R1：设置 Libevent

Libevent 中有一些在整个进程中共享的全局设置，这些设置会影响整个库。

在调用 Libevent 库的其他任何部分之前，您必须对这些设置进行一些处理，否则 Libevent 可能会处于不一致的状态。

Libevent 日志消息

Libevent 可以记录内部错误和警告。如果在编译时开启日志支持，也可以记录调试信息。默认情况下这些消息会被写到标准错误（`stderr`），您可以提供自己的日志函数以覆盖默认行为。

接口

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

/* 强烈反对，参考本小节最后的兼容性说明 */
#define _EVENT_LOG_DEBUG EVENT_LOG_DEBUG
#define _EVENT_LOG_MSG   EVENT_LOG_MSG
#define _EVENT_LOG_WARN  EVENT_LOG_WARN
#define _EVENT_LOG_ERR   EVENT_LOG_ERR

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

void event_set_log_callback(event_log_cb cb);
```

要覆盖 Libevent 的日志行为，只需编写符合 `event_log_cb` 签名的函数，将其作为参数传递给 `event_set_log_calback()`。Libevent 需要记录消息时，将会传递给您提供的函数。当需要恢复默认日志行为时，调用 `event_set_log_callback()` 并传递参数 `NULL` 即可。

示例

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

static void discard_cb(int severity, const char *msg)
{
    /* 这个自定义回调函数没有做任何事。 */
}
```

```c
static FILE *logfile = NULL;
static void write_to_file_cb(int severity, const char *msg)
{
    const char *s;
    if (!logfile)
        return;
    switch (severity) {
        case _EVENT_LOG_DEBUG: s = "debug"; break;
        case _EVENT_LOG_MSG:   s = "msg";   break;
        case _EVENT_LOG_WARN:  s = "warn";  break;
        case _EVENT_LOG_ERR:   s = "error"; break;
        default:               s = "?";     break; /* 不会命中 */
    }
    fprintf(logfile, "[%s] %s\n", s, msg);
}

/* 关闭 Libevent 的所有日志 */
void suppress_logging(void)
{
    event_set_log_callback(discard_cb);
}

/* 把 Libevent 所有日志消息重定向至 C stdio 文件 'f'。 */
void set_logfile(FILE *f)
{
    logfile = f;
    event_set_log_callback(write_to_file_cb);
}
```

说明

在用户提供的 `event_log_cb` 回调函数中调用 Libevent 函数是不安全的！例如，如果试图编写一个使用 `bufferevents` 将警告消息发送至网络套接字的回调函数，可能会遇见奇怪且难以调试的 bug。这个限制可能会在未来版本的某些函数中移除。

通常情况下，调试日志不会开启，也不会被发送到日志回调函数。如果 Libevent 被构建成支持这些功能的话，可以手动开启。

接口

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

void event_enable_debug_logging(ev_uint32_t which);
```

调试日志是冗长的，并且在大多数情况下都不是必要的。调用 `event_enable_debug_logging` 并传递参数 `EVENT_DBG_NONE` 可以得到默认行为；传递 `EVENT_DBG_ALL` 将开启所有调试日志。未来版本可能会提供更细粒度的选择。

以上函数声明在 `<event2/event.h>` 中， `event_enable_debug_logging()` 首次首次出现在 Libevent 2.1.1-alpha 中，其余函数首次出现在 Libevent 1.0c 中。

兼容性说明

在 Libevent 2.0.19-stable 之前， `EVENT_LOG_*` 宏定义均以下划线开头，即 `_EVENT_LOG_*` 。强烈不推荐使用旧的定义，这些定义应当只能用于对 Libevent 2.0.18-stable 或更早版本的向后兼容。这些定义可能会在将来的 Libevent 版本中移除。

处理致命错误

当 Libevent 检测到一个不可恢复的内部错误（例如损坏的数据结构），其默认行为是调用 `exit()` 或 `abort()` 以退出当前进程。这些错误通常意味着某处有 bug：要么在您的代码中，要么在 Libevent 中。

如果希望应用程序更优雅地处理致命错误，可以通过提供退出时需要调用的函数来重写 Libevent 的默认行为。

接口

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

为了使用这些功能，首先需要为 Libevent 定义一个在遭遇致命错误时应当调用的函数并传递给 `event_set_fatal_callback()` 。随后 Libevent 会在遭遇致命错误时调用您提供的函数。

您的函数不应当将控制权交给 Libevent，这样做可能会导致未知行为，并且 Libevent 可能会退出以避免崩溃。一旦您的函数被调用，就不应当调用任何 Libevent 函数。

这些函数声明在 `<event2/event.h>` 中，最早出现在 Libevent 2.0.3-alpha 中。

内存管理

默认情况下，Libevent 使用 C 库的内存管理函数从堆中申请内存。您也可以通过提供 `malloc` ， `realloc` 和 `free` 的替代函数，让 Libevent 使用其他的内存管理。如果您希望 Libevent 使用一个更有效的分配器，或者希望 Libevent 使用一个检测型分配器（instrumented allocator）来查找内存泄露，那么您可能需要这样做。

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));
```

下面是一个简单的示例，它可以计算已经分配的字节数。在实际应用中，您可能需要添加锁以避免多线程运行时发生错误。

示例

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

/* 这个联合体目的是保证大小与它包含的所有类型的最大值一样。 */
union alignment {
    size_t sz;
    void *ptr;
    double dbl;
};
/* 我们需要确保返回的所有内容都是对齐的，包括 double。 */
#define ALIGNMENT sizeof(union alignment)

/* 我们需要将指针转为 char* 来调整，直接使用 void* 是不准的。 */
#define OUTPTR(ptr) (((char*)ptr)+ALIGNMENT)
#define INPTR(ptr) (((char*)ptr)-ALIGNMENT)

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

```
    *(size_t*)ptr = sz;
    total_allocated = total_allocated - old_size + sz;
    return OUTPTR(ptr);
}
static void replacement_free(void *ptr)
{
    ptr = INPTR(ptr);
    total_allocated -= *(size_t*)ptr;
    free(ptr);
}
void start_counting_bytes(void)
{
    event_set_mem_functions(replacement_malloc,
                            replacement_realloc,
                            replacement_free);
}
```

说明

- 替换内存管理函数将会影响之后 Libevent 中所有需要分配、调整和释放内存的操作，因此您需要确保在调用任何其他 Libevent 函数之前进行替换。否则 Libevent 可能调用您定义的内存释放函数来释放通过 C 库分配的内存。
- 您的 `malloc` 和 `realloc` 函数返回的内存块需要具有和 C 库函数返回的内存块一样的内存对齐。
- 您的 `realloc` 函数需要正确处理 `realloc(NULL, sz)`（即当做 `malloc(sz)` 处理）。
- 您的 `realloc` 函数需要正确处理 `realloc(ptr, 0)`（即当做 `free(ptr)` 处理）。
- 您的 `free` 函数不需要处理 `free(NULL)`。
- 您的 `malloc` 函数不需要处理 `malloc(0)`。
- 如果在多个线程中使用 LIbevent，替换的内存管理函数需要是线程安全的。
- Libevent 会使用替换函数分配需要返回给您的内存。因此，如果您替换了 Libevent 的内存管理函数，且需要释放由 Libevent 函数返回的内存，那么您应当使用替换的 `free` 函数。

`event_set_mem_functions()` 声明在 `<event2/event.h>` 中，最早出现在 Libevent 2.0.1-alpha 中。

编译 Libevent 时可以禁用 `event_set_mem_functions()`。如果被禁用，则使用 `event_set_mem_functions()` 的程序将不会被编译或链接。在 Libevent 2.0.2-alpha 以及更新版本中，您可以通过检查 `EVENT_SET_MEM_FUNCTIONS_IMPLEMENTED` 宏是否定义来判断 `event_set_mem_functions()` 是否可用。

锁和线程

如果您正在编写多线程程序，您可能知道多个线程同时访问同一个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模式：

- 有些结构体本身就是单线程的：同时超过一个线程使用这些结构是不安全的。
- 有些结构体是可选锁定的：您可以告知 Libevent 是否需要在多个线程中使用对象。
- 有些结构体总是被锁定：如果 Libevent 是在支持锁的环境下运行的，那么它们总是可以安全地同时在多个线程中使用。

为获得 Libevent 中的锁，在调用分配需要在多个线程中共享的结构体的 Libevent 函数之前，您必须告知 Libevent 使用哪种锁。

如果使用 pthreads 库或者 Windows 内置多线程代码，那么您是幸运的：有一些预定义的函数可以将 Libevent 设置成使用正确的 pthreads 或 Windows 函数。

接口

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

这两个函数在成功时返回 0，失败时返回 -1。

如果使用其他线程库，则需要一些额外的工作。需要使用您的库编写函数以完成：

- 锁
- 锁定
- 解锁
- 分配锁
- 销毁锁
- 条件变量
- 创建条件变量
- 销毁条件变量

- 等待条件变量
- 触发/广播条件变量
- 线程
- 线程 ID 检测

随后通过 `evthread_set_lock_backs` 和 `evthread_set_id_callback` 接口告知 Libevent 使用这些函数。

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` 结构体描述了回调函数及其能力。在上述版本中，`lock_api_version` 必须设置成 `EVTHREAD_LOCK_API_VERSION`，`supported_locktypes` 必须设置成 `EVTHREAD_LOCKTYPE_*` 的比特掩码以描述支持的锁的类型。（在 2.0.4-alpha 中，`EVTHREAD_LOCKTYPE_RECURSIVE` 是必须的，而 `EVTHREAD_LOCKTYPE_READWRITE` 没有被使用。）`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` 函数必须尝试以指定模式请求加锁；`unlock` 函数必须尝试释放锁，成功返回 0，失败返回非 0。

可识别的锁类型有：

0

常规的，不必是递归的锁

EVTHREAD_LOCKTYPE_RECURSIVE

持有该锁的线程再次请求不会被阻塞，一旦该线程进行锁定次数的解锁操作（即加锁次数与解锁次数相同），其他线程就可以请求该锁。

EVTHREAD_LOCKTYPE_READWRITE

读时可允许多个线程同时拥有，写时只允许一个线程拥有。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：为读操作请求或释放锁。

EVTHREAD_WRITE

仅用于读写锁：为写操作请求或释放锁。

`id_fn` 参数必须返回一个无符号长整数以标识调用该函数的线程。对于相同的线程，该函数必须总是返回相同的值；而对于不同的线程，返回值必须不同。

`evthread_condition_callbacks` 结构体描述了与条件变量相关的回调函数。在上述版本中，`condition_api_version` 必须设置成 `EVTHREAD_CONDITION_API_VERSION`。`alloc_condition` 函数必须返回新的条件变量的指针，该函数接收 0 作为参数。`free_condition` 函数必须释放条件变量的存储空间和资源。`wait_condition` 函数接收三个参数：由 `alloc_condition` 分配的条件变量、由 `evthread_lock_callbacks.alloc` 分配的锁和可选的超时值。调用该函数时必须持有锁，该函数应当释放锁，并等待条件变量成为授信状态或超时（可选）。`wait_condition` 发生错误时返回 -1，条件变量授信时返回 0，超时返回 1。在返回之前，函数应当再次确认其持有锁。最后，`signal_condition` 函数唤醒一个（`broadcast` 为 `false`）或所有（`broadcast` 为 `true`）等待条件变量的线程。只有持有与条件变量相关的锁时，才能执行该操作。

关于条件变量的更多信息，请参考 pthreads 库的 `pthread_cond_*` 部分，或 Windows 的 `CONDITION_VARIABLE` 函数的相关文档。

示例

关于使用这些函数的示例，请参考 Libevent 源码中的 evthread_pthread.c 和 evthread_win32.c 文件。

这一节的函数声明在 `<event2/thread.h>` 中，大多数首次出现在 Libevent 2.0.4-alpha 中。Libevent 2.0.1-alpah 到 2.0.3-alpah 曾用更老的接口设置锁相关函数。使用 `event_use_pthreads()` 函数需要链接 event_pthreads 库。

条件变量相关函数是 Libevent 2.0.7-rc 引入的，用于解决某些棘手的死锁问题。

编译 Libevent 可以禁用对锁的支持，这时使用上述线程相关函数的程序将不能运行。

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（lock debugging）" 特性。该特性包装了锁调用，以便捕获常见的锁错误，包括：

- 释放一个未持有的的锁
- 重复申请一个非递归锁

以上任意错误发生，Libevent 给出断言失败并退出。

接口

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

> 注意：该函数必须在创建或使用任何锁之前调用。安全起见，请在设置线程函数后立即调用该函数。

该函数在 Libevent 2.0.4-alpha 中引入时错误拼写成 `evthread_enable_lock_debuging()`。在 Libevent 2.1.2-alpha 中被修正为 `evthread_enable_lock_debugging()`，目前两种拼写都支持。

调试事件的使用

在使用事件时，Libevent 可以检测并报告一些常见的错误，包括：

- 将未初始化的结构体事件当成已初始化的。
- 尝试重新初始化一个处于未决状态的结构体事件。

跟踪需要被初始化的事件会让 Libevent 消耗额外的内存和 CPU 资源，因此调试模式只应当在调试程序时开启。

接口

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

该函数必须在创建任何 event_base 之前调用。

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。这是因为没有途径可以告知 Libevent 由 `event_assign()` 创建的事件何时将不再使用。（而由 `event_new()` 创建的事件可以通过 `event_free()` 告知。）如果想在调试时避免内存耗尽，可以显式地告知 Libevent 哪些事件不再被当做是已分配的：

接口

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

没有启动调试模式时调用 `event_debug_unassign()` 不会有任何效果。

示例

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

#include <stdlib.h>

void cb(evutil_socket_t fd, short what, void *ptr)
{
    /* 对于堆上事件，传递参数为 'NULL'，
     * 对于栈上事件，传递参数为指向其本身
     * 的指针。 */
    struct event *ev = ptr;

    if (ev)
        event_debug_unassign(ev);
}

/* 这是一个等待 fd1 和 fd2 直到均为可读的主循环。 */
void mainloop(evutil_socket_t fd1, evutil_socket_t fd2, int debug_mode)
{
    struct event_base *base;
    struct event event_on_stack, *event_on_heap;
```

```
    if (debug_mode)
       event_enable_debug_mode();

    base = event_base_new();

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

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

    event_base_dispatch(base);

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

详细的事件调试功能是一个只能在编译时使用 `CFLAGS` 环境变量 "-DUSE_DEBUG" 启用的特性。启用此标志后，使用 Libevent 的任何程序都会输出详细描述后端低级活动的及其冗长的日志。这些日志包括但不限于：

- 添加事件
- 删除事件
- 特定于平台的事件提醒信息

该特性无法通过 API 开启或关闭，因此只能在开发环境中使用。

以上调试函数在 Libevent 2.0.4-alpha 中初次出现。

检测 Libevent 版本

新版本 Libevent 会添加新特性并移除 bug。某些情况下您可能希望检测 Libevent 版本以便：

- 检测安装的 Libevent 版本是否能用于构建您的程序。
- 出于调试目的显示 Libevent 版本。
- 检测 Libevent 版本以便向用户警告 bug，或者绕开这些 bug。

接口

```
#define LIBEVENT_VERSION_NUMBER 0x02000300
#define LIBEVENT_VERSION "2.0.3-alpha"
```

```
const char *event_get_version(void);
ev_uint32_t event_get_version_number(void);
```

这些宏返回 Libevent 编译时的版本；函数则返回运行时版本。注意如果您的程序是动态链接 Libevent 的，则这些版本可能不一样。

可以获得两种格式的 Libevent 版本号：一种适合显示给用户的字符串格式，以及一种适合数值比较的 4 字节整数格式。整数格式使用最高位字节表示大版本，第二个字节表示小版本，第三个字节表示补丁版本，最低字节用于表示发行状态（0 为发行版，非 0 表示某个特定版本的后序开发序列）。

由此可知，Libevent 2.0.1-alpha 的版本数为 [02 00 01 00]，或者 0x02000100。2.0.1-alpha 和 2.0.2-alpha 之间的某个开发版本的版本数可能为 [02 00 01 08]，或者 0x02000108。

示例：编译时检测

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

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

int
make_sandwich(void)
{
        /* 假设 Libevent 6.0.5 中引入了 evutil_make_me_a_sandwich 函数. */
#if LIBEVENT_VERSION_NUMBER >= 0x06000500
        evutil_make_me_a_sandwich();
        return 0;
#else
        return -1;
#endif
}
```

示例：运行时检测

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

int
check_for_old_version(void)
{
```

```
    const char *v = event_get_version();
    /* 这是一种愚蠢但在 Libevent 2.0 之前唯一可行的方法。 */
    if (!strncmp(v, "0.", 2) ||
        !strncmp(v, "1.1", 3) ||
        !strncmp(v, "1.2", 3) ||
        !strncmp(v, "1.3", 3)) {

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

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

这一小节的宏和函数声明在 `<event2/event.h>` 中，`event_get_version()` 函数首次出现在 Libevent 1.0c 中，其他首次出现在 Libevent 2.0.1-alpha 中。

释放 Libevent 的全局结构体

即使已经释放了所有通过 Libevent 分配的对象，仍然会存在一些全局分配的结构体。这通常不是什么问题：一旦进程退出，这些结构体将会被销毁。但是保留这些结构体可能会使一些调试工具误认为 Libevent 出现了资源泄露。如果需要确保 Libevent 释放所有内部全局数据结构，可以调用：

接口

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

该函数不会释放任何通过 Libevent 函数返回的结构体。如果希望退出之前释放所有资源，需要手动释放 `event`、`event_base`、`bufferevent` 等资源。

调用 `libevent_global_shutdown()` 会让其他 Libevent 函数的行为无法预测，除非将其作为程序最后调用的 Libevent 函数，否则不要调用它。唯一一个例外是：`libevent_global_shutdown()` 是幂等的，调用该函数后可以再次调用。

这个函数声明在 `<event2/event.h>` 中，首次在 Libevent 2.1.1-alpha 中引入。