R2：创建 event_base

在使用任何有趣的 Libevent 函数之前，您需要申请一个或多个 event_base 结构体。每个结构体持有一个事件集，并可以轮询以确定哪些事件处于活跃状态。

如果 `event_base` 设置成使用锁，则可以在多个线程中安全地访问。然而，其事件循环只能运行在一个线程中。如果希望使用多个线程轮询 IO，则需要为每个线程都分配 `event_base`。

> 提示：未来版本的 Libevent 可能会支持跨线程的 event_base。

每个 `event_base` 都有一个 "方法"，或者后端来检测哪些事件已经就绪，可识别的方法有：

- select
- poll
- epoll
- kqueue
- evport
- win32

用户可以通过环境变量禁用特定后端。例如，可以通过设置 `EVENT_NOKQUEUE` 环境变量来禁用 `kqueue`，其他后端同理。如果想要在程序内部禁用某个后端，请参考下方关于 `event_config_avoid_method()` 的说明。

创建默认的 event_base

`event_base_new()` 函数申请并返回一个新的包含默认配置 `event_base`。该函数检查环境变量并返回一个指向 `event_base` 的指针，如果发生错误则返回 `NULL`。

在选择方法时，该函数会选择受操作系统支持的最快的方法。

接口

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

对于大多数程序来说，这个函数就足够了。

`event_base_new()` 函数声明在 `<event2/event.h>` 中，首次出现在 Libevent 1.4.3 中。

创建复杂的 event_base

如果想对 `event_base` 的种类获取更多控制权，您需要使用 `event_config`。`event_config` 是一个不透明的结构，保存了有关 `event_base` 的首选项的信息。需要创建 `event_base` 时，将 `event_config` 传递给 `event_base_new_with_config()` 函数。

接口

```
struct event_config *event_config_new(void);
struct event_base *event_base_new_with_config(const struct event_config *cfg);
void event_config_free(struct event_config *cfg);
```

通过这些函数分配 `event_base`，首先需要调用 `event_config_new()` 函数分配一个新的 `event_config` 结构体，随后对 `event_config` 调用其他函数，设置需要的特性，最后调用 `event_base_new_with_config()` 完成创建。创建完成后，可以调用 `event_config_free()` 释放 `event_config` 结构体。

接口

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

enum event_method_feature {
    EV_FEATURE_ET = 0x01,
    EV_FEATURE_O1 = 0x02,
    EV_FEATURE_FDS = 0x04,
};
int event_config_require_features(struct event_config *cfg,
                                  enum event_method_feature feature);

enum event_base_config_flag {
    EVENT_BASE_FLAG_NOLOCK = 0x01,
    EVENT_BASE_FLAG_IGNORE_ENV = 0x02,
    EVENT_BASE_FLAG_STARTUP_IOCP = 0x04,
    EVENT_BASE_FLAG_NO_CACHE_TIME = 0x08,
    EVENT_BASE_FLAG_EPOLL_USE_CHANGELIST = 0x10,
    EVENT_BASE_FLAG_PRECISE_TIMER = 0x20
};
int event_config_set_flag(struct event_config *cfg,
    enum event_base_config_flag flag);
```

调用 `event_config_avoid_method` 可以通过名字告知 Libevent 避免使用指定的后端；调用 `event_config_require_feature()` 可以告知 Libevent 不要使用任何不能提供所有特性集的后端；调用 `event_config_set_flag()` 可以告知 Libevent 在创建 `event_base` 时设置一个或多个将在下面介绍的运行时标志。

`event_config_require_feature` 可识别的的特性值有：

EV_FEATURE_ET

要求支持边缘触发 IO 的后端。

EV_FEATURE_O1

要求后端可以在 $O(1)$ 时间添加、删除或激活单个事件。

EV_FEATURE_FDS

要求后端支持任意类型的文件描述符，而不仅仅是套接字。

`event_config_set_flag()` 可以识别的标志有：

EVENT_BASE_FLAG_NOLOCK

不为 `event_base` 分配锁。设置这个标志也许会节省加锁和释放锁的时间，但是在多线程环境下会让 `event_base` 变得不安全且失去功能。

EVENT_BASE_FLAG_IGNORE_ENV

选择后端函数时不检查 `EVENT_*` 开头的环境变量。使用这个标识前请仔细考虑：这会让用户调试程序与 Libevent 之间的交互变得困难。

EVENT_BASE_FLAG_STARTUP_IOCP

只针对 Windows，这个标志使 Libevent 在启动时就启用所有必要的 IOCP 调度，而不是按需启用。

EVENT_BASE_FLAG_NO_CACHE_TIME

不在每次事件循环准备调用超时回调函数之前检查当前事件，而是在每次调用回调函数之后检查。这会消耗比预期更多的 CPU 资源，所以要小心！

EVENT_BASE_FLAG_EPOLL_USE_CHANGELIST

告知 Libevent 如果决定选用 `epoll` 作为后端，则可以安全地使用更快的基于 "changelist" 的后端。如果一个文件描述符在调用后端调度函数期间需要多次修改其状态，`epoll-changelist` 后端可以避免不必要的系统调用。但是如果传递任何使用 `dup()` 或其变体函数拷贝的文件描述符给 Libevent，`epoll-changelist` 后端将会触发一个内核 bug，导致错误的结果。如果不适用 `epoll` 作为后端，则该标志不起作用。您也可以通过设置 `EVENT_EPOLL_USE_CHANGELIST` 环境变量来开启 `epoll-changelist`。

EVENT_BASE_FLAG_PRECISE_TIMER

默认情况下，Libevent 会尝试使用操作系统提供的最快的定时器。如果存在一个较慢但能提供更细粒度定时精度的定时器，这个标志会告知 Libevent 替换定时器。如果操作系统没有提供这样更慢但更细粒度的定时器，这个标志不会生效。

以上设置 `event_config` 的函数都在成功时返回 0，失败时返回 -1。

> 注意：event_config 要求的后端很可能是您的操作系统无法提供的。例如就 Libevent 2.0.1-alpha 而言，Windows 平台上不存在 $O(1)$ 的后端，Linux 平台上没有同时满足 `EV_FEATURE_FDS` 和 `EV_FEATURE_O1` 的后端。如果您构造了一个 Libevent 无法满足的配置，`event_base_new_with_config()` 会返回 `NULL`。

接口

```
int event_config_set_num_cpus_hint(struct event_config *cfg, int cpus);
```

这个函数目前只在 Windows 上使用 IOCP 时才会用到，未来可能在其他平台也可用。调用这个函数会告知 `event_config` 生成的 `vent_base` 在多线程运行时尽量充分利用给定数量的 CPU。注意这只是一个提示，`event_base` 最终可能会使用比您选择的更多或更少的 CPU。

接口

```
int event_config_set_max_dispatch_interval(struct event_config *cfg,
    const struct timeval *max_interval, int max_callbacks,
    int min_priority);
```

这个函数通过限制在检查高优先级事件之前可以调用的低优先级事件回调函数的数量，来防止优先级反转。如果 `max_interval` 不是 `NULL`，事件循环检查每次回调后的事件，如果时间已经流逝了 `max_interval`，则重新扫描高优先级事件。如果 `max_callbacks` 非负，事件循环也会在 `max_callbacks` 次回调后检查检查更多的事件。这些规则可用于任何 `min_priority` 或更高优先级的事件。

示例：优先使用边缘触发后端

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

/* 如果可能的话，我的程序希望使用边缘触发事件。
   因此我会尝试获取 base 两次：一次使用非边缘触发，
   一次不使用。 */
for (i=0; i<2; ++i) {
    cfg = event_config_new();

    /* 我不想使用 select。 */
    event_config_avoid_method(cfg, "select");
```

```
    if (i == 0)
        event_config_require_features(cfg, EV_FEATURE_ET);

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

    /* 如果运行到了这里，意味着 event_base_new_with_config() 返回了 NULL。
       如果这是首次循环，我们还会尝试不设置 EV_FEATURE_ET。如果这是第二次循环，
       那就放弃。 */
}
```

示例：避免优先级反转

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

cfg = event_config_new();
if (!cfg)
   /* 处理错误 */;

/* 我的事件有两种优先级。我预期一些优先级为 priority-1 的事件运行回调会比较慢，
   因此我希望不超过 100 毫秒（或 5 次回调）检查一次优先级为 priority-0 的事件。 */
struct timeval msec_100 = { 0, 100*1000 };
event_config_set_max_dispatch_interval(cfg, &msec_100, 5, 1);

base = event_base_new_with_config(cfg);
if (!base)
   /* 处理错误 */;

event_base_priority_init(base, 2);
```

这些函数和数据类型定义在 `<event2/event.h>` 中。

`EVENT_BASE_FLAG_IGNORE_ENV` 标志首次出现在 Libevent 2.0.2-alpha 中；
`EVENT_BASE_FLAG_PRECISE_TIMER` 标志首次出现在 Libevent 2.1.2-alpha 中；
`event_config_set_num_cpus_hint()` 函数首次出现在 Libevent 2.0.7-rc 中；
`event_config_set_max_dispatch_interval()` 函数首次出现在 2.1.1-alpha 中；本小节的其余部分均首次出现在 Libevent 2.0.1-alpha 中。

检查 event_base 的后端方法

有时需要检查 `event_base` 支持哪些特性，或使用哪种方法。

接口

```
const char **event_get_supported_methods(void);
```

`event_get_supported_methods()` 方法返回数组指针，该数组存储了当前版本的 Libevent 所支持的方法名，数组的最后一个元素是 `NULL`。

示例

```
int i;
const char **methods = event_get_supported_methods();
printf("Starting Libevent %s.  Available methods are:\n",
    event_get_version());
for (i=0; methods[i] != NULL; ++i) {
    printf("    %s\n", methods[i]);
}
```

> 注意：该函数返回的是 Libevent 支持的方法列表，但是您的操作系统实际上可能不能完全支持这些方法。例如，在某些版本的 OSX 上， `kqueue` 的 bug 多到无法使用。

接口

```
const char *event_base_get_method(const struct event_base *base);
enum event_method_feature event_base_get_features(const struct event_base *base);
```

`event_base_get_method()` 函数返回 `event_base` 实际使用的方法名。
`event_base_get_features()` 函数返回 `event_base` 支持的特性的比特掩码。

示例

```
struct event_base *base;
enum event_method_feature f;

base = event_base_new();
if (!base) {
    puts("Couldn't get an event_base!");
} else {
    printf("Using Libevent with backend method %s.",
```

```
        event_base_get_method(base));
    f = event_base_get_features(base);
    if ((f & EV_FEATURE_ET))
        printf("  Edge-triggered events are supported.");
    if ((f & EV_FEATURE_O1))
        printf("  O(1) event notification is supported.");
    if ((f & EV_FEATURE_FDS))
        printf("  All FD types are supported.");
    puts("");
}
```

这些函数声明在 `<event2/event.h>` 中，`event_base_get_method()` 首次出现在 Libevent 1.4.3 中，其余首次出现在 Libevent 2.0.1-alpha 中。

释放 event_base

当 `event_base` 使用完毕，您可以通过 `event_base_free()` 释放。

接口

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

注意：这个函数不会释放当前与 `event_base` 关联的事件，或者关闭它们的套接字，或者释放任何它们的指针。

`event_base_free()` 函数声明在 `<event2/event.h> 中，首次出现在 Libevent 1.2 中实现。

设置 event_base 优先级

Libevent 支持为同一个事件设置多个优先级。然后，默认情况下 `event_base` 只支持一个优先级。您可以通过调用 `event_base_priority_init()` 来设置 `event_base` 的优先级。

接口

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

该函数成功返回 0，失败返回 -1。参数 `base` 是需要修改的 `event_base`，`n_priorities` 是支持的优先级的数量，该值至少是 1。事件的优先级范围为 0（最重要） 到 `n_priorities` - 1（最不重要）。

常数 `EVENT_MAX_PRIORITIES` 设定了 `n_priorities` 的上界，调用上述函数时给定的 `n_priorities` 超过这个值是错误的。

> 注意：您必须在任何事件激活之前调用该函数，最好在创建 `event_base` 后立刻调用。

您可以调用 `event_base_getnpriorities()` 获取某个 `event_base` 当前支持的优先级数量。

接口

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

返回值是 `event_base` 中设置的支持的优先级数量。如果 `event_base_get_npriorities()` 返回 3，则允许的优先级值为 0，1 和 2。

示例

关于示例，请查看下方关于 `event_priority_set` 的文档。

默认情况下，所有与该 `event_base` 关联的新事件的优先级会被初始化成 `n_priorities` / 2。

`event_base_priority_init` 函数声明在 `<event2/event.h>` 中，自 Libevent 1.0 起可用。`event_base_get_npriorities()` 函数是 Libevent 2.1.1-alpha 新引入的。

fork() 后重新初始化 event_base

并非所有事件后端都在调用 `fork()` 后仍然可用。因此，如果您的程序在创建新进程时使用了 `fork()` 或相关的系统调用，并且希望 fork 之后继续使用 `event_base`，就需要重新初始化。

接口

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

该函数成功返回 0，失败返回 -1。

示例

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

/* ... 向 event_base 添加一些事件 ... */

if (fork()) {
    /* 父进程 */
    continue_running_parent(base); /*...*/
```

```
} else {
    /* 子进程 */
    event_reinit(base);
    continue_running_child(base); /*...*/
}
```

`event_reinit()` 函数声明在 `<event2/event.h>` 中，在 Libevent 1.4.3-alpha 中首次可用。

废弃的 event_base 函数

老版本的 Libevent 非常依赖于 "当前" `event_base` 的概念，"当前" `event_base` 是一个所有线程共享的全局设置。如果您忘记指定 `event_base`，则会使用 "当前" 的。由于 `event_base` 不是线程安全的，这很容易出错。

老版本的 Libevent 没有 `event_base_new()`，而是：

接口

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

这个函数的行为与 `event_base_new()` 类似，并且会把申请的 `event_base` 设置成 "当前" 的。没有其他方法可以改变 "当前" `event_base`。

本小节的一些函数有操作 "当前" `event_base` 的变种，这些函数与新版本函数的行为一致，只是它们没有 `base` 参数。

目前函数	废弃的 "当前" 函数
event_base_priority_init()	event_priority_init()
event_base_get_method()	event_get_method()