libevent文档

官方网站

官方文档

官方GitHub

libevent-2.1.11-stable.tar.gz

前言

Libevent是用于开发可伸缩网络服务器的事件通知库。Libevent API提供了一种机制，在文件描述符上发生特定事件或达到超时后执行回调函数。此外，Libevent还支持由于信号或定期超时而进行的回调。make

Libevent用于取代在事件驱动的网络服务器中的事件的循环。应用程序只需调用 event_base_dispatch（），然后动态添加或删除事件，而无需更改事件循环。

目前，Libevent支持/dev/poll、kqueue（2）、select（2）、poll（2）、epoll（4）和evports。内部事件机制完全独立于公开的事件API，简单的Libevent更新可以提供新的功能，而无需重新设计应用程序。因此，Libevent允许进行可移植的应用程序开发，并提供操作系统上可用的最可伸缩的事件通知机制。Libevent也可以用于多线程程序。Libevent应该在Linux、*BSD、Mac OS X、Solaris和Windows上编译。

其**设计目标**是：

- 可移植性：使用 libevent 编写的程序应该可以在 libevent 支持的所有平台上工作。即使 没有好的方式进行非阻塞 IO，libevent 也应该支持一般的方式，让程序可以在受限的环境中运行。
- 速度：libevent 尝试使用每个平台上最高速的非阻塞 IO 实现，并且不引入太多的额外开销。
- 可扩展性：libevent 被设计为程序即使需要上万个活动套接字的时候也可以良好工作。
- 方便：无论何时，最自然的使用 libevent 编写程序的方式应该是稳定的、可移植的。

libevent 由**下列组件构成**：

- evutil：用于抽象不同平台网络实现差异的通用功能。
- event 和 event_base：libevent 的核心，为各种平台特定的、基于事件的非阻塞 IO 后 端提供抽象 API，让程序可以知道套接字何时已经准备好，可以读或者写，并且处理基 本的超时功能，检测 OS 信号。
- bufferevent：为 libevent 基于事件的核心提供使用更方便的封装。除了通知程序套接字 已经准备好读写之外，还让程序可以请求缓冲的读写操作，可以知道何时 IO 已经真正 发生。
- evbuffer：在 bufferevent 层之下实现了缓冲功能，并且提供了方便有效的访问函数。
- evhttp：一个简单的 HTTP 客户端/服务器实现。
- evdns：一个简单的 DNS 客户端/服务器实现。
- evrpc：一个简单的 RPC 实现。

编译

linux

```
$ ./configure
$ make
$ make verify   # (optional)
$ sudo make install
```

ARM

```
$ ./configure --host=arm-linux --prefix=/usr/local/libevent_arm CC=arm-none-linux-gnueabi-gcc CXX=arm-none-linux-gnueabi-g++
$ make
$ make verify   # (optional)
$ sudo make install
```

Building and installing Libevent

生成库

创建 libevent 时，默认安装下列库：

- ibevent_core：所有核心的事件和缓冲功能，包含了所有的 event_base、evbuffer、 bufferevent 和工具函数。
- ibevent_extra：定义了程序可能需要，也可能不需要的协议特定功能，包括 HTTP、 DNS 和 RPC。
- libevent：这个库因为历史原因而存在，它包含 libevent_core 和 libevent_extra 的内容。 不应该使用这个库，未来版本的 libevent 可能去掉这个库。

某些平台上可能安装下列库：

- libevent_pthreads：添加基于 pthread 可移植线程库的线程和锁定实现。它独立于 libevent_core，这样程序使用 libevent 时就不需要链接到 pthread，除非实际上是以多线程的方式使用libevent。
- libevent_openssl：这个库为使用 bufferevent 和 OpenSSL 进行加密的通信提供支持。 它独立于 libevent_core，这样程序使用 libevent 时就不需要链接到 OpenSSL，除非实际使用的是加密连接。

概述

标准用法

使用Libevent的每个程序都必须包含<event2/event.h>头文件，并将-levent传递给链接器。（如果只需要主事件和缓冲的基于IO的代码，而不想链接任何协议代码，则可以改为链接-levent_core。）

库设置

在调用任何其他Libevent函数之前，需要设置库。如果要在多线程应用程序的多个线程中使用Libevent，则需要初始化线程支持：通常使用evthread_use_pthreads()或evthread_use_windows_threads()。有关更多信息，请参见<event2/thread.h>。

这也是可以用event_set_mem_functions替换Libevent的内存管理函数，并用event_enable_debug_mode()启用调试模式的地方。

创建event base

接下来，需要使用event_base_new()或event_base_new_with_config()创建一个event_base结构体。event_base负责跟踪哪些事件是“待处理的”（也就是说，被监视以查看它们是否变为活动的）以及哪些事件是“活动的”。每个事件都与单个event_base相关联。

事件通知

对于要监视的每个文件描述符，必须使用event_new（）创建事件结构。（您也可以声明一个事件结构并调用event_assign（）来初始化该结构的成员。）要启用通知，您可以通过调用event_add（）将该结构添加到监视的事件列表中。只要事件结构处于活动状态，它就必须保持分配状态，因此通常应该在堆上分配它。

调度事件。

最后，调用event_base_dispatch（）循环和分派事件。您还可以使用event_base_loop（）进行更细粒度的控制。

目前，一次只能有一个线程调度给定的event_base 。如果希望一次在多个线程中运行事件，可以有一个将工作添加到工作队列中的event_base ，也可以创建多个event_base对象。

I/O缓冲区

Libevent在常规事件回调的基础上提供了缓冲I/O抽象。这个抽象称为bufferevent。bufferevent提供自动填充和移除的输入和输出缓冲区。缓冲事件的用户不再直接处理I/O，而是读取输入和写入输出缓冲区。

一旦通过bufferevent_socket_new（）初始化，bufferevent结构就可以与bufferevent_enable（）和bufferevent_disable（）一起重复使用。您将调用bufferevent_read（）和bufferevent_write（），而不是直接读取和写入套接字。

启用读取后，bufferevent将尝试从文件描述符读取并调用读取回调。每当输出缓冲区被排放到写低水位线（默认为0）以下时，就会执行写回调。

有关更多信息，请参阅<event2/bufferevent*.h>。

计时器

Libevent还可以用来创建计时器，在一定时间过期后调用回调。evtimer_new（）宏返回要用作计时器的事件结构。要激活计时器，请调用evtimer_add（）。可以通过调用evtimer_del（）来停用计时器。（这些宏是围绕event_new（）、event_add（）和event_del（）的精简封装；您也可以使用它们。）

异步DNS解析

Libevent提供了一个异步DNS解析器，应该使用它来代替标准DNS解析器函数。有关更多详细信息，请参阅<event2/dns.h>函数。

事件驱动的HTTP服务器

Libevent提供了一个非常简单的事件驱动HTTP服务器，可以嵌入到程序中并用于服务HTTP请求。

要使用此功能，您需要在程序中包含<event2/http.h>头。有关详细信息，请参阅该标题。

RPC服务器和客户机的框架

Libevent提供了一个创建RPC服务器和客户端的框架。它负责对所有数据结构进行编组和解组。

API参考

要浏览libevent API的完整文档，请单击以下任何链接。

event2/event.h： 主libevent头

event2/thread.h： 多线程程序

event2/buffer.h 和event2/bufferevent.h： 网络读写的缓冲区管理

event2/util.h： 可移植非阻塞网络代码的实用函数

event2/dns.h ： 异步DNS解析

event2/http.h ： 基于libevent的嵌入式HTTP服务器

event2/rpc.h ： 创建RPC服务器和客户端的框架

event2/watch.h ： “准备”和“检查”观察者

详细说明

基于Libevent 2.0+，在C语言中编写快速可移植的异步网络IO程序。

一、设置libevent库

Libevent具有一些在整个进制中共享的会影响整个库的全局设置。

在调用Libevent库的任何其他部分之前，必须对这些设置进行任何更改。 如果您不这样做，Libevent可能会处于不一致状态。

1. Libevent中的日志消息

Libevent可以记录内部错误和警告。 如果它是在日志支持下编译的，它还会记录调试消息。 默认情况下，这些消息将写入stderr。 可以通过提供自己的日志记录功能来覆盖此行为。

接口

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

/* Deprecated; see note at the end of this section */
#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

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

void event_set_log_callback(event_log_cb cb);
```

要覆盖Libevent的日志记录行为，请编写与event_log_cb参数匹配的自己的函数，并将其作为参数传递给event_set_log_callback（）。 只要Libevent想要记录一条消息，它将把它传递给该函数。 可以通过以NULL作为参数再次调用event_set_log_callback（）来使Libevent恢复其默认行为。

示例

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

static void discard_cb(int severity, const char *msg)
{
    /* This callback does nothing. */
}

static FILE *logfile = NULL;
```

```
static void write_to_file_cb(int severity, const char *msg)
{
    const char *s;
    if (!logfile)
        return;
    switch (severity) {
        case _EVENT_LOG_DEBUG: s = "debug"; break;
        case _EVENT_LOG_MSG:   s = "msg";   break;
        case _EVENT_LOG_WARN:  s = "warn";  break;
        case _EVENT_LOG_ERR:   s = "error"; break;
        default:               s = "?";     break; /* never reached */
    }
    fprintf(logfile, "[%s] %s\n", s, msg);
}

/* Turn off all logging from Libevent. */
void suppress_logging(void)
{
    event_set_log_callback(discard_cb);
}

/* Redirect all Libevent log messages to the C stdio file 'f'. */
void set_logfile(FILE *f)
{
    logfile = f;
    event_set_log_callback(write_to_file_cb);
}
```

注意

在用户提供的event_log_cb回调中调用Libevent函数是不安全的！ 例如，如果您尝试编写一个使用bufferevents向网络套接字发送警告消息的日志回调，则很可能会遇到奇怪且难以诊断的错误。 在将来的Libevent版本中，某些功能可能会取消此限制。

通常，调试日志不会启用，并且不会发送到日志回调。 如果Libevent要支持它们，则可以手动将其打开。

接口

```
#define EVENT_DBG_NONE 0
#define EVENT_DBG_ALL 0xffffffffu

void event_enable_debug_logging(ev_uint32_t which);
```

调试日志很冗长，在大多数情况下不一定有用。 使用EVENT_DBG_NONE调用event_enable_debug_logging（）将获得默认行为。 使用EVENT_DBG_ALL调用它会打开所有受支持的调试日志。 将来的版本中可能会支持更多细致的选项。

这些函数在<event2 / event.h>中声明。 它们首先出现在Libevent 1.0c中，除event_enable_debug_logging（）首次出现在Libevent 2.1.1-alpha中。

兼容性说明

在Libevent 2.0.19-稳定版本之前，EVENT_LOG_ *宏的名称以下划线开头：*EVENT_LOG_DEBUG*，EVENT_LOG_MSG，*EVENT_LOG_WARN*和EVENT_LOG_ERR。 这些较早的名称已被弃用，仅应用于与Libevent 2.0.18-stable和更早版本的向后兼容。 在将来的Libevent版本中可能会删除它们。

2. 处理致命错误

当Libevent检测到不可恢复的内部错误（例如，损坏的数据结构）时，其默认行为是调用exit（）或abort（）退出当前运行的进程。 这些错误几乎总是意味着某个地方存在bug：在你的代码中，或者在Libevent本身中。

如果您希望应用程序更优雅地处理致命错误，则可以通过提供Libevent代替退出而调用的函数来覆盖Libevent的行为。

接口

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

要使用这些函数，首先定义一个新函数，遇到致命错误时Libevent应该调用该函数，然后将其传递给event_set_fatal_callback（）。以后，如果Libevent遇到致命错误，它将调用您提供的函数。

该函数不应将控制权返回给Libevent。这样做可能会导致不确定的行为，并且Libevent可能仍会退出以避免崩溃。一旦调用了函数，就不应调用任何其他Libevent函数。

这些函数在<event2 / event.h>中声明。首先出现在Libevent 2.0.3-alpha中。

3. 内存管理

默认情况下，Libevent使用C库的内存管理功能从堆中分配内存。您可以通过提供自己的malloc，realloc和free替换项来使Libevent使用另一个内存管理器。如果有想要使用Libevent的更高效的分配器，或者如果想要使用Libevent的检测化分配器来查找内存泄漏，则可能需要这样做。

接口

```
void event_set_mem_functions(void *(*malloc_fn)(size_t sz),
                             void *(*realloc_fn)(void *ptr, size_t sz),
                             void (*free_fn)(void *ptr));
```

这是一个简单的示例，该示例将Libevent的分配函数替换为对已分配的字节总数进行计数的变体。 实际上，您可能希望在此处添加锁，以防止Libevent在多个线程中运行时出错。

示例

```
#include <event2/event.h>
#include <sys/types.h>
#include <stdlib.h>

/* This union's purpose is to be as big as the largest of all the
 * types it contains. */
union alignment {
    size_t sz;
    void *ptr;
    double dbl;
};
/* We need to make sure that everything we return is on the right
   alignment to hold anything, including a double. */
#define ALIGNMENT sizeof(union alignment)

/* We need to do this cast-to-char* trick on our pointers to adjust
   them; doing arithmetic on a void* is not standard. */
#define OUTPTR(ptr) (((char*)ptr)+ALIGNMENT)
#define INPTR(ptr) (((char*)ptr)-ALIGNMENT)
```

```
static size_t total_allocated = 0;
static void *replacement_malloc(size_t sz)
{
    void *chunk = malloc(sz + ALIGNMENT);
    if (!chunk) return chunk;
    total_allocated += sz;
    *(size_t*)chunk = sz;
    return OUTPTR(chunk);
}
static void *replacement_realloc(void *ptr, size_t sz)
{
    size_t old_size = 0;
    if (ptr) {
        ptr = INPTR(ptr);
        old_size = *(size_t*)ptr;
    }
    ptr = realloc(ptr, sz + ALIGNMENT);
    if (!ptr)
        return NULL;
    *(size_t*)ptr = sz;
    total_allocated = total_allocated - old_size + sz;
    return OUTPTR(ptr);
}
static void replacement_free(void *ptr)
{
    ptr = INPTR(ptr);
    total_allocated -= *(size_t*)ptr;
    free(ptr);
}
void start_counting_bytes(void)
{
    event_set_mem_functions(replacement_malloc,
                            replacement_realloc,
                            replacement_free);
}
```

注意

- 替换内存管理功能会影响以后所有从Libevent分配，调整大小或释放内存的调用。因此，在调用任何其他Libevent函数之前，需要确保已替换函数。否则，Libevent将使用你的free版本释放从C库的malloc版本分配的内存。
- 你的malloc和realloc函数需要返回与C库相同的内存块。
- 你的realloc函数需要正确处理realloc（NULL，sz）（即，将其视为malloc（sz））。
- 你的realloc函数需要正确处理realloc（ptr，0）（也就是说，将其视为free（ptr））。
- 你的free函数不需要处理free（NULL）。
- 你的malloc函数不需要处理malloc（0）。
- 如果从多个线程中使用Libevent，则替换后的内存管理功能必须是线程安全的。
- Libevent将使用这些函数来分配返回给您的内存。因此，如果要释放由Libevent函数分配和返回的内存，并且已经替换了malloc和realloc函数，则可能必须使用替换free函数来释放它。

在<event2 / event.h>中声明了event_set_mem_functions（）函数。它首先出现在Libevent 2.0.1-alpha中。

可以在禁用event_set_mem_functions（）的情况下构建Libevent。如果是这样，则使用event_set_mem_functions的程序将不会编译或链接。在Libevent 2.0.2-alpha和更高版本中，您可以通过检查是否已定义EVENT_SET_MEM_FUNCTIONS_IMPLEMENTED宏来检测是否存在event_set_mem_functions（）。

4. 锁和线程

编写多线程程序，在多个线程同时访问相同的数据并不总是安全的。

Libevent结构通常可以在多线程中以三种方式工作。

- 有些结构本质上是单线程的：从多个线程同时使用它们永远是不安全的。
- 某些结构是有可选的锁：可以告诉每个对象Libevent是否需要在多线程使用每个对象。
- 某些结构始终是锁定的：如果Libevent在锁定支持下运行，则始终可以安全地多个线程使用它们。

为获取锁，在调用分配需要在多个线程间共享的结构体的 libevent 函数之前，必须告知 libevent 使用哪个锁函数。

如果使用 pthreads 库，或者使用 Windows 本地线程代码，那么已经有设置 libevent 使用正确的 pthreads 或者 Windows 函数的预定义函数。

接口

```
#ifdef WIN32
int evthread_use_windows_threads(void);
#define EVTHREAD_USE_WINDOWS_THREADS_IMPLEMENTED
#endif
#ifdef _EVENT_HAVE_PTHREADS
int evthread_use_pthreads(void);
#define EVTHREAD_USE_PTHREADS_IMPLEMENTED
#endif
```

这些函数在成功时都返回0，失败时返回-1。

如果使用不同的线程库，则需要一些额外的工作，必须使用你的线程库来定义函数去实现：

- 锁
- 锁定
- 解锁
- 分配锁
- 析构锁
- 条件变量
- 创建条件变量
- 析构条件变量
- 等待条件变量
- 触发/广播某条件变量
- 线程
- 线程 ID 检测

使用 evthread_set_lock_callbacks 和 evthread_set_id_callback 接口告知 libevent 这些函

数。

接口

```
#define EVTHREAD_WRITE  0x04
#define EVTHREAD_READ   0x08
#define EVTHREAD_TRY    0x10

#define EVTHREAD_LOCKTYPE_RECURSIVE 1
#define EVTHREAD_LOCKTYPE_READWRITE 2

#define EVTHREAD_LOCK_API_VERSION 1
```

```
struct evthread_lock_callbacks {
        int lock_api_version;
        unsigned supported_locktypes;
        void *(*alloc)(unsigned locktype);
        void (*free)(void *lock, unsigned locktype);
        int (*lock)(unsigned mode, void *lock);
        int (*unlock)(unsigned mode, void *lock);
};

int evthread_set_lock_callbacks(const struct evthread_lock_callbacks *);

void evthread_set_id_callback(unsigned long (*id_fn)(void));

struct evthread_condition_callbacks {
         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);
};

int evthread_set_condition_callbacks(
         const struct evthread_condition_callbacks *);
```

evthread_lock_callbacks 结构体描述的锁回调函数及其能力。对于上述版本，lock_api_version 字段必须设置为 EVTHREAD_LOCK_API_VERSION 。必须设置 supported_locktypes 字段为 EVTHREAD_LOCKTYPE_*常量的组合以描述支持的锁类型（在 2.0.4-alpha 版本中， EVTHREAD_LOCK_RECURSIVE 是必须的，EVTHREAD_LOCK_READWRITE 则没有使用）。alloc 函数必须返回指定类型的新锁；free 函数必须释放指定类型锁持有的所有资源；lock 函数必须试图以指定模式请求锁定，如果成功则返回0，失败则返回非零；unlock 函数必须试图解锁，成功则返回0，否则返回非零。

可识别的锁类型有：

- 0：通常的，不必递归的锁。
- EVTHREAD_LOCKTYPE_RECURSIVE：不会阻塞已经持有它的线程的锁。一旦持有它的线程进行原来锁定次数的解锁，其他线程立刻就可以请求它了。
- EVTHREAD_LOCKTYPE_READWRITE：可以让多个线程同时因为读而持有它，但是任何时刻只有一个线程因为写而持有它。写操作排斥所有读操作。

可识别的锁模式有：

- 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

id_fn 参数必须是一个函数，它返回一个无符号长整数，标识调用此函数的线程。对于相同线程，这个函数应该总是返回同样的值；而对于同时调用该函数的不同线程，必须返回不同的值。
evthread_condition_callbacks 结构体描述了与条件变量相关的回调函数。对于上述版本，condition_api_version 字段必须设置为EVTHREAD_CONDITION_API_VERSION 。alloc_condition 函数必须返回到新条件变量的指针。它接受0作为其参数。free_condition 函数必须释放条件变量持有

的存储器和资源。wait_condition 函数要求三个参数：一个由alloc_condition 分配的条件变量，一个由你提供的 evthread_lock_callbacks.alloc 函数分配的锁，以及一个可选的超时值。调用本函数时，必须已经持有参数指定的锁；本函数应该释放指定的锁，等待条件变量成为授信状态，或者直到指定的超时时间已经流逝（可选）。wait_condition 应该在错误时返回-1，条件变量授信时返回0，超时时返回1。返回之前，函数应该确定其再次持有锁。最后，signal_condition 函数应该唤醒等待该条件变量的某个线程（broadcast 参数为 false 时），或者唤醒等待条件变量的所有线程（broadcast 参数为 true 时）。只有在持有与条件变量相关的锁的时候，才能够进行这些操作。

关于条件变量的更多信息，请查看 pthreads 的 pthread_cond_*函数文档，或者 Windows的 CONDITION_VARIABLE（Windows Vista 新引入的）函数文档。

示例

关于使用这些函数的示例，请查看 Libevent 源代码发布版本中的 evthread_pthread.c 和 evthread_win32.c 文件。

这些函数在<event2/thread.h>中声明，其中大多数在 2.0.4-alpha 版本中首次出现。2.0.1-alpha 到 2.0.3-alpha 使用较老版本的锁函数。event_use_pthreads 函数要求程序链接event_pthreads 库。

条件变量函数是2.0.7-rc 版本新引入的，用于解决某些棘手的死锁问题。

可以创建禁止锁支持的 libevent。这时候已创建的使用上述线程相关函数的程序将不能运行。

5. 调试锁的使用

为帮助调试锁的使用，libevent 有一个可选的“锁调试”特征。这个特征包装了锁调用，以便捕获典型的锁错误，包括：

- 解锁并没有持有的锁
- 重新锁定一个非递归锁

如果发生这些错误中的某一个，libevent 将给出断言失败并且退出。

接口

```
void evthread_enable_lock_debugging(void);
#define evthread_enable_lock_debuging() evthread_enable_lock_debugging()
```

注意

必须在创建或者使用任何锁之前调用这个函数。为安全起见，请在设置完线程函数后立即调用这个函数。

此功能是Libevent 2.0.4-alpha中的新增功能，拼写错误的名称为“evthread_enable_lock_debuging（）”。 拼写固定为2.1.2-alpha中的evthread_enable_lock_debugging（）； 目前都支持这两个名称。

6. 调试事件的使用

libevent 可以检测使用事件时的一些常见错误并且进行报告。这些错误包括：

- 将未初始化的 event 结构体当作已经初始化的
- 试图重新初始化未决的 event 结构体

跟踪哪些事件已经初始化需要使用额外的内存和处理器时间，所以只应该在真正调试程序的时候才启用调试模式。

接口

```
void event_enable_debug_mode(void);
```

必须在创建任何 event_base 之前调用这个函数。

如果在调试模式下使用大量由 event_assign（而不是 event_new）创建的事件，程序可能会耗尽内存，这是因为没有方式可以告知 libevent 由 event_assign 创建的事件不会再被使用了（可以调用 event_free 告知由 event_new 创建的事件已经无效了）。如果想在调试时避免耗尽内存，可以显式告知 libevent 这些事件不再被当作已分配的了：

接口

```
void event_debug_unassign(struct event *ev);
```

没有启用调试的时候调用 event_debug_unassign 没有效果。

示例

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

#include <stdlib.h>

void cb(evutil_socket_t fd, short what, void *ptr)
{
    /* We pass 'NULL' as the callback pointer for the heap allocated
     * event, and we pass the event itself as the callback pointer
     * for the stack-allocated event. */
    struct event *ev = ptr;

    if (ev)
        event_debug_unassign(ev);
}

/* Here's a simple mainloop that waits until fd1 and fd2 are both
 * ready to read. */
void mainloop(evutil_socket_t fd1, evutil_socket_t fd2, int debug_mode)
{
    struct event_base *base;
    struct event event_on_stack, *event_on_heap;

    if (debug_mode)
       event_enable_debug_mode();

    base = event_base_new();

    event_on_heap = event_new(base, fd1, EV_READ, cb, NULL);
    event_assign(&event_on_stack, base, fd2, EV_READ, cb, &event_on_stack);

    event_add(event_on_heap, NULL);
    event_add(&event_on_stack, NULL);

    event_base_dispatch(base);

    event_free(event_on_heap);
    event_base_free(base);
}
```

详细的事件调试是一项只能在编译时使用CFLAGS环境变量“ -DUSE_DEBUG”启用的功能。 启用此标志后，针对Libevent编译的任何程序都将输出非常详细的日志，详细说明后端的低级活动。 这些日志包括但不限于以下内容：

- 活动添加
- 事件删除
- 平台特定的事件通知信息

无法通过API调用启用或禁用此功能，因此只能在开发人员内部使用。

这些调试功能已添加到Libevent 2.0.4-alpha中。

7.检测libevent的版本

新版本的 libevent 会添加特征，移除 bug。有时候需要检测 libevent 的版本，以便：

- 检测已安装的 libevent 版本是否可用于创建你的程序
- 为调试显示 libevent 的版本
- 检测 libevent 的版本，以便向用户警告 bug，或者提示要做的工作

接口

```
#define LIBEVENT_VERSION_NUMBER 0x02000300
#define LIBEVENT_VERSION "2.0.3-alpha"
const char *event_get_version(void);
ev_uint32_t event_get_version_number(void);
```

宏返回编译时的 libevent 版本；函数返回运行时的 libevent 版本。注意：如果动态链接到libevent，这两个版本可能不同。

可以获取两种格式的 libevent 版本：用于显示给用户的字符串版本，或者用于数值比较的4字节整数版本。整数格式使用高字节表示主版本，低字节表示副版本，第三字节表示修正版本，最低字节表示发布状态：0表示发布，非零表示某特定发布版本的后续开发序列。

所以，libevent 2.0.1-alpha 发布版本的版本号是[02 00 01 00]，或者说0x02000100。2.0.1-alpha 和 2.0.2-alpha 之间的开发版本可能是[02 00 01 08]，或者说0x02000108。

示例：编译时检测

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

#if !defined(LIBEVENT_VERSION_NUMBER) || LIBEVENT_VERSION_NUMBER < 0x02000100
#error "This version of Libevent is not supported; Get 2.0.1-alpha or later."
#endif

int
make_sandwich(void)
{
        /* Let's suppose that Libevent 6.0.5 introduces a make-me-a
           sandwich function. */
#if LIBEVENT_VERSION_NUMBER >= 0x06000500
        evutil_make_me_a_sandwich();
        return 0;
#else
        return -1;
#endif
}
```

示例：运行时检测

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

int
check_for_old_version(void)
{
    const char *v = event_get_version();
    /* This is a dumb way to do it, but it is the only thing that works
       before Libevent 2.0. */
    if (!strncmp(v, "0.", 2) ||
        !strncmp(v, "1.1", 3) ||
        !strncmp(v, "1.2", 3) ||
        !strncmp(v, "1.3", 3)) {

        printf("Your version of Libevent is very old.  If you run into bugs,"
               " consider upgrading.\n");
        return -1;
    } else {
        printf("Running with Libevent version %s\n", v);
        return 0;
    }
}

int
check_version_match(void)
{
    ev_uint32_t v_compile, v_run;
    v_compile = LIBEVENT_VERSION_NUMBER;
    v_run = event_get_version_number();
    if ((v_compile & 0xffff0000) != (v_run & 0xffff0000)) {
        printf("Running with a Libevent version (%s) very different from the "
               "one we were built with (%s).\n", event_get_version(),
               LIBEVENT_VERSION);
        return -1;
    }
    return 0;
}
```

本节描述的宏和函数定义在<event2/event.h>中。event_get_version 函数首次出现在1.0c版本；其他的首次出现在2.0.1-alpha 版本。

8. 释放全局的Libevent结构

即使释放了使用Libevent分配的所有对象，也将剩下一些全局分配的结构。通常这不是问题：退出流程后，无论如何都将对其进行清理。但是拥有这些结构可能会使某些调试工具误以为Libevent正在泄漏资源。如果需要确保Libevent已发布所有内部库全局数据结构，则可以调用：

接口

```
void libevent_global_shutdown（void）；
```

此函数不会释放Libevent函数返回给您的任何结构。如果要在退出之前释放所有内容，则需要自己释放所有事件，event_bases，bufferevents等。

调用libevent_global_shutdown（）将使其他Libevent函数的行为无法预期。除了作为您的程序调用的最后一个Libevent函数外，不要调用它。一个例外是libevent_global_shutdown（）是幂等(idempotent)的：可以调用它，即使它已经被调用也可以。

此函数在<event2 / event.h>中声明。它是在Libevent 2.1.1-alpha中引入的。

二、创建event_base

使用 libevent 函数之前需要分配一个或者多个 event_base 结构体。每个 event_base 结构体持有一个事件集合，可以检测以确定哪个事件是激活的。

如果设置 event_base 使用锁，则可以安全地在多个线程中访问它。然而，其事件循环只能运行在一个线程中。如果需要用多个线程检测 IO，则需要为每个线程使用一个 event_base。

每个 event_base 都有一种用于检测哪种事件已经就绪的“方法”，或者说后端。可以识别的方法有：

- select
- poll
- epoll
- kqueue
- devpoll
- evport
- win32

用户可以用环境变量禁止某些特定的后端。比如说，要禁止 kqueue 后端，可以设置 EVENT_NOKQUEUE 环境变量 。 如果要用编程的方法禁止后端 ， 请看下面关于 event_config_avoid_method（）的说明。

1. 建立默认的 event_base

event_base_new（）函数分配并且返回一个新的具有默认设置的 event_base。函数会检测环境变量，返回一个到 event_base 的指针。如果发生错误，则返回 NULL。选择各种方法时，函数会选择 OS 支持的最快方法。

接口

```
struct event_base *event_base_new(void);
```

大多数程序使用这个函数就够了。

event_base_new（）函数声明在<event2/event.h>中，首次出现在 libevent 1.4.3版。

2. 创建复杂的 event_base

要对取得什么类型的 event_base 有更多的控制，就需要使用 event_config。event_config是一个容纳 event_base 配置信息的不透明结构体。需要 event_base 时，将 event_config传递给 event_base_new_with_config（） 。

接口

```
struct event_config *event_config_new(void);
struct event_base *event_base_new_with_config(const struct event_config *cfg);
void event_config_free(struct event_config *cfg);
```

要使用这些函数分配 event_base，先调用 event_config_new（）分配一个 event_config。 然后，对 event_config 调用其它函数，设置所需要的 event_base 特征。最后，调用 event_base_new_with_config（）获取新的 event_base。完成工作后，使用 event_config_free（）释放 event_config。

接口

```
int event_config_avoid_method(struct event_config *cfg, const char *method);

enum event_method_feature {
    EV_FEATURE_ET = 0x01,
    EV_FEATURE_O1 = 0x02,
    EV_FEATURE_FDS = 0x04,
};
int event_config_require_features(struct event_config *cfg,
                                  enum event_method_feature feature);

enum event_base_config_flag {
    EVENT_BASE_FLAG_NOLOCK = 0x01,
    EVENT_BASE_FLAG_IGNORE_ENV = 0x02,
    EVENT_BASE_FLAG_STARTUP_IOCP = 0x04,
    EVENT_BASE_FLAG_NO_CACHE_TIME = 0x08,
    EVENT_BASE_FLAG_EPOLL_USE_CHANGELIST = 0x10,
    EVENT_BASE_FLAG_PRECISE_TIMER = 0x20
};
int event_config_set_flag(struct event_config *cfg,
```

调用 event_config_avoid_method（）可以通过名字让 libevent 避免使用特定的可用后端。

调用 event_config_require_feature（）让 libevent 不要使用不能提供所有功能的后端。

调用 event_config_set_flag（）让 libevent 在创建 event_base 时设置一个或者多个将在下面介绍的运行时标志。

event_config_require_features（）可识别的特征值有：

- EV_FEATURE_ET：要求支持边沿触发的后端
- EV_FEATURE_O1：要求添加、删除单个事件，或者确定哪个事件激活的操作是 O（1）复杂度的后端
- EV_FEATURE_FDS：要求支持任意文件描述符，而不仅仅是套接字的后端

event_config_set_flag（）可识别的选项值有：

- **EVENT_BASE_FLAG_NOLOCK**：不要为 event_base 分配锁。设置这个选项可以为event_base 节省一点用于锁定和解锁的时间，但是让在多个线程中访问 event_base成为不安全的。
- **EVENT_BASE_FLAG_IGNORE_ENV**：选择使用的后端时，不要检测 EVENT_*环境变量。使用这个标志需要三思：这会让用户更难调试你的程序与 libevent 的交互。
- **EVENT_BASE_FLAG_STARTUP_IOCP**：仅用于 Windows，让 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。
- **EVENT_BASE_FLAG_NO_CACHE_TIME**：不是在事件循环每次准备执行超时回调时检测当前时间，而是在每次超时回调后进行检测。注意：这会消耗更多的 CPU 时间。
- **EVENT_BASE_FLAG_EPOLL_USE_CHANGELIST**：告诉 libevent，如果决定使用epoll 后端，可以安全地使用更快的基于 changelist 的后端。epoll-changelist 后端可以在后端的分发函数调用之间，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后端会触发一个内核 bug，导致不正确的结果。在不使用 epoll 后端的情况下，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。

- **EVENT_BASE_FLAG_PRECISE_TIMER**: 默认情况下，Libevent尝试使用操作系统提供的最快的可用计时机制。 如果存在较慢的计时机制，可以提供更精细的计时精度，则此标志告诉Libevent改用该计时机制。 如果操作系统不提供这种慢速但精确的机制，则此标志无效。

上述操作 event_config 的函数都在成功时返回0，失败时返回-1。

> **注意** 设置 event_config，请求 OS 不能提供的后端是很容易的。比如说，对于 libevent 2.0.1-alpha，在 Windows 中是没有 O（1）后端的；在 Linux 中也没有同时提供 EV_FEATURE_FDS 和 EV_FEATURE_O1 特征的后端。如果创建了 libevent 不能满足的配置，event_base_new_with_config（）会返回 NULL

接口

```
int event_config_set_num_cpus_hint(struct event_config *cfg, int cpus)
```

这个函数当前仅在 Windows 上使用 IOCP 时有用，虽然将来可能在其他平台上有用。这个函数告诉 event_config 在生成多线程 event_base 的时候，应该试图使用给定数目的 CPU。注意这仅仅是一个提示：event_base 使用的CPU 可能比你选择的要少。

接口

```
int event_config_set_max_dispatch_interval(struct event_config *cfg,
    const struct timeval *max_interval, int max_callbacks,
    int min_priority);
```

此功能通过限制在检查更多高优先级事件之前可以调用多少个低优先级事件回调来防止优先级倒置。 如果max_interval为非null，则事件循环将检查每个回调之后的时间，如果max_interval已超过，则重新扫描高优先级事件。 如果max_callbacks为非负数，则在调用max_callbacks回调后，事件循环还会检查更多事件。 这些规则适用于min_priority或更高级别的任何事件。

示例：优先使用边缘触发的后端

```
struct event_config *cfg;
struct event_base *base;
int i;

/* My program wants to use edge-triggered events if at all possible.  So
   I'll try to get a base twice: Once insisting on edge-triggered IO, and
   once not. */
for (i=0; i<2; ++i) {
    cfg = event_config_new();

    /* I don't like select. */
    event_config_avoid_method(cfg, "select");

    if (i == 0)
        event_config_require_features(cfg, EV_FEATURE_ET);

    base = event_base_new_with_config(cfg);
    event_config_free(cfg);
    if (base)
        break;

    /* If we get here, event_base_new_with_config() returned NULL.  If
       this is the first time around the loop, we'll try again without
       setting EV_FEATURE_ET.  If this is the second time around the
```

```
        loop, we'll give up. */
}
```

示例：避免优先级倒置

```
struct event_config *cfg;
struct event_base *base;

cfg = event_config_new();
if (!cfg)
   /* Handle error */;

/* I'm going to have events running at two priorities.  I expect that
   some of my priority-1 events are going to have pretty slow callbacks,
   so I don't want more than 100 msec to elapse (or 5 callbacks) before
   checking for priority-0 events. */
struct timeval msec_100 = { 0, 100*1000 };
event_config_set_max_dispatch_interval(cfg, &msec_100, 5, 1);

base = event_base_new_with_config(cfg);
if (!base)
   /* Handle error */;

event_base_priority_init(base, 2);
```

这些函数和类型在<event2 / event.h>中声明。

EVENT_BASE_FLAG_IGNORE_ENV标志首先出现在Libevent 2.0.2-alpha中。
EVENT_BASE_FLAG_PRECISE_TIMER标志首先出现在Libevent 2.1.2-alpha中。
event_config_set_num_cpus_hint（）函数是Libevent 2.0.7-rc中的新增功能，而event_config_set_max_dispatch_interval（）是2.1.1-alpha中的新增功能。 本节中的所有其他内容首先出现在Libevent 2.0.1-alpha中。

3. 检查 event_base 的后端方法

有时候需要检查 event_base 支持哪些特征，或者当前使用哪种方法。

接口

```
const char **event_get_supported_methods(void);
```

event_get_supported_methods（）函数返回一个指针，指向 libevent 支持的方法名字数组。这个数组的最后一个元素是 NULL。

示例

```
int i;
const char **methods = event_get_supported_methods();
printf("Starting Libevent %s.  Available methods are:\n",
    event_get_version());
for (i=0; methods[i] != NULL; ++i) {
    printf("    %s\n", methods[i]);
}
```

> **注意**这个函数返回 libevent 被编译以支持的方法列表。然而 libevent 运行的时候，操作系统可能不能支持所有方法。比如说，可能 OS X 版本中的 kqueue 的 bug 太多，无法使用。

接口

```
const char *event_base_get_method(const struct event_base *base);
enum event_method_feature event_base_get_features(const struct event_base
*base);
```

event_base_get_method（）返回 event_base 正在使用的方法。

event_base_get_features（）返回 event_base 支持的特征的比特掩码。

示例

```
struct event_base *base;
enum event_method_feature f;

base = event_base_new();
if (!base) {
    puts("Couldn't get an event_base!");
} else {
    printf("Using Libevent with backend method %s.",
        event_base_get_method(base));
    f = event_base_get_features(base);
    if ((f & EV_FEATURE_ET))
        printf("  Edge-triggered events are supported.");
    if ((f & EV_FEATURE_O1))
        printf("  O(1) event notification is supported.");
    if ((f & EV_FEATURE_FDS))
        printf("  All FD types are supported.");
    puts("");
}
```

这个函数定义在<event2/event.h>中。event_base_get_method（）首次出现在1.4.3版本中，其他函数首次出现在2.0.1-alpha 版本中。

4. 释放 event_base

使用完 event_base 之后，使用 event_base_free（）进行释放。
接口

```
void event_base_free(struct event_base *base);
```

> 注意：这个函数不会释放当前与 event_base 关联的任何事件，或者关闭他们的套接字，或者释放任何指针。

event_base_free（）定义在<event2/event.h>中，首次由 libevent 1.2实现。

5. 设置 event_base 的优先级

libevent 支持为事件设置多个优先级。然而，event_base 默认只支持单个优先级。可以调用 event_base_priority_init（）设置 event_base 的优先级数目。

接口

```
int event_base_priority_init(struct event_base *base, int n_priorities);
```

成功时这个函数返回0，失败时返回-1。base 是要修改的 event_base，n_priorities 是要支持的优先级数目，这个数目至少是1。每个新的事件可用的优先级将从0（最高）到n_priorities-1（最低）。
常量 EVENT_MAX_PRIORITIES 表示 n_priorities 的上限。调用这个函数时为 n_priorities给出更大的值是错误的。

> **注意**
> 必须在任何事件激活之前调用这个函数，最好在创建 event_base 后立刻调用。

要查找某个数据库当前支持的优先级数量，可以调用event_base_getnpriorities（）。

接口

```
int event_base_get_npriorities(struct event_base *base);
```

返回值等于基础中配置的优先级数。 因此，如果event_base_get_npriorities（）返回3，则允许的优先级值为0、1和2。

示例
关于示例，请看 event_priority_set 的文档。
默认情况下，与 event_base 相关联的事件将被初始化为具有优先级 n_priorities / 2。
event_base_priority_init（）函数定义在<event2/event.h>中，从 libevent 1.0版就可用了。

6. 在 fork（）之后重新初始化 event_base

不是所有事件后端都在调用 fork（）之后可以正确工作。所以，如果在使用 fork（）或者其他相关系统调用启动新进程之后，希望在新进程中继续使用 event_base，就需要进行重新初始化。
接口

```
int event_reinit(struct event_base *base);
```

成功时这个函数返回0，失败时返回-1。
示例

```
struct event_base *base = event_base_new();

/* ... add some events to the event_base ... */

if (fork()) {
    /* In parent */
    continue_running_parent(base); /*...*/
} else {
    /* In child */
    event_reinit(base);
    continue_running_child(base); /*...*/
}
```

event_reinit（）定义在<event2/event.h>中，在 libevent 1.4.3-alpha 版中首次可用。

三、创建event loop

1. 运行循环

一旦有了一个已经注册了某些事件的 event_base（关于如何创建和注册事件请看下一节），就需要让 libevent 等待事件并且通知事件的发生。

接口

```
#define EVLOOP_ONCE             0x01
#define EVLOOP_NONBLOCK         0x02
#define EVLOOP_NO_EXIT_ON_EMPTY 0x04

int event_base_loop(struct event_base *base, int flags);
```

默认情况下，event_base_loop（）函数将运行一个event_base，直到其中没有注册更多事件为止。为了运行循环，它反复检查是否已触发任何已注册的事件（例如，读取事件的文件描述符是否已准备好读取，或者超时事件的超时是否已准备就绪）。一旦发生这种情况，它将所有触发的事件标记为“活动”，并开始运行它们。

可以通过在event_base_loop（）的flags参数中设置一个或多个标志来更改其行为。

如果设置了**EVLOOP_ONCE**，则循环将等待，直到某些事件变为活动状态，然后运行活动事件，直到没有其他要运行的状态，然后返回。

如果设置了**EVLOOP_NONBLOCK**，则该循环将不等待事件触发：它将仅检查是否有任何事件准备立即触发，并在可能时运行其回调。

通常，一旦没有挂起或活动事件，循环将立即退出。您可以通过传递**EVLOOP_NO_EXIT_ON_EMPTY**标志来覆盖此行为-例如，如果要从其他线程添加事件。如果您确实设置了EVLOOP_NO_EXIT_ON_EMPTY，则循环将一直运行，直到有人调用event_base_loopbreak（）或调用event_base_loopexit（）或发生错误为止。

完成后，如果event_base_loop（）正常退出，则返回0；如果由于后端发生一些未处理的错误而退出，则返回-1；如果由于没有更多pending 或active 事件而退出，则返回1。

为了帮助理解，以下是event_base_loop算法的大致摘要：

伪代码

```
while (any events are registered with the loop,
        or EVLOOP_NO_EXIT_ON_EMPTY was set) {

    if (EVLOOP_NONBLOCK was set, or any events are already active)
        If any registered events have triggered, mark them active.
    else
        Wait until at least one event has triggered, and mark it active.

    for (p = 0; p < n_priorities; ++p) {
       if (any event with priority of p is active) {
          Run all active events with priority of p.
          break; /* Do not run any events of a less important priority */
       }
    }

    if (EVLOOP_ONCE was set or EVLOOP_NONBLOCK was set)
       break;
}
```

为方便起见，也可以调用:

接口

```
int event_base_dispatch(struct event_base *base);
```

event_base_dispatch （）等同于没有设置标志的event_base_loop （）。

event_base_dispatch （）将一直运行，直到没有已经注册的事件了，或者调用 了event_base_loopbreak（）或者 event_base_loopexit（）为止。

这些函数定义在<event2/event.h>中，从 libevent 1.0版就存在了。

2. 停止循环

如果想在移除所有已注册的事件之前停止活动的事件循环，可以调用两个稍有不同的函数。 接口

```
int event_base_loopexit(struct event_base *base,
                        const struct timeval *tv);
int event_base_loopbreak(struct event_base *base);
```

event_base_loopexit（）让 event_base 在给定时间之后停止循环。如果 tv 参数为 NULL，event_base 会立即停止循环，没有延时。如果 event_base 当前正在执行任何激活事件的回调，则回调会继续运行，直到**运行完所有激活事件的回调**之才退出。

event_base_loopbreak （） 让 event_base 立即退出循环。 它与event_base_loopexit（base,NULL）的不同在于，如果 event_base 当前正在执行激活事件的回调，它将在**执行完当前正在处理的事件后立即退出。**

注意 event_base_loopexit(base,NULL)和 event_base_loopbreak(base)在事件循环没有运行时的行为不同：前者安排下一次事件循环在下一轮回调完成后立即停止（就好像 带EVLOOP_ONCE 标志调用一样）；后者却仅仅停止当前正在运行的循环，如果事件循环没有运行，则没有任何效果。

这两个函数都在成功时返回0，失败时返回-1。

示例：立即关闭

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

/* Here's a callback function that calls loopbreak */
void cb(int sock, short what, void *arg)
{
    struct event_base *base = arg;
    event_base_loopbreak(base);
}

void main_loop(struct event_base *base, evutil_socket_t watchdog_fd)
{
    struct event *watchdog_event;

    /* Construct a new event to trigger whenever there are any bytes to
       read from a watchdog socket.  When that happens, we'll call the
       cb function, which will make the loop exit immediately without
       running any other active events at all.
     */
    watchdog_event = event_new(base, watchdog_fd, EV_READ, cb, base);

    event_add(watchdog_event, NULL);

    event_base_dispatch(base);
}
```

示例：执行事件循环10秒，然后退出

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

void run_base_with_ticks(struct event_base *base)
{
  struct timeval ten_sec;

  ten_sec.tv_sec = 10;
  ten_sec.tv_usec = 0;

  /* Now we run the event_base for a series of 10-second intervals, printing
     "Tick" after each.  For a much better way to implement a 10-second
     timer, see the section below about persistent timer events. */
  while (1) {
     /* This schedules an exit ten seconds from now. */
     event_base_loopexit(base, &ten_sec);

     event_base_dispatch(base);
     puts("Tick");
  }
}
```

有时候需要知道对 event_base_dispatch（）或者 event_base_loop（）的调用是正常退出的，还是因为调用 event_base_loopexit（）或者 event_base_break（）而退出的。可以调用下述函数来确定是否调用了 loopexit 或者 break 函数。

接口

```
int event_base_got_exit(struct event_base *base);
int event_base_got_break(struct event_base *base);
```

这两个函数分别会在因为调用 event_base_loopexit（）或者 event_base_break（）而退出循环的时候返回 true，否则返回 false。下次启动事件循环的时候，这些值会被重设。

这些函数声明在<event2/event.h>中。event_break_loopexit()函数首次在 libevent 1.0c 版本中实现；event_break_loopbreak()首次在 libevent 1.4.3版本中实现。

3. 检查内部时间缓存

有时候需要在事件回调中获取当前时间的近似视图，但不想调用 gettimeofday()（可能是因为 OS 将 gettimeofday()作为系统调用实现，而你试图避免系统调用的开销）。

在回调中，可以请求 libevent 开始本轮回调时的当前时间视图。

接口

```
int event_base_gettimeofday_cached(struct event_base *base,
    struct timeval *tv_out);
```

如果当前正在执行回调，event_base_gettimeofday_cached()函数将 tv_out 参数的值置为缓存的时间。否则，函数调用 evutil_gettimeofday()获取真正的当前时间。成功时函数返回0，失败时返回负数。

注意，因为 libevent 在开始执行回调的时候缓存时间值，所以这个值至少是有一点不精确的。如果回调执行很长时间，这个值将非常不精确。

要强制立即更新缓存，可以调用以下函数：

接口

```
int event_base_update_cache_time(struct event_base *base);
```

如果成功，则返回0，失败则返回-1，如果event base未运行其事件循环，则无效。

event_base_gettimeofday_cached（）函数是Libevent 2.0.4-alpha中的新增功能。 Libevent 2.1.1-alpha添加了event_base_update_cache_time（）。

4. 转储 event_base 的状态

接口

```
void event_base_dump_events(struct event_base *base, FILE *f);
```

为帮助调试程序（或者调试 libevent），有时候可能需要加入到 event_base 的事件及其状态的完整列表。调用 event_base_dump_events()可以将这个列表输出到指定的文件中。

这个列表是人可读的，未来版本的 libevent 将会改变其格式。

这个函数在 libevent 2.0.1-alpha 版本中引入。

四、创建event

libevent 的基本操作单元是事件。每个事件代表一组条件的集合，这些条件包括：

- 文件描述符已经就绪，可以读取或者写入
- 文件描述符变为就绪状态，可以读取或者写入（仅对于边沿触发 IO）
- 超时事件
- 发生某信号
- 用户触发事件

所有事件具有相似的生命周期。调用 libevent 函数设置事件并且关联到 event_base 之后，事件进入“**已初始化（initialized）**”状态。此时可以将事件添加到 event_base 中，这使之进入“**未决（pending）**”状态。在未决状下，如果触发事件的条件发生（比如说，文件描述符的状态改变，或者超时时间到达），则事件进入“**激活（active）**”状态，（用户提供的）事件回调函数将被执行。如果配置为“**持久的（persistent）**”，事件将保持为未决状态。否则，执行完回调后，事件不再是未决的。删除操作可以让未决事件成为**非未决（已初始化）**的；添加操作可以让非未决事件再次成为未决的。

1. 构造事件对象

1.1 创建事件

使用 event_new（）接口创建事件。

接口

```
#define EV_TIMEOUT      0x01
#define EV_READ         0x02
#define EV_WRITE        0x04
#define EV_SIGNAL       0x08
#define EV_PERSIST      0x10
#define EV_ET           0x20

typedef void (*event_callback_fn)(evutil_socket_t, short, void *);

struct event *event_new(struct event_base *base, evutil_socket_t fd,
```

```
    short what, event_callback_fn cb,
    void *arg);

void event_free(struct event *event);
```

event_new()试图分配和构造一个用于 base 的新的事件。what 参数是上述标志的集合。如果 fd 非负，则它是将被观察其读写事件的文件。事件被激活时，libevent 将调用 cb 函数，传递这些参数：

- 文件描述符 fd，表示所有被触发事件的位字段，
- 以及构造事件时的 arg 参数。

发生内部错误，或者传入无效参数时，event_new（）将返回 NULL。

所有新创建的事件都处于已初始化和非未决状态，调用 event_add（）可以使其成为未决的。

要释放事件，调用 event_free（）。对未决或者激活状态的事件调用 event_free（）是安全的：在释放事件之前，函数将会使事件成为非激活和非未决的。

示例

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

void cb_func(evutil_socket_t fd, short what, void *arg)
{
        const char *data = arg;
        printf("Got an event on socket %d:%s%s%s%s [%s]",
            (int) fd,
            (what&EV_TIMEOUT) ? " timeout" : "",
            (what&EV_READ)    ? " read" : "",
            (what&EV_WRITE)   ? " write" : "",
            (what&EV_SIGNAL)  ? " signal" : "",
            data);
}

void main_loop(evutil_socket_t fd1, evutil_socket_t fd2)
{
        struct event *ev1, *ev2;
        struct timeval five_seconds = {5,0};
        struct event_base *base = event_base_new();

        /* The caller has already set up fd1, fd2 somehow, and make them
           nonblocking. */

        ev1 = event_new(base, fd1, EV_TIMEOUT|EV_READ|EV_PERSIST, cb_func,
           (char*)"Reading event");
        ev2 = event_new(base, fd2, EV_WRITE|EV_PERSIST, cb_func,
           (char*)"Writing event");

        event_add(ev1, &five_seconds);
        event_add(ev2, NULL);
        event_base_dispatch(base);
}
```

上述函数定义在<event2/event.h>中，首次出现在libevent 2.0.1-alpha版本中。event_callback_fn 类型首次在2.0.4-alpha 版本中作为 typedef 出现。

1.2 事件标志

- EV_TIMEOUT

这个标志表示某超时时间流逝后事件成为激活的。构造事件的时候，EV_TIMEOUT 标志是被忽略的：可以在添加事件的时候设置超时，也可以不设置。超时发生时，回调函数的 what参数将带有这个标志。

- EV_READ

表示指定的文件描述符已经就绪，可以读取的时候，事件将成为激活的。

- EV_WRITE

表示指定的文件描述符已经就绪，可以写入的时候，事件将成为激活的。

- EV_SIGNAL

用于实现信号检测，请看下面的“构造信号事件”章节。

- EV_PERSIST

表示事件是“持久的”，请看下面的“关于事件持久性”章节。

- EV_ET

表示如果底层的 event_base 后端支持边沿触发事件，则事件应该是边沿触发的。这个标志影响 EV_READ 和 EV_WRITE 的语义。

从2.0.1-alpha 版本开始，可以有任意多个事件因为同样的条件而未决。比如说，可以有两 个事件因为某个给定的 fd 已经就绪，可以读取而成为激活的。这种情况下，多个事件回调 被执行的次序是不确定的。

这些标志定义在<event2/event.h>中。除了 EV_ET 在2.0.1-alpha 版本中引入外，所有标志从1.0版本开始就存在了。

1.3 关于事件持久性

默认情况下，每当未决事件成为激活的（因为 fd 已经准备好读取或者写入，或者因为超时），事件将在其回调被执行前成为非未决的。如果想让事件再次成为未决的，可以在回调函数中再次对其调用 event_add（）。
然而，如果设置了 EV_PERSIST 标志，事件就是持久的。这意味着即使其回调被激活，事件还是会保持为未决状态。如果想在回调中让事件成为非未决的，可以对其调用 event_del（）。
每次执行事件回调的时候 ，持久事件的超时值会被复位 。因此如果具有EV_READ|EV_PERSIST 标志，以及5秒的超时值，则事件将在以下情况下成为激活的：

- 套接字已经准备好被读取的时候
- 从最后一次成为激活的开始，已经逝去5秒

1.4 创建事件作为其的回调参数

通常，可能要创建一个将自身作为回调参数接收的事件。 但是，您不能仅将指向事件的指针作为 event_new（）的参数传递，因为它尚不存在。 要解决此问题，可以使用event_self_cbarg（）。

接口

```
void *event_self_cbarg();
```

event_self_cbarg（）函数返回一个“魔术”指针，该指针在作为事件回调参数传递时，告诉 event_new（）创建一个将自身作为其回调参数接收的事件。

示例

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

static int n_calls = 0;

void cb_func(evutil_socket_t fd, short what, void *arg)
{
    struct event *me = arg;

    printf("cb_func called %d times so far.\n", ++n_calls);

    if (n_calls > 100)
       event_del(me);
}

void run(struct event_base *base)
{
    struct timeval one_sec = { 1, 0 };
    struct event *ev;
    /* We're going to set up a repeating timer to get called called 100
       times. */
    ev = event_new(base, -1, EV_PERSIST, cb_func, event_self_cbarg());
    event_add(ev, &one_sec);
    event_base_dispatch(base);
}
```

此函数也可以与event_new（），evtimer_new（），evsignal_new（），event_assign（），evtimer_assign（）和evsignal_assign（）一起使用。 但是，**它不能用作非事件的回调参数。**

在libbevent 2.1.1-alpha中引入了event_self_cbarg（）函数。

1.5 只有超时的事件

为使用方便，libevent 提供了一些以 evtimer*开头的宏，用于替代 event*调用来操作纯超时事件。使用这些宏能改进代码的清晰性。

接口

```
#define evtimer_new(base, callback, arg) \
    event_new((base), -1, 0, (callback), (arg))
#define evtimer_add(ev, tv) \
    event_add((ev),(tv))
#define evtimer_del(ev) \
    event_del(ev)
#define evtimer_pending(ev, tv_out) \
    event_pending((ev), EV_TIMEOUT, (tv_out))
```

除了 evtimer_new（）首次出现在2.0.1-alpha 版本中之外，这些宏从0.6版本就存在了。

1.6 构造信号事件

libevent 也可以监测 POSIX 风格的信号。要构造信号处理器，使用：

接口

```
#define evsignal_new(base, signum, cb, arg) \
    event_new(base, signum, EV_SIGNAL|EV_PERSIST, cb, arg)
```

除了提供一个信号编号代替文件描述符之外，各个参数与 event_new（）相同。

示例

```
struct event *hup_event;
struct event_base *base = event_base_new();

/* call sighup_function on a HUP signal */
hup_event = evsignal_new(base, SIGHUP, sighup_function, NULL);
```

注意：信号回调是信号发生后在事件循环中被执行的，所以可以安全地调用通常不能在POSIX 风格信号处理器中使用的函数。

> 警告：不要在信号事件上设置超时，这可能是不被支持的。[待修正：真是这样的吗？]

libevent 也提供了一组方便使用的宏用于处理信号事件：

接口

```
#define evsignal_add(ev, tv) \
    event_add((ev),(tv))
#define evsignal_del(ev) \
    event_del(ev)
#define evsignal_pending(ev, what, tv_out) \
    event_pending((ev), (what), (tv_out))
```

evsignal_*宏从2.0.1-alpha 版本开始存在。先前版本中这些宏叫做 signal_add（）、signal_del （）等等。

关于信号的警告
在当前版本的 libevent 和大多数后端中，每个进程任何时刻只能有一个 event_base 可以监听信号。如果同时向两个 event_base 添加信号事件，即使是不同的信号，也只有一个event_base 可以取得信号。kqueue 后端没有这个限制。

1.7 设置不使用堆分配的事件

出于性能考虑或者其他原因，有时需要将事件作为一个大结构体的一部分。对于每个事件的

使用，这可以节省：

- 内存分配器在堆上分配小对象的开销
- 对 event 结构体指针取值的时间开销
- 如果事件不在缓存中，因为可能的额外缓存丢失而导致的时间开销

使用此方法可能会破坏与其他版本的Libevent的二进制兼容性，这些版本的事件结构可能具有不同的大小。

这是非常小的成本，对于大多数应用来说都没有关系。 除非知道使用堆分配事件会严重降低性能，否则应该坚持使用event_new（）。 如果将来的Libevent版本使用的事件结构比构建时使用的事件结构大，则使用event_assign（）可能会导致难以诊断的错误。

接口

```
int event_assign(struct event *event, struct event_base *base,
    evutil_socket_t fd, short what,
    void (*callback)(evutil_socket_t, short, void *), void *arg);
```

除了 event 参数必须指向一个未初始化的事件之外，event_assign（）的参数与 event_new （）的参数相同。成功时函数返回0，如果发生内部错误或者使用错误的参数，函数返回-1。

示例

```
#include <event2/event.h>
/* Watch out!  Including event_struct.h means that your code will not
 * be binary-compatible with future versions of Libevent. */
#include <event2/event_struct.h>
#include <stdlib.h>

struct event_pair {
         evutil_socket_t fd;
         struct event read_event;
         struct event write_event;
};
void readcb(evutil_socket_t, short, void *);
void writecb(evutil_socket_t, short, void *);
struct event_pair *event_pair_new(struct event_base *base, evutil_socket_t fd)
{
        struct event_pair *p = malloc(sizeof(struct event_pair));
        if (!p) return NULL;
        p->fd = fd;
        event_assign(&p->read_event, base, fd, EV_READ|EV_PERSIST, readcb, p);
        event_assign(&p->write_event, base, fd, EV_WRITE|EV_PERSIST, writecb,
p);
        return p;
}
```

也可以用 event_assign（）初始化栈上分配的，或者静态分配的事件。

警告

不要对已经在 event_base 中未决的事件调用 event_assign（），这可能会导致难以诊断的错误。如果已经初始化和成为未决的，调用 event_assign（）之前需要调用 event_del（）。libevent 提供了方便的宏将event_assign（）用于仅超时事件或者信号事件。

接口

```
#define evtimer_assign(event, base, callback, arg) \
    event_assign(event, base, -1, 0, callback, arg)
#define evsignal_assign(event, base, signum, callback, arg) \
    event_assign(event, base, signum, EV_SIGNAL|EV_PERSIST, callback, arg)
```

如果需要使用 event_assign（），又要保持与将来版本 libevent 的二进制兼容性，可以请求libevent 告知 struct event 在运行时应该有多大：

接口

```
size_t event_get_struct_event_size(void);
```

这个函数返回需要为 event 结构体保留的字节数。再次强调，只有在确信堆分配是一个严重的性能问题时才应该使用这个函数，因为这个函数让代码难以阅读和编写。

注意，将来版本的 event_get_struct_event_size()的返回值可能比 sizeof(struct event)小，这表示 event 结构体末尾的额外字节仅仅是保留用于将来版本 libevent 的填充字节。

下面这个例子跟上面的那个相同，但是不依赖于 event_struct.h 中的 event 结构体的大小，而是使用 event_get_struct_size（）来获取运行时的正确大小。

示例

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

/* When we allocate an event_pair in memory, we'll actually allocate
 * more space at the end of the structure.  We define some macros
 * to make accessing those events less error-prone. */
struct event_pair {
        evutil_socket_t fd;
};

/* Macro: yield the struct event 'offset' bytes from the start of 'p' */
#define EVENT_AT_OFFSET(p, offset) \
            ((struct event*) ( ((char*)(p)) + (offset) ))
/* Macro: yield the read event of an event_pair */
#define READEV_PTR(pair) \
            EVENT_AT_OFFSET((pair), sizeof(struct event_pair))
/* Macro: yield the write event of an event_pair */
#define WRITEEV_PTR(pair) \
            EVENT_AT_OFFSET((pair), \
                sizeof(struct event_pair)+event_get_struct_event_size())

/* Macro: yield the actual size to allocate for an event_pair */
#define EVENT_PAIR_SIZE() \
            (sizeof(struct event_pair)+2*event_get_struct_event_size())

void readcb(evutil_socket_t, short, void *);
void writecb(evutil_socket_t, short, void *);
struct event_pair *event_pair_new(struct event_base *base, evutil_socket_t fd)
{
        struct event_pair *p = malloc(EVENT_PAIR_SIZE());
        if (!p) return NULL;
        p->fd = fd;
        event_assign(READEV_PTR(p), base, fd, EV_READ|EV_PERSIST, readcb, p);
        event_assign(WRITEEV_PTR(p), base, fd, EV_WRITE|EV_PERSIST, writecb, p);
        return p;
}
```

event_assign（）定义在<event2/event.h>中，从2.0.1-alpha 版本开始就存在了。从2.0.3-alpha 版本开始，函数返回 int，在这之前函数返回 void。event_get_struct_event_size（）在2.0.4-alpha 版本中引入。event 结构体定义在<event2/event_struct.h>中。

2. 让事件未决和非未决

构造事件之后，在将其添加到 event_base 之前实际上是不能对其做任何操作的。使用event_add（）将事件添加到 event_base。

接口

```
int event_add(struct event *ev, const struct timeval *tv);
```

在非未决的事件上调用 event_add（）将使其在配置的 event_base 中成为未决的。成功时函数返回0，失败时返回-1。如果 tv 为 NULL，添加的事件不会超时。否则，tv 以秒和微秒指定超时值。

如果对已经未决的事件调用 event_add（），事件将保持未决状态，并在指定的超时时间被重新调度。

> 注意：不要设置tv为希望超时事件执行的时间。如果在2010年1月1日设置“tv->tv_sec=time(NULL)+10;”，超时事件将会等待40年，而不是10秒。

接口

```
int event_remove_timer(struct event *ev);
```

对已经初始化的事件调用 event_del（）将使其成为非未决和非激活的。如果事件不是未决 的或者激活的，调用将没有效果。成功时函数返回0，失败时返回-1。

注意：如果在事件激活后，其回调被执行前删除事件，回调将不会执行。

这些函数定义在<event2/event.h>中，从0.1版本就存在了。

3. 带优先级的事件

多个事件同时触发时，libevent 没有定义各个回调的执行次序。可以使用优先级来定义某些事件比其他事件更重要。
在前一章讨论过，每个 event_base 有与之相关的一个或者多个优先级。在初始化事件之后，但是在添加到 event_base 之前，可以为其设置优先级。
接口

```
int event_priority_set(struct event *event, int priority);
```

事件的优先级是一个在0和 event_base 的优先级减去1之间的数值。成功时函数返回0，失败时返回-1。
多个不同优先级的事件同时成为激活的时候，低优先级的事件不会运行。libevent 会执行高优先级的事件，然后重新检查各个事件。只有在没有高优先级的事件是激活的时候，低优先级的事件才会运行。

示例

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

void read_cb(evutil_socket_t, short, void *);
void write_cb(evutil_socket_t, short, void *);

void main_loop(evutil_socket_t fd)
{
  struct event *important, *unimportant;
  struct event_base *base;

  base = event_base_new();
  event_base_priority_init(base, 2);
  /* Now base has priority 0, and priority 1 */
  important = event_new(base, fd, EV_WRITE|EV_PERSIST, write_cb, NULL);
  unimportant = event_new(base, fd, EV_READ|EV_PERSIST, read_cb, NULL);
  event_priority_set(important, 0);
  event_priority_set(unimportant, 1);
```

```
  /* Now, whenever the fd is ready for writing, the write callback will
     happen before the read callback.  The read callback won't happen at
     all until the write callback is no longer active. */
}
```

如果不为事件设置优先级，则默认的优先级将会是 event_base 的优先级数目除以2。
这个函数声明在<event2/event.h>中，从1.0版本就存在了。

4. 检查事件状态

有时候需要了解事件是否已经添加，检查事件代表什么。
接口

```
int event_pending(const struct event *ev, short what, struct timeval *tv_out);

#define event_get_signal(ev) /* ... */
evutil_socket_t event_get_fd(const struct event *ev);
struct event_base *event_get_base(const struct event *ev);
short event_get_events(const struct event *ev);
event_callback_fn event_get_callback(const struct event *ev);
void *event_get_callback_arg(const struct event *ev);
int event_get_priority(const struct event *ev);

void 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,
        void **arg_out);
```

event_pending（）函数确定给定的事件是否是未决的或者激活的。如果是，而且 what 参数设置了 EV_READ、EV_WRITE、EV_SIGNAL 或者 EV_TIMEOUT 等标志，则函数会返回事件当前为之未决或者激活的所有标志。如果提供了 tv_out 参数，并且 what 参数中设置了 EV_TIMEOUT 标志，而事件当前正因超时事件而未决或者激活，则 tv_out 会返回事件的超时值。

event_get_fd（）和 event_get_signal（）返回为事件配置的文件描述符或者信号值。

event_get_base（）返回为事件配置的 event_base。

event_get_events（）返回事件的标志（EV_READ、EV_WRITE 等）。

event_get_callback（）和 event_get_callback_arg（）返回事件的回调函数及其参数指针。

event_get_assignment（）复制所有为事件分配的字段到提供的指针中。任何为 NULL 的参数会被忽略。

示例

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

/* Change the callback and callback_arg of 'ev', which must not be
 * pending. */
int replace_callback(struct event *ev, event_callback_fn new_callback,
    void *new_callback_arg)
{
    struct event_base *base;
    evutil_socket_t fd;
```

```
    short events;

    int pending;

    pending = event_pending(ev, EV_READ|EV_WRITE|EV_SIGNAL|EV_TIMEOUT,
                            NULL);
    if (pending) {
        /* We want to catch this here so that we do not re-assign a
         * pending event.  That would be very very bad. */
        fprintf(stderr,
                "Error! replace_callback called on a pending event!\n");
        return -1;
    }

    event_get_assignment(ev, &base, &fd, &events,
                         NULL /* ignore old callback */ ,
                         NULL /* ignore old callback argument */);

    event_assign(ev, base, fd, events, new_callback, new_callback_arg);
    return 0;
}
```

这些函数声明在<event2/event.h>中。event_pending（）函数从0.1版就存在了。2.0.1-alpha 版引入了event_get_fd（）和 event_get_signal（）。2.0.2-alpha 引入了 event_get_base（）。其他的函数在2.0.4-alpha 版中引入。

5. 查找当前正在运行的事件

为了调试或其他目的，您可以获取当前运行事件的指针。

接口

```
struct event *event_base_get_running_event(struct event_base *base);
```

请注意，仅在从提供的event_base循环中调用此函数时，才定义该函数的行为。 不支持从另一个线程调用它，这可能导致未定义的行为。

此函数在<event2 / event.h>中声明。 它是在Libevent 2.1.1-alpha中引入的。

6. 配置一次触发事件

如果不需要多次添加一个事件，或者要在添加后立即删除事件，而事件又不需要是持久的，则可以使用event_base_once（）。

接口

```
int event_base_once(struct event_base *, evutil_socket_t, short,
  void (*)(evutil_socket_t, short, void *), void *, const struct timeval *);
```

除了不支持 EV_SIGNAL 或者 EV_PERSIST 之外，这个函数的接口与 event_new（）相同。安排的事件将以默认的优先级加入到 event_base 并执行。回调被执行后，libevent 内部将会释放 event 结构。成功时函数返回0，失败时返回-1。

不能删除或者手动激活使用 event_base_once（）插入的事件：如果希望能够取消事件，应该使用 event_new（）或者 event_assign（）。

另请注意，在Libevent 2.0之前的版本中，如果从未触发该事件，则将永远不会释放用于保存该事件的内部存储器。 从Libevent 2.1.2-alpha开始，释放event_base时将释放这些事件，即使它们尚未激活，但仍要注意：如果有一些与其回调参数相关联的存储，则除非释放该存储，否则除非 您的程序做了一些跟踪和发布的操作。

7. 手动激活事件

极少数情况下，需要在事件的条件没有触发的时候让事件成为激活的。

接口

```
void event_active(struct event *ev, int what, short ncalls);
```

此功能使事件ev带有标志what（EV_READ，EV_WRITE和EV_TIMEOUT的组合）变为活动状态。 事件不需要已经处于未决状态，激活事件也不会让它成为未决的。

警告：在同一事件上递归调用event_active（）可能会导致资源耗尽。 以下代码段是如何**错误**使用event_active的示例。

错误的例子：使用event_active（）进行无限循环

```
struct event *ev;

static void cb(int sock, short which, void *arg) {
        /* Whoops: Calling event_active on the same event unconditionally
           from within its callback means that no other events might not get
           run! */

        event_active(ev, EV_WRITE, 0);
}

int main(int argc, char **argv) {
        struct event_base *base = event_base_new();

        ev = event_new(base, -1, EV_PERSIST | EV_READ, cb, NULL);

        event_add(ev, NULL);

        event_active(ev, EV_WRITE, 0);

        event_base_loop(base, 0);

        return 0;
}
```

这将导致事件循环仅执行一次并永远调用函数“ cb”的情况。

示例：使用计时器解决上述问题的替代方法

```
struct event *ev;
struct timeval tv;

static void cb(int sock, short which, void *arg) {
   if (!evtimer_pending(ev, NULL)) {
       event_del(ev);
       evtimer_add(ev, &tv);
   }
```

```
}

int main(int argc, char **argv) {
   struct event_base *base = event_base_new();

   tv.tv_sec = 0;
   tv.tv_usec = 0;

   ev = evtimer_new(base, cb, NULL);

   evtimer_add(ev, &tv);

   event_base_loop(base, 0);

   return 0;
}
```

示例：使用event_config_set_max_dispatch_interval（）解决上述问题的替代解决方案

```
struct event *ev;

static void cb(int sock, short which, void *arg) {
        event_active(ev, EV_WRITE, 0);
}

int main(int argc, char **argv) {
        struct event_config *cfg = event_config_new();
        /* Run at most 16 callbacks before checking for other events. */
        event_config_set_max_dispatch_interval(cfg, NULL, 16, 0);
        struct event_base *base = event_base_new_with_config(cfg);
        ev = event_new(base, -1, EV_PERSIST | EV_READ, cb, NULL);

        event_add(ev, NULL);

        event_active(ev, EV_WRITE, 0);

        event_base_loop(base, 0);

        return 0;
}
```

这个函数定义在<event2/event.h>中，从0.3版本就存在了。

8. 优化公用超时

当前版本的 libevent 使用二进制堆算法跟踪未决事件的超时值，这让添加和删除事件超时值具有O(logN)性能。对于随机分布的超时值集合，这是优化的，但对于大量具有相同超时值的事件集合，则不是。

比如说，假定有10000个事件，每个都需要在添加后5秒触发超时事件。这种情况下，使用双链队列实现才可以取得 O（1）性能。

自然地，不希望为所有超时值使用队列，因为队列仅对常量超时值更快。如果超时值或多或少地随机分布，则向队列添加超时值的性能将是 O（n），这显然比使用二进制堆糟糕得多。

libevent 通过放置一些超时值到队列中，另一些到二进制堆中来解决这个问题。要使用这个机制，需要向 libevent 请求一个“**公用超时(common timeout)**”值，然后使用它来添加事件。如果有大量具有单个公用超时值的事件，使用这个优化应该可以改进超时处理性能。

接口

```
const struct timeval *event_base_init_common_timeout(
    struct event_base *base, const struct timeval *duration);
```

这个函数需要 event_base 和要初始化的公用超时值作为参数。函数返回一个到特别的timeval 结构体的指针，可以使用这个指针指示事件应该被添加到 O（1）队列，而不是 O （logN）堆。可以在代码中自由地复制这个特的 timeval 或者进行赋值，但它仅对用于构造它的特定 event_base 有效。不能依赖于其实际内容：libevent 使用这个内容来告知自身使用哪个队列。

示例

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

/* We're going to create a very large number of events on a given base,
 * nearly all of which have a ten-second timeout.  If initialize_timeout
 * is called, we'll tell Libevent to add the ten-second ones to an O(1)
 * queue. */
struct timeval ten_seconds = { 10, 0 };

void initialize_timeout(struct event_base *base)
{
    struct timeval tv_in = { 10, 0 };
    const struct timeval *tv_out;
    tv_out = event_base_init_common_timeout(base, &tv_in);
    memcpy(&ten_seconds, tv_out, sizeof(struct timeval));
}

int my_event_add(struct event *ev, const struct timeval *tv)
{
    /* Note that ev must have the same event_base that we passed to
       initialize_timeout */
    if (tv && tv->tv_sec == 10 && tv->tv_usec == 0)
        return event_add(ev, &ten_seconds);
    else
        return event_add(ev, tv);
}
```

与所有优化函数一样，除非确信适合使用，应该避免使用公用超时功能。

这个函数由2.0.4-alpha 版本引入。

9.从已清除的内存识别事件

libevent 提供了函数，可以从已经通过设置为0（比如说，通过 calloc （）分配的，或者使用 memset （）或者 bzero （）清除了的）而清除的内存识别出已初始化的事件。

接口

```
int event_initialized(const struct event *ev);

#define evsignal_initialized(ev) event_initialized(ev)
#define evtimer_initialized(ev) event_initialized(ev)
```

警告

这个函数不能可靠地从没有初始化的内存块中识别出已经初始化的事件。除非知道被查询的 内存要么是已清除的，要么是已经初始化为事件的，才能使用这个函数。

除非编写一个非常特别的应用，通常不需要使用这个函数。event_new（）返回的事件总是 已经初始化的。

示例

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

struct reader {
    evutil_socket_t fd;
};

#define READER_ACTUAL_SIZE() \
    (sizeof(struct reader) + \
     event_get_struct_event_size())

#define READER_EVENT_PTR(r) \
    ((struct event *) (((char*)(r))+sizeof(struct reader)))

struct reader *allocate_reader(evutil_socket_t fd)
{
    struct reader *r = calloc(1, READER_ACTUAL_SIZE());
    if (r)
        r->fd = fd;
    return r;
}

void readcb(evutil_socket_t, short, void *);
int add_reader(struct reader *r, struct event_base *b)
{
    struct event *ev = READER_EVENT_PTR(r);
    if (!event_initialized(ev))
        event_assign(ev, b, r->fd, EV_READ, readcb, r);
    return event_add(ev, NULL);
}
```

从Libevent 0.3开始，已经提供了event_initialized（）函数。

五、辅助类型和函数

<event2/util.h>定义了很多在实现可移植应用时有用的函数，libevent 内部也使用这些类型和函数。

1. 基本类型

1.1 evutil_socket_t

在除 Windows 之外的大多数地方，套接字是个整数，操作系统按照数值次序进行处理。然而，使用 Windows 套接字 API 时，socket 具有类型 SOCKET，它实际上是个类似指针的句柄，收到这个句柄的次序是未定义的。在 Windows 中，libevent 定义 evutil_socket_t 类型为整型指针，可以处理 socket() 或者 accept()的输出，而没有指针截断的风险。

定义

```
#ifdef WIN32
#define evutil_socket_t intptr_t
#else
#define evutil_socket_t int
#endif
```

这个类型在2.0.1-alpha 版本中引入。

1.2 标准整数类型

落后于21世纪的 C 系统常常没有实现 C99标准规定的 stdint.h 头文件。考虑到这种情况，libevent 定义了来自于 stdint.h 的、位宽度确定（bit-width-specific）的整数类型：

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

跟 C99标准一样，这些类型都有明确的位宽度。

这些类型由1.4.0-alpha 版本引入。MAX/MIN 常量首次出现在2.0.4-alpha 版本。

1.3 各种兼容性类型

在有 ssize_t（有符号的 size_t）类型的平台上，ev_ssize_t 定义为 ssize_t；而在没有的平台上，则定义为某合理的默认类型。ev_ssize_t 类型的最大可能值是 EV_SSIZE_MAX；最小可能值是 EV_SSIZE_MIN。（在平台没有定义 SIZE_MAX 的时候，size_t 类型的最大可能值是 EV_SIZE_MAX）

ev_off_t 用于代表文件或者内存块中的偏移量。在有合理 off_t 类型定义的平台，它被定义为 off_t；在 Windows 上则定义为 ev_int64_t。

某些套接字 API 定义了 socklen_t 长度类型，有些则没有定义。在有这个类型定义的平台中，ev_socklen_t 定义为 socklen_t，在没有的平台上则定义为合理的默认类型。

ev_intptr_t 是一个有符号整数类型，足够容纳指针类型而不会产生截断；而 ev_uintptr_t 则是相应的无符号类型。

ev_ssize_t 类型由2.0.2-alpha 版本加入。ev_socklen_t 类型由2.0.3-alpha 版本加入。ev_intptr_t 与 ev_uintptr_t 类型，以及 EV_SSIZE_MAX/MIN 宏定义由2.0.4-alpha 版本加入。ev_off_t 类型首次出现在2.0.9-rc 版本。

2. 定时器可移植函数

不是每个平台都定义了标准 timeval 操作函数，所以 libevent 也提供了自己的实现。

接口

```
#define evutil_timeradd(tvp, uvp, vvp) /* ... */
#define evutil_timersub(tvp, uvp, vvp) /* ... */
```

这些宏分别对前两个参数进行加或者减运算，将结果存放到第三个参数中。

接口

```
#define evutil_timerclear(tvp) /* ... */
#define evutil_timerisset(tvp) /* ... */
```

清除 timeval 会将其值设置为0。evutil_timerisset 宏检查 timeval 是否已经设置，如果已经设置为非零值，返回 ture，否则返回 false。

接口

```
#define evutil_timercmp(tvp, uvp, cmp)
```

evutil_timercmp 宏比较两个 timeval，如果其关系满足 cmp 关系运算符，返回 true。比如说，evutil_timercmp(t1,t2,<=)的意思是"是否 t1<=t2？"。注意：与某些操作系统版本不同的是，libevent 的时间比较支持所有 C 关系运算符（也就是<、>、==、!=、<=和>=）。

接口

```
int evutil_gettimeofday(struct timeval *tv, struct timezone *tz);
```

evutil_gettimeofdy（）函数设置 tv 为当前时间，tz 参数未使用。

示例

```
struct timeval tv1, tv2, tv3;

/* Set tv1 = 5.5 seconds */
tv1.tv_sec = 5; tv1.tv_usec = 500*1000;

/* Set tv2 = now */
evutil_gettimeofday(&tv2, NULL);

/* Set tv3 = 5.5 seconds in the future */
evutil_timeradd(&tv1, &tv2, &tv3);

/* all 3 should print true */
if (evutil_timercmp(&tv1, &tv1, ==))  /* == "If tv1 == tv1" */
   puts("5.5 sec == 5.5 sec");
if (evutil_timercmp(&tv3, &tv2, >=))  /* == "If tv3 >= tv2" */
   puts("The future is after the present.");
if (evutil_timercmp(&tv1, &tv2, <))   /* == "If tv1 < tv2" */
```

```
    puts("It is no longer the past.");
```

除 evutil_gettimeofday（）由2.0版本引入外，这些函数由1.4.0-beta 版本引入。

> 注意：在1.4.4之前的版本中使用<=或者>=是不安全的。

3. 套接字 API 兼容性

本节由于历史原因而存在：Windows 从来没有以良好兼容的方式实现 Berkeley 套接字 API。

接口

```
int evutil_closesocket(evutil_socket_t s);

#define EVUTIL_CLOSESOCKET(s) evutil_closesocket(s)
```

这个接口用于关闭套接字。在 Unix 中，它是 close()的别名；在 Windows 中，它调用closesocket()。（在 Windows 中不能将 close()用于套接字，也没有其他系统定义 了closesocket())

evutil_closesocket() 函数在 2.0.5-alpha 版本引入。在此之前，需要使用 EVUTIL_CLOSESOCKET 宏。

接口

```
#define EVUTIL_SOCKET_ERROR()
#define EVUTIL_SET_SOCKET_ERROR(errcode)
#define evutil_socket_geterror(sock)
#define evutil_socket_error_to_string(errcode)
```

这些宏访问和操作套接字错误代码。

EVUTIL_SOCKET_ERROR（）返回本线程最后一次套接字操作的全局错误号

evutil_socket_geterror（）则返回某特定套接字的错误号。（在类 Unix 系统中都是 errno）

EVUTIL_SET_SOCKET_ERROR()修改当前套接字错误号（与设置 Unix 中的 errno 类似）

evutil_socket_error_to_string（）返回代表某给定套接字错误号的字符串（与 Unix 中的 strerror()类似）。

（因为对于来自套接字函数的错误，Windows 不使用 errno，而是使用 WSAGetLastError()，所以需要这些函数。）

> 注意：Windows 套接字错误与从 errno 看到的标准 C 错误是不同的。

接口

```
int evutil_make_socket_nonblocking(evutil_socket_t sock);
```

用于对套接字进行非阻塞 IO 的调用也不能移植到 Windows 中。evutil_make_socket_nonblocking()函数要求一个套接字（来自 socket()或者 accept()）作为参数，将其设置为非阻塞的。（设置Unix中O_NONBLOCK标志和Windows中的FIONBIO标志）

接口

```
int evutil_make_listen_socket_reuseable(evutil_socket_t sock);
```

这个函数确保关闭监听套接字后，它使用的地址可以立即被另一个套接字使用 。（在 Unix中它设置 SO_REUSEADDR 标志，在 Windows 中则不做任何操作。不能在 Windows 中使用 SO_REUSEADDR 标志：它有另外不同的含义）（多个套接字绑定到相同地址）。

接口

```
int evutil_make_socket_closeonexec(evutil_socket_t sock);
```

这个函数告诉操作系统，如果调用了 exec()，应该关闭指定的套接字。在 Unix 中函数设置 FD_CLOEXEC 标志，在 Windows 上则没有操作。

接口

```
int evutil_socketpair(int family, int type, int protocol, evutil_socket_t
sv[2]);
```

这个函数的行为跟 Unix 的 socketpair（）调用相同：创建两个相互连接起来的套接字，可对其使用普通套接字 IO 调用。函数将两个套接字存储在 sv[0]和 sv[1]中，成功时返回0，失败时返回-1。

在 Windows 中，这个函数仅能支持 AF_INET 协议族、SOCK_STREAM 类型和0协议的套接字。注意：在防火墙软件明确阻止127.0.0.1，禁止主机与自身通话的情况下，函数可能失败。

除了 evutil_make_socket_closeonexec（）由2.0.4-alpha 版本引入外，这些函数都由1.4.0-alpha 版本引入。

4. 可移植的字符串操作函数

接口

```
ev_int64_t evutil_strtoll(const char *s, char **endptr, int base);
```

这个函数与 strtol 行为相同，只是用于64位整数。在某些平台上，仅支持十进制。

接口

```
int evutil_snprintf(char *buf, size_t buflen, const char *format, ...);
int evutil_vsnprintf(char *buf, size_t buflen, const char *format, va_list ap);
```

这些 snprintf 替代函数的行为与标准 snprintf 和 vsnprintf 接口相同。函数返回在缓冲区足够长的情况下将写入的字节数，不包括结尾的 NULL 字节。（这个行为遵循 C99的 snprintf()标准，但与 Windows 的*snprintf()相反：如果字符串无法放入缓冲区，*snprintf()会返回负数）

evutil_strtoll（）从1.4.2-rc 版本就存在了，其他函数首次出现在1.4.5版本中。

5. 区域无关的字符串操作函数

实现基于 ASCII 的协议时，可能想要根据字符类型的 ASCII 记号来操作字符串，而不管当前的区域设置。libevent 为此提供了一些函数：

接口

```
int evutil_ascii_strcasecmp(const char *str1, const char *str2);
int evutil_ascii_strncasecmp(const char *str1, const char *str2, size_t n);
```

这些函数与 strcasecmp()和 strncasecmp()的行为类似，只是它们总是使用 ASCII 字符集进行比较，而不管当前的区域设置。这两个函数首次在2.0.3-alpha 版本出现。

6. IPv6辅助和兼容性函数

接口

```
const char *evutil_inet_ntop(int af, const void *src, char *dst, size_t len);
int evutil_inet_pton(int af, const char *src, void *dst);
```

这些函数根据 RFC 3493的规定解析和格式化 IPv4与 IPv6地址，与标准 inet_ntop()和inet_pton()函数行为相同。

要格式化 IPv4地址，调用 evutil_inet_ntop()，设置 af 为 AF_INET，src 指向 in_addr 结构体，dst 指向大小为 len 的字符缓冲区。对于 IPv6地址，af 应该是AF_INET6，src 则指向 in6_addr 结构体。

要解析 IP 地址，调用 evutil_inet_pton()，设置af 为 AF_INET 或者 AF_INET6，src 指向要解析的字符串，dst 指向一个 in_addr 或者in_addr6结构体。

失败时 evutil_inet_ntop()返回 NULL，成功时返回到 dst 的指针。成功时 evutil_inet_pton()返回0，失败时返回-1。

接口

```
int evutil_parse_sockaddr_port(const char *str, struct sockaddr *out, int
*outlen);
```

这个接口解析来自 str 的地址，将结果写入到 out 中。outlen 参数应该指向一个表示 out 中 可用字节数的整数；函数返回时这个整数将表示实际使用了的字节数。成功时函数返回0， 失败时返回-1。

函数识别下列地址格式：

-
- ipv6（如 ffff::）
- [ipv6]（如[ffff::]）
- ipv4:端口号（如1.2.3.4:80）
- ipv4（如1.2.3.4）

如果没有给出端口号，结果中的端口号将被设置为0。

接口

```
int evutil_sockaddr_cmp(const struct sockaddr *sa1,
                        const struct sockaddr *sa2, int include_port);
```

evutil_sockaddr_cmp()函数比较两个地址，如果 sa1在 sa2前面，返回负数；如果二者相等，则返回0；如果 sa2在 sa1前面，则返回正数。函数可用于 AF_INET 和 AF_INET6地址；对于其他地址，返回值未定义。函数确保考虑地址的完整次序，但是不同版本中的次序可能不同。

如果 include_port 参数为 false，而两个地址只有端口号不同，则它们被认为是相等的。否则，具有不同端口号的地址被认为是不等的。

除 evutil_sockaddr_cmp()在2.0.3-alpha 版本引入外，这些函数在2.0.1-alpha 版本中引入。

7. 结构体可移植性函数

接口

```
#define evutil_offsetof(type, field) /* ... */
```

跟标准 offsetof 宏一样，这个宏返回从 type 类型开始处到 field 字段的字节数。

这个宏由2.0.1-alpha 版本引入，但2.0.3-alpha 版本之前是有 bug 的。

8. 安全随机数发生器

很多应用（包括 evdns）为了安全考虑需要很难预测的随机数。

接口

```
void evutil_secure_rng_get_bytes(void *buf, size_t n);
```

这个函数用随机数据填充 buf 处的 n 个字节。

如果所在平台提供了 arc4random()，libevent 会使用这个函数。否则，libevent 会使用自己的 arc4random()实现，种子则来自操作系统的熵池（entropy pool）（Windows 中的 CryptGenRandom，其他平台中的/dev/urandom）

接口

```
int evutil_secure_rng_init(void);
void evutil_secure_rng_add_bytes(const char *dat, size_t datlen);
```

不需要手动初始化安全随机数发生器，但是如果要确认已经成功初始化，可以调 用 evutil_secure_rng_init()。函数会播种 RNG（如果没有播种过），并在成功时返回0。函数返回-1则表示 libevent 无法在操作系统中找到合适的熵源（source of entropy），如果不自己初始化 RNG，就无法安全使用 RNG 了。

如果程序可能会放弃特权的环境中运行（例如，通过运行chroot（）），则应先调 evutil_secure_rng_init（）。

您可以自己调用evutil_secure_rng_add_bytes（）;将更多的随机字节添加到熵池中。 通常这不是必需的。

这些功能是Libevent 2.0.4-alpha中的新增功能。

六：bufferevent：概念和入门

很多时候，除了响应事件之外，应用还希望做一定的数据缓冲。比如说，写入数据的时候， 通常的运行模式是：

- 决定要向连接写入一些数据，把数据放入到缓冲区中
- 等待连接可以写入
- 写入尽量多的数据
- 记住写入了多少数据，如果还有更多数据要写入，等待连接再次可以写入

这种缓冲 IO 模式很通用，libevent 为此提供了一种通用机制，即 bufferevent。bufferevent 由一个底层的传输端口（如套接字），一个读取缓冲区和一个写入缓冲区组成。与通常的事件在底层传输端口已经就绪，可以读取或者写入的时候执行回调不同的是，bufferevent 在读取或者写入了足够量的数据之后调用用户提供的回调。

有多种共享公用接口的 bufferevent 类型，编写本文时已存在以下类型：

- 基于套接字的 bufferevent (socket-based bufferevents)：使用 event_*接口作为后端，通过底层流式套接字发送或者 接收数据的 bufferevent
- 异步 IO bufferevent (asynchronous-IO bufferevents)：使用 Windows IOCP 接口，通过底层流式套接字发送或者接收数 据的 bufferevent（仅用于 Windows，试验中）

- 过滤型 bufferevent (filtering bufferevents)：将数据传输到底层 bufferevent 对象之前，处理输入或者输出数据 的 bufferevent：比如说，为了压缩或者转换数据。
- 成对的 bufferevent (paired bufferevents)：相互传输数据的两个 bufferevent。

> **注意：**截止2.0.2-alpha版，这里列出的bufferevent接口还没有完全正交于所有的bufferevent 类型。也就是说，下面将要介绍的接口不是都能用于所有 bufferevent 类型。libevent 开发者在未来版本中将修正这个问题。

> 也请注意：当前 bufferevent 只能用于像 TCP 这样的面向流的协议，将来才可能会支持像UDP 这样的面向数据报的协议。

本节描述的所有函数和类型都在 event2/bufferevent.h 中声明。特别提及的关于 evbuffer 的函数声明在 event2/buffer.h 中，详细信息请参考下一章。

1. bufferevent 和 evbuffer

每个 bufferevent 都有一个输入缓冲区和一个输出缓冲区，它们的类型都是“struct evbuffer”。

有数据要写入到 bufferevent 时，添加数据到输出缓冲区；bufferevent 中有数据供读取的时候，从输入缓冲区抽取（drain）数据。

evbuffer 接口支持很多种操作，后面的章节将讨论这些操作。

2. 回调和水位

每个 bufferevent 有两个数据相关的回调：一个读取回调和一个写入回调。默认情况下，从底层传输端口读取了任意量的数据之后会调用读取回调；输出缓冲区中足够量的数据被清空到底层传输端口后写入回调会被调用。通过调整 bufferevent 的读取和写入 “水位（watermarks）”可以覆盖这些函数的默认行为。

每个 bufferevent 有四个水位：

- 读取低水位：读取操作使得输入缓冲区的数据量在此级别或者更高时，读取回调将被调用。默认值为0，所以每个读取操作都会导致读取回调被调用。
- 读取高水位：输入缓冲区中的数据量达到此级别后，bufferevent 将停止读取，直到输入缓冲区中足够量的数据被抽取，使得数据量低于此级别。默认值是无限，所以永远不会因为输入缓冲区的大小而停止读取。
- 写入低水位：写入操作使得输出缓冲区的数据量达到或者低于此级别时，写入回调将被调用。默认值是0，所以只有输出缓冲区空的时候才会调用写入回调。
- 写入高水位：bufferevent 没有直接使用这个水位。它在 bufferevent 用作另外一个bufferevent 的底层传输端口时有特殊意义。请看后面关于过滤型 bufferevent 的介绍。

bufferevent 也有“错误”或者“事件”回调，用于向应用通知非面向数据的事件，如连接已经关闭或者发生错误。定义了下列事件标志：

- BEV_EVENT_READING：读取操作时发生某事件，具体是哪种事件请看其他标志。
- BEV_EVENT_WRITING：写入操作时发生某事件，具体是哪种事件请看其他标志。
- BEV_EVENT_ERROR： 操作时发生错误 。关于错误的更多信息 ， EVUTIL_SOCKET_ERROR()。
- BEV_EVENT_TIMEOUT：发生超时。
- BEV_EVENT_EOF：遇到文件结束指示。
- BEV_EVENT_CONNECTED：请求的连接过程已经完成。

上述标志由2.0.2-alpha 版新引入。

3. 延迟回调

默认情况下，bufferevent 的回调在相应的条件发生时立即被执行。（evbuffer 的回调也是这样的，随后会介绍）在依赖关系复杂的情况下，这种立即调用会制造麻烦。比如说，假如某个回调在 evbuffer A 空的时候向其中移入数据，而另一个回调在 evbuffer A 满的时候从中取出数据。这些调用都是在栈上发生的，在依赖关系足够复杂的时候，有栈溢出的风险。

要解决此问题，可以请求 bufferevent（或者 evbuffer）延迟其回调。条件满足时，延迟回调不会立即调用，而是在 event_loop（）调用中被排队，然后在通常的事件回调之后执行。

延迟回调由 libevent 2.0.1-alpha 版引入。

4. bufferevent 的选项标志

创建 bufferevent 时可以使用一个或者多个标志修改其行为。可识别的标志有：

- BEV_OPT_CLOSE_ON_FREE：释放 bufferevent 时关闭底层传输端口。这将关闭底层套接字，释放底层 bufferevent 等。
- BEV_OPT_THREADSAFE：自动为 bufferevent 分配锁，这样就可以安全地在多个线程中使用 bufferevent。
- BEV_OPT_DEFER_CALLBACKS：设置这个标志时，bufferevent 延迟所有回调，如上所述。
- BEV_OPT_UNLOCK_CALLBACKS：默认情况下，如果设置 bufferevent 为线程安全的，则 bufferevent 会在调用用户提供的回调时进行锁定。设置这个选项会让 libevent在执行回调的时候不进行锁定。

BEV_OPT_UNLOCK_CALLBACKS 由2.0.5-beta 版引入，其他选项由2.0.1-alpha 版引入。

5. 与基于套接字的 bufferevent 一起工作

基于套接字的 bufferevent 是最简单的，它使用 libevent 的底层事件机制来检测底层网络套接字是否已经就绪，可以进行读写操作，并且使用底层网络调用（如 readv、writev、WSASend、WSARecv）来发送和接收数据。

5.1 创建基于套接字的 bufferevent

可以使用 bufferevent_socket_new()创建基于套接字的 bufferevent。

接口

```
struct bufferevent *bufferevent_socket_new(
    struct event_base *base,
    evutil_socket_t fd,
    enum bufferevent_options options);
```

base 是 event_base，options 是表示 bufferevent 选项（BEV_OPT_CLOSE_ON_FREE 等）的位掩码，fd 是一个可选的表示套接字的文件描述符。如果想以后设置文件描述符，可以设置 fd 为-1。

> 注意：
> [请确保您提供给bufferevent_socket_new的套接字处于非阻塞模式。 Libevent为此提供了便捷方法evutil_make_socket_nonblocking。]

成功时函数返回一个 bufferevent，失败则返回 NULL。

bufferevent_socket_new()函数由2.0.1-alpha 版新引入。

5.2 在基于套接字的 bufferevent 上启动连接

如果 bufferevent 的套接字还没有连接上，可以启动新的连接。

接口

```
int bufferevent_socket_connect(struct bufferevent *bev,
    struct sockaddr *address, int addrlen);
```

address 和 addrlen 参数跟标准调用 connect()的参数相同。如果还没有为 bufferevent 设置套接字，调用函数将为其分配一个新的流套接字，并且设置为非阻塞的。

如果已经为 bufferevent 设置套接字，调用 bufferevent_socket_connect()将告知 libevent套接字还未连接，直到连接成功之前不应该对其进行读取或者写入操作。

连接完成之前可以向输出缓冲区添加数据。

如果连接成功启动，函数返回0；如果发生错误则返回-1。

示例

```
#include <event2/event.h>
#include <event2/bufferevent.h>
#include <sys/socket.h>
#include <string.h>

void eventcb(struct bufferevent *bev, short events, void *ptr)
{
    if (events & BEV_EVENT_CONNECTED) {
         /* We're connected to 127.0.0.1:8080.   Ordinarily we'd do
            something here, like start reading or writing. */
    } else if (events & BEV_EVENT_ERROR) {
         /* An error occured while connecting. */
    }
}

int main_loop(void)
{
    struct event_base *base;
    struct bufferevent *bev;
    struct sockaddr_in sin;

    base = event_base_new();

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = htonl(0x7f000001); /* 127.0.0.1 */
    sin.sin_port = htons(8080); /* Port 8080 */

    bev = bufferevent_socket_new(base, -1, BEV_OPT_CLOSE_ON_FREE);

    bufferevent_setcb(bev, NULL, NULL, eventcb, NULL);

    if (bufferevent_socket_connect(bev,
        (struct sockaddr *)&sin, sizeof(sin)) < 0) {
        /* Error starting connection */
        bufferevent_free(bev);
        return -1;
    }

    event_base_dispatch(base);
    return 0;
}
```

bufferevent_socket_connect()函数由2.0.2-alpha 版引入。在此之前，必须自己手动在套接字上调用connect()，连接完成时，bufferevent 将报告写入事件。

> **注意：** 如果使用bufferevent_socket_connect()发起连接，将只会收到BEV_EVENT_CONNECTED 事件。如果自己调用 connect()，则连接上将被报告为写入事件。

如果想自己调用connect（），但在连接成功时仍然会收到BEV_EVENT_CONNECTED事件，请在connect（）返回errno等于EAGAIN或EINPROGRESS的-1之后，调用bufferevent_socket_connect（bev，NULL，0）。

此功能在Libevent 2.0.2-alpha中引入。

5.3 通过主机名启动连接

常常需要将解析主机名和连接到主机合并成单个操作，libevent 为此提供了：

接口

```
int bufferevent_socket_connect_hostname(struct bufferevent *bev,
    struct evdns_base *dns_base, int family, const char *hostname,
    int port);
int bufferevent_socket_get_dns_error(struct bufferevent *bev);
```

这个函数解析名字hostname，查找其family类型的地址（允许的地址族类型有AF_INET,AF_INET6和 AF_UNSPEC）。如果名字解析失败，函数将调用事件回调，报告错误事件。如果解析成功，函数将启动连接请求，就像 bufferevent_socket_connect()一样。

dns_base 参数是可选的：如果为 NULL，等待名字查找完成期间调用线程将被阻塞，而这通常不是期望的行为；如果提供 dns_base 参数，libevent 将使用它来异步地查询主机名。关于 DNS 的更多信息，请看第九章。

跟 bufferevent_socket_connect()一样，函数告知 libevent，bufferevent 上现存的套接字还没有连接，在名字解析和连接操作成功完成之前，不应该对套接字进行读取或者写入操作。

函数返回的错误可能是DNS主机名查询错误，可以调用bufferevent_socket_get_dns_error()来获取最近的错误。返回值0表示没有检测到 DNS 错误。

示例：简单的 HTTP v0客户端

```
/* Don't actually copy this code: it is a poor way to implement an
   HTTP client.  Have a look at evhttp instead.
*/
#include <event2/dns.h>
#include <event2/bufferevent.h>
#include <event2/buffer.h>
#include <event2/util.h>
#include <event2/event.h>

#include <stdio.h>

void readcb(struct bufferevent *bev, void *ptr)
{
    char buf[1024];
    int n;
    struct evbuffer *input = bufferevent_get_input(bev);
    while ((n = evbuffer_remove(input, buf, sizeof(buf))) > 0) {
        fwrite(buf, 1, n, stdout);
    }
```

```
}

void eventcb(struct bufferevent *bev, short events, void *ptr)
{
    if (events & BEV_EVENT_CONNECTED) {
         printf("Connect okay.\n");
    } else if (events & (BEV_EVENT_ERROR|BEV_EVENT_EOF)) {
         struct event_base *base = ptr;
         if (events & BEV_EVENT_ERROR) {
                int err = bufferevent_socket_get_dns_error(bev);
                if (err)
                        printf("DNS error: %s\n", evutil_gai_strerror(err));
         }
         printf("Closing\n");
         bufferevent_free(bev);
         event_base_loopexit(base, NULL);
    }
}

int main(int argc, char **argv)
{
    struct event_base *base;
    struct evdns_base *dns_base;
    struct bufferevent *bev;

    if (argc != 3) {
        printf("Trivial HTTP 0.x client\n"
               "Syntax: %s [hostname] [resource]\n"
               "Example: %s www.google.com /\n",argv[0],argv[0]);
        return 1;
    }

    base = event_base_new();
    dns_base = evdns_base_new(base, 1);

    bev = bufferevent_socket_new(base, -1, BEV_OPT_CLOSE_ON_FREE);
    bufferevent_setcb(bev, readcb, NULL, eventcb, base);
    bufferevent_enable(bev, EV_READ|EV_WRITE);
    evbuffer_add_printf(bufferevent_get_output(bev), "GET %s\r\n", argv[2]);
    bufferevent_socket_connect_hostname(
        bev, dns_base, AF_UNSPEC, argv[1], 80);
    event_base_dispatch(base);
    return 0;
}
```

bufferevent_socket_connect_hostname（）函数是Libevent 2.0.3-alpha中的新增功能；

bufferevent_socket_get_dns_error（）是2.0.5-beta中的新增功能。

6. 通用 bufferevent 操作

本节描述的函数可用于多种 bufferevent 实现。

6.1 释放 bufferevent

接口

```
void bufferevent_free(struct bufferevent *bev);
```

这个函数释放 bufferevent。bufferevent 内部具有引用计数，所以，如果释放 bufferevent 时还有未决的延迟回调，则在回调完成之前 bufferevent 不会被删除。

但是，bufferevent_free（）函数确实会尝试尽快释放bufferevent。 如果存在要在bufferevent上写入的待处理数据，则在释放bufferevent之前可能不会刷新该数据。

如果设置了 BEV_OPT_CLOSE_ON_FREE 标志，并且 bufferevent 有一个套接字或者底层bufferevent 作为其传输端口，则释放 bufferevent 将关闭这个传输端口。

这个函数由 libevent 0.8版引入。

6.2 操作回调、水位和启用/禁用

接口

```
typedef void (*bufferevent_data_cb)(struct bufferevent *bev, void *ctx);
typedef void (*bufferevent_event_cb)(struct bufferevent *bev,
    short events, void *ctx);

void bufferevent_setcb(struct bufferevent *bufev,
    bufferevent_data_cb readcb, bufferevent_data_cb writecb,
    bufferevent_event_cb eventcb, void *cbarg);

void bufferevent_getcb(struct bufferevent *bufev,
    bufferevent_data_cb *readcb_ptr,
    bufferevent_data_cb *writecb_ptr,
    bufferevent_event_cb *eventcb_ptr,
    void **cbarg_ptr);
```

bufferevent_setcb()函数修改 bufferevent 的一个或者多个回调。readcb、writecb 和 eventcb 函数将分别在已经读取足够的数据、已经写入足够的数据，或者发生错误时被调用。每个回调 函 数 的 第 一 个 参 数 都 是 发 生 了 事 件 的 bufferevent ， 最 后 一 个参数都是调用bufferevent_setcb()时用户提供的 cbarg 参数：可以通过它向回调传递数据。事件回调的events 参数是一个表示事件标志的位掩码：请看前面的“回调和水位”节。

要禁用回调，传递NULL而不是回调函数。注意：bufferevent的所有回调函数共享单个cbarg， 所以修改它将影响所有回调函数。

可以通过将指针传递给bufferevent_getcb（）来检索bufferevent的当前设置的回调，该指针将* readcb_ptr设置为当前的读回调，将* writecb_ptr设置为当前的写回调，将* eventcb_ptr设置为当前事件的回调，并将* cbarg_ptr设置为当前的回调 回调参数字段。 这些指针中的任何一个设置为NULL都将被忽略。

Libevent 1.4.4中引入了bufferevent_setcb（）函数。 在Libevent 2.0.2-alpha中，类型名称“ bufferevent_data_cb”和“ bufferevent_event_cb”是新的。 在2.1.1-alpha中添加了 bufferevent_getcb（）函数。

接口

```
void bufferevent_enable(struct bufferevent *bufev, short events);
void bufferevent_disable(struct bufferevent *bufev, short events);

short bufferevent_get_enabled(struct bufferevent *bufev);
```

可以启用或者禁用 bufferevent 上的 EV_READ、EV_WRITE 或者 EV_READ | EV_WRITE 事件。没有启用读取或者写入事件时，bufferevent 将不会试图进行数据读取或者写入。

没有必要在输出缓冲区空时禁用写入事件：bufferevent 将自动停止写入，然后在有数据等待写入时重新开始。

类似地，没有必要在输入缓冲区高于高水位时禁用读取事件：bufferevent 将自动停止读取，然后在有空间用于读取时重新开始读取。

默认情况下，新创建的 bufferevent 的写入是启用的，但是读取没有启用。

可以调用 bufferevent_get_enabled()确定 bufferevent 上当前启用的事件。

除了 bufferevent_get_enabled()由2.0.3-alpha 版引入外，这些函数都由0.8版引入。

接口

```
void bufferevent_setwatermark(struct bufferevent *bufev, short events,
    size_t lowmark, size_t highmark);
```

bufferevent_setwatermark()函数调整单个 bufferevent 的读取水位、写入水位，或者同时调整二者。（如果events参数设置了EV_READ，调整读取水位。如果events设置了EV_WRITE标志，调整写入水位）

对于高水位，0表示“无限”。

这个函数首次出现在1.4.4版。

示例

```
#include <event2/event.h>
#include <event2/bufferevent.h>
#include <event2/buffer.h>
#include <event2/util.h>

#include <stdlib.h>
#include <errno.h>
#include <string.h>

struct info {
    const char *name;
    size_t total_drained;
};

void read_callback(struct bufferevent *bev, void *ctx)
{
    struct info *inf = ctx;
    struct evbuffer *input = bufferevent_get_input(bev);
    size_t len = evbuffer_get_length(input);
    if (len) {
        inf->total_drained += len;
        evbuffer_drain(input, len);
        printf("Drained %lu bytes from %s\n",
            (unsigned long) len, inf->name);
    }
}

void event_callback(struct bufferevent *bev, short events, void *ctx)
{
    struct info *inf = ctx;
    struct evbuffer *input = bufferevent_get_input(bev);
    int finished = 0;
```

```
    if (events & BEV_EVENT_EOF) {
        size_t len = evbuffer_get_length(input);
        printf("Got a close from %s.  We drained %lu bytes from it, "
            "and have %lu left.\n", inf->name,
            (unsigned long)inf->total_drained, (unsigned long)len);
        finished = 1;
    }
    if (events & BEV_EVENT_ERROR) {
        printf("Got an error from %s: %s\n",
            inf->name, evutil_socket_error_to_string(EVUTIL_SOCKET_ERROR()));
        finished = 1;
    }
    if (finished) {
        free(ctx);
        bufferevent_free(bev);
    }
}

struct bufferevent *setup_bufferevent(void)
{
    struct bufferevent *b1 = NULL;
    struct info *info1;

    info1 = malloc(sizeof(struct info));
    info1->name = "buffer 1";
    info1->total_drained = 0;

    /* ... Here we should set up the bufferevent and make sure it gets
       connected... */

    /* Trigger the read callback only whenever there is at least 128 bytes
       of data in the buffer. */
    bufferevent_setwatermark(b1, EV_READ, 128, 0);

    bufferevent_setcb(b1, read_callback, NULL, event_callback, info1);

    bufferevent_enable(b1, EV_READ); /* Start reading. */
    return b1;
}
```

6.3 操作 bufferevent 中的数据

如果只是通过网络读取或者写入数据，而不能观察操作过程，是没什么好处的。bufferevent提供了下列函数用于观察要写入或者读取的数据。

接口

```
struct evbuffer *bufferevent_get_input(struct bufferevent *bufev);
struct evbuffer *bufferevent_get_output(struct bufferevent *bufev);
```

这两个函数提供了非常强大的基础：它们分别返回输入和输出缓冲区。关于可以对 evbuffer类型进行的所有操作的完整信息，请看下一章。

请注意，应用程序只能删除（不添加）输入缓冲区中的数据，并且只能添加（不删除）输出缓冲区中的数据。

如果写入操作因为数据量太少而停止（或者读取操作因为太多数据而停止），则向输出缓冲区添加数据（或者从输入缓冲区移除数据）将自动重启操作。

这些函数由2.0.1-alpha 版引入。

接口

```
int bufferevent_write(struct bufferevent *bufev,
    const void *data, size_t size);
int bufferevent_write_buffer(struct bufferevent *bufev,
    struct evbuffer *buf);
```

这些函数向 bufferevent 的输出缓冲区添加数据。

bufferevent_write()将内存中从 data 处开始的 size 字节数据添加到输出缓冲区的末尾。

bufferevent_write_buffer()移除 buf 的所有内容，将其放置到输出缓冲区的末尾。成功时这些函数都返回0，发生错误时则返回-1。

这些函数从0.8版就存在了。

接口

```
size_t bufferevent_read(struct bufferevent *bufev, void *data, size_t size);
int bufferevent_read_buffer(struct bufferevent *bufev,
    struct evbuffer *buf);
```

这些函数从 bufferevent 的输入缓冲区移除数据。

bufferevent_read()至多从输入缓冲区移除size 字节的数据，将其存储到内存中 data 处。函数返回实际移除的字节数。

bufferevent_read_buffer()函数抽空输入缓冲区的所有内容，将其放置到 buf 中，成功时返回0，失败时返回-1。

注意，对于 bufferevent_read()，data 处的内存块必须有足够的空间容纳 size 字节数据。

bufferevent_read()函数从0.8版就存在了；bufferevnet_read_buffer()由2.0.1-alpha 版引入。

示例

```
#include <event2/bufferevent.h>
#include <event2/buffer.h>

#include <ctype.h>

void
read_callback_uppercase(struct bufferevent *bev, void *ctx)
{
        /* This callback removes the data from bev's input buffer 128
           bytes at a time, uppercases it, and starts sending it
           back.

           (Watch out!  In practice, you shouldn't use toupper to implement
           a network protocol, unless you know for a fact that the current
           locale is the one you want to be using.)
         */

        char tmp[128];
```

```
        size_t n;
        int i;
        while (1) {
                n = bufferevent_read(bev, tmp, sizeof(tmp));
                if (n <= 0)
                        break; /* No more data. */
                for (i=0; i<n; ++i)
                        tmp[i] = toupper(tmp[i]);
                bufferevent_write(bev, tmp, n);
        }
}

struct proxy_info {
        struct bufferevent *other_bev;
};
void
read_callback_proxy(struct bufferevent *bev, void *ctx)
{
        /* You might use a function like this if you're implementing
           a simple proxy: it will take data from one connection (on
           bev), and write it to another, copying as little as
           possible. */
        struct proxy_info *inf = ctx;

        bufferevent_read_buffer(bev,
            bufferevent_get_output(inf->other_bev));
}

struct count {
        unsigned long last_fib[2];
};

void
write_callback_fibonacci(struct bufferevent *bev, void *ctx)
{
        /* Here's a callback that adds some Fibonacci numbers to the
           output buffer of bev.  It stops once we have added 1k of
           data; once this data is drained, we'll add more. */
        struct count *c = ctx;

        struct evbuffer *tmp = evbuffer_new();
        while (evbuffer_get_length(tmp) < 1024) {
                unsigned long next = c->last_fib[0] + c->last_fib[1];
                c->last_fib[0] = c->last_fib[1];
                c->last_fib[1] = next;

                evbuffer_add_printf(tmp, "%lu", next);
        }

        /* Now we add the whole contents of tmp to bev. */
        bufferevent_write_buffer(bev, tmp);

        /* We don't need tmp any longer. */
        evbuffer_free(tmp);
}
```

6.4 读写超时

跟其他事件一样，可以要求在一定量的时间已经流逝，而没有成功写入或者读取数据的时候调用一个超时回调。

接口

```
void bufferevent_set_timeouts(struct bufferevent *bufev,
    const struct timeval *timeout_read, const struct timeval *timeout_write);
```

设置超时为 NULL 会移除超时回调。

试图读取数据的时候，如果至少等待了 timeout_read 秒，则读取超时事件将被触发。试图写入数据的时候，如果至少等待了 timeout_write 秒，则写入超时事件将被触发。

注意，只有在读取或者写入的时候才会计算超时。也就是说，如果 bufferevent 的读取被禁止，或者输入缓冲区满（达到其高水位），则读取超时被禁止。类似的，如果写入被禁止，或者没有数据待写入，则写入超时被禁止。

读取或者写入超时发生时，相应的读取或者写入操作被禁止，然后超时事件回调被调用，带有标志 BEV_EVENT_TIMEOUT | BEV_EVENT_READING或者BEV_EVENT_TIMEOUT | BEV_EVENT_WRITING。

这个函数从2.0.1-alpha 版就存在了，但是直到2.0.4-alpha 版才对于各种 bufferevent 类型行为一致。

6.5 对 bufferevent 发起清空操作

接口

```
int bufferevent_flush(struct bufferevent *bufev,
    short iotype, enum bufferevent_flush_mode state);
```

清空 bufferevent 要求 bufferevent 强制从底层传输端口读取或者写入尽可能多的数据，而忽略其他可能保持数据不被写入的限制条件。函数的细节功能依赖于 bufferevent 的具体类型。

iotype 参数应该是 EV_READ、EV_WRITE 或者 EV_READ | EV_WRITE，用于指示应该处理读取、写入，还是二者都处理。state 参数可以是 BEV_NORMAL、BEV_FLUSH 或者BEV_FINISHED。BEV_FINISHED 指示应该告知另一端，没有更多数据需要发送了； 而BEV_NORMAL 和 BEV_FLUSH 的区别依赖于具体的 bufferevent 类型。失败时 bufferevent_flush()返回-1，如果没有数据被清空则返回0，有数据被清空则返回1。

当前（2.0.5-beta 版）仅有一些 bufferevent 类型实现了 bufferevent_flush()。特别是，基于套接字 bufferevent 没有实现。

7. 类型特定的 bufferevent 函数

这些 bufferevent 函数不能支持所有 bufferevent 类型。

接口

```
int bufferevent_priority_set(struct bufferevent *bufev, int pri);
int bufferevent_get_priority(struct bufferevent *bufev);
```

这个函数调整 bufev 的优先级为 pri。关于优先级的更多信息请看 event_priority_set()。

成功时函数返回0，失败时返回-1。这个函数仅能用于基于套接字的 bufferevent。

这个函数由Libevent 1.0中引入了bufferevent_priority_set（）函数； 直到Libevent 2.1.2-alpha才出现bufferevent_get_priority（）。

接口

```
int bufferevent_setfd(struct bufferevent *bufev, evutil_socket_t fd);
evutil_socket_t bufferevent_getfd(struct bufferevent *bufev);
```

这些函数设置或者返回基于 fd 的事件的文件描述符。只有基于套接字的 bufferevent 支持setfd()。两个函数都在失败时返回-1；setfd()成功时返回0。

bufferevent_setfd()函数由1.4.4版引入；bufferevent_getfd()函数由2.0.2-alpha 版引入。

接口

```
struct event_base *bufferevent_get_base(struct bufferevent *bev);
```

这个函数返回 bufferevent 的 event_base，由2.0.9-rc 版引入。

接口

```
struct bufferevent *bufferevent_get_underlying(struct bufferevent *bufev);
```

这个函数返回作为 bufferevent 底层传输端口的另一个 bufferevent。关于这种情况，请看关于过滤型 bufferevent 的介绍。

这个函数由2.0.2-alpha 版引入。

8. 手动锁定和解锁

有时候需要确保对 bufferevent 的一些操作是原子地执行的。为此，libevent 提供了手动锁定和解锁 bufferevent 的函数。

接口

```
void bufferevent_lock(struct bufferevent *bufev);
void bufferevent_unlock(struct bufferevent *bufev);
```

注意：如果创建 bufferevent 时没有指定 BEV_OPT_THREADSAFE 标志，或者没有激活libevent 的线程支持，则锁定操作是没有效果的。

用这个函数锁定 bufferevent 将自动同时锁定相关联的 evbuffer。这些函数是递归的：锁定已经持有锁的 bufferevent 是安全的。当然，对于每次锁定都必须进行一次解锁。

这些函数由2.0.6-rc 版引入。

七、Bufferevent：高级话题

本章介绍了Libevent的bufferevent实现的一些高级功能，这些功能对于典型用途不是必需的。 如果您只是在学习如何使用bufferevents，则应该暂时跳过本章，然后继续阅读evbuffer章。

1. 成对的 bufferevent

有时候网络程序需要与自身通信。比如说，通过某些协议对用户连接进行隧道操作的程序，有时候也需要通过同样的协议对自身的连接进行隧道操作。当然，可以通过打开一个到自身监听端口的连接，让程序使用这个连接来达到这种目标。但是，通过网络栈来与自身通信比较浪费资源。

替代的解决方案是，创建一对成对的 bufferevent。这样，写入到一个 bufferevent 的字节都被另一个接收（反过来也是），但是不需要使用套接字。

接口

```
int bufferevent_pair_new(struct event_base *base, int options,
    struct bufferevent *pair[2]);
```

调用 bufferevent_pair_new()会设置 pair[0]和 pair[1]为一对相互连接的 bufferevent。除了 BEV_OPT_CLOSE_ON_FREE 无效、BEV_OPT_DEFER_CALLBACKS 总是打开的之外， 所有通常的选项都是支持的。

为什么 bufferevent 对需要带延迟回调运行？通常某一方上的操作会调用一个通知另一方的回调，从而调用另一方的回调，如此这样进行很多步。如果不延迟回调，这种调用链常常会导致栈溢出或者饿死其他连接，而且还要求所有的回调是可重入的。

成对的 bufferevent 支持 flush：设置模式参数为 BEV_NORMAL 或者 BEV_FLUSH 会强制要求所有相关数据从对中的一个 bufferevent 传输到另一个中，而忽略可能会限制传输的水位设置。增加 BEV_FINISHED 到模式参数中还会让对端的 bufferevent 产生 EOF 事件。

释放对中的任何一个成员不会自动释放另一个，也不会产生 EOF 事件。释放仅仅会使对中的另一个成员成为断开的。bufferevent 一旦断开，就不能再成功读写数据或者产生任何事件了。

接口

```
struct bufferevent *bufferevent_pair_get_partner(struct bufferevent *bev)
```

有时可能只需要给一个缓冲事件对中的另一个成员。 为此，您可以调用 bufferevent_pair_get_partner（）函数。 如果bev是该对中的一个成员，并且另一个成员仍然存在，它将返回该对中的另一个成员。 否则，它返回NULL。

bufferevent 对由2.0.1-alpha 版本引入，而 bufferevent_pair_get_partner()函数由2.0.6版本引入。

2. 过滤 bufferevent

有时候需要转换传递给某 bufferevent 的所有数据，这可以通过添加一个压缩层，或者将协议包装到另一个协议中进行传输来实现。

接口

```
enum bufferevent_filter_result {
        BEV_OK = 0,
        BEV_NEED_MORE = 1,
        BEV_ERROR = 2
};
typedef enum bufferevent_filter_result (*bufferevent_filter_cb)(
    struct evbuffer *source, struct evbuffer *destination, ev_ssize_t dst_limit,
    enum bufferevent_flush_mode mode, void *ctx);


struct bufferevent *bufferevent_filter_new(struct bufferevent *underlying,
        bufferevent_filter_cb input_filter,
        bufferevent_filter_cb output_filter,
        int options,
        void (*free_context)(void *),
        void *ctx);
```

bufferevent_filter_new （）函数创建一个封装现有的“底层”bufferevent 的过滤 bufferevent。所有通过底层 bufferevent 接收的数据在到达过滤 bufferevent 之前都会经过“输入”过滤器的转换；所有通过底层 bufferevent 发送的数据在被发送到底层 bufferevent 之前都会经过“输出”过滤器的转换。

向底层 bufferevent 添加过滤器将替换其回调函数。可以向底层 bufferevent 的 evbuffer 添加回调函数，但是如果想让过滤器正确工作，就不能再设置 bufferevent 本身的回调函数。

input_filter 和 output_filter 函数将随后描述。options 参数支持所有通常的选项。如果设置了 BEV_OPT_CLOSE_ON_FREE ， 那 么 释 放 过 滤 bufferevent 也会同时释放底层bufferevent。ctx 参数是传递给过滤函数的任意指针；如果提供了 free_context，则在释放 ctx 之前它会被调用。

底层输入缓冲区有数据可读时，输入过滤器函数会被调用；过滤器的输出缓冲区有新的数据待写入时，输出过滤器函数会被调用。两个过滤器函数都有一对 evbuffer 参数：从 source 读取数据；向 destination 写入数据，而 dst_limit 参数描述了可以写入 destination 的字节数 上限。过滤器函数可以忽略这个参数，但是这样可能会违背高水位或者速率限制。如果dst_limit 是-1，则没有限制。mode 参数向过滤器描述了写入的方式。值 BEV_NORMAL 表示应该在方便转换的基础上写入尽可能多的数据；而 BEV_FLUSH 表示写入尽可能多的数据；BEV_FINISHED 表示过滤器函数应该在流的末尾执行额外的清理操作。最后，过滤器函数的 ctx 参数就是传递给 bufferevent_filter_new（）函数的指针（ctx 参数）。

如果成功向目标缓冲区写入了任何数据，过滤器函数应该返回 BEV_OK；如果不获得更多的输入，或者不使用不同的清空（flush）模式，就不能向目标缓冲区写入更多的数据，则应该返回 BEV_NEED_MORE；如果过滤器上发生了不可恢复的错误，则应该返回BEV_ERROR。

创建过滤器将启用底层 bufferevent 的读取和写入。随后就不需要自己管理读取和写入了：过滤器在不想读取的时候会自动挂起底层 bufferevent 的读取。从2.0.8-rc 版本开始，可以在过滤器之外独立地启用/禁用底层 bufferevent 的读取和写入。然而，这样可能会让过滤器不能成功取得所需要的数据。

不需要同时指定输入和输出过滤器：没有给定的过滤器将被一个不进行数据转换的过滤器取代。

3. 限制最大单读/写大小

默认情况下，每次事件循环调用时，bufferevent都不会读取或写入最大可能的字节数； 这样做会导致奇怪的不公平行为和资源匮乏。 另一方面，默认值可能并非在所有情况下都合理。

接口

```
int bufferevent_set_max_single_read（struct bufferevent * bev，size_t size）；
int bufferevent_set_max_single_write（struct bufferevent * bev，size_t size）；

ev_ssize_t bufferevent_get_max_single_read（struct bufferevent * bev）；
ev_ssize_t bufferevent_get_max_single_write（struct bufferevent * bev）；
```

这两个“设置”功能分别替换了当前的最大读取和写入最大值。 如果大小值为0或大于EV_SSIZE_MAX，则将最大值设置为默认值。 这些函数成功返回0，失败返回-1。

这两个“ get”函数分别返回当前的每个循环读取和写入最大值。

这些功能已添加到2.1.1-alpha中。

4. bufferevent 和速率限制

某些程序需要限制单个或者一组 bufferevent 使用的带宽。2.0.4-alpha 和2.0.5-alpha 版本添加了为单个或者一组 bufferevent 设置速率限制的基本功能。

4.1 速率限制模型

libevent 的速率限制使用记号存储器（token bucket）算法确定在某时刻可以写入或者读取多少字节。每个速率限制对象在任何给定时刻都有一个“读存储器（read bucket）”和一个“写存储器（write bucket）”，其大小决定了对象可以立即读取或者写入多少字节。每个存储器有一个填充速率，一个最大突发尺寸，和一个时间单位，或者说“滴答（tick）”。一个时间单位流逝后，存储器被填充一些字节（决定于填充速率）——但是如果超过其突发尺寸，则超出的字节会丢失。

因此，填充速率决定了对象发送或者接收字节的最大平均速率，而突发尺寸决定了在单次突发中可以发送或者接收的最大字节数；时间单位则确定了传输的平滑程度。

4.2 为 bufferevent 设置速率限制

接口

```
#define EV_RATE_LIMIT_MAX EV_SSIZE_MAX
struct ev_token_bucket_cfg;
struct ev_token_bucket_cfg *ev_token_bucket_cfg_new(
        size_t read_rate, size_t read_burst,
        size_t write_rate, size_t write_burst,
        const struct timeval *tick_len);
void ev_token_bucket_cfg_free(struct ev_token_bucket_cfg *cfg);
int bufferevent_set_rate_limit(struct bufferevent *bev,
    struct ev_token_bucket_cfg *cfg);
```

ev_token_bucket_cfg 结构体代表用于限制单个或者一组 bufferevent 的一对记号存储器的配置值。要创建 ev_token_bucket_cfg，调用 ev_token_bucket_cfg_new 函数，提供最大平均读取速率、最大突发读取量、最大平均写入速率、最大突发写入量，以及一个滴答的长度。如果 tick_len 参数为 NULL，则默认的滴答长度为一秒。如果发生错误，函数会返回 NULL。

注意：read_rate 和 write_rate 参数的单位是字节每滴答。也就是说，如果滴答长度是十分之一秒，read_rate 是300，则最大平均读取速率是3000字节每秒。此外，不支持大于EV_RATE_LIMIT_MAX 的速率或者突发量。

要限制bufferevent的传输速率，使用一个 ev_token_bucket_cfg ， 对其调用 bufferevent_set_rate_limit（）。成功时函数返回0，失败时返回-1。可以对任意数量的bufferevent 使用相同的 ev_token_bucket_cfg 。要移除速率限制，可以调用 bufferevent_set_rate_limit（），传递 NULL 作为 cfg 参数值。

调用 ev_token_bucket_cfg_free（）可以释放 ev_token_bucket_cfg。注意：当前在没有任何 bufferevent 使用 ev_token_bucket_cfg 之前进行释放是不安全的。

4.3 为一组 bufferevent 设置速率限制

如果要限制一组 bufferevent 总的带宽使用，可以将它们分配到一个速率限制组中。

接口

```
struct bufferevent_rate_limit_group;

struct bufferevent_rate_limit_group *bufferevent_rate_limit_group_new(
        struct event_base *base,
        const struct ev_token_bucket_cfg *cfg);
int bufferevent_rate_limit_group_set_cfg(
        struct bufferevent_rate_limit_group *group,
        const struct ev_token_bucket_cfg *cfg);
void bufferevent_rate_limit_group_free(struct bufferevent_rate_limit_group *);
int bufferevent_add_to_rate_limit_group(struct bufferevent *bev,
    struct bufferevent_rate_limit_group *g);
int bufferevent_remove_from_rate_limit_group(struct bufferevent *bev);
```

要创建速率限制组，使用一个 event_base 和一个已经初始化的 ev_token_bucket_cfg 作为参数调用 bufferevent_rate_limit_group_new 函数。使用bufferevent_add_to_rate_limit_group 将 bufferevent 添加到组中；使用bufferevent_remove_from_rate_limit_group 从组中删除 bufferevent。这些函数成功时返回

0，失败时返回-1。

单个 bufferevent 在某时刻只能是一个速率限制组的成员。bufferevent 可以同时有单独的速率限制（通过 bufferevent_set_rate_limit 设置）和组速率限制。设置了这两个限制时，对每个 bufferevent，较低的限制将被应用。

调用 bufferevent_rate_limit_group_set_cfg 修改组的速率限制。函数成功时返回0，失败时返回-1。bufferevent_rate_limit_group_free 函数释放速率限制组，移除所有成员。

在2.0版本中，组速率限制试图实现总体的公平，但是具体实现可能在小的时间范围内并不公平。如果你强烈关注调度的公平性，请帮助提供未来版本的补丁。

4.4 检查当前速率限制

有时候需要得知应用到给定 bufferevent 或者组的速率限制，为此，libevent 提供了函数：

接口

```
ev_ssize_t bufferevent_get_read_limit(struct bufferevent *bev);
ev_ssize_t bufferevent_get_write_limit(struct bufferevent *bev);
ev_ssize_t bufferevent_rate_limit_group_get_read_limit(
        struct bufferevent_rate_limit_group *);
ev_ssize_t bufferevent_rate_limit_group_get_write_limit(
        struct bufferevent_rate_limit_group *);
```

上述函数返回以字节为单位的 bufferevent 或者组的读写记号存储器大小。注意：如果bufferevent 已经被强制超过其配置（清空(flush)操作就会这样），则这些值可能是负数。

接口

```
ev_ssize_t bufferevent_get_max_to_read(struct bufferevent *bev);
ev_ssize_t bufferevent_get_max_to_write(struct bufferevent *bev);
```

这些函数返回在考虑了应用到 bufferevent 或者组（如果有）的速率限制，以及一次最大读写数据量的情况下，现在可以读或者写的字节数。

接口

```
void bufferevent_rate_limit_group_get_totals(
    struct bufferevent_rate_limit_group *grp,
    ev_uint64_t *total_read_out, ev_uint64_t *total_written_out);
void bufferevent_rate_limit_group_reset_totals(
    struct bufferevent_rate_limit_group *grp);
```

每个 bufferevent_rate_limit_group 跟踪经过其发送的总的字节数，这可用于跟踪组中所有 bufferevent 总的使用情况。对一个组调用 bufferevent_rate_limit_group_get_totals 会分别设置 total_read_out 和 total_written_out 为组的总读取和写入字节数。组创建的时候这些计数从0开始，调用 bufferevent_rate_limit_group_reset_totals 会复位计数为0。

4.5 手动调整速率限制

对于有复杂需求的程序，可能需要调整记号存储器的当前值。比如说，如果程序不通过使用 bufferevent 的方式产生一些通信量时。

接口

```
int bufferevent_decrement_read_limit(struct bufferevent *bev, ev_ssize_t decr);
int bufferevent_decrement_write_limit(struct bufferevent *bev, ev_ssize_t decr);
int bufferevent_rate_limit_group_decrement_read(
        struct bufferevent_rate_limit_group *grp, ev_ssize_t decr);
int bufferevent_rate_limit_group_decrement_write(
        struct bufferevent_rate_limit_group *grp, ev_ssize_t decr);
```

这些函数减小某个 bufferevent 或者速率限制组的当前读或者写存储器。注意：减小是有符号的。如果要增加存储器，就传入负值。

4.6 设置速率限制组的最小可能共享

通常，不希望在每个滴答中为速率限制组中的所有 bufferevent 平等地分配可用的字节。比如说，有一个含有10000个活动 bufferevent 的速率限制组，它在每个滴答中可以写入10000字节，那么，因为系统调用和 TCP 头部的开销，让每个 bufferevent 在每个滴答中仅写入1字节是低效的。

为解决此问题，速率限制组有一个“最小共享（minimum share）”的概念。在上述情况下，不是允许每个 bufferevent 在每个滴答中写入1字节，而是在每个滴答中允许某个 bufferevent写入一些（最小共享）字节，而其余的 bufferevent 将不允许写入。允许哪个 bufferevent写入将在每个滴答中随机选择。

默认的最小共享值具有较好的性能，当前（2.0.6-rc 版本）其值为64。可以通过这个函数调整最小共享值：

接口

```
int bufferevent_rate_limit_group_set_min_share(
        struct bufferevent_rate_limit_group *group, size_t min_share);
```

设置 min_share 为0将会完全禁止最小共享。

速率限制功能从引入开始就具有最小共享了，而修改最小共享的函数在2.0.6-rc 版本首次引入。

4.7 速率限制实现的限制

2.0版本的 libevent 的速率限制具有一些实现上的限制：

- 不是每种 bufferevent 类型都良好地或者说完整地支持速率限制。
- bufferevent 速率限制组不能嵌套，一个 bufferevent 在某时刻只能属于一个速率限制组。
- 速率限制实现仅计算 TCP 分组传输的数据，不包括 TCP 头部。
- 读速率限制实现依赖于 TCP 栈通知应用程序仅仅以某速率消费数据，并且在其缓冲区满的时候将数据推送到 TCP 连接的另一端。
- 某些 bufferevent 实现（特别是 Windows 中的 IOCP 实现）可能调拨过度。
- 存储器开始于一个滴答的通信量。这意味着 bufferevent 可以立即开始读取或者写入，而不用等待一个滴答的时间。但是这也意味着速率被限制为 N.1个滴答的 bufferevent 可能传输 N+1个滴答的通信量。
- 滴答不能小于1毫秒，毫秒的小数部分都被忽略。

5. bufferevent 和 SSL

bufferevent 可以使用 OpenSSL 库实现 SSL/TLS 安全传输层。因为很多应用不需要或者不想链接 OpenSSL，这部分功能在单独的 libevent_openssl 库中实现。未来版本的 libevent可能会添加其他 SSL/TLS 库，如 NSS 或者 GnuTLS，但是当前只有 OpenSSL。

OpenSSL 功能在2.0.3-alpha 版本引入，然而直到2.0.5-beta 和2.0.6-rc 版本才能良好工作。

这一节不包含对 OpenSSL、SSL/TLS 或者密码学的概述。

这一节描述的函数都在 event2/bufferevent_ssl.h 中声明。

创建和使用基于 OpenSSL 的 bufferevent

接口

```
enum bufferevent_ssl_state {
        BUFFEREVENT_SSL_OPEN = 0,
        BUFFEREVENT_SSL_CONNECTING = 1,
        BUFFEREVENT_SSL_ACCEPTING = 2
};

struct bufferevent *
bufferevent_openssl_filter_new(struct event_base *base,
    struct bufferevent *underlying,
    SSL *ssl,
    enum bufferevent_ssl_state state,
    int options);

struct bufferevent *
bufferevent_openssl_socket_new(struct event_base *base,
    evutil_socket_t fd,
    SSL *ssl,
    enum bufferevent_ssl_state state,
    int options);
```

可以创建两种SSL bufferevent：基于过滤器的bufferevent，该事件通过另一个基础bufferevent进行通信，或者基于套接字的bufferevent，通知OpenSSL直接通过网络与网络进行通信。无论哪种情况，都必须提供SSL对象和SSL对象状态的描述。如果SSL当前正在作为客户端进行协商，则状态应为BUFFEREVENT_SSL_CONNECTING；如果SSL当前正在作为服务器进行协商，则状态应为BUFFEREVENT_SSL_ACCEPTING；如果SSL握手已完成，则状态应为BUFFEREVENT_SSL_OPEN。

接受常规选项； BEV_OPT_CLOSE_ON_FREE使当关闭openssl bufferevent本身时，SSL对象和基础fd或bufferevent关闭。

握手完成后，将使用标志中的BEV_EVENT_CONNECTED调用新的bufferevent的事件回调。

如果您要创建基于套接字的bufferEvent，并且SSL对象已经设置了套接字，则无需自己提供套接字：只需传递-1。您还可以稍后使用bufferevent_setfd（）设置fd。

/// TODO：bufferevent_shutdown（）API完成后，请删除此代码。

请注意，在SSL缓冲区事件中设置BEV_OPT_CLOSE_ON_FREE时，将不会在SSL连接上执行干净关闭。这有两个问题：首先，连接似乎已被另一端“断开”，而不是被完全关闭：另一方将无法告知您是否关闭了连接，或断开了连接攻击者或第三方。其次，OpenSSL将会话视为“不良”会话，并从会话缓存中删除。这可能导致负载下的SSL应用程序性能显着下降。

当前，唯一的解决方法是手动关闭惰性SSL。虽然这破坏了TLS RFC，但可以确保会话一旦关闭就将保留在缓存中。下面的代码实现此解决方法。

示例

```
SSL *ctx = bufferevent_openssl_get_ssl(bev);

/*
 * SSL_RECEIVED_SHUTDOWN tells SSL_shutdown to act as if we had already
 * received a close notify from the other end.  SSL_shutdown will then
 * send the final close notify in reply.  The other end will receive the
```

```
 * close notify and send theirs.  By this time, we will have already
 * closed the socket and the other end's real close notify will never be
 * received.  In effect, both sides will think that they have completed a
 * clean shutdown and keep their sessions valid.  This strategy will fail
 * if the socket is not ready for writing, in which case this hack will
 * lead to an unclean shutdown and lost session on the other end.
 */
SSL_set_shutdown(ctx, SSL_RECEIVED_SHUTDOWN);
SSL_shutdown(ctx);
bufferevent_free(bev);
```

接口

```
SSL *bufferevent_openssl_get_ssl(struct bufferevent *bev);
```

这个函数返回 OpenSSL bufferevent 使用的 SSL 对象。如果 bev 不是一个基于 OpenSSL 的 bufferevent，则返回 NULL。

接口

```
unsigned long bufferevent_get_openssl_error(struct bufferevent *bev);
```

这个函数返回给定 bufferevent 的第一个未决的 OpenSSL 错误；如果没有未决的错误，则返回0。错误值的格式与 openssl 库中的 ERR_get_error（）返回的相同。

接口

```
int bufferevent_ssl_renegotiate(struct bufferevent *bev);
```

调用这个函数要求 SSL 重新协商，bufferevent 会调用合适的回调函数。这是个高级功能，通常应该避免使用，除非你确实知道自己在做什么，特别是有些 SSL 版本具有与重新协商相关的安全问题。

接口

```
int bufferevent_openssl_get_allow_dirty_shutdown(struct bufferevent *bev);
void bufferevent_openssl_set_allow_dirty_shutdown(struct bufferevent *bev,
    int allow_dirty_shutdown);
```

SSL协议的所有良好版本（即SSLv3和所有TLS版本）都支持经过身份验证的关闭操作，该操作使各方可以区分下意识缓冲区中的故意关闭与意外或恶意引发的终止。 默认情况下，我们将正确关闭以外的所有内容都视为连接错误。 但是，如果allow_dirty_shutdown标志设置为1，则将连接中的关闭视为BEV_EVENT_EOF。

Libevent 2.1.1-alpha中添加了allow_dirty_shutdown函数。

示例：一个简单的基于SSL的回显服务器

八、evbuffer：缓冲 IO 实用功能

libevent 的 evbuffer 实现了为向后面添加数据和从前面移除数据而优化的字节队列。

evbuffer 用于处理缓冲网络 IO 的“缓冲”部分。它不提供调度 IO 或者当 IO 就绪时触发 IO 的功能：这是 bufferevent 的工作。

除非特别说明，本章描述的函数都在 event2/buffer.h 中声明。

1. 创建和释放 evbuffer

接口

```
struct evbuffer *evbuffer_new(void);
void evbuffer_free(struct evbuffer *buf);
```

这两个函数的功能很简明： evbuffer_new() 分配和返回一个新的空 evbuffer； 而 evbuffer_free()释放 evbuffer 和其内容。

这两个函数从 libevent 0.8版就存在了。

2. evbuffer 与线程安全

接口

```
int evbuffer_enable_locking(struct evbuffer *buf, void *lock);
void evbuffer_lock(struct evbuffer *buf);
void evbuffer_unlock(struct evbuffer *buf);
```

默认情况下，在多个线程中同时访问 evbuffer 是不安全的。如果需要这样的访问，可以调用 evbuffer_enable_locking() 。如果lock参数为NULL ， libevent会使用 evthread_set_lock_creation_callback 提供的锁创建函数创建一个锁。否则，libevent 将 lock参数用作锁。

evbuffer_lock()和 evbuffer_unlock()函数分别请求和释放 evbuffer 上的锁。可以使用这两个函数让一系列操作是原子的。如果 evbuffer 没有启用锁，这两个函数不做任何操作。

> 注意：对于单个操作，不需要调用 evbuffer_lock()和 evbuffer_unlock()：如果 evbuffer启用了锁，单个操作就已经是原子的。只有在需要多个操作连续执行，不让其他线程介入的时候，才需要手动锁定 evbuffer

这些函数都在2.0.1-alpha 版本中引入。

3. 检查evbuffer

接口

```
size_t evbuffer_get_length(const struct evbuffer *buf);
```

这个函数返回 evbuffer 存储的字节数，它在2.0.1-alpha 版本中引入。

接口

```
size_t evbuffer_get_contiguous_space(const struct evbuffer *buf);
```

这个函数返回连续地存储在 evbuffer 前面的字节数。evbuffer 中的数据可能存储在多个分隔开的内存块中，这个函数返回当前第一个块中的字节数。

这个函数在2.0.1-alpha 版本引入。

4. 向 evbuffer 添加数据：基础

接口

```
int evbuffer_add(struct evbuffer *buf, const void *data, size_t datlen);
```

这个函数添加 data 处的 datalen 字节到 buf 的末尾，成功时返回0，失败时返回-1。

接口

```
int evbuffer_add_printf(struct evbuffer *buf, const char *fmt, ...)
int evbuffer_add_vprintf(struct evbuffer *buf, const char *fmt, va_list ap);
```

这些函数添加格式化的数据到 buf 末尾。格式参数和其他参数的处理分别与 C 库函数 printf 和 vprintf 相同。函数返回添加的字节数。

接口

```
int evbuffer_expand(struct evbuffer *buf, size_t datlen);
```

这个函数修改缓冲区的最后一块，或者添加一个新的块，使得缓冲区足以容纳 datlen 字节，而不需要更多的内存分配。

示例

```
/* Here are two ways to add "Hello world 2.0.1" to a buffer. */
/* Directly: */
evbuffer_add(buf, "Hello world 2.0.1", 17);

/* Via printf: */
evbuffer_add_printf(buf, "Hello %s %d.%d.%d", "world", 2, 0, 1);
```

evbuffer_add()和 evbuffer_add_printf()函数在 libevent 0.8版本引入；evbuffer_expand()首次出现在0.9版本，而 evbuffer_add_printf()首次出现在1.1版本。

5. 将数据从一个 evbuffer 移动到另一个

为提高效率，libevent 具有将数据从一个 evbuffer 移动到另一个的优化函数。

接口

```
int evbuffer_add_buffer(struct evbuffer *dst, struct evbuffer *src);
int evbuffer_remove_buffer(struct evbuffer *src, struct evbuffer *dst,
    size_t datlen);
```

evbuffer_add_buffer()将 src 中的所有数据移动到 dst 末尾，成功时返回0，失败时返回-1。

evbuffer_remove_buffer()函数从 src 中移动 datlen 字节到 dst 末尾，尽量少进行复制。如果字节数小于 datlen，所有字节被移动。函数返回移动的字节数。

evbuffer_add_buffer()在0.8版本引入；evbuffer_remove_buffer()是2.0.1-alpha 版本新增加的。

6. 添加数据到 evbuffer 前面

接口

```
int evbuffer_prepend(struct evbuffer *buf, const void *data, size_t size);
int evbuffer_prepend_buffer(struct evbuffer *dst, struct evbuffer* src);
```

除了将数据移动到目标缓冲区前面之外，这两个函数的行为分别与 evbuffer_add()和 evbuffer_add_buffer()相同。

使用这些函数时要当心，永远不要对与 bufferevent 共享的 evbuffer 使用。这些函数是2.0.1-alpha 版本新添加的。

7. 重新排列 evbuffer 的内部布局

有时候需要取出 evbuffer 前面的 N 字节，将其看作连续的字节数组。要做到这一点，首先必须确保缓冲区的前面确实是连续的。

接口

```
unsigned char *evbuffer_pullup(struct evbuffer *buf, ev_ssize_t size);
```

evbuffer_pullup()函数“线性化”buf 前面的 size 字节，必要时将进行复制或者移动，以保证这些字节是连续的，占据相同的内存块。如果 size 是负的，函数会线性化整个缓冲区。如果 size 大于缓冲区中的字节数，函数返回 NULL。否则，evbuffer_pullup()返回指向 buf 中首字节的指针。

调用 evbuffer_pullup()时使用较大的 size 参数可能会非常慢，因为这可能需要复制整个缓冲区的内容。

示例

```
#include <event2/buffer.h>
#include <event2/util.h>

#include <string.h>

int parse_socks4(struct evbuffer *buf, ev_uint16_t *port, ev_uint32_t *addr)
{
    /* Let's parse the start of a SOCKS4 request!  The format is easy:
     * 1 byte of version, 1 byte of command, 2 bytes destport, 4 bytes of
     * destip. */
    unsigned char *mem;

    mem = evbuffer_pullup(buf, 8);

    if (mem == NULL) {
        /* Not enough data in the buffer */
        return 0;
    } else if (mem[0] != 4 || mem[1] != 1) {
        /* Unrecognized protocol or command */
        return -1;
    } else {
        memcpy(port, mem+2, 2);
        memcpy(addr, mem+4, 4);
        *port = ntohs(*port);
        *addr = ntohl(*addr);
        /* Actually remove the data from the buffer now that we know we
           like it. */
        evbuffer_drain(buf, 8);
        return 1;
    }
}
```

提示

使用 evbuffer_get_contiguous_space()返回的值作为尺寸值调用 evbuffer_pullup()不会导致任何数据复制或者移动。

evbuffer_pullup()函数由2.0.1-alpha 版本新增加：先前版本的 libevent 总是保证 evbuffer 中的数据是连续的，而不计开销。

8. 从 evbuffer 中移除数据

接口

```
int evbuffer_drain(struct evbuffer *buf, size_t len);
int evbuffer_remove(struct evbuffer *buf, void *data, size_t datlen);
```

evbuffer_remove（）函数从 buf 前面复制和移除 datlen 字节到 data 处的内存中。如果可用字节少于 datlen，函数复制所有字节。失败时返回-1，否则返回复制了的字节数。

evbuffer_drain（）函数的行为与 evbuffer_remove（）相同，只是它不进行数据复制：而只是将数据从缓冲区前面移除。成功时返回0，失败时返回-1。

evbuffer_drain（）由0.8版引入，evbuffer_remove（）首次出现在0.9版。

9. 从 evbuffer 中复制出数据

有时候需要获取缓冲区前面数据的副本，而不清除数据。比如说，可能需要查看某特定类型的记录是否已经完整到达，而不清除任何数据（像 evbuffer_remove 那样），或者在内部重新排列缓冲区（像 evbuffer_pullup那样）。

接口

```
ev_ssize_t evbuffer_copyout(struct evbuffer *buf, void *data, size_t datlen);
ev_ssize_t evbuffer_copyout_from(struct evbuffer *buf,
     const struct evbuffer_ptr *pos,
     void *data_out, size_t datlen);
```

evbuffer_copyout（）的行为与 evbuffer_remove（）相同，但是它不从缓冲区移除任何数据。也就是说，它从 buf 前面复制 datlen 字节到 data 处的内存中。如果可用字节少于 datlen，函数会复制所有字节。失败时返回-1，否则返回复制的字节数。

evbuffer_copyout_from（）函数的行为类似于evbuffer_copyout（），但不是从缓冲区的开头复制字节，而是从pos中提供的位置开始复制字节。 有关evbuffer_ptr结构的信息，请参见下面的“在evbuffer中搜索”。

如果从缓冲区复制数据太慢，可以使用 evbuffer_peek（）。

示例

```
#include <event2/buffer.h>
#include <event2/util.h>
#include <stdlib.h>
#include <stdlib.h>

int get_record(struct evbuffer *buf, size_t *size_out, char **record_out)
{
    /* Let's assume that we're speaking some protocol where records
       contain a 4-byte size field in network order, followed by that
       number of bytes.  We will return 1 and set the 'out' fields if we
       have a whole record, return 0 if the record isn't here yet, and
       -1 on error.  */
    size_t buffer_len = evbuffer_get_length(buf);
    ev_uint32_t record_len;
```

```
    char *record;

    if (buffer_len < 4)
       return 0; /* The size field hasn't arrived. */

   /* We use evbuffer_copyout here so that the size field will stay on
       the buffer for now. */
    evbuffer_copyout(buf, &record_len, 4);
    /* Convert len_buf into host order. */
    record_len = ntohl(record_len);
    if (buffer_len < record_len + 4)
        return 0; /* The record hasn't arrived */

    /* Okay, _now_ we can remove the record. */
    record = malloc(record_len);
    if (record == NULL)
        return -1;

    evbuffer_drain(buf, 4);
    evbuffer_remove(buf, record, record_len);

    *record_out = record;
    *size_out = record_len;
    return 1;
}
```

evbuffer_copyout（）函数首先出现在Libevent 2.0.5-alpha中； evbuffer_copyout_from（）已添加到Libevent 2.1.1-alpha中。

10. 面向行的输入

接口

```
enum evbuffer_eol_style {
        EVBUFFER_EOL_ANY,
        EVBUFFER_EOL_CRLF,
        EVBUFFER_EOL_CRLF_STRICT,
        EVBUFFER_EOL_LF,
        EVBUFFER_EOL_NUL
};
char *evbuffer_readln(struct evbuffer *buffer, size_t *n_read_out,
    enum evbuffer_eol_style eol_style);
```

很多互联网协议使用基于行的格式。evbuffer_readln()函数从 evbuffer 前面取出一行，用一个新分配的空字符结束的字符串返回这一行。如果 n_read_out 不是 NULL，则它被设置为返回的字符串的字节数。如果没有整行供读取，函数返回空。返回的字符串不包括行结束符。

evbuffer_readln()理解5种行结束格式：

- EVBUFFER_EOL_LF

  行尾是单个换行符（也就是\n，ASCII 值是0x0A）

- EVBUFFER_EOL_CRLF_STRICT

  行尾是一个回车符，后随一个换行符（也就是\r\n，ASCII 值是0x0D 0x0A）

- EVBUFFER_EOL_CRLF

行尾是一个可选的回车，后随一个换行符（也就是说，可以是\r\n 或者\n）。这种格式对于解析基于文本的互联网协议很有用，因为标准通常要求\r\n 的行结束符，而不遵循标准的客户端有时候只使用\n。

- EVBUFFER_EOL_ANY

  行尾是任意数量、任意次序的回车和换行符。这种格式不是特别有用。它的存在主要是为了向后兼容。

- EVBUFFER_EOL_NUL

  行尾是一个值为0的单个字节，即ASCII NUL。

> 注意，如果使用 event_se_mem_functions()覆盖默认的 malloc，则 evbuffer_readln 返回的字符串将由你指定的 malloc 替代函数分配

示例

```
char *request_line;
size_t len;

request_line = evbuffer_readln(buf, &len, EVBUFFER_EOL_CRLF);
if (!request_line) {
    /* The first line has not arrived yet. */
} else {
    if (!strncmp(request_line, "HTTP/1.0 ", 9)) {
        /* HTTP 1.0 detected ... */
    }
    free(request_line);
}
```

evbuffer_readln（）接口在Libevent 1.4.14-stable和更高版本中可用。 EVBUFFER_EOL_NUL已添加到Libevent 2.1.1-alpha中。

11. 在 evbuffer 中搜索

evbuffer_ptr 结构体指示 evbuffer 中的一个位置，包含可用于在 evbuffer 中迭代的数据。

接口

```
struct evbuffer_ptr {
        ev_ssize_t pos;
        struct {
                /* internal fields */
        } _internal;
};
```

pos 是唯一的公有字段，用户代码不应该使用其他字段。pos 指示 evbuffer 中的一个位置，以到开始处的偏移量表示。

接口

```
struct evbuffer_ptr evbuffer_search(struct evbuffer *buffer,
    const char *what, size_t len, const struct evbuffer_ptr *start);
struct evbuffer_ptr evbuffer_search_range(struct evbuffer *buffer,
    const char *what, size_t len, const struct evbuffer_ptr *start,
    const struct evbuffer_ptr *end);
struct evbuffer_ptr evbuffer_search_eol(struct evbuffer *buffer,
    struct evbuffer_ptr *start, size_t *eol_len_out,
    enum evbuffer_eol_style eol_style);
```

evbuffer_search()函数在缓冲区中查找含有 len 个字符的字符串 what。函数返回包含字符串位置，或者在没有找到字符串时包含-1的 evbuffer_ptr 结构体。如果提供了 start 参数，则 从指定的位置开始搜索；否则，从开始处进行搜索。

evbuffer_search_range()函数和 evbuffer_search 行为相同，只是它只考虑在 end 之前出现的 what。

evbuffer_search_eol()函数像 evbuffer_readln()一样检测行结束，但是不复制行，而是返回指向行结束符的 evbuffer_ptr。如果 eol_len_out 非空，则它被设置为 EOL 字符串长度。

接口

```
enum evbuffer_ptr_how {
        EVBUFFER_PTR_SET,
        EVBUFFER_PTR_ADD
};
int evbuffer_ptr_set(struct evbuffer *buffer, struct evbuffer_ptr *pos,
    size_t position, enum evbuffer_ptr_how how);
```

evbuffer_ptr_set函数操作 buffer 中的位置 pos。如果 how 等于 EVBUFFER_PTR_SET,指针被移动到缓冲区中的绝对位置 position；如果等于 EVBUFFER_PTR_ADD，则向前移动position 字节。成功时函数返回0，失败时返回-1。

示例

```
#include <event2/buffer.h>
#include <string.h>

/* Count the total occurrences of 'str' in 'buf'. */
int count_instances(struct evbuffer *buf, const char *str)
{
    size_t len = strlen(str);
    int total = 0;
    struct evbuffer_ptr p;

    if (!len)
        /* Don't try to count the occurrences of a 0-length string. */
        return -1;

    evbuffer_ptr_set(buf, &p, 0, EVBUFFER_PTR_SET);

    while (1) {
         p = evbuffer_search(buf, str, len, &p);
         if (p.pos < 0)
             break;
         total++;
         evbuffer_ptr_set(buf, &p, 1, EVBUFFER_PTR_ADD);
    }
```

```
    return total;
}
```

警告

任何修改 evbuffer 或者其布局的调用都会使得 evbuffer_ptr 失效，不能再安全地使用。这些接口是 2.0.1-alpha 版本新增加的。

12. 检测数据而不复制

有时候需要读取 evbuffer 中的数据而不进行复制（像 evbuffer_copyout()那样），也不重新排列内部内存布局（像 evbuffer_pullup()那样）。有时候可能需要查看 evbuffer 中间的数据。

接口

```
struct evbuffer_iovec {
        void *iov_base;
        size_t iov_len;
};

int evbuffer_peek(struct evbuffer *buffer, ev_ssize_t len,
    struct evbuffer_ptr *start_at,
    struct evbuffer_iovec *vec_out, int n_vec);
```

调用 evbuffer_peek()的时候，通过 vec_out 给定一个 evbuffer_iovec 数组，数组的长度是n_vec。函数会让每个结构体包含指向 evbuffer 内部内存块的指针（iov_base)和块中数据长度。

如果 len 小于0，evbuffer_peek()会试图填充所有 evbuffer_iovec 结构体。否则，函数会进行填充，直到使用了所有结构体，或者见到 len 字节为止。如果函数可以给出所有请求的数据，则返回实际使用的结构体个数；否则，函数返回给出所有请求数据所需的结构体个数。

如果 ptr 为 NULL，函数从缓冲区开始处进行搜索。否则，从 ptr 处开始搜索。

示例

```
{
    /* Let's look at the first two chunks of buf, and write them to stderr. */
    int n, i;
    struct evbuffer_iovec v[2];
    n = evbuffer_peek(buf, -1, NULL, v, 2);
    for (i=0; i<n; ++i) { /* There might be less than two chunks available. */
        fwrite(v[i].iov_base, 1, v[i].iov_len, stderr);
    }
}

{
    /* Let's send the first 4906 bytes to stdout via write. */
    int n, i, r;
    struct evbuffer_iovec *v;
    size_t written = 0;

    /* determine how many chunks we need. */
    n = evbuffer_peek(buf, 4096, NULL, NULL, 0);
    /* Allocate space for the chunks.  This would be a good time to use
       alloca() if you have it. */
    v = malloc(sizeof(struct evbuffer_iovec)*n);
    /* Actually fill up v. */
```

```
    n = evbuffer_peek(buf, 4096, NULL, v, n);
    for (i=0; i<n; ++i) {
        size_t len = v[i].iov_len;
        if (written + len > 4096)
            len = 4096 - written;
        r = write(1 /* stdout */, v[i].iov_base, len);
        if (r<=0)
            break;
        /* We keep track of the bytes written separately; if we don't,
           we may write more than 4096 bytes if the last chunk puts
           us over the limit. */
        written += len;
    }
    free(v);
}

{
    /* Let's get the first 16K of data after the first occurrence of the
       string "start\n", and pass it to a consume() function. */
    struct evbuffer_ptr ptr;
    struct evbuffer_iovec v[1];
    const char s[] = "start\n";
    int n_written;

    ptr = evbuffer_search(buf, s, strlen(s), NULL);
    if (ptr.pos == -1)
        return; /* no start string found. */

    /* Advance the pointer past the start string. */
    if (evbuffer_ptr_set(buf, &ptr, strlen(s), EVBUFFER_PTR_ADD) < 0)
        return; /* off the end of the string. */

    while (n_written < 16*1024) {
        /* Peek at a single chunk. */
        if (evbuffer_peek(buf, -1, &ptr, v, 1) < 1)
            break;
        /* Pass the data to some user-defined consume function */
        consume(v[0].iov_base, v[0].iov_len);
        n_written += v[0].iov_len;

        /* Advance the pointer so we see the next chunk next time. */
        if (evbuffer_ptr_set(buf, &ptr, v[0].iov_len, EVBUFFER_PTR_ADD)<0)
            break;
    }
}
```

注意

- 修改 evbuffer_iovec 所指的数据会导致不确定的行为
- 如果任何函数修改了 evbuffer，则 evbuffer_peek()返回的指针会失效
- 如果在多个线程中使用 evbuffer ，确保在调用evbuffer_peek()之前使用evbuffer_lock()，在使用完 evbuffer_peek()给出的内容之后进行解锁

这个函数是2.0.2-alpha 版本新增加的。

13. 直接向 evbuffer 添加数据

有时候需要能够直接向 evbuffer 添加数据，而不用先将数据写入到字符数组中，然后再使用 evbuffer_add()进行复制。有一对高级函数可以完成这种功能：evbuffer_reserve_space() 和 evbuffer_commit_space()。跟 evbuffer_peek()一样，这两个函数使用 evbuffer_iovec 结构体来提供对 evbuffer 内部内存的直接访问。

接口

```
int evbuffer_reserve_space(struct evbuffer *buf, ev_ssize_t size,
    struct evbuffer_iovec *vec, int n_vecs);
int evbuffer_commit_space(struct evbuffer *buf,
    struct evbuffer_iovec *vec, int n_vecs);
```

evbuffer_reserve_space()函数给出 evbuffer 内部空间的指针。函数会扩展缓冲区以至少提供 size 字节的空间。到扩展空间的指针，以及其长度，会存储在通过 vec 传递的向量数组中，n_vec 是数组的长度。

n_vec 的值必须至少是1。如果只提供一个向量，libevent 会确保请求的所有连续空间都在 单个扩展区中，但是这可能要求重新排列缓冲区，或者浪费内存。为取得更好的性能，应该至少提供2个向量。函数返回提供请求的空间所需的向量数。

写入到向量中的数据不会是缓冲区的一部分，直到调用 evbuffer_commit_space()，使得写入的数据进入缓冲区。如果需要提交少于请求的空间，可以减小任何 evbuffer_iovec 结构体的 iov_len 字段，也可以提供较少的向量。函数成功时返回0，失败时返回-1。

提示和警告

- 调用任何重新排列 evbuffer 或者向其添加数据的函数都将使从evbuffer_reserve_space()获取的指针失效。
- 当前实现中，不论用户提供多少个向量，evbuffer_reserve_space()从不使用多于两个。未来版本可能会改变这一点。
- 如果在多个线程中使用 evbuffer，确保在调用 evbuffer_reserve_space()之前使用evbuffer_lock()进行锁定，然后在提交后解除锁定。

示例

```
/* Suppose we want to fill a buffer with 2048 bytes of output from a
   generate_data() function, without copying. */
struct evbuffer_iovec v[2];
int n, i;
size_t n_to_add = 2048;

/* Reserve 2048 bytes.*/
n = evbuffer_reserve_space(buf, n_to_add, v, 2);
if (n<=0)
   return; /* Unable to reserve the space for some reason. */

for (i=0; i<n && n_to_add > 0; ++i) {
   size_t len = v[i].iov_len;
   if (len > n_to_add) /* Don't write more than n_to_add bytes. */
      len = n_to_add;
   if (generate_data(v[i].iov_base, len) < 0) {
      /* If there was a problem during data generation, we can just stop
         here; no data will be committed to the buffer. */
      return;
   }
   /* Set iov_len to the number of bytes we actually wrote, so we
```

```
      don't commit too much. */
   v[i].iov_len = len;
}

/* We commit the space here.  Note that we give it 'i' (the number of
   vectors we actually used) rather than 'n' (the number of vectors we
   had available. */
if (evbuffer_commit_space(buf, v, i) < 0)
   return; /* Error committing */
```

不好的示例

```
/* Here are some mistakes you can make with evbuffer_reserve().
   DO NOT IMITATE THIS CODE. */
struct evbuffer_iovec v[2];

{
  /* Do not use the pointers from evbuffer_reserve_space() after
     calling any functions that modify the buffer. */
  evbuffer_reserve_space(buf, 1024, v, 2);
  evbuffer_add(buf, "X", 1);
  /* WRONG: This next line won't work if evbuffer_add needed to rearrange
     the buffer's contents.  It might even crash your program. Instead,
     you add the data before calling evbuffer_reserve_space. */
  memset(v[0].iov_base, 'Y', v[0].iov_len-1);
  evbuffer_commit_space(buf, v, 1);
}

{
  /* Do not modify the iov_base pointers. */
  const char *data = "Here is some data";
  evbuffer_reserve_space(buf, strlen(data), v, 1);
  /* WRONG: The next line will not do what you want.  Instead, you
     should _copy_ the contents of data into v[0].iov_base. */
  v[0].iov_base = (char*) data;
  v[0].iov_len = strlen(data);
  /* In this case, evbuffer_commit_space might give an error if you're
     lucky */
  evbuffer_commit_space(buf, v, 1);
}
```

这个函数及其提出的接口从2.0.2-alpha 版本就存在了。

14. 使用 evbuffer 的网络 IO

libevent 中 evbuffer 的最常见使用场合是网络 IO。将 evbuffer 用于网络 IO 的接口是：

接口

```
int evbuffer_write(struct evbuffer *buffer, evutil_socket_t fd);
int evbuffer_write_atmost(struct evbuffer *buffer, evutil_socket_t fd,
        ev_ssize_t howmuch);
int evbuffer_read(struct evbuffer *buffer, evutil_socket_t fd, int howmuch);
```

evbuffer_read()函数从套接字 fd 读取至多 howmuch 字节到 buffer 末尾。成功时函数返回读取的字节数，0表示 EOF，失败时返回-1。注意，错误码可能指示非阻塞操作不能立即成功，应该检查错误码 EAGAIN（或者 Windows 中的 WSAWOULDBLOCK）。如果 howmuch 为负，evbuffer_read()试图猜测要读取多少数据。

evbuffer_write_atmost()函数试图将 buffer 前面至多 howmuch 字节写入到套接字 fd 中。成功时函数返回写入的字节数，失败时返回-1。跟 evbuffer_read()一样，应该检查错误码，看是真的错误，还是仅仅指示非阻塞 IO 不能立即完成。如果为 howmuch 给出负值，函数会试图写入 buffer 的所有内容。

调用 evbuffer_write()与使用负的 howmuch 参数调用 evbuffer_write_atmost()一样：函数会试图尽量清空 buffer 的内容。

在 Unix 中，这些函数应该可以在任何支持 read 和 write 的文件描述符上正确工作。在 Windows 中，仅仅支持套接字。

注意，如果使用 bufferevent，则不需要调用这些函数，bufferevent 的代码已经为你调用了。

evbuffer_write_atmost()函数在2.0.1-alpha 版本中引入。

15. evbuffer 和回调

evbuffer 的用户常常需要知道什么时候向 evbuffer 添加了数据，什么时候移除了数据。为支持这个，libevent 为 evbuffer 提高了通用回调机制。

接口

```
struct evbuffer_cb_info {
        size_t orig_size;
        size_t n_added;
        size_t n_deleted;
};

typedef void (*evbuffer_cb_func)(struct evbuffer *buffer,
    const struct evbuffer_cb_info *info, void *arg);
```

向 evbuffer 添加数据，或者从中移除数据的时候，回调函数会被调用。函数收到缓冲区指针、一个 evbuffer_cb_info 结构体指针，和用户提供的参数。evbuffer_cb_info 结构体的orig_size 字段指示缓冲区改变大小前的字节数，n_added 字段指示向缓冲区添加了多少字节；n_deleted 字段指示移除了多少字节。

接口

```
struct evbuffer_cb_entry;
struct evbuffer_cb_entry *evbuffer_add_cb(struct evbuffer *buffer,
    evbuffer_cb_func cb, void *cbarg);
```

evbuffer_add_cb()函数为 evbuffer 添加一个回调函数，返回一个不透明的指针，随后可用于代表这个特定的回调实例。cb 参数是将被调用的函数，cbarg 是用户提供的将传给这个函数的指针。

可以为单个 evbuffer 设置多个回调，添加新的回调不会移除原来的回调。

示例

```
#include <event2/buffer.h>
#include <stdio.h>
#include <stdlib.h>

/* Here's a callback that remembers how many bytes we have drained in
```

```
   total from the buffer, and prints a dot every time we hit a
   megabyte. */
struct total_processed {
    size_t n;
};
void count_megabytes_cb(struct evbuffer *buffer,
    const struct evbuffer_cb_info *info, void *arg)
{
    struct total_processed *tp = arg;
    size_t old_n = tp->n;
    int megabytes, i;
    tp->n += info->n_deleted;
    megabytes = ((tp->n) >> 20) - (old_n >> 20);
    for (i=0; i<megabytes; ++i)
        putc('.', stdout);
}

void operation_with_counted_bytes(void)
{
    struct total_processed *tp = malloc(sizeof(*tp));
    struct evbuffer *buf = evbuffer_new();
    tp->n = 0;
    evbuffer_add_cb(buf, count_megabytes_cb, tp);

    /* Use the evbuffer for a while.  When we're done: */
    evbuffer_free(buf);
    free(tp);
}
```

注意：释放非空 evbuffer 不会清空其数据，释放 evbuffer 也不会为回调释放用户提供的数据指针。

如果不想让缓冲区上的回调永远激活，可以移除或者禁用回调：

接口

```
int evbuffer_remove_cb_entry(struct evbuffer *buffer,
    struct evbuffer_cb_entry *ent);
int evbuffer_remove_cb(struct evbuffer *buffer, evbuffer_cb_func cb,
    void *cbarg);

#define EVBUFFER_CB_ENABLED 1
int evbuffer_cb_set_flags(struct evbuffer *buffer,
                          struct evbuffer_cb_entry *cb,
                          ev_uint32_t flags);
int evbuffer_cb_clear_flags(struct evbuffer *buffer,
                          struct evbuffer_cb_entry *cb,
                          ev_uint32_t flags);
```

可以通过添加回调时候的 evbuffer_cb_entry 来移除回调，也可以通过回调函数和参数指针来移除。成功时函数返回0，失败时返回-1。

evbuffer_cb_set_flags()和 evbuffer_cb_clear_flags()函数分别为回调函数设置或者清除给定的标志。当前只有一个标志是用户可见的：EVBUFFER_CB_ENABLED。这个标志默认是打开的。如果清除这个标志，对 evbuffer 的修改不会调用回调函数。

接口

```
int evbuffer_defer_callbacks(struct evbuffer *buffer, struct event_base *base);
```

跟 bufferevent 回调一样，可以让 evbuffer 回调不在 evbuffer 被修改时立即运行，而是延迟到某 event_base 的事件循环中执行。如果有多个 evbuffer，它们的回调潜在地让数据添加到 evbuffer 中，或者从中移除，又要避免栈崩溃，延迟回调是很有用的。

如果回调被延迟，则最终执行时，它可能是多个操作结果的总和。

与 bufferevent 一样，evbuffer 具有内部引用计数的，所以即使还有未执行的延迟回调，释放 evbuffer 也是安全的。

整个回调系统是 2.0.1-alpha 版本新引入的。 evbuffer_cb*(set|clear)*flags() 函数从2.0.2-alpha 版本开始存在。

16. 为基于 evbuffer 的 IO 避免数据复制

真正高速的网络编程通常要求尽量少的数据复制，libevent 为此提供了一些机制：

接口

```
typedef void (*evbuffer_ref_cleanup_cb)(const void *data,
    size_t datalen, void *extra);

int evbuffer_add_reference(struct evbuffer *outbuf,
    const void *data, size_t datlen,
    evbuffer_ref_cleanup_cb cleanupfn, void *extra);
```

这个函数通过引用向 evbuffer 末尾添加一段数据。不会进行复制：evbuffer 只会存储一个到data 处的 datlen 字节的指针。因此，在 evbuffer 使用这个指针期间，必须保持指针是有效的。evbuffer 会在不再需要这部分数据的时候调用用户提供的 cleanupfn 函数，带有提供的data 指针、datlen 值和 extra 指针参数。函数成功时返回0，失败时返回-1。

示例

```
#include <event2/buffer.h>
#include <stdlib.h>
#include <string.h>

/* In this example, we have a bunch of evbuffers that we want to use to
   spool a one-megabyte resource out to the network.  We do this
   without keeping any more copies of the resource in memory than
   necessary. */

#define HUGE_RESOURCE_SIZE (1024*1024)
struct huge_resource {
    /* We keep a count of the references that exist to this structure,
       so that we know when we can free it. */
    int reference_count;
    char data[HUGE_RESOURCE_SIZE];
};

struct huge_resource *new_resource(void) {
    struct huge_resource *hr = malloc(sizeof(struct huge_resource));
    hr->reference_count = 1;
    /* Here we should fill hr->data with something.  In real life,
       we'd probably load something or do a complex calculation.
       Here, we'll just fill it with EEs. */
```

```
    memset(hr->data, 0xEE, sizeof(hr->data));
    return hr;
}

void free_resource(struct huge_resource *hr) {
    --hr->reference_count;
    if (hr->reference_count == 0)
        free(hr);
}

static void cleanup(const void *data, size_t len, void *arg) {
    free_resource(arg);
}

/* This is the function that actually adds the resource to the
   buffer. */
void spool_resource_to_evbuffer(struct evbuffer *buf,
    struct huge_resource *hr)
{
    ++hr->reference_count;
    evbuffer_add_reference(buf, hr->data, HUGE_RESOURCE_SIZE,
        cleanup, hr);
}
```

一些操作系统提供了将文件写入到网络，而不需要将数据复制到用户空间的方法。如果存在，可以使用下述接口访问这种机制：

17. 将文件添加到evbuffer

一些操作系统提供了无需将数据复制到用户空间即可将文件写入网络的方法。 您可以使用简单的界面访问以下机制：

接口

```
int evbuffer_add_file(struct evbuffer *output, int fd, ev_off_t offset,
    size_t length);
```

evbuffer_add_file()要求一个打开的可读文件描述符 fd（注意：不是套接字）。函数将文件中offset 处开始的length 字节添加到 output 末尾。成功时函数返回0，失败时返回-1。

在2.0.2-alpha 版中，对于使用这种方式添加的数据的可靠操作只有：通过 evbuffer_write()*将其发送到网络、使用 evbuffer_drain()清空数据，或者使用 evbuffer*__buffer()将其移动到另一个 evbuffer 中。不能使用evbuffer_remove()取出数据，使用 evbuffer_pullup()进行线性化等。

如果操作系统支持 splice()或者 sendfile()，则调用 evbuffer_write()时 libevent 会直接使用这些函数来将来自 fd 的数据发送到网络中，而根本不将数据复制到用户内存中。如果不存在splice()和 sendfile()，但是支持 mmap()，libevent 将进行文件映射，而内核将意识到永远不需要将数据复制到用户空间。否则，libevent 会将数据从磁盘读取到内存。

从evbuffer刷新数据后或释放evbuffer时，文件描述符将关闭。 如果这不是想要的，或者想要对文件进行更细粒度的控制，请参见下面的file_segment功能。

此功能在Libevent 2.0.1-alpha中引入。

18. 带有文件段的细粒度控制

evbuffer_add_file（）接口无法多次添加同一文件，因为它拥有文件所有权。

接口

```
struct evbuffer_file_segment;

struct evbuffer_file_segment *evbuffer_file_segment_new(
        int fd, ev_off_t offset, ev_off_t length, unsigned flags);
void evbuffer_file_segment_free(struct evbuffer_file_segment *seg);
int evbuffer_add_file_segment(struct evbuffer *buf,
    struct evbuffer_file_segment *seg, ev_off_t offset, ev_off_t length);
```

evbuffer_file_segment_new（）函数创建并返回一个新的evbuffer_file_segment对象，以表示存储在fd中的基础文件的一部分，该文件从offset开始并包含长度字节。 错误时，它将返回NULL。

文件段可以根据需要使用sendfile，splice，mmap，CreateFileMapping或malloc（）和read（）来实现。 它们是使用受支持程度最轻的机制创建的，并根据需要过渡到较重的机制。 （例如，如果操作系统支持sendfile和mmap，则可以仅使用sendfile来实现文件段，直到尝试实际检查其内容为止。此时，需要对它进行mmap（）处理。）可以控制 具有以下标志的文件段的细粒度行为：

- EVBUF_FS_CLOSE_ON_FREE
  如果设置了此标志，则使用evbuffer_file_segment_free（）释放文件段将关闭基础文件。
- EVBUF_FS_DISABLE_MMAP
  如果设置了此标志，则file_segment将永远不会对此文件使用映射内存样式的后端（CreateFileMapping，mmap），即使这样做比较合适。
- EVBUF_FS_DISABLE_SENDFILE
  如果设置了此标志，则file_segment将永远不会为此文件使用sendfile样式的后端（sendfile，splice），即使那是适当的。
- EVBUF_FS_DISABLE_LOCKING
  如果设置了此标志，则不会为文件段分配任何锁：以任何方式被多个线程看到的方式使用它都是不安全的。

有了evbuffer_file_segment后，可以使用evbuffer_add_file_segment（）将部分或全部添加到evbuffer中。 这里的offset参数是指文件段内的偏移量，而不是文件本身内的偏移量。

当不再希望使用文件段时，可以使用evbuffer_file_segment_free（）释放它。 直到没有evbuffer不再保存对文件段的引用之前，才会释放实际的存储空间。

接口

```
typedef void (*evbuffer_file_segment_cleanup_cb)(
    struct evbuffer_file_segment const *seg, int flags, void *arg);

void evbuffer_file_segment_add_cleanup_cb(struct evbuffer_file_segment *seg,
        evbuffer_file_segment_cleanup_cb cb, void *arg);
```

可以在文件段中添加一个回调函数，当释放对该文件段的最终引用并且该文件段即将被释放时，将调用该函数。 此回调不得尝试重新激活文件段，将其添加到任何缓冲区等。

这些文件段功能首先出现在Libevent 2.1.1-alpha中； evbuffer_file_segment_add_cleanup_cb（）已添加到2.1.2-alpha中。

19. 通过引用将evbuffer添加到另一个evbuffer

可以通过引用将一个evbuffer添加到另一个evbuffer：不必删除一个缓冲区的内容并将其添加到另一个缓冲区，而是给一个evbuffer引用另一个缓冲区，并且它的行为就好像已复制了所有字节。

接口

```
int evbuffer_add_buffer_reference(struct evbuffer *outbuf,
    struct evbuffer *inbuf);
```

evbuffer_add_buffer_reference（）函数的行为就像已将所有数据从outbuf复制到inbuf一样，但是不执行任何不必要的复制。 如果成功则返回0，失败则返回-1。

请注意，对inbuf内容的后续更改不会反映在outbuf中：此函数通过引用而不是evbuffer本身添加evbuffer的当前内容。

还要注意，不能嵌套缓冲区引用：已经是一个evbuffer_add_buffer_reference调用的缓冲区的缓冲区不能是另一个evbuffer_add_buffer_reference调用的缓冲区。

此功能在Libevent 2.1.1-alpha中引入。

20. 让 evbuffer 只能添加或者只能移除

接口

```
int evbuffer_freeze(struct evbuffer *buf, int at_front);
int evbuffer_unfreeze(struct evbuffer *buf, int at_front);
```

可以使用这些函数暂时禁止修改 evbuffer 的开头或者末尾。bufferevent 的代码在内部使用这些函数阻止对输出缓冲区头部，或者输入缓冲区尾部的意外修改。

evbuffer_freeze()函数是2.0.1-alpha 版本引入的。

八、连接监听器：接受TCP连接

evconnlistener 机制提供了监听和接受 TCP 连接的方法。

本章的所有函数和类型都在 event2/listener.h 中声明，除非特别说明，它们都在2.0.2-alpha 版本中首次出现。

1. 创建和释放 evconnlistener

接口

```
struct evconnlistener *evconnlistener_new(struct event_base *base,
    evconnlistener_cb cb, void *ptr, unsigned flags, int backlog,
    evutil_socket_t fd);
struct evconnlistener *evconnlistener_new_bind(struct event_base *base,
    evconnlistener_cb cb, void *ptr, unsigned flags, int backlog,
    const struct sockaddr *sa, int socklen);
void evconnlistener_free(struct evconnlistener *lev);
```

两个 evconnlistener_new*()函数都分配和返回一个新的连接监听器对象。连接监听器使用event_base来得知什么时候在给定的监听套接字上有新的 TCP 连接。新连接到达时，监听器调用你给出的回调函数。

两个函数中，base 参数都是监听器用于监听连接的 event_base。cb 是收到新连接时要调用的回调函数；如果 cb 为 NULL，则监听器是禁用的，直到设置了回调函数为止。ptr 指针将传递给回调函数。flags 参数控制回调函数的行为，下面会更详细论述。backlog 是任何时刻网络栈允许处于还未接受状态的最大未决连接数。更多细节请查看系统的 listen()函数文档。如果 backlog 是负的，libevent 会试图挑选一个较好的值；如果为0，libevent 认为已 经对提供的套接字调用了 listen()。

两个函数的不同在于如何建立监听套接字。evconnlistener_new()函数假定已经将套接字绑定到要监听的端口，然后通过 fd 传入这个套接字。如果要 libevent 分配和绑定套接字，可以调用 evconnlistener_new_bind()，传输要绑定到的地址和地址长度。

> 提示: 使用evconnlistener_new时，通过使用evutil_make_socket_nonblocking或手动设置正确的套接字选项，确保侦听套接字处于非阻止模式。 当侦听套接字处于阻塞模式时，可能会发生未定义的行为。

要释放连接监听器，调用 evconnlistener_free()。

可识别的标志

可以给 evconnlistener_new()函数的 flags 参数传入一些标志。可以用或(OR)运算任意连接

下述标志：

- LEV_OPT_LEAVE_SOCKETS_BLOCKING

  默认情况下，连接监听器接收新套接字后，会将其设置为非阻塞的，以便将其用于 libevent。如果不想要这种行为，可以设置这个标志。

- LEV_OPT_CLOSE_ON_FREE

  如果设置了这个选项，释放连接监听器会关闭底层套接字。

- LEV_OPT_CLOSE_ON_EXEC

  如果设置了这个选项，连接监听器会为底层套接字设置 close-on-exec 标志。更多信息请查看 fcntl 和 FD_CLOEXEC 的平台文档。

- LEV_OPT_REUSEABLE

  某些平台在默认情况下，关闭某监听套接字后，要过一会儿其他套接字才可以绑定到同一个端口。设置这个标志会让 libevent 标记套接字是可重用的，这样一旦关闭，可以立即打开其他套接字，在相同端口进行监听。

- LEV_OPT_THREADSAFE

  为监听器分配锁，这样就可以在多个线程中安全地使用了。这是2.0.8-rc 的新功能。

- LEV_OPT_DISABLED
  初始化监听器以将其禁用，而不是启用。 您可以使用evconnlistener_enable（）手动将其打开。 Libevent 2.1.1-alpha中的新功能。

- LEV_OPT_DEFERRED_ACCEPT
  如果可能的话，告诉内核在接收到套接字上的某些数据并且可以读取之前，不要宣布套接字已被接受。 如果您的协议不是从客户端传输数据开始的，则不要使用此选项，因为在这种情况下，此选项有时会导致内核从不告诉您有关连接的信息。 并非所有操作系统都支持此选项：在不支持的操作系统上，此选项无效。 Libevent 2.1.1-alpha中的新功能。

连接监听器回调

接口

```
typedef void (*evconnlistener_cb)(struct evconnlistener *listener,
    evutil_socket_t sock, struct sockaddr *addr, int len, void *ptr);
```

接收到新连接会调用提供的回调函数。listener 参数是接收连接的连接监听器。sock 参数是新接收的套接字 。 addr 和 len 参数是接收连接的地址和地址长度 。ptr是调用evconnlistener_new()时用户提供的指针。

2. 启用和禁用 evconnlistener

接口

```
int evconnlistener_disable(struct evconnlistener *lev);
int evconnlistener_enable(struct evconnlistener *lev);
```

这两个函数暂时禁止或者重新允许监听新连接。

3. 调整 evconnlistener 的回调函数

接口

```
void evconnlistener_set_cb(struct evconnlistener *lev,
    evconnlistener_cb cb, void *arg);
```

函数调整 evconnlistener 的回调函数和其参数。它是2.0.9-rc 版本引入的。

4. 检测 evconnlistener

接口

```
evutil_socket_t evconnlistener_get_fd(struct evconnlistener *lev);
struct event_base *evconnlistener_get_base(struct evconnlistener *lev);
```

这些函数分别返回监听器关联的套接字和 event_base。

evconnlistener_get_fd()函数首次出现在2.0.3-alpha 版本。

5. 侦测错误

可以设置一个一旦监听器上的 accept()调用失败就被调用的错误回调函数。对于一个不解决就会锁定进程的错误条件，这很重要。

接口

```
typedef void (*evconnlistener_errorcb)(struct evconnlistener *lis, void *ptr);
void evconnlistener_set_error_cb(struct evconnlistener *lev,
    evconnlistener_errorcb errorcb);
```

如果使用 evconnlistener_set_error_cb()为监听器设置了错误回调函数，则监听器发生错误时回调函数就会被调用。第一个参数是监听器，第二个参数是调用 evconnlistener_new() 时传入的 ptr。

这个函数在2.0.8-rc 版本引入。

6. 示例代码：回显服务器

示例

```
#include <event2/listener.h>
#include <event2/bufferevent.h>
#include <event2/buffer.h>

#include <arpa/inet.h>

#include <string.h>
#include <stdlib.h>
#include <stdio.h>
```

```c
#include <errno.h>

static void
echo_read_cb(struct bufferevent *bev, void *ctx)
{
        /* This callback is invoked when there is data to read on bev. */
        struct evbuffer *input = bufferevent_get_input(bev);
        struct evbuffer *output = bufferevent_get_output(bev);

        /* Copy all the data from the input buffer to the output buffer. */
        evbuffer_add_buffer(output, input);
}

static void
echo_event_cb(struct bufferevent *bev, short events, void *ctx)
{
        if (events & BEV_EVENT_ERROR)
                perror("Error from bufferevent");
        if (events & (BEV_EVENT_EOF | BEV_EVENT_ERROR)) {
                bufferevent_free(bev);
        }
}

static void
accept_conn_cb(struct evconnlistener *listener,
    evutil_socket_t fd, struct sockaddr *address, int socklen,
    void *ctx)
{
        /* We got a new connection! Set up a bufferevent for it. */
        struct event_base *base = evconnlistener_get_base(listener);
        struct bufferevent *bev = bufferevent_socket_new(
                base, fd, BEV_OPT_CLOSE_ON_FREE);

        bufferevent_setcb(bev, echo_read_cb, NULL, echo_event_cb, NULL);

        bufferevent_enable(bev, EV_READ|EV_WRITE);
}

static void
accept_error_cb(struct evconnlistener *listener, void *ctx)
{
        struct event_base *base = evconnlistener_get_base(listener);
        int err = EVUTIL_SOCKET_ERROR();
        fprintf(stderr, "Got an error %d (%s) on the listener. "
                "Shutting down.\n", err, evutil_socket_error_to_string(err));

        event_base_loopexit(base, NULL);
}

int
main(int argc, char **argv)
{
        struct event_base *base;
        struct evconnlistener *listener;
        struct sockaddr_in sin;

        int port = 9876;
```

```
        if (argc > 1) {
                port = atoi(argv[1]);
        }
        if (port<=0 || port>65535) {
                puts("Invalid port");
                return 1;
        }

        base = event_base_new();
        if (!base) {
                puts("Couldn't open event base");
                return 1;
        }

        /* Clear the sockaddr before using it, in case there are extra
         * platform-specific fields that can mess us up. */
        memset(&sin, 0, sizeof(sin));
        /* This is an INET address */
        sin.sin_family = AF_INET;
        /* Listen on 0.0.0.0 */
        sin.sin_addr.s_addr = htonl(0);
        /* Listen on the given port. */
        sin.sin_port = htons(port);

        listener = evconnlistener_new_bind(base, accept_conn_cb, NULL,
            LEV_OPT_CLOSE_ON_FREE|LEV_OPT_REUSEABLE, -1,
            (struct sockaddr*)&sin, sizeof(sin));
        if (!listener) {
                perror("Couldn't create listener");
                return 1;
        }
        evconnlistener_set_error_cb(listener, accept_error_cb);

        event_base_dispatch(base);
        return 0;
}
```

九、使用 libevent 的 DNS：高层和底层功能

libevent 提供了少量用于解析 DNS 名字的 API，以及用于实现简单 DNS 服务器的机制。

我们从用于名字查询的高层机制开始介绍，然后介绍底层机制和服务器机制。

> **注意**
> libevent 当前的 DNS 客户端实现存在限制：不支持 TCP 查询、DNSSec 以及任意记录类型。未来版本的 libevent 会修正这些限制。

1. 预备：可移植的阻塞式名字解析

为移植已经使用阻塞式名字解析的程序，libevent 提供了标准 getaddrinfo()接口的可移植实现。对于需要运行在没有 getaddrinfo() 函数，或者 getaddrinfo()不像我们的替代函数那样遵循标准的平台上的程序，这个替代实现很有用。

getaddrinfo()接口由 RFC 3493的6.1节定义。关于 libevent 如何不满足其一致性实现的概述，请看下面的“兼容性提示”节。

接口

```
struct evutil_addrinfo {
    int ai_flags;
    int ai_family;
    int ai_socktype;
    int ai_protocol;
    size_t ai_addrlen;
    char *ai_canonname;
    struct sockaddr *ai_addr;
    struct evutil_addrinfo *ai_next;
};

#define EVUTIL_AI_PASSIVE     /* ... */
#define EVUTIL_AI_CANONNAME   /* ... */
#define EVUTIL_AI_NUMERICHOST /* ... */
#define EVUTIL_AI_NUMERICSERV /* ... */
#define EVUTIL_AI_V4MAPPED    /* ... */
#define EVUTIL_AI_ALL         /* ... */
#define EVUTIL_AI_ADDRCONFIG  /* ... */

int evutil_getaddrinfo(const char *nodename, const char *servname,
    const struct evutil_addrinfo *hints, struct evutil_addrinfo **res);
void evutil_freeaddrinfo(struct evutil_addrinfo *ai);
const char *evutil_gai_strerror(int err);
```

evutil_getaddrinfo()函数试图根据 hints 给出的规则，解析指定的 nodename 和 servname，建立一个 evutil_addrinfo 结构体链表，将其存储在*res 中。成功时函数返回0，失败时返回非零的错误码。

必须至少提供 nodename 和 servname 中的一个。如果提供了 nodename，则它是 IPv4字面地址（如 127.0.0.1）、IPv6字面地址（如::1），或者是 DNS 名字（如 www.example.com）。

如果提供了 servname，则它是某网络服务的符号名（如 https），或者是一个包含十进制端口号的字符串（如443）。

如果不指定 servname，则 res 中的端口号将是零。如果不指定 nodename，则 res 中的地址要么是 localhost(默认），要么是“任意”（如果设置了 EVUTIL_AI_PASSIVE）。hints 的 ai_flags 字段指示 evutil_getaddrinfo 如何进行查询，它可以包含0个或者多个以或运算连接的下述标志：

- EVUTIL_AI_PASSIVE
  这个标志指示将地址用于监听，而不是连接。通常二者没有差别，除非 nodename 为空：对于连接，空的 nodename 表示 localhost（127.0.0.1或者::1）；而对于监听，空的 nodename表示任意（0.0.0.0或者::0）。
- EVUTIL_AI_CANONNAME
  如果设置了这个标志，则函数试图在 ai_canonname 字段中报告标准名称。
- EVUTIL_AI_NUMERICHOST
  如果设置了这个标志，函数仅仅解析数值类型的 IPv4和 IPv6地址；如果 nodename 要求名字查询，函数返回 EVUTIL_EAI_NONAME 错误。
- EVUTIL_AI_NUMERICSERV
  如果设置了这个标志，函数仅仅解析数值类型的服务名。如果 servname 不是空，也不是十进制整数，函数返回 EVUTIL_EAI_NONAME 错误。
- EVUTIL_AI_V4MAPPED
  这个标志表示，如果 ai_family 是 AF_INET6，但是找不到 IPv6地址，则应该以 v4映射(v4-mapped)型 IPv6地址的形式返回结果中的 IPv4地址。当前 evutil_getaddrinfo()不支持这个标志，除非操作系统支持它。

- EVUTIL_AI_ALL
  如果设置了这个标志和 EVUTIL_AI_V4MAPPED，则无论结果是否包含 IPv6地址，IPv4地址都应该以 v4映射型 IPv6地址的形式返回。当前 evutil_getaddrinfo()不支持这个标志，除非操作系统支持它。

- EVUTIL_AI_ADDRCONFIG
  如果设置了这个标志，则只有系统拥有非本地的 IPv4地址时，结果才包含 IPv4地址；只有系统拥有非本地的 IPv6地址时，结果才包含 IPv6地址。

hints 的 ai_family 字段指示 evutil_getaddrinfo()应该返回哪个地址。字段值可以是 AF_INET，表示只请求 IPv4地址；也可以是 AF_INET6，表示只请求 IPv6地址；或者用 AF_UNSPEC表示请求所有可用地址。

hints 的 ai_socktype 和 ai_protocol 字段告知 evutil_getaddrinfo()将如何使用返回的地址。这两个字段值的意义与传递给 socket()函数的 socktype 和 protocol 参数值相同。

成功时函数新建一个 evutil_addrinfo 结构体链表，存储在*res 中，链表的每个元素通过ai_next 指针指向下一个元素。因为链表是在堆上分配的，所以需要调用 evutil_freeaddrinfo()进行释放。

如果失败，函数返回数值型的错误码：

- EVUTIL_EAI_ADDRFAMILY
  请求的地址族对 nodename 没有意义。

- EVUTIL_EAI_AGAIN
  名字解析中发生可以恢复的错误，请稍后重试。

- EVUTIL_EAI_FAIL
  名字解析中发生不可恢复的错误：解析器或者 DNS 服务器可能已经崩溃。

- EVUTIL_EAI_BADFLAGS
  hints 中的 ai_flags 字段无效。

- EVUTIL_EAI_FAMILY
  不支持 hints 中的 ai_family 字段。

- EVUTIL_EAI_MEMORY
  回应请求的过程耗尽内存。

- EVUTIL_EAI_NODATA
  请求的主机不存在。

- EVUTIL_EAI_SERVICE
  请求的服务不存在。

- EVUTIL_EAI_SOCKTYPE
  不支持请求的套接字类型，或者套接字类型与 ai_protocol 不匹配。

- EVUTIL_EAI_SYSTEM
  名字解析中发生其他系统错误，更多信息请检查 errno。

- EVUTIL_EAI_CANCEL
  应用程序在解析完成前请求取消。evutil_getaddrinfo()函数从不产生这个错误，但是后面描述的 evdns_getaddrinfo()可能产生这个错误。

调用 evutil_gai_strerror()可以将上述错误值转化成描述性的字符串。

注意如果操作系统定义了 addrinfo 结构体，则 evutil_addrinfo 仅仅是操作系统内置的addrinfo 结构体的别名。类似地，如果操作系统定义了 AI*_标志，则相应的 EVUTIL_AI 标志仅仅是本地标志的别名；如果操作系统定义了 EAI_*_错误，则相应的 EVUTIL_EAI_*只是本地错误码的别名。

示例：解析主机名，建立阻塞的连接

```
#include <event2/util.h>

#include <sys/socket.h>
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <unistd.h>

evutil_socket_t
get_tcp_socket_for_host(const char *hostname, ev_uint16_t port)
{
    char port_buf[6];
    struct evutil_addrinfo hints;
    struct evutil_addrinfo *answer = NULL;
    int err;
    evutil_socket_t sock;

    /* Convert the port to decimal. */
    evutil_snprintf(port_buf, sizeof(port_buf), "%d", (int)port);

    /* Build the hints to tell getaddrinfo how to act. */
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC; /* v4 or v6 is fine. */
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP; /* We want a TCP socket */
    /* Only return addresses we can use. */
    hints.ai_flags = EVUTIL_AI_ADDRCONFIG;

    /* Look up the hostname. */
    err = evutil_getaddrinfo(hostname, port_buf, &hints, &answer);
    if (err != 0) {
          fprintf(stderr, "Error while resolving '%s': %s",
                  hostname, evutil_gai_strerror(err));
          return -1;
    }

    /* If there was no error, we should have at least one answer. */
    assert(answer);
    /* Just use the first answer. */
    sock = socket(answer->ai_family,
                  answer->ai_socktype,
                  answer->ai_protocol);
    if (sock < 0)
        return -1;
    if (connect(sock, answer->ai_addr, answer->ai_addrlen)) {
        /* Note that we're doing a blocking connect in this function.
         * If this were nonblocking, we'd need to treat some errors
         * (like EINTR and EAGAIN) specially. */
        EVUTIL_CLOSESOCKET(sock);
        return -1;
    }

    return sock;
}
```

上述函数和常量是2.0.3-alpha 版本新增加的，声明在 event2/util.h 中。

2. 使用 evdns_getaddrinfo()的非阻塞式名字解析

通常的 getaddrinfo()，以及上面的 evutil_getaddrinfo()的问题是，它们是阻塞的：调用线程必须等待函数查询 DNS 服务器，等待回应。对于 libevent，这可能不是期望的行为。对于非阻塞式应用，libevent 提供了一组函数用于启动 DNS 请求，让 libevent 等待服务器回应。

接口

```
typedef void (*evdns_getaddrinfo_cb)(
    int result, struct evutil_addrinfo *res, void *arg);
struct evdns_getaddrinfo_request;

struct evdns_getaddrinfo_request *evdns_getaddrinfo(
    struct evdns_base *dns_base,
    const char *nodename, const char *servname,
    const struct evutil_addrinfo *hints_in,
    evdns_getaddrinfo_cb cb, void *arg);

void evdns_getaddrinfo_cancel(struct evdns_getaddrinfo_request *req);
```

除了不会阻塞在 DNS 查询上，而是使用 libevent 的底层 DNS 机制进行查询外，evdns_getaddrinfo()和 evutil_getaddrinfo()是一样的。因为函数不是总能立即返回结果，所以需要提供一个 evdns_getaddrinfo_cb 类型的回调函数，以及一个给回调函数的可选的用户参数。

此外，调用 evdns_getaddrinfo()还要求一个 evdns_base 指针。evdns_base 结构体为libevent 的 DNS 解析器保持状态和配置。关于如何获取 evdns_base 指针，请看下一节。

如果失败或者立即成功，函数返回 NULL。否则，函数返回一个 evdns_getaddrinfo_request指针。在解析完成之前可以随时使用 evdns_getaddrinfo_cancel()和这个指针来取消解析。

注意：不论 evdns_getaddrinfo()是否返回 NULL，是否调用了 evdns_getaddrinfo_cancel()，回调函数总是会被调用。

evdns_getaddrinfo()内部会复制 nodename、servname 和 hints 参数，所以查询进行过程中不必保持这些参数有效。

示例：使用evdns_getaddrinfo()的非阻塞查询

```
#include <event2/dns.h>
#include <event2/util.h>
#include <event2/event.h>

#include <sys/socket.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

int n_pending_requests = 0;
struct event_base *base = NULL;

struct user_data {
    char *name; /* the name we're resolving */
    int idx; /* its position on the command line */
};
```

```
void callback(int errcode, struct evutil_addrinfo *addr, void *ptr)
{
    struct user_data *data = ptr;
    const char *name = data->name;
    if (errcode) {
        printf("%d. %s -> %s\n", data->idx, name, evutil_gai_strerror(errcode));
    } else {
        struct evutil_addrinfo *ai;
        printf("%d. %s", data->idx, name);
        if (addr->ai_canonname)
            printf(" [%s]", addr->ai_canonname);
        puts("");
        for (ai = addr; ai; ai = ai->ai_next) {
            char buf[128];
            const char *s = NULL;
            if (ai->ai_family == AF_INET) {
                struct sockaddr_in *sin = (struct sockaddr_in *)ai->ai_addr;
                s = evutil_inet_ntop(AF_INET, &sin->sin_addr, buf, 128);
            } else if (ai->ai_family == AF_INET6) {
                struct sockaddr_in6 *sin6 = (struct sockaddr_in6 *)ai->ai_addr;
                s = evutil_inet_ntop(AF_INET6, &sin6->sin6_addr, buf, 128);
            }
            if (s)
                printf("    -> %s\n", s);
        }
        evutil_freeaddrinfo(addr);
    }
    free(data->name);
    free(data);
    if (--n_pending_requests == 0)
        event_base_loopexit(base, NULL);
}

/* Take a list of domain names from the command line and resolve them in
 * parallel. */
int main(int argc, char **argv)
{
    int i;
    struct evdns_base *dnsbase;

    if (argc == 1) {
        puts("No addresses given.");
        return 0;
    }
    base = event_base_new();
    if (!base)
        return 1;
    dnsbase = evdns_base_new(base, 1);
    if (!dnsbase)
        return 2;

    for (i = 1; i < argc; ++i) {
        struct evutil_addrinfo hints;
        struct evdns_getaddrinfo_request *req;
        struct user_data *user_data;
        memset(&hints, 0, sizeof(hints));
        hints.ai_family = AF_UNSPEC;
```

```
        hints.ai_flags = EVUTIL_AI_CANONNAME;
        /* Unless we specify a socktype, we'll get at least two entries for
         * each address: one for TCP and one for UDP. That's not what we
         * want. */
        hints.ai_socktype = SOCK_STREAM;
        hints.ai_protocol = IPPROTO_TCP;

        if (!(user_data = malloc(sizeof(struct user_data)))) {
            perror("malloc");
            exit(1);
        }
        if (!(user_data->name = strdup(argv[i]))) {
            perror("strdup");
            exit(1);
        }
        user_data->idx = i;

        ++n_pending_requests;
        req = evdns_getaddrinfo(
                        dnsbase, argv[i], NULL /* no service name given */,
                        &hints, callback, user_data);
        if (req == NULL) {
          printf("    [request for %s returned immediately]\n", argv[i]);
          /* No need to free user_data or decrement n_pending_requests; that
           * happened in the callback. */
        }
    }

    if (n_pending_requests)
      event_base_dispatch(base);

    evdns_base_free(dnsbase, 0);
    event_base_free(base);

    return 0;
}
```

上述函数是2.0.3-alpha 版本新增加的，声明在 event2/dns.h 中。

3. 创建和配置 evdns_base

使用 evdns 进行非阻塞 DNS 查询之前需要配置一个 evdns_base。evdns_base 存储名字服务器列表和 DNS 配置选项，跟踪活动的、进行中的 DNS 请求。

接口

```
struct evdns_base *evdns_base_new(struct event_base *event_base,
       int initialize);
void evdns_base_free(struct evdns_base *base, int fail_requests);
```

成功时 evdns_base_new()返回一个新建的 evdns_base，失败时返回 NULL。如果 initialize参数为 true，函数试图根据操作系统的默认值配置 evdns_base；否则，函数让 evdns_base为空，不配置名字服务器和选项。

可以用 evdns_base_free()释放不再使用的 evdns_base。如果 fail_request 参数为 true，函数会在释放 evdns_base 前让所有进行中的请求使用取消错误码调用其回调函数。

3.1 使用系统配置初始化 evdns

如果需要更多地控制 evdns_base 如何初始化，可以为 evdns_base_new()的 initialize 参数传递0，然后调用下述函数。

接口

```
#define DNS_OPTION_SEARCH 1
#define DNS_OPTION_NAMESERVERS 2
#define DNS_OPTION_MISC 4
#define DNS_OPTION_HOSTSFILE 8
#define DNS_OPTIONS_ALL 15
int evdns_base_resolv_conf_parse(struct evdns_base *base, int flags,
                                 const char *filename);

#ifdef WIN32
int evdns_base_config_windows_nameservers(struct evdns_base *);
#define EVDNS_BASE_CONFIG_WINDOWS_NAMESERVERS_IMPLEMENTED
#endif
```

evdns_base_resolv_conf_parse()函数扫描 resolv.conf 格式的文件 filename，从中读取 flags 指示的选项（关于 resolv.conf 文件的更多信息，请看 Unix 手册）。

- DNS_OPTION_SEARCH
  请求从 resolv.conf 文件读取 domain 和 search 字段以及 ndots 选项，使用它们来确定使用哪个域（如果存在）来搜索不是全限定的主机名。
- DNS_OPTION_NAMESERVERS
  请求从 resolv.conf 中读取名字服务器地址。
- DNS_OPTION_MISC
  请求从 resolv.conf 文件中读取其他配置选项。
- DNS_OPTION_HOSTSFILE
  请求从/etc/hosts 文件读取主机列表。
- DNS_OPTION_ALL
  请求从 resolv.conf 文件获取尽量多的信息。

Windows 中没有可以告知名字服务器在哪里的 resolv.conf 文件，但可以用 evdns_base_config_windows_nameservers()函数从注册表（或者 NetworkParams，或者其他隐藏的地方）读取名字服务器。

3.2 resolv.conf 文件格式

resolv.conf 是一个文本文件，每一行要么是空行，要么包含以#开头的注释，要么由一个跟随零个或者多个参数的标记组成。可以识别的标记有：

- nameserver

  必须后随一个名字服务器的 IP 地址。作为一个扩展，libevent 允许使用 IP:Port 或者[IPv6]:port 语法为名字服务器指定非标准端口。
- domain

  本地域名
- search

解析本地主机名时要搜索的名字列表。如果不能正确解析任何含有少于“ndots”个点的本地名字，则在这些域名中进行搜索。比如说，如果“search”字段值为 example.com，“ndots” 为1，则用户请求解析“www”时，函数认为那是“www.example.com”。

- options

  空格分隔的选项列表。选项要么是空字符串，要么具有格式 option:value（如果有参数）。

  可识别的选项有：

  - ndots:INTEGER
    用于配置搜索，请参考上面的“search”，默认值是1。
  - timeout:FLOAT
    等待 DNS 服务器响应的时间，单位是秒。默认值为5秒。
  - max-timeouts:INT
    名字服务器响应超时几次才认为服务器当机？默认是3次。
  - max-inflight:INT
    最多允许多少个未决的 DNS 请求？（如果试图发出多于这么多个请求，则过多的请求将被延迟，直到某个请求被响应或者超时）。默认值是 XXX。
  - attempts:INT
    在放弃之前重新传输多少次 DNS 请求？默认值是 XXX。
  - randomize-case:INT
    如果非零，evdns 会为发出的 DNS 请求设置随机的事务 ID，并且确认回应具有同样的随机事务 ID 值。这种称作 “0x20 hack” 的机制可以在一定程度上阻止对 DNS 的简单激活事件攻击。这个选项的默认值是1。
  - bind-to:ADDRESS
    如果提供，则向名字服务器发送数据之前绑定到给出的地址。对于2.0.4-alpha 版本，这个设置仅应用于后面的名字服务器条目。
  - initial-probe-timeout:FLOAT
    确定名字服务器当机后，libevent 以指数级降低的频率探测服务器以判断服务器是否恢复。这个选项配置（探测时间间隔）序列中的第一个超时，单位是秒。默认值是10。
  - getaddrinfo-allow-skew:FLOAT
    同时请求IPv4和IPv6地址时，evdns_getaddrinfo()用单独的DNS请求包分别请求两种地址，因为有些服务器不能在一个包中同时处理两种请求。服务器回应一种地址类型后，函数等待一段时间确定另一种类型的地址是否到达。这个选项配置等待多长时间，单位是秒。默认值是3秒。

    不识别的字段和选项会被忽略。

3.3 手动配置 evdns

如果需要更精细地控制 evdns 的行为，可以使用下述函数：

接口

```
int evdns_base_nameserver_sockaddr_add(struct evdns_base *base,
                                       const struct sockaddr *sa, ev_socklen_t len,
                                       unsigned flags);
int evdns_base_nameserver_ip_add(struct evdns_base *base,
                                       const char *ip_as_string);
int evdns_base_load_hosts(struct evdns_base *base, const char *hosts_fname);

void evdns_base_search_clear(struct evdns_base *base);
void evdns_base_search_add(struct evdns_base *base, const char *domain);
void evdns_base_search_ndots_set(struct evdns_base *base, int ndots);
```

```
int evdns_base_set_option(struct evdns_base *base, const char *option,
    const char *val);

int evdns_base_count_nameservers(struct evdns_base *base);
```

evdns_base_nameserver_sockaddr_add()函数通过地址向 evdns_base 添加名字服务器。当前忽略 flags 参数，为向前兼容考虑，应该传入0。成功时函数返回0，失败时返回负值。（这个函数在2.0.7-rc 版本加入）

evdns_base_nameserver_ip_add()函数向 evdns_base 加入字符串表示的名字服务器，格式可以是 IPv4地址、IPv6地址、带端口号的 IPv4地址（IPv4:Port），或者带端口号的 IPv6地址（[IPv6]:Port）。成功时函数返回0，失败时返回负值。

evdns_base_load_hosts()函数从 hosts_fname 文件中载入主机文件（格式与/etc/hosts 相 同）。成功时函数返回0，失败时返回负值。

evdns_base_search_clear()函数从 evdns_base 中移除所有（通过 search 配置的）搜索后缀；evdns_base_search_add()则添加后缀。

evdns_base_set_option()函数设置 evdns_base 中某选项的值。选项和值都用字符串表示。 （2.0.3版本之前，选项名后面必须有一个冒号）

解析一组配置文件后，可以使用 evdns_base_count_nameservers()查看添加了多少个名字服务器。

3.4 库端配置

有一些为 evdns 模块设置库级别配置的函数：

接口

```
typedef void (*evdns_debug_log_fn_type)(int is_warning, const char *msg);
void evdns_set_log_fn(evdns_debug_log_fn_type fn);
void evdns_set_transaction_id_fn(ev_uint16_t (*fn)(void));
```

因为历史原因，evdns 子系统有自己单独的日志。evdns_set_log_fn()可以设置一个回调函数，以便在丢弃日志消息前做一些操作。

为安全起见，evdns 需要一个良好的随机数发生源：使用0x20 hack 的时候，evdns 通过这个源来获取难以猜测（hard-to-guess）的事务 ID 以随机化查询（请参考“randomize-case”选项）。然而，较老版本的 libevent 没有自己的安全的 RNG（随机数发生器）。此时可以通过调用 evdns_set_transaction_id_fn()，传入一个返回难以预测（hard-to-predict）的两字节无符号整数的函数，来为 evdns 设置一个更好的随机数发生器。

2.0.4-alpha 以及后续版本中， libevent 有自己内置的安全的 RNG，evdns_set_transaction_id_fn()就没有效果了。

4. 底层 DNS 接口

有时候需要启动能够比从 evdns_getaddrinfo()获取的 DNS 请求进行更精细控制的特别的DNS 请求，libevent 也为此提供了接口。

缺少的特征

当前 libevent 的 DNS 支持缺少其他底层 DNS 系统所具有的一些特征，如支持任意请求类型和 TCP 请求。如果需要 evdns 所不具有的特征，欢迎贡献一个补丁。也可以看看其他全特征的 DNS 库，如 c-ares。

接口

```
#define DNS_QUERY_NO_SEARCH /* ... */

#define DNS_IPv4_A          /* ... */
#define DNS_PTR             /* ... */
#define DNS_IPv6_AAAA       /* ... */

typedef void (*evdns_callback_type)(int result, char type, int count,
    int ttl, void *addresses, void *arg);

struct evdns_request *evdns_base_resolve_ipv4(struct evdns_base *base,
    const char *name, int flags, evdns_callback_type callback, void *ptr);
struct evdns_request *evdns_base_resolve_ipv6(struct evdns_base *base,
    const char *name, int flags, evdns_callback_type callback, void *ptr);
struct evdns_request *evdns_base_resolve_reverse(struct evdns_base *base,
    const struct in_addr *in, int flags, evdns_callback_type callback,
    void *ptr);
struct evdns_request *evdns_base_resolve_reverse_ipv6(
    struct evdns_base *base, const struct in6_addr *in, int flags,
    evdns_callback_type callback, void *ptr);
```

这些解析函数为一个特别的记录发起 DNS 请求。每个函数要求一个 evdns_base 用于发起请求、一个要查询的资源（正向查询时的主机名，或者反向查询时的地址）、一组用以确定如何进行查询的标志、一个查询完成时调用的回调函数，以及一个用户提供的传给回调函数的指针。

flags 参数可以是0，也可以用 DNS_QUERY_NO_SEARCH 明确禁止原始查询失败时在搜索列表中进行搜索。DNS_QUERY_NO_SEARCH 对反向查询无效，因为反向查询不进行搜索。

请求完成（不论是否成功）时回调函数会被调用。回调函数的参数是指示成功或者错误码（参看下面的 DNS 错误表）的 result、一个记录类型（DNS_IPv4_A、DNS_IPv6_AAAA，或者 DNS_PTR）、addresses 中的记录数、以秒为单位的存活时间、地址（查询结果），以及用户提供的指针。

发生错误时传给回调函数的 addresses 参数为 NULL。没有错误时：对于 PTR 记录，addresses 是空字符结束的字符串；对于 IPv4记录，则是网络字节序的四字节地址值数组；对于 IPv6记录，则是网络字节序的16字节记录数组。（注意：即使没有错误，addresses 的个数也可能是0。名字存在，但是没有请求类型的记录时就会出现这种情况）

可能传递给回调函数的错误码如下：

Code	Meaning
DNS_ERR_NONE	没有错误
DNS_ERR_FORMAT	服务器不识别查询请求DNS_ERR_SERVERFAILED 服务器内部错误
DNS_ERR_SERVERFAILED	服务器报告内部错误
DNS_ERR_NOTEXIST	没有给定名字的记录
DNS_ERR_NOTIMPL	服务器不识别这种类型的查询
DNS_ERR_REFUSED	因为策略设置，服务器拒绝查询
DNS_ERR_TRUNCATED	DNS 记录不适合 UDP 分组
DNS_ERR_UNKNOWN	未知的内部错误
DNS_ERR_TIMEOUT	等待超时
DNS_ERR_SHUTDOWN	用户请求关闭 evdns 系统

Code	Meaning
DNS_ERR_CANCEL	用户请求取消查询
DNS_ERR_NODATA	相应回复了，但没有答案

（DNS_ERR_NODATA是2.0.15稳定的新版本。）

您可以使用以下命令将这些错误代码解码为人类可读的字符串：

接口

```
const char *evdns_err_to_string(int err);
```

每个解析函数都返回不透明的 evdns_request 结构体指针。回调函数被调用前的任何时候都可以用这个指针来取消请求：

接口

```
void evdns_cancel_request(struct evdns_base *base,
    struct evdns_request *req);
```

用这个函数取消请求将使得回调函数被调用，带有错误码 DNS_ERR_CANCEL

挂起DNS客户端操作，更换名字服务器

有时候需要重新配置或者关闭 DNS 子系统，但不能影响进行中的 DNS 请求。

接口

```
int evdns_base_clear_nameservers_and_suspend(struct evdns_base *base);
int evdns_base_resume(struct evdns_base *base);
```

evdns_base_clear_nameservers_and_suspend()会移除所有名字服务器，但未决的请求会被保留，直到随后重新添加名字服务器，调用 evdns_base_resume()。

这些函数成功时返回0，失败时返回-1。它们在2.0.1-alpha 版本引入。

5. DNS 服务器接口

libevent 为实现不重要的 DNS 服务器，响应通过 UDP 传输的 DNS 请求提供了简单机制。

本节要求读者对 DNS 协议有一定的了解。

5.1 创建和关闭 DNS 服务器

接口

```
struct evdns_server_port *evdns_add_server_port_with_base(
    struct event_base *base,
    evutil_socket_t socket,
    int flags,
    evdns_request_callback_fn_type callback,
    void *user_data);

typedef void (*evdns_request_callback_fn_type)(
    struct evdns_server_request *request,
    void *user_data);

void evdns_close_server_port(struct evdns_server_port *port);
```

要开始监听 DNS 请求，调用 evdns_add_server_port_with_base()。函数要求用于事件处理的 event_base、用于监听的 UDP 套接字、可用的标志（现在总是0）、一个收到 DNS查询时要调用的回调函数，以及要传递给回调函数的用户数据指针 。函数返回 evdns_server_port 对象。

使用 DNS 服务器完成工作后，需要调用 evdns_close_server_port()。

evdns_add_server_port_with_base() 是 2.0.1-alpha 版本引入的，而evdns_close_server_port()则由1.3版本引入。

5.2 检测 DNS 请求

不幸的是，当前 libevent 没有提供较好的获取 DNS 请求的编程接口，用户需要包含

event2/dns_struct.h 文件，查看 evdns_server_request 结构体。

未来版本的 libevent 应该会提供更好的方法。

接口

```
struct evdns_server_request {
        int flags;
        int nquestions;
        struct evdns_server_question **questions;
};
#define EVDNS_QTYPE_AXFR 252
#define EVDNS_QTYPE_ALL  255
struct evdns_server_question {
        int type;
        int dns_question_class;
        char name[1];
};
```

flags 字段包含请求中设置的 DNS 标志；nquestions 字段是请求中的问题数；questions 是 evdns_server_question 结构体指针数组。每个 evdns_server_question 包含请求的资源类型（请看下面的 EVDNS*TYPE 宏列表）、请求类别（通常为 EVDNS_CLASS_INET），以及请求的主机名。

这些结构体在1.3版本中引入，但是1.4版之前的名字是 dns_question_class。名字中的“class”会让 C++用户迷惑。仍然使用原来的“class”名字的 C 程序将不能在未来发布版本中正确工作。

接口

```
int evdns_server_request_get_requesting_addr(struct evdns_server_request *req,
        struct sockaddr *sa, int addr_len);
```

有时想知道哪个地址发出了特定的DNS请求。可以通过在其上调用evdns_server_request_get_requesting_addr（）进行检查。 您应该传入一个具有足够存储空间的sockaddr来保存该地址：建议使用struct sockaddr_storage。

此功能在Libevent 1.3c中引入。

5.3 响应 DNS 请求

DNS 服务器收到每个请求后，会将请求传递给用户提供的回调函数，还带有用户数据指针。

回调函数必须响应请求或者忽略请求，或者确保请求最终会被回答或者忽略。回应请求前可以向回应中添加一个或者多个答案：

接口

```
int evdns_server_request_add_a_reply(struct evdns_server_request *req,
    const char *name, int n, const void *addrs, int ttl);
int evdns_server_request_add_aaaa_reply(struct evdns_server_request *req,
    const char *name, int n, const void *addrs, int ttl);
int evdns_server_request_add_cname_reply(struct evdns_server_request *req,
    const char *name, const char *cname, int ttl);
```

上述函数为请求 req 的 DNS 回应的结果节添加一个 RR（类型分别为 A、AAAA 和 CNAME）。 各个函数中，name 是要为之添加结果的主机名，ttl 是以秒为单位的存活时间。对于 A 和 AAAA 记录，n 是要添加的地址个数，addrs 是到原始地址的指针：对于 A 记录，是以 n*4 字节序列格式给出的 IPv4地址；对于 AAAA 记录，是以 n*16字节序列格式给出的 IPv6地址。

成功时函数返回0，失败时返回-1。

接口

```
int evdns_server_request_add_ptr_reply(struct evdns_server_request *req,
    struct in_addr *in, const char *inaddr_name, const char *hostname,
    int ttl);
```

这个函数为请求的结果节添加一个 PTR 记录。参数 req 和 ttl 跟上面的函数相同。必须提供 in（一个IPv4地址）和 inaddr_name(一个 arpa 域的地址）中的一个，而且只能提供一个， 以指示为回应提供哪种地址。hostname 是 PTR 查询的答案。

接口

```
#define EVDNS_ANSWER_SECTION 0
#define EVDNS_AUTHORITY_SECTION 1
#define EVDNS_ADDITIONAL_SECTION 2

#define EVDNS_TYPE_A       1
#define EVDNS_TYPE_NS      2
#define EVDNS_TYPE_CNAME   5
#define EVDNS_TYPE_SOA     6
#define EVDNS_TYPE_PTR    12
#define EVDNS_TYPE_MX     15
#define EVDNS_TYPE_TXT    16
#define EVDNS_TYPE_AAAA   28

#define EVDNS_CLASS_INET   1

int evdns_server_request_add_reply(struct evdns_server_request *req,
```

```
    int section, const char *name, int type, int dns_class, int ttl,
    int datalen, int is_name, const char *data);
```

这个函数为请求 req 的 DNS 回应添加任意 RR。section 字段指示添加到哪一节，其值应该是某个 EVDNS*SECTION。name 参数是 RR 的名字字段。type 参数是 RR 的类型字段，其值应该是某个 EVDNS_TYPE_*。dns_class 参数是 RR 的类别字段。RR 的 rdata 和 rdlength 字段将从 data 处的 datalen 字节中产生。如果 is_name 为 true，data 将被编码成 DNS 名字（例如，使用 DNS 名字压缩）。否则，data 将被直接包含到 RR 中。

接口

```
int evdns_server_request_respond(struct evdns_server_request *req, int err);
int evdns_server_request_drop(struct evdns_server_request *req);
```

evdns_server_request_respond()函数为请求发送 DNS 回应，带有用户添加的所有 RR，以及错误码 err。如果不想回应某个请求，可以调用 evdns_server_request_drop()来忽略请求，释放请求关联的内存和结构体。

接口

```
#define EVDNS_FLAGS_AA  0x400
#define EVDNS_FLAGS_RD  0x080

void evdns_server_request_set_flags(struct evdns_server_request *req,
                                    int flags);
```

如果要为回应消息设置任何标志，可以在发送回应前的任何时候调用这个函数。

除了evdns_server_request_set_flags()首次在2.0.1-alpha 版本中出现外，本节描述的所有 函数都在1.3 版本中引入。

5.4 DNS 服务器示例

示例：普通的DNS响应器

```
#include <event2/dns.h>
#include <event2/dns_struct.h>
#include <event2/util.h>
#include <event2/event.h>

#include <sys/socket.h>

#include <stdio.h>
#include <string.h>
#include <assert.h>

/* Let's try binding to 5353.  Port 53 is more traditional, but on most
   operating systems it requires root privileges. */
#define LISTEN_PORT 5353

#define LOCALHOST_IPV4_ARPA "1.0.0.127.in-addr.arpa"
#define LOCALHOST_IPV6_ARPA ("1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0."         \
                             "0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.ip6.arpa")

const ev_uint8_t LOCALHOST_IPV4[] = { 127, 0, 0, 1 };
const ev_uint8_t LOCALHOST_IPV6[] = { 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,1 };
```

```
#define TTL 4242

/* This toy DNS server callback answers requests for localhost (mapping it to
   127.0.0.1 or ::1) and for 127.0.0.1 or ::1 (mapping them to localhost).
 */
void server_callback(struct evdns_server_request *request, void *data)
{
    int i;
    int error=DNS_ERR_NONE;
    /* We should try to answer all the questions.  Some DNS servers don't do
       this reliably, though, so you should think hard before putting two
       questions in one request yourself. */
    for (i=0; i < request->nquestions; ++i) {
        const struct evdns_server_question *q = request->questions[i];
        int ok=-1;
        /* We don't use regular strcasecmp here, since we want a locale-
           independent comparison. */
        if (0 == evutil_ascii_strcasecmp(q->name, "localhost")) {
            if (q->type == EVDNS_TYPE_A)
                ok = evdns_server_request_add_a_reply(
                        request, q->name, 1, LOCALHOST_IPV4, TTL);
            else if (q->type == EVDNS_TYPE_AAAA)
                ok = evdns_server_request_add_aaaa_reply(
                        request, q->name, 1, LOCALHOST_IPV6, TTL);
        } else if (0 == evutil_ascii_strcasecmp(q->name, LOCALHOST_IPV4_ARPA)) {
            if (q->type == EVDNS_TYPE_PTR)
                ok = evdns_server_request_add_ptr_reply(
                        request, NULL, q->name, "LOCALHOST", TTL);
        } else if (0 == evutil_ascii_strcasecmp(q->name, LOCALHOST_IPV6_ARPA)) {
            if (q->type == EVDNS_TYPE_PTR)
                ok = evdns_server_request_add_ptr_reply(
                        request, NULL, q->name, "LOCALHOST", TTL);
        } else {
            error = DNS_ERR_NOTEXIST;
        }
        if (ok<0 && error==DNS_ERR_NONE)
            error = DNS_ERR_SERVERFAILED;
    }
    /* Now send the reply. */
    evdns_server_request_respond(request, error);
}

int main(int argc, char **argv)
{
    struct event_base *base;
    struct evdns_server_port *server;
    evutil_socket_t server_fd;
    struct sockaddr_in listenaddr;

    base = event_base_new();
    if (!base)
        return 1;

    server_fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (server_fd < 0)
        return 2;
    memset(&listenaddr, 0, sizeof(listenaddr));
```

```
    listenaddr.sin_family = AF_INET;
    listenaddr.sin_port = htons(LISTEN_PORT);
    listenaddr.sin_addr.s_addr = INADDR_ANY;
    if (bind(server_fd, (struct sockaddr*)&listenaddr, sizeof(listenaddr))<0)
        return 3;
    /*The server will hijack the event loop after receiving the first request if
the socket is blocking*/
    if(evutil_make_socket_nonblocking(server_fd)<0)
        return 4;
    server = evdns_add_server_port_with_base(base, server_fd, 0,
                                             server_callback, NULL);

    event_base_dispatch(base);

    evdns_close_server_port(server);
    event_base_free(base);

    return 0;
}
```

示例

Timer

示例

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

static int numCalls = 0;
static int numCalls_now = 0;
struct timeval lasttime;
struct timeval lasttime_now;

static void timeout_cb(evutil_socket_t fd, short event, void *arg)
{
    struct event *ev = (struct event *)arg;
    struct timeval newtime,tv_diff;
    double elapsed;

    evutil_gettimeofday(&newtime, NULL);
    evutil_timersub(&newtime, &lasttime, &tv_diff);

    elapsed = tv_diff.tv_sec +  (tv_diff.tv_usec / 1.0e6);

    lasttime = newtime;
    printf("[%.3f]timeout_cb %d \n",elapsed,++numCalls);

}

static void now_timeout_cb(evutil_socket_t fd, short event, void *arg)
{
    struct event *ev = (struct event *)arg;
    struct timeval tv = {3,0};
    struct timeval newtime,tv_diff;
    double elapsed;
```

```
    evutil_gettimeofday(&newtime, NULL);
    evutil_timersub(&newtime, &lasttime_now, &tv_diff);
    elapsed = tv_diff.tv_sec +  (tv_diff.tv_usec / 1.0e6);

    lasttime_now = newtime;
    printf("[%.3f]now_timeout_cb %d \n",elapsed,++numCalls_now);

    //每次回调都将当前先del，再次添加
    //if (!event_pending(ev,EV_PERSIST|EV_TIMEOUT,NULL))
    //{
    //  printf("if\n");
    //  event_del(ev);
    //  event_add(ev, &tv);
    //}
    //添加新的event
    if (!event_pending(ev,EV_PERSIST|EV_TIMEOUT,NULL))
    {
        struct event_base *evBase = event_get_base(ev);
        event_del(ev);
        event_free(ev);

        ev = event_new(evBase, -1, EV_PERSIST|EV_TIMEOUT, now_timeout_cb,
event_self_cbarg());
        event_add(ev, &tv);
    }
}

int main(int argc, char **argv)
{
    struct event_base *evBase = NULL;
    struct event_config *evConf = NULL;
    struct event *ev_now_timeout = NULL;
    struct event *ev_timeout = NULL;
    struct timeval tv = {0,0};
    struct timeval tv_now = {0,0};
    //创建简单的event_base
    evBase = event_base_new();

    //创建带配置的event_base
    evConf = event_config_new();//创建event_config
    evBase = event_base_new_with_config(evConf);

    //创建event
    //传递自己event_self_cbarg()
    ev_now_timeout = evtimer_new(evBase, now_timeout_cb, event_self_cbarg());

    //设置时间
    tv.tv_sec = 1;
    tv.tv_usec = 500 * 1000;
    ev_timeout = event_new(evBase, -1, EV_PERSIST|EV_TIMEOUT, timeout_cb,
event_self_cbarg());

    //添加event
    event_add(ev_now_timeout, &tv_now);//立即执行一次，然后定时
    event_add(ev_timeout, &tv);

    //获取时间
```

```
    evutil_gettimeofday(&lasttime, NULL);
    evutil_gettimeofday(&lasttime_now, NULL);
    //循环
    //event_base_loop(evBase, 0);
    event_base_dispatch(evBase);

    //释放
    event_free(ev_timeout);
    event_free(ev_now_timeout);
    event_config_free(evConf);
    event_base_free(evBase);
}
```

实现了一个Timer定时器，添加了2个event，ev_now_timeout立即执行一次，然后按3s周期执行；ev_timeout以1.5s周期执行。

TCP

Server

```
#include <string.h>
#include <unistd.h>
#include <event2/event.h>
#include <event2/listener.h>
#include <event2/bufferevent.h>
#include <event2/buffer.h>
// 读缓冲区回调
static void read_cb(struct bufferevent *bev, void *arg)
{
    //读到buff
    char buf[1024] = {0};
    bufferevent_read(bev, buf, sizeof(buf));

    //读取evbuffer
    //struct evbuffer *input = bufferevent_get_input(bev);
    //size_t size = evbuffer_get_length(input);
    //
    //char *buf = (char *)malloc(size);
    //evbuffer_remove(input, buf, size);

    printf("[server]rece client data\n");
    printf("[server]client say: %s\n", buf);

}

// 写缓冲区回调
static void write_cb(struct bufferevent *bev, void *arg)
{
    printf("[server]write_cb\n");
}

// 事件
static void event_cb(struct bufferevent *bev, short events, void *arg)
{
    if (events & BEV_EVENT_EOF)
    {
        printf("[server]connection close\n");
```

```
    }
    else if(events & BEV_EVENT_ERROR)
    {
        printf("[server]connection error\n");
    }else if(events & BEV_EVENT_CONNECTED)
    {
        printf("[server]connection success\n");
        return;
    }

    bufferevent_free(bev);
    printf("[server]bufferevent free\n");
}

static void send_cb(evutil_socket_t fd, short what, void *arg)
{
    char buf[1024] = {0};
    struct bufferevent* bev = (struct bufferevent*)arg;

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

    //struct evbuffer *output = bufferevent_get_output(bev);
    //evbuffer_add(output, buf, strlen(buf)+1);

    bufferevent_write(bev, buf, strlen(buf)+1);
}

static void cb_listener(
        struct evconnlistener *listener,
        evutil_socket_t fd,
        struct sockaddr *addr,
        int len, void *ptr)
{
   printf("[server]connect new client\n");

   struct event_base* base = (struct event_base*)ptr;
   // 通信操作
   // 添加新事件
   struct bufferevent *bev;
   bev = bufferevent_socket_new(base, fd, BEV_OPT_CLOSE_ON_FREE);

   if (!bev) {
        fprintf(stderr, "[server]error constructing bufferevent!");
        event_base_loopbreak(base);
        return;
    }

   // 给bufferevent缓冲区设置回调
   bufferevent_setcb(bev, read_cb, write_cb, event_cb, NULL);

   bufferevent_enable(bev, EV_READ);

   // 创建一个事件
   struct event* ev = event_new(base, STDIN_FILENO,
                                EV_READ | EV_PERSIST,
                                send_cb, bev);
   event_add(ev, NULL);
```

```
}

static void accept_error_cb(struct evconnlistener *listener, void *ctx)
{
    struct event_base *base = evconnlistener_get_base(listener);
    int err = EVUTIL_SOCKET_ERROR();
    printf("[Server]Got an error %d (%s) on the listener.Shutting down.\n",
        err, evutil_socket_error_to_string(err));
    event_base_loopexit(base, NULL);
}


int main(int argc, const char* argv[])
{
    struct sockaddr_in serv;
    struct event_base* base;
    struct evconnlistener* listener;

    // init server
    memset(&serv, 0, sizeof(serv));
    serv.sin_family = AF_INET;
    serv.sin_port = htons(6666);
    serv.sin_addr.s_addr = htonl(INADDR_ANY);

    base = event_base_new();

    if (!base) {
        printf("[server]Couldn't open event base\n");
        return 1;
    }

    // 创建套接字
    // 绑定
    // 接收连接请求
    //LEV_OPT_CLOSE_ON_FREE 如果设置了这个选项，释放连接监听器会关闭底层套接字。
    //LEV_OPT_REUSEABLE 某些平台在默认情况下，关闭某监听套接字后，要过一会儿其他套接字才可以绑定到同一个端口。
    //设置这个标志会让 libevent 标记套接字是可重用的，这样一旦关闭，可以立即打开其他套接字，在相同端口进行监听。
    listener = evconnlistener_new_bind(base, cb_listener, base,
                                       LEV_OPT_CLOSE_ON_FREE | LEV_OPT_REUSEABLE,
                                       36, (struct sockaddr*)&serv, sizeof(serv));

    if (!listener) {
        perror("[server]Couldn't create listener");
        return 1;
    }

    //error处理
    evconnlistener_set_error_cb(listener, accept_error_cb);

    event_base_dispatch(base);

    //释放
    evconnlistener_free(listener);
    event_base_free(base);

    return 0;
```

```
}
```

实现了一个TCP server，监听6666端口，与客户端建立连接，以后可以互相发送消息。

Client

实现了一个TCP client，连接本地6666端口，与客户端建立连接，以后可以互相发送消息。

```
#include <unistd.h>
#include <string.h>
#include <event2/event.h>
#include <event2/util.h>
#include <event2/bufferevent.h>
#include <event2/buffer.h>

void read_cb(struct bufferevent *bev, void *arg)
{
    char buf[1024] = {0};
    bufferevent_read(bev, buf, sizeof(buf));
    printf("[client]rece server data\n");
    printf("[client]server say: %s\n", buf);
}

void write_cb(struct bufferevent *bev, void *arg)
{
   printf("[client]write_cb\n");
}

void event_cb(struct bufferevent *bev, short events, void *arg)
{
    if (events & BEV_EVENT_EOF)
    {
        printf("[client]connection close\n");
    }
    else if(events & BEV_EVENT_ERROR)
    {
        printf("[client]connection error\n");
    }
    else if(events & BEV_EVENT_CONNECTED)
    {
        printf("[client]connection success\n");
        return;
    }

    bufferevent_free(bev);
    printf("[client]bufferevent free\n");
}

void send_cb(evutil_socket_t fd, short what, void *arg)
{
    char buf[1024] = {0};
    struct bufferevent* bev = (struct bufferevent*)arg;
    read(fd, buf, sizeof(buf));
    bufferevent_write(bev, buf, strlen(buf)+1);
}

int main(int argc, const char* argv[])
```

```
{
    struct event_base *base;
    struct bufferevent* bev;
    struct sockaddr_in serv;
    struct event* ev;

    base = event_base_new();

    if (!base) {
        printf("[client]Couldn't open event base\n");
        return 1;
    }

    bev = bufferevent_socket_new(base, -1, BEV_OPT_CLOSE_ON_FREE);

    //连接服务器
    memset(&serv, 0, sizeof(serv));
    serv.sin_family = AF_INET;
    serv.sin_port = htons(6666);
    //解析 IP 地址
    evutil_inet_pton(AF_INET, "127.0.0.1", &serv.sin_addr.s_addr);
    //连接
    bufferevent_socket_connect(bev, (struct sockaddr*)&serv, sizeof(serv));
    //设置回调
    bufferevent_setcb(bev, read_cb, write_cb, event_cb, NULL);
    bufferevent_enable(bev, EV_READ);

    // 创建一个事件
    //STDIN_FILENO：接收键盘的输入
    ev = event_new(base, STDIN_FILENO, EV_READ | EV_PERSIST,
                                    send_cb, bev);
    event_add(ev, NULL);

    event_base_dispatch(base);

    //释放
    bufferevent_free(bev);
    event_free(ev);
    event_base_free(base);

}
```

HTTP

Server

实现了一个http server。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <sys/types.h>
#include <sys/stat.h>

#include <sys/stat.h>
#include <sys/socket.h>
```

```c
#include <fcntl.h>
#include <unistd.h>
#include <dirent.h>

#include <signal.h>

#include <event2/event.h>
#include <event2/http.h>
#include <event2/listener.h>
#include <event2/buffer.h>
#include <event2/util.h>
#include <event2/keyvalq_struct.h>

char uri_root[512];

static const struct table_entry {
    const char *extension;
    const char *content_type;
} content_type_table[] = {
    { "txt", "text/plain" },
    { "c", "text/plain" },
    { "h", "text/plain" },
    { "html", "text/html" },
    { "htm", "text/htm" },
    { "css", "text/css" },
    { "gif", "image/gif" },
    { "jpg", "image/jpeg" },
    { "jpeg", "image/jpeg" },
    { "png", "image/png" },
    { "pdf", "application/pdf" },
    { "ps", "application/postscript" },
    { NULL, NULL },
};

/* 尝试猜测“path”的内容类型 */
static const char *
guess_content_type(const char *path)
{
    const char *last_period, *extension;
    const struct table_entry *ent;
    last_period = strrchr(path, '.');
    if (!last_period || strchr(last_period, '/'))
        goto not_found; /* no exension */
    extension = last_period + 1;
    for (ent = &content_type_table[0]; ent->extension; ++ent) {
        if (!evutil_ascii_strcasecmp(ent->extension, extension))
            return ent->content_type;
    }

not_found:
    return "application/misc";
}

/* 用于/test URI请求的回调 */
static void
dump_request_cb(struct evhttp_request *req, void *arg)
{
    const char *cmdtype;
```

```
    struct evkeyvalq *headers;
    struct evkeyval *header;
    struct evbuffer *buf_in;
    struct evbuffer *buf_out;
    struct evhttp_uri *decoded = NULL;
    struct evkeyvalq params;
    char *decoded_path;
    const char *path;
    char cbuf[1024] = {0};
    const char *uri = evhttp_request_get_uri(req);

    switch (evhttp_request_get_command(req)) {
    case EVHTTP_REQ_GET: cmdtype = "GET"; break;
    case EVHTTP_REQ_POST: cmdtype = "POST"; break;
    case EVHTTP_REQ_HEAD: cmdtype = "HEAD"; break;
    case EVHTTP_REQ_PUT: cmdtype = "PUT"; break;
    case EVHTTP_REQ_DELETE: cmdtype = "DELETE"; break;
    case EVHTTP_REQ_OPTIONS: cmdtype = "OPTIONS"; break;
    case EVHTTP_REQ_TRACE: cmdtype = "TRACE"; break;
    case EVHTTP_REQ_CONNECT: cmdtype = "CONNECT"; break;
    case EVHTTP_REQ_PATCH: cmdtype = "PATCH"; break;
    default: cmdtype = "unknown"; break;
    }

    printf("Received a %s request for %s\nHeaders:\n",
        cmdtype, uri);

    headers = evhttp_request_get_input_headers(req);

    for (header = headers->tqh_first; header;
        header = header->next.tqe_next) {
        printf("  %s: %s\n", header->key, header->value);
    }

    /*********************************/
    /* 解析 URI */
    decoded = evhttp_uri_parse(uri);
    if (!decoded) {
        printf("It's not a good URI. Sending BADREQUEST\n");
        evhttp_send_error(req, HTTP_BADREQUEST, 0);
        return;
    }

    /* 获取path */
    path = evhttp_uri_get_path(decoded);
    if (!path){
        evhttp_send_error(req, HTTP_BADREQUEST, 0);
        return;
    }
    printf("path: %s\n", path);


    //解析URI的参数
    //将URL数据封装成key-value格式,q=value1, s=value2
    evhttp_parse_query(uri, &params);
    //得到a所对应的value
    const char *a_data = evhttp_find_header(&params, "a");
```

```
    printf("a=%s\n",a_data);
    /*********************************/
    if (strcmp(cmdtype,"POST") == 0)
    {
        //获取POST方法的数据
        buf_in = evhttp_request_get_input_buffer(req);

        if (buf_in==NULL)
        {
            printf("evBuf null, err\n");
            goto err;
        }
        //获取长度
        int buf_in_len = evbuffer_get_length(buf_in);

        printf("evBuf len:%d\n",buf_in_len);

        if(buf_in_len <= 0)
        {
            goto err;
        }
        //将数据从evbuff中移动到char *
        int str_len = evbuffer_remove(buf_in,cbuf,sizeof(cbuf));

        if (str_len <= 0)
        {
            printf("post parameter null err\n");
            goto err;
        }
        printf("str_len:%d cbuf:%s\n",str_len,cbuf);
    }
    /*********************************/
    buf_out = evbuffer_new();
    if(!buf_out)
    {
        puts("failed to create response buffer \n");
        return;
    }
    evbuffer_add_printf(buf_out,"%s","success");
    evhttp_send_reply(req, 200, "OK", buf_out);
    return;
err:
    evhttp_send_error(req, HTTP_INTERNAL, 0);
}

static void
send_document_cb(struct evhttp_request *req, void *arg)
{
    evhttp_send_error(req, 404, "url was not found");
}

static void
do_term(int sig, short events, void *arg)
{
    struct event_base *base = (struct event_base *)arg;
    event_base_loopbreak(base);
    fprintf(stderr, "Got %i, Terminating\n", sig);
}
```

```
static int
display_listen_sock(struct evhttp_bound_socket *handle)
{
    struct sockaddr_storage ss;
    evutil_socket_t fd;
    ev_socklen_t socklen = sizeof(ss);
    char addrbuf[128];
    void *inaddr;
    const char *addr;
    int got_port = -1;

    fd = evhttp_bound_socket_get_fd(handle);
    memset(&ss, 0, sizeof(ss));
    if (getsockname(fd, (struct sockaddr *)&ss, &socklen)) {
        perror("getsockname() failed");
        return 1;
    }

    if (ss.ss_family == AF_INET) {
        got_port = ntohs(((struct sockaddr_in*)&ss)->sin_port);
        inaddr = &((struct sockaddr_in*)&ss)->sin_addr;
    } else if (ss.ss_family == AF_INET6) {
        got_port = ntohs(((struct sockaddr_in6*)&ss)->sin6_port);
        inaddr = &((struct sockaddr_in6*)&ss)->sin6_addr;
    }
    else {
        fprintf(stderr, "Weird address family %d\n",
            ss.ss_family);
        return 1;
    }

    addr = evutil_inet_ntop(ss.ss_family, inaddr, addrbuf,
        sizeof(addrbuf));
    if (addr) {
        printf("Listening on %s:%d\n", addr, got_port);
        evutil_snprintf(uri_root, sizeof(uri_root),
            "http://%s:%d",addr,got_port);
    } else {
        fprintf(stderr, "evutil_inet_ntop failed\n");
        return 1;
    }

    return 0;
}

int
main(int argc, char **argv)
{

    struct event_base *base = NULL;
    struct evhttp *http = NULL;
    struct evhttp_bound_socket *handle = NULL;
    struct evconnlistener *lev = NULL;
    struct event *term = NULL;
    int ret = 0;

    if (signal(SIGPIPE, SIG_IGN) == SIG_ERR) {
```

```
        ret = 1;
        goto err;
    }

    //event_base
    base = event_base_new();

    if (!base) {
        fprintf(stderr, "Couldn't create an event_base: exiting\n");
        ret = 1;
    }

    /* 创建一个新的evhttp对象来处理请求。 */
    http = evhttp_new(base);
    if (!http) {
        fprintf(stderr, "couldn't create evhttp. Exiting.\n");
        ret = 1;
    }

    /* / test URI将所有请求转储到stdout并说200 OK。 */
    evhttp_set_cb(http, "/test", dump_request_cb, NULL);

    /*要接受任意请求，需要设置一个“通用”cb。 还可以为特定路径添加回调。 */
    evhttp_set_gencb(http, send_document_cb, NULL);

    //绑定socket
    handle = evhttp_bind_socket_with_handle(http, "0.0.0.0", 8888);

    if (!handle) {
        fprintf(stderr, "couldn't bind to port %d. Exiting.\n", 8888);
        ret = 1;
        goto err;
    }

    //监听socket
    if (display_listen_sock(handle)) {
        ret = 1;
        goto err;
    }

    //终止信号
    term = evsignal_new(base, SIGINT, do_term, base);
    if (!term)
        goto err;
    if (event_add(term, NULL))
        goto err;

    //事件分发
    event_base_dispatch(base);

err:

    if (http)
        evhttp_free(http);
    if (term)
        event_free(term);
    if (base)
        event_base_free(base);
```

```
    return ret;
}
```

Client

实现了一个http client。

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <event2/bufferevent.h>
#include <event2/buffer.h>
#include <event2/listener.h>
#include <event2/util.h>
#include <event2/http.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

static int ignore_cert = 0;

static void
http_request_done(struct evhttp_request *req, void *ctx)
{
    char buffer[256];
    int nread;
    struct evbuffer *buf_in;
    char cbuf[1024] = {0};

    //错误处理，打印
    if (!req || !evhttp_request_get_response_code(req)) {

        struct bufferevent *bev = (struct bufferevent *) ctx;
        unsigned long oslerr;
        int printed_err = 0;
        int errcode = EVUTIL_SOCKET_ERROR();
        fprintf(stderr, "some request failed - no idea which one though!\n");

        /* 尝试打印 */
        if (! printed_err)
            fprintf(stderr, "socket error = %s (%d)\n",
                evutil_socket_error_to_string(errcode),
                errcode);
        return;
    }

    fprintf(stderr, "Response line: %d %s\n",
        evhttp_request_get_response_code(req),
        evhttp_request_get_response_code_line(req));

    //获取数据
    buf_in = evhttp_request_get_input_buffer(req);
```

```
    if (buf_in==NULL)
    {
        printf("evBuf null, err\n");
    }
    //获取长度
    int buf_in_len = evbuffer_get_length(buf_in);

    printf("evBuf len:%d\n",buf_in_len);

    if(buf_in_len <= 0)
    {
    }
    //将数据从evbuff中移动到char *
    int str_len = evbuffer_remove(buf_in,cbuf,sizeof(cbuf));

    if (str_len <= 0)
    {
        printf("post parameter null err\n");
    }
    printf("str_len:%d cbuf:%s\n",str_len,cbuf);
}

static void
err(const char *msg)
{
    fputs(msg, stderr);
}

int
main(int argc, char **argv)
{
    int r;
    struct event_base *base = NULL;
    struct evhttp_uri *http_uri = NULL;
    const char *url = NULL, *data_file = NULL;
    const char *scheme, *host, *path, *query;
    char uri[256];
    int port;
    int retries = 0;
    int timeout = -1;

    struct bufferevent *bev;
    struct evhttp_connection *evcon = NULL;
    struct evhttp_request *req;
    struct evkeyvalq *output_headers;
    struct evbuffer *output_buffer;

    int i;
    int ret = 0;

    //初始化url
    url = "http://127.0.0.1:8888/test?a=123";
    if (!url) {
        goto error;
    }

    http_uri = evhttp_uri_parse(url);
```

```
    if (http_uri == NULL) {
        err("malformed url");
        goto error;
    }

    scheme = evhttp_uri_get_scheme(http_uri);
    //忽略大小写比较字符串
    if (scheme == NULL || strcasecmp(scheme, "http") != 0) {
        err("url must be http");
        goto error;
    }

    host = evhttp_uri_get_host(http_uri);
    if (host == NULL) {
        err("url must have a host");
        goto error;
    }

    port = evhttp_uri_get_port(http_uri);

    if (port == -1) {
        port = 80;
    }

    path = evhttp_uri_get_path(http_uri);
    if (strlen(path) == 0) {
        path = "/";
    }

    query = evhttp_uri_get_query(http_uri);

    if (query == NULL) {
        //将可变参数 “…” 按照format的格式格式化为字符串，然后再将其拷贝至str中。
        snprintf(uri, sizeof(uri) - 1, "%s", path);
    } else {
        snprintf(uri, sizeof(uri) - 1, "%s?%s", path, query);
    }

    uri[sizeof(uri) - 1] = '\0';

    // 创建 event base
    base = event_base_new();
    if (!base) {
        perror("event_base_new()");
        goto error;
    }

    if (strcasecmp(scheme, "http") == 0) {
        bev = bufferevent_socket_new(base, -1, BEV_OPT_CLOSE_ON_FREE);
    }

    if (bev == NULL) {
        fprintf(stderr, "bufferevent_socket_new() failed\n");
        goto error;
    }

    evcon = evhttp_connection_base_bufferevent_new(base, NULL, bev, host, port);
    if (evcon == NULL) {
```

```
        fprintf(stderr, "evhttp_connection_base_bufferevent_new() failed\n");
        goto error;
    }

    //重试
    if (retries > 0) {
        evhttp_connection_set_retries(evcon, retries);
    }

    //超时
    if (timeout >= 0) {
        evhttp_connection_set_timeout(evcon, timeout);
    }

    //回调
    req = evhttp_request_new(http_request_done, bev);

    if (req == NULL) {
        fprintf(stderr, "evhttp_request_new() failed\n");
        goto error;
    }

    output_headers = evhttp_request_get_output_headers(req);
    evhttp_add_header(output_headers, "Host", host);
    evhttp_add_header(output_headers, "Connection", "close");

    //文件路径
    data_file = "/home/zza/libevent/demo/test.txt";

    if (data_file) {

        char buf[1024];
        output_buffer = evhttp_request_get_output_buffer(req);

        //post file传统复制
        //FILE* f = fopen(data_file, "rb");
        //size_t s;
        //size_t bytes = 0;
        //if (!f) {
        //  goto error;
        //}
        //
        //while ((s = fread(buf, 1, sizeof(buf), f)) > 0) {
        //  evbuffer_add(output_buffer, buf, s);
        //  bytes += s;
        //}
        //evutil_snprintf(buf, sizeof(buf)-1, "%lu", (unsigned long)bytes);
        //evhttp_add_header(output_headers, "Content-Length", buf);
        //fclose(f);

        //*************************************************************
        //post file使用evbuffer_add_file()或
        //evbuffer_add_file_segment()，以避免不必要的复制
        //int fd = open(data_file, O_RDONLY);

        //if (fd == -1)
        //{
        //  fprintf(stderr, "open %s failed\n", data_file);
```

```
        //  goto error;
        //}

        //evbuffer_add_file(output_buffer,fd,0,-1);
        //ev_ssize_t size =  evbuffer_copyout(output_buffer, buf, sizeof(buf));
        //evhttp_add_header(output_headers, "Content-Length", buf);

        //*************************************************************
        //http post json
        //sprintf(buf,"%s","{\"a\":\"b\"}");
        //evbuffer_add(output_buffer, buf, strlen(buf));
        //
        //evhttp_add_header(output_headers, "Content-Type",
"application/json;charset=UTF-8");

        //*************************************************************
        //http post format
        sprintf(buf,"%s\r\n%s\r\n\r\n%s\r\n%s\r\n",
            "--------------------------123",
            "Content-Disposition: form-data; name=\"hello\"",
            "world!!!!",
            "--------------------------123");
        evbuffer_add(output_buffer, buf, strlen(buf));

        evhttp_add_header(output_headers, "Content-Type",
            "multipart/form-data; boundary=--------------------------123");
        //*************************************************************
    }
    //文件路径为NULL时为get请求，非空post
    //请求
    r = evhttp_make_request(evcon, req, data_file ? EVHTTP_REQ_POST :
EVHTTP_REQ_GET, uri);

    if (r != 0) {
        fprintf(stderr, "evhttp_make_request() failed\n");
        goto error;
    }

    event_base_dispatch(base);

    goto cleanup;

error:
    printf("error stop");
    ret = 1;
cleanup:
    if (evcon)
        evhttp_connection_free(evcon);
    if (http_uri)
        evhttp_uri_free(http_uri);
    if (base)
        event_base_free(base);

    return ret;
}
```

HTTPS

Client

```
/*
  This is an example of how to hook up evhttp with bufferevent_ssl

  It just GETs an https URL given on the command-line and prints the response
  body to stdout.

  Actually, it also accepts plain http URLs to make it easy to compare http vs
  https code paths.

  Loosely based on le-proxy.c.
 */

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#include <sys/socket.h>
#include <netinet/in.h>

#include <event2/bufferevent_ssl.h>
#include <event2/bufferevent.h>
#include <event2/buffer.h>
#include <event2/listener.h>
#include <event2/util.h>
#include <event2/http.h>

#include <openssl/ssl.h>
#include <openssl/err.h>
#include <openssl/rand.h>

static int ignore_cert = 0;

static void
http_request_done(struct evhttp_request *req, void *ctx)
{
    char buffer[256];
    int nread;

    if (!req || !evhttp_request_get_response_code(req)) {
        /* If req is NULL, it means an error occurred, but
         * sadly we are mostly left guessing what the error
         * might have been.  We'll do our best... */
        struct bufferevent *bev = (struct bufferevent *) ctx;
        unsigned long oslerr;
        int printed_err = 0;
        int errcode = EVUTIL_SOCKET_ERROR();
        fprintf(stderr, "some request failed - no idea which one though!\n");
        /* Print out the OpenSSL error queue that libevent
         * squirreled away for us, if any. */
        while ((oslerr = bufferevent_get_openssl_error(bev))) {
            ERR_error_string_n(oslerr, buffer, sizeof(buffer));
            fprintf(stderr, "%s\n", buffer);
            printed_err = 1;
```

```
        }
        /* If the OpenSSL error queue was empty, maybe it was a
         * socket error; let's try printing that. */
        if (! printed_err)
            fprintf(stderr, "socket error = %s (%d)\n",
                evutil_socket_error_to_string(errcode),
                errcode);
        return;
    }

    fprintf(stderr, "Response line: %d %s\n",
        evhttp_request_get_response_code(req),
        evhttp_request_get_response_code_line(req));

    while ((nread = evbuffer_remove(evhttp_request_get_input_buffer(req),
            buffer, sizeof(buffer)))
           > 0) {
        /* These are just arbitrary chunks of 256 bytes.
         * They are not lines, so we can't treat them as such. */
        fwrite(buffer, nread, 1, stdout);
    }
}

static void
err(const char *msg)
{
    fputs(msg, stderr);
}

static void
err_openssl(const char *func)
{
    fprintf (stderr, "%s failed:\n", func);

    /* This is the OpenSSL function that prints the contents of the
     * error stack to the specified file handle. */
    ERR_print_errors_fp (stderr);

    exit(1);
}

int
main(int argc, char **argv)
{
    int r;
    struct event_base *base = NULL;
    struct evhttp_uri *http_uri = NULL;
    const char *url = NULL, *data_file = NULL;
    const char *crt = NULL;
    const char *scheme, *host, *path, *query;
    char uri[256];
    int port;
    int retries = 0;
    int timeout = -1;

    SSL_CTX *ssl_ctx = NULL;
    SSL *ssl = NULL;
    struct bufferevent *bev;
```

```
    struct evhttp_connection *evcon = NULL;
    struct evhttp_request *req;
    struct evkeyvalq *output_headers;
    struct evbuffer *output_buffer;

    int i;
    int ret = 0;
    enum { HTTP, HTTPS } type = HTTP;

    //初始化url
    url = "https://127.0.0.1:8888/login";

    if (!url) {
        goto error;
    }

    http_uri = evhttp_uri_parse(url);
    if (http_uri == NULL) {
        err("malformed url");
        goto error;
    }

    scheme = evhttp_uri_get_scheme(http_uri);
    if (scheme == NULL || (strcasecmp(scheme, "https") != 0 &&
                           strcasecmp(scheme, "http") != 0)) {
        err("url must be http or https");
        goto error;
    }

    host = evhttp_uri_get_host(http_uri);
    if (host == NULL) {
        err("url must have a host");
        goto error;
    }

    port = evhttp_uri_get_port(http_uri);
    if (port == -1) {
        port = (strcasecmp(scheme, "http") == 0) ? 80 : 443;
    }

    path = evhttp_uri_get_path(http_uri);
    if (strlen(path) == 0) {
        path = "/";
    }

    query = evhttp_uri_get_query(http_uri);
    if (query == NULL) {
        snprintf(uri, sizeof(uri) - 1, "%s", path);
    } else {
        snprintf(uri, sizeof(uri) - 1, "%s?%s", path, query);
    }
    uri[sizeof(uri) - 1] = '\0';

#if (OPENSSL_VERSION_NUMBER < 0x10100000L) || \
    (defined(LIBRESSL_VERSION_NUMBER) && LIBRESSL_VERSION_NUMBER < 0x20700000L)
    // Initialize OpenSSL
    SSL_library_init();
    ERR_load_crypto_strings();
```

```
    SSL_load_error_strings();
    OpenSSL_add_all_algorithms();
#endif

    /* This isn't strictly necessary... OpenSSL performs RAND_poll
     * automatically on first use of random number generator. */
    r = RAND_poll();
    if (r == 0) {
        err_openssl("RAND_poll");
        goto error;
    }

    /* Create a new OpenSSL context */
    ssl_ctx = SSL_CTX_new(SSLv23_method());
    if (!ssl_ctx) {
        err_openssl("SSL_CTX_new");
        goto error;
    }
    // Create event base
    base = event_base_new();
    if (!base) {
        perror("event_base_new()");
        goto error;
    }

    // Create OpenSSL bufferevent and stack evhttp on top of it
    ssl = SSL_new(ssl_ctx);
    if (ssl == NULL) {
        err_openssl("SSL_new()");
        goto error;
    }

    #ifdef SSL_CTRL_SET_TLSEXT_HOSTNAME
    // Set hostname for SNI extension
    SSL_set_tlsext_host_name(ssl, host);
    #endif

    if (strcasecmp(scheme, "http") == 0) {
        bev = bufferevent_socket_new(base, -1, BEV_OPT_CLOSE_ON_FREE);
    } else {
        type = HTTPS;
        bev = bufferevent_openssl_socket_new(base, -1, ssl,
            BUFFEREVENT_SSL_CONNECTING,
            BEV_OPT_CLOSE_ON_FREE|BEV_OPT_DEFER_CALLBACKS);
    }

    if (bev == NULL) {
        fprintf(stderr, "bufferevent_openssl_socket_new() failed\n");
        goto error;
    }

    bufferevent_openssl_set_allow_dirty_shutdown(bev, 1);

    // For simplicity, we let DNS resolution block. Everything else should be
    // asynchronous though.
    evcon = evhttp_connection_base_bufferevent_new(base, NULL, bev,
        host, port);
    if (evcon == NULL) {
```

```
        fprintf(stderr, "evhttp_connection_base_bufferevent_new() failed\n");
        goto error;
    }

    if (retries > 0) {
        evhttp_connection_set_retries(evcon, retries);
    }
    if (timeout >= 0) {
        evhttp_connection_set_timeout(evcon, timeout);
    }

    // Fire off the request
    req = evhttp_request_new(http_request_done, bev);
    if (req == NULL) {
        fprintf(stderr, "evhttp_request_new() failed\n");
        goto error;
    }

    output_headers = evhttp_request_get_output_headers(req);
    evhttp_add_header(output_headers, "Host", host);
    evhttp_add_header(output_headers, "Connection", "close");

    if (data_file) {
        /* NOTE: In production code, you'd probably want to use
         * evbuffer_add_file() or evbuffer_add_file_segment(), to
         * avoid needless copying. */
        FILE * f = fopen(data_file, "rb");
        char buf[1024];
        size_t s;
        size_t bytes = 0;

        if (!f) {
            goto error;
        }

        output_buffer = evhttp_request_get_output_buffer(req);
        while ((s = fread(buf, 1, sizeof(buf), f)) > 0) {
            evbuffer_add(output_buffer, buf, s);
            bytes += s;
        }
        evutil_snprintf(buf, sizeof(buf)-1, "%lu", (unsigned long)bytes);
        evhttp_add_header(output_headers, "Content-Length", buf);
        fclose(f);
    }

    r = evhttp_make_request(evcon, req, data_file ? EVHTTP_REQ_POST :
EVHTTP_REQ_GET, uri);
    if (r != 0) {
        fprintf(stderr, "evhttp_make_request() failed\n");
        goto error;
    }

    event_base_dispatch(base);
    goto cleanup;

error:
    ret = 1;
cleanup:
```

```
    if (evcon)
        evhttp_connection_free(evcon);
    if (http_uri)
        evhttp_uri_free(http_uri);
    if (base)
        event_base_free(base);

    if (ssl_ctx)
        SSL_CTX_free(ssl_ctx);
    if (type == HTTP && ssl)
        SSL_free(ssl);
#if (OPENSSL_VERSION_NUMBER < 0x10100000L) || \
    (defined(LIBRESSL_VERSION_NUMBER) && LIBRESSL_VERSION_NUMBER < 0x20700000L)
    EVP_cleanup();
    ERR_free_strings();

#if OPENSSL_VERSION_NUMBER < 0x10000000L
    ERR_remove_state(0);
#else
    ERR_remove_thread_state(NULL);
#endif

    CRYPTO_cleanup_all_ex_data();

    sk_SSL_COMP_free(SSL_COMP_get_compression_methods());
#endif /* (OPENSSL_VERSION_NUMBER < 0x10100000L) || \
    (defined(LIBRESSL_VERSION_NUMBER) && LIBRESSL_VERSION_NUMBER < 0x20700000L)
*/

#ifdef _WIN32
    WSACleanup();
#endif

    return ret;
}
```

Server

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <signal.h>

#include <sys/types.h>
#include <sys/stat.h>

#include <sys/stat.h>
#include <sys/socket.h>
#include <fcntl.h>
#include <unistd.h>
#include <dirent.h>
#include <netinet/in.h>

#include <openssl/ssl.h>
```

```c
#include <openssl/err.h>

#include <event2/bufferevent.h>
#include <event2/bufferevent_ssl.h>
#include <event2/event.h>
#include <event2/http.h>
#include <event2/buffer.h>
#include <event2/util.h>
#include <event2/keyvalq_struct.h>

unsigned short serverPort = 8888;

void die_most_horribly_from_openssl_error (const char *func)
{
    fprintf (stderr, "%s failed:\n", func);

    /* This is the OpenSSL function that prints the contents of the
    * error stack to the specified file handle. */
    ERR_print_errors_fp (stderr);

    exit (EXIT_FAILURE);
}

/* This callback gets invoked when we get any http request that doesn't match
 * any other callback.  Like any evhttp server callback, it has a simple job:
 * it must eventually call evhttp_send_error() or evhttp_send_reply().
 */
static void
login_cb (struct evhttp_request *req, void *arg)
{
    struct evbuffer *evb = NULL;
    const char *uri = evhttp_request_get_uri (req);
    struct evhttp_uri *decoded = NULL;

    /* 判断 req 是否是GET 请求 */
    if (evhttp_request_get_command (req) == EVHTTP_REQ_GET)
    {
        struct evbuffer *buf = evbuffer_new();
        if (buf == NULL) return;
        evbuffer_add_printf(buf, "Requested: %s\n", uri);
        evhttp_send_reply(req, HTTP_OK, "OK", buf);
        return;
    }

    /* Get请求，直接return 200 OK  */
    if (evhttp_request_get_command (req) != EVHTTP_REQ_POST)
    {
        evhttp_send_reply (req, 200, "OK", NULL);
        return;
    }

}

/**
 *  该回调负责创建新的SSL连接并将其包装在OpenSSL bufferevent中。
 *  这是我们实现https服务器而不是普通的http服务器的方式。
 */
static struct bufferevent* bevcb(struct event_base *base, void *arg)
```

```
{
    struct bufferevent* r;
    SSL_CTX *ctx = (SSL_CTX *) arg;

    r = bufferevent_openssl_socket_new (base,
            -1,
            SSL_new (ctx),
            BUFFEREVENT_SSL_ACCEPTING,
            BEV_OPT_CLOSE_ON_FREE);
    return r;
}

static void server_setup_certs (SSL_CTX *ctx,
        const char *certificate_chain,
        const char *private_key)
{
    printf ("Loading certificate chain from '%s'\n"
            "and private key from '%s'\n",
            certificate_chain, private_key);

    if (1 != SSL_CTX_use_certificate_chain_file (ctx, certificate_chain))
        die_most_horribly_from_openssl_error
("SSL_CTX_use_certificate_chain_file");

    if (1 != SSL_CTX_use_PrivateKey_file (ctx, private_key, SSL_FILETYPE_PEM))
        die_most_horribly_from_openssl_error ("SSL_CTX_use_PrivateKey_file");

    if (1 != SSL_CTX_check_private_key (ctx))
        die_most_horribly_from_openssl_error ("SSL_CTX_check_private_key");
}


static int
display_listen_sock(struct evhttp_bound_socket *handle)
{
    struct sockaddr_storage ss;
    evutil_socket_t fd;
    ev_socklen_t socklen = sizeof(ss);
    char addrbuf[128];
    void *inaddr;
    const char *addr;
    int got_port = -1;

    fd = evhttp_bound_socket_get_fd(handle);
    memset(&ss, 0, sizeof(ss));
    if (getsockname(fd, (struct sockaddr *)&ss, &socklen)) {
        perror("getsockname() failed");
        return 1;
    }

    if (ss.ss_family == AF_INET) {
        got_port = ntohs(((struct sockaddr_in*)&ss)->sin_port);
        inaddr = &((struct sockaddr_in*)&ss)->sin_addr;
    } else if (ss.ss_family == AF_INET6) {
        got_port = ntohs(((struct sockaddr_in6*)&ss)->sin6_port);
        inaddr = &((struct sockaddr_in6*)&ss)->sin6_addr;
    }
    else {
```

```
        fprintf(stderr, "Weird address family %d\n",
            ss.ss_family);
        return 1;
    }

    addr = evutil_inet_ntop(ss.ss_family, inaddr, addrbuf,
        sizeof(addrbuf));
    if (addr) {
        printf("Listening on %s:%d\n", addr, got_port);
    } else {
        fprintf(stderr, "evutil_inet_ntop failed\n");
        return 1;
    }

    return 0;
}


static int serve_some_http (void)
{
    struct event_base *base;
    struct evhttp *http;
    struct evhttp_bound_socket *handle;

    //创建event_base
    base = event_base_new ();
    if (! base)
    {
        fprintf (stderr, "Couldn't create an event_base: exiting\n");
        return 1;
    }

    /* 创建一个 evhttp 句柄，去处理用户端的requests请求 */
    http = evhttp_new (base);
    if (! http)
    {
        fprintf (stderr, "couldn't create evhttp. Exiting.\n");
        return 1;
    }
    /*****************************************/
    /* 创建SSL上下文环境 ，可以理解为 SSL句柄 */
    SSL_CTX *ctx = SSL_CTX_new (SSLv23_server_method ());
    SSL_CTX_set_options (ctx,
            SSL_OP_SINGLE_DH_USE |
            SSL_OP_SINGLE_ECDH_USE |
            SSL_OP_NO_SSLv2);

    /* Cheesily pick an elliptic curve to use with elliptic curve ciphersuites.
     * We just hardcode a single curve which is reasonably decent.
     * See http://www.mail-archive.com/openssl-dev@openssl.org/msg30957.html */
    EC_KEY *ecdh = EC_KEY_new_by_curve_name (NID_X9_62_prime256v1);
    if (! ecdh)
        die_most_horribly_from_openssl_error ("EC_KEY_new_by_curve_name");
    if (1 != SSL_CTX_set_tmp_ecdh (ctx, ecdh))
        die_most_horribly_from_openssl_error ("SSL_CTX_set_tmp_ecdh");

    /* 选择服务器证书 和 服务器私钥. */
    const char *certificate_chain = "server-certificate-chain.pem";
```

```
    const char *private_key = "server-private-key.pem";
    /* 设置服务器证书 和 服务器私钥 到
     OPENSSL ctx上下文句柄中 */
    server_setup_certs (ctx, certificate_chain, private_key);

    /*
        使我们创建好的evhttp句柄 支持 SSL加密
        实际上，加密的动作和解密的动作都已经帮
        我们自动完成，我们拿到的数据就已经解密之后的

        设置用于为与给定evhttp对象的连接创建新的bufferevent的回调。
        您可以使用它来覆盖默认的bufferevent类型，
        例如，使此evhttp对象使用SSL缓冲区事件而不是未加密的事件。
        新的缓冲区事件必须在未设置fd的情况下进行分配。
    */
    evhttp_set_bevcb (http, bevcb, ctx);
    /*****************************************/
    /* 设置http回调函数 */
    //默认回调
    //evhttp_set_gencb (http, send_document_cb, NULL);
    //专属uri路径回调
    evhttp_set_cb(http, "/login", login_cb, NULL);

    /* 设置监听IP和端口 */
    handle = evhttp_bind_socket_with_handle (http, "0.0.0.0", serverPort);
    if (! handle)
    {
        fprintf (stderr, "couldn't bind to port %d. Exiting.\n",(int)
serverPort);
        return 1;
    }

    //监听socket
    if (display_listen_sock(handle)) {
        return 1;
    }

    /* 开始阻塞监听 (永久执行) */
    event_base_dispatch (base);


    return 0;
}

int main (int argc, char **argv)
{
    /*OpenSSL 初始化 */
    signal (SIGPIPE, SIG_IGN);

    SSL_library_init ();
    SSL_load_error_strings ();
    OpenSSL_add_all_algorithms ();

    printf ("Using OpenSSL version \"%s\"\nand libevent version \"%s\"\n",
            SSLeay_version (SSLEAY_VERSION),
            event_get_version ());
    /* now run http server (never returns) */
    return serve_some_http ();
```

```
}
```