Библиотека libevent для асинхронного неблокирующего ввода/вывода: Часть 2. Основы использования

Алексей Снастин
независимый разработчик ПО
начальник отдела

14.07.2011

Первая статья цикла, посвященного библиотеке libevent, предназначенной для обработки оповещений о событиях и организации асинхронного ввода/вывода, описывает общую структуру библиотеки.

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

В реальных приложениях чаще возникает необходимость обработки не одного, а нескольких различных событий. Библиотека libevent предоставляет разработчику возможность организовать своего рода пул таких ожидаемых событий и единый универсальный цикл для их обработки.

1. Инициализация цикла обработки событий

Прежде чем запускать цикл обработки событий, необходимо позаботиться о его основе, то есть об объекте наблюдения, который в терминах libevent соответствует структуре event_base.

1.1. Создание пула отслеживаемых событий

Каждая структура типа event_base содержит некоторый набор событий. На основе этой структуры выполняется опрос, целью которого является выявление произошедших (активных) событий из заданного набора.

Если предварительные настройки event_base позволяют использовать механизм блокировок для синхронизации, то возможен корректный доступ к такой структуре из нескольких потоков. При этом не следует забывать о том, что цикл обработки событий тем не менее может быть инициализирован только в одном потоке.

Каждому экземпляру структуры event_base соответствует определённый "метод", то есть внутренний механизм, который используется для определения активизированных событий. Пользователь может выбрать один из поддерживаемых системных (которые зависят от текущей платформы) механизмов: select, poll, epoll, kqueue, devpoll, evport или win32.

1.2. Создание структуры event_base с параметрами по умолчанию

Для создания экземпляра структуры event_base, инициализируемого значениями, принятыми по умолчанию, достаточно вызвать функцию, описываемую следующей сигнатурой:

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

В качестве внутреннего механизма мониторинга событий по умолчанию выбирается наиболее быстрый метод, доступный в текущей операционной системе. Функция возвращает указатель на экземпляр структуры, полностью готовой к дальнейшей работе. Разумеется, если при вызове не возникла ошибка, и вместо указателя на созданный объект функция не вернула NULL.

1.3. Создание структуры event_base с заданной конфигурацией

Для более детальной настройки основного набора обрабатываемых событий предназначена специальная структура event_config, которая содержит параметры для event_base. Объект с параметрами конфигурации создаётся при вызове функции

```
struct event_config *event_config_new( void );
```

После создания экземпляра структуры конфигурации вызываются функции из группы event_config для установки требуемых значений параметров. Например, отключить определённый внутренний механизм обработки событий можно с помощью функции (имя механизма передаётся в функцию в виде строки)

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

Кроме того, для метода обработки событий можно определить критерии его выбора. Набор критериев (определяемый перечислением enum event_method_feature; описание см. в документации) для выбора полностью соответствующего им механизма передаётся в функцию

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

При конфигурировании также можно сформировать набор флагов, предлагаемых перечислением enum event_base_config_flag:

- EVENT_BASE_FLAG_NOLOCK = 0x01 - запретить блокировки для создаваемого объекта event_base.
- EVENT_BASE_FLAG_IGNORE_ENV = 0x02 - не проверять переменные среды EVENT_* при выборе внутреннего механизма обработки событий. Этот флаг следует применять с большой осторожностью, поскольку выбор метода обработки событий без учёта текущей среды может привести к непредсказуемому поведению программы и затруднить её отладку.
- EVENT_BASE_FLAG_STARTUP_IOCP = 0x04 - флаг только для ОС Windows.
- EVENT_BASE_FLAG_NO_CACHE_TIME = 0x08 - проверять текущее время после каждого обратного вызова обработки таймаута, а не в момент готовности обратного вызова обработки таймаута (то есть, перед обратным вызовом) в основном цикле обработки событий.
- EVENT_BASE_FLAG_EPOLL_USE_CHANGELIST = 0x10 - при использовании механизма epoll этот флаг сообщает о возможности безопасного использования внутреннего кода библиотеки libevent, основанного на так называемом "списке изменений" ("changelist"). Это позволяет избежать избыточных системных вызовов в тех случаях, когда состояние одного и того же дескриптора файла изменяется более одного раза в интервале между вызовами метода обработки событий. Здесь следует отметить, что установка этого флага может ускорить выполнение программы, но при этом существует опасность возникновения ошибки ядра Linux, если какие-либо дескрипторы файлов дублировались с помощью системного вызова dup() или его вариантов.

Сформированный набор необходимых флагов передаётся в функцию

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

После того, как все параметры конфигурации заданы, необходимо создать базовую структуру набора событий с помощью функции

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

После этого структура конфигурации становится ненужной, и от неё можно избавиться:

```
void event_config_free( struct event_config *cfg );
```

В Листинге 1 приведён небольшой фрагмент кода, иллюстрирующий определение параметров конфигурации и создание соответствующей структуры событий.

Листинг 1. Создание структуры событий с заданной конфигурацией

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

/* Запретить использование механизма обработки событий select */
event_config_avoid_method( cfg, "select" );

/* Запретить блокировки для данного набора событий
 * (предположим, что программа однопоточная)
 */
event_config_set_flag( cfg, EVENT_BASE_FLAG_NOLOCK );

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

1.4. Удаление объекта event_base

После завершения работы с экземпляром event_base в большинстве случаев его необходимо удалить:

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

Особое внимание следует обратить на то, что эта функция не производит никаких действий с событиями, связанными с удаляемым экземпляром event_base, то есть не освобождает выделенную им память, не закрывает открытые ими сокеты и т.п. Об этом должен позаботиться сам разработчик.

1.5. Настройка уровней приоритетов событий

По умолчанию объект event_base поддерживает только один уровень приоритета. Это можно изменить с помощью функции

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

где n_priorities задаёт количество уровней приоритета для создаваемых событий (n_priorities >= 1). Наивысшим уровнем приоритета считается нулевой, а самому низкому соответствует значение n_priorities-1.

В библиотеке libevent определена константа EVENT_MAX_PRIORITIES (её значение в рассматриваемой версии равно 256), которая устанавливает предельное допустимое количество уровней приоритета.

Функцию event_base_priority_init() необходимо вызывать сразу после создания объекта event_base, то есть до активизации какого бы то ни было события, связанного с этим объектом.

1.6. Инициализация цикла обработки событий

После создания объекта event_base и связывания с ним нескольких событий (об этом немного позже) начинает работу основная функциональная часть программы - цикл

обработки событий, позволяющий следить за наступлением того или иного события и выдавать соответствующее оповещение. Для запуска этого цикла служит функция

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

По умолчанию эта функция работает с заданным набором событий base до тех пор, пока в этом наборе существуют зарегистрированные события. Внутри цикла отслеживается момент наступления любого такого события - это может быть, например, сигнал о готовности к чтению/записи определённого сокета или файла или сигнал о таймауте, означающий завершение заданного интервала времени ожидания. Это событие получает статус активного (active), и начинается его обработка, то есть реакция программы на наступившее событие.

Разработчик может изменить принятое по умолчанию действие функции event_base_loop() с помощью флагов, передаваемых через аргумент flags. При установке флага EVLOOP_ONCE (значение 0x01) основной цикл ожидает наступления любого зарегистрированного события или нескольких событий, обрабатывает их и завершает работу. Если установлен флаг EVLOOP_NONBLOCK (значение 0x02), то процесс ожидания, как таковой, отсутствует, то есть, в цикле сразу же проверяются состояния событий, и при обнаружении "готовности" события немедленно запускается соответствующая функция обратного вызова.

При успешном завершении функция event_base_loop() возвращает 0, а при возникновении ошибки во внутреннем механизме обработки событий возвращается -1.

Если необходимости в установке флагов нет, то более удобной в использовании является функция

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

Работает эта функция точно так же, как event_base_loop() по умолчанию.

1.7. Завершение цикла обработки событий

Чтобы завершить цикл обработки при непустом наборе зарегистрированных событий, необходимо воспользоваться одной из двух специально предусмотренных для подобных случаев функций.

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

Эта функция завершает цикл обработки событий по истечении заданного интервала времени, определяемого значением аргумента tv. Если аргумент tv содержит значение NULL, то выполняется попытка немедленного выхода из цикла. Но даже в этом случае завершение цикла может быть отложено до тех пор, пока не будут завершены все функции обратного вызова, обрабатывающие текущие активные события.

На первый взгляд функция

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

представляет собой аналог вызова функции event_base_loopexit(base, NULL), но это не так. Данная функция является более жёстким вариантом завершения цикла обработки: она предоставляет возможность корректно доработать только одной выполняющейся в текущий момент функции обратного вызова, после чего немедленно завершает цикл вне зависимости от того, имеются ли ещё не до конца обработанные активные события.

Кроме того, существует различие в действиях этих функций в том случае, когда цикл обработки событий не активизирован. Вызов event_base_loopexit(base, NULL) "запланирует" останов следующего цикла обработки сразу после одного раунда обратных вызовов для наступивших событий (в определённой степени это похоже на инициализацию цикла с флагом EVLOOP_ONCE), в то время как вызов event_base_loopbreak(base) не оказывает никакого воздействия на инициализированный позже цикл обработки событий.

Листинг 2. Принудительное завершение цикла обработки событий по истечении заданного интервала времени

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

void force_exit_loop_base( struct event_base *base )
{
  struct timeval seconds;
  seconds.tv_sec = 3;     /* секунды */
  seconds.tv_usec = 0;    /* микросекунды */

  for( int i=0; i < 5; i++ )
  {
    /* Планирование выхода из цикла через 3 секунды */
    event_base_loopexit( base, &seconds );
    event_base_dispatch( base );
    printf( "%d-й запуск цикла...\n", i+1 );
  }
}
```

2. Обработка событий

В библиотеке libevent главным объектом операций является событие (event). Каждое событие представляет некоторый набор условий, в который могут быть включены: дескриптор файла, готовый к чтению и/или записи; дескриптор файла, приходящий в состояние готовности к чтению/записи (только для механизма ввода/вывода edge-triggered); наступивший таймаут; сгенерированный сигнал; событие, сгенерированное пользователем.

После вызова функции создания события и связывания его с набором событий (event base) оно становится инициализированным (initialized). В этот момент для такого события можно выполнить операцию присоединения (add), после чего данное событие приобретает статус ожидающего (pending) в своём наборе. В период ожидания могут быть выполнены условия, при которых событие считается произошедшим (например, изменилось состояние дескриптора файла или исчерпан таймаут), то есть активным (active). Если при конфигурировании событие было определено как постоянное (persistent), то после

обработки оно возвращается в режим ожидания. Непостоянные события после того, как отработает соответствующая им функция обратного вызова, не возвращаются в режим ожидания. Отключить режим ожидания для события можно с помощью операции удаления (delete), а снова вернуть отключённое событие в состояние ожидания позволяет ранее упомянутая операция присоединения (add).

2.1. Создание объекта события

Для создания события (точнее, объекта или экземпляра события) служит функция

```
struct event *event_new( struct event_base *base, evutil_socket_t fd,
                         short what, event_callback_fn cb, void *arg );
```

которая пытается выделить память для создания нового события и связать его с заданным набором base. Положительное значение fd обозначает дескриптор файла, для которого будут отслеживаться события чтения и/или записи. Аргумент what представляет собой комбинацию следующих возможных флагов, определённых в заголовочном файле <event2/event.h>:

- EV_TIMEOUT 0x01 - наступление таймаута;
- EV_READ 0x02 - операция чтения;
- EV_WRITE 0x04 - операция записи;
- EV_SIGNAL 0x08 - генерация сигнала
- EV_PERSIST 0x10 - событие постоянное; не удаляется после активизации;
- EV_ET 0x20 - edge-triggered ввод/вывод, только если поддерживается внутренним механизмом обработки событий.

При активизации данного события будет вызвана указанная функция обратного вызова cb, которой передаются следующие аргументы: дескриптор файла fd, битовая комбинация, описывающая все сгенерированные события, и переданный извне аргумент arg. Для функции обратного вызова определён специальный тип:

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

При возникновении ошибки функция event_new() возвращает значение NULL.

Все новые события после инициализации не переводятся в состояние ожидания автоматически, для этого необходимо явно вызвать функцию

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

Для удаления объекта события вызывается функция

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

Эту функцию можно вызывать даже для удаления ожидающих и активных событий, она корректно выполнит все необходимые операции перед фактическим удалением объекта.

Листинг 3. Простой пример создания цикла обработки набора событий и функции обратного вызова

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

void call_back( evutil_socket_t fd, short ev_flag, void *arg )
{
  const char *data = arg;
  printf( "Сокет %d - активные события: %s%s%s%s; %s\n", (int)fd,
          (ev_flag & EV_TIMEOUT) ? " таймаут" : "",
          (ev_flag & EV_READ)    ? " чтение"  : "",
          (ev_flag & EV_WRITE)   ? " запись"  : "",
          (ev_flag & EV_SIGNAL)  ? " сигнал"  : "", data );
}

/* Предположим, что вызывающая функция создала два сокета
   и определила их, как неблокируемые */
void main_loop( evutil_socket_t fd1, evutil_socket_t fd2 )
{
  struct event *ev1, *ev2;
  struct timeval seconds = { 10, 0 };

  struct event_base *base = event_base_new();

  ev1 = event_new( base, fd1, (EV_TIMEOUT|EV_READ|EV_PERSIST), call_back,
                   (char *)"тип события: чтение" );
  ev2 = event_new( base, fd2, (EV_WRITE|EV_PERSIST), call_back,
                   (char *)"тип события: запись" );

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

3. Замечания о параметрах настройки

В libevent имеется возможность изменять некоторые глобальные параметры настройки, которые влияют на функционирование библиотеки в целом. При этом следует помнить, что изменения необходимо внести до того, как будет вызвана какая-либо рабочая функция из любого модуля libevent. Наиболее важными аспектами настройки являются обработка невосстановимых ошибок и управление памятью.

3.1. Обработка невосстановимых ошибок

При возникновении невосстановимой внутренней ошибки libevent по умолчанию вызывает exit() или abort() для экстренного завершения текущего процесса. Такие ошибки можно обработать более корректно с помощью функции обратного вызова специально определённого типа:

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

Такую функцию обработки невосстановимой ошибки event_fatal_cb er_callback() разработчик должен определить сам и затем передать её в настроечную функцию

```
void event_set_fatal_callback( event_fatal_cb er_callback );
```

Следует обратить особое внимание на то, что функция обработки невосстановимой ошибки не должна возвращать управление библиотеке libevent или вызывать какую-либо функцию libevent. После выполнения требуемой обработки ошибки в любом случае необходимо завершить выполнение программы.

3.2. Управление памятью

По умолчанию libevent пользуется функциями управления памятью из стандартной библиотеки C. При необходимости разработчик может написать собственные реализации функций malloc, realloc и free. Ввод в действие этих заменяющих функций осуществляется посредством специальной настроечной функции

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

Заключение

Библиотека libevent представляет собой мощный, но тем не менее достаточно простой в использовании инструмент для эффективной организации обработки событий. Хорошо продуманная структура библиотеки позволяет гибко настраивать её для конкретных приложений.

В данной статье рассмотрены основы использования libevent: основное внимание уделено обработке событий. В третьей статье речь идёт о механизмах 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.