R4：处理事件

Libevent 的基本处理单元是事件。每个事件代表一个条件集合，包括：

- 文件描述符就绪，可读或可写。
- 文件描述符变成就绪状态，可读或可写（仅限于边缘触发 IO）。
- 即将超时。
- 信号发生。
- 用户触发事件。

事件拥有相似的生命周期。一旦调用 Libevent 函数创建一个事件并关联到 `event_base`，事件进入**已初始化**状态。此时可以添加事件，使其在 `event_base` 中成为**未决**状态（事件需要先关联再添加）。当事件处于未决状态，如果可以触发事件的条件满足（例如，文件描述符改变状态或超时），该事件变为**活跃**状态。如果事件配置为**持久**（persistent），则转成未决状态；如果不是持久的，则回调时不会变成未决状态。可以通过删除将未决事件转变为非未决事件，也可以通过添加将非未决事件转变为未决。

构造事件对象

使用 `event_new()` 接口创建对象。

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()` 尝试申请并构造一个用于 `base` 的新事件。`what` 参数是上述列表中标志的集合。（标志的语义将会在下方描述。）如果 `fd` 非负，我们将观察该文件的读或写。当事件被激活，Libevent 将会回调提供的 `cb` 函数，并传递参数：文件描述符 `fd`，表示所有被触发事件（这里的事件是使事件对象活跃的行为或信号）的比特字段，以及在构造函数时传递的 `arg`。

发生内部错误或参数不正确时，`event_new()` 会返回 `NULL`。

新事件是的状态是已初始化而非未决。调用 `event_add()` 可使其转变为未决状态（相关文档在下方）。

调用 `event_free()` 可以释放事件。对未决或活跃事件调用该函数是安全的：该函数会在释放前将事件状态转变为非未决和非活跃。

示例

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

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

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

        /* 调用者已经通过某种方式设置了 fd1 和 fd2，并且是非阻塞的。 */

        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);
}
```

上述函数声明在 `<event2/event.h>` 中，首次出现在 Libevent 2.0.1-alpha 中，`event_callback_fn` 在 Libevent 2.0.4-alpha 中首次作为类型定义出现。

事件标志

EV_TIMEOUT

此标志表明超时时间流逝后事件会变成活跃状态。

构造事件时 `EV_TIMEOUT` 标志会被忽略：您可以在添加时选择是否设置超时时间。超时时间到达时此标志会设置在回调函数的 `what` 参数中。

EV_READ

此标志表明事件会在给定的文件描述符可读时变成活跃状态。

EV_WRITE

此标志表明事件会在给定的文件描述符可写时变成活跃状态。

EV_SIGNAL

用于检测信号，参考下方 "构建信号事件"。

EV_PERSIST

表明事件是持久的，参考下方 "关于事件持久性"。

EV_ET

表明如果 `event_base` 的后端支持边缘触发，则事件应当是边缘触发的。这个标志会影响 `EV_READ` 和 `EV_WRITE` 的语义。

从 Libevent 2.0.1-alpha 开始，同事时刻多个事件可以因相同的条件处于未决状态。例如，可能有两个事件因为给定的文件描述符变成可读时活跃，他们的回调顺序是不确定的。

这些标志定义在 `<event2/event.h>` 中，`EV_ET` 在 Libevent 2.0.1-alpha 被引入，其余均从 Libevent 1.0 开始就存在。

> 译者注：边缘触发只在文件状态改变时触发，即文件从不可读变为可读。若程序只读了一部分，文件仍然是可读的。但是由于可读与可读之间并没有发生状态转变，因此不会再次触发。与之对应的是水平触发，若文件未读完，则会一直触发。

关于事件持久性

默认情况下，任何未决事件活跃后（由于文件描述符可写或可读，或到达超时时间），都会在回调之前变成非未决状态。因此，如果希望让事件重新进入未决状态，您可以在回调函数内部再次调用 `event_add()`。

如果事件设置了 `EV_PERSIST` 标志，则该事件是持久的。这意味着回调时该事件会保持未决状态。如果需将在回调内部将事件设置成非未决，您可以调用 `event_del()` 函数。

每次回调都会重置超时。因此，如果您的事件被设置成 `EV_READ|EV_PERSIST` 且超时时间为 5s，则该事件会在这些情况下活跃：

- 每次套接字可写。
- 上次活跃后 5s。

创建回调参数为自身的事件

您可能经常遇到事件的回调函数需要事件本身作为参数的情况。不能只是简单地给 `event_new()` 传递指针，因为此时事件还不存在（`event_new()` 内部才会创建事件，调用该函数时指针还没有指向任何事件）。可以通过调用 `event_self_cbarg()` 解决这个问题。

接口

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

`event_self_cbarg()` 会返回一个 "魔术" 指针，作为参数传递给回调函数时，会通知 `event_new()` 创建一个以自身作为回调参数的事件。

示例

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

static int n_calls = 0;

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

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

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

void run(struct event_base *base)
{
    struct timeval one_sec = { 1, 0 };
    struct event *ev;
    /* 我们要创建一个重复 100 次的定时器。 */
    ev = event_new(base, -1, EV_PERSIST, cb_func, event_self_cbarg());
```

```
    event_add(ev, &one_sec);
    event_base_dispatch(base);
}
```

这个函数可用于 `event_new()`，`evtimer_new()`，`evsignal_new()`，`event_assign()`，`evtimer_assign()` 和 `evsignal_assign()` 函数，但不能用作非事件的回调参数。

纯超时事件（定时器）

为方便使用，有一些以 `evtimer_` 开头的宏可以替代 `event_*` 函数来申请和操作纯超时事件。这些宏除了能简化代码没有任何其他作用。

接口

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

`evtimer_new()` 首次出现在 Libevent 2.0.1-alpha 中，其他宏自 Libevent 0.6 就存在。

构建信号事件

Libevent 也可以监测 POSIX 式的信号。要构建一个信号处理器，使用：

接口

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

除提供信号编号代替文件描述符外，其他参数与 `event_new` 一致。

示例

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

```
/* HUP 信号（终端挂起时触发）触发时调用 sighup_function */
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调用的，因此回调函数可以安全地调用通常不能在 POSIX 信号处理器中使用的函数。

> 警告：不要为信号事件设置超时，这可能是不受支持的。[待修正：真的是这样吗？]

Libevent 也提供了一些方便使用的宏来处理信号事件。

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_*` 宏自 Libevent 2.0.1-alpha 出现，更早的版本中这些宏叫做 `signal_add()`、`signal_del()` 等.....

关于信号的警告

当前版本 Libevent 的大部分后端中，每个进程中只有一个 `event_base` 可以监听信号。如果向两个 `event_base` 添加信号，即使是不同信号，也只有一个 `event_base` 可以接收信号。

`kqueue` 后端没有这个限制。

创建用户触发事件

有时，创建可以在所有高优先级事件完成后激活并执行的事件是很有用的。任何形式的清理以及垃圾回收都是此类事件。有关设置较低优先级的解释，请参考 "具有优先级的事件"。

创建用户触发事件：

```
struct event *user = event_new(evbase, -1, 0, user_cb, myhandle);
```

注意不需要调用 `event_add()`，只需使用下面的方式激活：

```
event_active(user, 0, 0);
```

对于非信号事件来说第三个参数没有意义（目前已废弃）。

创建不在堆上分配的事件

出于性能和其他原因，一些人喜欢将事件分配为更大结构体的一部分。每次使用事件时，这将节省：

- 内存分配器在堆上分配小对象的开销。
- 解引用（取值）指向结构体的指针的时间开销。
- 如果事件不在缓存中，则可能产生额外的缓存未命中的时间开销。

由于不同版本 Libevent 的事件结构体大小可能有出入，这种方法有打破与其他 Libevent 版本之间二进制兼容性的风险。

以上都是极小的开销，对大多数程序来说都无关紧要。除非您知道对事件进行堆分配会导致严重的性能损失，否则应当坚持使用 `event_new()`。如果未来版本的 Libevent 采用了比目前您使用的更大的事件结构体，`event_assign()` 会导致难以诊断的错误。

接口

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

除 `event` 参数必须指向一个未初始化的事件外，`event_assign()` 参数与 `event_new()` 参数一致。该函数成功返回 0，失败或参数无效返回 -1。

示例

```
#include <event2/event.h>
/* 当心，包含 event_struct.h 头文件意味着您的代码与未来版本的 Libevent
   失去二进制兼容新。 */
#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;
}
```

您也可以用 `event_assign()` 来初始化栈分配或静态分配事件。

警告

绝对不要对 `event_base` 中的未决事件调用 `event_assign()`，这样做会导致极难定位的错误。如果事件已经初始化并处于未决状态，再次调用 `event_assign()` 之前请先调用 `event_del()`。

有一些宏可以方便使用 `event_assign()` 分配纯超时事件和信号事件：

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)
```

如果既想使用 `event_assign()` 也想与未来版本 Libevent 保持二进制兼容，您可以在运行时询问 Libevent 结构体事件应该是多大：

接口

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

这个函数以字节为单位返回结构体时间的大小。与之前一样，由于这会使得代码既难读又难写，只有当您直到堆分配实际上是程序的一个重要问题时，才使用这个函数。

注意未来 `event_get_struct_event_size()` 可能会返回一个比 `sizeof(struct event)` 更小的值。如果发生这种情况，这意味着 `struct event` 末尾的额外字节只是为 Libevent 未来版本保留的填充字节。

下面的例子与上面相同，只是从依赖 `event_struct.h` 中的 `struct event` 大小转为运行时通过 `event_get_struct_size()` 获取正确的大小。

示例

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

/* 当我们在内存分配 event_pair 时，实际上会在结构体结尾分配更多空间。
   我们定义了一些宏来降低访问这些事件的出错率。 */
struct event_pair {
        evutil_socket_t fd;
};

/* 宏：从 'p' 开始偏移 'offset' 比特作为结构体事件 */
#define EVENT_AT_OFFSET(p, offset) \
           ((struct event*) ( ((char*)(p)) + (offset) ))
/* 宏：event_pair 的读事件 */
#define READEV_PTR(pair) \
           EVENT_AT_OFFSET((pair), sizeof(struct event_pair))
/* 宏：event_pair 的写事件 */
#define WRITEEV_PTR(pair) \
           EVENT_AT_OFFSET((pair), \
               sizeof(struct event_pair)+event_get_struct_event_size())

/* 宏：event_pair 的实际大小 */
#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()` 声明在 `<event2/event.h>` 中，自 Libevent 2.0.1-alpha 开始出现。该函数自 2.0.3-alpha 起返回整数，在此之前没有返回值。`event_get_struct_event_size()` 自 Libevent 2.0.4-alpha 引入。事件结构体本身定义在 `<event2/event_struct.h>` 中。

使事件未决和非未决

创建事件后，通过添加使其变为未决状态前，事件不会有任何行为。添加事件可以使用 `event_add`：

接口

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

对非未决事件调用 `event_add` 会使其在配置的 `event_base` 中转为未决，该函数成功返回 0，失败返回 -1。如果 `tv` 是 `NULL`，则不会设置超时时间，否则 `tv` 就是以秒和微秒为单位的超时时间。

如果对已经处于未决状态的事件调用 `event_add()`，则会继续保持在未决状态，并且会在给定的超时时间后重新调度。如果超时时间为 `NULL`，则 `event_add()` 没有任何效果。

> 注意：不要将 `tv` 设置成希望超时的时刻（这个参数表示一段时间后触发超时，而不是指定时刻触发超时）。如果在 2021 年 1 月 1 日执行 `tv->tv_sec = time(NULL) + 10;`，程序会等待 40 年（计算机的时间起点是 1970 年 1 月 1 日，因此 `time(NULL)` 作为时间跨度是 40 年），而不是 10 秒。

接口

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

对初始化过的事件调用 `event_del` 会使其非未决和非活跃。如果事件并不处在未决或活跃状态，则没有任何效果。该函数成功返回 0，失败返回 -1。

> 注意：如果在事件活跃但还没有机会执行回调时删除事件，则回调不会被执行。

接口

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

终于，您可以在不删除未决事件的 IO 或信号组件的前提下移除未决时间的定时器了。如果事件没有超时组件，`event_remove_timer()` 没有任何作用。如果是一个纯超时事件，则 `event_remove_timer()` 与 `event_del()` 效果相同。该函数成功返回 0，失败返回 -1。

这些函数声明在 `<event2/event.h>` 中。`event_add()` 和 `event_del()` 自 Libeven 0.1 其存在，`event_remove_timer()` 是 2.1.2-alpha 新增的。

具有优先级的事件

当多个事件同时触发，Libevent 没有定义任何关于何时回调的顺序。通过使用优先级，您可以将一些事件定义为比其他事件更重要。

正如在先前章节讨论的，每个 `event_base` 有与之相关的一个或多个优先级。在初始化事件后，添加到 `event_base` 前，可以设置事件的优先级。

接口

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

优先级的取值范围是 0 到优先级数量减 1。该函数成功时返回 0，失败时返回 -1。

当多个不同优先级的事件活跃时，低优先级事件不会执行。Libevent 会执行高优先级事件，然后重新检查事件。只有不再有高优先级事件活跃时，低优先级事件才会执行。

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);
  /* 现在 base 可以容纳 0 和 1 两种优先级 */
  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);

  /* 现在，不管 fd 什么时候可写，写回调都会先于读回调执行。
     直到写回调不在活跃才会执行读回调。 */
}
```

如果没有设置事件的优先级，默认是 `event_base` 支持的优先级数量除以 2。

该函数声明在 `<event2/event.h>` 中，自 Libevent 1.0 便存在。

检查事件状态

有时可能需要知道事件是否被添加，检查事件代表什么。

接口

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

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

void event_get_assignment(const struct event *event,
        struct event_base **base_out,
        evutil_socket_t *fd_out,
        short *events_out,
        event_callback_fn *callback_out,
        void **arg_out);
```

`event_pending` 函数检查给定的事件是否为未决或活跃状态。如果是，且在 `what` 参数中设置了 `EV_READ`、`EV_WRITE`、`EV_SIGNAL` 和 `EV_TIMEOUT` 中的任何一个标志，该函数都会返回事件正处于未决或活跃状态的所有（设置了的）标志。如果提供了 `tv_out` 并且设置了 `EV_TIMEOUT`，而且事件正因超时而未决或活跃，则 `tv_out` 会被设置成超时触发时间（注意并还有多久超时，而是超时的时间点）。

获取正在运行的事件

您可以获取指向当前正在运行的事件的指针，用于调试或其他目的。

接口

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

注意该函数只在提供的 `event_base` 循环中调用时其行为才是有意义的，在其他线程中调用并不被支持，并且可能导致未定义的行为。

该函数声明在 `<event2/event.h>` 中，在 Libevent 2.1.1-alpha 中引入。

配置单次触发事件

如果不需要多次添加事件，或者添加后立即删除，并且事件是非持久的，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 *);
```

该函数接口与 `event_new()` 一致，除了不支持 `EV_SIGNAL` 或 `EV_PERSIST`。事件会以默认优先级添加进 `event_base` 并执行。回调结束后，Libevent 会释放事件结构体。该函数成功返回 0，失败返回 -1。

使用 `event_base_once` 添加的事件无法被删除或手动触发：如果希望能够取消事件，请通过常规的 `event_new()` 或 `event_assign()` 接口创建事件。

值得一提的是直到 Libevent 2.0，如果事件从来没有触发，事件对应的内存不会被释放。从 Libevent 2.1.2-alpha 开始，这些事件会随着 `event_base` 的释放一起释放，即使从来没有触发。但是还需要小心：如果有一些数据与回调函数的参数相关联，那么这些数据依然不会释放，除非您的程序通过某些手段监控并释放了这些数据。

手动激活事件

极少数情况下，即使条件没有触发，您也希望激活某个事件。

接口

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

这个函数会使用 `what` 标志位（`EV_READ`，`EV_WRITE` 和 `EV_TIMEOUT` 的组合）激活事件 `ev`。该事件不需要事先处于未决状态，激活事件也不会使其未决。

警告：对同一个事件递归调用 `event_active()` 可能会导致资源耗尽。下面代码是错误使用 `event_active` 的示例。

示例：`event_active()` 死循环

```
struct event *ev;

static void cb(int sock, short which, void *arg) {
        /* 噢：对同一个事件无条件地调用其回调函数意味着其他事件都不能执行！ */
```

```
        event_active(ev, EV_WRITE, 0);
}

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

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

        event_add(ev, NULL);

        event_active(ev, EV_WRITE, 0);

        event_base_loop(base, 0);

        return 0;
}
```

这会导致事件循环只执行一次，并且永远调用函数 `cb`。

示例：使用定时器解决上述问题

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

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

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

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

   ev = evtimer_new(base, cb, NULL);

   evtimer_add(ev, &tv);

   event_base_loop(base, 0);
```

```
    return 0;
}
```

示例：使用 `event_config_set_max_dispatch_interval()` 解决上述问题

```
struct event *ev;

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

int main(int argc, char **argv) {
        struct event_config *cfg = event_config_new();
        /* 检查其他事件之前最多执行 16 次回调。 */
        event_config_set_max_dispatch_interval(cfg, NULL, 16, 0);
        struct event_base *base = event_base_new_with_config(cfg);
        ev = event_new(base, -1, EV_PERSIST | EV_READ, cb, NULL);

        event_add(ev, NULL);

        event_active(ev, EV_WRITE, 0);

        event_base_loop(base, 0);

        return 0;
}
```

该函数声明在 `<event2/event.h>` 中，自 Libevent 0.3 便存在。

公共超时优化

当前版本的 Libevent 使用二叉堆跟踪未决事件的超时。二叉堆能以 $O(\log n)$ 的性能完成超时时间的添加和删除。如果添加的超时值是随机分布的，那么这是最好的选择。但是如果需要添加很多相同的超时值则并不是最优选择。

例如：假设有 10,000 个事件，每一个事件都在添加后 5s 超时。在这种情况下，双端队列的性能可以达到 $O(1)$。

自然地，您不会希望对所有的超时值都使用队列，因为队列只在超时值是常数情况下更快。如果一些超时值是或多或少随机分布的，则向队列添加这些超时会消耗 $O(n)$ 的事件，这必二叉堆要糟糕得多。

Libevent 允许您将一些超时放进队列，而另一些超时放进二叉堆中以解决这个问题。您可以申请一个 "公共超时（common timeout）" `timeval`，然后使用该值添加事件。如果很多事件都有一个公共的超时，这种优化可以提高超时的性能。

接口

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

这个函数接收 `event_base` 和公共超时时长作为参数，返回指向一个特殊的时间结构体，该结构体指明事件应当以 $O(1)$ 添加进队列，而不是以 $O(\log(n))$ 添加进堆。在您的代码中可以对这个特殊的 `timeval` 自由地拷贝或赋值，但只会对构造时使用的 `event_base` 起作用。不要依赖于其实际内容：Libevent 仅仅通过这些内容来告知自己使用哪个队列。

示例

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

/* 我们将要在给定的 base 中创建很多事件，绝大部分的超时时间是 10s。
   如果调用了 initialize_timeout，会通知 Libevent 向 O(1) 队列
   中添加一个 10s 的超时。 */
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)
{
    /* 注意 ev 对应的 base 必须与 initialize_timeout 中的 base 一致 */
    if (tv && tv->tv_sec == 10 && tv->tv_usec == 0)
        return event_add(ev, &ten_seconds);
    else
        return event_add(ev, tv);
}
```

和所有优化函数一样，除非您非常确认这会影响性能，否则应当避免使用 `common_timeout` 函数。

这个函数在 Libevent 2.0.4-alpha 中被引入。

从已清理的内存中识别事件

Libevent 提供了一些能够从通过置 0 清理的内存（例如使用 `calloc()` 重新分配或使用 `memset()` 或 `bzero()` 清理）中识别已经初始化事件的函数。

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

这些函数并不能可靠地区分已初始化的事件和一块未初始化的内存。除非您知道这片内存区域已经被清除或初始化成事件，否则不要使用它们。

通常，除非您在编写一个非常特殊的程序，否则应该不需要用到这些函数。通过 `event_new()` 返回的事件总是已初始化的。

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);
}
```

译者注：从示例代码中可以看出，`event_initialized(ev)` 接收一个事件指针，不管指向的内存是什么，都把这块内存当做一个事件，然后返回该事件是否已初始化，仅此而已。从源码来看，`event_initialized(ev)` 仅仅检查是否满足 `ev->flags & EVLIST_INIT` 为 `true`，满足返回 1，不满足返回 0。

`event_initialized()` 函数自 Libevent 0.3 被引入。

废弃的事件处理函数

Libevent 2.0 之前没有 `event_assign()` 或 `event_new()` 函数，取而代之的是将事件与 "当前" `event_base` 关联的函数 `event_set()`。如果有多个 `event_base`，需要记住在之后调用 `event_base_set()`，以确保事件与实际想要使用的 `event_base` 关联。

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);
```

`event_set()` 函数除使用 "当前" `event_base` 外，其他与 `event_assign()` 类似。`event_base_set()` 函数更改事件所关联的 `event_base`。

对于处理定时器和信号事件，`event_set()` 有更方便的变种：`evtimer_set()` 大致对应 `evtimer_assign()`，`evsignal_set()` 大致对应 `evsignal_assign()`。

Libevent 2.0 之前，`event_set()` 的信号事件变种版本使用 `signal_` 而不是 `evsignal_` 作为前缀（也就是说，有 `signal_set()`，`signal_add()`，`signal_del()`，`signal_pending()` 和 `signal_initialized()`）。非常古老的版本（Libevent 0.6 之前）使用 `timeout_` 而不是 `evtimer_`。因此，如果您在考古代码，可能会见到 `timeout_add()`，`timeout_del()`，`timeout_initialized()`，`timeout_set()`，`timeout_pending()` 等函数。

旧版本的 Libevent（2.0 之前）使用 `EVENT_FD()` 和 `EVENT_SIGNAL()` 这两个宏来替代 `event_get_fd()` 和 `event_get_signal()` 函数。这些宏直接检查事件结构体的内容，因此会妨碍不同版本之间的二进制兼容性。在 2.0 以及之后的版本中，这两个宏仅仅是 `event_get_fd()` 和 `event_get_signal()` 的别名。

由于 Libevent 2.0 之前不支持锁，因此在运行 `event_base` 之外的线程调用任何改变事件状态的函数都是不安全的，包括 `event_add()`，`event_del()`，`event_active()` 和 `event_base_once()`。

还有一个 `event_once()` 函数与 `event_base_once()` 类似，不过仅作用于 "当前" `event_base`。

在 Libevent 2.0 之前，`EV_PERSIST` 标志不能与超时进行合理的互操作，`EV_PERSIST` 标志不会对超时做任何处理，而不是在事件活跃时重置超时。（目前设置 `EV_PERSIST` 会在每次活跃时重置超时，详情参考关于事件持久性）。

2.0 版本之前的 Libevent 不支持同时添加多个具有相同的文件描述符和 `READ/WRITE` 属性的事件。换句话说，同时只能有一个事件在等待读某个文件描述符；同时只能有一个事件在等待写某个文件描述符。