the table, replace that element and return
 * it. */
static inline struct event_map_entry *
event_io_map_HT_REPLACE(struct event_io_map *head,
                        struct event_map_entry *elm)
{
    struct event_map_entry **p, *r;

    if (!head->hth_table || head->hth_n_entries >= head->hth_load_limit)
        event_io_map_HT_GROW(head, head->hth_n_entries+1);

#ifdef HT_CACHE_HASH_VALUES
    do
    {
        (elm)->map_node.hte_hash = hashsocket(elm);
    } while(0);
#endif

    p = _event_io_map_HT_FIND_P(head, elm);

    //由前面的英文注释可知，这个函数是替换插入一起进行的。如果哈希表
    //中有和elm相同的元素(指的是event_map_entry的fd成员变量值相等)
    //那么就发生替代(其他成员变量值不同，所以不是完全相同，有替换意义)
    //如果哈希表中没有和elm相同的元素，那么就进行插入操作

    //指针p存放的是hte_next成员变量的地址
    //这里的p存放的是被替换元素的前驱元素的hte_next变量地址
    r = *p; //r指向了要替换的元素。有可能为NULL
    *p = elm; //hte_next变量被赋予新值elm

    //找到了要被替换的元素r(不为NULL)
    //而且要插入的元素地址不等于要被替换的元素地址
    if (r && (r!=elm))
    {
        elm->map_node.hte_next = r->map_node.hte_next;

        r->map_node.hte_next = NULL;
        return r; //返回被替换掉的元素
    }
    else //进行插入操作
    {
        //这里貌似有一个bug。如果前一个判断中，r 不为 NULL，而且r == elm
        //对于同一个元素，多次调用本函数。就会出现这样情况。
        //此时，将会来到这个else里面
```

```
        //实际上没有增加元素，但元素的个数却被++了。因为r 的地址等于 elm
        //所以 r = *p; *p = elm; 等于什么也没有做。（r == elm)
        //当然，因为这些函数都是Libevent内部使用的。如果它保证不会同于同
        //一个元素调用本函数，那么就不会出现bug
        ++head->hth_n_entries;
        return NULL; //插入操作返回NULL，表示没有替换到元素
    }
}


/* Remove any element matching 'elm' from the table 'head'.  If such
 * an element is found, return it; otherwise return NULL. */
static inline struct event_map_entry *
event_io_map_HT_REMOVE(struct event_io_map *head,
                       struct event_map_entry *elm)
{
    struct event_map_entry **p, *r;

#ifdef HT_CACHE_HASH_VALUES
    do
    {
        (elm)->map_node.hte_hash = hashsocket(elm);
    } while (0);
#endif

    p = _event_io_map_HT_FIND_P(head,elm);

    if (!p || !*p)//没有找到
        return NULL;

    //指针p存放的是hte_next成员变量的地址
    //这里的p存放的是被替换元素的前驱元素的hte_next变量地址
    r = *p; //r现在指向要被删除的元素
    *p = r->map_node.hte_next;
    r->map_node.hte_next = NULL;

    --head->hth_n_entries;

    return r;
}


/* Invoke the function 'fn' on every element of the table 'head',
 using 'data' as its second argument.  If the function returns
 nonzero, remove the most recently examined element before invoking
 the function again. */
static inline void
event_io_map_HT_FOREACH_FN(struct event_io_map *head,
                           int (*fn)(struct event_map_entry *, void *),
                           void *data)
{
    unsigned idx;
```

```
    struct event_map_entry **p, **nextp, *next;

    if (!head->hth_table)
        return;

    for (idx=0; idx < head->hth_table_length; ++idx)
    {
        p = &head->hth_table[idx];

        while (*p)
        {
            //像A->B->C链表。p存放了A元素中hte_next变量的地址
            //*p则指向B元素。nextp存放B的hte_next变量的地址
            //next指向C元素。
            nextp = &(*p)->map_node.hte_next;
            next = *nextp;

            //对B元素进行检查
            if (fn(*p, data))
            {
                --head->hth_n_entries;
                //p存放了A元素中hte_next变量的地址
                //所以*p = next使得A元素的hte_next变量值被赋值为
                //next。此时链表变成A->C。即使抛弃了B元素。不知道
                //调用方是否能释放B元素的内存。
                *p = next;
            }
            else
            {
                p = nextp;
            }
        }
    }
}


/* Return a pointer to the first element in the table 'head', under
 * an arbitrary order.  This order is stable under remove operations,
 * but not under others. If the table is empty, return NULL. */
//获取第一条冲突链的第一个元素
static inline struct event_map_entry **
event_io_map_HT_START(struct event_io_map *head)
{
    unsigned b = 0;

    while (b < head->hth_table_length)
    {
        //返回哈希表中，第一个不为NULL的节点
        //即有event_map_entry元素的节点。
        //找到链。因为是哈希。所以不一定哈希表中的每一个节点都存有元素
        if (head->hth_table[b])
            return &head->hth_table[b];
```

```
        ++b;
    }

    return NULL;
}



/* Return the next element in 'head' after 'elm', under the arbitrary
 * order used by HT_START.  If there are no more elements, return
 * NULL.  If 'elm' is to be removed from the table, you must call
 * this function for the next value before you remove it.
 */
static inline struct event_map_entry **
event_io_map_HT_NEXT(struct event_io_map *head,
                     struct event_map_entry **elm)
{
    //本哈希解决冲突的方式是链地址
    //如果参数elm所在的链地址中，elm还有下一个节点，就直接返回下一个节点
    if ((*elm)->map_node.hte_next)
    {
        return &(*elm)->map_node.hte_next;
    }
    else //否则取哈希表中的下一条链中第一个元素
    {
#ifdef HT_CACHE_HASH_VALUES
        unsigned b = (((*elm)->map_node.hte_hash)
                      % head->hth_table_length) + 1;
#else
        unsigned b = ( (hashsocket(*elm)) % head->hth_table_length) + 1;
#endif

        while (b < head->hth_table_length)
        {
            //找到链。因为是哈希。所以不一定哈希表中的每一个节点都存有元素
            if (head->hth_table[b])
                return &head->hth_table[b];
            ++b;
        }

        return NULL;
    }
}



//功能同上一个函数。只是参数不同，另外本函数还会使得--hth_n_entries
//该函数主要是返回elm的下一个元素。并且哈希表的总元素个数减一。
//主调函数会负责释放*elm指向的元素。无需在这里动手
//在evmap_io_clear函数中，会调用本函数。
static inline struct event_map_entry **
```

```
event_io_map_HT_NEXT_RMV(struct event_io_map *head,
                         struct event_map_entry **elm)
{
#ifdef HT_CACHE_HASH_VALUES
    unsigned h = ((*elm)->map_node.hte_hash);
#else
    unsigned h = (hashsocket(*elm));
#endif

    //elm变量变成存放下一个元素的hte_next的地址
    *elm = (*elm)->map_node.hte_next;

    --head->hth_n_entries;

    if (*elm)
    {
        return elm;
    }
    else
    {
        unsigned b = (h % head->hth_table_length)+1;

        while (b < head->hth_table_length)
        {
            if (head->hth_table[b])
                return &head->hth_table[b];

            ++b;
        }

        return NULL;
    }
}



//素数表。之所以去素数，是因为在取模的时候，素数比合数更有优势。
//听说是更散，更均匀
static unsigned event_io_map_PRIMES[] =
{
    //素数表的元素具有差不多2倍的关系。
    //这使得扩容操作不会经常发生。每次扩容都预留比较大的空间
    53, 97, 193, 389, 769, 1543, 3079,
    6151, 12289, 24593, 49157, 98317,
    196613, 393241, 786433, 1572869, 3145739,
    6291469, 12582917, 25165843, 50331653, 100663319,
    201326611, 402653189, 805306457, 1610612741
};


//素数表中，元素的个数。
static unsigned event_io_map_N_PRIMES =
```

```
        (unsigned)(sizeof(event_io_map_PRIMES)
                   /sizeof(event_io_map_PRIMES[0]));


/* Expand the internal table of 'head' until it is large enough to
 * hold 'size' elements.  Return 0 on success, -1 on allocation
 * failure. */
int event_io_map_HT_GROW(struct event_io_map *head, unsigned size)
{
    unsigned new_len, new_load_limit;
    int prime_idx;

    struct event_map_entry **new_table;
    //已经用到了素数表中最后一个素数，不能再扩容了。
    if (head->hth_prime_idx == (int)event_io_map_N_PRIMES - 1)
        return 0;

    //哈希表中还够容量，无需扩容
    if (head->hth_load_limit > size)
        return 0;

    prime_idx = head->hth_prime_idx;

    do {
        new_len = event_io_map_PRIMES[++prime_idx];

        //从素数表中的数值可以看到，后一个差不多是前一个的2倍。
        //从0.5和后的new_load_limit <= size，可以得知此次扩容
        //至少得是所需大小(size)的2倍以上。免得经常要进行扩容
        new_load_limit = (unsigned)(0.5*new_len);
    } while (new_load_limit <= size
             && prime_idx < (int)event_io_map_N_PRIMES);

    if ((new_table = mm_malloc(new_len*sizeof(struct event_map_entry*))))
    {
        unsigned b;
        memset(new_table, 0, new_len*sizeof(struct event_map_entry*));

        for (b = 0; b < head->hth_table_length; ++b)
        {
            struct event_map_entry *elm, *next;
            unsigned b2;

            elm = head->hth_table[b];
            while (elm) //该节点有冲突链。遍历冲突链中所有的元素
            {
                next = elm->map_node.hte_next;

                //冲突链中的元素，相对于前一个素数同余(即模素数后，结果相当)
                //但对于现在的新素数就不一定同余了，即它们不一定还会冲突
                //所以要对冲突链中的所有元素都再次哈希，并放到它们应该在的地方
                //b2存放了再次哈希后，元素应该存放的节点下标。
```

```
#ifdef HT_CACHE_HASH_VALUES
                b2 = (elm)->map_node.hte_hash % new_len;
#else
                b2 = (hashsocket(*elm)) % new_len;
#endif
                //用头插法插入数据
                elm->map_node.hte_next = new_table[b2];
                new_table[b2] = elm;

                elm = next;
            }
        }

        if (head->hth_table)
            mm_free(head->hth_table);

        head->hth_table = new_table;
    }
    else
    {
        unsigned b, b2;

        //刚才mm_malloc失败，可能是内存不够。现在用更省内存的
        //mm_realloc方式。当然其代价就是更耗时(下面的代码可以看到)。
        //前面的mm_malloc会同时有hth_table和new_table两个数组。
        //而mm_realloc则只有一个数组，所以省内存，省了一个hth_table数组
        //的内存。此时，new_table数组的前head->hth_table_length个
        //元素存放了原来的冲突链的头部。也正是这个原因导致后面代码更耗时。
        //其实，只有在很特殊的情况下，这个函数才会比mm_malloc省内存.
        //就是堆内存head->hth_table区域的后面刚好有一段可以用的内存。
        //具体的，可以搜一下realloc这个函数。
        new_table = mm_realloc(head->hth_table,
                               new_len*sizeof(struct event_map_entry*));

        if (!new_table)
            return -1;

        memset(new_table + head->hth_table_length, 0,
               (new_len - head->hth_table_length)*sizeof(struct event_map
_entry*)
               );

        for (b=0; b < head->hth_table_length; ++b)
        {
            struct event_map_entry *e, **pE;

            for (pE = &new_table[b], e = *pE; e != NULL; e = *pE)
            {

#ifdef HT_CACHE_HASH_VALUES
                b2 = (e)->map_node.hte_hash % new_len;
#else
```

```
                b2 = (hashsocket(*elm)) % new_len;
#endif
                //对于冲突链A->B->C.
                //pE是二级指针，存放的是A元素的hte_next指针的地址值
                //e指向B元素。

                //对新的素数进行哈希，刚好又在原来的位置
                if (b2 == b)
                {
                    //此时，无需修改。接着处理冲突链中的下一个元素即可
                    //pE向前移动，存放B元素的hte_next指针的地址值
                    pE = &e->map_node.hte_next;
                }
                else//这个元素会去到其他位置上。
                {
                    //此时冲突链修改成A->C。
                    //所以pE无需修改，还是存放A元素的hte_next指针的地址值
                    //但A元素的hte_next指针要指向C元素。用*pE去修改即可
                    *pE = e->map_node.hte_next;

                    //将这个元素放到正确的位置上。
                    e->map_node.hte_next = new_table[b2];
                    new_table[b2] = e;
                }

                //这种再次哈希的方式，很有可能会对某些元素操作两次。
                //当某个元素第一次在else中处理，那么它就会被哈希到正确的节点
                //的冲突链上。随着外循环的进行，处理到正确的节点时。在遍历该节点
                //的冲突链时，又会再次处理该元素。此时，就会在if中处理。而不会
                //进入到else中。
            }
        }

        head->hth_table = new_table;
    }


    //一般是当hth_n_entries >= hth_load_limit时，就会调用
    //本函数。hth_n_entries表示的是哈希表中节点的个数。而hth_load_limit
    //是hth_table_length的一半。hth_table_length则是哈希表中
    //哈希函数被模的数字。所以，当哈希表中的节点个数到达哈希表长度的一半时
    //就会发生增长，本函数被调用。这样的结果是：平均来说，哈希表应该比较少发生
    //冲突。即使有，冲突链也不会太长。这样就能有比较快的查找速度。
    head->hth_table_length = new_len;
    head->hth_prime_idx = prime_idx;
    head->hth_load_limit = new_load_limit;

    return 0;
}


/* Free all storage held by 'head'.  Does not free 'head' itself,
```

```
 * or individual elements. 并不需要释放独立的元素*/
//在evmap_io_clear函数会调用该函数。其是在删除所有哈希表中的元素后
//才调用该函数的。
void event_io_map_HT_CLEAR(struct event_io_map *head)
{
    if (head->hth_table)
        mm_free(head->hth_table);

    head->hth_table_length = 0;

    event_io_map_HT_INIT(head);
}


/* Debugging helper: return false iff the representation of 'head' is
 * internally consistent. */
int _event_io_map_HT_REP_IS_BAD(const struct event_io_map *head)
{
    unsigned n, i;
    struct event_map_entry *elm;

    if (!head->hth_table_length)
    {
        //刚被初始化，还没申请任何空间
        if (!head->hth_table && !head->hth_n_entries
                && !head->hth_load_limit && head->hth_prime_idx == -1
                )
            return 0;
        else
            return 1;
    }

    if (!head->hth_table || head->hth_prime_idx < 0
            || !head->hth_load_limit
            )
        return 2;

    if (head->hth_n_entries > head->hth_load_limit)
        return 3;

    if (head->hth_table_length != event_io_map_PRIMES[head->hth_prime_id
x])
        return 4;

    if (head->hth_load_limit != (unsigned)(0.5*head->hth_table_length))
        return 5;

    for (n = i = 0; i < head->hth_table_length; ++i)
    {
        for (elm = head->hth_table[i]; elm; elm = elm->map_node.hte_next)
        {
```

```
#ifdef HT_CACHE_HASH_VALUES

            if (elm->map_node.hte_hash != hashsocket(elm))
                return 1000 + i;

            if( (elm->map_node.hte_hash % head->hth_table_length) != i)
                return 10000 + i;

#else
            if ( (hashsocket(*elm)) != hashsocket(elm))
                return 1000 + i;

            if( ( (hashsocket(*elm)) % head->hth_table_length) != i)
                return 10000 + i;
#endif
            ++n;
        }
    }

    if (n != head->hth_n_entries)
        return 6;

    return 0;
}
```

代码中的注释已经对这个哈希表的一些特征进行了描述，这里就不多说了。

哈希表在Libevent的使用

现在来讲一下event_io_map的应用。

在event_base这个结构体中有一个event_io_map类型的成员变量io。它就是一个哈希表。当一个监听读或者写操作的event，调用event_add函数插入到event_base中时，就会调用evmap_io_add函数。evmap_io_add函数应用到这个event_io_map结构体。该函数的定义如下，其中使用到了一个宏定义，我已经展开了。

```
int
evmap_io_add(struct event_base *base, evutil_socket_t fd, struct event *e
v)
{
    const struct eventop *evsel = base->evsel;
    struct event_io_map *io = &base->io;
    struct evmap_io *ctx = NULL;
    int nread, nwrite, retval = 0;
    short res = 0, old = 0;
    struct event *old_ev;

    EVUTIL_ASSERT(fd == ev->ev_fd);

    if (fd < 0)
        return 0;

    //GET_IO_SLOT_AND_CTOR(ctx, io, fd, evmap_io, evmap_io_init,
    //                     evsel->fdinfo_len);SLOT指的是fd
    //GET_IO_SLOT_AND_CTOR宏将展开成下面这个do{}while(0);
    do
    {
        struct event_map_entry _key, *_ent;
        _key.fd = fd;

        struct event_io_map *_ptr_head = io;
        struct event_map_entry **ptr;

        //哈希表扩容，减少冲突的可能性
        if (!_ptr_head->hth_table
                || _ptr_head->hth_n_entries >= _ptr_head->hth_load_limit)
        {
            event_io_map_HT_GROW(_ptr_head,
                                 _ptr_head->hth_n_entries + 1);
        }

#ifdef HT_CACHE_HASH_VALUES
        do{
            (&_key)->map_node.hte_hash = hashsocket((&_key));
        } while(0);
#endif

        //返回值ptr,是要查找节点的前驱节点的hte_next成员变量的地址.
        //所以返回值肯定不会为NULL,而*ptr就可能为NULL。说明hte_next
        //不指向任何节点。也正由于这个原因，所以即使*ptr 为NULL,但是可以
        //给*ptr赋值。此时，是修改前驱节点的hte_next成员变量的值，使之
        //指向另外一个节点。
        //这里调用_event_io_map_HT_FIND_P原因有二：1.查看该fd是否已经
        //插入过这个哈希表中。2.得到这个fd计算哈希位置。
        ptr = _event_io_map_HT_FIND_P(_ptr_head, (&_key));

        //在event_io_map这个哈希表中查找是否已经存在该fd的event_map_entry了
```

```
        //因为同一个fd可以调用event_new多次，然后event_add多次的。
        if (*ptr)
        {
            _ent = *ptr;
        }
        else
        {
            _ent = mm_calloc(1, sizeof(struct event_map_entry) + evsel->f
dinfo_len);
            if (EVUTIL_UNLIKELY(_ent == NULL))
                return (-1);

            _ent->fd = fd;
              //调用初始化函数初始化这个evmap_io
            (evmap_io_init)(&_ent->ent.evmap_io);

#ifdef HT_CACHE_HASH_VALUES
            do
            {
                ent->map_node.hte_hash = (&_key)->map_node.hte_hash;
            }while(0);
#endif
            _ent->map_node.hte_next = NULL;

            //把这个新建的节点插入到哈希表中。ptr已经包含了哈希位置
            *ptr = _ent;
            ++(io->hth_n_entries);
        }


        //这里是获取该event_map_entry的next和prev指针。因为
        //evmap_io含有next、prev变量。这样在之后就可以把这个
        //event_map_entry连起来。这个外do{}while(0)的功能是
        //为这个fd分配一个event_map_entry,并且插入到现有的哈希
        //表中。同时，这个fd还是结构体event的一部分。而event必须
        //插入到event队列中。
        (ctx) = &_ent->ent.evmap_io;

    } while (0);


    ....

    //ctx->events是一个TAILQ_HEAD。结合之前讲到的TAILQ_QUEUE队列，
    //就可以知道：同一个fd，可能有多个event结构体。这里就把这些结构体连
    //起来。依靠的链表是，event结构体中的ev_io_next。ev_io_next是
    //一个TAILQ_ENTRY,具有前驱和后驱指针。队列头部为event_map_entry
    //结构体中的evmap_io成员的events成员。
    TAILQ_INSERT_TAIL(&ctx->events, ev, ev_io_next);

    return (retval);
}
```

GET_IO_SLOT_AND_CTOR宏的作用就是让ctx指向struct event_map_entry结构体中的TAILQ_HEAD。这样就可以使用TAILQ_INSERT_TAIL宏，把ev变量插入到队列中。如果有现成的event_map_entry就直接使用，没有的话就新建一个。

9.event_signal_map

原文地址

相关结构体

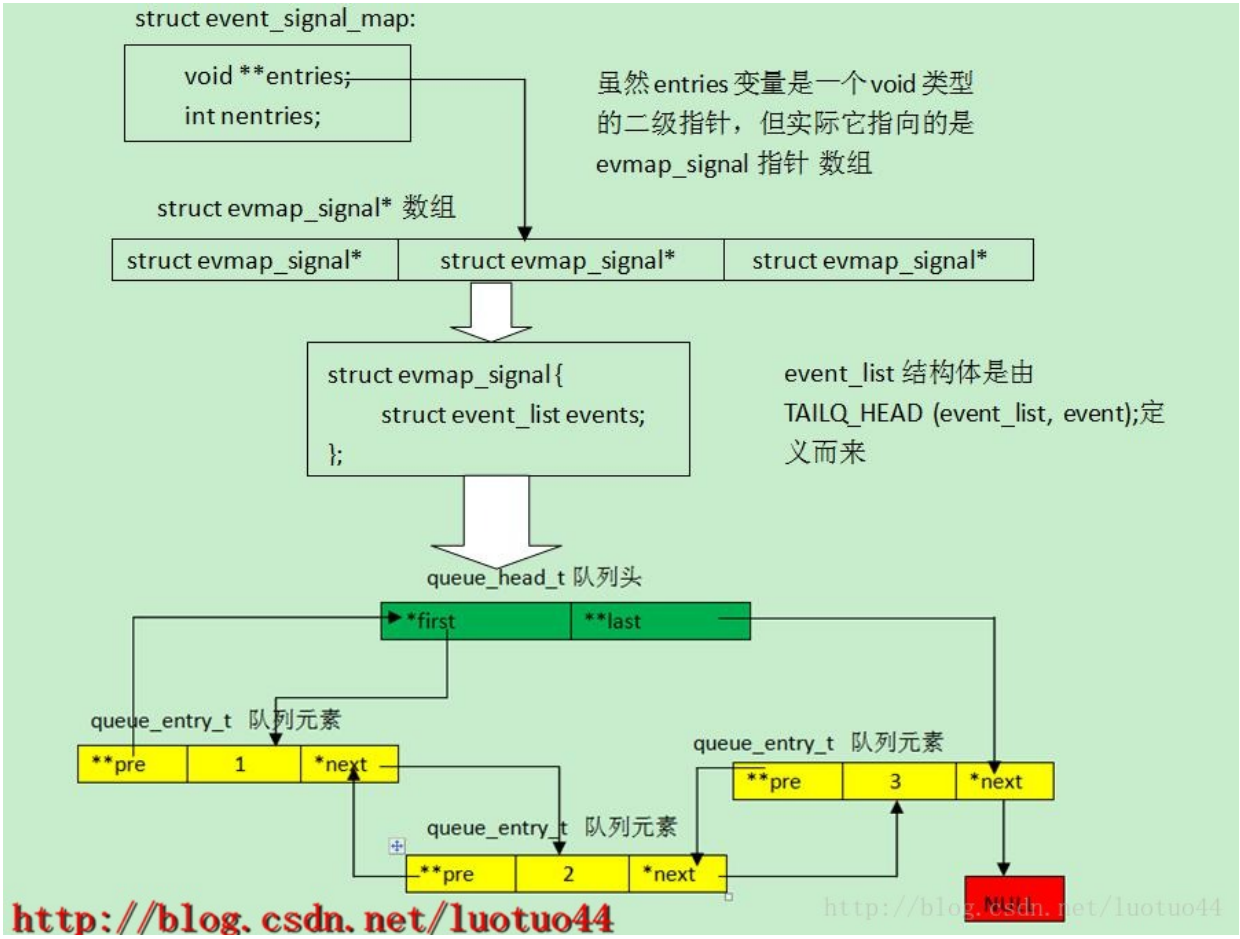
因为event_signal_map结构体实在太简单了，所以不像event_io_map那样，有一个专门的文件。由于没有专门的文件，那么只能从蛛丝马迹上探索这个event_signal_map结构了。

通过一些搜索，可以得到与event_signal_map相关联的一些结构体有下面这些：

```
//TAILQ_HEAD (event_list, event); event_struct.h
struct event_list
{
    struct event *tqh_first;
    struct event **tqh_last;
};

struct evmap_signal {
    struct event_list events;
};

struct event_signal_map {
    /* An array of evmap_io * or of evmap_signal *; empty entries are
     * set to NULL. */
    void **entries; //二级指针，evmap_signal*数组
    int nentries; //元素个数
};
```

相对于event_io_map是一个哈希表，event_signal_map是一个数组。并且entries虽然是void**，但实际它是一个structevmap_signal指针数组。而evmap_signal成员只有一个TAILQ_HEAD (event_list, event);

由上面这些结构配合得到的结构如下所示：



从上图可以看到，event_signal_map结构体指向了一个evmap_signal指针数组，而evmap_signal结构体可以说就是一个TAILQ_QUEUE队列的队列头。通过TAILQ_QUEUE队列和图中的结构，event_signal_map就可以把event结构体都关联并管理起来。

上图中会有一个event结构体队列，原因和前一小节是一样的。因为同一个文件描述符fd或者信号值sig是可以多次调用event_new、event_add函数的。即存在一对多。这个队列就是把同一个fd或者sig的event连起来，方便管理。

操作函数

虽然从上图看到event_signal_map结构是比较简单，现在还是要看一下与event_signal_map相关联的函数。先看一个内存分配函数。

```
//evmap.c文件
//slot是信号值sig，或者文件描述符fd.
//当sig或者fd >= map的nentries变量时就会调用此函数
static int   //msize 等于 sizeof(struct evmap_signal *)
evmap_make_space(struct event_signal_map *map, int slot, int msize)
{
    if (map->nentries <= slot) {
        //posix标准中，信号的种类就只有32种。
        //http://blog.csdn.net/luotuo44/article/details/16799607
        //在Windows中，信号的种类就更少了，只有6种。
        //http://msdn.microsoft.com/zh-cn/library/xdkz3x12.aspx
        //所以一开始取32还是比较合理的
        int nentries = map->nentries ? map->nentries : 32;
        void **tmp;

        //当slot是一个文件描述符时，就会大于32
        while (nentries <= slot)
            nentries <<= 1;

        tmp = (void **)mm_realloc(map->entries, nentries * msize);
        if (tmp == NULL)
            return (-1);

        //清零是很有必要的。因为tmp是二级指针，数组里面的元素是一个指针
        memset(&tmp[map->nentries], 0,
            (nentries - map->nentries) * msize);

        map->nentries = nentries;
        map->entries = tmp;
    }

    return (0);
}
```

在非Windows系统中，event_io_map被定义成event_signal_map,此时slot将是文件描述符fd。但对于这些遵循POSIX标准的OS来说，fd都是从0递增的比较小的值。所以代码中的while(nentrie <= slot) nentries <<= 1; 并不会很大。对于slot为一个信号值，最大也就只有32。所以总得来说，nentries的值并不会太大。

从上面的代码也可以想到，对于参数slot，它要么是信号值sig，要么是文件描述符fd。而event_signal_map要求的数组长度一定要大于slot。那么之后给定一个sig或者fd，就可以直接通过下标操作快速定位了。这是因为一个sig或者fd就对应在数组中占有一个位置，并且sig或者fd的值等于其在数组位置的下标值。这种方式无论是管理还是代码复杂度都要比前一篇博文说到的哈希表要简单。

在Libevent中的应用

同event_io_map一样，event_signal_map同样也是有event_signal_add函数的。不过后者比前者简单了很多。

```
//event.c文件
int
evmap_signal_add(struct event_base *base, int sig, struct event *ev)
{
    const struct eventop *evsel = base->evsigsel;
    struct event_signal_map *map = &base->sigmap;
    struct evmap_signal *ctx = NULL;

    if (sig >= map->nentries) {

        if (evmap_make_space(map, sig, sizeof(struct evmap_signal *)) ==
 -1)
            return (-1);
    }

    //无论是GET_SIGNAL_SLOT_AND_CTOR还是GET_IO_SLOT_AND_CTOR，其作用
    //都是在数组(哈希表也是一个数组)中找到fd中的一个结构。
    //GET_SIGNAL_SLOT_AND_CTOR(ctx, map, sig, evmap_signal, evmap_signal_
init,
    //base->evsigsel->fdinfo_len);
    do
    {
        //同event_io_map一样，同一个信号或者fd可以被多次event_new、event_add
        //所以，当同一个信号或者fd被多次event_add后，entries[sig]就不会为NULL
        if ((map)->entries[sig] == NULL)//第一次
        {
            //evmap_signal成员只有一个TAILQ_HEAD (event_list, event);
            //可以说evmap_signal本身就是一个TAILQ_HEAD
            //这个赋值操作很重要。
            (map)->entries[sig] = mm_calloc(1, sizeof(struct evmap_signa
l)
                                                + base->evsigsel->fdinfo_l
en
                                            );

            if (EVUTIL_UNLIKELY((map)->entries[sig] == NULL))
                return (-1);


            //内部调用TAILQ_INIT(&entry->events);
            (evmap_signal_init)((struct evmap_signal *)(map)->entries[si
g]);
        }

        (ctx) = (struct evmap_signal *)((map)->entries[sig]);
    } while (0);

    ...

    //将所有有相同信号值的event连起来
    TAILQ_INSERT_TAIL(&ctx->events, ev, ev_signal_next);
```

```
    return (1);
}
```

同样，GET_SIGNAL_SLOT_AND_CTOR宏的作用就是让ctx指向structevmap_signal结构体中的TAILQ_HEAD。这样就可以使用TAILQ_INSERT_TAIL宏，把ev变量插入到队列中。如果有现成的struct evmap_signal就直接使用，没有的话就新建一个。

有一点要提出，虽然前一小节中提到在非Windows系统中，event_io_map也被定义成了event_signal_map，但实际上他们并不会由链表连在一起。因为在event_base结构体中，分别有struct event_io_map变量io和event_signal_map变量sigmap。所有的io类型的event结构体会被放到io中，而信号类型的event则会被放到sigmap变量中。

10.配置event_base

原文地址

前面都是讲一些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。

```
//event.c文件
struct event_base *
event_base_new(void)
{
    struct event_base *base = NULL;
    struct event_config *cfg = event_config_new();
    if (cfg) {
        base = event_base_new_with_config(cfg);
        event_config_free(cfg);
    }
    return base;
}
```

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。

```
struct event_config {
    TAILQ_HEAD(event_configq, event_config_entry) entries;

    int n_cpus_hint;
    enum event_method_feature require_features;
    enum event_base_config_flag flags;
};

struct event_config_entry {
    TAILQ_ENTRY(event_config_entry) next;

    const char *avoid_method;
};
```

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》。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。

```
int
event_config_avoid_method(struct event_config *cfg, const char *method)
{
    struct event_config_entry *entry = mm_malloc(sizeof(*entry));
    if (entry == NULL)
        return (-1);

    //复制字符串
    if ((entry->avoid_method = mm_strdup(method)) == NULL) {
        mm_free(entry);
        return (-1);
    }

    //插入到队列中
    TAILQ_INSERT_TAIL(&cfg->entries, entry, next);

    return (0);
}
```

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。

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。

```
//event.h文件
enum event_method_feature {
    //支持边沿触发
    EV_FEATURE_ET = 0x01,
    //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)
    //select、poll是不能满足这个特征的.epoll则满足
    EV_FEATURE_O1 = 0x02,
    //支持任意的文件描述符，而不能仅仅支持套接字
    EV_FEATURE_FDS = 0x04
};
```

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。

```
int
event_config_require_features(struct event_config *cfg,
    int features)
{
    if (!cfg)
        return (-1);
    cfg->require_features = features;
    return (0);
}
```

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。

```
#include<event.h>
#include<stdio.h>

int main()
{
    event_config *cfg = event_config_new();
    event_config_require_features(cfg,  EV_FEATURE_O1 | EV_FEATURE_FDS);


    event_base *base = event_base_new_with_config(cfg);
    if( base == NULL )
    {
        printf("don't support this features\n");
         base = event_base_new(); //使用默认的。
    }

     …..
    return 0;
}
```

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。

```
//event.c文件
int
event_config_set_flag(struct event_config *cfg, int flag)
{
    if (!cfg)
        return -1;
    cfg->flags |= flag;
    return 0;
}
```

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。

```
const char **event_get_supported_methods(void);
const char *event_base_get_method(const struct event_base *);
int event_base_get_features(const struct event_base *base);
static int  event_config_is_avoided_method(const struct event_config *cf
g, 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。

```
#include<event2/event.h>
#include<stdio.h>

#ifdef WIN32
#include<WinSock2.h>
#endif

int main()
{
#ifdef WIN32
    WSADATA wsa_data;
    WSAStartup(0x0201, &wsa_data);
#endif

    const char** all_methods = event_get_supported_methods();

    while( all_methods && *all_methods )
    {
        printf("%s\t", *all_methods++);
    }

    printf("\n");

    event_base *base = event_base_new();
    if( base )
        printf("current method:\t %s\n", event_base_get_method(base) );
    else
        printf("base == NULL\n");

#ifdef WIN32
    WSACleanup();
#endif

    return 0;
}
```

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

11. 跨平台Reactor接口的实现

原文地址

之前的文章讲了怎么实现线程、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：

```
//event-internal.h文件
struct event_base {
    const struct eventop *evsel;
    void *evbase;

    ...
};
    struct eventop {
    const char *name; //多路IO复用函数的名字

    void *(*init)(struct event_base *);

    int (*add)(struct event_base *, evutil_socket_t fd, short old, short
events, void *fdinfo);
    int (*del)(struct event_base *, evutil_socket_t fd, short old, short
events, void *fdinfo);
    int (*dispatch)(struct event_base *, struct timeval *);
    void (*dealloc)(struct event_base *);

    int need_reinit; //是否要重新初始化
    //多路IO复用的特征。参考http://blog.csdn.net/luotuo44/article/details/384
43569
    enum event_method_feature features;
    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
};
```

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：

```
//select.c文件
static void *select_init(struct event_base *);
static int select_add(struct event_base *, int, short old, short events,
void*);
static int select_del(struct event_base *, int, short old, short events,
void*);
static int select_dispatch(struct event_base *, struct timeval *);
static void select_dealloc(struct event_base *);

const struct eventop selectops = {
    "select",
    select_init,
    select_add,
    select_del,
    select_dispatch,
    select_dealloc,
    0, /* doesn't need reinit. */
    EV_FEATURE_FDS,
    0,
};

//poll.c文件
static void *poll_init(struct event_base *);
static int poll_add(struct event_base *, int, short old, short events, vo
id *_idx);
static int poll_del(struct event_base *, int, short old, short events, vo
id *_idx);
static int poll_dispatch(struct event_base *, struct timeval *);
static void poll_dealloc(struct event_base *);

const struct eventop pollops = {
    "poll",
    poll_init,
    poll_add,
    poll_del,
    poll_dispatch,
    poll_dealloc,
    0, /* doesn't need_reinit */
    EV_FEATURE_FDS,
    sizeof(struct pollidx),
};
```

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：

```c
//event.c文件
#ifdef _EVENT_HAVE_EVENT_PORTS
extern const struct eventop evportops;
#endif
#ifdef _EVENT_HAVE_SELECT
extern const struct eventop selectops;
#endif
#ifdef _EVENT_HAVE_POLL
extern const struct eventop pollops;
#endif
#ifdef _EVENT_HAVE_EPOLL
extern const struct eventop epollops;
#endif
#ifdef _EVENT_HAVE_WORKING_KQUEUE
extern const struct eventop kqops;
#endif
#ifdef _EVENT_HAVE_DEVPOLL
extern const struct eventop devpollops;
#endif
#ifdef WIN32
extern const struct eventop win32ops;
#endif

/* Array of backends in order of preference. */
static const struct eventop *eventops[] = {
#ifdef _EVENT_HAVE_EVENT_PORTS
    &evportops,
#endif
#ifdef _EVENT_HAVE_WORKING_KQUEUE
    &kqops,
#endif
#ifdef _EVENT_HAVE_EPOLL
    &epollops,
#endif
#ifdef _EVENT_HAVE_DEVPOLL
    &devpollops,
#endif
#ifdef _EVENT_HAVE_POLL
    &pollops,
#endif
#ifdef _EVENT_HAVE_SELECT
    &selectops,
#endif
#ifdef WIN32
    &win32ops,
#endif
    NULL
};
```

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》。

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：

```c
//event.c文件
struct event_base *
event_base_new_with_config(const struct event_config *cfg)
{
    int i;
    struct event_base *base;
    int should_check_environment;

    //分配并清零event_base内存. event_base里的所有成员都会为0
    if ((base = mm_calloc(1, sizeof(struct event_base))) == NULL) {
        event_warn("%s: calloc", __func__);
        return NULL;
    }

    ...
    should_check_environment =
        !(cfg && (cfg->flags & EVENT_BASE_FLAG_IGNORE_ENV));
        //遍历数组的元素
    for (i = 0; eventops[i] && !base->evbase; i++) {
        if (cfg != NULL) {
            /* determine if this backend should be avoided */
            if (event_config_is_avoided_method(cfg,
                eventops[i]->name))
                continue;
            if ((eventops[i]->features & cfg->require_features)
                != cfg->require_features)
                continue;
        }

        /* also obey the environment variables */
        if (should_check_environment &&
            event_is_method_disabled(eventops[i]->name))
            continue;

        //找到了一个满足条件的多路IO复用函数
        base->evsel = eventops[i];

        //初始化evbase，后面会说到
        base->evbase = base->evsel->init(base);
    }

    if (base->evbase == NULL) {
        event_warnx("%s: no event mechanism available",
            __func__);
        base->evsel = NULL;
        event_base_free(base);
        return NULL;
    }

    ....
```

```
    return (base);
}
```

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》。

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：

```
//select.c文件
struct selectop {
    int event_fds;      /* Highest fd in fd set */
    int event_fdsz;
    int resize_out_sets;
    fd_set *event_readset_in;
    fd_set *event_writeset_in;
    fd_set *event_readset_out;
    fd_set *event_writeset_out;
};

//poll.c文件
struct pollop {
    int event_count;        /* Highest number alloc */
    int nfds;           /* Highest number used */
    int realloc_copy;       /* True iff we must realloc
                     * event_set_copy */
    struct pollfd *event_set;
    struct pollfd *event_set_copy;
};
```

前面event_base_new_with_config的代码中，有下面一行代码：

```
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：

```
//poll.c文件
static void *
poll_init(struct event_base *base)
{
    struct pollop *pollop;

    if (!(pollop = mm_calloc(1, sizeof(struct pollop))))
        return (NULL);

    evsig_init(base);//其他的一些初始化

    return (pollop);
}
```

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：

```
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。

12.Libevent工作流程探究

原文地址

之前的小节讲了很多Libevent的基础构件，现在以一个实际例子来初步探究Libevent的基本工作流程。由于还有很多Libevent的细节并没有讲所以，这里的探究还是比较简洁，例子也相当简单。

```c
#include<unistd.h>
#include<stdio.h>
#include<event.h>
#include<thread.h>


void cmd_cb(int fd, short events, void *arg)
{
    char buf[1024];
    printf("in the cmd_cb\n");

    read(fd, buf, sizeof(buf));
}


int main()
{
    evthread_use_pthreads();

    //使用默认的event_base配置
    struct event_base *base = event_base_new();

    struct event *cmd_ev = event_new(base, STDIN_FILENO,
                                     EV_READ | EV_PERSIST, cmd_cb, NULL);


    event_add(cmd_ev, NULL); //没有超时

    event_base_dispatch(base);

    return 0;
}
```

上面代码估计是不会比读者写的第一个Libevent程序复杂。但这已经包含了Libevent的基础工作流程。这里将进入这些函数的内部探究，并且只会讲解之前博文出现过的，没出现的，尽量不讲。在讲解之前，要先了解一下struct event这个结构体。

event结构体

```
struct event {
    TAILQ_ENTRY(event) ev_active_next; //激活队列
    TAILQ_ENTRY(event) ev_next; //注册事件队列
    /* for managing timeouts */
    union {
        TAILQ_ENTRY(event) ev_next_with_common_timeout;
        int min_heap_idx; //指明该event结构体在堆的位置
    } ev_timeout_pos; //仅用于定时事件处理器(event).EV_TIMEOUT类型

    //对于I/O事件，是文件描述符；对于signal事件，是信号值
    evutil_socket_t ev_fd;

    struct event_base *ev_base; //所属的event_base

    //因为信号和I/O是不能同时设置的。所以可以使用共用体以省内存
    //在低版本的Libevent，两者是分开的，不在共用体内。
    union {
    //无论是信号还是IO，都有一个TAILQ_ENTRY的队列。它用于这样的情景：
    //用户对同一个fd调用event_new多次，并且都使用了不同的回调函数。
    //每次调用event_new都会产生一个event*。这个xxx_next成员就是把这些
    //event连接起来的。

        /* used for io events */
        //用于IO事件
        struct {
            TAILQ_ENTRY(event) ev_io_next;
            struct timeval ev_timeout;
        } ev_io;

        /* used by signal events */
        //用于信号事件
        struct {
            TAILQ_ENTRY(event) ev_signal_next;
            short ev_ncalls; //事件就绪执行时，调用ev_callback的次数          /
* Allows deletes in callback */
            short *ev_pncalls; //指针，指向次数
        } ev_signal;
    } _ev;

    short ev_events;//记录监听的事件类型 EV_READ EVTIMEOUT之类
    short ev_res;       /* result passed to event callback *///记录了当前激
活事件的类型
    //libevent用于标记event信息的字段，表明其当前的状态.
    //可能值为前面的EVLIST_XXX
    short ev_flags;

    //本event的优先级。调用event_priority_set设置
    ev_uint8_t ev_pri;
    ev_uint8_t ev_closure;
    struct timeval ev_timeout;//用于定时器,指定定时器的超时值
```

```
        /* allows us to adopt for different types of events */
        void (*ev_callback)(evutil_socket_t, short, void *arg); //回调函数
        void *ev_arg; //回调函数的参数
    };
```

event结构体里面有几个TAILQ_ENTRY队列节点类型。这里因为一个event是会同时处于多个队列之中。比如前几篇博文说到的同一个文件描述符或者信号值对应的多个event会被连在一起，所有的被加入到event_base的event也会连在一起，所有被激活的event也会被连在一起。所以会有多个QAILQ_ENTRY。

event结构体只有一两个之前没有说到的概念，这不妨碍理解event结构体。而event_base结构体则会太多之前没有说到的概念，所以这里就不贴出event_base的代码了。

~在读这篇博文前，最好读一下前面几篇博文，因为会用到其他讲到的东西。如果之前有讲过的东西，这里也将一笔带过。~

好了，开始探究。

最前面的evthread_use_pthreads();就不多说了，看《多线程、锁、条件变量(一)》和《多线程、锁、条件变量(二)》这两个小节吧。

创建event_base

下面看一下event_base_new函数。它是由event_base_new_with_config函数实现的。我们还是看后面那个函数吧。

```
//event.c文件
struct event_base *
event_base_new_with_config(const struct event_config *cfg)
{
    int i;
    struct event_base *base;
    int should_check_environment;


    //之所以不用mm_malloc是因为mm_malloc并不会清零该内存区域。
    //而这个函数是会清零申请到的内存区域，这相当于被base初始化
    if ((base = mm_calloc(1, sizeof(struct event_base))) == NULL) {
        event_warn("%s: calloc", __func__);
        return NULL;
    }

    ...

    TAILQ_INIT(&base->eventqueue);

    ...

    if (cfg)
        base->flags = cfg->flags;

    evmap_io_initmap(&base->io);
    evmap_signal_initmap(&base->sigmap);

    base->evbase = NULL;

    should_check_environment =
        !(cfg && (cfg->flags & EVENT_BASE_FLAG_IGNORE_ENV));

    //选择IO复用结构体
    for (i = 0; eventops[i] && !base->evbase; i++) {
        if (cfg != NULL) {
            /* determine if this backend should be avoided */
            if (event_config_is_avoided_method(cfg,
                eventops[i]->name))
                continue;
            if ((eventops[i]->features & cfg->require_features)
                != cfg->require_features)
                continue;
        }

        if (should_check_environment &&
            event_is_method_disabled(eventops[i]->name))
            continue;

        //找到一个满足条件的多路IO复用函数
        base->evsel = eventops[i];
```

```
        //初始化ev_base。并且会对信号监听的处理也进行初始化
        base->evbase = base->evsel->init(base);
    }



#ifndef _EVENT_DISABLE_THREAD_SUPPORT
    //测试evthread_lock_callbacks结构中的lock指针函数是否为NULL
    //即测试Libevent是否已经初始化为支持多线程模式。
    //由于一开始是用mm_calloc申请内存的，所以该内存区域的值为0
    //对于th_base_lock变量，目前的值为NULL.
    if (EVTHREAD_LOCKING_ENABLED() &&
        (!cfg || !(cfg->flags & EVENT_BASE_FLAG_NOLOCK))) { //配置是支持锁的
        EVTHREAD_ALLOC_LOCK(base->th_base_lock,
            EVTHREAD_LOCKTYPE_RECURSIVE); //申请一个锁
        base->defer_queue.lock = base->th_base_lock;
        EVTHREAD_ALLOC_COND(base->current_event_cond);//申请一个条件变量
    }
#endif

    return (base);
}
```

这里用到了event_config结构体，关于这个结构体可以参考《配置event_base》小节。这个结构体主要是对event_base进行一些配置。另外代码中还讲到了怎么使用选择一个多IO复用函数，这个可以参考《跨平台Reactor接口的实现》小节。

宏EVTHREAD_LOCKING_ENABLED主要是检测是否已经支持锁了。检测的方式也很简单，也就是检测_evthread_lock_fns全局变量中的lock成员变量是否不为NULL。有关这个_evthread_lock_fns全局变量可以查看《多线程、锁、条件变量(一)》小节。

创建event

好了，现在event_base已经新建出来了。下面看一下event_new函数，它和前面的event_base_new一样，把主要是的初始化工作交给另一个函数。event_new函数的工作只是创建一个struct event结构体，然后把它的参数原封不动地传给event_assign，所以还是看event_assign函数。

```
//event.c文件
int
event_assign(struct event *ev, struct event_base *base, evutil_socket_t f
d,
                  short events, void (*callback)(evutil_socket_t, short,
void *), void *arg)
{
    //进行一些赋值和初始化。
    ev->ev_base = base;
    ev->ev_callback = callback;
    ev->ev_arg = arg;
    ev->ev_fd = fd;
    ev->ev_events = events;
    ev->ev_res = 0;
    ev->ev_flags = EVLIST_INIT; //初始化状态
    ev->ev_ncalls = 0;
    ev->ev_pncalls = NULL;

    if (events & EV_SIGNAL) {
        if ((events & (EV_READ|EV_WRITE)) != 0) {
            event_warnx("%s: EV_SIGNAL is not compatible with "
                "EV_READ or EV_WRITE", __func__);
            return -1;
        }
    }

    ...

    return 0;
}
```

从event_assign函数的名字可以得知它是进行赋值操作的。所以它能可以在event被初始化后再次调用。不过，初始化后再次调用的话，有些事情要注意。这个在后面的博客中会说到。

从上面的代码可看到：**如果这个event是用来监听一个信号的，那么就不能让这个event监听读或者写事件**。原因是其与信号event的实现方法相抵触，具体可以参考《信号event的处理》小节。

注意，此时event结构体的变量ev_flags的值是EVLIST_INIT。对变量的追踪是很有帮助的。它指明了event结构体的状态。它通过以或运算的方式取下面的值：

```
//event_struct.h文件
#define EVLIST_TIMEOUT  0x01 //event从属于定时器队列或者时间堆
#define EVLIST_INSERTED 0x02 //event从属于注册队列
#define EVLIST_SIGNAL   0x04 //没有使用
#define EVLIST_ACTIVE   0x08 //event从属于活动队列
#define EVLIST_INTERNAL 0x10 //该event是内部使用的。信号处理时有用到
#define EVLIST_INIT 0x80 //event已经被初始化了

/* EVLIST_X_ Private space: 0x1000-0xf000 */
#define EVLIST_ALL  (0xf000 | 0x9f) //所有标志。这个不能取
```

将event加入到event_base中

创建完一个event结构体后，现在看一下event_add。它同前面的函数一样，内部也是调用其他函数完成工作。因为它用到了锁，所以给出它的代码：

```
//event.c文件
int
event_add(struct event *ev, const struct timeval *tv)
{
    int res;

    //加锁
    EVBASE_ACQUIRE_LOCK(ev->ev_base, th_base_lock);
    res = event_add_internal(ev, tv, 0);
    //解锁
    EVBASE_RELEASE_LOCK(ev->ev_base, th_base_lock);
    return (res);
}

static inline int
event_add_internal(struct event *ev, const struct timeval *tv,
    int tv_is_absolute)
{
    struct event_base *base = ev->ev_base;
    int res = 0;
    int notify = 0;
    ...
    if ((ev->ev_events & (EV_READ|EV_WRITE|EV_SIGNAL)) &&
        !(ev->ev_flags & (EVLIST_INSERTED|EVLIST_ACTIVE))) {
        if (ev->ev_events & (EV_READ|EV_WRITE))
            res = evmap_io_add(base, ev->ev_fd, ev); //加入io队列
        else if (ev->ev_events & EV_SIGNAL)
            res = evmap_signal_add(base, (int)ev->ev_fd, ev);//加入信号队列
        if (res != -1)
            event_queue_insert(base, ev, EVLIST_INSERTED);//向event_base注
册事件
    }
    ...
    return (res);
}
```

event_add函数只是对event_base加了锁，然后调用event_add_internal函数完成工作。所以函数event_add是线程安全的。

event_add_internal函数会调用前几篇博文讲到的evmap_io_add和evmap_signal_add，把有相同文件描述符fd和信号值sig的event连在一个队列里面。成功之后，就会调用event_queue_insert，向event_base注册事件。

前面小节中的evmap_io_add和evmap_signal_add函数内部还有一些地方并没有说到。那就是把要监听的fd或者sig添加到多路IO复用函数中，使得其是可以监听的。

```
//evmap.c文件
int
evmap_io_add(struct event_base *base, evutil_socket_t fd, struct event *e
v)
{
    const struct eventop *evsel = base->evsel;
    struct event_io_map *io = &base->io;
    struct evmap_io *ctx = NULL;
    int nread, nwrite, retval = 0;
    short res = 0, old = 0;
    struct event *old_ev;

    ...

    //GET_IO_SLOT_AND_CTOR宏的作用就是让ctx指向struct event_map_entry结构体中
的TAILQ_HEAD
    //宏的展开，可以到http://blog.csdn.net/luotuo44/article/details/38403241
查看
    GET_IO_SLOT_AND_CTOR(ctx, io, fd, evmap_io, evmap_io_init,
                         evsel->fdinfo_len);

    //同一个fd可以调用event_new,event_add
    //多次。nread、nwrite就是记录有多少次。如果每次event_new的回调函数
    //都不一样，那么当fd有可读或者可写时，这些回调函数都是会触发的。
    //对一个fd不能event_new、event_add太多次的。后面会进行判断
    nread = ctx->nread;
    nwrite = ctx->nwrite;

    if (nread)
        old |= EV_READ;
    if (nwrite)
        old |= EV_WRITE;

    if (ev->ev_events & EV_READ) {
        //记录是不是第一次。如果是第一次，那么就说明该fd还没被
        //加入到多路IO复用中。即还没被加入到像select、epoll这些
        //函数中。那么就要加入。这个在后面可以看到。
        if (++nread == 1)
            res |= EV_READ;
    }
    if (ev->ev_events & EV_WRITE) {
        if (++nwrite == 1)
            res |= EV_WRITE;
    }
    if (EVUTIL_UNLIKELY(nread > 0xffff || nwrite > 0xffff)) {
        event_warnx("Too many events reading or writing on fd %d",
                    (int)fd);
        return -1;
    }
```

```
    //把fd加入到多路IO复用中。
    if (res) {
        void *extra = ((char*)ctx) + sizeof(struct evmap_io);
        if (evsel->add(base, ev->ev_fd,
                       old, (ev->ev_events & EV_ET) | res, extra) == -1)
            return (-1);
        retval = 1;
    }

    //nread进行了++。把次数记录下来。下次对于同一个fd，这个次数就有用了
    ctx->nread = (ev_uint16_t) nread;
    ctx->nwrite = (ev_uint16_t) nwrite;

    TAILQ_INSERT_TAIL(&ctx->events, ev, ev_io_next);

    return (retval);
}
```

代码中有两个计数nread和nwrite，当其值为1时，就说明是第一次监听对应的事件。此时，就要把这个fd添加到多路IO复用函数中。这就完成fd与select、poll、epoll之类的多路IO复用函数的相关联。这完成对fd监听的第一步。

下面再看event_queue_insert函数的实现。

```
//event.c文件
static void
event_queue_insert(struct event_base *base, struct event *ev, int queue)

{
    ...

    ev->ev_flags |= queue;
    switch (queue) {
    case EVLIST_INSERTED:
        TAILQ_INSERT_TAIL(&base->eventqueue, ev, ev_next);
        break;
        ...
    }
}
```

这个函数的主要作为是把event加入到对应的队列中。在这里，是为了把event加入到eventqueue这个已注册队列中，即将event向event_base注册。**注意，此时event结构体的ev_flags变量为EVLIST_INIT | EVLIST_INSERTED了。**

进入主循环，开始监听event

现在事件已经添加完毕，开始进入主循环event_base_dispatch函数。还是同样，该函数内部调用event_base_loop完成工作。

```
//event.c文件
int
event_base_loop(struct event_base *base, int flags)
{
    const struct eventop *evsel = base->evsel;
    int res, done, retval = 0;

    //加锁
    EVBASE_ACQUIRE_LOCK(base, th_base_lock);

    done = 0;

    while (!done) {
        //该函数的内部会解锁，然后调用OS提供的的多路IO复用函数。
        //这个函数退出后，又会立即加锁。这有点像条件变量。
        res = evsel->dispatch(base, tv_p);

        if (N_ACTIVE_CALLBACKS(base)) {
            int n = event_process_active(base);
        }
    }

done:
    //解锁
    EVBASE_RELEASE_LOCK(base, th_base_lock);
    return (retval);
}
```

在event_base_loop函数内部会进行加锁，这是因为这里要对event_base里面的多个队列进行一些数据操作(增删操作)，此时要用锁来保护队列不被另外一个线程所破坏。

上面代码中有两个函数evsel->dispatch和event_process_active。前一个将调用多路IO复用函数，对event进行监听，并且把满足条件的event放到event_base的激活队列中。后一个则遍历这个激活队列的所有event，逐个调用对应的回调函数。

可以看到整个流程如下图所示：

将已激活event插入到激活列表

我们还是深入看看Libevent是怎么把event添加到激活队列的。dispatch是一个函数指针，它的实现都包含是一个多路IO复用函数。这里选择poll这个多路IO复用函数来作分析。

```c
//poll.c文件
static int
poll_dispatch(struct event_base *base, struct timeval *tv)
{
    int res, i, j, nfds;
    long msec = -1;
    struct pollop *pop = base->evbase;
    struct pollfd *event_set;

    nfds = pop->nfds;

    event_set = pop->event_set;

    //解锁
    EVBASE_RELEASE_LOCK(base, th_base_lock);
    res = poll(event_set, nfds, msec);
    //再次加锁
    EVBASE_ACQUIRE_LOCK(base, th_base_lock);

    ...

    i = random() % nfds;
    for (j = 0; j < nfds; j++) {
        int what;
        if (++i == nfds)
            i = 0;
        what = event_set[i].revents;
        if (!what)
            continue;

        res = 0;

        //如果fd发生错误，就把之当作读和写事件。之后调用read
        //或者write时，就能得知具体是什么错误了。这里的作用是
        //通知到上层。
        if (what & (POLLHUP|POLLERR))
            what |= POLLIN|POLLOUT;

        if (what & POLLIN)
            res |= EV_READ;
        if (what & POLLOUT)
            res |= EV_WRITE;
        if (res == 0)
            continue;

        //把这个ev放到激活队列中。
        evmap_io_active(base, event_set[i].fd, res);
    }

    return (0);
}
```

pollfd数组的数据是在evmap_io_add函数中添加的，在evmap_io_add函数里面，有一个evsel->add调用，它会把数据(fd和对应的监听类型)放到pollfd数组中。

当主线程从poll返回时，没有错误的话，就说明有些监听的事件发生了。在函数的后面，它会遍历这个pollfd数组，查看哪个fd是有事件发生。如果事件发生，就调用evmap_io_active(base, event_set[i].fd, res);在这个函数里面会把这个fd对应的event放到event_base的激活event队列中。下面是evmap_io_active的代码。

```
void //evmap.c文件
evmap_io_active(struct event_base *base, evutil_socket_t fd, short event
s)
{
    struct event_io_map *io = &base->io;
    struct evmap_io *ctx;
    struct event *ev;

    //由这个fd找到对应event_map_entry的TAILQ_HEAD.
    GET_IO_SLOT(ctx, io, fd, evmap_io);

    //遍历这个队列。将所有与fd相关联的event结构体都处理一遍
    TAILQ_FOREACH(ev, &ctx->events, ev_io_next) {
        if (ev->ev_events & events)
            event_active_nolock(ev, ev->ev_events & events, 1);
    }
}

void //event.c文件
event_active_nolock(struct event *ev, int res, short ncalls)
{
    struct event_base *base;
    base = ev->ev_base;

    ...
    //将ev插入到激活队列
    event_queue_insert(base, ev, EVLIST_ACTIVE);

    ...
}


//将event 插入到event_base的对应(由queue指定)的队列里面
static void //event.c文件
event_queue_insert(struct event_base *base, struct event *ev, int queue)
{
    ...

    ev->ev_flags |= queue;
    switch (queue) {
    case EVLIST_ACTIVE:
        base->event_count_active++;
        //将event插入到对应对应优先级的激活队列中
        TAILQ_INSERT_TAIL(&base->activequeues[ev->ev_pri],
            ev,ev_active_next);
        break;
    }
}
```

经过上面三个函数的调用，就可以把一个fd对应的所有符合条件的event插入到激活队列中。因为Libevent还对事件处理设有优先级，所以有一个激活数组队列，而不是只有一个激活队列。

注意，此时event结构体的ev_flags变量为EVLIST_INIT | EVLIST_INSERTED | EVLIST_ACTIVE了。

处理激活列表中的event

现在已经完成了将event插入到激活队列中。接下来就是遍历激活数组队列，把所有激活的event都处理即可。

现在来追踪event_process_active函数。

```
//event.c文件
static int
event_process_active(struct event_base *base)
{
    struct event_list *activeq = NULL;
    int i, c = 0;

    //从高优先级到低优先级遍历优先级数组
    for (i = 0; i < base->nactivequeues; ++i) {
        //对于特定的优先级，遍历该优先级的所有激活event
        if (TAILQ_FIRST(&base->activequeues[i]) != NULL) {
            activeq = &base->activequeues[i];
            c = event_process_active_single_queue(base, activeq);
            ...
        }
    }
    return c;
}

static int
event_process_active_single_queue(struct event_base *base,
    struct event_list *activeq)
{
    struct event *ev;
    int count = 0;

    for (ev = TAILQ_FIRST(activeq); ev; ev = TAILQ_FIRST(activeq)) {
        //如果是永久事件，那么只需从active队列中删除。
        if (ev->ev_events & EV_PERSIST)
            event_queue_remove(base, ev, EVLIST_ACTIVE);
        else //不是的话，那么就要把这个event删除掉。
            event_del_internal(ev);
        if (!(ev->ev_flags & EVLIST_INTERNAL))
            ++count;

        //下面开始处理这个event
        switch (ev->ev_closure) {
        ...
        case EV_CLOSURE_NONE:
            //调用用户设置的回调函数。
            (*ev->ev_callback)(ev->ev_fd, ev->ev_res, ev->ev_arg);
            break;
        }

        EVBASE_ACQUIRE_LOCK(base, th_base_lock);

    }
    return count;
}
```

上面的代码，从高到低优先级遍历激活event优先级数组。对于激活的event，要调用event_queue_remove将之从激活队列中删除掉。然后再对这个event调用其回调函数。

event_queue_remove函数的调用会改变event结构体的ev_flags变量的值。调用后， ev_flags变量为EVLIST_INIT | EVLIST_INSERTED。现在又可以等待下一次事件的到来了。

13.event优先级设置

原文地址

event_base允许用户对它里面的event设置优先级，这样可以使得有些更重要的event能够得到优先处理。

Libevent实现优先级功能的方法是：用一个激活队列数组来存放激活event。即数组的元素是一个激活队列，所以有多个激活队列。并且规定不同的队列有不同的优先级。

可以通过event_base_priority_init函数设置event_base的优先级个数，该函数实现如下：

```
//event.c文件
int
event_base_priority_init(struct event_base *base, int npriorities)
{
    int i;

    //由N_ACTIVE_CALLBACKS宏可以知道，本函数应该要在event_base_dispatch
    //函数调用前调用。不然将无法设置。
    if (N_ACTIVE_CALLBACKS(base) || npriorities < 1
        || npriorities >= EVENT_MAX_PRIORITIES)
        return (-1);

    //之前和现在要设置的优先级数是一样的。
    if (npriorities == base->nactivequeues)
        return (0);

    //释放之前的，因为N_ACTIVE_CALLBACKS,所以没有active的event。
    //可以随便mm_free
    if (base->nactivequeues) {
        mm_free(base->activequeues);
        base->nactivequeues = 0;
    }

    //分配一个优先级数组。
    base->activequeues = (struct event_list *)
      mm_calloc(npriorities, sizeof(struct event_list));
    if (base->activequeues == NULL) {
        event_warn("%s: calloc", __func__);
        return (-1);
    }
    base->nactivequeues = npriorities;

    for (i = 0; i < base->nactivequeues; ++i) {
        TAILQ_INIT(&base->activequeues[i]);
    }

    return (0);
}
```

从前面一个判断可知，因为event_base_dispatch函数会改动激活事件的个数，即会使得N_ACTIVE_CALLBACKS(base)为真。所以event_base_priority_init函数要在event_base_dispatch函数之前调用。此外要设置的优先级个数，要小于EVENT_MAX_PRIORITIES。这个宏是在event.h文件中定义，在2.0.21版本中，该宏被定义成256。在调用event_base_new得到的event_base只有一个优先级，也就是所有event都是同级的。

上面的代码调用mm_alloc分配了一个优先级数组。不同优先级的event会被放到数组的不同位置上(下面可以看到这一点)。这样就可以区分不同event的优先级了。以后处理event时，就可以从高优先级到低优先级处理event。

上面是设置event_base的优先级个数。现在来看一下怎么设置event的优先级。可以通过event_priority_set函数设置，该函数如下：

```
//event.c文件
int
event_priority_set(struct event *ev, int pri)
{
    _event_debug_assert_is_setup(ev);

    if (ev->ev_flags & EVLIST_ACTIVE)
        return (-1);

//优先级不能越界
    if (pri < 0 || pri >= ev->ev_base->nactivequeues)
        return (-1);

    //pri值越小，其优先级就越高。
    ev->ev_pri = pri;

    return (0);
}
```

在上面代码的第一个判断中，可以知道当event的状态是EVLIST_ACTIVE时，就不能对这个event进行优先级设置了。因此，如果要对event进行优先级设置，那么得在调用event_base_dispatch函数之前。因为一旦调用了event_base_dispatch，那么event就随时可能变成EVLIST_ACTIVE状态。

现在看一下一个event是怎么插入到event_base的优先级数组中。

```
//event.c文件
//event_active_nolock以event_queue_insert(base, ev, EVLIST_ACTIVE);方式调用
static void
event_queue_insert(struct event_base *base, struct event *ev, int queue)
{
    ....

    ev->ev_flags |= queue; //加入EVLIST_ACTIVE状态
    switch (queue) {
    ...
    case EVLIST_ACTIVE:
        base->event_count_active++;
        //从优先级数组中找到对应的优先级
        TAILQ_INSERT_TAIL(&base->activequeues[ev->ev_pri],
            ev,ev_active_next);
        break;
    ...
    }
}
```

最后，我们来看一下默认的event优先级是多少。想必大家都能想到这个默认优先级是在新建event结构体时设置的。不错，看下面的event_assign函数。

```
//event.c文件  由event_new函数调用本函数
int
event_assign(struct event *ev, struct event_base *base, evutil_socket_t f
d,
                  short events, void (*callback)(evutil_socket_t, short,
void *), void *arg)
{
    ...

    ev->ev_base = base;
    ev->ev_callback = callback;
    ev->ev_arg = arg;
    ev->ev_fd = fd;
    ev->ev_events = events;
    ev->ev_res = 0;
    ev->ev_flags = EVLIST_INIT;
    ev->ev_ncalls = 0;
    ev->ev_pncalls = NULL;

    ....

    if (base != NULL) {
        /* by default, we put new events into the middle priority */
        ev->ev_pri = base->nactivequeues / 2;//默认优先级
    }

    ...
}
```

在这个函数里面，对event的成员变量进行了一些设置。其中，优先级的设置值为优先级数组长度的一半，所以是中间优先级。

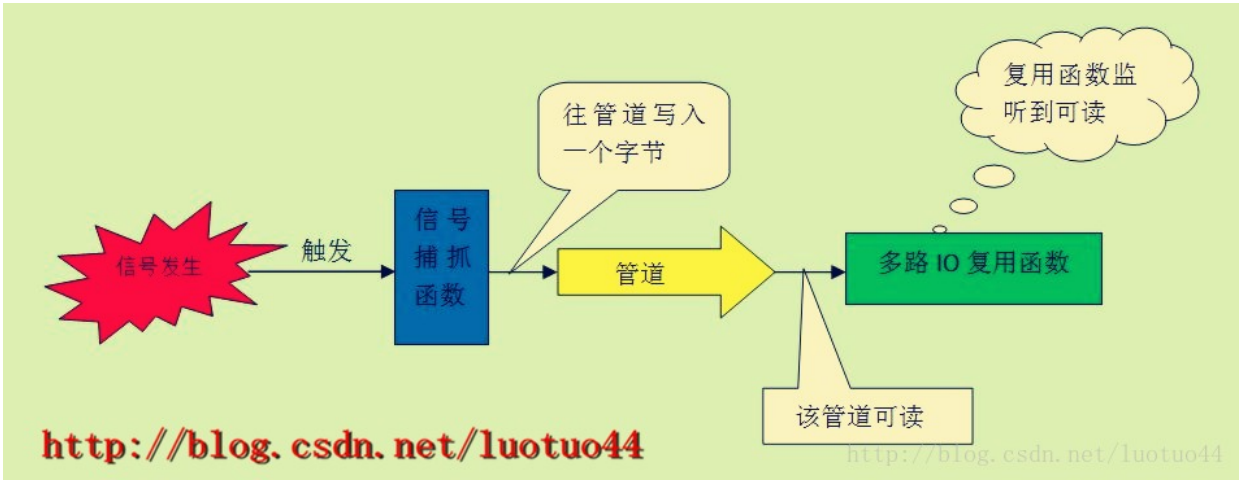
14.信号event的处理

原文地址

信号event的工作原理

前面讲解了Libevent如何监听一个IO事件，现在来讲一下Libevent如何监听信号。Libevent对于信号的处理是采用统一事件源的方式。简单地说，就是把信号也转换成IO事件，集成到Libevent中。

统一事件源的工作原理如下：假如用户要监听SIGINT信号，那么在实现的内部就对SIGINT这个信号设置捕抓函数。此外，在实现的内部还要建立一条管道(pipe)，并把这个管道加入到多路IO复用函数中。当SIGINT这个信号发生后，捕抓函数将会被调用。而这个捕抓函数的工作就是往管道写入一个字符(这个字符往往等于所捕抓到信号的信号值)。此时，这个管道就变成是可读的了，多路IO复用函数能检测到这个管道变成可读的了。换言之，多路IO复用函数检测到SIGINT信号的发生，也就完成了对信号的监听工作。这个过程如下图所示：



了解完统一事件源的工作原理，现在来看一下Libevent具体的实现细节。按照上述的介绍，内部实现的工作有：

1. 创建一个管道(Libevent实际上使用的是socketpair)
2. 为这个socketpair的一个读端创建一个event，并将之加入到多路IO复用函数的监听之中
3. 设置信号捕抓函数
4. 有信号发生，就往socketpair写入一个字节

统一事件源能够工作的一个原因是：多路IO复用函数都是可中断的。即处理完信号后，会从多路IO复用函数中退出，并将errno赋值为EINTR。有些OS的某些系统调用，比如Linux的read，即使被信号终端了，还是会自启动的。即不会从read函数中退出来。

用于信号event的结构体和变量

event_base为信号监听提供了的成员如下：

```
//event-internal.h文件
struct event_base {

    const struct eventop *evsigsel;
    struct evsig_info sig;

    ...
    struct event_signal_map sigmap;
    ...
};


//evsignal-internal.h文件
struct evsig_info {
    //用于监听socketpair读端的event. ev_signal_pair[1]为读端
    struct event ev_signal;
    //socketpair
    evutil_socket_t ev_signal_pair[2];
    //用来标志是否已经将ev_signal这个event加入到event_base中了
    int ev_signal_added;
    //用户一共要监听多少个信号
    int ev_n_signals_added;

    //数组。用户可能已经设置过某个信号的信号捕抓函数。但
    //Libevent还是要为这个信号设置另外一个信号捕抓函数，
    //此时，就要保存用户之前设置的信号捕抓函数。当用户不要
    //监听这个信号时，就能够恢复用户之前的捕抓函数。
    //因为是有多个信号，所以得用一个数组保存。
#ifdef _EVENT_HAVE_SIGACTION
    struct sigaction **sh_old;
#else//保存的是捕抓函数的函数指针，又因为是数组。所以是二级指针
    ev_sighandler_t **sh_old;
#endif
    /* Size of sh_old. */
    int sh_old_max; //数组的长度
};
```

在上面代码中，已经可以看到用于socketpair的ev_signal_pair变量，还有struct event结构体变量ev_signal。那么Libevent是在何时创建socketpair以及如何将socketpair和ev_signal相关联的呢？

初始化

在前面的小节中《跨平台Reactor接口的实现》中，介绍了Libevent是如何选择一个多路IO复用函数的。在选定一个多路IO复用函数后，就会调用下面一行代码：

```
base->evbase = base->evsel->init(base);
```

这是初始化代码函数。下面给出poll的init函数。

```
//poll.c文件
static void *
poll_init(struct event_base *base)
{
    struct pollop *pollop;

    if (!(pollop = mm_calloc(1, sizeof(struct pollop))))
        return (NULL);

    evsig_init(base);

    return (pollop);
}
```

可以看到，其调用了evsig_init函数。而正是这个evsig_init函数完成了创建socketpair并将socketpair的一个读端与ev_signal相关联。

```
//signal.c文件
int
evsig_init(struct event_base *base)
{
    //创建一个socketpair
    if (evutil_socketpair(
            AF_UNIX, SOCK_STREAM, 0, base->sig.ev_signal_pair) == -1) {
#ifdef WIN32
        /* Make this nonfatal on win32, where sometimes people
           have localhost firewalled. */
        event_sock_warn(-1, "%s: socketpair", __func__);
#else
        event_sock_err(1, -1, "%s: socketpair", __func__);
#endif
        return -1;
    }

    //子进程不能访问该socketpair
    evutil_make_socket_closeonexec(base->sig.ev_signal_pair[0]);
    evutil_make_socket_closeonexec(base->sig.ev_signal_pair[1]);
    base->sig.sh_old = NULL;
    base->sig.sh_old_max = 0;


    evutil_make_socket_nonblocking(base->sig.ev_signal_pair[0]);
    evutil_make_socket_nonblocking(base->sig.ev_signal_pair[1]);

    //将ev_signal_pair[1]与ev_signal这个event相关联。ev_signal_pair[1]为读端
    //该函数的作用等同于event_new。实际上event_new内部也是调用event_assign函数完
成工作的
    event_assign(&base->sig.ev_signal, base, base->sig.ev_signal_pair[1],
        EV_READ | EV_PERSIST, evsig_cb, base);

    //标明是内部使用的
    base->sig.ev_signal.ev_flags |= EVLIST_INTERNAL;
    //Libevent中，event是有优先级的。前一篇博文已经说到这一点
    event_priority_set(&base->sig.ev_signal, 0); //最高优先级

    base->evsigsel = &evsigops;

    return 0;
}
```

socketpair的两个端都调用evutil_make_socket_closeonexec，因为不能让子进程可以访问的这个socketpair。因为子进程的访问可能会出现扰乱。比如，子进程往socketpair发送信息，使得父进程的多路IO复用函数误以为信号发生了；父进程确实发生了信号，也往socketpair发送了一个字节，但却被子进程接收了这个字节。父进程没有监听到可读。

在Windows中，并没有直接的可以使用的socketpair API。此时，Libevent就自己实现了一个socketpair。具体可以参考《通用类型和函数》小节。

在函数的最后可以看到event_base的一个成员evsignal被赋值。evsignal是一个IO复用结构体，而evsigops是专门用于信号处理的IO复用结构体变量。定义如下：

```
//signal.c文件
static const struct eventop evsigops = {
    "signal",
    NULL,
    evsig_add,
    evsig_del,
    NULL,
    NULL,
    0, 0, 0
};
```

该结构体只有evsig_add和evsig_del这两个函数指针。实际在工作时有这两个函数就足够了。

将信号event加入到event_base

前面的代码已经完成了“创建socketpair并将socketpair的一个读端于ev_signal相关联”。接下来看其他的工作。假如要对一个绑定了某个信号的event调用event_add函数，那么在event_add的内部会调用event_add_internal函数。而event_add_internal函数又会调用evmap_signal_add函数。如果看了之前的博文，应该对这个流程不陌生。下面看看evmap_signal_add函数：

```
//evmap.c文件
int
evmap_signal_add(struct event_base *base, int sig, struct event *ev)
{
    //注意这里调用的是base的evsigsel变量。而不是evsel。
    const struct eventop *evsel = base->evsigsel;
    struct event_signal_map *map = &base->sigmap;

    ...

    if (TAILQ_EMPTY(&ctx->events)) {
        //实际调用的是evsig_add函数
        if (evsel->add(base, ev->ev_fd, 0, EV_SIGNAL, NULL)
            == -1)
            return (-1);
    }

    return (1);
}
```

上面函数的内部调用了IO复用结构体的add函数指针，即调用了evsig_add。现在我们深入evsig_add函数。

```
/signal.c文件
static int
evsig_add(struct event_base *base, evutil_socket_t evsignal, short old, s
hort events, void *p)
{
    struct evsig_info *sig = &base->sig;
    (void)p;

    //NSIG是信号的个数。定义在系统头文件中
    EVUTIL_ASSERT(evsignal >= 0 && evsignal < NSIG);

    /* catch signals if they happen quickly */
    //加锁保护。但实际其锁变量为NULL。所以并没有保护。应该会在以后的版本有所改正
    //在2.1.4-alpha版本中，就已经改进了这个问题。为锁变量分配了锁
    EVSIGBASE_LOCK();
    //如果有多个event_base，那么捕抓信号这个工作只能由其中一个完成。
    if (evsig_base != base && evsig_base_n_signals_added) {
        event_warnx("Added a signal to event base %p with signals "
            "already added to event_base %p.  Only one can have "
            "signals at a time with the %s backend.  The base with "
            "the most recently added signal or the most recent "
            "event_base_loop() call gets preference; do "
            "not rely on this behavior in future Libevent versions.",
            base, evsig_base, base->evsel->name);
    }
    evsig_base = base;
    evsig_base_n_signals_added = ++sig->ev_n_signals_added;
    evsig_base_fd = base->sig.ev_signal_pair[0]; //写端。0是写端，这确实与之前
所接触到的有所不同
    EVSIGBASE_UNLOCK();

    //设置Libevent的信号捕抓函数
    if (_evsig_set_handler(base, (int)evsignal, evsig_handler) == -1) {
        goto err;
    }

    //event_base第一次监听信号事件。要添加ev_signal到event_base中
    if (!sig->ev_signal_added) {
        //注意，本函数的调用路径为event_add->event_add_internal->evmap_signal
_map->evsig_add
        //所以这里是递归调用event_add函数。而event_add函数是会加锁的。所以需要锁为
递归锁
        if (event_add(&sig->ev_signal, NULL))//添加一个内部的event
            goto err;
        sig->ev_signal_added = 1;
    }

    return (0);

err:
    EVSIGBASE_LOCK();
```

```
        --evsig_base_n_signals_added;
        --sig->ev_n_signals_added;
        EVSIGBASE_UNLOCK();
        return (-1);
    }
```

从后面的那个if语句可以得知，当sig->ev_signal_added变量为0时(即用户第一次监听一个信号)，就会将ev_signal这个event加入到event_base中。从前面的“统一事件源”可以得知，这个ev_signal的作用就是通知event_base，有信号发生了。只需一个event即可完成工作，即使用户要监听多个不同的信号，因为这个event已经和socketpair的读端相关联了。如果要监听多个信号，那么就在信号处理函数中往这个socketpair写入不同的值即可。event_base能监听到可读，并可以从读到的内容可以判断是哪个信号发生了。

从代码中也可得知，Libevent并不会为每一个信号监听创建一个event。它只会创建一个全局的专门用于监听信号的event。这个也是“统一事件源”的工作原理。

设置信号捕捉函数

evsig_add函数还调用了_evsig_set_handler函数完成设置Libevent内部的信号捕抓函数。

```c
//signal.c文件
typedef void (*ev_sighandler_t)(int);

//evsignal是信号值，handler是信号捕抓函数
int
_evsig_set_handler(struct event_base *base,
    int evsignal, void (__cdecl *handler)(int))
{
    //如果有sigaction就优先使用之
#ifdef _EVENT_HAVE_SIGACTION
    struct sigaction sa;
#else
    ev_sighandler_t sh;
#endif
    struct evsig_info *sig = &base->sig;
    void *p;


    //数组的一个元素就存放一个信号。信号值等于其下标
    if (evsignal >= sig->sh_old_max) { //不够内存。重新分配
        int new_max = evsignal + 1;
        event_debug(("%s: evsignal (%d) >= sh_old_max (%d), resizing",
                __func__, evsignal, sig->sh_old_max));
        p = mm_realloc(sig->sh_old, new_max * sizeof(*sig->sh_old));
        if (p == NULL) {
            event_warn("realloc");
            return (-1);
        }

        memset((char *)p + sig->sh_old_max * sizeof(*sig->sh_old),
            0, (new_max - sig->sh_old_max) * sizeof(*sig->sh_old));

        sig->sh_old_max = new_max;
        sig->sh_old = p;
    }

    //注意sh_old是一个二级指针。元素是一个一级指针。为这个一级指针分配内存
    /* allocate space for previous handler out of dynamic array */
    sig->sh_old[evsignal] = mm_malloc(sizeof *sig->sh_old[evsignal]);
    if (sig->sh_old[evsignal] == NULL) {
        event_warn("malloc");
        return (-1);
    }

    /* save previous handler and setup new handler */
#ifdef _EVENT_HAVE_SIGACTION
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = handler;
    sa.sa_flags |= SA_RESTART;
    sigfillset(&sa.sa_mask);
```

```
        //设置信号处理函数
        if (sigaction(evsignal, &sa, sig->sh_old[evsignal]) == -1) {
            event_warn("sigaction");
            mm_free(sig->sh_old[evsignal]);
            sig->sh_old[evsignal] = NULL;
            return (-1);
        }
    #else
        //设置信号处理函数
        if ((sh = signal(evsignal, handler)) == SIG_ERR) {
            event_warn("signal");
            mm_free(sig->sh_old[evsignal]);
            sig->sh_old[evsignal] = NULL;
            return (-1);
        }
         //保存之前的信号捕抓函数。当用户event_del这个信号监听后，就可以恢复了
        *sig->sh_old[evsignal] = sh;
    #endif

        return (0);
    }
```

如果看过《UNIX环境高级编程》信号那章的话，上面这段代码很容易看懂。这里就不讲了。

这里我们做一个猜测：**当我们对某个信号进行event_new和event_add后，就不应该再次设置该信号的信号捕抓函数**。否则event_base将无法监听到信号的发生。下面代码验证这猜测。

```c
#include<unistd.h>
#include<stdio.h>
#include<signal.h>
#include<event.h>


void sig_cb(int fd, short events, void *arg)
{
    printf("in the sig_cb\n");
}

void signal_handle(int sig)
{
    printf("catch the sig %d\n", sig);
}

int main()
{

    struct event_base *base = event_base_new();

    struct event *ev = evsignal_new(base, SIGUSR1, sig_cb, NULL);
    event_add(ev, NULL);

    signal(SIGUSR1, signal_handle);

    printf("pid = %d\n", getpid());

    printf("begin\n");
    event_base_dispatch(base);
    printf("end\n");

    return 0;
}
```

运行上面代码, 通过在外部给这个进程发生信号的方式。可以看到，event_base确实无法监听到信号了。所有信号都被signal_handle捕抓了。

捕捉信号

前面的代码中有两个函数并没有讲，分别是信号捕抓函数evsig_handler和调用event_assign时的信号回调函数evsig_cb。

```
//signal.c文件
static void __cdecl
evsig_handler(int sig)
{
    ...
    ev_uint8_t msg;

    if (evsig_base == NULL) {
        event_warnx(
            "%s: received signal %d, but have no base configured",
            __func__, sig);
        return;
    }

#ifndef _EVENT_HAVE_SIGACTION
    //这主要是为了应对旧时代的信号不可靠
    //现在的OS并不会出现这个问题
    signal(sig, evsig_handler);
#endif

    /* Wake up our notification mechanism */
    msg = sig;
    send(evsig_base_fd, (char*)&msg, 1, 0); //向socketpair写入一个字节

    ...
}
```

从evsig_handler函数的实现可以看到，实现得相当简单。只是将信号对应的值写入到socketpair中。evsig_base_fd是socketpair的写端，这是一个全局变量，在evsig_add函数中被赋值的。

从“统一事件源”的工作原理来看，现在已经完成了对信号的捕抓，已经将该信号的当作IO事件写入到socketpair中了。现在event_base应该已经监听到socketpair可读了，并且会为调用回调函数evsig_cb了。下面看看evsig_cb函数。

```c
//signal.c文件
static void
evsig_cb(evutil_socket_t fd, short what, void *arg)
{
    static char signals[1024];
    ev_ssize_t n;
    int i;

    //NSIG是信号的个数
    int ncaught[NSIG];
    struct event_base *base;

    base = arg;

    memset(&ncaught, 0, sizeof(ncaught));

    while (1) {
        //读取socketpair中的数据。从中可以知道有哪些信号发生了
        //已经socketpair的读端已经设置为非阻塞的。所以不会被阻塞在
        //recv函数中。这个循环要把socketpair的所有数据都读取出来
        n = recv(fd, signals, sizeof(signals), 0);
        if (n == -1) {
            int err = evutil_socket_geterror(fd);
            if (! EVUTIL_ERR_RW_RETRIABLE(err))
                event_sock_err(1, fd, "%s: recv", __func__);
            break;
        } else if (n == 0) {
            /* XXX warn? */
            break;
        }

        //遍历数据数组，把每一个字节当作一个信号
        for (i = 0; i < n; ++i) {
            ev_uint8_t sig = signals[i];
            if (sig < NSIG)
                ncaught[sig]++; //该信号发生的次数
        }
    }

    EVBASE_ACQUIRE_LOCK(base, th_base_lock);
    for (i = 0; i < NSIG; ++i) {
        if (ncaught[i]) //有信号发生就为之调用evmap_signal_active
            evmap_signal_active(base, i, ncaught[i]);
    }
    EVBASE_RELEASE_LOCK(base, th_base_lock);
}
```

该回调函数的作用是读取socketpair的所有数据，并将数据当作信号，再根据信号值调用evmap_signal_active。

有一点要注意，evsig_cb这个回调函数并不是用户为监听一个信号调用event_new时设置的用户回调函数，而是Libevent内部为了处理信号而设置的内部回调函数。累！！

激活信号event

虽然如此，但是现在的情况是：当有信号发生时，就会调用evmap_signal_active函数。

```
//event-internal.h文件
#define ev_signal_next  _ev.ev_signal.ev_signal_next

#define ev_ncalls   _ev.ev_signal.ev_ncalls
#define ev_pncalls  _ev.ev_signal.ev_pncalls

//evmap.c文件
void //后两个参数分别是信号值和发生的次数。
evmap_signal_active(struct event_base *base, evutil_socket_t sig, int nca
lls)
{
    struct event_signal_map *map = &base->sigmap;
    struct evmap_signal *ctx;
    struct event *ev;

    //通过这个fd找到对应的TAILQ_HEAD
    GET_SIGNAL_SLOT(ctx, map, sig, evmap_signal);

    //遍历该fd的队列
    TAILQ_FOREACH(ev, &ctx->events, ev_signal_next)
        event_active_nolock(ev, EV_SIGNAL, ncalls);
}


//event.c文件
void
event_active_nolock(struct event *ev, int res, short ncalls)
{
    struct event_base *base;


    base = ev->ev_base;
    ev->ev_res = res;

    //这将停止处理低优先级的event。一路回退到event_base_loop中。
    if (ev->ev_pri < base->event_running_priority)
        base->event_continue = 1;

    if (ev->ev_events & EV_SIGNAL) {
#ifndef _EVENT_DISABLE_THREAD_SUPPORT
        if (base->current_event == ev && !EVBASE_IN_THREAD(base)) {
            ++base->current_event_waiters;
            //由于此时是主线程执行，所以并不会进行这个判断里面
            EVTHREAD_COND_WAIT(base->current_event_cond, base->th_base_lo
ck);
        }
#endif
        ev->ev_ncalls = ncalls;
        ev->ev_pncalls = NULL;
    }
```

```
        //插入到激活队列中.插入到队尾
        event_queue_insert(base, ev, EVLIST_ACTIVE);

    }
```

通过evmap_signal_active、event_active_nolock和event_queue_insert这三个函数的调用后，就可以把一个event插入到激活队列了。

由于这些函数的执行本身就是在Libevent处理event的回调函数之中的(Libevent正在处理内部的信号处理event)。所以并不需要从event_base_loop里的while循环里面再次执行一次evsel->dispatch()，才能执行到这次信号event。即无需等到下一次处理激活队列，就可以执行该信号event了。

首先要明确，现在执行上面三个函数相当于在执行event的回调函数。所以其是运行在event_process_active函数之中的。为什么是在这里，可以参考《Libevent工作流程探究》小节。

分析如下：

```
//event.c文件
static int
event_process_active(struct event_base *base)
{
    struct event_list *activeq = NULL;
    int i, c = 0;

    //从高优先级到低优先级遍历优先级数组
    for (i = 0; i < base->nactivequeues; ++i) {
        //对于特定的优先级，遍历该优先级的所有激活event
        if (TAILQ_FIRST(&base->activequeues[i]) != NULL) {
            base->event_running_priority = i;
            activeq = &base->activequeues[i];
            c = event_process_active_single_queue(base, activeq);
        }
    }

    return c;
}

static int
event_process_active_single_queue(struct event_base *base,
    struct event_list *activeq)
{
    struct event *ev;

    //遍历该优先级的所有event，并处理之
    for (ev = TAILQ_FIRST(activeq); ev; ev = TAILQ_FIRST(activeq)) {

        ...//开始处理这个event。会调用event的回调函数
    }

}
```

从上面的代码可以看到，Libevent在处理内部的那个信号处理event的回调函数时，其实是在event_process_active_single_queue的一个循环里面。因为Libevent内部的信号处理event的优先级最高优先级，并且在前面的将用户信号event插入到队列(即event_queue_insert)，在插入到队列的尾部。所以无论用户的这个信号event的优先级是多少，都是在Libevent的内部信号处理event的后面。所以在遍历上面两个函数的里外两个循环时，肯定会执行到用户的信号event。

执行已激活信号event

现在看看Libevent是怎么处理已激活的信号event的。

```
//event.c文件
static inline void
event_signal_closure(struct event_base *base, struct event *ev)
{
    short ncalls;
    int should_break;

    /* Allows deletes to work */
    ncalls = ev->ev_ncalls;
    if (ncalls != 0)
        ev->ev_pncalls = &ncalls;

    //while循环里面会调用用户设置的回调函数。该回调函数可能会执行很久
    //所以要解锁先.
    EVBASE_RELEASE_LOCK(base, th_base_lock);
    //如果该信号发生了多次，那么就需要多次执行回调函数
    while (ncalls) {
        ncalls--;
        ev->ev_ncalls = ncalls;
        if (ncalls == 0)
            ev->ev_pncalls = NULL;
        (*ev->ev_callback)(ev->ev_fd, ev->ev_res, ev->ev_arg);

        EVBASE_ACQUIRE_LOCK(base, th_base_lock);
        //其他线程调用event_base_loopbreak函数中断之
        should_break = base->event_break;
        EVBASE_RELEASE_LOCK(base, th_base_lock);

        if (should_break) {
            if (ncalls != 0)
                ev->ev_pncalls = NULL;
            return;
        }
    }
}
```

可以看到，如果对应的信号发生了多次，那么该信号event的回调函数将被执行多次。

15.evthread_notify_base通知主线程

原文地址

一般来说，是主线程执行event_base_dispatch函数。本文也是如此，如无特别说明，event_base_dispatch函数是由主线程执行的。

notify的理由

本文要说明的问题是，当主线程在执行event_base_dispatch进入多路IO复用函数时，会处于休眠状态，休眠前解锁。此时，其他线程可能想往event_base添加一个event，这个event可能是一般的IO event也可能是超时event。无论哪个，都需要及时告知主线程：有新的event要加进来。要实现这种功能就需要Libevent提供一种机制来提供唤醒主线程。

工作原理

Libevent提供的唤醒主线程机制也是挺简单的，其原理和《信号event的处理》一文中提到的方法是一样的。提供一个内部的IO event，专门用于唤醒主线程。当其他线程有event要add进来时，就往这个内部的IO event写入一个字节。此时，主线程在dispatch时，就能检测到可读，也就醒来了。这就完成了通知。这过程和Libevent处理信号event是一样的。

相关结构体

下面看一下Libevent的实现代码。同信号处理一样，先来看一下event_base提供了什么成员。

```
struct event_base {
    …
    //event_base是否处于通知的未决状态。即次线程已经通知了，但主线程还没处理这个通知

    int is_notify_pending;
    evutil_socket_t th_notify_fd[2]; //通信管道
    struct event th_notify;//用于监听th_notify_fd的读端
    //有两个可供选择的通知函数，指向其中一个通知函数
    int (*th_notify_fn)(struct event_base *base);
};
```

创建通知event并将之加入到event_base

现在看Libevent怎么创建通信通道，以及怎么和event相关联。在event_base_new_with_config(event_base_new会调用该函数)里面会调用evthread_make_base_notifiable函数，使得libevent变成可通知的。只有在已经支持多线程的情况下才会调用evthread_make_base_notifiable函数的。

```
//event.c文件
int
evthread_make_base_notifiable(struct event_base *base)
{
    //默认event回调函数和默认的通知函数
    void (*cb)(evutil_socket_t, short, void *) = evthread_notify_drain_de
fault;
    int (*notify)(struct event_base *) = evthread_notify_base_default;

    /* XXXX grab the lock here? */
    if (!base)
        return -1;

    //th_notify_fd[0]被初始化为-1,如果>=0,就说明已经被设置过了
    if (base->th_notify_fd[0] >= 0)
        return 0;

#if defined(_EVENT_HAVE_EVENTFD) && defined(_EVENT_HAVE_SYS_EVENTFD_H)
#ifndef EFD_CLOEXEC

#define EFD_CLOEXEC 0
#endif
    //Libevent优先使用eventfd，但eventfd的通信机制和其他的不一样。所以
    //要专门为eventfd创建通知函数和event回调函数
    base->th_notify_fd[0] = eventfd(0, EFD_CLOEXEC);
    if (base->th_notify_fd[0] >= 0) {
        evutil_make_socket_closeonexec(base->th_notify_fd[0]);
        notify = evthread_notify_base_eventfd;
        cb = evthread_notify_drain_eventfd;
    }
#endif
#if defined(_EVENT_HAVE_PIPE)
    //<0，说明之前的通知方式没有用上
    if (base->th_notify_fd[0] < 0) {
        //有些多路IO复用函数并不支持文件描述符。如果不支持，那么就不能使用这种
        //通知方式。有关这个的讨论.查看http://blog.csdn.net/luotuo44/article/d
etails/38443569
        if ((base->evsel->features & EV_FEATURE_FDS)) {
            if (pipe(base->th_notify_fd) < 0) {
                event_warn("%s: pipe", __func__);
            } else {
                evutil_make_socket_closeonexec(base->th_notify_fd[0]);
                evutil_make_socket_closeonexec(base->th_notify_fd[1]);
            }
        }
    }
#endif

#ifdef WIN32
#define LOCAL_SOCKETPAIR_AF AF_INET
#else
```

```
#define LOCAL_SOCKETPAIR_AF AF_UNIX
#endif

    if (base->th_notify_fd[0] < 0) {
        if (evutil_socketpair(LOCAL_SOCKETPAIR_AF, SOCK_STREAM, 0,
            base->th_notify_fd) == -1) {
            event_sock_warn(-1, "%s: socketpair", __func__);
            return (-1);
        } else {
            evutil_make_socket_closeonexec(base->th_notify_fd[0]);
            evutil_make_socket_closeonexec(base->th_notify_fd[1]);
        }
    }

    //无论哪种通信机制，都要使得读端不能阻塞
    evutil_make_socket_nonblocking(base->th_notify_fd[0]);

    //设置回调函数
    base->th_notify_fn = notify;

    //同样为了让写端不阻塞。虽然，如果同时出现大量需要notify的操作，会塞满通信通道。
    //本次的notify会没有写入到通信通道中(已经变成非阻塞了)。但这无所谓，因为目的是
    //唤醒主线程，通信通道有数据就肯定能唤醒。
    if (base->th_notify_fd[1] > 0)
        evutil_make_socket_nonblocking(base->th_notify_fd[1]);

    //该函数的作用等同于event_new。实际上event_new内部也是调用event_assign函数完
成工作的
    //函数cb作为这个event的回调函数
    event_assign(&base->th_notify, base, base->th_notify_fd[0],
                 EV_READ|EV_PERSIST, cb, base);

    //标明是内部使用的
    base->th_notify.ev_flags |= EVLIST_INTERNAL;
    event_priority_set(&base->th_notify, 0); //最高优先级

    return event_add(&base->th_notify, NULL);//加入到event_base中。
}
```

上面代码展示了，Libevent会从eventfd、pipe和socketpair中选择一种通信方式。由于有三种通信通道可供选择，下文为了方便叙述，就假定它选定的是pipe。

上面代码的工作过程和普通的Libevent例子程序差不多，首先创建一个文件描述符fd，然后用这个fd创建一个event，最后添加到event_base中。

唤醒流程

现在沿着这个内部的event的工作流程走一遍。

启动notify

首先往event写入一个字节，开启一切。由于这个event是内部的，用户是接触不到的。所以只能依靠Libevent提供的函数。当然这个函数也不会开放给用户，它只是供Libevent内部使用。

现在来看Libevent内部的需要。在event_add_internal函数中需要通知主线程，在该函数的最后面会调用evthread_notify_base。

```
//event.c文件
static int
evthread_notify_base(struct event_base *base)
{
    //确保已经加锁了
    EVENT_BASE_ASSERT_LOCKED(base);
    if (!base->th_notify_fn)
        return -1;

    //写入一个字节，就能使event_base被唤醒。
    //如果处于未决状态，就没必要写多一个字节
    if (base->is_notify_pending)
        return 0;

    //通知处于未决状态，当event_base醒过来就变成已决的了。
    base->is_notify_pending = 1;
    return base->th_notify_fn(base);
}
```

在evthread_notify_base中，会调用th_notify_fn函数指针。这个指针是在前面的evthread_make_base_notifiable函数中被赋值的。这里以evthread_notify_base_default作为例子。这个evthread_notify_base_default完成实际的通知操作。

激活内部event

```
//event.c文件。
static int
evthread_notify_base_default(struct event_base *base)
{
    char buf[1];
    int r;
    buf[0] = (char) 0;
    //通知一下，用来唤醒。写一个字节足矣
#ifdef WIN32
    r = send(base->th_notify_fd[1], buf, 1, 0);
#else
    r = write(base->th_notify_fd[1], buf, 1);
#endif
    //即使errno 等于 EAGAIN也无所谓，因为这是由于通信通道已经塞满了
    //这已经能唤醒主线程了。没必要一定要再写入一个字节
    return (r < 0 && errno != EAGAIN) ? -1 : 0;
}
```

从上面两个函数看到，其实通知也是蛮简单的。只是往管道里面写入一个字节。当然这已经能使得event_base检测到管道可读，从而实现唤醒event_base。

往管道写入一个字节，event_base就会被唤醒，然后调用这个管道对应event的回调函数。当然，在event_base醒来的时候，还能看到其他东西。这也是Libevent提供唤醒功能的原因。

现在看一下这个唤醒event的回调函数，也是看默认的那个。

```
//event.c
static void
evthread_notify_drain_default(evutil_socket_t fd, short what, void *arg)
{
    unsigned char buf[1024];
    struct event_base *base = arg;

    //读完fd的所有数据，免得再次被唤醒
#ifdef WIN32
    while (recv(fd, (char*)buf, sizeof(buf), 0) > 0)
        ;
#else
    while (read(fd, (char*)buf, sizeof(buf)) > 0)
        ;
#endif

    EVBASE_ACQUIRE_LOCK(base, th_base_lock);
    //修改之，使得其不再是未决的了。当然这也能让其他线程可以再次唤醒值。参看evthread_
notify_base函数
    base->is_notify_pending = 0;
    EVBASE_RELEASE_LOCK(base, th_base_lock);
}
```

这个函数也比较简单，也就是读取完管道里的所有数据，免得被多路IO复用函数检测到管道可读，而再次被唤醒。

上面的流程就完成了Libevent的通知唤醒主线程的功能，思路还是蛮清晰的。实现起来也是很简单。

注意事项

有一点要注意：要让Libevent支持这种可通知机制，就必须让Libevent使用多线程，即在代码的一开始调用evthread_use_pthreads()或者evthread_use_windows_threads()。虽然用户可以手动调用函数evthread_make_base_notifiable。但实际上是不能实现通知功能的。分析如下：

Libevent代码中是通过调用函数evthread_notify_base来通知的。但这个函数都是在一个if语句中调用的。判断的条件为是否需要通知。If成立的条件中肯定会&&上一个EVBASE_NEED_NOTIFY(base)。比如在event_add_internal函数中的为：

```
if (res != -1 && notify && EVBASE_NEED_NOTIFY(base))
        evthread_notify_base(base);
```

notify是根据函数中的判断而来，而**EVBASE_NEED_NOTIFY这个宏定义的作用是判断当前的线程是否不等于主线程**(即为event_base执行event_base_dispatch函数的线程)。它是一个条件宏。其中一个实现为：

```
#define EVBASE_NEED_NOTIFY(base)            \
    (_evthread_id_fn != NULL &&          \
        (base)->running_loop &&           \
        (base)->th_owner_id != _evthread_id_fn())

#define EVTHREAD_GET_ID() \
    (_evthread_id_fn ? _evthread_id_fn() : 1)
```

event_base结构体中th_owner_id变量指明当前为event_base执行event_base_dispatch函数的是哪个线程。在event_base_loop函数中用宏EVTHREAD_GET_ID()赋值。

如果一开始没有调用evthread_use_pthreads或者evthread_use_windows_threads，那么全局变量evthread_id_fn就为NULL。也就不能获取线程的ID了。EVBASE_NEED_NOTIFY宏也只会返回0，使得不能调用evthread_notify_base函数。关于线程这部分的分析，可以参考《多线程、锁、条件变量(一)》和《多线程、锁、条件变量(二)》。

下面的用一个例子验证。

```c
#include<event.h>
#include<stdio.h>
#include<unistd.h>
#include<thread.h>

#include<pthread.h> //Linux thread

struct event_base *base = NULL;


void pipe_cb(int fd, short events, void *arg)
{
    printf("in the cmd_cb\n");
}

void timeout_cb(int fd, short events, void *arg)
{
    printf("in the timeout_cb\n");
}

void* thread_fn(void *arg)
{
    char ch;
    scanf("%c", &ch); //just for wait

    struct event *ev = event_new(base, -1, EV_TIMEOUT | EV_PERSIST,
                                 timeout_cb, NULL);

    struct timeval tv = {2, 0};
    event_add(ev, &tv);
}

int main(int argc, char ** argv)
{
    if( argc >= 2 && argv[1][0] == 'y')
        evthread_use_pthreads();

    base = event_base_new();
    evthread_make_base_notifiable(base);

    int pipe_fd[2];
    pipe(pipe_fd);

    struct event *ev = event_new(base, pipe_fd[0],
                                 EV_READ | EV_PERSIST, pipe_cb, NULL);

    event_add(ev, NULL);

    pthread_t thread;
    pthread_create(&thread, NULL, thread_fn, NULL);
```

```
    event_base_dispatch(base);

    return 0;
}
```

如果次线程的event被add到event_base中，那么每2秒timeout_cb函数就会被调用一次。如果没有被add的话，就永远等待下去，没有任何输出。

16.超时event的处理

原文地址

如何成为超时event

Libevent允许创建一个超时event，使用evtimer_new宏。

```
//event.h文件
#define evtimer_new(b, cb, arg)        event_new((b), -1, 0, (cb), (arg))
```

从宏的实现来看，它一样是用到了一般的event_new，并且不使用任何的文件描述符。从超时event宏的实现来看，无论是evtimer创建的event还是一般event_new创建的event，都能使得Libevent进行超时监听。其实，使得Libevent对一个event进行超时监听的原因是：在调用event_add的时候，第二参数不能为NULL，要设置一个超时值。如果为NULL，那么Libevent将不会为这个event监听超时。下文统一称设置了超时值的event为超时event。

超时event的原理

Libevent对超时进行监听的原理不同于之前讲到的对信号的监听，Libevent对超时的监听的原理是，多路IO复用函数都是有一个超时值。如果用户需要Libevent同时监听多个超时event，那么Libevent就把超时值最小的那个作为多路IO复用函数的超时值。自然，当时间一到，就会从多路IO复用函数返回。此时对超时event进行处理即可。

Libevent运行用户同时监听多个超时event，那么就必须要对这个超时值进行管理。Libevent提供了小根堆和通用超时(common timeout)这两种管理方式。下文为了叙述方便，就假定使用的是小根堆。

工作流程

下面来看一下超时event的工作流程。

设置超时值

首先调用event_add时要设置一个超时值，这样才能成为一个超时event。

```
//event.c文件
//在event_add中，会把第三个参数设为0.使得使用的是相对时间
static inline int
event_add_internal(struct event *ev, const struct timeval *tv,
    int tv_is_absolute)
{
    struct event_base *base = ev->ev_base;
    int res = 0;
    int notify = 0;

    //tv不为NULL,就说明是一个超时event,在小根堆中为其留一个位置
    if (tv != NULL && !(ev->ev_flags & EVLIST_TIMEOUT)) {
        if (min_heap_reserve(&base->timeheap,
            1 + min_heap_size(&base->timeheap)) == -1)
            return (-1);  /* ENOMEM == errno */
    }

    ...//将IO或者信号event插入到对应的队列中。


    if (res != -1 && tv != NULL) {
        struct timeval now;

        //用户把这个event设置成EV_PERSIST。即永久event.
        //如果没有这样设置的话，那么只会超时一次。设置了，那么就
        //可以超时多次。那么就要记录用户设置的超时值。
        if (ev->ev_closure == EV_CLOSURE_PERSIST && !tv_is_absolute)
            ev->ev_io_timeout = *tv;

        //该event之前被加入到超时队列。用户可以对同一个event调用多次event_add
        //并且可以每次都用不同的超时值。
        if (ev->ev_flags & EVLIST_TIMEOUT) {
            /* XXX I believe this is needless. */
            //之前为该event设置的超时值是所有超时中最小的。
            //从下面的删除可知，会删除这个最小的超时值。此时多路IO复用函数
            //的超时值参数就已经改变了。
            if (min_heap_elt_is_top(ev))
                notify = 1; //要通知主线程。可能是次线程为这个event调用本函数

            //从超时队列中删除这个event。因为下次会再次加入。
            //多次对同一个超时event调用event_add,那么只能保留最后的那个。
            event_queue_remove(base, ev, EVLIST_TIMEOUT);
        }

        //因为可以在次线程调用event_add。而主线程刚好在执行event_base_dispatch
        if ((ev->ev_flags & EVLIST_ACTIVE) &&
            (ev->ev_res & EV_TIMEOUT)) {//该event被激活的原因是超时

            ...
            event_queue_remove(base, ev, EVLIST_ACTIVE);
        }
```

```
        //获取现在的时间
        gettime(base, &now);

        //虽然用户在event_add时只需用一个相对时间，但实际上在Libevent内部
        //还是要把这个时间转换成绝对时间。从存储的角度来说，存绝对时间只需
        //一个变量。而相对时间则需两个，一个存相对值，另一个存参照物。
        if (tv_is_absolute) { //该参数指明时间是否为一个绝对时间
            ev->ev_timeout = *tv;
        } else {
            //参照时间 + 相对时间  ev_timeout存的是绝对时间
            evutil_timeradd(&now, tv, &ev->ev_timeout);
        }


        //将该超时event插入到超时队列中
        event_queue_insert(base, ev, EVLIST_TIMEOUT);

        //本次插入的超时值，是所有超时中最小的。那么此时就需要通知主线程。
        if (min_heap_elt_is_top(ev))
            notify = 1;
    }

    //如果代码逻辑中是需要通知的。并且本线程不是主线程。那么就通知主线程
    if (res != -1 && notify && EVBASE_NEED_NOTIFY(base))
        evthread_notify_base(base);

    return (res);
}
```

对于同一个event，如果是IO event或者信号event，那么将无法多次添加。但如果是一个超时event，那么是可以多次添加的。并且对应超时值会使用最后添加时指明的那个，之前的统统不要，即替换掉之前的超时值。

代码中出现了多次使用了notify变量。这主要是用在：次线程在执行这个函数，而主线程在执行event_base_dispatch。前面说到Libevent能对超时event进行监听的原理是：多路IO复用函数有一个超时参数。在次线程添加的event的超时值更小，又或者替换了之前最小的超时值。在这种情况下，都是要通知主线程，告诉主线程，最小超时值已经变了。关于通知主线程evthread_notify_base，可以参考博文《evthread_notify_base通知主线程》。

代码中的第三个判断体中用到了ev->ev_io_timeout。但event结构体中并没有该变量。其实，ev_io_timeout是一个宏定义。

```
//event-internal.h文件
#define ev_io_timeout   _ev.ev_io.ev_timeout
```

要注意的一点是，在调用event_add时设定的超时值是一个时间段(可以认为隔多长时间就触发一次)，相对于现在，即调用event_add的时间，而不是调用event_base_dispatch的时间。

调用多路IO复用函数等待超时

现在来看一下event_base_loop函数，看其是怎么处理超时event的。

```c
//event.c文件
int
event_base_loop(struct event_base *base, int flags)
{
    const struct eventop *evsel = base->evsel;
    struct timeval tv;
    struct timeval *tv_p;
    int res, done, retval = 0;

    EVBASE_ACQUIRE_LOCK(base, th_base_lock);

    base->running_loop = 1;

    done = 0;
    while (!done) {
        tv_p = &tv;
        if (!N_ACTIVE_CALLBACKS(base) && !(flags & EVLOOP_NONBLOCK)) {
            // 根据Timer事件计算evsel->dispatch的最大等待时间(超时值最小)
            timeout_next(base, &tv_p);
        } else { //不进行等待
            //把等待时间置为0，即可不进行等待，马上触发事件
            evutil_timerclear(&tv);
        }

        res = evsel->dispatch(base, tv_p);

        //处理超时事件，将超时事件插入到激活链表中
        timeout_process(base);

        if (N_ACTIVE_CALLBACKS(base)) {
            int n = event_process_active(base);
        }
    }

done:
    base->running_loop = 0;
    EVBASE_RELEASE_LOCK(base, th_base_lock);

    return (retval);
}


//选出超时值最小的那个
static int
timeout_next(struct event_base *base, struct timeval **tv_p)
{
    /* Caller must hold th_base_lock */
    struct timeval now;
    struct event *ev;
    struct timeval *tv = *tv_p;
    int res = 0;
```

```
    // 堆的首元素具有最小的超时值，这个是小根堆的性质。
    ev = min_heap_top(&base->timeheap);

    //堆中没有元素
    if (ev == NULL) {
        *tv_p = NULL;
        goto out;
    }

    //获取当然时间
    if (gettime(base, &now) == -1) {
        res = -1;
        goto out;
    }

    // 如果超时时间<=当前时间，不能等待，需要立即返回
    // 因为ev_timeout这个时间是由event_add调用时的绝对时间 + 相对时间。所以ev_tim
eout是
    // 绝对时间。可能在调用event_add之后，过了一段时间才调用event_base_diapatch,
所以
    // 现在可能都过了用户设置的超时时间。
    if (evutil_timercmp(&ev->ev_timeout, &now, <=)) {
        evutil_timerclear(tv); //清零，这样可以让dispatcht不会等待，马上返回
        goto out;
    }

    // 计算等待的时间=当前时间-最小的超时时间
    evutil_timersub(&ev->ev_timeout, &now, tv);

out:
    return (res);
}
```

上面代码的流程是：计算出本次调用多路IO复用函数的等待时间，然后调用多路IO复用函数中等待超时。

激活超了时的event

上面代码中的timeout_process函数就是处理超了时的event。

```
//event.c文件
//把超时了的event，放到激活队列中。并且，其激活原因设置为EV_TIMEOUT
static void
timeout_process(struct event_base *base)
{
    /* Caller must hold lock. */
    struct timeval now;
    struct event *ev;

    if (min_heap_empty(&base->timeheap)) {
        return;
    }

    gettime(base, &now);

    //遍历小根堆的元素。之所以不是只取堆顶那一个元素，是因为当主线程调用多路IO复用函数
    //进入等待时，次线程可能添加了多个超时值更小的event
    while ((ev = min_heap_top(&base->timeheap))) {
        //ev->ev_timeout存的是绝对时间
        //超时时间比此刻时间大，说明该event还没超时。那么余下的小根堆元素更不用检查
了。
        if (evutil_timercmp(&ev->ev_timeout, &now, >))
            break;


        //下面说到的del是等同于调用event_del.把event从这个event_base中(所有的队
列都)
        //删除。event_base不再监听之。
        //这里是timeout_process函数。所以对于有超时的event，才会被del掉。
        //对于有EV_PERSIST选项的event，在处理激活event的时候，会再次添加进event_b
ase的。
        //这样做的一个好处就是，再次添加的时候，又可以重新计算该event的超时时间(绝对
时间)。
        event_del_internal(ev);

        //把这个event加入到event_base的激活队列中。
        //event_base的激活队列又有该event了。所以如果该event是EV_PERSIST的，是可
以
        //再次添加进该event_base的
        event_active_nolock(ev, EV_TIMEOUT, 1);
    }
}
```

当从多路IO复用函数返回时，就检查时间小根堆，看有多少个event已经超时了。如果超时了，那就把这个event加入到event_base的激活队列中。并且把这个超时del(删除)掉，这主要是用于非PERSIST 超时event的。删除一个event的具体操作可以查看这里。

处理永久超时event

现在来看一下如果该超时event有EV_PERSIST选项，在后面是怎么再次添加进event_base，因为前面的代码注释中已经说了，在选出超时event时，会把超时的event从event_base中delete掉。

```c
//event.c文件
int
event_assign(struct event *ev, struct event_base *base, evutil_socket_t f
d,
                  short events, void (*callback)(evutil_socket_t, short,
void *), void *arg)
{
    ...
    if (events & EV_PERSIST) {
        ev->ev_closure = EV_CLOSURE_PERSIST;
    } else {
        ev->ev_closure = EV_CLOSURE_NONE;
    }

    return 0;
}


static int
event_process_active_single_queue(struct event_base *base,
    struct event_list *activeq)
{
    struct event *ev;

    //遍历同一优先级的所有event
    for (ev = TAILQ_FIRST(activeq); ev; ev = TAILQ_FIRST(activeq)) {

        //下面这个if else 是用于IO event的。这里贴出，是为了了解一些非超时event是
        //怎么处理永久事件(EV_PERSIST)的。
        //如果是永久事件，那么只需从active队列中删除。
        if (ev->ev_events & EV_PERSIST)
            event_queue_remove(base, ev, EVLIST_ACTIVE);
        else //不是的话，那么就要把这个event删除掉。
            event_del_internal(ev);


        switch (ev->ev_closure) {
        //这个case只对超时event的EV_PERSIST才有用。IO的没有用
        case EV_CLOSURE_PERSIST:
            event_persist_closure(base, ev);
            break;

        default: //默认是EV_CLOSURE_NONE
        case EV_CLOSURE_NONE:
            //没有设置EV_PERSIST的超时event，就只有一次的监听机会
            (*ev->ev_callback)(
                ev->ev_fd, ev->ev_res, ev->ev_arg);
            break;
        }
    }
}
```

```
static inline void
event_persist_closure(struct event_base *base, struct event *ev)
{
    //在event_add_internal函数中，如果是超时event并且有EV_PERSIST，那么就会把ev_
io_timeout设置成
    //用户设置的超时时间(相对时间)。否则为0。即不进入判断体中。
    //说明这个if只用于处理具有EV_PERSIST属性的超时event
    if (ev->ev_io_timeout.tv_sec || ev->ev_io_timeout.tv_usec) {
        struct timeval run_at, relative_to, delay, now;
        ev_uint32_t usec_mask = 0;

        gettime(base, &now);

        //delay是用户设置的超时时间。event_add的第二个参数
        delay = ev->ev_io_timeout;
        //是因为超时才执行到这里，event可以同时监听多种事件。如果是由于可读而执行
        //到这里，那么就说明还没超时。
        if (ev->ev_res & EV_TIMEOUT) { //如果是因为超时而激活，那么下次超时就是
本次超时的
            relative_to = ev->ev_timeout; // 加上 delay 时间。
        } else {
            relative_to = now; //重新计算超时值
        }

        evutil_timeradd(&relative_to, &delay, &run_at);
        //无论relative是哪个时间，run_at都不应该小于now。
        //如果小于，则说明是用户手动修改了系统时间，使得gettime()函数获取了一个
        //之前的时间。比如现在是9点，用户手动调回到7点。
        if (evutil_timercmp(&run_at, &now, <)) {
            //那么就以新的系统时间为准
            evutil_timeradd(&now, &delay, &run_at);
        }

        //把这个event再次添加到event_base中。注意，此时第三个参数为1，说明是一个绝
对时间
        event_add_internal(ev, &run_at, 1);
    }
    EVBASE_RELEASE_LOCK(base, th_base_lock);
    (*ev->ev_callback)(ev->ev_fd, ev->ev_res, ev->ev_arg);//执行回调函数
}
```

这段代码的处理流程是：如果用户指定了EV_PERSIST，那么在event_assign中就记录下来。在event_process_active_single_queue函数中会针对永久event进行调用event_persist_closure函数对之进行处理。在event_persist_closure函数中，如果是一般的永久event，那么就直接调用该event的回调函数。如果是超时永久event，那么就需要再次计算新的超时时间，并将这个event再次插入到event_base中。

这段代码也指明了，如果一个event因可读而被激活，那么其超时时间就要重新计算。而不是之前的那个了。也就是说，如果一个event设置了3秒的超时，但1秒后就可读了，那么下一个超时值，就要重新计算设置，而不是2秒后。

从前面的源码分析也可以得到：如果一个event监听可读的同时也设置了超时值，并且一直没有数据可读，最后超时了，那么这个event将会被删除掉，不会再等。

17.Libevent时间管理

基本时间操作函数

Libevent采用的时间类型是struct timeval，这个类型在很多平台都提供了。此外，Libevent还提供了一系列的时间操作函数。比如两个struct timeval相加、相减、比较大小。有些平台直接提供了一些时间操作函数，但有些则没有，那么Libevent就自己实现。这些宏如下：

```
#ifdef _EVENT_HAVE_TIMERADD
#define evutil_timeradd(tvp, uvp, vvp) timeradd((tvp), (uvp), (vvp))
#define evutil_timersub(tvp, uvp, vvp) timersub((tvp), (uvp), (vvp))
#else
#define evutil_timeradd(tvp, uvp, vvp)                  \
    do {                                    \
        (vvp)->tv_sec = (tvp)->tv_sec + (uvp)->tv_sec;      \
        (vvp)->tv_usec = (tvp)->tv_usec + (uvp)->tv_usec;       \
        if ((vvp)->tv_usec >= 1000000) {            \
            (vvp)->tv_sec++;                \
            (vvp)->tv_usec -= 1000000;          \
        }                           \
    } while (0)
#define evutil_timersub(tvp, uvp, vvp)                  \
    do {                                    \
        (vvp)->tv_sec = (tvp)->tv_sec - (uvp)->tv_sec;      \
        (vvp)->tv_usec = (tvp)->tv_usec - (uvp)->tv_usec;   \
        if ((vvp)->tv_usec < 0) {               \
            (vvp)->tv_sec--;                \
            (vvp)->tv_usec += 1000000;          \
        }                           \
    } while (0)
#endif

#ifdef _EVENT_HAVE_TIMERCLEAR
#define evutil_timerclear(tvp) timerclear(tvp)
#else
#define evutil_timerclear(tvp)  (tvp)->tv_sec = (tvp)->tv_usec = 0
#endif


#define evutil_timercmp(tvp, uvp, cmp)                  \
    (((tvp)->tv_sec == (uvp)->tv_sec) ?             \
     ((tvp)->tv_usec cmp (uvp)->tv_usec) :              \
     ((tvp)->tv_sec cmp (uvp)->tv_sec))

#ifdef _EVENT_HAVE_TIMERISSET
#define evutil_timerisset(tvp) timerisset(tvp)
#else
#define evutil_timerisset(tvp)  ((tvp)->tv_sec || (tvp)->tv_usec)
#endif
```

代码中的那些条件宏，是在配置Libevent的时候检查所在的系统环境而定义的。具体的内容，可以参考《event-config.h指明所在系统的环境》一文。

Libevent的时间一般是用在超时event的。对于超时event，用户只需给出一个超时时间，比如多少秒，而不是一个绝对时间。但在Libevent内部，要将这个时间转换成绝对时间。所以在Libevent内部会经常获取系统时间(绝对时间)，然后进行一些处理，比如，转换、比较。

cache时间

Libevent封装了一个evutil_gettimeofday函数用来获取系统时间，该函数在POSIX的系统是直接调用gettimeofday函数，在Windows系统是通过_ftime函数。虽然gettimeofday的耗时成本不大，不过Libevent还是使用了一个cache保存时间，使得更加高效。在event_base结构体有一个struct timeval类型的cache变量 tv_cache。处理超时event的两个函数event_add_internal和event_base_loop内部都是调用gettime函数获取时间的。gettime函数如下：

```
//event.c文件
static int
gettime(struct event_base *base, struct timeval *tp)
{
    if (base->tv_cache.tv_sec) { //cache可用
        *tp = base->tv_cache;
        return (0);
    }

    …//没有cache的时候就使用其他方式获取时间
}
```

从上面代码可以看到，Libevent优先使用cache时间。tv_bache变量处理作为cache外，还有另外一个作用，下面会讲到。

cache的时间也是通过调用系统的提供的时间函数得到的。

```
//event.c文件
static inline void
update_time_cache(struct event_base *base)
{
    base->tv_cache.tv_sec = 0;
    if (!(base->flags & EVENT_BASE_FLAG_NO_CACHE_TIME))
        gettime(base, &base->tv_cache);
}
```

tv_cache是通过调用gettime来获取时间。由于tv_cache.tv_sec已经赋值为0，所以它将使用系统提供的时间函数得到时间。代码也展示了，如果event_base的配置中定义了EVENT_BASE_FLAG_NO_CACHE_TIME宏，将不能使用cache时间。关于这个宏的设置可以参考《配置event_base》一文。

处理用户手动修改系统时间

如果用户能老老实实，或许代码就不需要写得很复杂。由于用户的不老实，所以有时候要考虑很多很特殊的情况。在Libevent的时间管理这方面也是如此。

Libevent在实际使用时还有一个坑爹的现象，那就是，用户手动把时钟(wall time)往回调了。比如说现在是上午9点，但用户却把OS的系统时间调成了上午7点。这是很坑爹的。对于超时event和event_add的第二个参数，都是一个时间长度。但在内部Libevent要把这个时间转换成绝对时间。

如果用户手动修改了OS的系统时间。那么Libevent把超时时间长度转换成绝对时间将是弄巧成拙。拿上面的时间例子。如果用户设置的超时为1分钟。那么到了9：01就会超时。如果用户把系统时间调成了7点，那么要过2个小时01分才能发生超时。这就和用户原先的设置差得很远了。

读者可能会说，这个责任应该是由用户负。呵呵，但Libevent提供的函数接口是一个时间长度，既然是时间长度，那么无论用户怎么改变OS的系统时间，这个时间长度都是相对于event_add ()被调用的那一刻算起，这是不会变的。如果Libevent做不到这一点，这说明是Libevent没有遵循接口要求。

为此，Libevent提出了一些解决方案。

使用monotonic时间

问题的由来是因为用户能修改系统时间，所以最简单的解决方案就是能获取到一个用户不能修改的时间，然后以之为绝对时间。因为event_add提供给用户的接口使用的是一个时间长度，所以无论是使用哪个绝对时间都是无所谓的。

基于这一点，Libevent找到了monotonic时间，从字面来看monotonic翻译成单调。我们高中学过的单调函数英文名就是monotonic function。monotonic时间就像单调递增函数那样，只增不减的，没有人能手动修改之。

monotonic时间是boot启动后到现在的时间。用户是不能修改这个时间。如果Libevent所在的系统支持monotonic时间的话，那么Libevent就会选用这个monotonic时间为绝对时间。

首先，Libevent检查所在的系统是否支持monotonic时间。在event_base_new_with_config函数中会调用detect_monotonic函数检测。

```
//event.c文件
static void
detect_monotonic(void)
{
#if defined(_EVENT_HAVE_CLOCK_GETTIME) && defined(CLOCK_MONOTONIC)
    struct timespec ts;
    static int use_monotonic_initialized = 0;

    if (use_monotonic_initialized)
        return;

    if (clock_gettime(CLOCK_MONOTONIC, &ts) == 0)
        use_monotonic = 1; //系统支持monotonic时间
    use_monotonic_initialized = 1;
#endif
}
```

从上面代码可以看到，如果Libevent所在的系统支持monotonic时间，就将全局变量use_monotonic赋值1，作为标志。

如果Libevent所在的系统支持monotonic时间，那么Libevent将使用monotonic时间，也就是说Libevent用于获取系统时间的函数gettime将由monotonic提供时间。

```
//event.c文件
static int
gettime(struct event_base *base, struct timeval *tp)
{
    EVENT_BASE_ASSERT_LOCKED(base);

    if (base->tv_cache.tv_sec) {
        *tp = base->tv_cache;
        return (0);
    }

#if defined(_EVENT_HAVE_CLOCK_GETTIME) && defined(CLOCK_MONOTONIC)
    if (use_monotonic) {
        struct timespec ts;

        if (clock_gettime(CLOCK_MONOTONIC, &ts) == -1)
            return (-1);

        tp->tv_sec = ts.tv_sec;
        tp->tv_usec = ts.tv_nsec / 1000;

        //额外的功能
        if (base->last_updated_clock_diff + CLOCK_SYNC_INTERVAL
            < ts.tv_sec) {
            struct timeval tv;
            evutil_gettimeofday(&tv,NULL);
            //tv_clock_diff记录两种时间的时间差
            evutil_timersub(&tv, tp, &base->tv_clock_diff);
            base->last_updated_clock_diff = ts.tv_sec;
        }

        return (0);
    }
#endif

    //如果所在的系统不支持monotonic时间，那么只能使用evutil_gettimeofday了
    return (evutil_gettimeofday(tp, NULL));
}
```

上面的代码虽然首先是使用cache时间，但实际上event_base结构体的cache时间也是通过调用gettime函数而得到的。上面代码也可以看到：如果所在的系统没有提供monotonic时间，那么就只能使用evutil_gettimeofday这个函数提供的系统时间了。

从上面的分析可知，如果Libevent所在的系统支持monotonic时间，那么根本就不用考虑用户手动修改系统时间这坑爹的事情。但如果所在的系统没有支持monotonic时间，那么Libevent就只能使用evutil_gettimeofday获取一个用户能修改的时间。

尽可能精确记录时间差

现在来看一下Libevent在这种情况下在怎么解决这个坑爹得的问题。

Libevent给出的方案是，尽可能精确地计算 用户往回调了多长时间。如果知道了用户往回调了多长时间，那么将小根堆中的全部event的时间都往回调一样的时间即可。Libevent调用timeout_correct函数处理这个问题。

```
//event.c文件
static void
timeout_correct(struct event_base *base, struct timeval *tv)
{
    /* Caller must hold th_base_lock. */
    struct event **pev;
    unsigned int size;
    struct timeval off;
    int i;

    //如果系统支持monotonic时间，那么就不需要校准时间了
    if (use_monotonic)
        return;

    //获取现在的系统时间
    gettime(base, tv);


    //tv的时间更大，说明用户没有往回调系统时间。那么不需要处理
    if (evutil_timercmp(tv, &base->event_tv, >=)) {
        base->event_tv = *tv;
        return;
    }

    evutil_timersub(&base->event_tv, tv, &off);//off差值，即用户调小了多少

    pev = base->timeheap.p;
    size = base->timeheap.n;
    //用户已经修改了OS的系统时间。现在需要对小根堆的所有event
    //都修改时间。使得之适应新的系统时间
    for (; size-- > 0; ++pev) {
        struct timeval *ev_tv = &(**pev).ev_timeout;
        //前面已经用off保存了，用户调小了多少。现在只需
        //将小根堆的所有event的超时时间(绝对时间)都减去这个off即可
        evutil_timersub(ev_tv, &off, ev_tv);
    }

    //保存现在的系统时间。以防用户再次修改系统时间
    base->event_tv = *tv;
}
```

Libevent用event_base的成员变量event_tv保存用户修改系统时间前的系统时间。如果刚保存完，用户就修改系统时间，这样就能精确地计算出用户往回调了多长时间。但毕竟Libevent是用户态的库，不能做到用户修改系统时间前的一刻保存系统时间。

于是Libevent采用多采点的方式，即时不时就保存一次系统时间。所以在event_base_loop函数中的while循环体里面会有gettime(base, &base->event_tv);这是为了能多采点。但这个while循环里面还会执行多路IO复用函数和处理被激活event的回调函数(这个回调函数执行多久也是个未知数)。这两个函数的执行需要的时间可能会比较长，如果用户刚才是在执行完这两个函数之后修改系统时间，那么event_tv保存的时间就不怎么精确了。这也是没有办法的啊！！唉！！

下面贴出event_base_loop函数

```
//event.c文件
int
event_base_loop(struct event_base *base, int flags)
{
    const struct eventop *evsel = base->evsel;
    struct timeval tv;
    struct timeval *tv_p;
    int res, done, retval = 0;

    //要使用cache时间，得在配置event_base时，没有加入EVENT_BASE_FLAG_NO_CACHE_T
IME选项
    clear_time_cache(base);


    while (!done) {
        timeout_correct(base, &tv);

        tv_p = &tv;
        if (!N_ACTIVE_CALLBACKS(base) && !(flags & EVLOOP_NONBLOCK)) {
            //参考http://blog.csdn.net/luotuo44/article/details/38637671
            timeout_next(base, &tv_p); //获取dispatch的最大等待时间
        } else {
            evutil_timerclear(&tv);
        }

        //保存系统时间。如果有cache，将保存cache时间。
        gettime(base, &base->event_tv);

        //之所以要在进入dispatch之前清零，是因为进入
        //dispatch后，可能会等待一段时间。cache就没有意义了。
        //如果第二个线程此时想add一个event到这个event_base里面，在
        //event_add_internal函数中会调用gettime。如果cache不清零，
        //那么将会取这个cache时间。这将取一个不准确的时间。
        clear_time_cache(base);

        //多路IO复用函数
        res = evsel->dispatch(base, tv_p);

        //将系统时间赋值到cache中
        update_time_cache(base);

        //处理超时事件。参考http://blog.csdn.net/luotuo44/article/details/38
637671
        timeout_process(base);

        if (N_ACTIVE_CALLBACKS(base)) {
            int n = event_process_active(base);//处理激活event
        }
    }
```

```
    return (retval);
}
```

可以看到，在dispatch和event_process_active之间有一个update_time_cache。而前面的gettime(base,&base->event_tv);实际上取的就是cache的时间。所以，如果该Libevent支持cache的话，会精确那么一些。一般来说，用户为event设置的回调函数，不应该执行太久的时间。这也是tv_cache时间的另外一个作用。

出现的bug

由于Libevent的解决方法并不是很精确，所以还是会有一些bug。下面给出一个bug。如果用户是在调用event_new函数之后，event_add之前对系统时间进行修改，那么无论用户设置的event超时有多长，都会马上触发超时。下面给出实际的例子。**这个例子要运行在不支持monotonic时间的系统**，我是在Windows运行的。

```
#include <event2/event.h>
#include<stdio.h>


void timeout_cb(int fd, short event, void *arg)
{
    printf("in the timeout_cb\n");
}


int main()
{
    struct event_base *base = event_base_new();

    struct event *ev = event_new(base, -1, EV_TIMEOUT, timeout_cb, NULL);

    int ch;
//暂停，让用户有时间修改系统时间。可以将系统时间往前1个小时
    scanf("%c", &ch);

    struct timeval tv = {100, 0};//这个超时时长要比较长。这里取100秒
    //第二个参数不能为NULL.不然也是不能触发超时的。毕竟没有时间
    event_add(ev, &tv);

    event_base_dispatch(base);

    return 0;
}
```

这个bug的出现是因为，在event_base_new_with_config函数中有gettime(base,&base->event_tv)，所以event_tv记录了修改前的时间。而event_add是在修改系统时间后才调用的。所以event结构体的ev_timeout变量使用的是修改系统时间后的超时时间，这是正确的时间。在执行timeout_correct函数时，Libevent发现用户修改了系统时间，所以就将本来正确的ev_timeout减去了off。所以ev_timeout就变得比较修改后的系统时间小了。在后面检查超时时，就会发现该event已经超时了(实际是没有超时)，就把它触发。

如果该event有EV_PERSIST属性，那么之后的超时则会是正确的。这个留给读者去分析吧。

另外，Libevent并没有考虑把时钟往后调，比如现在是9点，用户把系统时间调成10点。上面的代码如果用户是在event_add之后修改系统时间，就能发现这个bug。

18.管理超时event

原文地址

前面的博文已经说到，如果要对多个超时event同时进行监听，就要对这些超时event进行集中管理，能够方便地(时间复杂度小)获取、加入、删除一个event。

在之前的Libevent版本，Libevent使用小根堆管理这些超时event。小根堆的插入和删除时间复杂度都是O(logN)。在2.0.4-alpha版本时，Libevent引入了一个叫common-timeout的东西来管理超时event，要注意的是，它并不是替代小根堆，而是和小根堆配合使用的。事实上，common-timeout的实现要用到小根堆。

Libevent的小根堆和数据结构教科书上的小根堆几乎是一样的。看一下数据结构和Libevent的小根堆源码，很容易就懂的。这样就不多讲了。

本文主要讲一下common-timeout。从common的字面意思和它的实际使用来说，可以把它翻译成“公用超时”。

common-timeout的用途

要讲解common-timeout，得先说明它的用途。前面说到它和小根堆是配合使用的。小根堆是用在：多个超时event的超时时长是随机的。而common-timeout则是用在：大量的超时event具有相同的超时时长。其中，超时时长是指event_add参数的第二个参数。**要注意的是，这些大量超时event虽然有相同的超时时长，但它们的超时时间是不同的**。因为超时时间 = 超时时长+ 调用event_add时间。

毫无疑问，如果有相同超时时长的大量超时event都放到小根堆上，那么效率比较低的。虽然小根堆的插入和删除的时间复杂度都是O(logN)，但是如果有大量的N，效率也是会下降很多。

common-timeout的原理

common-timeout的思想是，既然有大量的超时event具有相同的超时时长，那么就它们必定依次激活。如果把它们按照超时时间升序地放到一个队列中(在Libevent中就是这样做的)，那么每次只需检查队列的第一个超时event即可。因为其他超时event肯定在第一个超时之后才超时的。

前面说到common-timeout和小根堆是配合使用的。从common-timeout中选出最早超时的那个event，将之插入到小根堆中。然后通过小根堆对这个event进行超时监控。超时后再从common-timeout中选出下一个最早超时的event。具体的超时监控处理过程可以参考《超时event的处理》一文。通过这样处理后，就不用把大量的超时event都插入到小根堆中。

下面看一下Libevent的具体实现吧。

相关结构体

首先看一下event_base为common-timeout提供了什么成员变量。

```
//event-internal.h文件
struct event_base {
    //因为可以有多个不同时长的超时event组。故得是数组
    //因为数组元素是common_timeout_list指针，所以得是二级指针
    struct common_timeout_list **common_timeout_queues;
    //数组元素个数
    int n_common_timeouts;
    //已分配的数组元素个数
    int n_common_timeouts_allocated;
};

struct common_timeout_list {
    //超时event队列。将所有具有相同超时时长的超时event放到一个队列里面
    struct event_list events;

    struct timeval duration;//超时时长
    struct event timeout_event;//具有相同超时时长的超时event代表
    struct event_base *base;
};
```

在实际应用时，可能超时时长为10秒的有1k个超时event，时长为20秒的也有1k个，这就需要一个数组。数组的每一个元素是common_timeout_list结构体指针。每一个common_timeout_list结构体就会处理所有具有相同超时时长的超时event。

common_timeout_list结构体里面有一个event结构体成员，所以并不是从多个具有相同超时时长的超时event中选择一个作为代表，而是在内部有一个event。

common_timeout_list是使用struct event_list结构体队列来管理event，它是一种TAILQ_QUEUE队列，可以参考博文《TAILQ_QUEUE队列》。

使用common-timeout

现在来看看怎么使用common-timeout。从上面的代码可以想到，如果要使用common-timeout，就必须把超时event插入到common_timeout_list的events队列中。又因为其要求具有相同的超时时长，所以要插入的超时event要和某个common_timeout_list结构体有相同的超时时长。所以，我们还是来看一下怎么设置common_timeout_list结构体的超时时长。

实际上，并不是设置。而是向event_base申请一个具有特定时长的common_timeout_list。每申请一个，就会在common_timeout_queues数组中加入一个common_timeout_list元素。可以通过event_base_init_common_timeout申请。申请后，就可以直接调用event_add把超时event插入到common-timeout中。但问题是，common-timeout和小根堆是共存的，event_add又没有第三个参数作为说明，要插入到common-timeout还是小根堆。

common-timeout标志

其实，event_add是根据第二个参数，即超时时长值进行区分的。

首先有一个基本事实，对一个struct timeval结构体,成员tv_usec的单位是微秒，所以最大也就是999999,只需低20比特位就能存储了。但成员tv_usec的类型是int或者long，肯定有32比特位。所以，就有高12比特位是空闲的。

Libevent就是利用那空闲的12个比特位做文章的。这12比特位是高比特位。Libevent使用最高的4比特位作为标志位，标志它是一个专门用于common-timeout的时间，下文将这个标志称为common-timeout标志。次8比特位用来记录该超时时长在common_timeout_queues数组中的位置，即下标值。这也限制了common_timeout_queues数组的长度，最大为2的8次方，即256。

为了方便地处理这些比特位，Libevent定义了下面这些宏定义和一个判断函数。

```
//event.c文件
#define COMMON_TIMEOUT_MICROSECONDS_MASK       0x000fffff
#define MICROSECONDS_MASK       COMMON_TIMEOUT_MICROSECONDS_MASK
#define COMMON_TIMEOUT_IDX_MASK 0x0ff00000
#define COMMON_TIMEOUT_IDX_SHIFT 20
#define COMMON_TIMEOUT_MASK     0xf0000000
#define COMMON_TIMEOUT_MAGIC    0x50000000

#define COMMON_TIMEOUT_IDX(tv) \
    (((tv)->tv_usec & COMMON_TIMEOUT_IDX_MASK)>>COMMON_TIMEOUT_IDX_SHIFT)

#define MAX_COMMON_TIMEOUTS 256

static inline int
is_common_timeout(const struct timeval *tv,
    const struct event_base *base)
{
    int idx;
    //不具有common-timeout标志位，那么就肯定不是commont-timeout时间了
    if ((tv->tv_usec & COMMON_TIMEOUT_MASK) != COMMON_TIMEOUT_MAGIC)
        return 0;

    idx = COMMON_TIMEOUT_IDX(tv);//获取数组下标
    return idx < base->n_common_timeouts;
}
```

代码最后面的那个判断函数，是用来判断一个给定的struct timeval时间，是否为common-timeout时间。在event_add_internal函数中会用之作为判断，然后根据判断结果来决定是插入小根堆还是common-timeout，这也就完成了区分。

申请并得到特定时长的common-timeout

那么怎么得到一个具有common-timeout标志的时间呢？其实，还是通过前面说到的event_base_init_common_timeout函数。该函数将返回一个具有common-timeout标志的时间。

```
//event.c文件
//申请一个时长为duration的common_timeout_list
const struct timeval *
event_base_init_common_timeout(struct event_base *base,
    const struct timeval *duration)
{
    int i;
    struct timeval tv;
    const struct timeval *result=NULL;
    struct common_timeout_list *new_ctl;

    //这个时间的微秒位应该进位。用户没有将之进位。比如二进制的103，个位的3应该进位
    if (duration->tv_usec > 1000000) {
        //将之进位，因为下面会用到高位
        memcpy(&tv, duration, sizeof(struct timeval));
        if (is_common_timeout(duration, base))
            tv.tv_usec &= MICROSECONDS_MASK;//去除common-timeout标志
        tv.tv_sec += tv.tv_usec / 1000000; //进位
        tv.tv_usec %= 1000000;
        duration = &tv;
    }

    for (i = 0; i < base->n_common_timeouts; ++i) {
        const struct common_timeout_list *ctl =
            base->common_timeout_queues[i];
        //具有相同的duration， 即之前有申请过这个超时时长。那么就不用分配空间。
        if (duration->tv_sec == ctl->duration.tv_sec &&
            duration->tv_usec ==
            (ctl->duration.tv_usec & MICROSECONDS_MASK)) {//要&这个宏，才能
是正确的时间
            result = &ctl->duration;
            goto done;
        }
    }

    //达到了最大申请个数，不能再分配了
    if (base->n_common_timeouts == MAX_COMMON_TIMEOUTS) {
        goto done;
    }

    //新的超时时长，需要分配一个common_timeout_list结构体。

    //之前分配的空间已经用完了，要重新申请空间
    if (base->n_common_timeouts_allocated == base->n_common_timeouts) {
        int n = base->n_common_timeouts < 16 ? 16 :
            base->n_common_timeouts*2;
        struct common_timeout_list **newqueues =
            mm_realloc(base->common_timeout_queues,
            n*sizeof(struct common_timeout_queue *));
        if (!newqueues) {
            goto done;
```

```
            }
            base->n_common_timeouts_allocated = n;
            base->common_timeout_queues = newqueues;
        }

        //为该超时时长分配一个common_timeout_list结构体
        new_ctl = mm_calloc(1, sizeof(struct common_timeout_list));
        if (!new_ctl) {
            goto done;
        }

        //为这个结构体进行一些设置
        TAILQ_INIT(&new_ctl->events);
        new_ctl->duration.tv_sec = duration->tv_sec;
        new_ctl->duration.tv_usec =
            duration->tv_usec | COMMON_TIMEOUT_MAGIC | //为这个时间加入common-ti
    meout标志
            (base->n_common_timeouts << COMMON_TIMEOUT_IDX_SHIFT);//加入下标值

        //对timeout_event这个内部event进行赋值。设置回调函数和回调参数。
        evtimer_assign(&new_ctl->timeout_event, base,
            common_timeout_callback, new_ctl);

        new_ctl->timeout_event.ev_flags |= EVLIST_INTERNAL; //标志成内部event
        event_priority_set(&new_ctl->timeout_event, 0); //优先级为最高级
        new_ctl->base = base;
        //放到数组对应的位置上
        base->common_timeout_queues[base->n_common_timeouts++] = new_ctl;
        result = &new_ctl->duration;

    done:

        return result;
    }
```

该函数只是在event_base的common_timeout_queues数组中申请一个特定超时时长的位置。同时该函数也会返回一个struct timeval结构体指针变量，该结构体已经被赋予了common-timeout标志。以后使用该变量作为event_add的第二个参数，就可以把超时event插入到common-timeout中了。不应该也不能自己手动为struct timeval变量加入common-timeout标志。

该函数中，也给内部的event进行了赋值,设置了回调函数和回调参数。要注意的是回调参数是这个common_timeout_list结构体变量指针。在回调函数中，有了这个指针，就可以访问events变量，即访问到该结构体上的所有超时event。于是就能手动激活这些超时event。

在Libevent的官方例子中，得到event_base_init_common_timeout的返回值后，就把它存放到另外一个struct timeval结构体中。而不是直接使用返回值作为event_add的参数。

将超时event存放到common-timeout中

现在已经向event_base申请了一个特定的超时时长，并得到了具有common-timeout标志的时间。那么，就调用event_add看看。

```
//event.c文件
static inline int
event_add_internal(struct event *ev, const struct timeval *tv,
    int tv_is_absolute)
{
    struct event_base *base = ev->ev_base;
    int res = 0;
    int notify = 0;

    ...//加入到IO队列或者信号队列

    if (res != -1 && tv != NULL) {
        struct timeval now;
        int common_timeout;

        gettime(base, &now);

        //判断这个时间是否为common-timeout标志
        common_timeout = is_common_timeout(tv, base);
        if (common_timeout) {
            struct timeval tmp = *tv;
            //只取真正的时间部分，common-timeout标志位和下标位不要
            tmp.tv_usec &= MICROSECONDS_MASK;
            //转换成绝对时间
            evutil_timeradd(&now, &tmp, &ev->ev_timeout);
            ev->ev_timeout.tv_usec |=
                (tv->tv_usec & ~MICROSECONDS_MASK); //加入标志位
        }

        event_queue_insert(base, ev, EVLIST_TIMEOUT);

        if (common_timeout) {
            struct common_timeout_list *ctl =
                get_common_timeout_list(base, &ev->ev_timeout);
            if (ev == TAILQ_FIRST(&ctl->events)) {
                common_timeout_schedule(ctl, &now, ev);
            }
        }
    }

    return (res);
}
```

由于在《超时event的处理》一文中已经对这个函数进行了一部分讲解，现在只讲有关common-timeout部分。

虽然上面的代码省略了很多东西，但是有一点要说明，当超时event被加入common-timeout时并不会设置notify变量的，即不需要通知主线程。

common-timeout与小根堆的配合

从上面的代码可以看到，首先是为超时event内部时间ev_timeout加入common-timeout标志。然后调用event_queue_insert进行插入。但此时调用event_queue_insert插入，并不是插入到小根堆。它只是插入到event_base的common_timeout_list数组的一个队列中。下面代码可以看到这一点。

```
//event.c文件
static void
event_queue_insert(struct event_base *base, struct event *ev, int queue)
{

    ev->ev_flags |= queue;
    switch (queue) {
    case EVLIST_TIMEOUT: {
        if (is_common_timeout(&ev->ev_timeout, base)) {
            //根据时间向event_base获取对应的common_timeout_list
            struct common_timeout_list *ctl =
                get_common_timeout_list(base, &ev->ev_timeout);
            insert_common_timeout_inorder(ctl, ev);
        }
        break;
    }

    }
}

static void //in order说明是有序的。
insert_common_timeout_inorder(struct common_timeout_list *ctl,
    struct event *ev)
{
    struct event *e;
    //虽然有相同超时时长，但超时时间却是 超时时长 + 调用event_add的时间。
    //所以是在不同的时间触发超时的。它们根据绝对超时时间，升序排在队列中。
    //一般来说，直接插入队尾即可。因为后插入的，绝对超时时间肯定大。
    //但由于线程抢占的原因，可能一个线程在evutil_timeradd(&now, &tmp, &ev->ev_t
imeout);
    //执行完，还没来得及插入，就被另外一个线程抢占了。而这个线程也是要插入一个
    //common-timeout的超时event。这样就会发生：超时时间小的反而后插入。
    //所以要从后面开始遍历队列，寻找一个合适的地方。
    TAILQ_FOREACH_REVERSE(e, &ctl->events,
        event_list, ev_timeout_pos.ev_next_with_common_timeout) {
        if (evutil_timercmp(&ev->ev_timeout, &e->ev_timeout, >=)) {
            TAILQ_INSERT_AFTER(&ctl->events, e, ev, //从队列后面插入
                ev_timeout_pos.ev_next_with_common_timeout);
            return; //插入后就返回
        }
    }

    //在队列头插入，只会发生在前面的寻找都没有寻找到的情况下
    TAILQ_INSERT_HEAD(&ctl->events, ev,
        ev_timeout_pos.ev_next_with_common_timeout);
}
```

既然event_queue_insert函数并没有完成插入到小根堆。那么就看event_add_internal的最后面的那个if判断。读者可能会问，为什么要插入到小根堆。其实，前面已经说到了。common-timeout是采用一个代表的方式进行工作的。所以肯定要有一个代表被插入小根堆中，这也是common-timeout和小根堆的相互配合。

```
//event.c文件
if (common_timeout) {
    struct common_timeout_list *ctl =
        get_common_timeout_list(base, &ev->ev_timeout);
    if (ev == TAILQ_FIRST(&ctl->events)) {
        common_timeout_schedule(ctl, &now, ev);
    }
}

static void
common_timeout_schedule(struct common_timeout_list *ctl,
    const struct timeval *now, struct event *head)
{
    struct timeval timeout = head->ev_timeout;
    timeout.tv_usec &= MICROSECONDS_MASK; //清除common-timeout标志
    //用common_timeout_list结构体的一个event成员作为超时event调用event_add_int
ernal
    //由于已经清除了common-timeout标志，所以这次将插入到小根堆中。
    event_add_internal(&ctl->timeout_event, &timeout, 1);
}
```

从判断可以看到，它判断要插入的这个超时event是否为这个队列的第一个元素。如果是的话，就说这个特定超时时长队列第一次有超时event要插入。这就要进行一些处理。

在common_timeout_schedule函数中，我们可以看到，它将一个event插入到小根堆中了。并且也可以看到，代表者不是用户给出的超时event中的一个，而是common_timeout_list结构体的一个event成员。

将common-timeout event激活

现在来看一下当common_timeout_list的内部event成员被激活时怎么处理。它的回调函数为common_timeout_callback。

```
//event.c文件
static void
common_timeout_callback(evutil_socket_t fd, short what, void *arg)
{
    struct timeval now;
    struct common_timeout_list *ctl = arg;
    struct event_base *base = ctl->base;
    struct event *ev = NULL;
    EVBASE_ACQUIRE_LOCK(base, th_base_lock);
    gettime(base, &now);
    while (1) {
        ev = TAILQ_FIRST(&ctl->events);

        //该超时event还没到超时时间。不要检查其他了。因为是升序的
        if (!ev || ev->ev_timeout.tv_sec > now.tv_sec ||
            (ev->ev_timeout.tv_sec == now.tv_sec &&
            (ev->ev_timeout.tv_usec&MICROSECONDS_MASK) > now.tv_usec))
            break;

        //一系列的删除操作。包括从这个超时event队列中删除
        event_del_internal(ev);
        //手动激活超时event。注意，这个ev是用户的超时event
        event_active_nolock(ev, EV_TIMEOUT, 1);
    }

    //不是NULL，说明该队列还有超时event。那么需要再次common_timeout_schedule，进
行监听
    if (ev)
        common_timeout_schedule(ctl, &now, ev);
    EVBASE_RELEASE_LOCK(base, th_base_lock);
}
```

在回调函数中，会手动把用户的超时event激活。于是，用户的超时event就能被处理了。

由于Libevent这个内部超时event的优先级是最高的，所以在接下来就会处理用户的超时event，而无需等到下一轮多路IO复用函数调用返回后。这一点同信号event是一样的，在《信号event的处理》博文的最后有一些论证。

19.与event相关的一些函数和操作

原文地址

Libevent提供了一些与event相关的操作函数和操作。本文就重点讲一下这方面的源代码。

在Libevent中，无论是event还是event_base，都是使用指针而不会使用变量。实际上，如果查看Libevent不同的版本，就可以发现event和event_base这两个结构体的成员是不同的。对比libevent-2.0.21-stable和libevent-1.4.13-stable这两个版本，就可以发现其具有相当大的区别。

event的参数

一个event结构体和很多东西相关联，比如event_base、文件描述符fd、回调函数、回调参数等等，**下文把这些东西统一称为参数**。这些参数都是在调用event_new创建一个event时指定的。如果在后面需要再次获取这些参数时，可以通过一些函数来获取，而不应该直接访问event结构体的成员。

```
//event.c文件
evutil_socket_t //监听的文件描述符fd
event_get_fd(const struct event *ev)
{
    return ev->ev_fd;
}

struct event_base * //获取event_base
event_get_base(const struct event *ev)
{
    return ev->ev_base;
}

short //获取该event监听的事件
event_get_events(const struct event *ev)
{
    return ev->ev_events;
}

event_callback_fn //获取回调函数的函数指针
event_get_callback(const struct event *ev)
{
    return ev->ev_callback;
}


void * //获取回调函数参数
event_get_callback_arg(const struct event *ev)
{
    return ev->ev_arg;
}


void //一个函数获取所有
event_get_assignment(const struct event *event, struct event_base **base_
out, evutil_socket_t *fd_out,
                     short *events_out, event_callback_fn *callback_out, v
oid **arg_out)
{
    if (base_out)
        *base_out = event->ev_base;
    if (fd_out)
        *fd_out = event->ev_fd;
    if (events_out)
        *events_out = event->ev_events;
    if (callback_out)
        *callback_out = event->ev_callback;
    if (arg_out)
        *arg_out = event->ev_arg;
}
```

前面的那些函数是获取单个参数的，最后那个函数可以同时获取多个参数。并且如果不想获取某个参数，可以对应地传入一个NULL。

event的状态

一个event是可以有多个状态的，比如已初始化状态(initialized)、未决状态(pending)、激活状态(active)。

可以用event_initialized函数检测一个event是否处于已初始化状态：

```
//event.c文件
int
event_initialized(const struct event *ev)
{
    if (!(ev->ev_flags & EVLIST_INIT))
        return 0;

    return 1;
}
```

可以看到event_initialized只是检查event的ev_flags是否有EVLIST_INIT标志。从之前的博文可以知道，当用户调用event_new后，就会为event加入该标志，所以用event_new创建的even都是处于已初始化状态的。

当用户调用event_new创建一个event后，它还没处于未决状态(non-pending)，当用户调用event_add函数，将一个event插入到event_base队列后，就处于未决状态(pending)。

如果event监听的事件发生了或者超时了，那么该event就会被激活，处于激活状态。当event的回调函数被调用后，它就不再是激活状态了，但还是处于未决状态。如果用户调用了event_del或者event_free(该函数内部调用event_del)，那么该event就不再是未决状态了。

可以调用event_pending函数来检查event处于哪种事件的未决状态。但是该函数不仅仅会检查event的未决状态，还会检查event的激活状态。名不副实啊！！下面就看一下这个函数吧。

```
//event.c文件
int
event_pending(const struct event *ev, short event, struct timeval *tv)
{
    int flags = 0;

    if (EVUTIL_FAILURE_CHECK(ev->ev_base == NULL)) {
        event_warnx("%s: event has no event_base set.", __func__);
        return 0;
    }

    EVBASE_ACQUIRE_LOCK(ev->ev_base, th_base_lock);

    //flags记录用户监听了哪些事件
    if (ev->ev_flags & EVLIST_INSERTED)
        flags |= (ev->ev_events & (EV_READ|EV_WRITE|EV_SIGNAL));

    //flags记录event被什么事件激活了.用户可以调用event_active
    //手动激活event，并且可以使用之前用户没有监听的事件作为激活原因
    if (ev->ev_flags & EVLIST_ACTIVE)
        flags |= ev->ev_res;

    //记录该event是否还有超时属性
    if (ev->ev_flags & EVLIST_TIMEOUT)
        flags |= EV_TIMEOUT;

    //event可以被用户乱设值，然后作为参数。这里为了保证
    //其值只能是下面的事件。
    event &= (EV_TIMEOUT|EV_READ|EV_WRITE|EV_SIGNAL);

    /* See if there is a timeout that we should report */
    if (tv != NULL && (flags & event & EV_TIMEOUT)) {
        struct timeval tmp = ev->ev_timeout;
        tmp.tv_usec &= MICROSECONDS_MASK;
#if defined(_EVENT_HAVE_CLOCK_GETTIME) && defined(CLOCK_MONOTONIC)
        /* correctly remamp to real time */
        evutil_timeradd(&ev->ev_base->tv_clock_diff, &tmp, tv);
#else
        *tv = tmp;
#endif
    }

    EVBASE_RELEASE_LOCK(ev->ev_base, th_base_lock);

    return (flags & event);
}
```

该函数的作用是检查某个事件(由第二个参数指定)是否处于未决或者激活状态。

由flags的几个 |= 操作可知，它会把event监听的事件种类都记录下来。并且还会把event被激活的原因(也是一个事件)记录下来。下面会讲到手动激活一个event。所以event可能会被一个没有监听的事件锁激活。

如果该函数的第三个参数不为NULL，并且用户之前也让这个event监听了超时事件，而且用户在第二个参数中指明了要检查超时事件，那么将第三个参数将被赋值为该event的下次超时时间(绝对时间)。

event_pending函数的一个作用是可以判断一个event是否已经从event_base中删除了。比如说，某个event监听写事件而加入了event_base，但可能在某个时刻被删除。那么可以用下面的代码判断这个event是否已经被删除了。

手动激活event

除了运行event_base_dispatch死等外界条件把event激活外，Libevent还提供了一个API函数event_active，可以手动地把一个event激活。

```
//event.c文件
//res是激活的原因，是诸如EV_READ EV_TIMEOUT之类的宏.
//ncalls只对EV_SIGNAL信号有用，表示信号的次数
//因为IO事件不讲究次数，信号才讲究次数
void
event_active(struct event *ev, int res, short ncalls)
{
    //加锁，可以线程安全地手动激活一个event
    EVBASE_ACQUIRE_LOCK(ev->ev_base, th_base_lock);
    event_active_nolock(ev, res, ncalls);
    EVBASE_RELEASE_LOCK(ev->ev_base, th_base_lock);
}


void
event_active_nolock(struct event *ev, int res, short ncalls)
{
    struct event_base *base;

    //该event已经是激活状态
    if (ev->ev_flags & EVLIST_ACTIVE) {
        ev->ev_res |= res;
        return;
    }

    base = ev->ev_base;
    ev->ev_res = res;//记录被激活原因。以后会用到

    ...
    if (ev->ev_events & EV_SIGNAL) {
#ifndef _EVENT_DISABLE_THREAD_SUPPORT
        if (base->current_event == ev && !EVBASE_IN_THREAD(base)) {
            ++base->current_event_waiters;
            EVTHREAD_COND_WAIT(base->current_event_cond, base->th_base_lo
ck);
        }
#endif
        ev->ev_ncalls = ncalls;
        ev->ev_pncalls = NULL;
    }

    //将event插入到激活队列
    event_queue_insert(base, ev, EVLIST_ACTIVE);

    //调用本函数的线程不是主线程的话，就会通知主线程。使得主线程能赶快处理激活event
    if (EVBASE_NEED_NOTIFY(base))
        evthread_notify_base(base);
}
```

手动激活一个event的原理是：把event插入到激活队列。如果执行激活动作的线程不是主线程，那么还要唤醒主线程，让主线程及时处理激活event，不再睡眠在多路IO复用函数中。

由于手动激活一个event是直接把这个event插入到激活队列的，所以event的被激活原因(由res参数所指定)可以不是该event监听的事件。比如说该event只监听了EV_READ事件，那么可以调用event_active(ev,EV_SIGNAL, 1);用信号事件激活该event。

删除event

之前的博文都只是讲怎么创建event和将之add到event_base中。现在来讲一下怎么删除一个event。

```
void
event_free(struct event *ev)
{
    event_del(ev);
    mm_free(ev);//释放内存
}


int
event_del(struct event *ev)
{
    int res;
    //加锁保证线程安全
    EVBASE_ACQUIRE_LOCK(ev->ev_base, th_base_lock);
    res = event_del_internal(ev);
    EVBASE_RELEASE_LOCK(ev->ev_base, th_base_lock);

    return (res);
}



static inline int
event_del_internal(struct event *ev)
{
    struct event_base *base;
    int res = 0, notify = 0;

    base = ev->ev_base;

    /* See if we are just active executing this event in a loop */
    if (ev->ev_events & EV_SIGNAL) {
        if (ev->ev_ncalls && ev->ev_pncalls) {
            /* Abort loop *///终止循环
            *ev->ev_pncalls = 0;
        }
    }

    //从超时集合中删除.超时集合可能是小根堆也可能是common-timeout
    if (ev->ev_flags & EVLIST_TIMEOUT) {
        //删除超时event并不需要通知主线程。如果该event不是最早超时的，
        //那肯定不用通知了。如果是的话，那么主线程会醒来。醒来后，
        //主线程还是会再次检查超时集合中有哪些超时event超时了。这个被
        //删除的超时event自然也检查不出来。主线程只会空手而回。
        event_queue_remove(base, ev, EVLIST_TIMEOUT);
    }

    //该event已经在active队列中了。那么需要在active队列中删除之
    if (ev->ev_flags & EVLIST_ACTIVE)
        event_queue_remove(base, ev, EVLIST_ACTIVE);
```

```
        //该event已经在注册队列(eventqueue)中了，那么需要在注册队列中删除之
        if (ev->ev_flags & EVLIST_INSERTED) {
            event_queue_remove(base, ev, EVLIST_INSERTED);

            //此外还要在该fd或者sig队列中删除之。同一个fd可以有多个event。
            //所以这里还有一个队列
            if (ev->ev_events & (EV_READ|EV_WRITE))
                res = evmap_io_del(base, ev->ev_fd, ev);
            else
                res = evmap_signal_del(base, (int)ev->ev_fd, ev);
            if (res == 1) {
                /* evmap says we need to notify the main thread. */
                notify = 1;
                res = 0;
            }
        }

        //可能需要通知主线程
        if (res != -1 && notify && EVBASE_NEED_NOTIFY(base))
            evthread_notify_base(base);

        return (res);
    }
```

虽然要调用三个函数才能删除一个event，不过思路还是挺清晰的。删除的时候要加锁，删除完后要释放内存。从之前的博文也可以知道，一个event是会被加入到各种队列中的。所以将一个event删除，所做的工作主要是：将这个event从各种队列中删除掉。

删除一个event这个操作可能不是主线程调用的，这时就可能需要通知主线程。关于通知主线程的原理可以参考博文《evthread_notify_base通知主线程》。

20.通用类型和函数

原文地址

Libevent定义了一系列的可移植的兼容类型和函数。这使得在各个系统上都有一致的效果，Libevent一般都会在兼容通用类型和函数的前面加上ev或evutil前缀。

在实现上，Libevent都是使用条件编译+宏定义的方式。使用这种方式，同一个宏名字，可以使得在不同的系统上， 编译时得到不同的值。这种方式在跨平台编程中，经常使用到。此外，对于Libevent的兼容类型，如果所在系统已经有对应功能的类型，那么Libevent将直接将ev_XXX宏的值定义为该类型。如果所在系统没有对应的类型，那么就会选择一个比较合理的类型作为宏值。

兼容类型

定长位宽类型

因为，C/C++中，整型int的位宽(多少bit)是没有限定的，而在有的时候却需要一个确定的长度。在C99标准中有一个stdint.h头文件定义了一些确定位宽的整型，如int64_t、int32_t。但Libevent考虑到可能有的环境并没有支持这个头文件，所以自己就定义了自己的一套确切位宽的整型。

在util.h文件的一开始，就定义了一些通用的位宽确定的整数类型。如下:

```
#ifdef _EVENT_HAVE_UINT64_T
#define ev_uint64_t uint64_t
#define ev_int64_t int64_t
#elif defined(WIN32)
#define ev_uint64_t unsigned __int64
#define ev_int64_t signed __int64
#elif _EVENT_SIZEOF_LONG_LONG == 8
#define ev_uint64_t unsigned long long
#define ev_int64_t long long
#elif _EVENT_SIZEOF_LONG == 8
#define ev_uint64_t unsigned long
#define ev_int64_t long
#elif defined(_EVENT_IN_DOXYGEN)
#define ev_uint64_t ...
#define ev_int64_t ...
#else
#error "No way to define ev_uint64_t"
#endif
```

从代码中可以看到，它最先考虑当前环境是否已经定义了64位宽的整型，如果有的话，就直接使用。然后再考虑是否在Windows系统、是否定义了_EVENT_SIZEOF_LONG_LONG，并且值为8 ……

正如《event-config.h指明所在系统的环境》博文所说的，像_EVENT_HAVE_UINT64_T、_EVENT_SIZEOF_LONG_LONG这些宏定义都是在Libevent检测系统环境时定义的。

Libevent定义了一系列位宽的整型，如下图：

Type	Width	Signed	Maximum	Minimum
ev_uint64_t	64	No	EV_UINT64_MAX	0
ev_int64_t	64	Yes	EV_INT64_MAX	EV_INT64_MIN
ev_uint32_t	32	No	EV_UINT32_MAX	0
ev_int32_t	32	Yes	EV_INT32_MAX	EV_INT32_MIN
ev_uint16_t	16	No	EV_UINT16_MAX	0
ev_int16_t	16	Yes	EV_INT16_MAX	EV_INT16_MIN
ev_uint8_t	8	No	EV_UINT8_MAX	0
ev_int8_t	8	Yes	EV_INT8_MAX	EV_INT8_MIN

表来自这里。

其最值是直接计算出来的，如下：

```
#define EV_UINT64_MAX ((((ev_uint64_t)0xffffffffUL)<< 32) | 0xffffffffUL)

#define EV_INT64_MAX  ((((ev_int64_t) 0x7fffffffL) << 32) |0xffffffffL)
#define EV_INT64_MIN  ((-EV_INT64_MAX) - 1)
#define EV_UINT32_MAX((ev_uint32_t)0xffffffffUL)
#define EV_INT32_MAX  ((ev_int32_t) 0x7fffffffL)
#define EV_INT32_MIN  ((-EV_INT32_MAX) - 1)
#define EV_UINT16_MAX((ev_uint16_t)0xffffUL)
#define EV_INT16_MAX  ((ev_int16_t) 0x7fffL)
#define EV_INT16_MIN  ((-EV_INT16_MAX) - 1)
#define EV_UINT8_MAX  255
#define EV_INT8_MAX   127
#define EV_INT8_MIN   ((-EV_INT8_MAX) - 1)
```

EV_UINT64_MAX是需要使用位操作才能得到的。因为对于UL（unsigned long）类型说，是可移植的最大值了。因为对于32位的OS来说，long类型位宽是32位的，64位的OS，long是64位的。对于0xffffffffUL这个只有32位的字面值来说保证了可移植性。接着把其强制转换成ev_uint64_t类型，此时就有了64位宽，无论是在32位的系统还是64位的系统。然后再利用位操作达到目的。

有符号类型size_t

Libevent定义了ev_ssize_t作为有符号size_t的兼容类型。因为在遵循POSIX标准的系统中，将有符号的size_t定义为ssize_t，而Windows系统则定义为SSIZE_T。其的具体实现是，在util.h文件中如下定义：

```
#ifdef _EVENT_ssize_t
#define ev_ssize_t _EVENT_ssize_t
#else
#define ev_ssize_t ssize_t
#endif
```

然后在Windows系统的event-config.h文件中，则有下面的定义:

```
#define _EVENT_ssize_t SSIZE_T
```

同样，Libevent也给ev_size_t和ev_ssize_t定义了范围：

```
#if _EVENT_SIZEOF_SIZE_T == 8
#define EV_SIZE_MAX EV_UINT64_MAX
#define EV_SSIZE_MAX EV_INT64_MAX
#elif _EVENT_SIZEOF_SIZE_T == 4
#define EV_SIZE_MAX EV_UINT32_MAX
#define EV_SSIZE_MAX EV_INT32_MAX
#elif defined(_EVENT_IN_DOXYGEN)
#define EV_SIZE_MAX ...
#define EV_SSIZE_MAX ...
#else
#error "No way to defineSIZE_MAX"
#endif
#define EV_SSIZE_MIN((-EV_SSIZE_MAX) - 1)
```

偏移类型

Libevent定义了ev_off_t作为兼容的偏移类型。其实现也很简单。

```
#ifdef WIN32
#define ev_off_t ev_int64_t
#else
#define ev_off_t off_t
#endif
```

socket类型

按照Libevent的说法，除了Windows系统外，其他OS的套接字类型大多数都是int类型的。而在Windows系统中，为SOCKET类型，实际为intptr_t类型。所以Libevent的实现也很简单：

```
#ifdef WIN32
#define evutil_socket_t intptr_t
#else
#define evutil_socket_t int
#endif
```

socklen_t类型

在Berkeley套接字中，有一些函数的参数类型是socklen_t类型，你不能传一个int或者size_t过去。但在Windows系统中，又没有这样的一个类型。比如bind函数。在使用Berkeley套接字的系统上，该函数的第三个参数为socklen_t，而在Windows系统上，该参数的类型只是简单的int。为此，Libevent定义了一个兼容的ev_socklen_t类型。其实现为：

```
#ifdef WIN32
#define ev_socklen_t int
#elif defined(_EVENT_socklen_t)
#define ev_socklen_t _EVENT_socklen_t
#else
#define ev_socklen_t socklen_t
#endif
```

通用可以在event-config.h文件中找到_EVENT_socklen_t的定义，要在没有定义socklen_t系统的event-config.h文件中才能找到该定义。

```
/* Define to unsigned int if you donthave it */
#define _EVENT_socklen_t unsigned int
```

指针类型

intptr_t是一个很重要的类型，特别是在64位系统中。如果你要对两个指针进行运算，最好是先将这两个指针转换成intptr_t类型，然后才进行运算。因为在一些64位系统中，int还是32位，而指针类型为64位，所以两个指针相减，其结果对于32位的int来说，可能会溢出。’

为了兼容，Libevent定义了兼容的intptr_t类型。

```
#ifdef _EVENT_HAVE_UINTPTR_T
#define ev_uintptr_t uintptr_t
#define ev_intptr_t intptr_t
#elif _EVENT_SIZEOF_VOID_P <= 4
#define ev_uintptr_t ev_uint32_t
#define ev_intptr_t ev_int32_t
#elif _EVENT_SIZEOF_VOID_P <= 8
#define ev_uintptr_t ev_uint64_t
#define ev_intptr_t ev_int64_t
#elif defined(_EVENT_IN_DOXYGEN)
#define ev_uintptr_t ...
#define ev_intptr_t ...
#else
#error "No way to defineev_uintptr_t"
#endif
```

从代码中可以看到，如果系统本身有intptr_t类型的话，那么Libevent直接使用之，如果没有，那么就选择一个完全能放得下指针的类型。

在event-config.h中，_EVENT_SIZEOF_VOID_P被定义成sizeof(void*)，即一个指针类型的字节数。一般来说，在32位系统中，为4字节； 在64位系统中，为8字节。在Windows版本的event-config.h文件中，定义如下：

```
/* The size of `void *', as computed by sizeof. */
#ifdef _WIN64
#define _EVENT_SIZEOF_VOID_P 8
#else
#define _EVENT_SIZEOF_VOID_P 4
#endif
```

在我的Linux（32位），直接定义为：

```
/* The size of `void *', as computed bysizeof. */
#define _EVENT_SIZEOF_VOID_P 4
```

读者可以Google一下”intptr_t”和” LP32 ILP32 LP64 LLP64 ILP64”。

兼容函数

时间函数

在《Libevent时间管理》一文中，列出了一些基本的时间操作函数。这里就不重复了。在Libevent还定义了一个evutil_gettimeofday函数，那篇文章并没有展开讲。

该函数作为一个兼容函数可以在各个平台上获取系统时间。在非Windows平台上，可以直接使用函数gettimeofday。Windows则通过_ftime函数获取系统时间，然后转换。实现如下：

```
//util.h文件
#ifdef _EVENT_HAVE_GETTIMEOFDAY
#define evutil_gettimeofday(tv, tz) gettimeofday((tv), (tz))
#else
struct timezone;
int evutil_gettimeofday(struct timeval *tv, struct timezone *tz);
#endif

//evutil.c文件
#ifndef _EVENT_HAVE_GETTIMEOFDAY
/* No gettimeofday; this muse be windows. */
int
evutil_gettimeofday(struct timeval *tv, struct timezone *tz)
{
        struct _timeb tb;

        if (tv == NULL)
                return -1;

        _ftime(&tb);
        tv->tv_sec = (long)tb.time;
        tv->tv_usec = ((int)tb.millitm) * 1000;
        return 0;
}
#endif
```

socket API函数

由于遵循POSIX标准的OS有“一切皆文件”的思想，所以在处理socket 的时候比较简单。而在Windows平台，处理socket就没这么简单。而且Windows也没有很好地兼容Berkeley socket API。为了统一兼容，Libevent定义了一些通用的函数。

Libevent定义了通用函数有下面这些：

```
int evutil_make_socket_nonblocking(evutil_socket_t sock);
int evutil_make_listen_socket_reuseable(evutil_socket_t sock);
int evutil_make_socket_closeonexec(evutil_socket_t sock);
int evutil_closesocket(evutil_socket_t sock);
int evutil_socketpair(int d, int type, int protocol, evutil_socket_t
sv[2]);
EVUTIL_SOCKET_ERROR();//宏定义
EVUTIL_SET_SOCKET_ERROR(errcode);//宏定义
```

上面的函数中，除了evutil_socketpair其他的都没有什么好讲的。因为都是一些Linux和Windows系统编程的基础内容。

在遵循POSIX的系统已经定义了socketpair，所以直接使用即可。但在Windows中并没有定义socketpair。Libevent的处理方法是：使用普通的socket，在函数内部创建一个服务器socket和客户端socket，并让客户端连接上服务器。其中，IP地址使用环路地址(ipv4中就是那个127.0.0.1，ipv6是::1)，端口号则由内核自动选定。实现起来还是有点麻烦的。因为具体的实现就是一个简单的C/S模式代码，这里就不贴代码了。

Libevent还定义了另外三个socket相关的通用操作函数。

```
const char *evutil_inet_ntop(int af, const void *src, char *dst, size_t l
en);
int evutil_inet_pton(int af, const char *src, void *dst);
int evutil_parse_sockaddr_port(const char *str, struct sockaddr *out, int
*outlen);
```

inet_ntop主要的一个功能是，它可以用于ipv6。对于ipv4，有inet_aton和inet_ntoa。这两个函数在POSIX和Windows中都是有的，不需要Libevent做什么工作。但对于inet_ntop和inet_pton，Windows中并没有提供(POSIX提供了)。

Libevent也是可以用于ipv6的。所以Libevent就定义了两个通用的函数。在实现上，Libevent也是自己写代码将之转换。这里也不贴代码了。

evutil_parse_sockaddr_port函数是用来解析一个字符串的。字符串的格式是,IP:port。比如，8.8.8.8:53。这个函数的作用就是将字符串所表示的ip和端口进行解析，并存放到参数out所指向的结构体上。之前，我们需要手动将一个ip和端口赋值给sockaddr_in 结构体上。现在有这个函数，这工作不用我们做了。第三个参数是一个 值-结果 参数。即调用函数时，它的值是第二个参数out指向空间的大小。函数返回后，它的值指明该函数写了多少字节在out上。具体的实现这里也不说了。

结构体偏移量

这个函数的功能主要是求结构体成员在结构体中的偏移量。定义如下：

```
#ifdef offsetof
#define evutil_offsetof(type, field) offsetof(type, field)
#else
#define evutil_offsetof(type, field) ((off_t)(&((type*)0)->field))
#endif
```

其中，type表示结构体名称，field表示成员名称。可以看到，Libevent还是优先使用所在系统本身提供的offsetof函数。Libevent自己实现的版本也是很巧妙的。它用(type*)0来让编译器认为有个结构体，它的起始地址为0。这样，编译器给field所在的地址就是编译器给field安排的偏移量。

这个求偏移量的功能是Libevent是很有用的。不过，Libevent不是直接使用这个宏evutil_offsetof。而是使用宏EVUTIL_UPCAST。

```
//util-internal.h文件
#define EVUTIL_UPCAST(ptr, type, field)                                 \
        ((type *)(((char*)(ptr))- evutil_offsetof(type, field)))
```

这个宏EVUTIL_UPCAST的作用是通过成员变量的地址获取其所在的结构体变量地址。比如有下面的结构体:

```
struct Parent
{
        struct Children ch;
        struct event ev;
};
```

假如变量child是struct Children类型指针，并且它是struct Parent结构体的成员。而且知道了child的地址，现在想获取child所在结构体的struct Parent的地址。此时就可以用EVUTIL_UPCAST宏了。如下使用就能转换了。

```
struct Parent *par = EVUTIL_UPCAST(child, struct Parent, ch);
//展开宏后，如下
struct Parent *par = ((struct Parent *)(((char*)(child))  - evutil_offset
of(struct Parent, ch)));
```

其中，并不需要ch为struct Parent的第一个成员变量。

EVUTIL_UPCAST宏的工作原理也是挺简单的，成员变量的地址减去其本身相对于所在结构体的偏移量就是所在结构体的起始地址了，再将这个地址强制转换成即可。

参考：http://www.wangafu.net/~nickm/libevent-book/Ref5_evutil.html

21.连接监听器evconnlistener

原文地址

使用evconnlistener

基于event和event_base已经可以写一个CS模型了。但是对于服务器端来说，仍然需要用户自行调用socket、bind、listen、accept等步骤。这个过程有点繁琐，为此在2.0.2-alpha版本的Libevent推出了一些对应的封装函数。

用户只需初始化struct sockaddr_in结构体变量，然后把它作为参数传给函数evconnlistener_new_bind即可。该函数会完成上面说到的那4个过程。下面的代码是一个使用例子。

```
#include<netinet/in.h>
#include<sys/socket.h>
#include<unistd.h>

#include<stdio.h>
#include<string.h>

#include<event.h>
#include<listener.h>
#include<bufferevent.h>
#include<thread.h>


void listener_cb(evconnlistener *listener, evutil_socket_t fd,
                 struct sockaddr *sock, int socklen, void *arg);

void socket_read_cb(bufferevent *bev, void *arg);
void socket_error_cb(bufferevent *bev, short events, void *arg);

int main()
{
    evthread_use_pthreads();//enable threads

    struct sockaddr_in sin;
    memset(&sin, 0, sizeof(struct sockaddr_in));
    sin.sin_family = AF_INET;
    sin.sin_port = htons(8989);

    event_base *base = event_base_new();
    evconnlistener *listener
            = evconnlistener_new_bind(base, listener_cb, base,
                                      LEV_OPT_REUSEABLE|LEV_OPT_CLOSE_ON_
FREE | LEV_OPT_THREADSAFE,
                                      10, (struct sockaddr*)&sin,
                                      sizeof(struct sockaddr_in));

    event_base_dispatch(base);

     evconnlistener_free(listener);
    event_base_free(base);

    return 0;
}


//有新的客户端连接到服务器
//当此函数被调用时，libevent已经帮我们accept了这个客户端。该客户端的
//文件描述符为fd
void listener_cb(evconnlistener *listener, evutil_socket_t fd,
                 struct sockaddr *sock, int socklen, void *arg)
{
```

```
    event_base *base = (event_base*)arg;

    //下面代码是为这个fd创建一个bufferevent
    bufferevent *bev =  bufferevent_socket_new(base, fd,
                                               BEV_OPT_CLOSE_ON_FREE);

    bufferevent_setcb(bev, socket_read_cb, NULL, socket_error_cb, NULL);
    bufferevent_enable(bev, EV_READ | EV_PERSIST);
}


void socket_read_cb(bufferevent *bev, void *arg)
{
    char msg[4096];

    size_t len = bufferevent_read(bev, msg, sizeof(msg)-1 );

    msg[len] = '\0';
    printf("server read the data %s\n", msg);

    char reply[] = "I has read your data";
    bufferevent_write(bev, reply, strlen(reply) );
}


void socket_error_cb(bufferevent *bev, short events, void *arg)
{
    if (events & BEV_EVENT_EOF)
        printf("connection closed\n");
    else if (events & BEV_EVENT_ERROR)
        printf("some other error\n");

    //这将自动close套接字和free读写缓冲区
    bufferevent_free(bev);
}
```

上面的代码是一个服务器端的例子，客户端代码可以使用《Libevent使用例子，从简单到复杂》博文中的客户端。这里就不贴客户端代码了。

从上面代码可以看到，当服务器端监听到一个客户端的连接请求后，就会调用listener_cb这个回调函数。这个回调函数是在evconnlistener_new_bind函数中设置的。现在来看一下这个函数的参数有哪些，下面是其函数原型。

```
//listener.h文件
typedef void (*evconnlistener_cb)(struct evconnlistener *, evutil_socket_
t, struct sockaddr *, int socklen, void *);

struct evconnlistener *evconnlistener_new_bind(struct event_base *base,
    evconnlistener_cb cb, void *ptr, unsigned flags, int backlog,
    const struct sockaddr *sa, int socklen);
```

第一个参数是很熟悉的event_base，无论怎么样都是离不开event_base这个发动机的。

第二个参数是一个函数指针，该函数指针的格式如代码所示。当有新的客户端请求连接时，该函数就会调用。要注意的是：当这个回调函数被调用时，Libevent已经帮我们accept了这个客户端。所以，该回调函数有一个参数是文件描述符fd。我们直接使用这个fd即可。真是方便。这个参数是可以为NULL的，此时用户并不能接收到客户端。当用户调用evconnlistener_set_cb函数设置回调函数后，就可以了。

第三个参数是传给回调函数的用户参数，作用就像event_new函数的最后一个参数。

参数flags是一些标志值，有下面这些：

- LEV_OPT_LEAVE_SOCKETS_BLOCKING：默认情况下，当连接监听器接收到新的客户端socket连接后，会把该socket设置为非阻塞的。如果设置该选项，那么就把之客户端socket保留为阻塞的
- LEV_OPT_CLOSE_ON_FREE：当连接监听器释放时，会自动关闭底层的socket
- LEV_OPT_CLOSE_ON_EXEC：为底层的socket设置close-on-exec标志
- LEV_OPT_REUSEABLE: 在某些平台，默认情况下当一个监听socket被关闭时，其他socket不能马上绑定到同一个端口，要等一会儿才行。设置该标志后，Libevent会把该socket设置成reuseable。这样，关闭该socket后，其他socket就能马上使用同一个端口
- LEV_OPT_THREADSAFE：为连接监听器分配锁。这样可以确保线程安全

参数backlog是系统调用listen的第二个参数。最后两个参数就不多说了。

evconnlistener的封装

接下来看一下Libevent是怎么封装evconnlistener的。

用到的结构体

```
//listener.c文件
struct evconnlistener_ops {//一系列的工作函数
    int (*enable)(struct evconnlistener *);
    int (*disable)(struct evconnlistener *);
    void (*destroy)(struct evconnlistener *);
    void (*shutdown)(struct evconnlistener *);
    evutil_socket_t (*getfd)(struct evconnlistener *);
    struct event_base *(*getbase)(struct evconnlistener *);
};

struct evconnlistener {
    const struct evconnlistener_ops *ops;//操作函数
    void *lock; //锁变量，用于线程安全
    evconnlistener_cb cb;//用户的回调函数
    evconnlistener_errorcb errorcb;//发生错误时的回调函数
    void *user_data;//回调函数的参数
    unsigned flags;//属性标志
    short refcnt;//引用计数
    unsigned enabled : 1;//位域为1.即只需一个比特位来存储这个成员
};

struct evconnlistener_event {
    struct evconnlistener base;
    struct event listener; //内部event,插入到event_base
};
```

在evconnlistener_event结构体有一个event结构体。可以想象，在实现时必然是将服务器端的socket fd赋值给struct event 类型变量listener的fd成员。然后将listener加入到event_base，这样就完成了自动监听工作。这也回归到之前学过的内容。

下面看一下具体是怎么实现的。

初始化服务器socket

```
//listener.c文件
struct evconnlistener *
evconnlistener_new_bind(struct event_base *base, evconnlistener_cb cb,
    void *ptr, unsigned flags, int backlog, const struct sockaddr *sa,
    int socklen)
{
    struct evconnlistener *listener;
    evutil_socket_t fd;
    int on = 1;
    int family = sa ? sa->sa_family : AF_UNSPEC;

    //监听个数不能为0
    if (backlog == 0)
        return NULL;

    fd = socket(family, SOCK_STREAM, 0);
    if (fd == -1)
        return NULL;

    //LEV_OPT_LEAVE_SOCKETS_BLOCKING选项是应用于accept到的客户端socket
    //所以对于服务器端的socket，直接将之设置为非阻塞的
    if (evutil_make_socket_nonblocking(fd) < 0) {
        evutil_closesocket(fd);
        return NULL;
    }

    if (flags & LEV_OPT_CLOSE_ON_EXEC) {
        if (evutil_make_socket_closeonexec(fd) < 0) {
            evutil_closesocket(fd);
            return NULL;
        }
    }

    if (setsockopt(fd, SOL_SOCKET, SO_KEEPALIVE, (void*)&on, sizeof(on))
<0) {
        evutil_closesocket(fd);
        return NULL;
    }
    if (flags & LEV_OPT_REUSEABLE) {
        if (evutil_make_listen_socket_reuseable(fd) < 0) {
            evutil_closesocket(fd);
            return NULL;
        }
    }

    if (sa) {
        if (bind(fd, sa, socklen)<0) {//绑定
            evutil_closesocket(fd);
            return NULL;
        }
    }
```

```
    listener = evconnlistener_new(base, cb, ptr, flags, backlog, fd);
    if (!listener) {
        evutil_closesocket(fd);
        return NULL;
    }

    return listener;
}
```

evconnlistener_new_bind函数申请一个socket，然后对之进行一些有关非阻塞、重用、保持连接的处理、绑定到特定的IP和端口。最后把业务逻辑交给evconnlistener_new处理。

```c
//listener.c文件
static const struct evconnlistener_ops evconnlistener_event_ops = {
    event_listener_enable,
    event_listener_disable,
    event_listener_destroy,
    NULL, /* shutdown */
    event_listener_getfd,
    event_listener_getbase
};


struct evconnlistener *
evconnlistener_new(struct event_base *base,
    evconnlistener_cb cb, void *ptr, unsigned flags, int backlog,
    evutil_socket_t fd)
{
    struct evconnlistener_event *lev;

    if (backlog > 0) {
        if (listen(fd, backlog) < 0)
            return NULL;
    } else if (backlog < 0) {
        if (listen(fd, 128) < 0)
            return NULL;
    }

    lev = mm_calloc(1, sizeof(struct evconnlistener_event));
    if (!lev)
        return NULL;

    //赋值
    lev->base.ops = &evconnlistener_event_ops;
    lev->base.cb = cb;
    lev->base.user_data = ptr;
    lev->base.flags = flags;
    lev->base.refcnt = 1;

    if (flags & LEV_OPT_THREADSAFE) {//线程安全就需要分配锁
        EVTHREAD_ALLOC_LOCK(lev->base.lock, EVTHREAD_LOCKTYPE_RECURSIVE);
    }

    //在多路IO复用函数中，新客户端的连接请求也被当作读事件
    event_assign(&lev->listener, base, fd, EV_READ|EV_PERSIST,
        listener_read_cb, lev);

    //会调用event_add，把event加入到event_base中
    evconnlistener_enable(&lev->base);

    return &lev->base;
}
```

```
int
evconnlistener_enable(struct evconnlistener *lev)
{
    int r;
    LOCK(lev);
    lev->enabled = 1;
    if (lev->cb)
        r = lev->ops->enable(lev);//实际上是调用下面的event_listener_enable函
数
    else
        r = 0;
    UNLOCK(lev);
    return r;
}

static int
event_listener_enable(struct evconnlistener *lev)
{
    struct evconnlistener_event *lev_e =
        EVUTIL_UPCAST(lev, struct evconnlistener_event, base);

    //加入到event_base，完成监听工作。
    return event_add(&lev_e->listener, NULL);
}
```

几个函数的一路调用，思路还是挺清晰的。就是申请一个socket，进行一些处理，然后用之赋值给event。最后把之add到event_base中。event_base会对新客户端的请求连接进行监听。

在evconnlistener_enable函数里面，如果用户没有设置回调函数，那么就不会调用event_listener_enable。也就是说并不会add到event_base中。

event_listener_enable函数里面的宏EVUTIL_UPCAST可以根据结构体成员变量的地址推算出结构体的起始地址。有关这个宏，可以查看”结构体偏移量”。

处理客户端的连接请求

现在来看一下event的回调函数listener_read_cb。

```
//listener.c文件
static void
listener_read_cb(evutil_socket_t fd, short what, void *p)
{
    struct evconnlistener *lev = p;
    int err;
    evconnlistener_cb cb;
    evconnlistener_errorcb errorcb;
    void *user_data;
    LOCK(lev);
    while (1) { //可能有多个客户端同时请求连接
        struct sockaddr_storage ss;
#ifdef WIN32
        int socklen = sizeof(ss);
#else
        socklen_t socklen = sizeof(ss);
#endif
        evutil_socket_t new_fd = accept(fd, (struct sockaddr*)&ss, &sockl
en);
        if (new_fd < 0)
            break;
        if (socklen == 0) {
            /* This can happen with some older linux kernels in
             * response to nmap. */
            evutil_closesocket(new_fd);
            continue;
        }

        if (!(lev->flags & LEV_OPT_LEAVE_SOCKETS_BLOCKING))
            evutil_make_socket_nonblocking(new_fd);

        //用户还没设置连接监听器的回调函数
        if (lev->cb == NULL) {
            UNLOCK(lev);
            return;
        }

        //由于refcnt被初始化为1.这里有++了，所以一般情况下并不会进入下面的
        //if判断里面。但如果程在下面UNLOCK之后，第二个线调用evconnlistener_free
        //释放这个evconnlistener时，就有可能使得refcnt为1了。即进入那个判断体里
        //执行listener_decref_and_unlock。在下面会讨论这个问题。
        ++lev->refcnt;
        cb = lev->cb;
        user_data = lev->user_data;
        UNLOCK(lev);
        cb(lev, new_fd, (struct sockaddr*)&ss, (int)socklen,
            user_data);//调用用户设置的回调函数，让用户处理这个fd
        LOCK(lev);
        if (lev->refcnt == 1) {
            int freed = listener_decref_and_unlock(lev);
            EVUTIL_ASSERT(freed);
```

```
                return;
            }
            --lev->refcnt;
        }

        err = evutil_socket_geterror(fd);
        if (EVUTIL_ERR_ACCEPT_RETRIABLE(err)) {//还可以accept
            UNLOCK(lev);
            return;
        }

        //当有错误发生时才会运行到这里
        if (lev->errorcb != NULL) {
            ++lev->refcnt;
            errorcb = lev->errorcb;
            user_data = lev->user_data;
            UNLOCK(lev);
            errorcb(lev, user_data);//调用用户设置的错误回调函数
            LOCK(lev);
            listener_decref_and_unlock(lev);
        }
    }
```

这个函数所做的工作也比较简单，就是accept客户端，然后调用用户设置的回调函数。所以，用户回调函数的参数fd是一个已经连接好了的socket。

上面函数说到了错误回调函数，可以通过下面的函数设置连接监听器的错误监听函数。

```
//listener.h文件
typedef void (*evconnlistener_errorcb)(struct evconnlistener *, void *);


//listener.c文件
void
evconnlistener_set_error_cb(struct evconnlistener *lev,
    evconnlistener_errorcb errorcb)
{
    LOCK(lev);
    lev->errorcb = errorcb;
    UNLOCK(lev);
}
```

释放evconnlistener

调用evconnlistener_free可以释放一个evconnlistener。由于evconnlistener拥有一些系统资源，在释放evconnlistener_free的时候会释放这些系统资源。

```
//listener.c文件
void
evconnlistener_free(struct evconnlistener *lev)
{
    LOCK(lev);
    lev->cb = NULL;
    lev->errorcb = NULL;
    if (lev->ops->shutdown)//这里的shutdown为NULL
        lev->ops->shutdown(lev);

    //引用次数减一，并解锁
    listener_decref_and_unlock(lev);
}

static int
listener_decref_and_unlock(struct evconnlistener *listener)
{
    int refcnt = --listener->refcnt;
    if (refcnt == 0) {
        //实际调用event_listener_destroy
        listener->ops->destroy(listener);
        UNLOCK(listener);
        //释放锁
        EVTHREAD_FREE_LOCK(listener->lock, EVTHREAD_LOCKTYPE_RECURSIVE);
        mm_free(listener);
        return 1;
    } else {
        UNLOCK(listener);
        return 0;
    }
}

static void
event_listener_destroy(struct evconnlistener *lev)
{
    struct evconnlistener_event *lev_e =
        EVUTIL_UPCAST(lev, struct evconnlistener_event, base);

    //把event从event_base中删除
    event_del(&lev_e->listener);
    if (lev->flags & LEV_OPT_CLOSE_ON_FREE)//如果用户设置了这个选项，那么要关闭
socket
        evutil_closesocket(event_get_fd(&lev_e->listener));
}
```

要注意一点，LEV_OPT_CLOSE_ON_FREE选项关闭的是服务器端的监听socket，而非那些连接客户端的socket。

现在来说一下那个listener_decref_and_unlock。前面注释说到，在函数listener_read_cb中，一般情况下是不会调用listener_decref_and_unlock，但在多线程的时候可能会调用。这种特殊情况是：当主线程accept到一个新客户端时，会解锁，并调用用户设置的回调函数。此时，引用计数等于2。就在这个时候，第二个线程执行evconnlistener_free函数。该函数会执行listener_decref_and_unlock。明显主线程还在用这个evconnlistener，肯定不能删除。此时引用计数也等于2也不会删除。但用户已经调用了evconnlistener_free。Libevent必须要响应。当第二个线程执行完后，主线程抢到CPU，此时引用计数就变成1了，也就进入到if判断里面了。在判断体里面执行函数listener_decref_and_unlock，并且完成删除工作。

总得来说，Libevent封装的这个evconnlistener和一系列操作函数，还是比较简单的。思路也比较清晰。

参考：http://www.wangafu.NET/~nickm/libevent-book/Ref8_listener.html

22.evbuffer结构与基本操作

原文地址

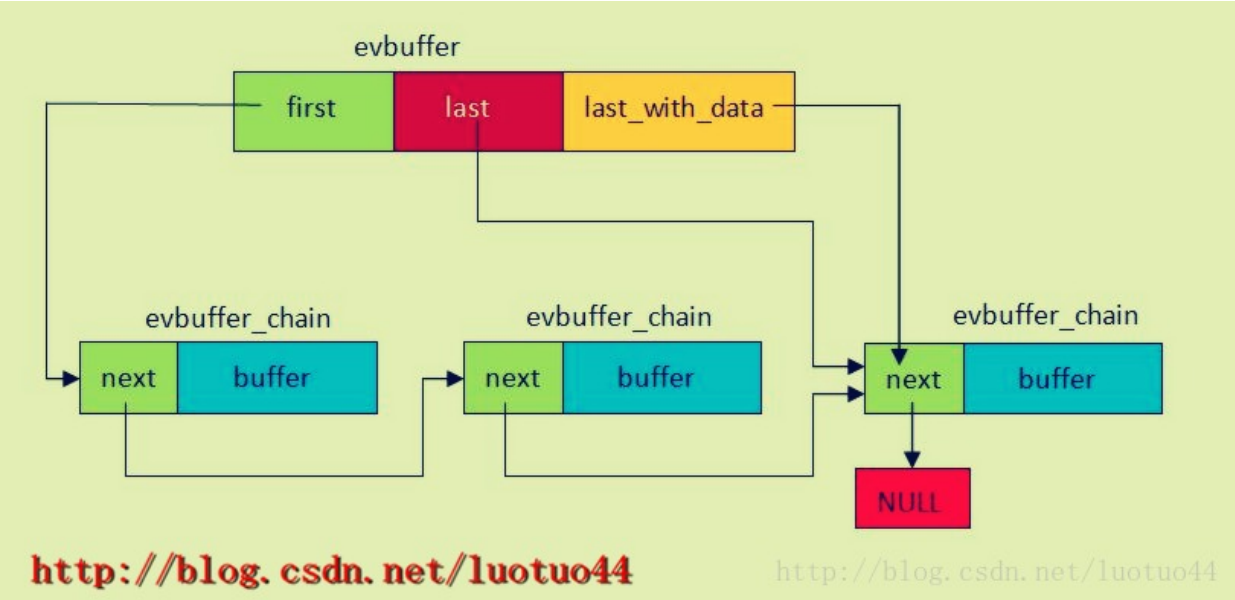
对于非阻塞IO的网络库来说，buffer几乎是必须的。Libevent在1.0版本之前就提供了buffer功能。现在来看一下Libevent的buffer。

buffer相关结构体

Libevent为buffer定义了下面的结构体：

```
//evbuffer-internal.h文件
struct evbuffer_chain;
struct evbuffer {
    struct evbuffer_chain *first;
    struct evbuffer_chain *last;
    //这是一个二级指针。使用*last_with_datap时，指向的是链表中最后一个有数据的evbuf
fer_chain。
    //所以last_with_datap存储的是倒数第二个evbuffer_chain的next成员地址。
    //一开始buffer->last_with_datap = &buffer->first;此时first为NULL。所以当
链表没有节点时
    //*last_with_datap为NULL。当只有一个节点时*last_with_datap就是first。

    struct evbuffer_chain **last_with_datap;

    size_t total_len;//链表中所有chain的总字节数

    ...
};


struct evbuffer_chain {
    struct evbuffer_chain *next;
    size_t buffer_len;//buffer的大小

    //错开不使用的空间。该成员的值一般等于0
    ev_off_t misalign;

    //evbuffer_chain已存数据的字节数
    //所以要从buffer + misalign + off的位置开始写入数据
    size_t off;

    ...

    unsigned char *buffer;
};
```

这两个结构体配合工作得到下图所示的存储结构：



因为last_with_datap成员比较特殊，上图只是展示了一种情况。后面还有一张图，展示另外一种情况。

Libevent将缓冲数据都存放到buffer中。通过一个个的evbuffer_chain连成的链表可以存放很多的缓冲数据。

这是一个很常见的链表形式。但Libevent有一个很独特的地方，就是那个evbuffer_chain结构体。

首先，该结构体有misalign成员。该成员表示错开不用的buffer空间。也就是说buffer中真正的数据是从buffer + misalign开始。

第二，evbuffer_chain结构体buffer是一个指针，按道理来说，应该单独调用malloc分配一个堆内存并让buffer指向之。但实际上buffer指向的内存和evbuffer_chain结构体本身的存储内存是一起分配的。下面代码展示了这一点：

```
//evbuffer-internal.h文件
#define EVBUFFER_CHAIN_SIZE sizeof(struct evbuffer_chain)

#if _EVENT_SIZEOF_VOID_P < 8
#define MIN_BUFFER_SIZE 512
#else
#define MIN_BUFFER_SIZE 1024
#endif

//宏的作用就是返回，chain + sizeof(evbuffer_chain) 的内存地址。
#define EVBUFFER_CHAIN_EXTRA(t, c) (t *)((struct evbuffer_chain *)(c) +
1)


//buffer.c文件
static struct evbuffer_chain *
evbuffer_chain_new(size_t size)//size是buffer所需的大小
{
    struct evbuffer_chain *chain;
    size_t to_alloc;

    //所需的大小size 再 加上evbuffer_chain结构体本身所需
    //的内存大小。这样做的原因是，evbuffer_chain本身是管理
    //buffer的结构体。但buffer内存就分配在evbuffer_chain结构体存储
    //内存的后面。所以要申请多一些内存。
    size += EVBUFFER_CHAIN_SIZE;//evbuffer_chain结构体本身的大小


    to_alloc = MIN_BUFFER_SIZE; //内存块的最小值
    while (to_alloc < size)
        to_alloc <<= 1;
   //从分配的内存大小可以知道，evbuffer_chain结构体和buffer是一起分配的
    //也就是说他们是存放在同一块内存中
    if ((chain = mm_malloc(to_alloc)) == NULL)
        return (NULL);

    //只需初始化最前面的结构体部分即可
    memset(chain, 0, EVBUFFER_CHAIN_SIZE);

    //buffer_len存储的是buffer的大小
    chain->buffer_len = to_alloc - EVBUFFER_CHAIN_SIZE;

     //宏的作用就是返回，chain + sizeof(evbuffer_chain) 的内存地址。
     //其效果就是buffer指向的内存刚好是在evbuffer_chain的后面。
    chain->buffer = EVBUFFER_CHAIN_EXTRA(u_char, chain);

    return (chain);
}
```

前面的图中，buffer内存区域(蓝色区域)连在next的后面也是基于这一点的。在代码的while循环中也可以看到申请的空间大小是512的倍数，也就是说evbuffer_chain申请的空间大小是512、1024、2048、4096……

上面贴出了函数evbuffer_chain_new，该函数是用来创建一个evbuffer_chain。现在贴出另外一个函数evbuffer_new，它是用来创建一个evbuffer的。

```
//buffer.c
struct evbuffer *
evbuffer_new(void)
{
    struct evbuffer *buffer;

    buffer = mm_calloc(1, sizeof(struct evbuffer));
    if (buffer == NULL)
        return (NULL);

    buffer->refcnt = 1;
    buffer->last_with_datap = &buffer->first;

    return (buffer);
}
```

Buffer的数据操作

在链表尾添加数据

Libevent提供给用户的添加数据接口是evbuffer_add，现在就通过这个函数看一下是怎么将数据插入到buffer中的。该函数是在链表的尾部添加数据，如果想在链表的前面添加数据可以使用evbuffer_prepend。在链表尾部插入数据，分下面几种情况：

1. 该链表为空，即这是第一次插入数据。这是最简单的，直接把新建的evbuffer_chain插入到链表中，通过调用evbuffer_chain_insert。
2. 链表的最后一个节点(即evbuffer_chain)还有一些空余的空间，放得下本次要插入的数据。此时直接把数据追加到最后一个节点即可。
3. 链表的最后一个节点并不能放得下本次要插入的数据，那么就需要把本次要插入的数据分开由两个evbuffer_chain存放。

具体的实现如下面所示：

```
//buffer.c文件
int
evbuffer_add(struct evbuffer *buf, const void *data_in, size_t datlen)
{
    struct evbuffer_chain *chain, *tmp;
    const unsigned char *data = data_in;
    size_t remain, to_alloc;
    int result = -1;

    EVBUFFER_LOCK(buf);//加锁,线程安全

    //冻结缓冲区尾部，禁止追加数据
    if (buf->freeze_end) {
        goto done;
    }

    //找到最后一个evbuffer_chain。
    chain = buf->last;

    //第一次插入数据时，buf->last为NULL
    if (chain == NULL) {
        chain = evbuffer_chain_new(datlen);
        if (!chain)
            goto done;
        evbuffer_chain_insert(buf, chain);
    }

    //EVBUFFER_IMMUTABLE 是 read-only chain
    if ((chain->flags & EVBUFFER_IMMUTABLE) == 0) {//等于0说明是可以写的
        //最后那个chain可以放的字节数
        remain = (size_t)(chain->buffer_len - chain->misalign - chain->of
f);
        if (remain >= datlen) {//最后那个chain可以放下本次要插入的数据

            memcpy(chain->buffer + chain->misalign + chain->off,
                data, datlen);
            chain->off += datlen;//偏移量，方便下次插入数据
            buf->total_len += datlen;//buffer的总字节数
            goto out;
        } else if (!CHAIN_PINNED(chain) &&//该evbuffer_chain可以修改
            evbuffer_chain_should_realign(chain, datlen)) {
            //通过调整后，也可以放得下本次要插入的数据

            //通过使用chain->misalign这个错位空间而插入数据
            evbuffer_chain_align(chain);

            memcpy(chain->buffer + chain->off, data, datlen);
            chain->off += datlen;
            buf->total_len += datlen;
            goto out;
        }
```

```
        } else {
            remain = 0; //最后一个节点是只写evbuffer_chain
        }

        //当这个evbuffer_chain是一个read-only buffer或者最后那个chain
        //放不下本次要插入的数据时才会执行下面代码
        //此时需要新建一个evbuffer_chain
        to_alloc = chain->buffer_len;
        //当最后evbuffer_chain的缓冲区小于等于2048时，那么新建的evbuffer_chain的
        //大小将是最后一个节点缓冲区的2倍。
        if (to_alloc <= EVBUFFER_CHAIN_MAX_AUTO_SIZE/2)//4096/2
            to_alloc <<= 1;

        //最后的大小还是有要插入的数据决定。要注意的是虽然to_alloc最后的值可能为
        //datlen。但在evbuffer_chain_new中，实际分配的内存大小必然是512的倍数。
        if (datlen > to_alloc)
            to_alloc = datlen;

        //此时需要new一个chain才能保存本次要插入的数据
        tmp = evbuffer_chain_new(to_alloc);
        if (tmp == NULL)
            goto done;

        //链表最后那个节点还是可以放下一些数据的。那么就先填满链表最后那个节点
        if (remain) {
            memcpy(chain->buffer + chain->misalign + chain->off,
                data, remain);
            chain->off += remain;
            buf->total_len += remain;
            buf->n_add_for_cb += remain;
        }

        data += remain;//要插入的数据指针
        datlen -= remain;

        //把要插入的数据复制到新建一个chain中。
        memcpy(tmp->buffer, data, datlen);
        tmp->off = datlen;
        //将这个chain插入到evbuffer中
        evbuffer_chain_insert(buf, tmp);
        buf->n_add_for_cb += datlen;

    out:
        evbuffer_invoke_callbacks(buf);//调用回调函数
        result = 0;
    done:
        EVBUFFER_UNLOCK(buf);//解锁
        return result;
    }
```

可以看到，evbuffer_add函数是复制一份数据，保存在链表中。这样做的好处是，用户调用该函数后，就可以丢弃该数据。读者比较熟知的函数bufferevent_write就是直接调用这个函数。当用户调用bufferevent_write后，就可以马上把数据丢弃，无需等到Libevent把这份数据写到socket的缓存区中。

前面的代码是把数据存放到evbuffer_chain中，至于怎么把evbuffer_chain插入到链表中，则是由函数evbuffer_chain_insert完成。

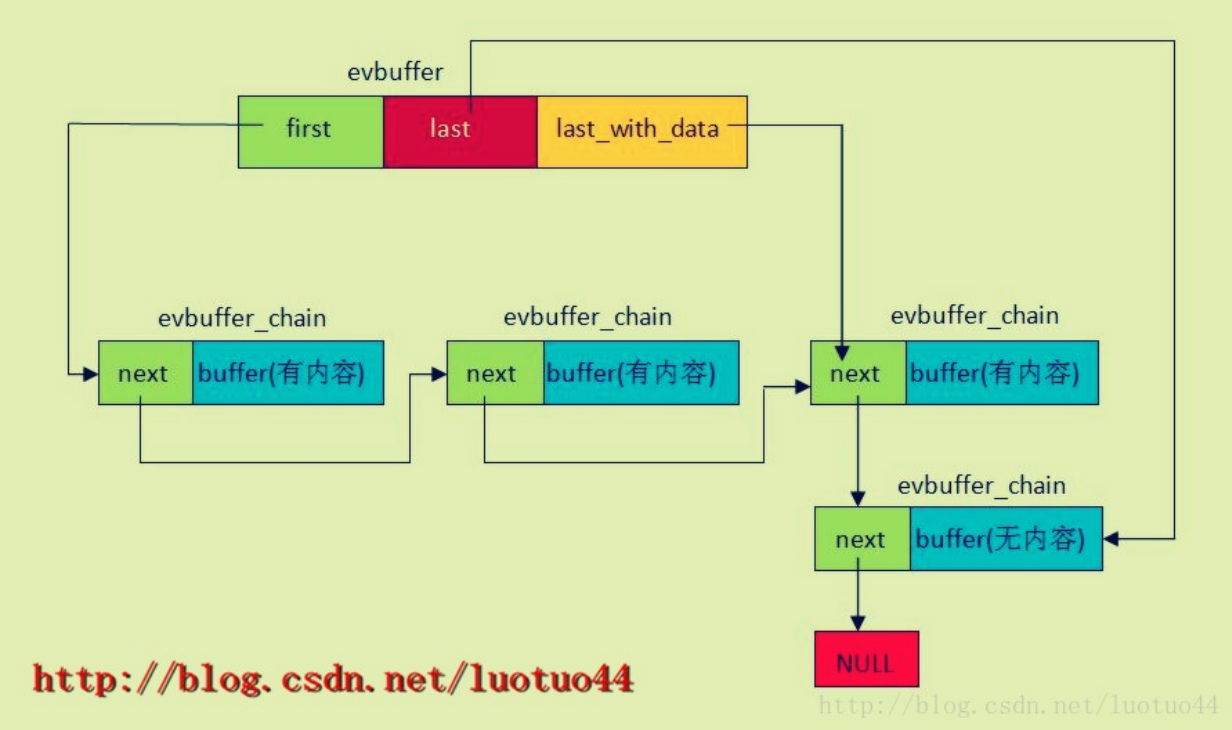
```
//buffer.c文件
static void
evbuffer_chain_insert(struct evbuffer *buf,
    struct evbuffer_chain *chain)
{
    //新建evbuffer时是把整个evbuffer结构体都赋值0，
    //并有buffer->last_with_datap = &buffer->first;
    //所以*buf->last_with_datap就是first的值，所以一开始为NULL
    if (*buf->last_with_datap == NULL) {
        buf->first = buf->last = chain;
    } else {
        struct evbuffer_chain **ch = buf->last_with_datap;
        /* Find the first victim chain.  It might be *last_with_datap */
        //(*ch)->off != 0表示该evbuffer_chain有数据了
        //CHAIN_PINNED(*ch)则表示该evbuffer_chain不能被修改
        //在链表中寻找到一个可以使用的evbuffer_chain.
        //可以使用是指该chain没有数据并且可以修改。
        while ((*ch) && ((*ch)->off != 0 || CHAIN_PINNED(*ch)))
            ch = &(*ch)->next;//取的还是next地址。 这样看&((*ch)->next)更清晰

        //在已有的链表中找不到一个满足条件的evbuffer_chain。一般都是这种情况
        if (*ch == NULL) {
            /* There is no victim; just append this new chain. */
            //此时buf->last指向的chain不再是最后了。因为last->next被赋值了
            buf->last->next = chain;

            if (chain->off)//要插入的这个chain是有数据的
                buf->last_with_datap = &buf->last->next;//last_with_datap
指向的是倒数第二个有数据的chain的next
        } else {//这种情况得到的链表可以参考下图
            /* Replace all victim chains with this chain. */
            //断言，从这个节点开始，后面的说有节点都是没有数据的
            EVUTIL_ASSERT(evbuffer_chains_all_empty(*ch));
            //释放从这个节点开始的余下链表节点
            evbuffer_free_all_chains(*ch);

            //把这个chain插入到最后
            *ch = chain;
        }
        buf->last = chain;//重新设置last指针，让它指向最后一个chain
    }
    buf->total_len += chain->off;
}

static void
evbuffer_free_all_chains(struct evbuffer_chain *chain)
{
    struct evbuffer_chain *next;
    for (; chain; chain = next) {//遍历余下的链表，删除之
        next = chain->next;
        evbuffer_chain_free(chain);
```

```
    }
}


static inline void
evbuffer_chain_free(struct evbuffer_chain *chain)
{
    ...//特殊buffer缓冲数据。一般的不用这些操作。直接释放内存即可
    mm_free(chain);
}
```

可以看到，evbuffer_chain_insert的插入并不是已经一个简单的链表插入，还要检测链表里面是否有没有数据(off为0)的节点。但这个buffer链表里面会有这样的节点吗？其实是有这样节点，这种节点一般是用于预留空间的。预留空间这个概念在STL中是很常见的，它的主要作用是使得当下次添加数据时，无需额外申请空间就能保存数据。

预留buffer空间

其中一个扩大预留空间的函数是evbuffer_expand。在讲evbuffer_expand前，看一下如果存在没有数据(off为0)的节点，链表又会是怎么样的。这涉及到last_with_data指针的指向，如下图所示：



好了，现在来说一下evbuffer_expand。

```
//buffer.c文件
int
evbuffer_expand(struct evbuffer *buf, size_t datlen)
{
    struct evbuffer_chain *chain;

    EVBUFFER_LOCK(buf);//加锁
    chain = evbuffer_expand_singlechain(buf, datlen);
    EVBUFFER_UNLOCK(buf);//解释
    return chain ? 0 : -1;
}
```

该函数的作用是扩大链表的buffer空间，使得下次add一个长度为datlen的数据时，无需动态申请内存。

由于确保的是无需动态申请内存，所以假如这个链表本身还有大于datlen的空闲空间，那么这个evbuffer_expand函数将不做任何操作。

如果这个链表的所有buffer空间都被用完了，那么解决需要创建一个buffer为datlen的evbuffer_chain，然后把这个evbuffer_chain插入到链表最后面即可。此时这个evbuffer_chain的off就等于0了，也就出现了前面说的的那个问题。

如果链表的最后一个有数据chain还有一些空闲空间，但小于datlen。那么就有点麻烦。evbuffer_expand 是调用evbuffer_expand_singlechain实现扩大空间的。而evbuffer_expand_singlechain函数有一个特点，预留空间datlen必须是在一个evbuffer_chain中，不能跨chain。该函数的返回值就指明了哪个chain预留了datlen空间。不能跨chain也就导致了一些麻烦事。

由于不能跨chain，但最后一个chain确实又还有一些空闲空间。前面的evbuffer_add函数会把链表的所有节点的buffer都填得满满的。这说明所有节点的buffer还是用完的好，比较统一。要明确的是，此种情况下，肯定是要新建一个evbuffer_chain插入到后面。

Libevent还是想把所有节点的buffer都填满。如果最后一个chain的数据比较少，那么就直接不要那个chain。当然chain上的数据还是要的。Libevent新建一个比datlen更大的chain，把最后一个chain上的数据迁移到这个新建的chain上。这样就既能保证该chain节点也能填满，也保证了预留空间datlen必须在是一个chain的。如果最后一个chain的数据比较多，Libevent就认为迁移不划算，那么Libevent就让这个chain最后留有一些空间不使用。

下面是该函数的代码展示了上面所说的：

```
//buffer.c文件

#define MAX_TO_COPY_IN_EXPAND 4096
//计算evbuffer_chain的可用空间是多少
#define CHAIN_SPACE_LEN(ch) ((ch)->flags & EVBUFFER_IMMUTABLE ? \
        0 : (ch)->buffer_len - ((ch)->misalign + (ch)->off))

static struct evbuffer_chain *
evbuffer_expand_singlechain(struct evbuffer *buf, size_t datlen)
{
    struct evbuffer_chain *chain, **chainp;
    struct evbuffer_chain *result = NULL;
    ASSERT_EVBUFFER_LOCKED(buf);

    chainp = buf->last_with_datap;

    //*chainp指向最后一个有数据的evbuffer_chain或者为NULL
    if (*chainp && CHAIN_SPACE_LEN(*chainp) == 0)//CHAIN_SPACE_LEN该chain
可用空间的大小
        chainp = &(*chainp)->next;

    //经过上面的那个if后，当最后一个有数据的evbuffer_chain还有空闲空间时
    //*chainp就指向之。否则*chainp指向最后一个有数据的evbuffer_chain的next。

    chain = *chainp;

    if (chain == NULL ||//这个chain是不可修改的，那么就只能插入一个新的chain了
        (chain->flags & (EVBUFFER_IMMUTABLE|EVBUFFER_MEM_PINNED_ANY))) {
        goto insert_new;
    }

    if (CHAIN_SPACE_LEN(chain) >= datlen) {//这个chain的可用空间大于扩展空间
        result = chain;
        //这种情况，Libevent并不会扩大buffer空间.因为Libevent认为现在的可用空间可
以用作用户提出的预留空间
        goto ok;
    }

    if (chain->off == 0) {//当前一个chain存满了时，就会出现这种情况
        goto insert_new;//插入一个新的chain
    }

     //通过使用misalign错位空间，也能使得可用空间大于等于预留空间，那么也不用
     //扩大buffer空间
    if (evbuffer_chain_should_realign(chain, datlen)) {
        evbuffer_chain_align(chain);
        result = chain;
        goto ok;
    }
```

```
    //空闲空间小于总空间的1/8 或者 已有的数据量大于MAX_TO_COPY_IN_EXPAND(4096)
    if (CHAIN_SPACE_LEN(chain) < chain->buffer_len / 8 ||
        chain->off > MAX_TO_COPY_IN_EXPAND) {//4096

        //本chain有比较多的数据，将这些数据迁移到另外一个chain是不划算的
        //此时，将不会改变这个chain。

        //下一个chain是否可以有足够的空闲空间.有则直接用之
        if (chain->next && CHAIN_SPACE_LEN(chain->next) >= datlen) {
            result = chain->next;
            goto ok;
        } else {
            goto insert_new;
        }
    } else {
        //由于本chain的数据量比较小，所以把这个chain的数据迁移到另外一个
        //chain上是值得的。
        size_t length = chain->off + datlen;
        struct evbuffer_chain *tmp = evbuffer_chain_new(length);
        if (tmp == NULL)
            goto err;

        tmp->off = chain->off;
        //进行数据迁移
        memcpy(tmp->buffer, chain->buffer + chain->misalign,
            chain->off);
        EVUTIL_ASSERT(*chainp == chain);
        result = *chainp = tmp;

        if (buf->last == chain)
            buf->last = tmp;

        tmp->next = chain->next;
        evbuffer_chain_free(chain);
        goto ok;
    }

insert_new:
    result = evbuffer_chain_insert_new(buf, datlen);
    if (!result)
        goto err;
ok:
    EVUTIL_ASSERT(result);
    EVUTIL_ASSERT(CHAIN_SPACE_LEN(result) >= datlen);
err:
    return result;
}


static inline struct evbuffer_chain *
evbuffer_chain_insert_new(struct evbuffer *buf, size_t datlen)
{
```

```
        struct evbuffer_chain *chain;
        if ((chain = evbuffer_chain_new(datlen)) == NULL)
            return NULL;
        evbuffer_chain_insert(buf, chain);
        return chain;
    }
```

上面代码中evbuffer_expand_singlechain函数的第一个if语句，可以联合前面的两张图一起看，更容易看懂。

evbuffer_expand_singlechain函数是要求一个节点就能提供大小为datlen的可用空间。其实Libevent还提供了_evbuffer_expand_fast函数，该函数还有一个整型的参数n，用来表示使用不超过n个节点的前提下，提供datlen的可用空间。不过这个函数只留给Libevent内部使用，用户不能使用之。

```
//buffer.c文件
int//用最多不超过n个节点就提供datlen大小的空闲空间。链表过长是不好的
_evbuffer_expand_fast(struct evbuffer *buf, size_t datlen, int n)
{
    struct evbuffer_chain *chain = buf->last, *tmp, *next;
    size_t avail;
    int used;

    EVUTIL_ASSERT(n >= 2); //n必须大于等于2

    //最后一个节点是不可用的
    if (chain == NULL || (chain->flags & EVBUFFER_IMMUTABLE)) {
        //这种情况下，直接新建一个足够大的evbuffer_chain即可
        chain = evbuffer_chain_new(datlen);
        if (chain == NULL)
            return (-1);

        evbuffer_chain_insert(buf, chain);
        return (0);
    }

    used = 0; /* number of chains we're using space in. */
    avail = 0; /* how much space they have. */
    for (chain = *buf->last_with_datap; chain; chain = chain->next) {
        if (chain->off) {//最后一个有数据的节点的可用空间也是要被使用
            size_t space = (size_t) CHAIN_SPACE_LEN(chain);
            EVUTIL_ASSERT(chain == *buf->last_with_datap);
            if (space) {
                avail += space;
                ++used;
            }
        } else {//链表中off为0的空buffer统统使用
            /* No data in chain; realign it. */
            chain->misalign = 0;
            avail += chain->buffer_len;
            ++used;
        }
        if (avail >= datlen) {//链表中的节点的可用空间已经足够了
            return (0);
        }
        if (used == n)//到达了最大可以忍受的链表长度
            break;
    }

    //前面的for循环，如果找够了空闲空间，那么是直接return。所以
    //运行到这里时，就说明还没找到空闲空间。一般是因为链表后面的off等于0
    //的节点已经被用完了都还不能满足datlen
    if (used < n) {
        EVUTIL_ASSERT(chain == NULL);

        //申请一个足够大的evbuffer_chain，把空间补足
```

```
            tmp = evbuffer_chain_new(datlen - avail);
            if (tmp == NULL)
                return (-1);

            buf->last->next = tmp;
            buf->last = tmp;
            return (0);
        } else { //used == n。把后面的n个节点都用了还是不够datlen空间
            //链表后面的n个节点都用上了，这个n个节点中，至少有n-1个节点的off等于
            //0。n个节点都不够，Libevent就认为这些节点都是饭桶，Libevent会统统删除
            //然后新建一个足够大的evbuffer_chain。

            //用来标志该链表的所有节点都是off为0的。在这种情况下，将删除所有的节点
            int rmv_all = 0; /* True iff we removed last_with_data. */
            chain = *buf->last_with_datap;
            if (!chain->off) {
                //这说明链表中的节点都是没有数据的evbuffer_chain
                EVUTIL_ASSERT(chain == buf->first);
                rmv_all = 1;//标志之
                avail = 0;
            } else {
                //最后一个有数据的chain的可用空间的大小。这个空间是可以用上的
                avail = (size_t) CHAIN_SPACE_LEN(chain);
                chain = chain->next;
            }


            //chain指向第一个off等于0的evbuffer_chain 或者等于NULL

            //将这些off等于0的evbuffer_chain统统free掉，不要了。
            //然后new一个足够大的evbuffer_chain即可。这能降低链表的长度
            for (; chain; chain = next) {
                next = chain->next;
                EVUTIL_ASSERT(chain->off == 0);
                evbuffer_chain_free(chain);
            }

            //new一个足够大的evbuffer_chain
            tmp = evbuffer_chain_new(datlen - avail);
            if (tmp == NULL) {//new失败
                if (rmv_all) {//这种情况下，该链表就根本没有节点了
                    ZERO_CHAIN(buf);//相当于初始化evbuffer的链表
                } else {
                    buf->last = *buf->last_with_datap;
                    (*buf->last_with_datap)->next = NULL;
                }
                return (-1);
            }

            if (rmv_all) {//这种情况下，该链表就只有一个节点了
                buf->first = buf->last = tmp;
                buf->last_with_datap = &buf->first;
```

```
            } else {
                (*buf->last_with_datap)->next = tmp;
                buf->last = tmp;
            }
            return (0);
        }
    }
```

在链表头添加数据

前面的evbuffer_add是在链表尾部追加数据，Libevent提供了另外一个函数evbuffer_prepend可以在链表头部添加数据。在这个函数里面可以看到evbuffer_chain结构体成员misalign的一些使用，也能知道为什么会有这个成员。

evbuffer_prepend函数并不复杂，只需弄懂misalign的作用就很容易明白该函数的实现。考虑这种情况：要在链表头插入数据，那么应该new一个新的evbuffer_chain，然后把要插入的数据放到这个新建个的evbuffer_chain中。但evbuffer_chain_new申请到的buffer空间可能会大于要插入的数据长度。插入数据后，buffer就必然会剩下一些空闲空间。那么这个空闲空间放在buffer的前面好还是后面好呢？Libevent认为放在前面会好些，此时misalign就有用了。它表示错开不用的空间，也就是空闲空间。如果再次在链表头插入数据，就可以使用到这些空闲空间了。所以，**misalign也可以认为是空闲空间，可以随时使用。**

```
//buffer.c文件
int
evbuffer_prepend(struct evbuffer *buf, const void *data, size_t datlen)
{
    struct evbuffer_chain *chain, *tmp;
    int result = -1;

    EVBUFFER_LOCK(buf);

    //冻结缓冲区头部，禁止在头部添加数据
    if (buf->freeze_start) {
        goto done;
    }

    chain = buf->first;

    //该链表暂时还没有节点
    if (chain == NULL) {
        chain = evbuffer_chain_new(datlen);
        if (!chain)
            goto done;
        evbuffer_chain_insert(buf, chain);
    }

    if ((chain->flags & EVBUFFER_IMMUTABLE) == 0) {//该chain可以修改
        /* If this chain is empty, we can treat it as
         * 'empty at the beginning' rather than 'empty at the end' */
        if (chain->off == 0)
            chain->misalign = chain->buffer_len;

        //考虑这种情况:一开始chain->off等于0，之后调用evbuffer_prepend插入
        //一些数据(还没填满这个chain),之后再次调用evbuffer_prepend插入一些
        //数据。这样就能分别进入下面的if else了

        if ((size_t)chain->misalign >= datlen) {//空闲空间足够大
            memcpy(chain->buffer + chain->misalign - datlen,
                data, datlen);
            chain->off += datlen;
            chain->misalign -= datlen;
            buf->total_len += datlen;
            buf->n_add_for_cb += datlen;
            goto out;
        } else if (chain->misalign) {//不够大，但也要用
            memcpy(chain->buffer,//用完这个chain,所以从头开始
                (char*)data + datlen - chain->misalign,
                (size_t)chain->misalign);
            chain->off += (size_t)chain->misalign;
            buf->total_len += (size_t)chain->misalign;
            buf->n_add_for_cb += (size_t)chain->misalign;
            datlen -= (size_t)chain->misalign;
            chain->misalign = 0;
```

```
            }
        }


        //为datlen申请一个evbuffer_chain。把datlen长的数据放到这个新建的chain
        if ((tmp = evbuffer_chain_new(datlen)) == NULL)
            goto done;
        buf->first = tmp;
        if (buf->last_with_datap == &buf->first)
            buf->last_with_datap = &tmp->next;

        tmp->next = chain;

        tmp->off = datlen;
        tmp->misalign = tmp->buffer_len - datlen;

        memcpy(tmp->buffer + tmp->misalign, data, datlen);
        buf->total_len += datlen;
        buf->n_add_for_cb += (size_t)chain->misalign;

    out:
        evbuffer_invoke_callbacks(buf);//调用回调函数
        result = 0;
    done:
        EVBUFFER_UNLOCK(buf);
        return result;
    }
```

读取数据

现在来看一下怎么从evbuffer中复制一些数据。Libevent提供了函数evbuffer_copyout用来复制evbuffer的数据。当然是从链表的前面开始复制。

```
//buffer.c文件
ev_ssize_t
evbuffer_copyout(struct evbuffer *buf, void *data_out, size_t datlen)
{
    struct evbuffer_chain *chain;
    char *data = data_out;
    size_t nread;
    ev_ssize_t result = 0;

    EVBUFFER_LOCK(buf);

    chain = buf->first;

    if (datlen >= buf->total_len)
        datlen = buf->total_len;//最大能提供的数据

    if (datlen == 0)
        goto done;

    //冻结缓冲区头部，禁止读取缓冲区的数据
    if (buf->freeze_start) {
        result = -1;
        goto done;
    }

    nread = datlen;
    while (datlen && datlen >= chain->off) {
        memcpy(data, chain->buffer + chain->misalign, chain->off);
        data += chain->off;
        datlen -= chain->off;

        chain = chain->next;
    }

    if (datlen) {
        memcpy(data, chain->buffer + chain->misalign, datlen);
    }

    result = nread;
done:
    EVBUFFER_UNLOCK(buf);
    return result;
}
```

这个函数逻辑比较简单，这里就不多讲了。

有时我们不仅仅想复制数据，还想删除数据，或者是复制后就删除数据。这些操作在socket编程中还是很常见的。

```
//buffer.c文件
int
evbuffer_drain(struct evbuffer *buf, size_t len)
{
    struct evbuffer_chain *chain, *next;
    size_t remaining, old_len;
    int result = 0;

    EVBUFFER_LOCK(buf);
    old_len = buf->total_len;

    if (old_len == 0)
        goto done;

    //冻结缓冲区头部，禁止删除头部数据
    if (buf->freeze_start) {
        result = -1;
        goto done;
    }

    //要删除的数据量大于等于已有的数据量。并且这个evbuffer是可以删除的
    if (len >= old_len && !HAS_PINNED_R(buf)) {
        len = old_len;
        for (chain = buf->first; chain != NULL; chain = next) {
            next = chain->next;
            evbuffer_chain_free(chain);
        }

        ZERO_CHAIN(buf);//相当于初试化evbuffer的链表
    } else {
        if (len >= old_len)
            len = old_len;

        buf->total_len -= len;
        remaining = len;
        for (chain = buf->first;
             remaining >= chain->off;
             chain = next) {
            next = chain->next;
            remaining -= chain->off;

            //已经删除到最后一个有数据的evbuffer_chain了
            if (chain == *buf->last_with_datap) {
                buf->last_with_datap = &buf->first;
            }

            //删除到倒数第二个有数据的evbuffer_chain
            if (&chain->next == buf->last_with_datap)
                buf->last_with_datap = &buf->first;

            //这个chain被固定了，不能删除
```

```
            if (CHAIN_PINNED_R(chain)) {
                EVUTIL_ASSERT(remaining == 0);
                chain->misalign += chain->off;
                chain->off = 0;
                break;//后面的evbuffer_chain也是固定的
            } else
                evbuffer_chain_free(chain);
        }

        buf->first = chain;
        if (chain) {
            chain->misalign += remaining;
            chain->off -= remaining;
        }
    }

    evbuffer_invoke_callbacks(buf);//因为删除数据，所以也要调用回调函数
done:
    EVBUFFER_UNLOCK(buf);
    return result;
}


int
evbuffer_remove(struct evbuffer *buf, void *data_out, size_t datlen)
{
    ev_ssize_t n;
    EVBUFFER_LOCK(buf);
    n = evbuffer_copyout(buf, data_out, datlen);
    if (n > 0) {
        if (evbuffer_drain(buf, n)<0)
            n = -1;
    }
    EVBUFFER_UNLOCK(buf);
    return (int)n;
}
```

可以看到evbuffer_remove是先复制数据，然后才删除evbuffer的数据。而evbuffer_drain则直接删除evbuffer的数据，而不会复制。

23.更多evbuffer操作函数

原文地址

锁操作

在上文可以看到很多函数在操作前都需要对这个evbuffer进行加锁。同event_base不同，如果evbuffer支持锁的话，要显式地调用函数evbuffer_enable_locking。

```
//buffer.c文件
int//参数可以是一个锁变量也可以是NULL
evbuffer_enable_locking(struct evbuffer *buf, void *lock)
{
#ifdef _EVENT_DISABLE_THREAD_SUPPORT
    return -1;
#else
    if (buf->lock)
        return -1;

    if (!lock) {
        //自己分配锁变量
        EVTHREAD_ALLOC_LOCK(lock, EVTHREAD_LOCKTYPE_RECURSIVE);
        if (!lock)
            return -1;
        buf->lock = lock;
        //该evbuffer拥有锁，到时需要释放锁内存
        buf->own_lock = 1;
    } else {
        buf->lock = lock;//使用参数提供的锁
        buf->own_lock = 0;//自己没有拥有锁。不需要释放锁内存
    }

    return 0;
#endif
}
```

可以看到，第二个参数可以为NULL。此时函数内部会申请一个锁。明显如果要让evbuffer能使用锁，就必须在一开始就调用evthread_use_windows_threads()或者evthread_use_pthreads(),关于这个可以参考一篇博文。

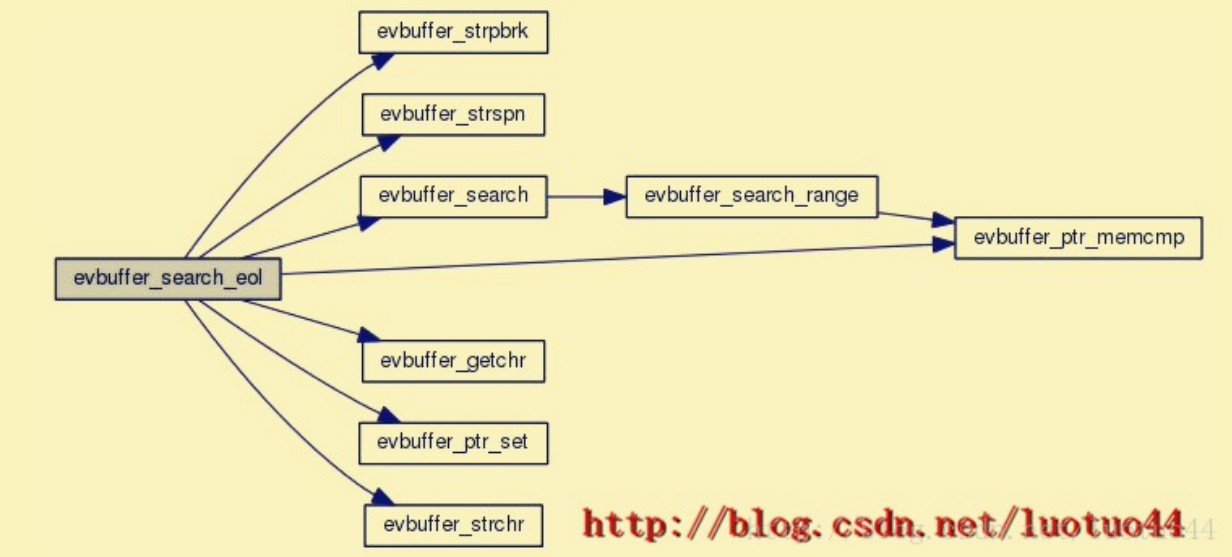
因为用户操控这个evbuffer，所以Libevent提供了加锁和解锁接口给用户使用。

```
//buffer.c文件
void
evbuffer_lock(struct evbuffer *buf)
{
    EVBUFFER_LOCK(buf);
}

void
evbuffer_unlock(struct evbuffer *buf)
{
    EVBUFFER_UNLOCK(buf);
}

//evbuffer-internal.h文件
#define EVBUFFER_LOCK(buffer)                       \
    do {                                    \
        EVLOCK_LOCK((buffer)->lock, 0);              \
    } while (0)
#define EVBUFFER_UNLOCK(buffer)                     \
    do {                                    \
        EVLOCK_UNLOCK((buffer)->lock, 0);            \
    } while (0)
```

在Libevent内部，一般不会使用这个两个接口，而是直接使用EVBUFFER_LOCK(buf)和EVBUFFER_UNLOCK(buf)。

查找操作

下图展示了evbuffer的一些查找操作以及调用关系。



查找结构体

对于一个数组或者一个文件，只需一个下标或者偏移量就可以定位查找了。但对于evbuffer来说，它的数据是由一个个的evbuffer_chain用链表连在一起的。所以在evbuffer中定位，不仅仅要有一个偏移量，还要指明是哪个evbuffer_chain，甚至是在evbuffer_chain中的偏移量。因此Libevent定义了一个查找(定位)结构体：

```
//buffer.h文件
struct evbuffer_ptr {
    ev_ssize_t pos;//总偏移量，相对于数据的开始位置

    /* Do not alter the values of fields. */
    struct {
        void *chain;//指明是哪个evbuffer_chain
        size_t pos_in_chain; //在evbuffer_chain中的偏移量
    } _internal;
};
```

有一点要注意，pos_in_chain是从misalign这个错开空间之后计算的，也就是说其实际偏移量为：chain->buffer+ chain->misalign + pos_in_chain。

定位结构体有一个对应的操作函数evbuffer_ptr_set，该函数就像fseek函数那样，可以设置或者移动偏移量，并且可以绝对和相对地移动。

```
//buffer.h文件
enum evbuffer_ptr_how {
    EVBUFFER_PTR_SET, //偏移量是一个绝对位置
    EVBUFFER_PTR_ADD //偏移量是一个相对位置
};


//buffer.c文件
//设置evbuffer_ptr。evbuffer_ptr_set(buf, &pos, 0, EVBUFFER_PTR_SET)
//将这个pos指向链表的开头
//position指明移动的偏移量，how指明该偏移量是绝对偏移量还是相对当前位置的偏移量。
int//这个函数的作用就像C语言中的fseek，设置文件指针的偏移量
evbuffer_ptr_set(struct evbuffer *buf, struct evbuffer_ptr *pos,
    size_t position, enum evbuffer_ptr_how how)
{
    size_t left = position;
    struct evbuffer_chain *chain = NULL;

    EVBUFFER_LOCK(buf);

    //这个switch的作用就是给pos设置新的总偏移量值。
    switch (how) {
    case EVBUFFER_PTR_SET://绝对位置
        chain = buf->first;//从第一个evbuffer_chain算起
        pos->pos = position; //设置总偏移量
        position = 0;
        break;
    case EVBUFFER_PTR_ADD://相对位置
        chain = pos->_internal.chain;//从当前evbuffer_chain算起
        pos->pos += position;//加上相对偏移量
        position = pos->_internal.pos_in_chain;
        break;
    }

    //这个偏移量跨了evbuffer_chain。可能不止跨一个chain。
    while (chain && position + left >= chain->off) {
        left -= chain->off - position;
        chain = chain->next;
        position = 0;
    }


    if (chain) {//设置evbuffer_chain内的偏移量
        pos->_internal.chain = chain;
        pos->_internal.pos_in_chain = position + left;
    } else {//跨过了所有的节点
        pos->_internal.chain = NULL;
        pos->pos = -1;
    }

    EVBUFFER_UNLOCK(buf);
```

```
    return chain != NULL ? 0 : -1;
}
```

可以看到，该函数只考虑了向后面的chain移动定位指针，不能向。当然如果参数position小于0，并且移动时并不会跨越当前的chain，还是可以的。不过最好不要这样做。如果确实想移回头，那么可以考虑下面的操作。

```
pos.position -= 20;//移回头20个字节。
evbuffer_ptr_set(buf, &pos, 0, EVBUFFER_PTR_SET);
```

查找一个字符

有下面代码的前一个函数是可以用来查找一个字符的，第二个函数则是获取对于位置的字符。

```
static inline ev_ssize_t
evbuffer_strchr(struct evbuffer_ptr *it, const char chr);

static inline char//获取对应位置的字符
evbuffer_getchr(struct evbuffer_ptr *it);

static inline int
evbuffer_strspn(struct evbuffer_ptr *ptr, const char *chrset);
```

函数evbuffer_strchr是从it指向的evbuffer_chain开始查找，会往后面的链表查找。it是一个值-结果参数，如果查找到了，那么it将会指明被查找字符的位置，并返回相对于evbuffer的总偏移量(即it->pos)。如果没有找到，就返回-1。由于实现都是一般的字符比较，所以就不列出代码了。函数evbuffer_getchr很容易理解也不列出代码了。

第三个函数evbuffer_strspn的参数chrset虽然是一个字符串，其实内部也是比较字符的。该函数所做的操作和C语言标准库里面的strspn函数是一样的。这里也不多说了。关于strspn函数的理解可以查看这里。

查找一个字符串

字符串的查找函数有evbuffer_search_range和evbuffer_search，后者调用前者完成查找。

在讲查找前，先看一个字符串比较函数evbuffer_ptr_memcmp。该函数是比较某一个字符串和从evbuffer中某个位置开始的字符是否相等。明显比较的时候需要考虑到跨evbuffer_chain的问题。

```
static int //匹配成功会返回0
evbuffer_ptr_memcmp(const struct evbuffer *buf, const struct evbuffer_ptr
 *pos,
    const char *mem, size_t len)
{
    struct evbuffer_chain *chain;
    size_t position;
    int r;

    //链表数据不够
    if (pos->pos + len > buf->total_len)
        return -1;

    //需要考虑这个要匹配的字符串被分散在两个evbuffer_chain中
    chain = pos->_internal.chain;
    position = pos->_internal.pos_in_chain;//从evbuffer_chain中的这个位置开
始
    while (len && chain) {
        size_t n_comparable;//该evbuffer_chain中可以比较的字符数
        if (len + position > chain->off)
            n_comparable = chain->off - position;
        else
            n_comparable = len;
        r = memcmp(chain->buffer + chain->misalign + position, mem,
            n_comparable);
        if (r)//不匹配
            return r;

        //考虑跨evbuffer_chain
        mem += n_comparable;
        len -= n_comparable;//还有这些是没有比较的
        position = 0;
        chain = chain->next;
    }

    return 0;//匹配成功
}
```

该函数首先比较pos指向的当前evbuffer_chain，如果字符mem还有一些字符没有参与比较，那么就需要用下一个evbuffer_chain的数据。

由于evbuffer的数据是由链表组成的，没办法直接用KMP查找算法或者直接调用strstr函数。有了evbuffer_ptr_memcmp函数，读者可能会想，一个字节一个字节地挪动evbuffer的数据，依次调用evbuffer_ptr_memcmp函数。但evbuffer_search_range函数也不是直接调用函数evbuffer_ptr_memcmp的。而是先用字符查找函数，找到要查找字符串中的第一个字符，然后才调用那个函数。下面是具体的代码。

```
//buffer.c文件
struct evbuffer_ptr
evbuffer_search(struct evbuffer *buffer, const char *what, size_t len, co
nst struct evbuffer_ptr *start)
{
    return evbuffer_search_range(buffer, what, len, start, NULL);
}

//参数start和end指明了查找的范围
struct evbuffer_ptr
evbuffer_search_range(struct evbuffer *buffer, const char *what, size_t l
en, const struct evbuffer_ptr *start, const struct evbuffer_ptr *end)
{
    struct evbuffer_ptr pos;
    struct evbuffer_chain *chain, *last_chain = NULL;
    const unsigned char *p;
    char first;

    EVBUFFER_LOCK(buffer);

    //初始化pos
    if (start) {
        memcpy(&pos, start, sizeof(pos));
        chain = pos._internal.chain;
    } else {
        pos.pos = 0;
        chain = pos._internal.chain = buffer->first;
        pos._internal.pos_in_chain = 0;
    }

    if (end)
        last_chain = end->_internal.chain;

    if (!len || len > EV_SSIZE_MAX)
        goto done;

    first = what[0];

    //在本函数里面并不考虑到what的数据量比较链表的总数据量还多。
    //但在evbuffer_ptr_memcmp函数中会考虑这个问题。此时该函数直接返回-1。
    //本函数之所以没有考虑这样情况，可能是因为，在[start, end]之间有多少
    //数据是不值得统计的，时间复杂度是O(n)。不是一个简单的buffer->total_len
    //就能获取到的
    while (chain) {
        const unsigned char *start_at =
            chain->buffer + chain->misalign +
            pos._internal.pos_in_chain;
        //const void * memchr ( const void * ptr, int value, size_t num
 );
        //函数的作用是:在ptr指向的内存块中(长度为num个字节),需找字符value。
        //如果找到就返回对应的位置，找不到返回NULL
```

```
            p = memchr(start_at, first,
                chain->off - pos._internal.pos_in_chain);
            if (p) {//找到了what[0]
                pos.pos += p - start_at;
                pos._internal.pos_in_chain += p - start_at;
                //经过上面的两个 += 后，pos指向了这个chain中出现等于what[0]字符的位置
                //但本函数是要匹配一个字符串，而非一个字符

                //evbuffer_ptr_memcmp比较整个字符串。如果有需要的话，该函数会跨
                //evbuffer_chain进行比较，但不会修改pos。如果成功匹配，那么返回0
                if (!evbuffer_ptr_memcmp(buffer, &pos, what, len)) {//匹配成功
                    //虽然匹配成功了，但可能是用到了end之后的链表数据。这也等于没有找到
                    if (end && pos.pos + (ev_ssize_t)len > end->pos)
                        goto not_found;
                    else
                        goto done;
                }

                //跳过这个等于what[0]的字符
                ++pos.pos;
                ++pos._internal.pos_in_chain;

                //这个evbuffer_chain已经全部都对比过了。没有发现目标
                if (pos._internal.pos_in_chain == chain->off) {
                    chain = pos._internal.chain = chain->next;
                    pos._internal.pos_in_chain = 0;//下一个chain从0开始
                }
            } else {//这个evbuffer_chain都没有找到what[0]
                if (chain == last_chain)
                    goto not_found;

                //此时直接跳过这个evbuffer_chain
                pos.pos += chain->off - pos._internal.pos_in_chain;
                chain = pos._internal.chain = chain->next;
                pos._internal.pos_in_chain = 0;//下一个chain从0开始
            }
        }

    not_found:
        pos.pos = -1;
        pos._internal.chain = NULL;
    done:
        EVBUFFER_UNLOCK(buffer);
        return pos;
    }
```

evbuffer_ptr_memcmp函数和evbuffer_search函数是有区别的。前者只会比较从pos指定位置开始的字符串，不会在另外的地方找一个字符串。而后者则会在后面另外找一个字符串进行比较。

查找换行符

换行符是一个比较重要的符号，例如http协议就基于行的。Libevent实现了一个简单的http服务器，因此在内部Libevent实现了一些读取一行数据函数以及与行相关的操作。

有些系统行尾用\r\n有些则直接用\n，这些不统一给编程造成了一些麻烦。因此在Libevent中定义了一个枚举类型，专门来用表示eol(end of line)的。

- EVBUFFER_EOL_LF：行尾是'\n'字符
- EVBUFFER_EOL_CRLF_STRICT：行尾是"\r\n"，一个回车符一个换行符
- EVBUFFER_EOL_CRLF：行尾是"\r\n"或者'\n'。这个是很有用的，因为可能标准的协议里面要求"\r\n"，但一些不遵循标准的用户可能使用'\n'
- EVBUFFER_EOL_ANY：行尾是任意次序或者任意数量的'\r'或者'\n'。这种格式不是很有用，只是用来向后兼容而已

函数evbuffer_readln是用来读取evbuffer中的一行数据(不会读取行尾符号)。

```
enum evbuffer_eol_style {
    EVBUFFER_EOL_ANY,
    EVBUFFER_EOL_CRLF,
    EVBUFFER_EOL_CRLF_STRICT,
    EVBUFFER_EOL_LF
};

//成功返回读取到的一行数据。否则返回NULL。该行数据会自动加上'\0'结尾
//如果n_read_out不为NULL，则被赋值为读取到的一行的字符数
char *
evbuffer_readln(struct evbuffer *buffer, size_t *n_read_out,
        enum evbuffer_eol_style eol_style)
{
    struct evbuffer_ptr it;
    char *line;
    size_t n_to_copy=0, extra_drain=0;
    char *result = NULL;

    EVBUFFER_LOCK(buffer);

    if (buffer->freeze_start) {
        goto done;
    }

    //根据eol_style行尾类型找到行尾。返回值的位置偏移量就指向那个行尾符号
    //行尾符号前面的evbuffer数据就是一行的内容。extra_drain指明这个行尾
    //有多少个字符。后面需要把这个行尾符号删除，方便以后再次读取一行
    it = evbuffer_search_eol(buffer, NULL, &extra_drain, eol_style);
    if (it.pos < 0)
        goto done;
    n_to_copy = it.pos;//并不包括换行符

    if ((line = mm_malloc(n_to_copy+1)) == NULL) {
        event_warn("%s: out of memory", __func__);
        goto done;
    }

    //复制并删除n_to_copy字节
    evbuffer_remove(buffer, line, n_to_copy);
    line[n_to_copy] = '\0';

    //extra_drain指明是行尾符号占有的字节数。现在要删除之
    evbuffer_drain(buffer, extra_drain);
    result = line;
done:
    EVBUFFER_UNLOCK(buffer);

    if (n_read_out)
        *n_read_out = result ? n_to_copy : 0;
```

```
        return result;
    }
```

在函数内部会申请空间，并且从evbuffer中提取出一行数据。下面看看函数evbuffer_search_eol是怎么实现的。该函数执行查找行尾的工作，它在内部区分4种不同的行尾类型。

```
struct evbuffer_ptr
evbuffer_search_eol(struct evbuffer *buffer,
    struct evbuffer_ptr *start, size_t *eol_len_out,
    enum evbuffer_eol_style eol_style)
{
    struct evbuffer_ptr it, it2;
    size_t extra_drain = 0;
    int ok = 0;

    EVBUFFER_LOCK(buffer);//加锁

    if (start) {
        memcpy(&it, start, sizeof(it));
    } else {//从头开始找
        it.pos = 0;
        it._internal.chain = buffer->first;
        it._internal.pos_in_chain = 0;
    }

    switch (eol_style) {
    case EVBUFFER_EOL_ANY:
        if (evbuffer_find_eol_char(&it) < 0)
            goto done;
        memcpy(&it2, &it, sizeof(it));
        //该case的就是寻找在最前面的任意数量的\r和\n。
        //evbuffer_strspn返回第一个不是\r或者\n的下标。
        //此时extra_drain前面的都是\r或者\n。直接删除即可
        extra_drain = evbuffer_strspn(&it2, "\r\n");
        break;
    case EVBUFFER_EOL_CRLF_STRICT: {//\r\n
        it = evbuffer_search(buffer, "\r\n", 2, &it);
        if (it.pos < 0)//没有找到
            goto done;
        extra_drain = 2;
        break;
    }
    case EVBUFFER_EOL_CRLF://\n或者\r\n
        while (1) {//这个循环的一个chain一个chain地检查的
            //从it指向的evbuffer_chain开始往链表后面查找\n和\r。
            //找到两个中的一个即可。两个都有就优先查找\n。
            //会修改it的内容，使得it指向查找到的位置。查找失败返回-1。
            //如果一个evbuffer_chain有\n或者\r就马上返回。
            if (evbuffer_find_eol_char(&it) < 0)
                goto done;
            if (evbuffer_getchr(&it) == '\n') {//获取对应位置的字符
                extra_drain = 1;
                break;
            } else if (!evbuffer_ptr_memcmp(
                    buffer, &it, "\r\n", 2)) {//如果刚才找到的是\r就再测测是不
是\r\n
                extra_drain = 2;
```

```
                    break;
                } else {
                    //\r的后面不是\n，此时跳过\r，继续查找
                    if (evbuffer_ptr_set(buffer, &it, 1,
                        EVBUFFER_PTR_ADD)<0)
                        goto done;
                }
            }
            break;
        case EVBUFFER_EOL_LF://\n
            if (evbuffer_strchr(&it, '\n') < 0)//没有找到
                goto done;
            extra_drain = 1;
            break;
        default:
            goto done;
        }

        ok = 1;
    done:
        EVBUFFER_UNLOCK(buffer);

        if (!ok) {
            it.pos = -1;
        }
        if (eol_len_out)
            *eol_len_out = extra_drain;

        return it;
    }
```

回调函数

evbuffer有一个回调函数队列成员callbacks，向evbuffer删除或者添加数据时，就会调用这些回调函数。之所以是回调函数队列，是因为一个evbuffer是可以添加多个回调函数的，而且同一个回调函数可以被添加多次。

使用回调函数时有一点要注意：因为当evbuffer被添加或者删除数据时，就会调用这些回调函数，所以在回调函数里面不要添加或者删除数据，不然将导致递归，死循环。

evbuffer的回调函数对bufferevent来说是非常重要的，bufferevent的一些重要功能都是基于evbuffer的回调函数完成的。

回调相关结构体

```
//buffer.h文件
struct evbuffer_cb_info {
    //添加或者删除数据之前的evbuffer有多少字节的数据
    size_t orig_size;
    size_t n_added;//添加了多少数据
    size_t n_deleted;//删除了多少数据

    //因为每次删除或者添加数据都会调用回调函数，所以上面的三个成员只能记录从上一次
    //回调函数被调用后，到本次回调函数被调用这段时间的情况。
};


//两个回调函数类型
typedef void (*evbuffer_cb_func)(struct evbuffer *buffer, const struct ev
buffer_cb_info *info, void *arg);

//buffer_compat.h文件。这个类型的回调函数已经不被推荐使用了
typedef void (*evbuffer_cb)(struct evbuffer *buffer, size_t old_len, size
_t new_len, void *arg);

//evbuffer-internal.h文件
//内部结构体，结构体成员对用户透明
struct evbuffer_cb_entry {
    /** Structures to implement a doubly-linked queue of callbacks */
    TAILQ_ENTRY(evbuffer_cb_entry) next;
    /** The callback function to invoke when this callback is called.
        If EVBUFFER_CB_OBSOLETE is set in flags, the cb_obsolete field is

        valid; otherwise, cb_func is valid. */
    union {//哪个回调类型。一般都是evbuffer_cb_func
        evbuffer_cb_func cb_func;
        evbuffer_cb cb_obsolete;
    } cb;
    void *cbarg;//回调函数的参数
    ev_uint32_t flags;//该回调的标志
};

struct evbuffer {
    ...
    //可以添加多个回调函数。所以需要一个队列存储
    TAILQ_HEAD(evbuffer_cb_queue, evbuffer_cb_entry) callbacks;
};
```

设置回调函数

下面看一下怎么设置回调函数。

```
struct evbuffer_cb_entry *
evbuffer_add_cb(struct evbuffer *buffer, evbuffer_cb_func cb, void *cbar
g)
{
    struct evbuffer_cb_entry *e;
    if (! (e = mm_calloc(1, sizeof(struct evbuffer_cb_entry))))
        return NULL;
    EVBUFFER_LOCK(buffer);//加锁
    e->cb.cb_func = cb;
    e->cbarg = cbarg;
    e->flags = EVBUFFER_CB_ENABLED;//标志位，允许回调
    TAILQ_INSERT_HEAD(&buffer->callbacks, e, next);
    EVBUFFER_UNLOCK(buffer);//解锁
    return e;
}
```

参数cbarg就是回调函数被调用时的那个arg参数，这点对于熟悉Libevent的读者应该不难理解。上面这个函数是被一个evbuffer_cb_entry结构体指针插入到callbacks队列的前面，有关TAILQ_HEAD队列和相关的插入操作可以参考博文《TAILQ_QUEUE队列》。

上面函数返回一个evbuffer_cb_entry结构体指针。用户可以利用这个返回的结构体作一些处理，因为这个结构体已经和添加的回调函数绑定了。比如可以设置这个回调函数的标志值。或者利用这个结构体指针作为标识，从队列中找到这个回调函数并删除之。如下面代码所示:

```
int //设置标志
evbuffer_cb_set_flags(struct evbuffer *buffer,
                struct evbuffer_cb_entry *cb, ev_uint32_t flags)
{
    /* the user isn't allowed to mess with these. */
    flags &= ~EVBUFFER_CB_INTERNAL_FLAGS;
    EVBUFFER_LOCK(buffer);
    cb->flags |= flags;
    EVBUFFER_UNLOCK(buffer);
    return 0;
}

int //清除某个标志
evbuffer_cb_clear_flags(struct evbuffer *buffer,
                struct evbuffer_cb_entry *cb, ev_uint32_t flags)
{
    /* the user isn't allowed to mess with these. */
    flags &= ~EVBUFFER_CB_INTERNAL_FLAGS;
    EVBUFFER_LOCK(buffer);
    cb->flags &= ~flags;
    EVBUFFER_UNLOCK(buffer);
    return 0;
}

int //从队列中删除这个回调函数
evbuffer_remove_cb_entry(struct evbuffer *buffer,
              struct evbuffer_cb_entry *ent)
{
    EVBUFFER_LOCK(buffer);
    TAILQ_REMOVE(&buffer->callbacks, ent, next);
    EVBUFFER_UNLOCK(buffer);
    mm_free(ent);
    return 0;
}


int //根据用户设置的回调函数和回调参数这两个量 从队列中删除
evbuffer_remove_cb(struct evbuffer *buffer, evbuffer_cb_func cb, void *cb
arg)
{
    struct evbuffer_cb_entry *cbent;
    int result = -1;
    EVBUFFER_LOCK(buffer);
    TAILQ_FOREACH(cbent, &buffer->callbacks, next) {
        if (cb == cbent->cb.cb_func && cbarg == cbent->cbarg) {
            result = evbuffer_remove_cb_entry(buffer, cbent);
            goto done;
        }
    }
done:
    EVBUFFER_UNLOCK(buffer);
```

```
    return result;
}

//Libevent还是提供了一个删除所有回调函数的接口
static void
evbuffer_remove_all_callbacks(struct evbuffer *buffer)
{
    struct evbuffer_cb_entry *cbent;

    while ((cbent = TAILQ_FIRST(&buffer->callbacks))) {
        TAILQ_REMOVE(&buffer->callbacks, cbent, next);
        mm_free(cbent);
    }
}
```

前面的代码展示了两个回调函数的类型，分别是evbuffer_cb_func和evbuffer_cb。后者在回调的时候可以得知删除或者添加数据之前的数据量和之后的数据量。但这两个数值都可以通过evbuffer_cb_info获取。所以evbuffer_cb回调类型的优势没有了。此外，还有一个问题。一般是通过evbuffer_setcb函数设置evbuffer_cb类型的回调函数。而这个函数会先删除之前添加的所有回调函数。

```
void
evbuffer_setcb(struct evbuffer *buffer, evbuffer_cb cb, void *cbarg)
{
    EVBUFFER_LOCK(buffer);

    if (!TAILQ_EMPTY(&buffer->callbacks))
        evbuffer_remove_all_callbacks(buffer);//清空之前的回调函数

    if (cb) {
        struct evbuffer_cb_entry *ent =
            evbuffer_add_cb(buffer, NULL, cbarg);
        ent->cb.cb_obsolete = cb;
        ent->flags |= EVBUFFER_CB_OBSOLETE;
    }
    EVBUFFER_UNLOCK(buffer);
}
```

可以看到，evbuffer_setcb为标志位flags加上了EVBUFFER_CB_OBSOLETE属性。从名字可以看到这是一个已经过时的属性。其实evbuffer_setcb已经被推荐使用了。

Libevent如何调用回调函数

下面看一下是怎么调用回调函数的。其实现也简单，直接遍历回调队列，然后依次调用回调函数。

```
static void //在evbuffer_add中调用的该函数,running_deferred为0
evbuffer_run_callbacks(struct evbuffer *buffer, int running_deferred)
{
    struct evbuffer_cb_entry *cbent, *next;
    struct evbuffer_cb_info info;
    size_t new_size;
    ev_uint32_t mask, masked_val;
    int clear = 1;

    if (running_deferred) {
        mask = EVBUFFER_CB_NODEFER|EVBUFFER_CB_ENABLED;
        masked_val = EVBUFFER_CB_ENABLED;
    } else if (buffer->deferred_cbs) {
        mask = EVBUFFER_CB_NODEFER|EVBUFFER_CB_ENABLED;
        masked_val = EVBUFFER_CB_NODEFER|EVBUFFER_CB_ENABLED;
        /* Don't zero-out n_add/n_del, since the deferred callbacks
           will want to see them. */
        clear = 0;
    } else { //一般都是这种情况
        mask = EVBUFFER_CB_ENABLED;
        masked_val = EVBUFFER_CB_ENABLED;
    }

    if (TAILQ_EMPTY(&buffer->callbacks)) {//用户没有设置回调函数
        //清零
        buffer->n_add_for_cb = buffer->n_del_for_cb = 0;
        return;
    }

    //没有添加或者删除数据
    if (buffer->n_add_for_cb == 0 && buffer->n_del_for_cb == 0)
        return;

    new_size = buffer->total_len;
    info.orig_size = new_size + buffer->n_del_for_cb - buffer->n_add_for_
cb;
    info.n_added = buffer->n_add_for_cb;
    info.n_deleted = buffer->n_del_for_cb;
    if (clear) {//清零，为下次计算做准备
        buffer->n_add_for_cb = 0;
        buffer->n_del_for_cb = 0;
    }

    //遍历回调函数队列，调用回调函数
    for (cbent = TAILQ_FIRST(&buffer->callbacks);
         cbent != TAILQ_END(&buffer->callbacks);
         cbent = next) {

        next = TAILQ_NEXT(cbent, next);

        //该回调函数没有enable
```

```
            if ((cbent->flags & mask) != masked_val)
                continue;

            if ((cbent->flags & EVBUFFER_CB_OBSOLETE))//已经不被推荐使用了
                cbent->cb.cb_obsolete(buffer,
                    info.orig_size, new_size, cbent->cbarg);
            else
                cbent->cb.cb_func(buffer, &info, cbent->cbarg);//调用用户设置的
回调函数
        }
    }
```

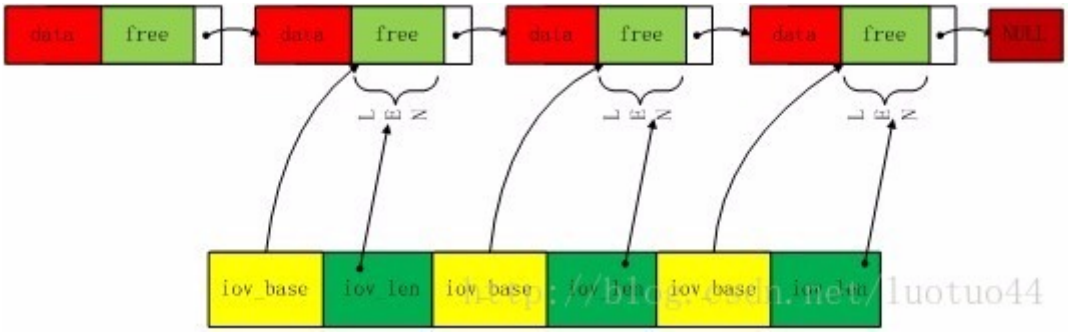
无论是删除数据还是添加数据的函数，例如evbuffer_add和evbuffer_drain函数，都是会调用evbuffer_invoke_callbacks函数的。而这个函数会调用evbuffer_run_callbacks函数。

evbuffer与网络IO

从Socket中读取数据

Libevent通过evbuffer_read函数从一个socket中读取数据到evbuffer中。在读取socket数据之前，Libevent会调用ioctl函数来获取这个socket的读缓冲区中有多少字节，进而确定本次要读多少字节到evbuffer中。Libevent会根据要读取的字节数，在真正read之前会先把evbuffer扩容，免得在read的时候缓冲区不够。

在扩容的时候，如果所在的系统是支持类似readv这样的可以把数据读取到像一个iovec结构体函数，那么Libevent就会选择在n个evbuffer_chain中找到足够的空闲空间(往往通过申请堆空间)，因为这样可以使用类似Linux的iovec结构体。把链表的各个evbuffer_chain的空闲空间的地址赋值给iovec数组(如下图所示)，然后调用readv函数直接读取，readv会把数据读取到相应的chain中。



上图中，有一点是和Libevent的实际情况不符合的。在evbuffer_read中，最后的那个几个evbuffer_chain一般是没有数据的，只有空闲的区域。上图为了好看，就加上了data这区域。

将链表的各个evbuffer_chain的空闲空间的地址赋值给iovec数组，这个操作是由函数_evbuffer_read_setup_vecs完成的。

```
//让vecs数组的指针指向evbuffer中的可用chain.标明哪个chain可用并且从chain的哪里开始，以及可用的字节数
//howmuch是要扩容的大小。vecs、n_vecs_avail分别是iovec数组和数组的大小
//chainp是值-结果参数，它最后指向第一个有可用空间的chain
int
_evbuffer_read_setup_vecs(struct evbuffer *buf, ev_ssize_t howmuch,
    struct evbuffer_iovec *vecs, int n_vecs_avail,
    struct evbuffer_chain ***chainp, int exact)
{
    struct evbuffer_chain *chain;
    struct evbuffer_chain **firstchainp;
    size_t so_far;
    int i;

    if (howmuch < 0)
        return -1;

    so_far = 0;

    //因为找的是evbuffer链表中的空闲空间，所以从最后一个有数据的chain中开始找
    firstchainp = buf->last_with_datap;
    if (CHAIN_SPACE_LEN(*firstchainp) == 0) {//这个chain已经没有空间了
        firstchainp = &(*firstchainp)->next;//那么只能下一个chain了
    }

    //因为Libevent在调用本函数之前，一般会调用_evbuffer_expand_fast来扩大
    //evbuffer的可用空间。所以下面的循环中并没有判断chain是否为NULL，就直接
    //chain->next
    chain = *firstchainp;
    for (i = 0; i < n_vecs_avail && so_far < (size_t)howmuch; ++i) {
        size_t avail = (size_t) CHAIN_SPACE_LEN(chain);
        //如果exact为真，那么即使这个chain有更多的可用空间，也不会使用。只会
        //要自己正需要的空间
        if (avail > (howmuch - so_far) && exact)
            avail = howmuch - so_far;
        vecs[i].iov_base = CHAIN_SPACE_PTR(chain);//这个chain的可用空间的开始
位置
        vecs[i].iov_len = avail;//可用长度
        so_far += avail;
        chain = chain->next;
    }

    *chainp = firstchainp; //指向第一个有可用空间的chain
    return i;//返回需要多少个chain才能有howmuch这么多的空闲空间
}
```

在Windows系统，虽然没有readv函数，但它有WSARecv函数，可以把数据读取到一个类似iovec的结构体中，所以在Windows系统中，Libevent还是选择在n个evbuffer_chain中找到足够的空闲空间。所以在Libevent中有下面的宏定义：

```
//buffer.c文件
    //sys/uio.h文件定义了readv函数
#if defined(_EVENT_HAVE_SYS_UIO_H) || defined(WIN32)
#define USE_IOVEC_IMPL //该宏标志所在的系统支持类似readv的函数
#endif

//Windows系统定义了下面结构体
    typedef struct __WSABUF {
        u_long    len;
        char FAR *buf;
} WSABUF, *LPWSABUF;
```

如果所在的系统不支持类似readv这样的函数，那么Libevent就只能在一个evbuffer_chain申请一个足够大的空间，然后直接调用read函数了。

前面说到的扩容，分别是由函数_evbuffer_expand_fast和函数evbuffer_expand_singlechain完成的。在《evbuffer结构与基本操作》一文中已经有对这两个函数的介绍，这里就不多说了。

由于存在是否支持类似readv函数 这两种情况，所以evbuffer_read在实现上也出现了两种实现。

上面说了这么多，还是来看一下evbuffer_read的具体实现吧。

```
//buffer.c文件
//返回读取到的字节数。错误返回-1，断开了连接返回0
int //howmuch指出此时evbuffer可以使用的空间大小
evbuffer_read(struct evbuffer *buf, evutil_socket_t fd, int howmuch)
{
    struct evbuffer_chain **chainp;
    int n;
    int result;

#ifdef USE_IOVEC_IMPL //所在的系统支持iovec或者是Windows操作系统
    int nvecs, i, remaining;
#else
    struct evbuffer_chain *chain;
    unsigned char *p;
#endif

    EVBUFFER_LOCK(buf);//加锁

    //冻结缓冲区尾部，禁止在尾部追加数据
    if (buf->freeze_end) {
        result = -1;
        goto done;
    }

    //获取这个socket接收缓冲区里面有多少字节可读.通过ioctl实现
    n = get_n_bytes_readable_on_socket(fd);
    if (n <= 0 || n > EVBUFFER_MAX_READ)//每次只读EVBUFFER_MAX_READ(4096)个
字符
        n = EVBUFFER_MAX_READ;
    if (howmuch < 0 || howmuch > n)
        howmuch = n;

#ifdef USE_IOVEC_IMPL //所在的系统支持iovec或者是Windows操作系统
     //NUM_READ_IOVEC等于4
    //扩大evbuffer,使得其有howmuch字节的空闲空间
    //在NUM_READ_IOVEC个evbuffer_chain中扩容足够的空闲空间
    if (_evbuffer_expand_fast(buf, howmuch, NUM_READ_IOVEC) == -1) {
        result = -1;
        goto done;
    } else {
        //在posix中IOV_TYPE为iovec，在Windows中为WSABUF
        IOV_TYPE vecs[NUM_READ_IOVEC];

#ifdef _EVBUFFER_IOVEC_IS_NATIVE //所在的系统支持iovec结构体
        nvecs = _evbuffer_read_setup_vecs(buf, howmuch, vecs,
            NUM_READ_IOVEC, &chainp, 1);
#else //Windows系统。因为没有native的 iovec

        //在Windows系统中，evbuffer_iovec定义得和posix中的iovec一样.
        //因为_evbuffer_read_setup_vecs函数只接受像iovec那样结构体的参数
        struct evbuffer_iovec ev_vecs[NUM_READ_IOVEC];
```

```
        nvecs = _evbuffer_read_setup_vecs(buf, howmuch, ev_vecs, 2,
            &chainp, 1);

        //因为在Windows中，需要使用WSABUF结构体读取数据，所以要从evbuffer_iovec
        //中将一些值提出出来，放到vecs中
        for (i=0; i < nvecs; ++i)
            WSABUF_FROM_EVBUFFER_IOV(&vecs[i], &ev_vecs[i]);
#endif

//完成把数据从socket fd中读取出来
#ifdef WIN32
        {
            DWORD bytesRead;
            DWORD flags=0;
            //虽然Windows支持类似readv的函数，但Windows没有readv函数，只有下面的
函数
            if (WSARecv(fd, vecs, nvecs, &bytesRead, &flags, NULL, NULL))
 {
                if (WSAGetLastError() == WSAECONNABORTED)
                    n = 0;
                else
                    n = -1;
            } else
                n = bytesRead;
        }
#else
        n = readv(fd, vecs, nvecs);//POSIX
#endif
    }

//如果所在的系统不支持 iovec并且不是Windows系统。也就是说不支持类似
//readv这样的函数。那么只能把所有的数据都读到一个chain中
#else /*!USE_IOVEC_IMPL*/

    //把一个chain扩大得可以有howmuch字节的空闲空间
    if ((chain = evbuffer_expand_singlechain(buf, howmuch)) == NULL) {
        result = -1;
        goto done;
    }

    p = chain->buffer + chain->misalign + chain->off;

//读取数据
#ifndef WIN32
    n = read(fd, p, howmuch);
#else
    n = recv(fd, p, howmuch, 0);
#endif

#endif /* USE_IOVEC_IMPL *///终止前面的if宏
```

```
    if (n == -1) {//错误
        result = -1;
        goto done;
    }
    if (n == 0) {//断开了连接
        result = 0;
        goto done;
    }


#ifdef USE_IOVEC_IMPL//使用了iovec结构体读取数据。需要做一些额外的处理

    //chainp是由_evbuffer_read_setup_vecs函数调用得到。它指向未从fd读取数据时
    //第一个有空闲空间位置的chain

    remaining = n;//n等于读取到的字节数
    //使用iovec读取数据时，只是把数据往chain中填充，并没有修改evbuffer_chain
    //的成员，比如off偏移量成员。此时就需要把这个off修改到正确值
    for (i=0; i < nvecs; ++i) {
        //CHAIN_SPACE_LEN(*chainp)返回的是填充数据前的空闲空间。
        //除了最后那个chain外，其他的chain都会被填满的。所以对于非last
        //chain，直接把off加上这个space即可。
        ev_ssize_t space = (ev_ssize_t) CHAIN_SPACE_LEN(*chainp);
        if (space < remaining) {//前面的chain
            (*chainp)->off += space;
            remaining -= (int)space;
        } else {//最后那个chain
            (*chainp)->off += remaining;
            buf->last_with_datap = chainp;//指向最后一个有数据的chain
            break;
        }
        chainp = &(*chainp)->next;
    }
#else
    chain->off += n;
    //调整last_with_datap，使得*last_with_datap指向最后一个有数据的chain
    advance_last_with_data(buf);
#endif

    buf->total_len += n;
    buf->n_add_for_cb += n;//添加了n字节

    /* Tell someone about changes in this buffer */
    evbuffer_invoke_callbacks(buf);//evbuffer添加了数据，就需要调用回调函数
    result = n;
done:
    EVBUFFER_UNLOCK(buf);
    return result;
}
```

往socket写入数据

因为evbuffer是用链表的形式存放数据，所以要把这些链表上的数据写入socket，那么使用writev这个函数是十分有效的。同前面一样，使用iovec结构体数组，就需要设置数组元素的指针。这个工作由evbuffer_write_iovec函数完成。

正如前面的从socket读出数据，可能所在的系统并不支持writev这样的函数。此时就只能使用一般的write函数了，但这个函数要求数据放在一个连续的空间。所以Libevent有一个函数evbuffer_pullup，用来把链表内存拉直，即把一定数量的数据从链表中copy到一个连续的内存空间。这个连续的空间也是由某个evbuffer_chain的buffer指针指向，并且这个evbuffer_chain会被插入到链表中。这个时候就可以直接使用write或者send函数发送这特定数量的数据了。

不同于读，写操作还有第三种可能。那就是sendfile。如果所在的系统支持sendfile，并且用户是通过evbuffer_add_file添加数据的，那么此时Libevent就是所在系统的sendfile函数发送数据。

Libevent内部一般通过evbuffer_write函数把数据写入到socket fd中。下面是具体的实现。

```
//buffer.c文件
int
evbuffer_write(struct evbuffer *buffer, evutil_socket_t fd)
{
    //把evbuffer的所有数据都写入到fd中
    return evbuffer_write_atmost(buffer, fd, -1);
}


int//howmuch是要写的字节数。如果小于0，那么就把buffer里的所有数据都写入fd
evbuffer_write_atmost(struct evbuffer *buffer, evutil_socket_t fd,
    ev_ssize_t howmuch)
{
    int n = -1;

    EVBUFFER_LOCK(buffer);

    //冻结了链表头，无法往fd写数据。因为写之后，还要把数据从evbuffer中删除
    if (buffer->freeze_start) {
        goto done;
    }

    if (howmuch < 0 || (size_t)howmuch > buffer->total_len)
        howmuch = buffer->total_len;

    if (howmuch > 0) {
#ifdef USE_SENDFILE //所在的系统支持sendfile
        struct evbuffer_chain *chain = buffer->first;
        //需通过evbuffer_add_file添加数据，才会使用sendfile
        if (chain != NULL && (chain->flags & EVBUFFER_SENDFILE)) //并且要求
使用sendfile
            n = evbuffer_write_sendfile(buffer, fd, howmuch);
        else {
#endif
#ifdef USE_IOVEC_IMPL //所在的系统支持writev这类函数
        //函数内部会设置数组元素的成员指针，以及长度成员
        n = evbuffer_write_iovec(buffer, fd, howmuch);
#elif defined(WIN32)
        /* XXX(nickm) Don't disable this code until we know if
         * the WSARecv code above works. */
        //把evbuffer前面的howmuch字节拉直。使得这howmuch字节都放在一个chain里面
        //也就是放在一个连续的空间，不再是之前的多个链表节点。这样就能直接用
        //send函数发送了。
        void *p = evbuffer_pullup(buffer, howmuch);
        n = send(fd, p, howmuch, 0);
#else
        void *p = evbuffer_pullup(buffer, howmuch);
        n = write(fd, p, howmuch);
#endif
#ifdef USE_SENDFILE
        }
```

```
#endif
    }

    if (n > 0)
        evbuffer_drain(buffer, n);//从链表中删除已经写入到socket的n个字节

done:
    EVBUFFER_UNLOCK(buffer);
    return (n);
}
```

参考：http://www.wangafu.net/~nickm/libevent-book/Ref7_evbuffer.html

24.bufferevent工作流程探究

和之前的《Libevent工作流程探究》一样，这里也是用一个例子来探究bufferevent的工作流程。具体的例子可以参考《Libevent使用例子，从简单到复杂》，这里就不列出了。其实要做的例子也就是bufferevent_socket_new、bufferevent_setcb、bufferevent_enable这几个函数。

因为本文会用到《Libevent工作流程探究》中提到的说法，比如将一个event插入到event_base中。所以读者最好先读一下那篇博文。此外，因为bufferevent结构体本身会使用evbuffer结构体和还会调用相应的一些操作，所以读者还应该先阅读《evbuffer结构与基本操作》和《更多evbuffer操作函数》。

bufferevent结构体

bufferevent其实也就是在event_base的基础上再进行一层封装，其本质还是离不开event和event_base，从bufferevent的结构体就可以看到这一点。

bufferevent结构体中有两个event，分别用来监听同一个fd的可读事件和可写事件。为什么不用一个event同时监听可读和可写呢？这是因为监听可写是困难的，下面会说到原因。读者也可以自问一下，自己之前有没有试过用最原始的event监听一个fd的可写。

由于socket 是全双工的，所以在bufferevent结构体中，也有两个evbuffer成员，分别是读缓冲区和写缓冲区。 bufferevent结构体定义如下：

```
//bufferevent_struct.h文件
struct bufferevent {
    struct event_base *ev_base;

    //操作结构体，成员有一些函数指针。类似struct eventop结构体
    const struct bufferevent_ops *be_ops;

    struct event ev_read;//读事件event
    struct event ev_write;//写事件event

    struct evbuffer *input;//读缓冲区

    struct evbuffer *output; //写缓冲区

    struct event_watermark wm_read;//读水位
    struct event_watermark wm_write;//写水位


    bufferevent_data_cb readcb;//可读时的回调函数指针
    bufferevent_data_cb writecb;//可写时的回调函数指针
    bufferevent_event_cb errorcb;//错误发生时的回调函数指针
    void *cbarg;//回调函数的参数

    struct timeval timeout_read;//读事件event的超时值
    struct timeval timeout_write;//写事件event的超时值

    /** Events that are currently enabled: currently EV_READ and EV_WRITE

        are supported. */
    short enabled;
};
```

如果看过Libevent的参考手册的话，应该还会知道bufferevent除了用于socket外，还可以用于socketpair 和 filter。如果用面向对象的思维，应从这个三个应用中抽出相同的部分作为父类，然后派生出三个子类。

Libevent虽然是用C语言写的，不过它还是提取出一些公共部分，然后定义一个bufferevent_private结构体，用于保存这些公共部分成员。从集合的角度来说，bufferevent_private应该是bufferevent的一个子集，即一部分。但在Libevent中，bufferevent确实bufferevent_private的一个成员。下面是bufferevent_private结构体。

```
//bufferevent-internal.h文件
struct bufferevent_private {
    struct bufferevent bev;

    //设置input evbuffer的高水位时，需要一个evbuffer回调函数配合工作
    struct evbuffer_cb_entry *read_watermarks_cb;

    /** If set, we should free the lock when we free the bufferevent. */

    //锁是Libevent自动分配的，还是用户分配的
    unsigned own_lock : 1;

    ...

    //这个socket是否处理正在连接服务器状态
    unsigned connecting : 1;
    //标志连接被拒绝
    unsigned connection_refused : 1;

    //标志是什么原因把 读 挂起来
    bufferevent_suspend_flags read_suspended;
        //标志是什么原因把 写 挂起来
    bufferevent_suspend_flags write_suspended;

    enum bufferevent_options options;
        int refcnt;// bufferevent的引用计数

    //锁变量
    void *lock;
};
```

新建一个bufferevent

函数bufferevent_socket_new可以完成这个工作。

```
//bufferevent-internal.h文件
struct bufferevent_ops {
    const char *type;//类型名称

    off_t mem_offset;//成员bev的偏移量

    //启动。将event加入到event_base中
    int (*enable)(struct bufferevent *, short);

    //关闭。将event从event_base中删除
    int (*disable)(struct bufferevent *, short);
    //销毁
    void (*destruct)(struct bufferevent *);
    //调整event的超时值
    int (*adj_timeouts)(struct bufferevent *);
    /** Called to flush data. */
    int (*flush)(struct bufferevent *, short, enum bufferevent_flush_mod
e);
    //获取成员的值。具体看实现
    int (*ctrl)(struct bufferevent *, enum bufferevent_ctrl_op, union buf
ferevent_ctrl_data *);
};


//bufferevent_sock.c文件
const struct bufferevent_ops bufferevent_ops_socket = {
    "socket",
    evutil_offsetof(struct bufferevent_private, bev),
    be_socket_enable,
    be_socket_disable,
    be_socket_destruct,
    be_socket_adj_timeouts,
    be_socket_flush,
    be_socket_ctrl,
};

//由于有几个不同类型的bufferevent，而且它们的enable、disable等操作是不同的。所以
//需要的一些函数指针指明某个类型的bufferevent应该使用哪些操作函数。结构体buffereven
t_ops_socket
//就应运而生。对于socket，其操作函数如上。

//bufferevent_sock.c文件
struct bufferevent *
bufferevent_socket_new(struct event_base *base, evutil_socket_t fd,
    int options)
{
    struct bufferevent_private *bufev_p;
    struct bufferevent *bufev;

    ...//win32
```

```
    //结构体内存清零，所有成员都为0
    if ((bufev_p = mm_calloc(1, sizeof(struct bufferevent_private)))== NU
LL)
        return NULL;

    //如果options中需要线程安全，那么就会申请锁
    //会新建一个输入和输出缓存区
    if (bufferevent_init_common(bufev_p, base, &bufferevent_ops_socket,
                    options) < 0) {
        mm_free(bufev_p);
        return NULL;
    }
    bufev = &bufev_p->bev;
    //设置将evbuffer的数据向fd传
    evbuffer_set_flags(bufev->output, EVBUFFER_FLAG_DRAINS_TO_FD);

    //将fd与event相关联。同一个fd关联两个event
    event_assign(&bufev->ev_read, bufev->ev_base, fd,
        EV_READ|EV_PERSIST, bufferevent_readcb, bufev);
    event_assign(&bufev->ev_write, bufev->ev_base, fd,
        EV_WRITE|EV_PERSIST, bufferevent_writecb, bufev);

    //设置evbuffer的回调函数，使得外界给写缓冲区添加数据时，能触发
    //写操作，这个回调对于写事件的监听是很重要的
    evbuffer_add_cb(bufev->output, bufferevent_socket_outbuf_cb, bufev);

    //冻结读缓冲区的尾部，未解冻之前不能往读缓冲区追加数据
    //也就是说不能从socket fd中读取数据
    evbuffer_freeze(bufev->input, 0);

    //冻结写缓冲区的头部，未解冻之前不能把写缓冲区的头部数据删除
    //也就是说不能把数据写到socket fd
    evbuffer_freeze(bufev->output, 1);

    return bufev;
}
```

留意函数里面的evbuffer_add_cb调用，后面会说到。

函数在最后面会冻结两个缓冲区。其实，虽然这里冻结了，但实际上Libevent在读数据或者写数据之前会解冻的读完或者写完数据后，又会马上冻结。这主要防止数据被意外修改。用户一般不会直接调用evbuffer_freeze或者evbuffer_unfreeze函数。一切的冻结和解冻操作都由Libevent内部完成。还有一点要注意，因为这里只是把写缓冲区的头部冻结了。所以还是可以往写缓冲区的尾部追加数据。同样，此时也是可以从读缓冲区读取数据。这个是必须的。因为在Libevent内部不解冻的时候，用户需要从读缓冲区中获取数据(这相当于从socket fd中读取数据)，用户也需要把数据写到写缓冲区中(这相当于把数据写入到socket fd中)。

在bufferevent_socket_new函数里面会调用函数bufferevent_init_common完成公有部分的初始化。

```
//bufferevent.c文件
int
bufferevent_init_common(struct bufferevent_private *bufev_private,
    struct event_base *base,
    const struct bufferevent_ops *ops,
    enum bufferevent_options options)
{
    struct bufferevent *bufev = &bufev_private->bev;

    //分配输入缓冲区
    if (!bufev->input) {
        if ((bufev->input = evbuffer_new()) == NULL)
            return -1;
    }

    //分配输出缓冲区
    if (!bufev->output) {
        if ((bufev->output = evbuffer_new()) == NULL) {
            evbuffer_free(bufev->input);
            return -1;
        }
    }

    bufev_private->refcnt = 1;//引用次数为1
    bufev->ev_base = base;

    /* Disable timeouts. */
    //默认情况下,读和写event都是不支持超时的
    evutil_timerclear(&bufev->timeout_read);
    evutil_timerclear(&bufev->timeout_write);

    bufev->be_ops = ops;

    /*
     * Set to EV_WRITE so that using bufferevent_write is going to
     * trigger a callback.  Reading needs to be explicitly enabled
     * because otherwise no data will be available.
     */
     //可写是默认支持的
    bufev->enabled = EV_WRITE;

#ifndef _EVENT_DISABLE_THREAD_SUPPORT
    if (options & BEV_OPT_THREADSAFE) {
        //申请锁。
        if (bufferevent_enable_locking(bufev, NULL) < 0) {
            /* cleanup */
            evbuffer_free(bufev->input);
            evbuffer_free(bufev->output);
            bufev->input = NULL;
            bufev->output = NULL;
            return -1;
```

```
        }
    }
#endif
    ...//延迟调用的初始化，一般不需要用到

    bufev_private->options = options;

    //将evbuffer和bufferevent相关联
    evbuffer_set_parent(bufev->input, bufev);
    evbuffer_set_parent(bufev->output, bufev);

    return 0;
}
```

代码中可以看到，默认是enable EV_WRITE的。

设置回调函数

函数bufferevent_setcb完成这个工作。该函数相当简单，也就是进行一些赋值操作。

```
//bufferevent.c文件
void
bufferevent_setcb(struct bufferevent *bufev,
    bufferevent_data_cb readcb, bufferevent_data_cb writecb,
    bufferevent_event_cb eventcb, void *cbarg)
{
    //bufferevent结构体内部有一个锁变量
    BEV_LOCK(bufev);

    bufev->readcb = readcb;
    bufev->writecb = writecb;
    bufev->errorcb = eventcb;

    bufev->cbarg = cbarg;
    BEV_UNLOCK(bufev);
}
```

如果不想设置某个操作的回调函数，直接设置为NULL即可。

令bufferevent可以工作

相信读者也知道，即使调用了bufferevent_socket_new和bufferevent_setcb，这个bufferevent还是不能工作，必须调用bufferevent_enable。为什么会这样的呢？

如果看过之前的那些博文，相信读者知道，一个event能够工作，不仅仅需要new出来，还要调用event_add函数，把这个event添加到event_base中。在本文前面的代码中，并没有看到event_add函数的调用。所以还需要调用一个函数，把event添加到event_base中。函数bufferevent_enable就是完成这个工作的。

```
//bufferevent.c文件
int
bufferevent_enable(struct bufferevent *bufev, short event)
{
    struct bufferevent_private *bufev_private =
        EVUTIL_UPCAST(bufev, struct bufferevent_private, bev);
    short impl_events = event;
    int r = 0;

    //增加引用并加锁
    //增加引用是为了防止其他线程调用bufferevent_free，释放了bufferevent
    _bufferevent_incref_and_lock(bufev);

    //挂起了读，此时不能监听读事件
    if (bufev_private->read_suspended)
        impl_events &= ~EV_READ;

    //挂起了写，此时不能监听写事情
    if (bufev_private->write_suspended)
        impl_events &= ~EV_WRITE;

    bufev->enabled |= event;

    //调用对应类型的enbale函数。因为不同类型的bufferevent有不同的enable函数
    if (impl_events && bufev->be_ops->enable(bufev, impl_events) < 0)
        r = -1;

    //减少引用并解锁
    _bufferevent_decref_and_unlock(bufev);
    return r;
}
```

上面代码可以看到，最终会调用对应bufferevent类型的enable函数，对于socket bufferevent，其enable函数是be_socket_enable，代码如下：

```
//bufferevent.c文件
int
_bufferevent_add_event(struct event *ev, const struct timeval *tv)
{
    if (tv->tv_sec == 0 && tv->tv_usec == 0)
        return event_add(ev, NULL);
    else
        return event_add(ev, tv);
}


//bufferevent_sock.c文件
#define be_socket_add(ev, t)            \
    _bufferevent_add_event((ev), (t))


static int
be_socket_enable(struct bufferevent *bufev, short event)
{
    if (event & EV_READ) {
        if (be_socket_add(&bufev->ev_read,&bufev->timeout_read) == -1)
            return -1;
    }
    if (event & EV_WRITE) {
        if (be_socket_add(&bufev->ev_write,&bufev->timeout_write) == -1)

            return -1;
    }
    return 0;
}
```

如果读者熟悉Libevent的超时事件，那么可以知道Libevent是在event_add函数里面确定一个event的超时的。上面代码也展示了这一点，如果读或者写event设置了超时(即其超时值不为0)，那么就会作为参数传给event_add函数。如果读者不熟悉的Libevent的超时事件的话，可以参考上文中的《超时event的处理》小节。

用户可以调用函数bufferevent_set_timeouts，设置读或者写事件的超时。代码如下：

```c
//bufferevent.c文件
int
bufferevent_set_timeouts(struct bufferevent *bufev,
              const struct timeval *tv_read,
              const struct timeval *tv_write)
{
    int r = 0;
    BEV_LOCK(bufev);
    if (tv_read) {
        bufev->timeout_read = *tv_read;
    } else {
        evutil_timerclear(&bufev->timeout_read);
    }
    if (tv_write) {
        bufev->timeout_write = *tv_write;
    } else {
        evutil_timerclear(&bufev->timeout_write);
    }

    if (bufev->be_ops->adj_timeouts)
        r = bufev->be_ops->adj_timeouts(bufev);
    BEV_UNLOCK(bufev);

    return r;
}


//bufferevent_sock.c文件
static int
be_socket_adj_timeouts(struct bufferevent *bufev)
{
    int r = 0;
    //用户监听了读事件
    if (event_pending(&bufev->ev_read, EV_READ, NULL))
        if (be_socket_add(&bufev->ev_read, &bufev->timeout_read) < 0)
            r = -1;

    //用户监听了写事件
    if (event_pending(&bufev->ev_write, EV_WRITE, NULL)) {
        if (be_socket_add(&bufev->ev_write, &bufev->timeout_write) < 0)
            r = -1;
    }
    return r;
}
```

从上面代码可以看到：用户不仅仅可以设置超时值，还可以修改超时值，也是通过这个函数进行修的。当然也是可以删除超时的，直接把超时参数设置成NULL即可。

至此，已经完成了bufferevent的初始化工作，只需调用event_base_dispatch函数，启动发动机就可以工作了。

处理读事件

接下来的任务：底层的socket fd接收数据后，bufferevent是怎么工作的。

读事件的水位

在讲解读事件之前，先来看一下水位问题，函数bufferevent_setwatermark可以设置读和写的水位。这里只讲解读事件的水位。

水位有两个：低水位和高水位。

低水位比较容易懂，就是当可读的数据量到达这个低水位后，才会调用用户设置的回调函数。比如用户想每次读取100字节，那么就可以把低水位设置为100。当可读数据的字节数小于100时，即使有数据都不会打扰用户(即不会调用用户设置的回调函数)。可读数据大于等于100字节后，才会调用用户的回调函数。

高水位是什么呢？其实，这和用户的回调函数没有关系。它的意义是：把读事件的evbuffer的数据量限制在高水位之下。比如，用户认为读缓冲区不能太大(太大的话，链表会很长)。那么用户就会设置读事件的高水位。当读缓冲区的数据量达到这个高水位后，即使socket fd还有数据没有读，也不会读进这个读缓冲区里面。一句话说，就是控制evbuffer的大小。

虽然控制了evbuffer的大小，但socket fd可能还有数据。有数据就会触发可读事件，但处理可读的时候，又会发现设置了高水位，不能读取数据evbuffer。socket fd的数据没有被读完，又触发……。这个貌似是一个死循环。实际上是不会出现这个死循环的，因为Libevent发现evbuffer的数据量到达高水位后，就会把可读事件给挂起来，让它不能再触发了。Libevent使用函数bufferevent_wm_suspend_read把监听读事件的event挂起来。下面看一下Libevent是怎么把一个event挂起来的。

```
//bufferevent-internal.h文件
#define bufferevent_wm_suspend_read(b) \
    bufferevent_suspend_read((b), BEV_SUSPEND_WM)


//bufferevent.c文件
void
bufferevent_suspend_read(struct bufferevent *bufev, bufferevent_suspend_f
lags what)
{
    struct bufferevent_private *bufev_private =
        EVUTIL_UPCAST(bufev, struct bufferevent_private, bev);
    BEV_LOCK(bufev);
    if (!bufev_private->read_suspended)//不能挂多次
        bufev->be_ops->disable(bufev, EV_READ);//实际调用be_socket_disable
函数
    bufev_private->read_suspended |= what;//因何而被挂起
    BEV_UNLOCK(bufev);
}


//bufferevent_sock.c文件
static int
be_socket_disable(struct bufferevent *bufev, short event)
{
    struct bufferevent_private *bufev_p =
        EVUTIL_UPCAST(bufev, struct bufferevent_private, bev);
    if (event & EV_READ) {
        if (event_del(&bufev->ev_read) == -1)
            return -1;
    }
    /* Don't actually disable the write if we are trying to connect. */
    if ((event & EV_WRITE) && ! bufev_p->connecting) {
        if (event_del(&bufev->ev_write) == -1)//删掉这个event
            return -1;
    }
    return 0;
}
```

居然是直接删除这个监听读事件的event，真的是挂了!!!

看来不能随便设置高水位，因为它会暂停读。如果只想设置低水位而不想设置高水位，那么在调用bufferevent_setwatermark函数时，高水位的参数设为0即可。

那么什么时候取消挂起，让bufferevent可以继续读socket 数据呢？从高水位的意义来说，当然是当evbuffer里面的数据量小于高水位时，就能再次读取socket数据了。现在来看一下Libevent是怎么恢复读的。看一下设置水位的函数bufferevent_setwatermark吧，它进行了一些为高水位埋下了一个回调函数。对，就是evbuffer的回调函数。前一篇博文说到，当evbuffer里面的数据添加或者删除时，是会触发一些回调函数的。当用户移除evbuffer的一些数据量时，Libevent就会检查这个evbuffer的数据量是否小于高水位，如果小于的话，那么就恢复读事件。

不说这么多了，上代码。

```c
//bufferevent.c文件
void
bufferevent_setwatermark(struct bufferevent *bufev, short events,
    size_t lowmark, size_t highmark)
{
    struct bufferevent_private *bufev_private =
        EVUTIL_UPCAST(bufev, struct bufferevent_private, bev);

    BEV_LOCK(bufev);

    if (events & EV_READ) {
        bufev->wm_read.low = lowmark;
        bufev->wm_read.high = highmark;

        if (highmark) {//高水位
            /* There is now a new high-water mark for read.
               enable the callback if needed, and see if we should
               suspend/bufferevent_wm_unsuspend. */

            //还没设置高水位的回调函数
            if (bufev_private->read_watermarks_cb == NULL) {
                bufev_private->read_watermarks_cb =
                    evbuffer_add_cb(bufev->input,
                            bufferevent_inbuf_wm_cb,
                            bufev);//添加回调函数
            }
            evbuffer_cb_set_flags(bufev->input,
                      bufev_private->read_watermarks_cb,
                      EVBUFFER_CB_ENABLED|EVBUFFER_CB_NODEFER);

            //设置(修改)高水位时，evbuffer的数据量已经超过了水位值
            //可能是把之前的高水位调高或者调低
            //挂起操作和取消挂起操作都是幂等的(即多次挂起的作用等同于挂起一次)
            if (evbuffer_get_length(bufev->input) > highmark)
                bufferevent_wm_suspend_read(bufev);
            else if (evbuffer_get_length(bufev->input) < highmark)//调低了
                bufferevent_wm_unsuspend_read(bufev);
        } else {
            //高水位值等于0，那么就要取消挂起 读事件
            //取消挂起操作是幂等的
            /* There is now no high-water mark for read. */
            if (bufev_private->read_watermarks_cb)
                evbuffer_cb_clear_flags(bufev->input,
                    bufev_private->read_watermarks_cb,
                    EVBUFFER_CB_ENABLED);
            bufferevent_wm_unsuspend_read(bufev);
        }
    }
    BEV_UNLOCK(bufev);
}
```

这个函数，不仅仅为高水位设置回调函数，还会检查当前evbuffer的数据量是否超过了高水位。因为这个设置水位函数可能是在bufferevent工作一段时间后才添加的，所以evbuffer是有可能已经有数据的了，因此需要检查。如果超过了水位值，那么就需要挂起读。当然也存在另外一种可能：用户之前设置过了一个比较大的高水位，挂起了读。现在发现错了，就把高水位调低一点，此时就需要恢复读。

现在假设用户移除了一些evbuffer的数据，进而触发了evbuffer的回调函数，当然也就调用了函数bufferevent_inbuf_wm_cb。下面看一下这个函数是怎么恢复读的。

```
//bufferevent.c文件
static void
bufferevent_inbuf_wm_cb(struct evbuffer *buf,
    const struct evbuffer_cb_info *cbinfo,
    void *arg)
{
    struct bufferevent *bufev = arg;
    size_t size;

    size = evbuffer_get_length(buf);

    if (size >= bufev->wm_read.high)
        bufferevent_wm_suspend_read(bufev);
    else
        bufferevent_wm_unsuspend_read(bufev);
}

//bufferevent-internal.h文件
#define bufferevent_wm_unsuspend_read(b) \
    bufferevent_unsuspend_read((b), BEV_SUSPEND_WM)

//bufferevent.c文件
void
bufferevent_unsuspend_read(struct bufferevent *bufev, bufferevent_suspend
_flags what)
{
    struct bufferevent_private *bufev_private =
        EVUTIL_UPCAST(bufev, struct bufferevent_private, bev);

    BEV_LOCK(bufev);
    bufev_private->read_suspended &= ~what;
    if (!bufev_private->read_suspended && (bufev->enabled & EV_READ))
        bufev->be_ops->enable(bufev, EV_READ);//重新把event插入到event_base
中
    BEV_UNLOCK(bufev);
}
```

因为用户可以手动为这个evbuffer添加数据，此时也会调用bufferevent_inbuf_wm_cb函数。此时就要检查evbuffer的数据量是否已经超过高水位了，而不能仅仅检查是否低于高水位。

高水位导致读的挂起和之后读的恢复，一切工作都是由Libevent内部完成的，用户不用做任何工作。

从socket中读取数据

从前面的一系列博文可以知道，如果一个socket可读了，那么监听可读事件的event的回调函数就会被调用。这个回调函数是在bufferevent_socket_new函数中被Libevent内部设置的，设置为bufferevent_readcb函数，用户并不知情。

当socket有数据可读时，Libevent就会监听到，然后调用bufferevent_readcb函数处理。该函数会调用evbuffer_read函数，把数据从socket fd中读取到evbuffer中。然后再调用用户在bufferevent_setcb函数中设置的读事件回调函数。所以，当用户的读事件回调函数被调用时，数据已经在evbuffer中了，用户拿来就用，无需调用read这类会阻塞的函数。

下面看一下bufferevent_readcb函数的具体实现。

```c
static void
bufferevent_readcb(evutil_socket_t fd, short event, void *arg)
{
    struct bufferevent *bufev = arg;
    struct bufferevent_private *bufev_p =
        EVUTIL_UPCAST(bufev, struct bufferevent_private, bev);
    struct evbuffer *input;
    int res = 0;
    short what = BEV_EVENT_READING;
    ev_ssize_t howmuch = -1, readmax=-1;

    _bufferevent_incref_and_lock(bufev);

    if (event == EV_TIMEOUT) {
        /* Note that we only check for event==EV_TIMEOUT. If
         * event==EV_TIMEOUT|EV_READ, we can safely ignore the
         * timeout, since a read has occurred */
        what |= BEV_EVENT_TIMEOUT;
        goto error;
    }

    input = bufev->input;


    //用户设置了高水位
    if (bufev->wm_read.high != 0) {
        howmuch = bufev->wm_read.high - evbuffer_get_length(input);
        /* we somehow lowered the watermark, stop reading */
        if (howmuch <= 0) {
            bufferevent_wm_suspend_read(bufev);
            goto done;
        }
    }

    //因为用户可以限速，所以这么要检测最大的可读大小。
    //如果没有限速的话，那么将返回16384字节，即16K
    //默认情况下是没有限速的。
    readmax = _bufferevent_get_read_max(bufev_p);
    if (howmuch < 0 || howmuch > readmax) /* The use of -1 for "unlimite
d"
                            * uglifies this code. XXXX */
        howmuch = readmax;

    //一些原因导致读 被挂起，比如加锁了。
    if (bufev_p->read_suspended)
        goto done;

    //解冻，使得可以在input的后面追加数据
    evbuffer_unfreeze(input, 0);
    res = evbuffer_read(input, fd, (int)howmuch); //从socket fd中读取数据
    evbuffer_freeze(input, 0);//冻结
```

```
    if (res == -1) {
        int err = evutil_socket_geterror(fd);
        if (EVUTIL_ERR_RW_RETRIABLE(err))//EINTER or EAGAIN
            goto reschedule;

        //不是 EINTER or EAGAIN 这两个可以重试的错误，那么就应该是其他致命的错误
        //此时，应该报告给用户
        what |= BEV_EVENT_ERROR;/**< unrecoverable error encountered */
    } else if (res == 0) {//断开了连接
        what |= BEV_EVENT_EOF;
    }

    if (res <= 0)
        goto error;

    //速率相关的操作
    _bufferevent_decrement_read_buckets(bufev_p, res);


    //evbuffer的数据量大于低水位值。
    if (evbuffer_get_length(input) >= bufev->wm_read.low)
        _bufferevent_run_readcb(bufev);//调用用户设置的回调函数

    goto done;

 reschedule:
    goto done;

 error:
    //把监听可读事件的event从event_base的事件队列中删除掉.event_del
    bufferevent_disable(bufev, EV_READ);//会调用be_socket_disable函数
    _bufferevent_run_eventcb(bufev, what);//会调用用户设置的错误处理函数

 done:
    _bufferevent_decref_and_unlock(bufev);
}
```

细心的读者可能会发现：**对用户的读事件回调函数的触发是边缘触发的。**这也就要求，在回调函数中，用户应该尽可能地把evbuffer的所有数据都读出来。如果想等到下一次回调时再读，那么需要等到下一次socketfd接收到数据才会触发用户的回调函数。如果之后socket fd一直收不到任何数据，那么即使evbuffer还有数据，用户的回调函数也不会被调用了。

处理写事件

对一个可读事件进行监听是比较容易的，但对于一个可写事件进行监听则比较困难。为什么呢？因为可读监听是监听fd的读缓冲区是否有数据了，如果没有数据那么就一直等待。对于可写，首先要明白“什么是可写”，可写就是fd的写缓冲区(这个缓冲区在内核)还没满，可以往里面放数据。这就有一个问题，如果写缓冲区没有满，那么就一直是可写状态。如果一个event监听了可写事件，那么这个event就会一直被触发（死循环）。因为一般情况下，如果不是发大量的数据这个写缓冲区是不会满的。

也就是说，不能监听可写事件。但我们确实要往fd中写数据，那怎么办？Libevent的做法是：当我们确实要写入数据时，才监听可写事件。也就是说我们调用bufferevent_write写入数据时，Libevent才会把监听可写事件的那个event注册到event_base中。当Libevent把数据都写入到fd的缓冲区后，Libevent又会把这个event从event_base中删除。比较烦琐。

bufferevent_writecb函数不仅仅要处理上面说到的那个问题，还要处理另外一个坑爹的问题。那就是：判断socket fd是不是已经连接上服务器了。这是因为这个socket fd是非阻塞的，所以它调用connect时，可能还没连接上就返回了。对于非阻塞socket fd，一般是通过判断这个socket是否可写，从而得知这个socket是否已经连接上服务器。如果可写，那么它就已经成功连接上服务器了。这个问题，这里先提一下，后面会详细讲。

同前面的监听可读一样，Libevent是在bufferevent_socket_new函数设置可写的回调函数，为bufferevent_writecb。

```
//bufferevent_sock.c文件
static void
bufferevent_writecb(evutil_socket_t fd, short event, void *arg)
{
    struct bufferevent *bufev = arg;
    struct bufferevent_private *bufev_p =
        EVUTIL_UPCAST(bufev, struct bufferevent_private, bev);
    int res = 0;
    short what = BEV_EVENT_WRITING;
    int connected = 0;
    ev_ssize_t atmost = -1;

    _bufferevent_incref_and_lock(bufev);

    if (event == EV_TIMEOUT) {
        /* Note that we only check for event==EV_TIMEOUT. If
         * event==EV_TIMEOUT|EV_WRITE, we can safely ignore the
         * timeout, since a read has occurred */
        what |= BEV_EVENT_TIMEOUT;
        goto error;
    }

    ...//判断这个socket是否已经连接上服务器了


    //用户可能设置了限速，如果没有限速，那么atmost将返回16384(16K)
    atmost = _bufferevent_get_write_max(bufev_p);

    //一些原因导致写被挂起来了
    if (bufev_p->write_suspended)
        goto done;

    //如果evbuffer有数据可以写到sockfd中
    if (evbuffer_get_length(bufev->output)) {
        //解冻链表头
        evbuffer_unfreeze(bufev->output, 1);
        //将output这个evbuffer的数据写到socket fd 的缓冲区中
        //会把已经写到socket fd缓冲区的数据，从evbuffer中删除
        res = evbuffer_write_atmost(bufev->output, fd, atmost);
        evbuffer_freeze(bufev->output, 1);

        if (res == -1) {
            int err = evutil_socket_geterror(fd);
            if (EVUTIL_ERR_RW_RETRIABLE(err))//可以恢复的错误。一般是EINTR或者
EAGAIN
                goto reschedule;
            what |= BEV_EVENT_ERROR;
        } else if (res == 0) {//该socket已经断开连接了
            what |= BEV_EVENT_EOF;
        }
        if (res <= 0)
```

```
                goto error;
        }

        //如果把写缓冲区的数据都写完成了。为了防止event_base不断地触发可写
        //事件，此时要把这个监听可写的event删除。
        //前面的atmost限制了一次最大的可写数据。如果还没写所有的数据
        //那么就不能delete这个event，而是要继续监听可写事情，知道把所有的
        //数据都写到socket fd中。
        if (evbuffer_get_length(bufev->output) == 0) {
            event_del(&bufev->ev_write);
        }


        //如果evbuffer里面的数据量已经写得七七八八了，小于设置的低水位值，那么
        //就会调用用户设置的写事件回调函数
        if ((res || !connected) &&
            evbuffer_get_length(bufev->output) <= bufev->wm_write.low) {
            _bufferevent_run_writecb(bufev);
        }

        goto done;

    reschedule:
        if (evbuffer_get_length(bufev->output) == 0) {
            event_del(&bufev->ev_write);
        }
        goto done;

    error:
        bufferevent_disable(bufev, EV_WRITE);//有错误。把这个写event删除
        _bufferevent_run_eventcb(bufev, what);

    done:
        _bufferevent_decref_and_unlock(bufev);
    }
```

上面代码的逻辑比较清晰，调用evbuffer_write_atmost函数把数据从evbuffer中写到evbuffer缓冲区中，此时要注意函数的返回值，因为可能写的时候发生错误。如果发生了错误，就要调用用户设置的event回调函数(网上也有人称其为错误处理函数)。

之后，还要判断evbuffer的数据是否已经全部写到socket 的缓冲区了。如果已经全部写了，那么就要把监听写事件的event从event_base的插入队列中删除。如果还没写完，那么就不能删除，因为还要继续监听可写事件，下次接着写。

现在来看一下，把监听写事件的event从event_base的插入队列中删除后，如果下次用户有数据要写的时候，怎么把这个event添加到event_base的插入队列。

用户一般是通过bufferevent_write函数把数据写入到evbuffer(写入evbuffer后，接着就会被写入socket，所以调用bufferevent_write就相当于把数据写入到socket。)。而这个bufferevent_write函数是直接调用evbuffer_add函数的。函数evbuffer_add没有调用什么可疑的函数，能够把监听可写的event添加到event_base中。唯一的可能就是那个回调函数。对就是evbuffer的回调函数。关于evbuffer的回调函数，可以参考这里。

```
//bufferevent.c文件
int
bufferevent_write(struct bufferevent *bufev, const void *data, size_t siz
e)
{
    if (evbuffer_add(bufev->output, data, size) == -1)
        return (-1);

    return 0;
}


//buffer.c文件
int
evbuffer_add(struct evbuffer *buf, const void *data_in, size_t datlen)
{
    ...

out:
    evbuffer_invoke_callbacks(buf);//调用回调函数
    result = 0;
done:
    return result;
}
```

还记得本文前面的bufferevent_socket_new函数吗？该函数里面会有

```
evbuffer_add_cb(bufev->output,bufferevent_socket_outbuf_cb, bufev);
```

当bufferevent的写缓冲区output的数据发生变化时，函数bufferevent_socket_outbuf_cb就会被调用。现在马上飞到这个函数。

```
//bufferevent_sock.c文件
static void
bufferevent_socket_outbuf_cb(struct evbuffer *buf,
    const struct evbuffer_cb_info *cbinfo,
    void *arg)
{
    struct bufferevent *bufev = arg;
    struct bufferevent_private *bufev_p =
        EVUTIL_UPCAST(bufev, struct bufferevent_private, bev);

    if (cbinfo->n_added && //evbuffer添加了数据
        (bufev->enabled & EV_WRITE) && //默认情况下是enable EV_WRITE的
        !event_pending(&bufev->ev_write, EV_WRITE, NULL) &&//这个event已经
被踢出event_base了
        !bufev_p->write_suspended) {//这个bufferevent的写并没有被挂起

        //把这个event添加到event_base中
        if (be_socket_add(&bufev->ev_write, &bufev->timeout_write) == -1)
 {
            /* Should we log this? */
        }
    }
}
```

这个函数首先进行一些判断，满足条件后就会把这个监听写事件的event添加到event_base中。其中event_pending函数就是判断这个bufev->ev_write是否已经被event_base删除了。关于event_pending，可以参考这里。

对于bufferevent_write，初次使用该函数的读者可能会有疑问：调用该函数后，参数data指向的内存空间能不能马上释放，还是要等到Libevent把data指向的数据都写到socket 缓存区才能删除？其实，从前一篇博文可以看到，evbuffer_add是直接复制一份用户要发送的数据到evbuffer缓存区的。所以，**调用完bufferevent_write，就可以马上释放参数data指向的内存空间**。

网上的关于Libevent的一些使用例子，包括我写的《Libevent使用例子，从简单到复杂》，都是在主线程中调用bufferevent_write函数写入数据的。从上面的分析可以得知，是可以马上把监听可写事件的event添加到event_base中。如果是在次线程调用该函数写入数据呢？此时，主线程可能还睡眠在poll、epoll这类的多路IO复用函数上。这种情况下能不能及时唤醒主线程呢？其实是可以的，只要你的Libevent在一开始使用了线程功能。具体的分析过程可以参考《evthread_notify_base通知主线程》。上面代码中的be_socket_add会调用event_add，而在次线程调用event_add就会调用evthread_notify_base通知主线程。

bufferevent_socket_connect

用户可以在调用bufferevent_socket_new函数时，传一个-1作为socket的文件描述符，然后调用bufferevent_socket_connect函数连接服务器，无需自己写代码调用connect函数连接服务器。

bufferevent_socket_connect函数会调用socket函数申请一个套接字fd，然后把这个fd设置成非阻塞的(这就导致了一些坑爹的事情)。接着就connect服务器，因为该socket fd是非阻塞的，所以不会等待，而是马上返回，连接这工作交给内核来完成。所以，返回后这个socket还没有真正连接上服务器。那么什么时候连接上呢？内核又是怎么通知通知用户呢？

一般来说，当可以往socket fd写东西了，那就说明已经连接上了。也就是说这个socket fd变成可写状态，就连接上了。

所以，对于“非阻塞connect”比较流行的做法是：用select或者poll这类多路IO复用函数监听该socket的可写事件。当这个socket触发了可写事件，然后再对这个socket调用getsockopt函数，做进一步的判断。

Libevent也是这样实现的，下面来看一下bufferevent_socket_connect函数。

```
//bufferevent_sock.c文件
int
bufferevent_socket_connect(struct bufferevent *bev,
    struct sockaddr *sa, int socklen)
{
    struct bufferevent_private *bufev_p =
        EVUTIL_UPCAST(bev, struct bufferevent_private, bev);

    evutil_socket_t fd;
    int r = 0;
    int result=-1;
    int ownfd = 0;

    _bufferevent_incref_and_lock(bev);

    if (!bufev_p)
        goto done;


    fd = bufferevent_getfd(bev);
    if (fd < 0) {//该bufferevent还没有设置fd
        if (!sa)
            goto done;
        fd = socket(sa->sa_family, SOCK_STREAM, 0);
        if (fd < 0)
            goto done;
        if (evutil_make_socket_nonblocking(fd)<0)//设置为非阻塞
            goto done;
        ownfd = 1;
    }
    if (sa) {
        r = evutil_socket_connect(&fd, sa, socklen);//非阻塞connect
        if (r < 0)
            goto freesock;
    }
    ...

    //为bufferevent里面的两个event设置监听的fd
    //后面会调用bufferevent_enable
    bufferevent_setfd(bev, fd);

    if (r == 0) {//暂时还没连接上，因为fd是非阻塞的
        //此时需要监听可写事件，当可写了，并且没有错误的话，就成功连接上了
        if (! be_socket_enable(bev, EV_WRITE)) {
            bufev_p->connecting = 1;//标志这个sockfd正在连接
            result = 0;
            goto done;
        }
    } else if (r == 1) {//已经连接上了
        /* The connect succeeded already. How very BSD of it. */
        result = 0;
```

```
            bufev_p->connecting = 1;
            event_active(&bev->ev_write, EV_WRITE, 1);//手动激活这个event
        } else {// connection refused
            /* The connect failed already.  How very BSD of it. */
            bufev_p->connection_refused = 1;
            bufev_p->connecting = 1;
            result = 0;
            event_active(&bev->ev_write, EV_WRITE, 1);//手动激活这个event
        }

        goto done;

    freesock:
        _bufferevent_run_eventcb(bev, BEV_EVENT_ERROR);//出现错误
        if (ownfd)
            evutil_closesocket(fd);
    done:
        _bufferevent_decref_and_unlock(bev);
        return result;
    }
```

这个函数比较多错误处理的代码，大致看一下就行了。有几个地方要注意，即使connect的时候被拒绝，或者已经连接上了，都会手动激活这个event。一个event即使没有加入event_base，也是可以手动激活的。具体原理参考这里。

无论是手动激活event，或者监听到这个event可写了，都是会调用bufferevent_writecb函数。现在再次看一下该函数，只看connect部分。

```
//bufferevent_sock.c文件
static void
bufferevent_writecb(evutil_socket_t fd, short event, void *arg)
{
    struct bufferevent_private *bufev_p =
        EVUTIL_UPCAST(bufev, struct bufferevent_private, bev);
    int connected = 0;

    _bufferevent_incref_and_lock(bufev);

    ...
    //正在连接。因为这个sockfd可能是非阻塞的，所以可能之前的connect还没
    //连接上。而判断该sockfd是否成功连接上了的一个方法是判断这个sockfd是否可写
    if (bufev_p->connecting) {
        //c等于1，说明已经连接成功
        //c等于0，说明还没连接上
        //c等于-1，说明发生错误
        int c = evutil_socket_finished_connecting(fd);


        if (bufev_p->connection_refused) {//在bufferevent_socket_connect中
被设置
          bufev_p->connection_refused = 0;
          c = -1;
        }

        if (c == 0)//还没连接上，继续监听可写吧
            goto done;


        //错误，或者已经连接上了
        bufev_p->connecting = 0;//修改标志值

        if (c < 0) {//错误
            event_del(&bufev->ev_write);
            event_del(&bufev->ev_read);
            _bufferevent_run_eventcb(bufev, BEV_EVENT_ERROR);
            goto done;

        } else {//连接上了。
            connected = 1;
            ...//win32

            //居然会调用用户设置的错误处理函数。太神奇了
            _bufferevent_run_eventcb(bufev,
                    BEV_EVENT_CONNECTED);
            if (!(bufev->enabled & EV_WRITE) || //默认都是enable EV_WRITE的
                bufev_p->write_suspended) {
                event_del(&bufev->ev_write);//不再需要监听可写。因为已经连接上
了
                goto done;
```

```
            }
        }
    }

    ...


 done:
    _bufferevent_decref_and_unlock(bufev);
}
```

可以看到无论是connect被拒绝、发生错误或者连接上了，都在这里做统一的处理。

如果已经连接上了，那么会调用用户设置event回调函数(网上也称之为错误处理函数)，通知用户已经连接上了。并且，还会把监听可写事件的event从event_base中删除，其理由在前面已经说过了。

函数evutil_socket_finished_connecting会检查这个socket，从而得知这个socket是处于什么状态。在bufferevent_socket_connect函数中，出现的一些错误，比如被拒绝，也是能通过这个函数检查出来的。所以可以在这里做统一的处理。该函数的内部是使用。贴一下这个函数的代码吧。

```
//evutil.c文件
//Return 1 for connected, 0 for not yet, -1 for error.
int
evutil_socket_finished_connecting(evutil_socket_t fd)
{
    int e;
    ev_socklen_t elen = sizeof(e);

    //用来检测这个fd是否已经连接上了，这个fd是非阻塞的
    //如果e的值被设为0，那么就说明连接上了。
    //否则e被设置为对应的错误值。
    if (getsockopt(fd, SOL_SOCKET, SO_ERROR, (void*)&e, &elen) < 0)
        return -1;

    if (e) {
        if (EVUTIL_ERR_CONNECT_RETRIABLE(e))//还没连接上
            return 0;
        EVUTIL_SET_SOCKET_ERROR(e);
        return -1;
    }

    return 1;
}
```

好长啊！终于写完了。