



Join GitHub today

Dismiss

GitHub is home to over 36 million developers working together to host and review code, manage projects, and build software together.

[Sign up](#)

Tree: 62e84b27e7 ▾

[liuyanjie.github.io](#) / [source](#) / [_posts](#) / 4-libuv源码分析（五）IO观察者（io_watcher）.md[Find file](#)[Copy path](#)

liuyanjie add articles

2c06277 on Apr 24

[1 contributor](#)

723 lines (558 sloc) | 27.6 KB

[Raw](#)[Blame](#)[History](#)

title

date

updated

tags

categories

title	date	updated	tags			categories
libuv源码分析（五）IO观察者（io_watcher）	2019-04-23 15:00:05 UTC	2019-04-24 01:31:28 UTC	libuv	node.js	eventloop	源码分析

在 libuv 内部，对所有IO操作进行了统一的抽象，在底层的操作系统IO操作基础上，结合事件循环机制，实现了IO观察者，对应结构体 `uv__io_s`。其他 `handle` 通过内嵌I/O观察者的方式获得IO的监测能力，例如有

`uv_stream_t`、`uv_udp_t`、`uv_poll_t` 及 `uv_stream_t` 的派生类型 `uv_tcp_t`、`uv_pipe_t`、`uv_tty_t`。
`uv_stream_t` 是一个抽象基类，一般不直接使用。内嵌IO观察者的 `handle` 本身就是IO观察者，和面向对象语言中常提到的 `has-a` 是一样的，所以，可以说 `uv_stream_t` 是一个I/O观察者，它的派生类也是I/O观察者。这类似于多重继承，即继承 `uv_handle_t`，又继承了 `uv__io_t`，所以即是 `handle`，又是 I/O观察者。

应用程序 通过创建并初始化 `handle` 或 `request` 的方式创建并初始化并注册I/O观察者，注册后被插入到 `loop-watchers` 队列中，并在 `start` 后将I/O观察者标记为准备状态，在事件循环启动后对 `loop-watchers` 队列中所有I/O观察者关注的文件描述符进行轮询（linux下使用 `epoll`）并在I/O事件触发时调用对应的回调函数。

libuv 的 15 个 `handle` 类型中，属于I/O观察者就有 7 个，还有几个类型的 `handle` 虽然不直接内嵌I/O观察者，但内部实现依然依赖I/O观察者。实际上，只有 `uv_timer_t`、`uv_prepare_t`、`uv_check_t`、`uv_idle_t` 这 4 个 `handle` 与I/O观察者无直接关系。足以见I/O观察者在 libuv 中的重要性，理解I/O观察者对于理解 libuv 甚至是 node.js 都是十分重要的。

下面开始讲其实现细节。

uv__io_s

`uv__io_s` 结构体 即为 I/O观察者 的抽象数据结构：

```
typedef struct uv__io_s uv__io_t;

struct uv__io_s {
    uv__io_cb cb;
    void* pending_queue[2];
    void* watcher_queue[2];
    unsigned int pevents; /* Pending event mask i.e. mask at next tick. */
    unsigned int events; /* Current event mask. */
    int fd;
    UV_IO_PRIVATE_PLATFORM_FIELDS
};
```

主要字段用途介绍：

- `fd`：感兴趣的 文件描述符；
- `cb`：当文件描述符 `fd` 上有I/O事件发生时，调用该函数进行处理，注意，该函数的类型 `uv__io_cb` 是内部类型，并不对外暴露，所以 `cb` 自然也是内部提供的。libuv针对不同类型的IO观察者实现了多个不同的 `cb` 函数；
- `watcher_queue`：作为队列节点，插入到 `loop->watcher_queue` 队列中，所有的I/O观察者都会被插入到这个队列中；
- `pending_queue`：作为队列节点，插入到 `loop->pending_queue` 队列中，所有被挂起的I/O观察者都会被插入到这个队列中；
- `pevents`：下次事件循环使用的事件掩码；
- `events`：当前正在使用的事件掩码。 `events |= pevents`。

在 `uv_stream_t`（包括 `uv_tcp_t`、`uv_pipe_t`、`uv_tty_t`）、`uv_udp_t`、`uv_poll_t` 类型结构体内部都内嵌了 `uv__io_t`：

```
uv__io_t io_watcher;
```

另外，在 `uv_loop_s` 中，同样也内嵌了多个 `uv__io_t` 关联字段：

<https://github.com/libuv/libuv/blob/v1.28.0/include/uv/unix.h#L218>

```
#define UV_LOOP_PRIVATE_FIELDS \
    uv__io_t** watchers; \
    unsigned int nwatchers; \
    unsigned int nfds; \
    uv__io_t async_io_watcher; \
    uv__io_t signal_io_watcher; \
    UV_PLATFORM_LOOP_FIELDS
```

linux 下的 `UV_PLATFORM_LOOP_FIELDS` 定义：

```
#define UV_PLATFORM_LOOP_FIELDS \
    uv__io_t inotify_read_watcher; \
    void* inotify_watchers; \
    int inotify_fd;
```

主要字段用途介绍：

- `watchers`：咋看是一个二级指针，实际上是当做数组来使用的，数组的每一项存储了 `uv__io_t*` 类型的指针，使用示例：`w = loop->watchers[i];`，所有的I/O观察者都有以文件描述符 `fd` 作为数组下标保存在这个数组中，可以通过 `fd` 快速找到对应的I/O观察者。请仔细理解C语言数组和指针的关系，这里不能当做二级指针理解；注意：
 - 该指针指向的内存空间是动态分配的，以适应I/O观察者的变化，动态分配算法见函数 `maybe_resize`；
 - 注意：实际分别长度为 `nwatchers + 2`；
- `nwatchers`：记录了能容纳的IO观察者的数量，`len(watchers) == nwatchers + 2`；

- `nfds` : 记录了 `watchers` 中文件描述符的数量，也是实际使用量；
- `async_io_watcher` : `loop` 内嵌的I/O观察者，这个I/O观察者用于其他异步任务（运行在其他线程中）通过IO与主事件循环进行通讯；
- `signal_io_watcher` : `loop` 内嵌的I/O观察者，这个I/O观察者用于接收信号IO事件的。在 `libuv` 中，信号也被异步化到事件循环中去处理了；
- `inotify_read_watcher` : `UV_PLATFORM_LOOP_FIELDS` 宏在 `linux` 平台下展开之后会有一个 `inotify_read_watcher`，这个是用来在 `linux` 下支持 `uv_fs_event_t` 的，在其他 `uni*` 平台上，有对应的 `event_watcher`。

除以上介绍的 `handle` 外，`uv_fs_poll_t` 实际通过 `uv_async_t` 来支持其文件状态轮询的。

至此，我们已经把所有和I/O观察者相关的 `handle` 都提及到了，可以发现，`libuv` 中绝大多数 `handle` 都是I/O相关的，少部分 `handle` 简介通过I/O观察者实现功能。所以，`libuv` 可以说是专门为I/O而设计的，正如文档中所诉，它就是专注于异步I/O的程序库。下面，我们将介绍I/O观察者的工作原理，理解了I/O观察者就可以很容易的弄懂大部分相关的 `handle` 的工作原理。

因为 `uv__io_t` 是内部结构，并不对外暴露，所以我们以 `uv_poll_t` 作为入口，探索其内部的 `uv__io_t` 工作原理。

`uv_poll_t` 用于监控文件描述符的 可读/可写 状态，和 `poll` 系统调用的用途类型，不过 `uv_poll_t` 是异步非阻塞，而操作系统的原生 `poll` 函数是同步阻塞的。

`uv_poll_t` 是I/O观察者的简单封装后的应用程序接口，它只关心状态变化并调用用户层代码，I/O事件之后将由外部库处理。相比之下，`uv_stream_t` 则复杂的多，`uv_stream_t` 更进一步封装了数据读写操作等方面的能力，并将数据派发给不同类型的派生 `handle` 处理。

Init

uv_poll_init

使用文件描述符初始化 poll。

以下是 uv_poll_init 的具体实现：

<https://github.com/libuv/libuv/blob/v1.28.0/src/unix/poll.c#L67>

```
int uv_poll_init(uv_loop_t* loop, uv_poll_t* handle, int fd) {
    int err;

    if (uv__fd_exists(loop, fd))
        return UV_EEXIST;

    err = uv__io_check_fd(loop, fd);
    if (err)
        return err;

    /* If ioctl(FIONBIO) reports ENOTTY, try fcntl(F_GETFL) + fcntl(F_SETFL).
     * Workaround for e.g. kqueue fds not supporting ioctls.
     */
    err = uv__nonblock(fd, 1);
    if (err == UV_ENOTTY)
        if (uv__nonblock == uv__nonblock_ioctl)
            err = uv__nonblock_fcntl(fd, 1);

    if (err)
        return err;

    uv__handle_init(loop, (uv_handle_t*) handle, UV_POLL);
    uv__io_init(&handle->io_watcher, uv__poll_io, fd);
    handle->poll_cb = NULL;
    return 0;
}
```

以上代码先是检查了文件描述符是否存在、文件描述符是否已经纳入监控等，然后调用 `uv__handle_init` 和 `uv__io_init` 进行基类初始化，`uv__io_init` 是I/O观察者的初始化函数。

`uv__io_init`

直接上源码：

<https://github.com/libuv/libuv/blob/v1.28.0/src/unix/core.c#L805>

```
void uv__io_init(uv__io_t* w, uv__io_cb cb, int fd) {
    assert(cb != NULL);
    assert(fd >= -1);
    QUEUE_INIT(&w->pending_queue);
    QUEUE_INIT(&w->watcher_queue);
    w->cb = cb;
    w->fd = fd;
    w->events = 0;
    w->pevents = 0;

    #if defined(UV_HAVE_KQUEUE)
        w->rcount = 0;
        w->wcount = 0;
    #endif /* defined(UV_HAVE_KQUEUE) */
}
```

`uv__io_init` 初始化了两个队列，绑定了回调函数和文件描述符，基本工作就做完了。

调用 `uv__io_init` 时传递的 `uv__poll_io` 是 libuv 内部实现的，用于在I/O事件到来时调用。

`uv__poll_io`

如果有I/O事件产生, `uv__poll_io` 会被 libuv 调用。

```
static void uv__poll_io(uv_loop_t* loop, uv__io_t* w, unsigned int events) {
    uv_poll_t* handle;
    int pevents;

    handle = container_of(w, uv_poll_t, io_watcher);

    /*
     * As documented in the kernel source fs/kernfs/file.c #780
     * poll will return POLLERR|POLLPRI in case of sysfs
     * polling. This does not happen in case of out-of-band
     * TCP messages.
     *
     * The above is the case on (at least) FreeBSD and Linux.
     *
     * So to properly determine a POLLPRI or a POLLERR we need
     * to check for both.
     */
    if ((events & POLLERR) && !(events & UV__POLLPRI)) {
        uv__io_stop(loop, w, POLLIN | POLLOUT | UV__POLLRDHUP | UV__POLLPRI);
        uv__handle_stop(handle);
        handle->poll_cb(handle, UV_EBADF, 0);
        return;
    }

    pevents = 0;
    if (events & POLLIN)
        pevents |= UV_READABLE;
    if (events & UV__POLLPRI)
        pevents |= UV_PRIORITIZED;
    if (events & POLLOUT)
        pevents |= UV_WRITABLE;
    if (events & UV__POLLRDHUP)
```



```
    pevents |= UV_DISCONNECT;

    handle->poll_cb(handle, 0, pevents);
}
```

`uv__poll_io` I/O事件的时候会被调用，工作逻辑如下：

1. 首先通过 `container_of` 获取 `handle` ；
2. 如果是一些异常的I/O事件，则会进入 `stop` 流程并调用 `handle->poll_cb` ；
3. 将事件记录到 `pevents` ；
4. 调用 `handle->poll_cb` 。

`handle->poll_cb` 是在 `start` 阶段设置的，所以 `uv__poll_io` 一定是在 `uv_poll_start` 调用后才能调用的，因为I/O事件在 `uv_poll_start` 后的下一次事件循环才能被处理。

`uv__poll_io` 比较简单，可以说就是直接调用用户层提供的回调函数，正如 `uv_poll_t` 的用途一样，负责监控文件描述符状态变化，但是不负责处理。

除了 `uv__poll_io` 外，还有多个同样功能的 `uv__io_cb` 类型的函数存在，他们用于不同的功能，通过全局搜索 `uv__io_init` 函数即可找到 `uv__io_init` 调用传递的不同 `uv__io_cb` 函数，如下：

- `uv__signal_event` ，用于处理 `loop->signal_io_watcher` 上的I/O事件；
- `uv__async_io` ，用于处理 `loop->async_io_watcher` 上的I/O事件；
- `uv__stream_io` ，用于处理 `stream_handle->io_watcher` 上的I/O事件；
- `uv__udp_io` ，用于处理 `udp_handle->io_watcher` 上的I/O事件。

以上这些 `uv__io_cb` 函数就没有 `uv__poll_io` 的实现简单了，它们都有更复杂的处理逻辑，如在 `uv__stream_io` 中，开始对文件描述符进行数据读写。

除了以上几个外，还有 `uv_fs_event_t` 相关I/O观察者的 `uv__poll_io`，因各平台实现不同，`uv__poll_io` 也有不同版本，就不列举了。

这些 `handle`，在 `Init`、`Start`、`Stop`、`Close` 等阶段多有不同，但是最大的不同还是在于I/O事件的处理函数 `uv__io_cb` 的实现不同，也就是以上列出的函数不同。

接下来，进入 `start` 阶段。

Start

`uv_poll_start`

开始对文件描述符进行事件轮询

```
int uv_poll_start(uv_poll_t* handle, int pevents, uv_poll_cb poll_cb) {
    int events;

    assert((pevents & ~(UV_READABLE | UV_WRITABLE | UV_DISCONNECT |
                        UV_PRIORITIZED)) == 0);
    assert(!uv__is_closing(handle));

    uv__poll_stop(handle);

    if (pevents == 0)
        return 0;

    events = 0;
    if (pevents & UV_READABLE)
        events |= POLLIN;
    if (pevents & UV_PRIORITIZED)
        events |= UV__POLLPRI;
    if (pevents & UV_WRITABLE)
```

```

    events |= POLLOUT;
    if (pevents & UV_DISCONNECT)
        events |= UV__POLLRDHUP;

    uv__io_start(handle->loop, &handle->io_watcher, events);
    uv__handle_start(handle);
    handle->poll_cb = poll_cb;

    return 0;
}

```

其他部分不用太多解释了，直接看关键步骤：`uv__io_start`

uv__io_start

```

void uv__io_start(uv_loop_t* loop, uv__io_t* w, unsigned int events) {
    assert(0 == (events & ~(POLLIN | POLLOUT | UV__POLLRDHUP | UV__POLLPRI)));
    assert(0 != events);
    assert(w->fd >= 0);
    assert(w->fd < INT_MAX);

    w->pevents |= events;
    maybe_resize(loop, w->fd + 1);

    #if !defined(__sun)
        /* The event ports backend needs to rearm all file descriptors on each and
         * every tick of the event loop but the other backends allow us to
         * short-circuit here if the event mask is unchanged.
         */
        if (w->events == w->pevents)
            return;
    #endif
}

```

```

if (QUEUE_EMPTY(&w->watcher_queue))
    QUEUE_INSERT_TAIL(&loop->watcher_queue, &w->watcher_queue);

if (loop->watchers[w->fd] == NULL) {
    loop->watchers[w->fd] = w;
    loop->nfds++;
}
}

```

关键步骤如下：

1. 将参数 `events` 或到 `w->pevents`，因为 `uv__io_start` 可以反复多次调用，相当于更新；
2. 按需扩容，判断当前的 `loop->watchers` 没有更多空间容纳 `fd` 及关联的I/O观察者，如果没有，指数级扩容，并拷贝内容到新的内存空间；
3. 将 `w->watcher_queue` 连接到 `loop->watcher_queue` 队列尾部，所有I/O观察者都被关联了起来。这里有个判断，为了防止重复操作。
4. 以 `w->fd` 为下标，将 `w` 保持到 `loop->watchers`，并更新引用计数 `loop->nfds`。

至此，完成了I/O观察者的准备工作，供事件循环处理。

Stop

```

int uv_poll_stop(uv_poll_t* handle) {
    assert(!uv__is_closing(handle));
    uv_poll_stop(handle);
    return 0;
}

```

```
static void uv__poll_stop(uv_poll_t* handle) {
    uv__io_stop(handle->loop,
                 &handle->io_watcher,
                 POLLIN | POLLOUT | UV__POLLRDHUP | UV__POLLPRI);
    uv__handle_stop(handle);
    uv__platform_invalidate_fd(handle->loop, handle->io_watcher.fd);
}
```

```
void uv__io_stop(uv_loop_t* loop, uv__io_t* w, unsigned int events) {
    assert(0 == (events & ~(POLLIN | POLLOUT | UV__POLLRDHUP | UV__POLLPRI)));
    assert(0 != events);

    if (w->fd == -1)
        return;

    assert(w->fd >= 0);

    /* Happens when uv__io_stop() is called on a handle that was never started. */
    if ((unsigned) w->fd >= loop->nwatchers)
        return;

    w->pevents &= ~events;

    if (w->pevents == 0) {
        QUEUE_REMOVE(&w->watcher_queue);
        QUEUE_INIT(&w->watcher_queue);

        if (loop->watchers[w->fd] != NULL) {
            assert(loop->watchers[w->fd] == w);
            assert(loop->nfds > 0);
            loop->watchers[w->fd] = NULL;
            loop->nfds--;
            w->events = 0;
        }
    }
}
```

```

    }
}
else if (QUEUE_EMPTY(&w->watcher_queue))
    QUEUE_INSERT_TAIL(&loop->watcher_queue, &w->watcher_queue);
}

```

`uv__io_stop` 实际上就是将I/O观察者从 `loop` 上移除，避免事件循环继续处理这个I/O观察者。

Close

`uv_poll_t` 并无 `close` 方法，但是存在 `uv__io_close` 方法，实现如下：

<https://github.com/libuv/libuv/blob/v1.28.0/src/unix/core.c#L882>

```

void uv__io_close(uv_loop_t* loop, uv__io_t* w) {
    uv__io_stop(loop, w, POLLIN | POLLOUT | UV__POLLRDHUP | UV__POLLPRI);
    QUEUE_REMOVE(&w->pending_queue);

    /* Remove stale events for this file descriptor */
    if (w->fd != -1)
        uv__platform_invalidate_fd(loop, w->fd);
}

```

调用了 `uv__io_stop` 完成 `close`。

Run : Poll for I/O

I/O观察者在事件循环启动后才会被真正的处理，主要是在 `uv__io_poll` 和 `uv__run_pending` 两个函数中处理的。

以下为 `uv__io_poll` 实现代码（含注释），这部分代码可以说是 libuv 中最核心的代码，因为这部分实现了 libuv 最核心功能异步IO的支持，实现了IO事件的轮询与事件的派发。

<https://github.com/libuv/libuv/blob/v1.28.0/src/unix/linux-core.c#L190>

```
void uv__io_poll(uv_loop_t* loop, int timeout) {
    /* A bug in kernels < 2.6.37 makes timeouts larger than ~30 minutes
     * effectively infinite on 32 bits architectures. To avoid blocking
     * indefinitely, we cap the timeout and poll again if necessary.
     */
    /* Note that "30 minutes" is a simplification because it depends on
     * the value of CONFIG_HZ. The magic constant assumes CONFIG_HZ=1200,
     * that being the largest value I have seen in the wild (and only once.)
     */
    static const int max_safe_timeout = 1789569;
    struct epoll_event events[1024];
    struct epoll_event* pe;
    struct epoll_event e;
    int real_timeout;
    QUEUE* q;
    uv__io_t* w;
    sigset_t sigset;
    sigset_t* pset;
    uint64_t base;
    int have_signals;
    int nevents;
    int count;
    int nfds;
    int fd;
    int op;
    int i;

    // 如果没有任何观察者，直接返回
    if (loop->nfds == 0) {
```

```

    assert(Queue_Empty(&loop->watcher_queue));
    return;
}

memset(&e, 0, sizeof(e));

// 向 `epoll` 系统注册所有 I/O观察者
while (!Queue_Empty(&loop->watcher_queue)) {
    // 获取队列头部, 并将队列从 `loop->watcher_queue` 摘除
    q = Queue_Head(&loop->watcher_queue);
    Queue_Remove(q);
    Queue_Init(q);

    // 获取 I/O观察者 结构
    w = Queue_Data(q, uv__io_t, watcher_queue);
    assert(w->pevents != 0);
    assert(w->fd >= 0);
    assert(w->fd < (int) loop->nwatchers);

    e.events = w->pevents;
    e.data.fd = w->fd;

    if (w->events == 0)
        op = EPOLL_CTL_ADD;
    else
        op = EPOLL_CTL_MOD;

    // 向 epoll 注册文件描述符及需要监控的IO事件
    /* XXX Future optimization: do EPOLL_CTL_MOD lazily if we stop watching
     * events, skip the syscall and squelch the events after epoll_wait().
     */
    if (epoll_ctl(loop->backend_fd, op, w->fd, &e)) {
        if (errno != EEXIST)
            abort();
    }
}

```



```

    assert(op == EPOLL_CTL_ADD);

    // loop->backend_fd 在事件循环初始化时也就是在 `uv_loop_init` 中 通过 `epoll_create` 创建
    /* We've reactivated a file descriptor that's been watched before. */
    if (epoll_ctl(loop->backend_fd, EPOLL_CTL_MOD, w->fd, &e))
        abort();
}

// 挂起的 `pevents` 设置为 `events` 将在下次事件循环中生效
w->events = w->pevents;
}

// 如果配置了 `UV_LOOP_BLOCK_SIGPROF`，则需要阻塞该信号
psigset = NULL;
if (loop->flags & UV_LOOP_BLOCK_SIGPROF) {
    sigemptyset(&sigset);
    sigaddset(&sigset, SIGPROF);
    psigset = &sigset;
}

assert(timeout >= -1);
base = loop->time;
// `count` 减少到 `0` 下面的循环跳出
count = 48; /* Benchmarks suggest this gives the best throughput. */
real_timeout = timeout;

// 开始进入 `epoll_pwait` 轮询IO事件
// 通常情况下，在 `timeout` 大于 `0` 的情况下，循环不断迭代到 `timeout` 减小到 `0` 时，循环跳出
// 在没有设置定时器的情况下，如果不出现错误，循环将一直不会跳出
// 以下循环主要由 `timeout` 和 `count` 控制是否跳出，符合整个事件循环
for (;;) {
    /* See the comment for max_safe_timeout for an explanation of why
     * this is necessary. Executive summary: kernel bug workaround.
     */
    if (sizeof(int32_t) == sizeof(long) && timeout >= max_safe_timeout)

```

```

    timeout = max_safe_timeout;

    // `epoll_pwait` 在 timeout 为 `0` 时立刻返回, 为 `-1` 时会一直阻塞直到有事件发生, 为 `正整数` 时则会最长
    // `nfds` 表示产生IO事件的文件描述符的数量, 为 `0` 则为没有事件发生, 可能因为超时时间到了, 或者 `timeout=0`
    // `events` 保存了从内核得到的事件集合, `nfds` 实际上相当于数组内有效数据的长度。
    nfds = epoll_pwait(loop->backend_fd,
                       events,
                       ARRAY_SIZE(events),
                       timeout,
                       psigset);

    /* Update loop->time unconditionally. It's tempting to skip the update when
     * timeout == 0 (i.e. non-blocking poll) but there is no guarantee that the
     * operating system didn't reschedule our process while in the syscall.
     */
    SAVE_ERRNO(uv__update_time(loop));

    // 没有事件发生
    if (nfds == 0) {
        // `timeout` 一定不为 `-1`
        assert(timeout != -1);

        // 如果 `timeout` 为 `0` 函数直接返回
        if (timeout == 0)
            return;

        /* We may have been inside the system call for longer than |timeout|
         * milliseconds so we need to update the timestamp to avoid drift.
         */
        // 减少下次 `epoll_pwait` 的 `timeout` 时间
        goto update_timeout;
    }

    // `epoll_wait` 返回错误
    if (nfds == -1) {

```

```

    if (errno != EINTR)
        abort();

    // 如果`timeout`为`-1`则继续循环
    if (timeout == -1)
        continue;

    // 如果`timeout`为`0`函数直接返回
    if (timeout == 0)
        return;

    /* Interrupted by a signal. Update timeout and poll again. */
    // 减少下次`epoll_pwait`的`timeout`时间
    goto update_timeout;
}

have_signals = 0;
nevents = 0;

assert(loop->watchers != NULL);
// `loop->watchers`的实际长度为`loop->nwatchers + 2`，观察者只使用`loop->watchers`的`0` ~ `lo
// `loop->nwatchers` ~ `loop->nwatchers + 1`被用来存储`events`和`nfds`，`uv__platform_invalid
// 后面以下两项又被赋值为`NULL`，`for`循环部分又没有代码能够使函数返回，所以看似以下两行并无实际作用
loop->watchers[loop->nwatchers] = (void*) events;
loop->watchers[loop->nwatchers + 1] = (void*) (uintptr_t) nfds;

// 处理IO事件：获取IO观察者，调用关联的回调函数
for (i = 0; i < nfds; i++) {
    pe = events + i;
    fd = pe->data.fd;

    /* Skip invalidated events, see uv__platform_invalidate_fd */
    if (fd == -1)
        continue;

```

```

assert(fd >= 0);
assert((unsigned) fd < loop->nwatchers);

w = loop->watchers[fd];

// 如果IO观察者已经被移除，则停止轮询这个文件描述符上的IO事件
// 在一次事件循环中，同一IO观察者上可能出现多次IO事件
// 继而调用多次回调函数，某次回调函数中，有可能移除了`w`自己
if (w == NULL) {
    /* File descriptor that we've stopped watching, disarm it.
     *
     * Ignore all errors because we may be racing with another thread
     * when the file descriptor is closed.
     */
    epoll_ctl(loop->backend_fd, EPOLL_CTL_DEL, fd, pe);
    continue;
}

// 进入事件处理

/* Give users only events they're interested in. Prevents spurious
 * callbacks when previous callback invocation in this loop has stopped
 * the current watcher. Also, filters out events that users has not
 * requested us to watch.
 */
pe->events &= w->pevents | POLLERR | POLLHUP;

/* Work around an epoll quirk where it sometimes reports just the
 * EPOLLERR or EPOLLHUP event. In order to force the event loop to
 * move forward, we merge in the read/write events that the watcher
 * is interested in; uv__read() and uv__write() will then deal with
 * the error or hangup in the usual fashion.
 *
 * Note to self: happens when epoll reports EPOLLIN|EPOLLHUP, the user
 * reads the available data, calls uv_read_stop(), then sometime later

```

```

* calls uv_read_start() again. By then, libuv has forgotten about the
* hangup and the kernel won't report EPOLLIN again because there's
* nothing left to read. If anything, libuv is to blame here. The
* current hack is just a quick bandaid; to properly fix it, libuv
* needs to remember the error/hangup event. We should get that for
* free when we switch over to edge-triggered I/O.
*/
if (pe->events == POLLERR || pe->events == POLLHUP)
    pe->events |=
        w->pevents & (POLLIN | POLLOUT | UV__POLLRDHUP | UV__POLLPRI);

// 如果存在有效事件
if (pe->events != 0) {
    /* Run signal watchers last. This also affects child process watchers
    * because those are implemented in terms of signal watchers.
    */
    // 如果 `w` 是 `loop->signal_IOWatcher` 在循环之外调用回调，避免重复触发回调
    if (w == &loop->signal_IOWatcher)
        have_signals = 1;
    else
        w->cb(loop, w, pe->events);
    // `w->cb` 是 `uv__io_cb` 类型的函数指针，对应的实现函数如 `uv__async_io` 已经在上文介绍
    // 这个回调函数指针由 libuv 内部实现的统一入口，在 `cb` 中再进行事件分发，交由特定逻辑处理

    nevents++;
}
}

// 如果信号事件触发
if (have_signals != 0)
    loop->signal_IOWatcher.cb(loop, &loop->signal_IOWatcher, POLLIN);

// 重新赋值为 `NULL`
loop->watchers[loop->nwatchers] = NULL;
loop->watchers[loop->nwatchers + 1] = NULL;

```

```

// 如果信号事件触发
if (have_signals != 0)
    return; /* Event loop should cycle now so don't poll again. */

// 如果 事件计数器 不为 `0`
if (nevents != 0) {
    // 如果 所有 所有文件描述符上都有事件产生 且 `count` 不为 `0`, 再循环一次
    if (nfdes == ARRAY_SIZE(events) && --count != 0) {
        /* Poll for more events but don't block this time. */
        timeout = 0;
        continue;
    }
    return;
}

// 如果 `timeout` 为 `0` 函数直接返回
if (timeout == 0)
    return;

// 如果 `timeout` 为 `-1` 则继续循环
if (timeout == -1)
    continue;

// 重新计算 `timeout`
update_timeout:
    assert(timeout > 0);

    real_timeout -= (loop->time - base);
    if (real_timeout <= 0)
        return;

    // 剩余 `timeout`
    timeout = real_timeout;

```

```
}  
}
```

在某些情况下，IO观察者绑定的回调函数并不是立即调用的，而是被延迟到下一次事件循环的固定阶段调用的，在 `uv_run` 中调用的 `uv__run_pending` 处理这些被延迟的IO观察者，实现如下：

<https://github.com/libuv/libuv/blob/view-v1.28.0/src/unix/core.c#L737>

```
static int uv__run_pending(uv_loop_t* loop) {  
    QUEUE* q;  
    QUEUE pq;  
    uv__io_t* w;  
  
    if (QUEUE_EMPTY(&loop->pending_queue))  
        return 0;  
  
    QUEUE_MOVE(&loop->pending_queue, &pq);  
  
    while (!QUEUE_EMPTY(&pq)) {  
        q = QUEUE_HEAD(&pq);  
        QUEUE_REMOVE(q);  
        QUEUE_INIT(q);  
        w = QUEUE_DATA(q, uv__io_t, pending_queue);  
        w->cb(loop, w, POLLOUT);  
    }  
  
    return 1;  
}
```

该函数遍历 `loop->pending_queue` 队列节点，取得IO观察者后调用 `cb`，并且指定 `events` 参数为固定值 `POLLOUT`（表示可写），因此可以猜测被插入到 `loop->pending_queue` 队列中的情形都是可写IO事件。该队列上

是用来保存被延迟到下次事件循环中处理的IO观察者。为什么需要延迟呢？

通过搜索可以找到只有 `uv__io_feed` 中存在向 `loop->pending_queue` 队列插入节点的代码，如下：

<https://github.com/libuv/libuv/blob/v1.28.0/src/unix/core.c#L892>

```
void uv__io_feed(uv_loop_t* loop, uv__io_t* w) {  
    if (QUEUE_EMPTY(&w->pending_queue))  
        QUEUE_INSERT_TAIL(&loop->pending_queue, &w->pending_queue);  
}
```

继续搜索 `uv__io_feed` 可找到多处调用，此处就不过多介绍了。

至此，整个I/O观察者工作原理已经分析完成了。

I/O观察者是 libuv 中I/O相关的基础抽象，实现了对I/O事件的监控，其他I/O相关的功能基于这个基础抽象，I/O观察者完成了基本的事件派发，事件处理中的I/O数据读写则由更高级的抽象 `handle` 和 `request` 完成。

源文件地址：<https://github.com/liuyanjie/knowledge/tree/master/node.js/libuv/5-libuv-io-watcher.md>