

原创: <http://blog.sina.com.cn/yaozj>
整理: <http://yaocoder.blog.51cto.com/>

前言

翻译自: http://www.wangafu.net/~nickm/libevent-book/Ref0_meta.html

1. 从一万英尺外看 Libevent

Libevent 是用于编写高速可移植非阻塞 IO 应用的库，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。（`bufferevent` 接口有多个后端，可以采用系统能够提供的更快的非阻塞 IO 方式，如 Windows 中的 IOCP。）
- `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。

2. 库

创建 libevent 时， 默认安装下列库：

- `libevent_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。
- `libevent_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`，除非是以多线程方式使用 `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，除非是进行加密通信。

3. 头文件

`libevent` 公用头文件都安装在 `event2` 目录中，分为三类：

- API 头文件：定义 `libevent` 公用接口。这类头文件没有特定后缀。
- 兼容头文件：为已废弃的函数提供兼容的头部包含定义。不应该使用这类头文件，除非是在移植使用较老版本 `libevent` 的程序时。
- 结构头文件：这类头文件以相对不稳定的布局定义各种结构体。这些结构体中的一些是为了提供快速访问而暴露；一些是因为历史原因而暴露。直接依赖这类头文件中的任何结构体都会破坏程序对其他版本 `libevent` 的二进制兼容性，有时候是以非常难以调试的方式出现。这类头文件具有后缀“`_struct.h`”。

（还存在不在 `event2` 目录中的较老版本 `libevent` 的头文件，请参考下节：如果需要使用老版本 `libevent`）

4. 如果需要使用老版本 `libevent`

`libevent 2.0` 以更合理的、不易出错的方式修正了 API。如果可能，编写新程序时应该使用 `libevent 2.0`。但是有时候可能需要使用较老的 API，例如在升级已存的应用时，或者支持因

为某些原因不能安装2.0或者更新版本 libevent 的环境时。

较老版本的 libevent 头文件较少，也不安装在 event2 目录中。

OLD HEADER...	...REPLACED BY CURRENT HEADERS
event.h	event2/event*.h, event2/buffer*.h, event2/bufferevent*.h event2/tag*.h
evdns.h	event2/dns*.h
evhttp.h	event2/http*.h
evrpc.h	event2/rpc*.h
evutil.h	event2/util*.h

在2.0以及以后版本的 libevent 中，老的头文件仍然会作为新头文件的封装而存在。

其他关于使用较老版本的提示：

- 1.4版之前只有一个库 libevent，它包含现在分散到 libevent_core 和 libevent_extra 中的所有功能。
- 2.0版之前不支持锁定：只有确定不同时在多个线程中使用同一个结构体时，libevent 才是线程安全的。

下面的节还将讨论特定代码区域可能遇到的已经废弃的 API。

5. 关于版本状态的提示

1.4.7及以前版本应该被认为是完全废弃的。1.3之前的版本应该被认为是充满 bug 的。

（此外，不要向 libevent 维护者发送任何关于1.4.x 或者更早版本的新特征，这些版本被认为是稳定的发布版本。如果在1.3x 或者更早版本中发现 bug，在报告之前请确定在最新的稳定发布版本中问题仍然存在：后续发布可能已经解决了问题。）

第一章：设置 libevent

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

libevent 有一些被整个进程共享的、影响整个库的全局设置。必须在调用 libevent 库的任何其他部分之前修改这些设置，否则，libevent 会进入不一致的状态。

1. Libevent 中的日志消息

libevent 可以记录内部错误和警告。如果编译进日志支持，还会记录调试信息。默认配置下这些信息被写到 `stderr`。通过提供定制的日志函数可以覆盖默认行为。

接口

```
#define _EVENT_LOG_DEBUG 0
#define _EVENT_LOG_MSG 1
#define _EVENT_LOG_WARN 2
#define _EVENT_LOG_ERR 3
typedef void (*event_log_cb)(int severity, const char *msg);
void event_set_log_callback(event_log_cb cb);
```

要覆盖 libevent 的日志行为，编写匹配 `event_log_cb` 签名的定制函数，将其作为参数传递给 `event_set_log_callback()`。随后 libevent 在日志信息的时候，将会把信息传递给你提供的函数。再次调用 `event_set_log_callback()`，传递参数 `NULL`，就可以恢复默认行为。

示例

..... 省略示例代码

注意

在用户提供的 `event_log_cb` 回调函数中调用 libevent 函数是不安全的。比如说，如果试图编写一个使用 `bufferevent` 将警告信息发送给某个套接字的日志回调函数，可能会遇到奇怪而难以诊断的 bug。未来版本 libevent 的某些函数可能会移除这个限制。

这个函数在`<event2/event.h>`中声明，在 libevent 1.0c 版本中首次出现。

2. 处理致命错误

libevent 在检测到不可恢复的内部错误时的默认行为是调用 `exit()` 或者 `abort()`，退出正在运行的进程。这类错误通常意味着某处有 bug：要么在你的代码中，要么在 **libevent** 中。

如果希望更优雅地处理致命错误，可以为 **libevent** 提供在退出时应该调用的函数，覆盖默认行为。

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 **libevent** 在遇到致命错误时应该调用的函数，将其传递给 `event_set_fatal_callback()`。随后 **libevent** 在遇到致命错误时将调用你提供的函数。

你的函数不应该将控制返回到 **libevent**：这样做可能导致不确定的行为。为了避免崩溃，**libevent** 还是会退出。你的函数被不应该调用其它 **libevent** 函数。

这些函数声明在`<event2/event.h>`中，在 **libevent 2.0.3-alpha** 版本中首次出现。

3. 内存管理

默认情况下，**libevent** 使用 C 库的内存管理函数在堆上分配内存。通过提供 `malloc`、`realloc` 和 `free` 的替代函数，可以让 **libevent** 使用其他的内存管理器。希望 **libevent** 使用一个更高效的分配器时；或者希望 **libevent** 使用一个工具分配器，以便检查内存泄漏时，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));
```

这里有个替换 **libevent** 分配器函数的示例，它可以计算已经分配的字节数。实际应用中可能需要添加锁，以避免运行在多个线程中时发生错误。

示例

..... 省略示例代码

注意

- 替换内存管理函数影响 libevent 随后的所有分配、调整大小和释放内存操作。所以，必须保证在调用任何其他 libevent 函数之前进行替换。否则，libevent 可能用你的 free 函数释放用 C 库的 malloc 分配的内存。
- 你的 malloc 和 realloc 函数返回的内存块应该具有和 C 库返回的内存块一样的地址对齐。
- 你的 realloc 函数应该正确处理 realloc(NULL,sz) (也就是当作 malloc(sz) 处理)
- 你的 realloc 函数应该正确处理 realloc(ptr,0) (也就是当作 free(ptr) 处理)
- 你的 free 函数不必处理 free(NULL)
- 你的 malloc 函数不必处理 malloc(0)
- 如果在多个线程中使用 libevent，替代的内存管理函数需要是线程安全的。
- libevent 将使用这些函数分配返回给你的内存。所以，如果要释放由 libevent 函数分配和返回的内存，而你已经替换 malloc 和 realloc 函数，那么应该使用替代的 free 函数。

`event_set_mem_functions` 函数声明在 `<event2/event.h>` 中，在 libevent 2.0.1-alpha 版本中首次出现。

可以在禁止 `event_set_mem_functions` 函数的配置下编译 libevent。这时候使用 `event_set_mem_functions` 将不会编译或者链接。在 2.0.2-alpha 及以后版本中，可以通过检查是否定义了 `EVENT_SET_MEM_FUNCTIONS_IMPLEMENTED` 宏来确定 `event_set_mem_functions` 函数是否存在。

4. 锁和线程

编写多线程程序的时候，在多个线程中同时访问同样的数据并不总是安全的。

libevent 的结构体在多线程下通常有三种工作方式：

- 某些结构体内地是单线程的：同时在多个线程中使用它们总是不安全的。
- 某些结构体具有可选的锁：可以告知 libevent 是否需要在多个线程中使用每个对象。
- 某些结构体总是锁定的：如果 libevent 在支持锁的配置下运行，在多个线程中使用它们总是安全的。

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，那么你是幸运的：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
#ifndef _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` 函数要求程序链接到 `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_debuging(void);
```

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。

这个函数是在2.0.4-alpha 版本新引入的。

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。

这些调试函数在 libevent 2.0.4-alpha 版本中加入。

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

```
#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);
```

接口

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

..... 省略示例代码

示例：运行时检测

..... 省略示例代码

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。

第二章：创建 `event_base`

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

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()`。

2.1 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`。

2.2 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_P0 = 0x04,
};

int event_config_require_features(struct event_config *cfg,
                                  enum event_method_feature feature);

enum event_base_config_flag {
    EVENT_BASE_FLAG_NOLOOP = 0x01,
    EVENT_BASE_FLAG_TIMEOUT = 0x02,
    EVENT_BASE_FLAG_STARTUP_DOCK = 0x04,
    EVENT_BASE_FLAG_NO_CACHE_TIME = 0x08,
};

int event_config_set_flag(struct event_config *cfg,
                         enum event_base_config_flag flag);
```

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 不使用不能提供所有指定特征的后端。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。

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。

2.3 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也有用。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。

示例

```
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. */
}
```

这些函数和类型在`<event2/event.h>`中声明。

`EVENT_BASE_FLAG_IGNORE_ENV` 标志首次出现在 2.0.2-alpha 版本。
`event_config_set_num_cpus_hint()` 函数是 2.0.7-rc 版本新引入的。本节的其他内容首次出现在 2.0.1-alpha 版本。

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。

3.1 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。

3.2 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

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

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。

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。

7 废弃的 **event_base** 函数

老版本的 libevent 严重依赖“当前”event_base 的概念。“当前”event_base 是一个由所有线程共享的全局设置。如果忘记指定要使用哪个 event_base，则得到的是当前的。因为 event_base 不是线程安全的，这很容易导致错误。

老版本的 libevent 没有 event_base_new ()，而有：

接口

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

这个函数的工作与 event_base_new () 类似，它将分配的 event_base 设置成当前的。没有其他方法改变当前 event_base。

本文描述的函数有一些用于操作当前 event_base 的变体，这些函数与新版本函数的行为类似，只是它们没有 event_base 参数。

Current function	Obsolete current-base version
event_base_priority_init()	event_priority_init()
event_base_get_method()	event_get_method()

第三章：与事件循环一起工作

译自 http://www.wangafu.net/~nickm/libevent-book/Ref3_eventloop.html

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

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

默认情况下, `event_base_loop()` 函数运行 `event_base` 直到其中没有已经注册的事件为止。执行循环的时候, 函数重复地检查是否有任何已经注册的事件被触发 (比如说, 读事件的文件描述符已经就绪, 可以读取了; 或者超时事件的超时时间即将到达)。如果有事件被触发, 函数标记被触发的事件为“激活的”, 并且执行这些事件。

在 `flags` 参数中设置一个或者多个标志就可以改变 `event_base_loop()` 的行为。如果设置了 `EVLOOP_ONCE`, 循环将等待某些事件成为激活的, 执行激活的事件直到没有更多的事件可以执行, 然后返回。如果设置了 `EVLOOP_NONBLOCK`, 循环不会等待事件被触发: 循环将仅仅检测是否有事件已经就绪, 可以立即触发, 如果有, 则执行事件的回调。

完成工作后, 如果正常退出, `event_base_loop()` 返回0; 如果因为后端中的某些未处理错误而退出, 则返回-1。

为帮助理解, 这里给出 `event_base_loop()` 的算法概要:

伪代码

```
while (any events are registered with the loop) {  
    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()`。所以，`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_looppbreak()` 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_looppbreak(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）；后者却仅仅停止目前正在运行的循环，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循环是因为调用 `event_base_loopexit()` 或者 `event_base_break()` 而退出的时候返回 `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_looppbreak()`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()`函数设置 `tv_out` 参数的值为缓存的时间。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。

这个函数是 `libevent 2.0.4-alpha` 新引入的。

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。

5. 废弃的事件循环函数

前面已经讨论过，老版本的 `libevent` 具有“当前”`event_base` 的概念。

本文讨论的某些事件循环函数具有操作当前 `event_base` 的变体。除了没有 `base` 参数外，这些函数跟当前新版本函数的行为相同。

Current function	Obsolete current-base version
event_base_dispatch()	event_dispatch()
event_base_loop()	event_loop()
event_base_loopexit()	event_loopexit()
event_base_loopbreak()	event_loopbreak()

注意

2.0版本之前的 `event_base` 是不支持锁的，所以这些函数并不是完全线程安全的：不允许在执行事件循环的线程之外的其他线程中调用`_loopbreak()`或者`_loopexit()`函数。

第四章：与事件一起工作

译自 http://www.wangafu.net/~nickm/libevent-book/Ref4_event.html

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。在未决状态下，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]",
           (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”节。

- EV_PERSIST

表示事件是“持久的”，请看下面的“关于事件持久性”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 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, what, tv_out) \
    event_pending((ev), (what), (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。

1.5 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, callback, 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。

1.6 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

对于大多数应用来说，这些开销是非常小的。所以，除非确定在堆上分配事件导致了严重的性能问题，应该坚持使用 `event_new()`。如果将来版本中的 `event` 结构体更大，不使用 `event_new()` 可能会导致难以诊断的错误。

不在堆上分配 `event` 具有破坏与其他版本 `libevent` 二进制兼容性的风险：其他版本中的 `event` 结构体大小可能不同。

接口

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_del(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);

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. 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()`。

6. 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);
```

这个函数让事件 `ev` 带有标志 `what` (EV_READ、EV_WRITE 和 EV_TIMEOUT 的组合) 成为激活的。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。

这个函数定义在<event2/event.h>中，从0.3版本就存在了。

7. 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 $O(\log N)$ 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 $O(\log N)$ 堆。可以在代码中自由地复制这个特别的 `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。

8. 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);
}

```

event_initialized () 函数从0.3版本就存在了。

9 废弃的事件操作函数

2.0版本之前的 libevent 没有 event_assign () 或者 event_new ()。替代的是将事件关联到“当前”event_base 的 event_set ()。如果有多个 event_base，需要记得调用 event_base_set () 来确定事件确实是关联到当前使用的 event_base 的。

接口

```

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

```

除了使用当前 `event_base` 之外, `event_set()` 跟 `event_assign()` 是相似的。`event_base_set()` 用于修改事件所关联到的 `event_base`。

`event_set()` 具有一些用于更方便地处理定时器和信号的变体: `evtimer_set()` 大致对应 `evtimer_assign()`; `evsignal_set()` 大致对应 `evsignal_assign()`。

2.0 版本之前的 libevent 使用“`signal_`”作为用于信号的 `event_set()` 等函数变体的前缀, 而不是“`evsignal_`”(也就是说, 有 `signal_set()`、`signal_add()`、`signal_del()`、`signal_pending()` 和 `signal_initialized()`)。远古版本(0.6 版之前)的 libevent 使用“`timeout_`”而不是“`evtimer_`”。因此, 做代码考古 (`code archeology`) (注: 这个翻译似乎不正确, 是否有更专业的术语? 比如说, “代码复审”) 时可能会看到 `timeout_add()`、`timeout_del()`、`timeout_initialized()`、`timeout_set()` 和 `timeout_pending()` 等等。

较老版本 (2.0 版之前) 的 libevent 用宏 `EVENT_FD()` 和 `EVENT_SIGNAL()` 代表现在的 `event_get_fd()` 和 `event_get_signal()` 函数。这两个宏直接检查 `event` 结构体的内容, 所以会妨碍不同版本之间的二进制兼容性。在 2.0 以及后续版本中, 这两个宏仅仅是 `event_get_fd()` 和 `event_get_signal()` 的别名。

因为 2.0 之前的版本不支持锁, 所以在运行 `event_base` 的线程之外的任何线程调用修改事件状态的函数都是不安全的。这些函数包括 `event_add()`、`event_del()`、`event_active()` 和 `event_base_once()`。

有一个 `event_once()` 与 `event_base_once()` 相似, 只是用于当前 `event_base`。

2.0 版本之前 `EV_PERSIST` 标志不能正确地操作超时。标志不会在事件激活时复位超时值, 而是没有任何操作。

2.0 之前的版本不支持同时添加多个带有相同 `fd` 和 `READ/WRITE` 标志的事件。也就是说, 在每个 `fd` 上, 某时刻只能有一个事件等待读取, 也只能有一个事件等待写入。

第五章：辅助类型和函数

译自 http://www.wangafu.net/~nickm/libevent-book/Ref5_evutil.html

<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 `true`，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 $t1 \leq t2$ ？”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_gettimeofday()` 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 中的 `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（译者注：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]:端口号 (如[ffff::]:80)

| 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。

如果程序运行在可能会放弃权限的环境中（比如说，通过执行 `chroot()`），在放弃权限前应该调用 `evutil_secure_rng_init()`。

可以调用 `evutil_secure_rng_add_bytes()` 向熵池加入更多随机字节，但通常不需要这么做。

这些函数是 2.0.4-alpha 版本引入的。

第六章：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`：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`：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`：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`：相互传输数据的两个 `bufferevent`。

注意：截至 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，请调用 `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。

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 occurred 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_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。

这个函数在2.0.2-alpha 版引入。

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;
}

```

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。

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);
```

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。**

这个函数由 1.4.4 版引入。类型名 bufferevent_data_cb 和 bufferevent_event_cb 由 2.0.2-alpha 版引入。

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 *bl = 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(bl, EV_READ, 128, 0);

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

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

```

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。(Reading and writing data from the network does you no good if you can't look at it.Bufferevents give you these methods to give them data to write, and to get the data to read.)

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。

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/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 *infol;

    infol = malloc(sizeof(struct info));
    infol->name = "buffer 1";
    infol->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, infol);

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

```

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，基于套接字的 `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);
```

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1.0版引入。

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。

9. 已废弃的 **bufferevent** 功能

从1.4到2.0版，**bufferevent** 的后端代码一直在进行修订。在老的接口中，访问 **bufferevent** 结构体的内部是很平常的，并且还会使用依赖于这种访问的宏。

更复杂的是，老的代码有时候将“**evbuffer**”前缀用于 **bufferevent** 功能。

这里有一个在2.0版之前使用过的东西的概要：

Current name	Old name
bufferevent_data_cb	evbuffercb
bufferevent_event_cb	everrorcb
BEV_EVENT_READING	EVBUFFER_READ
BEV_EVENT_WRITE	EVBUFFER_WRITE
BEV_EVENT_EOF	EVBUFFER_EOF
BEV_EVENT_ERROR	EVBUFFER_ERROR
BEV_EVENT_TIMEOUT	EVBUFFER_TIMEOUT
bufferevent_get_input(b)	EVBUFFER_INPUT(b)
bufferevent_get_output(b)	EVBUFFER_OUTPUT(b)

老的函数定义在 `event.h` 中，而不是在 `event2/bufferevent.h` 中。

如果仍然需要访问 `bufferevent` 结构体内部的某些公有部分，可以包含 `event2/bufferevent_struct.h`。但是不建议这么做：不同版本的 Libevent 中 `bufferevent` 结构体的内容可能会改变。本节描述的宏和名字只有在包含了 `event2/bufferevent_compat.h` 时才能使用。

较老版本中用于设置 `bufferevent` 的接口有所不同：

接口

```
struct bufferevent *bufferevent_new(evutil_socket_t fd,
    evbuffercb readcb, evbuffercb writecb, everrorcb errorcb, void *cbarg);
int bufferevent_base_set(struct event_base *base, struct bufferevent *bufev);
```

`bufferevent_new()` 函数仅仅在已经废弃的“默认”`event_base` 上创建一个套接字 `bufferevent`。调用 `bufferevent_base_set()` 可以调整套接字 `bufferevent` 的 `event_base`。

较老版本不使用 `timeval` 结构体设置超时，而是使用秒数：

接口

```
void bufferevent_settimeout(struct bufferevent *bufev,
    int timeout_read, int timeout_write);
```

最后要指出的是，2.0之前版本中的 `evbuffer` 实现是极其低效的，这对将 `bufferevent` 用于高性能应用是一个问题。

Bufferevent：高级话题

译自 http://www.wangafu.net/~nickm/libevent-book/Ref6a_advanced_bufferevents.html

本章描述 **bufferevent** 的一些对通常使用不必要的高级特征。如果只想学习如何使用 **bufferevent**，可以跳过这一章，直接阅读下一章。

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)
```

有时候在给出了对的一个成员时，需要获取另一个成员，这时候可以使用 `bufferevent_pair_get_partner()`。如果 `bev` 是对的成员，而且对的另一个成员仍然存在，函数将返回另一个成员；否则，函数返回 `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. **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。

3.1 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。

3.2 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。

3.3 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。

3.4 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。

3.5 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。

3.6 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。

3.7 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。

4. `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。

4.1 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);
```

可以创建两种类型的 SSL bufferevent: 基于过滤器的、在另一个底层 bufferevent 之上进行通信的 buffervent; 或者基于套接字的、直接使用 OpenSSL 进行网络通信的 bufferevent。这两种 bufferevent 都要求提供 SSL 对象及其状态描述。如果 SSL 当前作为客户端在进行协商, 状态应该是 BUFFEREVENT_SSL_CONNECTING; 如果作为服务器在进行协商, 则是 BUFFEREVENT_SSL_ACCEPTING; 如果 SSL 握手已经完成, 则状态是 BUFFEREVENT_SSL_OPEN。

接受通常的选项。BEV_OPT_CLOSE_ON_FREE 表示在关闭 openssl bufferevent 对象的时候同时关闭 SSL 对象和底层 fd 或者 bufferevent。

创建基于套接字的 bufferevent 时, 如果 SSL 对象已经设置了套接字, 就不需要提供套接字了: 只要传递-1就可以。也可以随后调用 bufferevent_setfd () 来设置。

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安全问题。

evbuffer：缓冲 IO 实用功能

译自 http://www.wangafu.net/~nickm/libevent-book/Ref7_evbuffer.html

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 datlen 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);
```

`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。

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 = ntohs(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;
}

```

这个函数首次出现在2.0.5-alpha 版本。

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
};
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4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。

（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1.4.14-stable 及以后版本中可用。

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_size_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 data 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_size_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：

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。

注意

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。

清空数据或者释放 `evbuffer` 时文件描述符将被关闭。

这一节描述的函数都在2.0.1-alpha 版本中引入。`evbuffer_add_referece()` 则从2.0.2-alpha 版本开始存在。

17. 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。

18. 废弃的 `evbuffer` 函数

2.0版对 `evbuffer` 接口进行了很多修改。在此之前，每个 `evbuffer` 实现为一个连续的内存块，访问效率是非常低的。

`event.h` 头文件用于暴露 `evbuffer` 结构体的内部，但该结构体已经不可用了，因为对于依赖于它们的代码，1.4和2.0版之间的改动太大了。

要访问 `evbuffer` 中的字节数，可以使用 `EVBUFFER_LENGTH()` 宏；而实际数据可以通过 `EVBUFFER_DATA()` 来访问。这两个宏都在 `event2/buffer_compat.h` 中。然而，请注意：

`EVBUFFER_DATA(b)`只是 `evbuffer_pullup(b,-1)`的别名，其开销可能非常大。

其他废弃的接口有：

废弃的接口

```
char *evbuffer_readline(struct evbuffer *buffer);
unsigned char *evbuffer_find(struct evbuffer *buffer,
    const unsigned char *what, size_t len);
```

`evbuffer_readline()` 函数的工作方式类似于当前的 `evbuffer_readln(buffer,NULL,EVBUFFER_EOL_ANY)`。

`evbuffer_find()`将在缓冲区中搜索字符串的首次出现，返回其指针。与 `evbuffer_search()`不同的是，它只能找到第一个字符串。为兼容使用这个函数的老代码，这个函数会线性化到被定位字符串末尾的整个缓冲区。

回调函数也有不同：

废弃的接口

```
typedef void (*evbuffer_cb)(struct evbuffer *buffer,
    size_t old_len, size_t new_len, void *arg);
void evbuffer_setcb(struct evbuffer *buffer, evbuffer_cb cb, void *cbarg);
```

`evbuffer` 某时刻只能有一个回调，设置新的回调会禁止先前的回调，设置回调为 `NULL` 是最佳的禁止回调的方法。

回调函数不使用 `evbuffer_cb_info` 结构体，而是使用 `evbuffer` 的原长度和新长度。因此，如果 `old_len` 大于 `new_len`，数据被抽取；如果 `new_len` 大于 `old_len`，数据被添加。回调不可能延迟，所以添加和删除操作不可能合并到单个回调的执行中。

这里给出的废弃函数依然在 `event2/buffer_compatible.h` 中，依然可用。

连接监听器：接受 TCP 连接

译自 http://www.wangafu.net/~nickm/libevent-book/Ref8_listener.html

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。

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。

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. 借测错误

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>
#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.\n"
            "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;
}

```

使用 libevent 的 DNS：高层和底层功能

译自 http://www.wangafu.net/~nickm/libevent-book/Ref9_dns.html

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](https://www.example.com)），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_ADDRCFG](#)

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(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。

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);

#ifndef 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。

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：](#)

`n ndots:INTEGER`

用于配置搜索，请参考上面的“`search`”，默认值是1。

`n timeout:FLOAT`

等待 DNS 服务器响应的时间，单位是秒。默认值为5秒。

`n 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。

`n 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。

`n attempts:INT`

在放弃之前重新传输多少次 DNS 请求？默认值是 XXX。

`n 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。

（这段原文不易理解，译文可能很不准确。这里给出原文：If nonzero, we randomize the case on outgoing DNS requests and make sure that replies have the same case as our requests. This so-called "0x20 hack" can help prevent some otherwise simple active events against DNS.）

`n 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。

`n 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。

`n 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。

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。

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_ip4(struct evdns_base *base,
    const char *name, int flags, evdns_callback_type callback, void *ptr);
struct evdns_request *evdns_base_resolve_ip6(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_ip6(
    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：

DNS 错误码

错误码	意义
<code>DNS_ERR_NONE</code>	没有错误
<code>DNS_ERR_FORMAT</code>	服务器不识别查询请求

DNS_ERR_SERVERFAILED	服务器内部错误
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
DNS_ERR_CANCEL	用户请求取消查询

可以用下述函数将错误码转换成错误描述字符串：

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。

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。

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);
```

有时候需要知道某特定 DNS 请求来自何方，这时调用 evdns_server_request_get_requesting_addr() 就可以了。应该传入有足够的存储空间以容量地址的 sockaddr：建议使用 sockaddr_storage 结构体。

这个函数在 1.3 版本中引入。

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。

DNS 服务器示例

```

#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.ip6.arpa" \
                           "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 1;
    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 1;

    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;
}

```

6. 废弃的 DNS 接口

废弃的接口

```

void evdns_base_search_ndots_set(struct evdns_base *base,
                                  const int ndots);
int evdns_base_nameserver_add(struct evdns_base *base,
                             unsigned long int address);
void evdns_set_random_bytes_fn(void (*fn)(char *, size_t));

struct evdns_server_port *evdns_add_server_port(evutil_socket_t socket,
                                                int flags, evdns_request_callback_type callback, void *user_data);

```

`evdns_base_search_ndots_set()`等价于使用 `evdns_base_set_option()`设置 `ndots` 选项。

除了只能添加 IPv4 地址的名字服务器外，`evdns_base_nameserver_add()`函数的行为与 `evdns_base_nameserver_ip_add()`相同。特别的是，`evdns_base_nameserver_add()`要求

网络字节序的四字节地址。

2.0.1-alpha 版本之前，不能为 DNS 服务端口指定 `event_base`。通过 `evdns_add_server_port()`添加的服务端口只能使用默认的 `event_base`。

从版本 2.0.1-alpha 到 2.0.3-alpha，可以使用 `evdns_set_random_bytes_fn()`，而不是 `evdns_set_transssaction_id_fn()`，来指定用于产生随机数的函数。这个函数现在没有效果了，因为 libevent 有自己的安全的随机数发生器了。

`DNS_QUERY_NO_SEARCH` 标志曾经称作 `DNS_NO_SEARCH`。

2.0.1-alpha 版本之前，没有单独的 `evdns_base` 记号：`evdns` 子系统中的所有信息都是全局存储的，操作这些信息的函数不需要 `evdns_base` 参数。这些函数现在都废弃了，但是还声明在 `event2/dns_compat.h` 中。它们通过一个单独的全局 `evdns_base` 实现，通过 2.0.3-alpha 版本引入的 `evdns_get_global_base()` 可以访问这个 `evdns_base`。

Current function	Obsolete global- <code>evdns_base</code> version
<code>event_base_new()</code>	<code>evdns_init()</code>
<code>evdns_base_free()</code>	<code>evdns_shutdown()</code>
<code>evdns_base_nameserver_add()</code>	<code>evdns_nameserver_add()</code>
<code>evdns_base_count_nameservers()</code>	<code>evdns_count_nameservers()</code>
<code>evdns_base_clear_nameservers_and_suspend()</code>	<code>evdns_clear_nameservers_and_suspend()</code>
<code>evdns_base_resume()</code>	<code>evdns_resume()</code>
<code>evdns_base_nameserver_ip_add()</code>	<code>evdns_nameserver_ip_add()</code>
<code>evdns_base_resolve_ipv4()</code>	<code>evdns_resolve_ipv4()</code>
<code>evdns_base_resolve_ipv6()</code>	<code>evdns_resolve_ipv6()</code>
<code>evdns_base_resolve_reverse()</code>	<code>evdns_resolve_reverse()</code>
<code>evdns_base_resolve_reverse_ipv6()</code>	<code>evdns_resolve_reverse_ipv6()</code>
<code>evdns_base_set_option()</code>	<code>evdns_set_option()</code>
<code>evdns_base_resolv_conf_parse()</code>	<code>evdns_resolv_conf_parse()</code>
<code>evdns_base_search_clear()</code>	<code>evdns_search_clear()</code>
<code>evdns_base_search_add()</code>	<code>evdns_search_add()</code>
<code>evdns_base_search_ndots_set()</code>	<code>evdns_search_ndots_set()</code>
<code>evdns_base_config_windows_nameservers()</code>	<code>evdns_config_windows_nameservers()</code>

如果 `evdns_config_windows_nameservers()` 可用，则 EVDNS_CONFIG_WINDOWS_NAMESERVERS_IMPLEMENTED 宏会被定义。