R3：使用事件循环

启动循环

一旦 `event_base` 注册了一些事件（如何创建并注册事件请参考下一节），您将希望 Libevent 等待事件并在触发时通知您。

接口

```
#define EVLOOP_ONCE             0x01
#define EVLOOP_NONBLOCK         0x02
// EVLOOP_NO_EXIT_ON_EMPTY 从 Libevent 2.1 开始存在
#define EVLOOP_NO_EXIT_ON_EMPTY 0x04

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

默认情况下，`event_base_loop()` 函数会一直运行直到内部没有注册事件。运行循环时，它会反复检查是否触发了任何已注册的事件（例如，读事件的文件描述符是可读的，或者到达超时事件的超时时间）。一旦有事件触发，该函数将所有触发事件标记为 "活跃的（active）" 并开始执行这些事件。

您可以通过 `flags` 参数设置一个或多个标志来改变 `event_base_loop()` 的行为。如果设置了 `EVLOOP_ONCE`，循环将会等待一些事件变成活跃状态，然后执行活跃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`，循环不会等待事件触发：它只会立刻检查是否有事件可以触发，如果有，则执行回调。

通常，循环会在没有任何未决或活跃事件时立即退出，可以通过传递 `EVLOOP_NO_EXIT_ON_EMPTY` 标志来覆盖这种行为——例如您需要通过其他线程添加事件。如果设置了 `EVLOOP_NO_EXIT_ON_EMPTY`，会一直循环直到调用 `event_base_loopbreak()`，`event_base_loopexit()` 或发生错误。

循环结束后，如果正常退出，则返回 0，若因后端出现了未处理的错误而退出则返回 -1，若因没有任何未决或活跃事件而退出则返回 1。

> 译者注：根据源码可知，返回 1 并非 `event_base` 中的事件全部处理完毕，而是没有事件可以处理，即 `event_base` 中没有注册任何事件。

为帮助理解，下面给出 `event_base_loop()` 算法的大致摘要：

伪代码

```
while (any events are registered with the loop,
        or EVLOOP_NO_EXIT_ON_EMPTY was set) {

    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; /* 不要运行任何优先级更低的事件 */
       }
    }

    if (EVLOOP_ONCE was set or EVLOOP_NONBLOCK was set)
       break;
}
```

方便起见，也可以调用：

接口

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

`event_base_dispatch()` 的调用与 `event_base_loop()` 一样，只是无需设置标志。因此，该函数会一直运行到没有注册事件或 `event_base_loopbreak()` 或 `event_base_loopexit()` 被调用。

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

终止循环

如果想在所有事件移除之前终止循环，您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_loopbreak(struct event_base *base);
```

`event_base_loopexit()` 函数通知 `event_base` 在给定的时间流逝后停止循环。如果 `tv` 参数是 `NULL`，则立即终止。如果当前 `event_base` 正在执行活跃事件回调，则会继续执行，直到执行完所有回调后才退出。

`event_base_loopbreak()` 函数通知 `event_base` 立即退出循环。与 `event_base_loopexit(base, NULL)` 不同的是，如果 `event_base` 正在执行活跃事件的回调，则会在完成该事件后立即退出。

同样需要注意 `event_base_loopexit(base, NULL)` 和 `event_base_loopbreak(base)` 在事件循环没有运行时的行为的不同：`loopexit` 安排下一次事件循环在下一轮回调完成后立即停止（就像设置了 `EVLOOP_ONCE` 一样），而 `loopbreak` 仅仅终止当前运行的循环，循环不在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：立即停止

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

/* 这是一个调用 loopbreak 的回调函数 */
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;

    /* 创建一个事件监听看门狗套接字是否可读，事件触发时回调 cb 函数，
       该函数会不管其他活跃事件而直接退出循环。
     */
    watchdog_event = event_new(base, watchdog_fd, EV_READ, cb, base);

    event_add(watchdog_event, NULL);

    event_base_dispatch(base);
}
```

示例：运行循环 10s 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;

  /* 现在我们以一系列的 10s 间隔运行 event_base，每次运行后输出
     "Tick"。关于一个更好地 10s 定时器的实现方法，请参考下面持久
     定时器的部分。 */
  while (1) {
     /* 10s 之后退出。 */
     event_base_loopexit(base, &ten_sec);

     event_base_dispatch(base);
     puts("Tick");
  }
}
```

有时可能需要判断循环是正常退出的，还是因为 `event_base_loopexit()` 或 `event_base_loopbreak()` 退出的，可以调用以下函数来判断 `loopexit` 或 `loopbreak` 是否被调用：

接口

```
int event_base_got_exit(struct event_base *base);
int event_base_got_break(struct event_base *base);
```

如果循环是因为调用 `event_base_loopexit()` 或 `event_base_loopbreak()` 退出的，这两个函数会分别返回真，否则返回假。下次启动事件循环会重置。

这些函数声明在 `<event2/event.h>` 中，`event_base_loopexit()` 首次出现在 Libevent 1.0c 中，`event_base_loopbreak()` 首次出现在 Libevent 1.4.3 中。

重新检查事件

通常，Libevent 会检查所有事件，随后运行所有最高优先级活跃事件，然后重新检查事件，依次循环。然而有时候可能需要 Libevent 在执行完当前回调后立即停止（调用其他相同优先级活跃事件的回调函数），并通知 Libevent 重新扫描。与 `event_base_loopbreak()` 类似，您可以通过 `event_base_loopcontinue()` 来完成这件事。

接口

```
int event_base_loopcontinue(struct event_base *);
```

如果当前没有正在执行的事件回调，则调用此函数不会产生任何影响。

这个函数在 Libevent 2.1.2-alpha 中引入。

检查内部时间缓存

有时可能希望在事件回调内部获取当前的近似时间，并且不希望调用 `gettimeofday()`（也许是因为您的操作系统将 `gettimeofday()` 实现成系统调用，而您试图避免系统调用的开销）。

在回调函数内部，您可以询问 Libevent 开始执行本轮回调时的时间：

接口

```
int event_base_gettimeofday_cached(struct event_base *base,
    struct timeval *tv_out);
```

如果 `event_base` 当前正在回调，`event_base_gettimeofday_cached()` 函数会将 `tv_out` 参数设置成缓存时间，否则会调用 `evutil_gettimeofday()` 获取当前的实际时间。该函数成功时返回 0，失败时返回负数。

注意时间是 Libevent 开始回调时缓存的，或多或少都是不准确的。如果回调函数运行了很长时间，则可能非常不准确。要强制立刻更新缓存，可以调用：

接口

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

该函数成功时返回 0，失败时返回 -1，`event_base` 不在运行循环时没有效果。

`event_base_gettimeofday_cached()` 函数首次出现在 Libevent 2.0.4-alpha中，Libevent 2.1.1-alpha 添加了 `event_base_update_cache_time()`。

转储 event_base 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！），有时可能需要一份 `event_base` 中所有事件的完整列表以及它们的状态。调用 `event_base_dump_events()` 会将这个列表写进提供的标准输入输出（`stdio`）文件。

这个列表是为了便于人类阅读，格式将会在未来版本 Libevent 中发生变化。

这个函数是 Libevent 2.0.1-alpha 引入的。

在 `event_base` 中的每个事件上运行函数

接口

```
typedef int (*event_base_foreach_event_cb)(const struct event_base *,
    const struct event *, void *);

int event_base_foreach_event(struct event_base *base,
                             event_base_foreach_event_cb fn,
                             void *arg);
```

您可以使用 `event_base_foreach_event()` 来遍历所有与 `event_base` 相关的活跃或未决事件。提供的回调函数将对每个事件调用一次，调用顺序是未指定的。
`event_base_foreach_event` 的第三个参数将会作为回调函数的第三个参数传递。

回调函数务必不要修改接收到的任何事件，或向 `event_base` 中添加任何事件，或修改与 `event_base` 相关的任何事件，否则会发生未定义的行为，上至崩溃和堆溢出。

`event_base_foreach_event()` 调用期间会锁定 `event_base`，其他线程对 `event_base` 的任何操作都会被阻塞，因此请确保回调函数不会耗费太多时间。

该函数首次首先在 Libevent 2.1.2-alpha 中。

废弃的事件循环函数

正如前面讨论过的，旧版本的 Libevent API 有一个 "当前" `event_base` 的概念。

本节介绍的一些事件循环函数有处理 "当前" `event_base` 版本，这些函数与目前版本的函数行为相同，除了没有 `base` 参数。

目前函数	废弃的 "当前" 函数
event_base_dispatch()	event_dispatch()
event_base_loop()	event_loop()
event_base_loopexit()	event_loopexit()
event_base_loopbreak()	event_loopbreak()

> 注意：由于 `event_base` 在 Libevent 2.0 之前不支持锁定，这些函数并不完全是线程安全的：不允许在执行事件循环的线程之外的线程中调用 `_loopbreak()` 或 `_loopexit()` 函数。