R8：接受 TCP 连接的监听器

监听器（`evconnlistener`）机制提供了监听并接受 TCP 连接的机制。

本章所有的函数和类型都定义在 `<event2/listener.h>`，除非额外说明，均首次出现在 Libevent 2.0.2-alpha 中。

创建或释放监听器

接口

```
struct evconnlistener *evconnlistener_new(struct event_base *base,
    evconnlistener_cb cb, void *ptr, unsigned flags, int backlog,
    evutil_socket_t fd);
struct evconnlistener *evconnlistener_new_bind(struct event_base *base,
    evconnlistener_cb cb, void *ptr, unsigned flags, int backlog,
    const struct sockaddr *sa, int socklen);
void evconnlistener_free(struct evconnlistener *lev);
```

两个 `evconnlistener_new*()` 函数都会申请并返回一个新的连接监听对象。连接监听器使用 `event_base` 检测给定的监听套接字上是否有新的 TCP 连接。当出现新的连接时，将调用给定的回调函数。

在这两个函数中，`base` 参数是监听器用于监听连接的 `event_base`。`cb` 函数是接收到新的连接时调用的回调函数，如果 `cb` 是 `NULL`，监听器被认为是无效的，知道设置了回调函数。`ptr` 指针会被传递给回调函数。`flag` 参数控制回调函数的行为，下面会有详细介绍。`backlog` 参数控制网络栈同时允许的最大挂起连接数量，更多信息请参考您的系统中 `listen()` 函数的相关文档。如果 `backlog` 是负的，Libevent 选择一个合适的值；如果是 0，Libevent 会假定您在提供套接字 `fd` 之前已经调用了 `listen()`。

这两个函数的区别之处在于如何设置监听套接字。`evconnlistener_new()` 函数假定您已经将套接字绑定到了需要监听的端口，并通过 `fd` 传递套接字。如果想要 Libevent 分配并绑定套接字，调用 `evconnlistener_new_bind()` 并传递需要绑定的 `sockaddr` 及其长度。

> 提示：[使用 `evconnlistener_new` 时，请确保使用 `evutil_make_socket_nonblocking` 或者其他手动方式将套接字设置成非阻塞模式。如果套接字处于阻塞模式，可能会发生未定义的行为。]

要释放一个连接监听器，将其传递给 `evconnlistener_free()` 即可。

可识别的标志

以下是可以通过 `evconnlistener_new()` 函数的 `flags` 参数传递的标志，您可以传递传递以下标志的任何一个或组合。

LEV_OPT_LEAVE_SOCKETS_BLOCKING

默认情况下，连接监听器接受的新连接都会被设置成非阻塞以便在 Libevent 中使用。如果不想要该行为可以设置此标志。

LEV_OPT_CLOSE_ON_FREE

如果设置了该标志，释放连接监听器时会关闭其底层套接字。

LEV_OPT_CLOSE_ON_EXEC

如果设置了此标志，连接监听器会为底层套接字设置 close-on-exec 标志，更多信息请参考平台关于 `fcntl` 和 `FD_CLOEXEC` 相关文档。

LEV_OPT_REUSEABLE

在一些平台中的默认配置下，一旦监听套接字关闭，该套接字端口短时间内不能重复使用。设置此标志会让 Libevent 将套接字设置成可重用，即套接字一旦关闭，其他套接字可以在相同端口上开启监听。

LEV_OPT_THREADSAFE

为监听器分配锁，使其在多线程使用中安全，于 Libevent 2.0.8-rc 新引入。

LEV_OPT_DISABLED

将监听器初始化为禁用状态，而非启用。可以通过手动调用 `evconnlistener_enable()` 启用，于 Libevent 2.1.1-alpha 新引入。

LEV_OPT_DEFERRED_ACCEPT

如果可能的话，通知内核在接收到套接字上的数据并准备好读取之前，不要宣布套接字已被接受。如果您的协议不是以客户端传输数据开始的，则不要启用该标志，因为在这种情况下内核可能永远不会通知程序已经建立连接。并不是所有的操作系统都支持这个选项，如果不支持，则该标志没有任何效果。该标志是 Libevent 2.1.1-alpha 新引入的。

连接监听器回调

接口

```
typedef void (*evconnlistener_cb)(struct evconnlistener *listener,
    evutil_socket_t sock, struct sockaddr *addr, int len, void *ptr);
```

当接收到新的连接时，提供的回调函数将会被调用。`listener` 参数是接收连接请求的连接监听器，`sock` 参数是新的套接字，`addr` 和 `len` 参数是新连接的地址和地址长度，`ptr` 参数是调用 `evconnlistener_new()` 时用户提供的指针。

开启和关闭监听器

接口

```
int evconnlistener_disable(struct evconnlistener *lev);
int evconnlistener_enable(struct evconnlistener *lev);
```

这些函数可以暂时关闭或重新开启监听器。

调整监听器回调

接口

```
void evconnlistener_set_cb(struct evconnlistener *lev,
    evconnlistener_cb cb, void *arg);
```

该函数为监听器调整回调函数及其参数，于 2.0.9-rc 新引入。

检查监听器

接口

```
evutil_socket_t evconnlistener_get_fd(struct evconnlistener *lev);
struct event_base *evconnlistener_get_base(struct evconnlistener *lev);
```

这些函数分别返回监听器关联的套接字和 `event_base`。

`evconnlistener_get_fd()` 函数于 Libevent 2.0.3-alpha 首次出现。

检测错误

您可以通过设置错误回调以在 `accept()` 调用失败时获取相关信息，这在面临一个不解决就会造成整个线程锁定的错误时是很重要的。

接口

```
typedef void (*evconnlistener_errorcb)(struct evconnlistener *lis, void *ptr);
void evconnlistener_set_error_cb(struct evconnlistener *lev,
    evconnlistener_errorcb errorcb);
```

如果使用 `evconnlistener_set_error_cb()` 为监听器设置错误回调，每次监听器发生错误时都会调用该回调。该回调会接收监听器作为第一个参数，`evconnlistener_new()` 的 `ptr` 作为第二个参数。

该函数于 Libevent 2.0.8-rc 引入。

示例代码：回显服务器

示例

```
#include <event2/listener.h>
#include <event2/bufferevent.h>
#include <event2/buffer.h>

#include <arpa/inet.h>

#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

static void
echo_read_cb(struct bufferevent *bev, void *ctx)
{
        /* 当 bev 有数据可读时会调用此回调。 */
        struct evbuffer *input = bufferevent_get_input(bev);
        struct evbuffer *output = bufferevent_get_output(bev);

        /* 将所有数据从输入缓冲区复制到输出缓冲区。 */
        evbuffer_add_buffer(output, input);
}

static void
echo_event_cb(struct bufferevent *bev, short events, void *ctx)
{
        if (events & BEV_EVENT_ERROR)
                perror("Error from bufferevent");
        if (events & (BEV_EVENT_EOF | BEV_EVENT_ERROR)) {
                bufferevent_free(bev);
        }
```

```
}

static void
accept_conn_cb(struct evconnlistener *listener,
    evutil_socket_t fd, struct sockaddr *address, int socklen,
    void *ctx)
{
        /* 接收了一个新的连接！为其分配缓冲事件。 */
        struct event_base *base = evconnlistener_get_base(listener);
        struct bufferevent *bev = bufferevent_socket_new(
                base, fd, BEV_OPT_CLOSE_ON_FREE);

        bufferevent_setcb(bev, echo_read_cb, NULL, echo_event_cb, NULL);

        bufferevent_enable(bev, EV_READ|EV_WRITE);
}

static void
accept_error_cb(struct evconnlistener *listener, void *ctx)
{
        struct event_base *base = evconnlistener_get_base(listener);
        int err = EVUTIL_SOCKET_ERROR();
        fprintf(stderr, "Got an error %d (%s) on the listener. "
                "Shutting down.\n", err, evutil_socket_error_to_string(err));

        event_base_loopexit(base, NULL);
}

int
main(int argc, char **argv)
{
        struct event_base *base;
        struct evconnlistener *listener;
        struct sockaddr_in sin;

        int port = 9876;

        if (argc > 1) {
                port = atoi(argv[1]);
        }
        if (port<=0 || port>65535) {
                puts("Invalid port");
                return 1;
        }

        base = event_base_new();
        if (!base) {
                puts("Couldn't open event base");
```

```
                return 1;
        }

        /* 使用 sockaddr 之前先清空，以防有额外的平台特定字段造成干扰。 */
        memset(&sin, 0, sizeof(sin));
        /* INET 地址 */
        sin.sin_family = AF_INET;
        /* 在 0.0.0.0 上监听 */
        sin.sin_addr.s_addr = htonl(0);
        /* 在给定的端口上监听。 */
        sin.sin_port = htons(port);

        listener = evconnlistener_new_bind(base, accept_conn_cb, NULL,
            LEV_OPT_CLOSE_ON_FREE|LEV_OPT_REUSEABLE, -1,
            (struct sockaddr*)&sin, sizeof(sin));
        if (!listener) {
                perror("Couldn't create listener");
                return 1;
        }
        evconnlistener_set_error_cb(listener, accept_error_cb);

        event_base_dispatch(base);
        return 0;
}
```