# Библиотека libevent для асинхронного неблокирующего ввода/вывода: Часть 3. Ввод/вывод с буферизацией

Алексей Снастин
независимый разработчик ПО
начальник отдела

19.07.2011

В третьей статье речь идёт о механизмах буферизации ввода/вывода.

В этом цикле статей рассматривается библиотека libevent, предназначенная для обработки оповещений о событиях и организации асинхронного ввода/вывода. В первой статье описывается общая структура библиотеки. Вторая статья посвящена основам использования libevent: наибольшее внимание уделено обработке событий. В третьей статье речь идёт о механизмах буферизации ввода/вывода. Тема четвёртой, заключительной статьи - применение libevent в сетевых приложениях.

## Введение

Достаточно часто для повышения эффективности операций ввода/вывода данных требуется использование механизмов буферизации. Библиотека libevent предоставляет удобный набор средств для реализации и применения этих механизмов.

## 1. Основные концепции

Идея относительно проста: данные помещаются в буфер (или извлекаются из буфера) в ответ на определённое событие, произошедшее на объекте, для которого был создан этот буфер. Библиотека libevent предоставляет обобщённую, можно даже сказать - универсальную реализацию этого процесса, которая в терминах библиотеки обозначается, как bufferevent ("событие буфера"). Объект типа bufferevent состоит из пункта передачи данных (например, это может быть сокет), буфера чтения и буфера записи. В отличие от простых событий, которые активизируют функции обратного вызова в тот момент, когда пункт передачи данных готов к чтению или записи, bufferevent обращается к функциям обратного вызова только в том случае, если в буфере содержится определённое количество данных, готовых для чтения или записи.

В настоящее время объекты bufferevent работают только с потоковыми протоколами, такими как TCP. Поддержка датаграмм (протоколов типа UDP) планируется в будущем.

Библиотека libevent позволяет работать с несколькими типами объектов bufferevent, которые тем не менее используют общий интерфейс:

- Объект bufferevent на основе сокета (socket-based bufferevent) - все операции чтения/записи выполняются на сокете;
- Фильтрующий объект bufferevent (filtering bufferevent) - объект выполняет предварительную обработку данных перед передачей их в объект более низкого уровня, например, сжатие или шифрование данных;
- Парные объекты bufferevent (paired bufferevent) - два объекта, которые обмениваются данными друг с другом;
- Объект bufferevent с асинхронным ВВ (asynchronous-IO bufferevent) - здесь для приёма/передачи данных используется интерфейс Windows IOCP (только для платформы Windows).

## 2. Механизм bufferevent

Как уже было отмечено ранее, каждый объект bufferevent содержит буфер ввода и буфер вывода. Для буферов определён специальный тип struct evbuffer. Все операции передачи данных выполняются только через буферы ввода/вывода.

Набор функций для работы с буферами будет рассмотрен несколько позже.

## 2.1. Обратные вызовы и пороговые значения

Для каждого объекта bufferevent существуют два обратных вызова: обратный вызов чтения и обратный вызов записи. По умолчанию обратный вызов чтения выполняется в тот момент, когда из пункта передачи данных в буфер чтения передаются какие-либо данные (вне зависимости от их количества), а обратный вызов записи выполняется, если буфер записи пуст после вывода данных в пункт передачи. Такая схема не всегда удобна в реальных приложениях, поэтому разработчик может написать собственные функции обратного вызова и определить пороговые значения количества данных (watermark) в буферах, при которых будут срабатывать функции обратного вызова.

## 2.1.1. Типы пороговых значений

Read low-water mark - когда количество данных достигает этого уровня или превышает его, активизируется функция обратного вызова чтения. По умолчанию это значение установлено равным 0, поэтому появление любого количества данных в буфере приводит к обратному вызову чтения.

Read high-water mark - когда количество данных достигает этого уровня, объект bufferevnt приостанавливает операцию чтения из пункта передачи данных до тех пор, пока из буфера не будет извлечено достаточное количество данных для достижения уровня ниже этого порогового значения. По умолчанию ограничения уровня нет, поэтому операция чтения не прерывается.

Write low-water mark - когда количество данных достигает этого уровня или становится меньшим, активизируется функция обратного вызова записи. По умолчанию это значение

установлено равным 0, поэтому функция обратного вызова записи активизируется только если буфер пуст.

Write high-water mark - это пороговое значение не используется объектом bufferevent напрямую, но его регулирование имеет смысл в тех случаях, когда объект bufferevent используется в качестве пункта передачи данных для другого объекта bufferevent, например, при создании фильтра.

## 2.2. Особые ситуации

Для объекта bufferevent существует специальная функция обратного вызова для обработки "внештатных" событий, не связанных непосредственно с передачей данных. В заголовочном файле <event2/bufferevent.h> определены флаги, позволяющие получить информацию о причинах возникновения особой ситуации: во-первых, можно узнать, что данное событие произошло во время операции чтения (BEV_EVENT_READING 0x01) или во время операции записи (BEV_EVENT_WRITING 0x02). Также можно определить, что был обнаружен признак конца файла (BEV_EVENT_EOF 0x10), или произошёл критический сбой, то есть "невосстановимая ошибка" (BEV_EVENT_ERROR 0x20), или истёк заданный интервал таймаута (BEV_EVENT_TIMEOUT 0x40), или же запрашиваемое соединение было установлено (BEV_EVENT_CONNECTED 0x80).

## 2.3. Флаги настройки объекта bufferevent

При создании объекта bufferevent можно изменить некоторые его настройки с помощью флагов, определённых в перечислении bufferevent_options в заголовочном файле <event2/bufferevent.h>. Флаг BEV_OPT_CLOSE_ON_FREE позволяет при удалении объекта bufferevent автоматически закрывать соответствующий пункт передачи данных (сокет, дескриптор файла или другой объект bufferevent). В многопоточном приложении может оказаться необходимым флаг BEV_OPT_THREADSAFE, включающий защиту операций с данными с помощью блокировок. В некоторых ситуациях, когда возможна одновременная генерация целой серии взаимозависимых обратных вызовов, от переполнения стека может спасти флаг BEV_OPT_DEFER_CALLBACKS, позволяющий создать очередь отложенных обратных вызовов в основном цикле обработки событий. При установке флага BEV_OPT_UNLOCK_CALLBACKS все функции обратного вызова данного объекта bufferevent будут выполняться без установки блокировок. В рассматриваемой версии libevent при установке этого флага обязательно требуется и установка флага BEV_OPT_DEFER_CALLBACKS; возможно, в следующей версии это ограничение будет снято.

## 2.4. Работа с объектом bufferevents на основе сокета

Это самый простой тип объекта bufferevents, который создаётся с помощью функции

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

Объект bufferevent, как и обычное событие, также связывается с набором обрабатываемых событий base. Аргумент fd содержит дескриптор файла, соответствующий сокету. Если передаётся значение -1, то в дальнейшем необходимо будет определить корректный дескриптор файла. В последнем аргументе options содержится битовая комбинация флагов настройки, описанных в предыдущем разделе. Функция возвращает указатель на созданный объект bufferevent или NULL, если объект создать не удалось.

## 2.5. Установление соединения

Если для сокета объекта bufferevent соединение ещё не установлено, то это можно сделать, вызвав функцию

```
int bufferevent_socket_connect( struct bufferevent *buf_ev,
                                struct sockaddr *address,
                                int addr_len );
```

Аргументы address и addr_len полностью аналогичны аргументам стандартного системного вызова connect(). Если до этого в объекте bufferevent сокет не был определён, то вызов данной функции создаст новый потоковый (stream) сокет и сделает его неблокируемым. Для существующего сокета полагается, что соединение не было установлено, и выполняется операция установления соединения, которая может завершиться успешно (возвращается 0) или неудачно (возвращается -1).

Следует отметить, что данные в буфер вывода могут быть добавлены до установления соединения, это вполне допустимое состояние объекта bufferevent.

### Листинг 1. Пример создания объекта bufferevent и установления соединения

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

void bufev_callback( struct bufferevent *buf_ev, short events, void *ptr )
{
  if( events & BEV_EVENT_CONNECTED )
  {
    printf( "Соединение установлено\n" );
    /* Здесь можно было бы выполнить и более полезные операции,
     * например, чтение или запись данных
     */
  }
  else if( events & BEV_EVENT_ERROR )
  {
    exit(-1);
    /* Перед принудительным завершением программы при необходимости
     * выполняются "спасательные действия" (освобождение памяти,
     * закрытие файлов и т.д.), если это возможно
     */
  }
}

int main_loop( void )
```

```
{
  struct event_base *base;
  struct bufferevent *buf_ev;
  struct sockaddr_in sin;

  base = event_base_new();

  memset( &sin, 0, sizeof(sin) );
  sin.sin_addr.s_addr = inet_addr( "127.0.0.1" );
  sin.sin_port = htons( 8080 );

  buf_ev = bufferevent_socket_new( base, -1, BEV_OPT_CLOSE_ON_FREE );
  bufferevent_setcb( buf_ev, NULL, NULL, bufev_callback, NULL );

  if( bufferevent_socket_connect( buf_ev, (struct sockaddr *)&sin, sizeof(sin) ) < 0 )
  {
    /* Попытка установить соединение была неудачной */
    bufferevent_free( buf_ev ); /* сокет закроется автоматически; см. флаг при создании */
    return -1;
  }

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

## 2.6. Основные операции объекта bufferevent

В Листинге 1 были использованы некоторые функции, выполняющие операции, характерные для объекта bufferevent. Понять действие функции bufferevent_free( buf_ev ) можно без затруднений, а вот аргументы функции настройки обратного вызова необходимо разъяснить. Сначала следует обратить внимание на объявления функций обратного вызова для данных и для событий:

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

Установка функций обратного вызова выполняется с помощью функции, применённой в Листинге 1:

```
void bufferevent_setcb( struct bufferevent *buf_ev, bufferevent_data_cb read_cb,
                        bufferevent_data_cb write_cb, bufferevent_event_cb event_cb,
                        void *cb_arg );
```

Следует ещё раз отметить наличие отдельных функций обратного вызова для чтения, для записи и для событий, не связанных непосредственно с данными. В Листинге 1 вместо имён функций обратного вызова для чтения и записи были переданы значения NULL. Это означает, что в данном случае используются стандартные системные вызовы readv и writev.

Разрешать и запрещать события EV_READ, EV_WRITE или (EV_READ|EV_WRITE) можно с помощью функций

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

Весьма важным инструментом является функция установки пороговых значений (watermarks):

```
void bufferevent_setwatermarks( struct bufferevent *buf_ev, short events,
                                size_t low_mark, size_t high_mark );
```

Доступ к содержимому буферов ввода и вывода предоставляют следующие функции:

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

И всё же самое главное - ввод/вывод данных:

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

Особенность функций с суффиксом _buffer состоит в том, что bufferevent_write_buffer() добавляет всё содержимое buf в буфер вывода (после этого buf становится пустым), а bufferevent_read_buffer() перемещает всё содержимое буфера ввода в заданный буфер buf (после этого буфер ввода становится пустым).

Учтена и необходимость установки таймаутов для операций ввода/вывода:

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

Передача значений NULL вместо указателей на объекты таймаутов отменяет их действие.

## 3. Механизм evbuffer

Механизм буферов библиотеки libevent достаточно прост: фактически буфер реализован в виде очереди (queue) байтов, в которую данные добавляются в конец очереди, а извлекаются (и удаляются) из начала очереди. Для буферов не существует функций предварительного планирования операций ввода/вывода и функций инициализации ввода/вывода по готовности данных, присущих рассмотренным выше объектам bufferevent.

## 3.1. Создание и удаление буфера

За создание буфера отвечает функция

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

Удалить ранее созданный буфер можно с помощью функции

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

## 3.2. Объекты evbuffer в многопоточной среде

Основным условием использования буферов в многопоточных приложениях является возможность применения механизма блокировок, и в libevent эту возможность обеспечивают функции

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

По умолчанию механизм блокировок отключён, поэтому его необходимо активизировать. После этого можно устанавливать и снимать блокировки для отдельных объектов evbuffer.

## 3.3. Размер буфера

Размер буфера в байтах возвращает функция

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

Имеется возможность изменить размер самого последнего фрагмента памяти в буфере или добавить новый фрагмент памяти для того, чтобы буфер смог принять заданное количество байтов данных:

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

## 3.4. Добавление данных в буфер

Данные могут быть добавлены как простая последовательность байтов

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

или с форматированием по правилам функций стандартной библиотеки языка C printf и vprintf:

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

Набор функций для работы с буферами evbuffer столь обширен, что полный их обзор в рамках данной статьи просто невозможен. Можно ограничиться лишь тривиальным советом: необходимо изучать документацию.

## Заключение

Средства буферизации библиотеки libevent тщательно проработаны и удобны в использовании. Разработчику предоставлена возможность буквально в несколько строк организовать набор буферов ввода/вывода, обработку событий, связанных с этим набором, установить требуемые интервалы таймаутов и пороговые значения заполненности буферов.

В данной статье рассмотрены механизмы буферизации ввода/вывода. Тема четвёртой, заключительной статьи - применение libevent в сетевых приложениях.

# Об авторе

**Алексей Снастин**

Алексей Снастин - независимый разработчик ПО, консультант и переводчик с английского языка технической и учебной литературы по ИТ. Принимал участие в разработке сетевых офисных приложений типа клиент/сервер на языке C в среде Linux.