Tree: 808602be43 Masutangu.github.io / _posts / 2016-10-13-libuv-source-code.md

Find file Copy path

Masutangu update tags 808602b on Oct 18, 2018

1 contributor

664 lines (509 sloc) 25.1 KB

Raw Blame History

layout	date	title	tags
post	2016-10-13 15:56:27 +0800	Libuv 源码阅读	源码阅读

花了几天时间读了下 libuv 的源码，整理成这篇文章。第一节是读官方教程做的笔记，主要是供自己备忘用，读者可以跳过。第二节解读 libuv 的源码，重点在 libuv 队列的实现和如何用线程池实现异步文件 IO。

概念

handles 和 requests

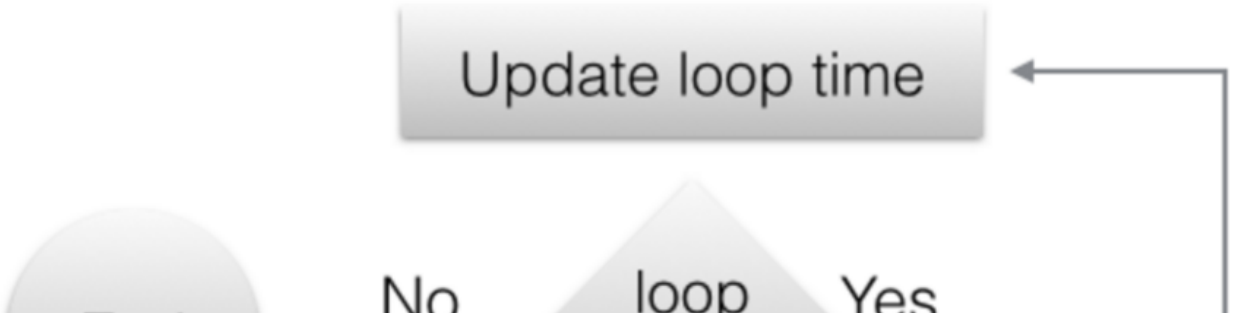
libuv 提供了两个抽象：handles 和 requests。handles 是 long－lived 的，会在其 active 的时候做特定的操作。requests 则为 short-lived 操作，request 可以独自执行或被 handle 调用执行。

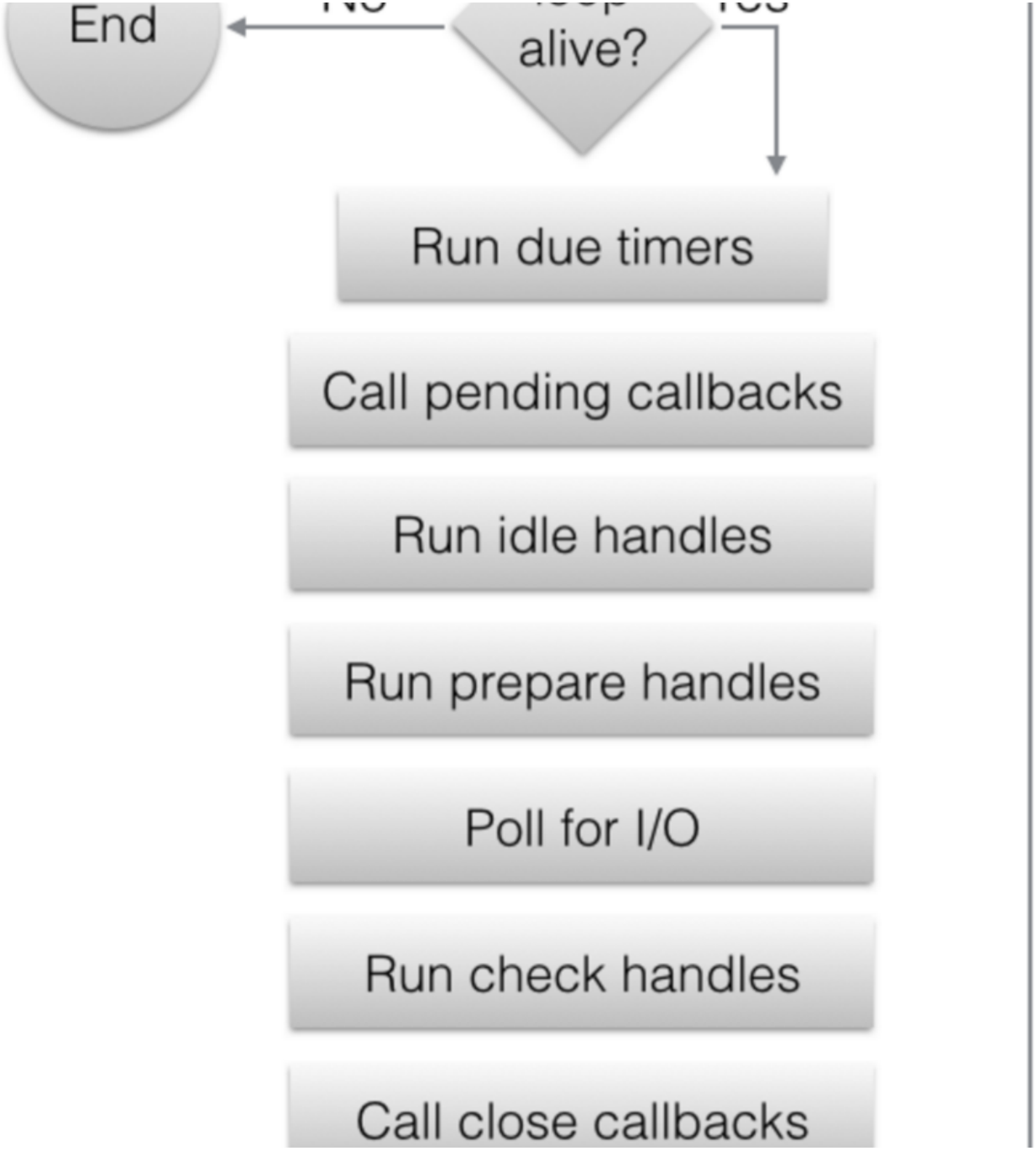
I/O loop

I/O loop（或 event loop）是 libuv 的核心。每个 I/O loop 绑定单一的线程。

> The libuv event loop (or any other API involving the loop or handles, for that matter) is not thread-safe except where stated otherwise.

event loop 采用单线程异步 IO 的形式：所有网络操作都使用 non-blocking 套接字，并使用各个平台上性能最好的 poll 机制例如 linux 上的 epoll，OSX 的 kqueue 等等。





I/O loop 的流程：

- event loop 在每次循环周期开始前都会缓存当前时间，以减少时间相关的系统调用
- 执行到期定时器的 callback
- 执行上一轮循环推迟的 I/O callback
- 执行 Idle handle 的 callback
- 执行 Prepare handle 的 callback
- 计算 poll timeout
- 阻塞处理 I/O，超时时间为上一步计算的 poll timeout
- 执行 Check handle 的 callback
- 执行 Close callback

> libuv uses a thread pool to make asynchronous file I/O operations possible, but network I/O is always performed in a single thread, each loop's thread.

File I/O

libuv 的异步文件 I/O 是通过线程池实现的。

> libuv provides a threadpool which can be used to run user code and get notified in the loop thread. **This thread pool is internally used to run all filesystem operations**, as well as getaddrinfo and getnameinfo requests.

> The threadpool is global and shared across all event loops. When a particular function makes use of the threadpool (i.e. when using `uv_queue_work()`) libuv preallocates and initializes the maximum number of threads allowed by UV_THREADPOOL_SIZE.

> libuv currently uses a global thread pool on which all loops can queue work on. 3 types of operations are currently run on this pool:

> - Filesystem operations
> - DNS functions (getaddrinfo and getnameinfo)
> - User specified code via uv_queue_work()

主要结构体

uv_loop_t

> The event loop is the central part of libuv's functionality. It takes care of polling for i/o and scheduling callbacks to be run based on different sources of events.

uv_loop_t 执行的三种模式

- **UV_RUN_DEFAULT**

  Runs the event loop until there are no more active and referenced handles or requests. Returns non-zero if uv_stop() was called and there are still active handles or requests. Returns zero in all other cases.

- **UV_RUN_ONCE**

  Poll for i/o once. Note that this function blocks if there are no pending callbacks. Returns zero when done (no active handles or requests left), or non-zero if more callbacks are expected (meaning you should run the event

loop again sometime in the future).

- **UV_RUN_NOWAIT**

  Poll for i/o once but don't block if there are no pending callbacks. Returns zero if done (no active handles or requests left), or non-zero if more callbacks are expected (meaning you should run the event loop again sometime in the future).

uv_handle_t

> uv_handle_t is the base type for all libuv handle types.

> Structures are aligned so that any libuv handle can be cast to uv_handle_t. All API functions defined here work with any handle type.

- `void uv_ref(uv_handle_t* handle)`

  Reference the given handle. References are idempotent, that is, if a handle is already referenced calling this function again will have no effect.

- `void uv_unref(uv_handle_t* handle)`

  Un-reference the given handle. References are idempotent, that is, if a handle is not referenced calling this function again will have no effect.

uv_unref 主要用于计时器。例子在此，摘抄如下：

> These functions can be used to allow a loop to exit even when a watcher is active or to use custom objects to keep the loop alive.

> The latter can be used with interval timers. You might have a garbage collector which runs every X seconds, or your network service might send a heartbeat to others periodically, but you don't want to have to stop them along all clean exit paths or error scenarios. Or you want the program to exit when all your other watchers are done. In that case just unref the timer immediately after creation so that if it is the only watcher running then uv_run will still exit.

```
uv_loop_t *loop;
uv_timer_t gc_req;
uv_timer_t fake_job_req;

int main() {
    loop = uv_default_loop();

    uv_timer_init(loop, &gc_req);
    uv_unref((uv_handle_t*) &gc_req);

    uv_timer_start(&gc_req, gc, 0, 2000);

    // could actually be a TCP download or something
    uv_timer_init(loop, &fake_job_req);
    uv_timer_start(&fake_job_req, fake_job, 9000, 0);
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

> We initialize the garbage collector timer, then immediately unref it. Observe how after 9 seconds, when the fake job is done, the program automatically exits, even though the garbage collector is still running.

uv_req_t

> uv_req_t is the base type for all libuv request types.

代码解读

队列

libuv 的队列是循环双向链表，队列在 libuv 中用到的地方很多，例如 event loop 用队列来存储 handle（handle_queue)，待监听的io事件（watcher_queue）等等。

定义

```
typedef void *QUEUE[2];
#define QUEUE_NEXT(q)       (*(QUEUE **) &((*(q))[0]))
#define QUEUE_PREV(q)       (*(QUEUE **) &((*(q))[1]))
#define QUEUE_PREV_NEXT(q)  (QUEUE_NEXT(QUEUE_PREV(q)))
#define QUEUE_NEXT_PREV(q)  (QUEUE_PREV(QUEUE_NEXT(q)))

#define QUEUE_INIT(q)                                                 \
  do {                                                                \
    QUEUE_NEXT(q) = (q);                                              \
    QUEUE_PREV(q) = (q);                                              \
  }                                                                   \
  while (0)
```

这段定义了 QUEUE 是元素类型为 void* 的数组，数组长度为 2。 如果按下面这样定义：

```
#define QUEUE_NEXT(q)       ((QUEUE *) ((*(q))[0]))
#define QUEUE_PREV(q)       ((QUEUE *) ((*(q))[1]))
```

返回值不是左值，在 QUEUE_INIT 函数中对 QUEUE_NEXT 和 QUEUE_PREV 的赋值会编译失败。 C/C++ 中类型转换有可能会返回左值（可以看 stackoverflow 的讲解)：

> The result of the expression (T) cast-expression is of type T. The result is an lvalue if T is an lvalue reference type or an rvalue reference to function type and an xvalue if T is an rvalue reference to object type; **otherwise the result is a prvalue.**[Note: if T is a non-class type that is cv-qualified, the cv-qualifiers are ignored when determining the type of the resulting prvalue; see 3.10. —end note]

因此需要先将 ((*(q))[0])) 取址再解引用（解引用返回左值）。

接口

下面是 QUEUE 几个重要的接口：

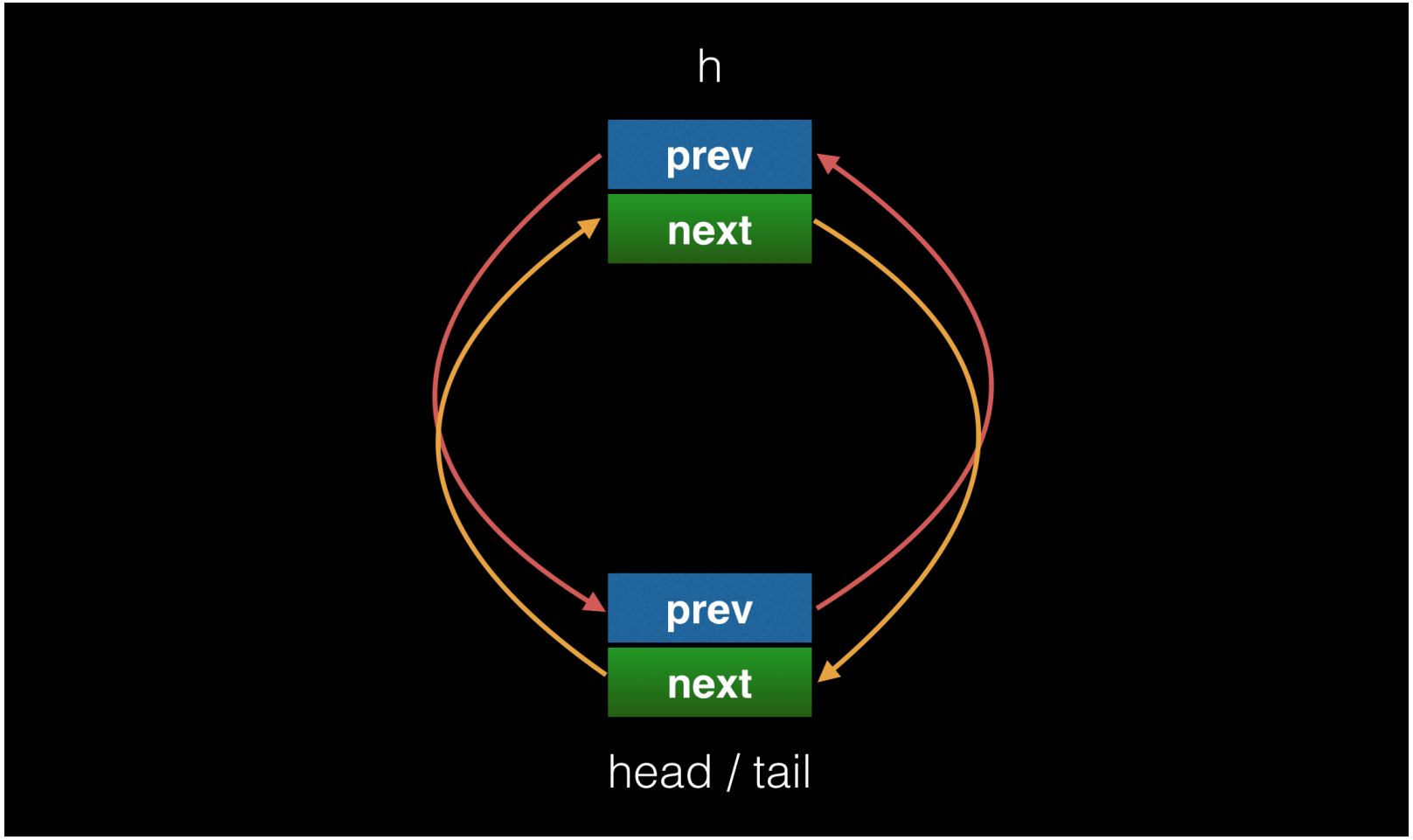
```
// 取出数据，具体例子可参考：https://gist.github.com/bodokaiser/5657156
#define QUEUE_DATA(ptr, type, field)                                          \
  ((type *) ((char *) (ptr) - offsetof(type, field)))

// 将 n 队列的元素添加到 h 队列，保留 h 队列原先的元素。注意操作后 n 队列的结构被破坏，不能在遍历 n 队列
#define QUEUE_ADD(h, n)                                                       \
  do {                                                                        \
    QUEUE_PREV_NEXT(h) = QUEUE_NEXT(n);                                       \
    QUEUE_NEXT_PREV(n) = QUEUE_PREV(h);                                       \
    QUEUE_PREV(h) = QUEUE_PREV(n);                                            \
    QUEUE_PREV_NEXT(h) = (h);                                                 \
  }                                                                           \
  while (0)

// QUEUE_MOVE 的 helper 函数
#define QUEUE_SPLIT(h, q, n)                                                  \
  do {                                                                        \
    QUEUE_PREV(n) = QUEUE_PREV(h);                                            \
```

```
    QUEUE_PREV_NEXT(n) = (n);                                   \
    QUEUE_NEXT(n) = (q);                                        \
    QUEUE_PREV(h) = QUEUE_PREV(q);                              \
    QUEUE_PREV_NEXT(h) = (h);                                   \
    QUEUE_PREV(q) = (n);                                        \
  }                                                             \
  while (0)

// 将 h 队列的元素添加到 n 队列， h 队列被清空，n 队列原先的元素也被清空
#define QUEUE_MOVE(h, n)                                        \
  do {                                                          \
    if (QUEUE_EMPTY(h))                                         \
      QUEUE_INIT(n);                                            \
    else {                                                      \
      QUEUE* q = QUEUE_HEAD(h);                                 \
      QUEUE_SPLIT(h, q, n);                                     \
    }                                                           \
  }                                                             \
  while (0)

// 添加元素 q 到 h 队列的尾部，QUEUE_PREV(h) 为原先队列的 tail 节点
#define QUEUE_INSERT_TAIL(h, q)                                 \
  do {                                                          \
    QUEUE_NEXT(q) = (h);                                        \
    QUEUE_PREV(q) = QUEUE_PREV(h);                              \
    QUEUE_PREV_NEXT(q) = (q);                                   \
    QUEUE_PREV(h) = (q);                                        \
  }                                                             \
  while (0)
```

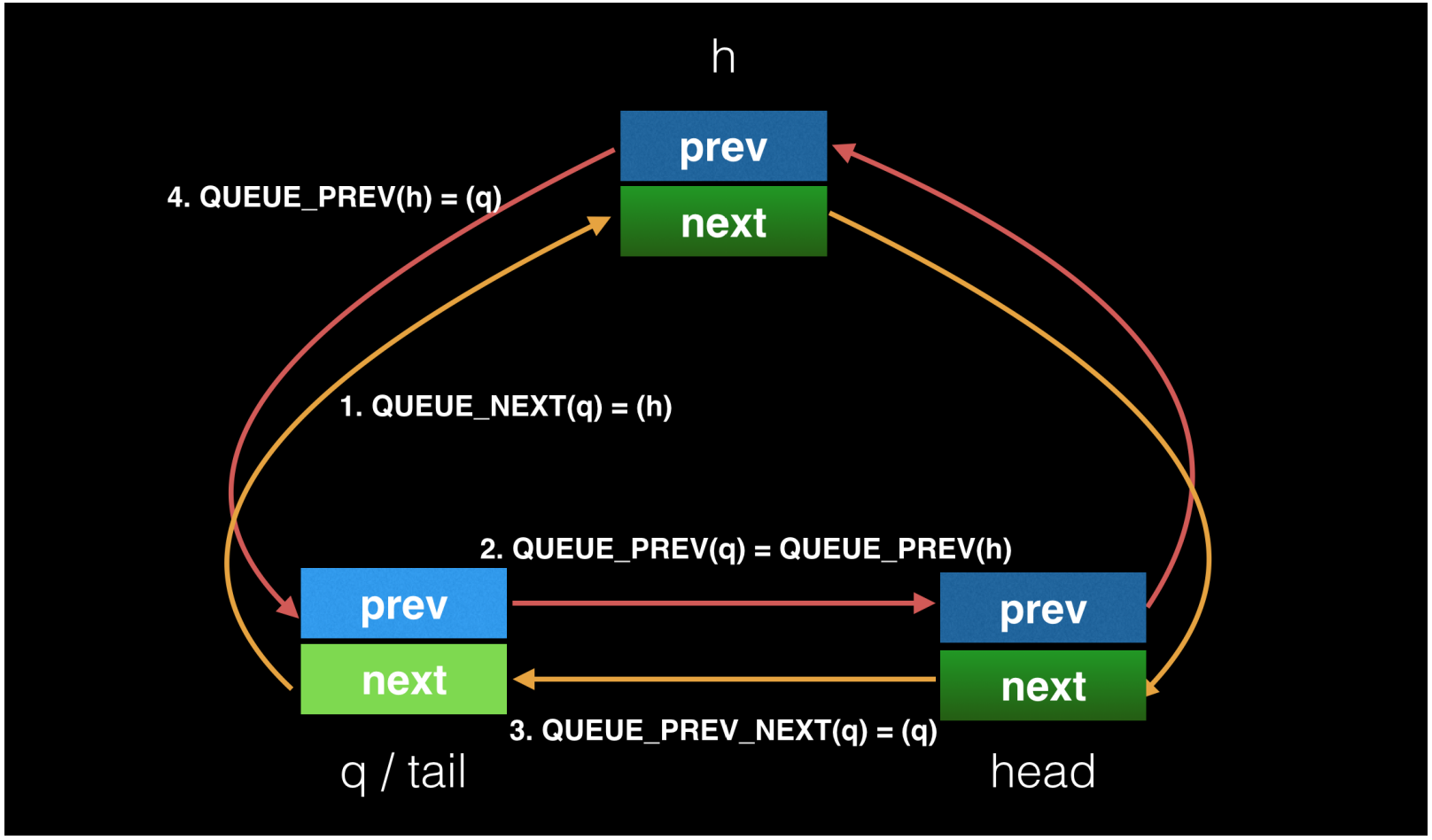
上面提到了 QUEUE 是一个循环链表。定义 h 为队列的哨兵节点，则 QUEUE_NEXT(h) 指向队列的 head 节点，QUEUE_PREV(h) 指向队列的 tail 节点。

结合图例来看看 QUEUE_INSERT_TAIL 的实现。

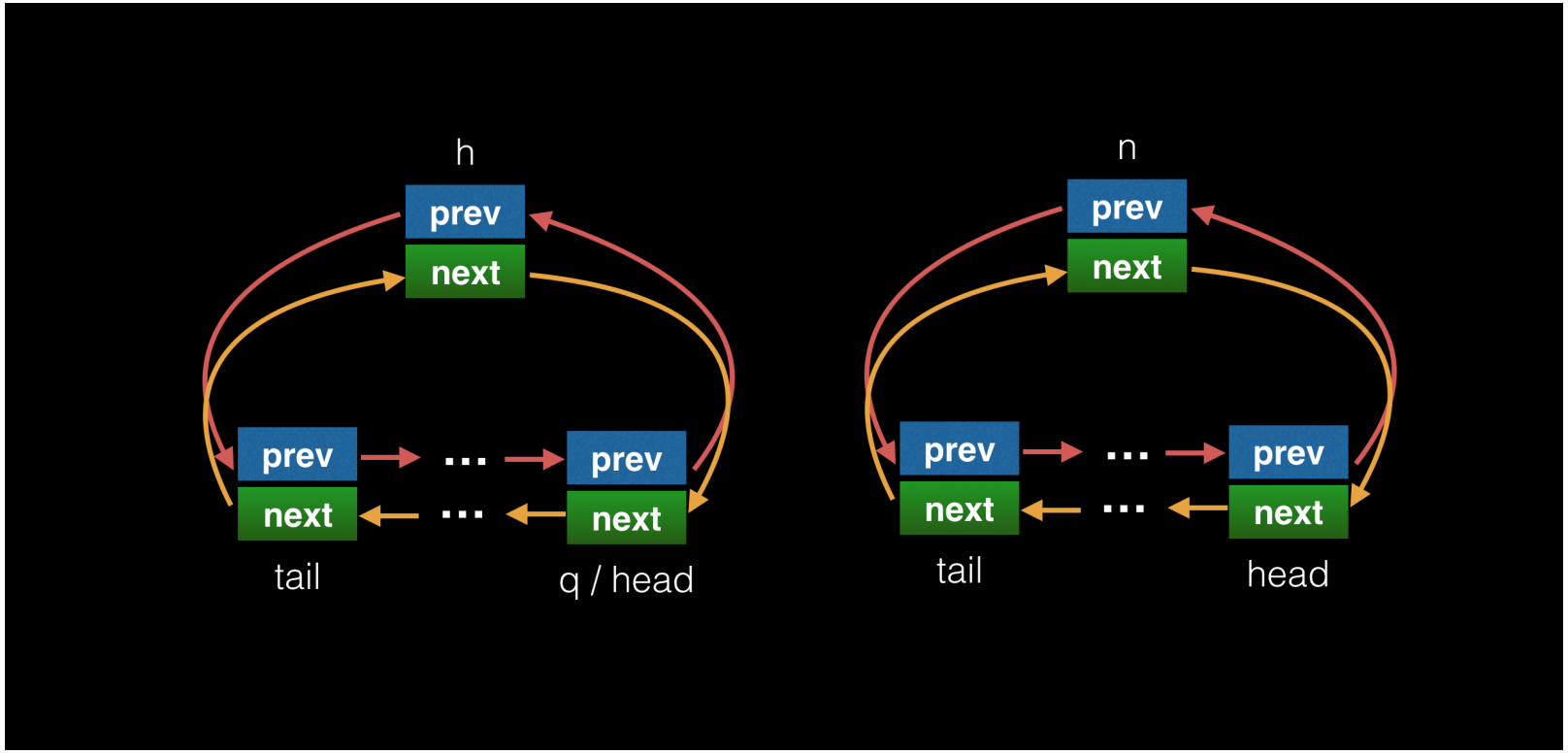
原队列，因为只有一个元素（h节点为哨兵节点，不算在内），因此其即是 head 也是 tail：



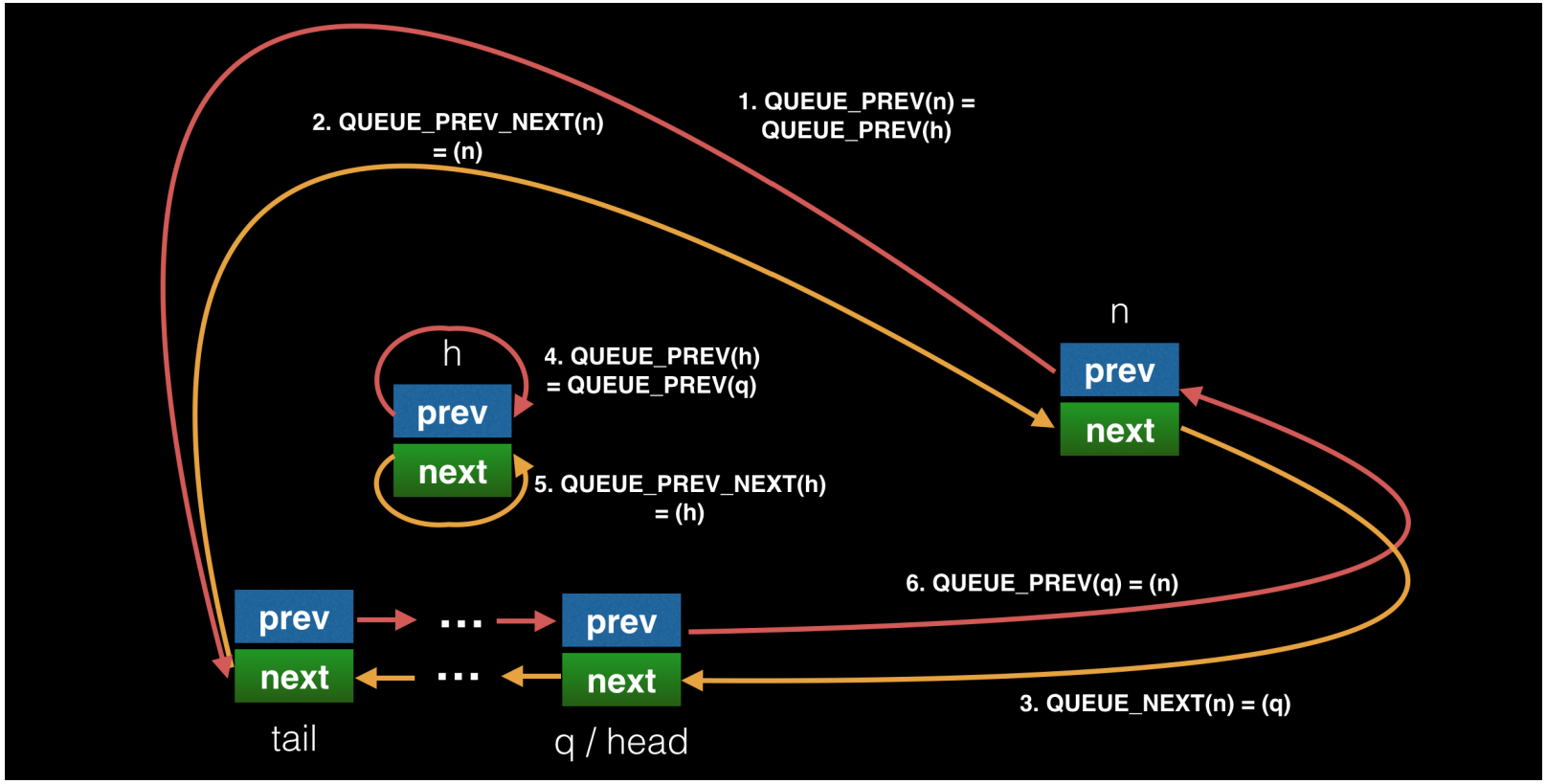
新增一个新元素后如下所示：



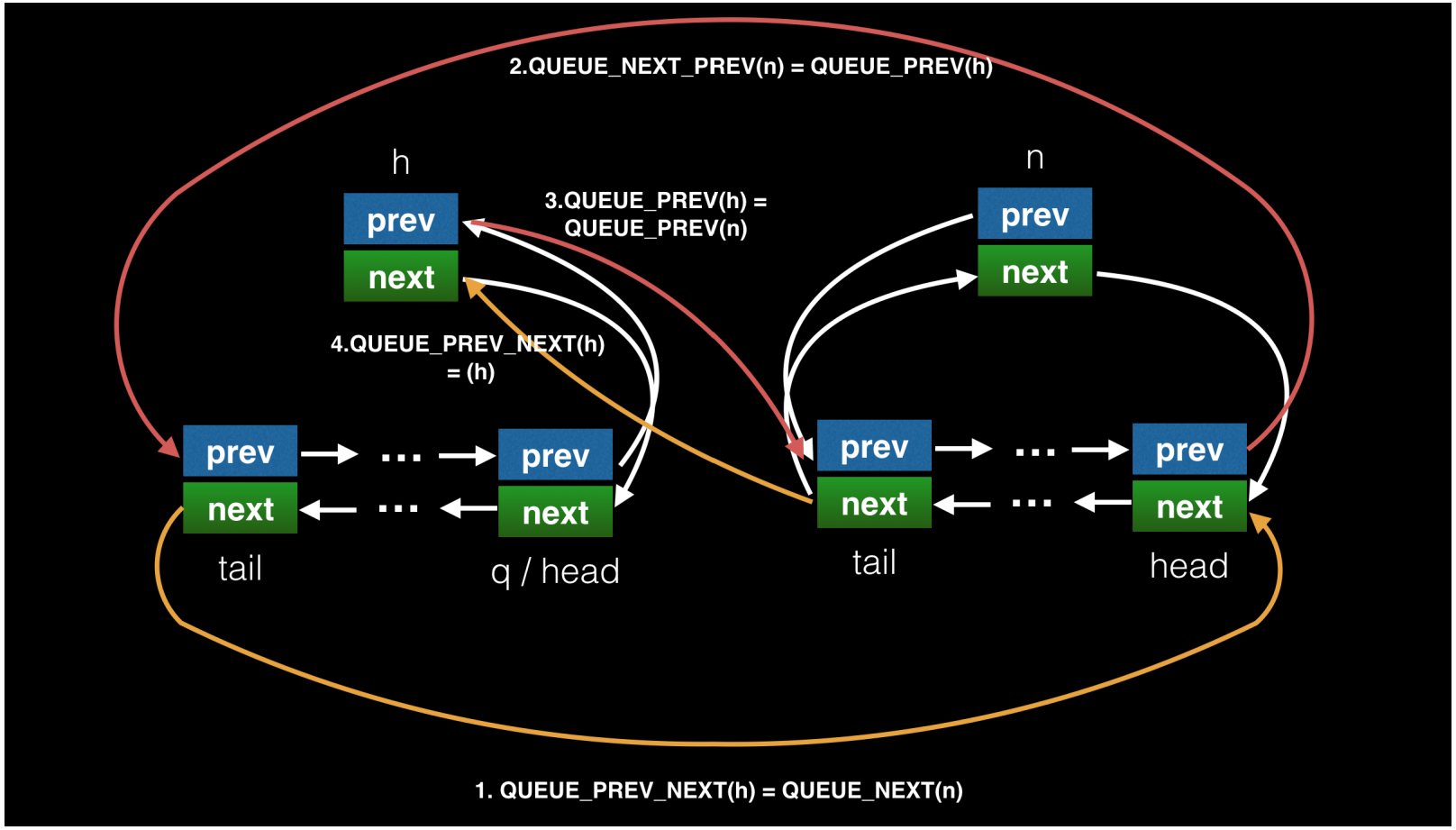
下面是 QUEUE_MOVE 的图例。 h 队列 和 n 队列 QUEUE_MOVE 操作前如下图：



QUEUE_MOVE 后，h 队列被清空，n 队列的哨兵节点连接到原 h 队列的 head 和 tail：



最后是 QUEUE_ADD 的图例，原理就是将 n 队列的 head 和 h 队列的 tail 相连，并把 h 队列的 tail 重置指向 n 队列的 tail：



epoll 事件管理

IO 事件都会调用 `uv__io_start` 函数，该函数将需要监听的事件保存到 event loop 的 watcher_queue 队列中：

```
void uv__io_start(uv_loop_t* loop, uv__io_t* w, unsigned int events) {
  ...

  if (QUEUE_EMPTY(&w->watcher_queue))
```

```
    QUEUE_INSERT_TAIL(&loop->watcher_queue, &w->watcher_queue);  // 添加到 loop->watcher_queue 队列

  ...
}
```

然后在 event loop 的循环中，会调用 `uv__io_poll`。该函数将 loop->watcher_queue 队列中的事件取出，添加到 epoll 进行监听：

```
void uv__io_poll(uv_loop_t* loop, int timeout) {
  ...
  uv__io_t* w;

  while (!QUEUE_EMPTY(&loop->watcher_queue)) {  // 遍历取出 loop->watcher_queue 队列中待监听的事件，直至
    q = QUEUE_HEAD(&loop->watcher_queue);
    QUEUE_REMOVE(q);
    QUEUE_INIT(q);

    w = QUEUE_DATA(q, uv__io_t, watcher_queue);  // 取出 uv__io_t 结构，该结构保存了用户注册的回调函数

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

    if (w->events == 0)
      op = UV__EPOLL_CTL_ADD;
    else
      op = UV__EPOLL_CTL_MOD;

    /* XXX Future optimization: do EPOLL_CTL_MOD lazily if we stop watching
     * events, skip the syscall and squelch the events after epoll_wait().
     */
    if (uv__epoll_ctl(loop->backend_fd, op, w->fd, &e)) {  // 添加到 epoll
      if (errno != EEXIST)
        abort();
```

```
    assert(op == UV__EPOLL_CTL_ADD);

    /* We've reactivated a file descriptor that's been watched before. */
    if (uv__epoll_ctl(loop->backend_fd, UV__EPOLL_CTL_MOD, w->fd, &e))
      abort();
  }

  w->events = w->pevents;
}

...


// 阻塞等待 epoll 返回直到超时
for (;;) {
  if (no_epoll_wait != 0 || (sigmask != 0 && no_epoll_pwait == 0)) {
    nfds = uv__epoll_pwait(loop->backend_fd,
                           events,
                           ARRAY_SIZE(events),
                           timeout,
                           sigmask);
    if (nfds == -1 && errno == ENOSYS)
      no_epoll_pwait = 1;
  } else {
    nfds = uv__epoll_wait(loop->backend_fd,
                          events,
                          ARRAY_SIZE(events),
                          timeout);
    if (nfds == -1 && errno == ENOSYS)
      no_epoll_wait = 1;
  }
  ...

  loop->watchers[loop->nwatchers] = (void*) events;  // 将 events 保存在 loop->watchers，为了在 uv_
```

```
    loop->watchers[loop->nwatchers + 1] = (void*) (uintptr_t) nfds;
    for (i = 0; i < nfds; i++) {
      pe = events + i;
      fd = pe->data;

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

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

      w = loop->watchers[fd];

      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.
         */
        uv__epoll_ctl(loop->backend_fd, UV__EPOLL_CTL_DEL, fd, pe);
        continue;
      }

      /* 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);

        if (pe->events != 0) {
          /* Run signal watchers last.  This also affects child process watchers
           * because those are implemented in terms of signal watchers.
           */
          if (w == &loop->signal_io_watcher)
            have_signals = 1;
          else
            w->cb(loop, w, pe->events);  // 调用用户注册的回调

          nevents++;
        }
      }

      loop->watchers[loop->nwatchers] = NULL;
      loop->watchers[loop->nwatchers + 1] = NULL;

      ...
    }
  }
```

线程池实现异步文件 IO

libuv 中文件操作的异步 IO 是通过线程池实现的。原理是将文件操作由工作线程来完成，当操作完成后工作线程通过 fd 通知主线程（该 fd 同样由 epoll 管理），主线程监听该 fd，当有 epoll 事件时根据层层回调，最终会调用到用户注册的回调函数。

下面看看这块逻辑，所有文件操作都调用了 POST 定义的宏。POST 判断是否注册了回调，如果有则表示该操作为异步调用，此时调用 `uv__work_submit` 向线程池提交任务。

```
#define POST                                                                  \
  do {                                                                        \
    if (cb != NULL) {                                                         \
      uv__work_submit(loop, &req->work_req, uv__fs_work, uv__fs_done);        \
      return 0;                                                               \
    }                                                                         \
    else {                                                                    \
      // 回调为 null 是同步调用                                                \
      uv__fs_work(&req->work_req);                                            \
      return req->result;                                                     \
    }                                                                         \
  }                                                                           \
  while (0)

// 操作完成后的回调函数
static void uv__fs_done(struct uv__work* w, int status) {
  uv_fs_t* req;

  req = container_of(w, uv_fs_t, work_req);
  uv__req_unregister(req->loop, req);

  if (status == -ECANCELED) {
    assert(req->result == 0);
    req->result = -ECANCELED;
```

```
  }

  req->cb(req);  // 调用用户注册的回调
}
```

`uv__work_submit` 先调用 `init_once` 初始化工作线程池，再调用 `post` 提交任务给工作线程：

```
void uv__work_submit(uv_loop_t* loop,
                     struct uv__work* w,
                     void (*work)(struct uv__work* w),
                     void (*done)(struct uv__work* w, int status)) {
  uv_once(&once, init_once);
  w->loop = loop;
  w->work = work;
  w->done = done;
  post(&w->wq);
}


static void init_once(void) {
  unsigned int i;
  const char* val;

  nthreads = ARRAY_SIZE(default_threads);
  val = getenv("UV_THREADPOOL_SIZE");
  if (val != NULL)
    nthreads = atoi(val);
  if (nthreads == 0)
    nthreads = 1;
  if (nthreads > MAX_THREADPOOL_SIZE)
    nthreads = MAX_THREADPOOL_SIZE;

  threads = default_threads;
  if (nthreads > ARRAY_SIZE(default_threads)) {
```

```
    threads = uv__malloc(nthreads * sizeof(threads[0]));
    if (threads == NULL) {
      nthreads = ARRAY_SIZE(default_threads);
      threads = default_threads;
    }
  }

  if (uv_cond_init(&cond))  // 初始化条件变量
    abort();

  if (uv_mutex_init(&mutex))  // 初始化互斥锁
    abort();

  QUEUE_INIT(&wq);

  for (i = 0; i < nthreads; i++)
    if (uv_thread_create(threads + i, worker, NULL))  // 创建工作线程
      abort();

  initialized = 1;
}

// 工作线程
static void worker(void* arg) {
  struct uv__work* w;
  QUEUE* q;

  (void) arg;

  for (;;) {
    uv_mutex_lock(&mutex);

    while (QUEUE_EMPTY(&wq)) {
      idle_threads += 1;
      uv_cond_wait(&cond, &mutex);   // wq 保存任务队列，当 wq 为空时阻塞等待任务，有新任务提交就会唤醒该 wo
```

```
      idle_threads -= 1;
    }

    q = QUEUE_HEAD(&wq);

    if (q == &exit_message)
      uv_cond_signal(&cond);
    else {
      QUEUE_REMOVE(q);
      QUEUE_INIT(q);  /* Signal uv_cancel() that the work req is
                         executing. */
    }

    uv_mutex_unlock(&mutex);

    if (q == &exit_message)
      break;

    w = QUEUE_DATA(q, struct uv__work, wq);
    w->work(w);   // work 执行文件操作

    uv_mutex_lock(&w->loop->wq_mutex);
    w->work = NULL;  /* Signal uv_cancel() that the work req is done
                        executing. */
    QUEUE_INSERT_TAIL(&w->loop->wq, &w->wq);  // 将已完成的 uv__work 添加到 loop->wq 队列
    uv_async_send(&w->loop->wq_async);  // 通知主线程该任务已经执行完成
    uv_mutex_unlock(&w->loop->wq_mutex);
  }
}

// 提交任务
static void post(QUEUE* q) {
  uv_mutex_lock(&mutex);
  QUEUE_INSERT_TAIL(&wq, q);  // 将任务提交到 wq 队列
  if (idle_threads > 0)
```

```
    uv_cond_signal(&cond);  // 有空闲工作线程时就唤醒 worker
  uv_mutex_unlock(&mutex);
}
```

最后看看工作线程和主线程的通信，在文件操作完成后，工作线程调用 `uv__async_send` ，该函数会往 wa->wfd 或 wa->io_watcher.fd 写一个空子节：

```
int uv_async_send(uv_async_t* handle) {
  /* Do a cheap read first. */
  if (ACCESS_ONCE(int, handle->pending) != 0)
    return 0;

  if (cmpxchgi(&handle->pending, 0, 1) == 0)
    uv__async_send(&handle->loop->async_watcher);

  return 0;
}

void uv__async_send(struct uv__async* wa) {
  const void* buf;
  ssize_t len;
  int fd;
  int r;

  buf = "";
  len = 1;
  fd = wa->wfd;

#if defined(__linux__)
  if (fd == -1) {
    static const uint64_t val = 1;
    buf = &val;
    len = sizeof(val);
```

```
    fd = wa->io_watcher.fd;  /* eventfd */
  }
#endif

  do
    r = write(fd, buf, len);
  while (r == -1 && errno == EINTR);

  if (r == len)
    return;

  if (r == -1)
    if (errno == EAGAIN || errno == EWOULDBLOCK)
      return;

  abort();
}
```

主线程监听 io_watcher.fd，当有 epoll 事件时回调的顺序如下：

调用 uv__io_t 的 cb 即 `uv__async_io` --> 调用 uv__async 的 cb 即 `uv__async_event` --> 调用 uv_async_t 的 cb 即 `uv__work_done` --> 调用 uv__work 的 done 即用户提交的回调函数。

```
int uv_loop_init(uv_loop_t* loop) {
  ...
  err = uv_async_init(loop, &loop->wq_async, uv__work_done);  // 初始化 async
  ...
}

void uv__work_done(uv_async_t* handle) {
  struct uv__work* w;
  uv_loop_t* loop;
  QUEUE* q;
```

```
  QUEUE wq;
  int err;

  loop = container_of(handle, uv_loop_t, wq_async);
  uv_mutex_lock(&loop->wq_mutex);
  QUEUE_MOVE(&loop->wq, &wq);  // 因为访问 loop->wq 需要锁，为了避免长时间锁，所以拷贝一份副本出来，下面的遍历
  uv_mutex_unlock(&loop->wq_mutex);

  while (!QUEUE_EMPTY(&wq)) {  // 遍历 loop->wq 的副本
    q = QUEUE_HEAD(&wq);
    QUEUE_REMOVE(q);

    w = container_of(q, struct uv__work, wq);
    err = (w->work == uv__cancelled) ? UV_ECANCELED : 0;
    w->done(w, err);  // 调用 done，即 uv__fs_done 函数，最终会调用用户注册的回调
  }
}

//  uv__async_start 函数会调用 uv__io_start，监听 wa->io_watcher.fd
int uv_async_init(uv_loop_t* loop, uv_async_t* handle, uv_async_cb async_cb) {
  int err;

  err = uv__async_start(loop, &loop->async_watcher, uv__async_event);
  if (err)
    return err;

  uv__handle_init(loop, (uv_handle_t*)handle, UV_ASYNC);
  handle->async_cb = async_cb;
  handle->pending = 0;

  QUEUE_INSERT_TAIL(&loop->async_handles, &handle->queue);
  uv__handle_start(handle);

  return 0;
}
```

```c
// 创建 wa->io_watcher.fd
int uv__async_start(uv_loop_t* loop, struct uv__async* wa, uv__async_cb cb) {
  int pipefd[2];
  int err;

  if (wa->io_watcher.fd != -1)
    return 0;

  // 下面一大段是创建 io_watcher.fd 的逻辑
  err = uv__async_eventfd();
  if (err >= 0) {
    pipefd[0] = err;
    pipefd[1] = -1;
  }
  else if (err == -ENOSYS) {
    err = uv__make_pipe(pipefd, UV__F_NONBLOCK);
#if defined(__linux__)
    /* Save a file descriptor by opening one of the pipe descriptors as
     * read/write through the procfs.  That file descriptor can then
     * function as both ends of the pipe.
     */
    if (err == 0) {
      char buf[32];
      int fd;

      snprintf(buf, sizeof(buf), "/proc/self/fd/%d", pipefd[0]);
      fd = uv__open_cloexec(buf, O_RDWR);
      if (fd >= 0) {
        uv__close(pipefd[0]);
        uv__close(pipefd[1]);
        pipefd[0] = fd;
        pipefd[1] = fd;
      }
    }
```

```
#endif
  }

  if (err < 0)
    return err;

  uv__io_init(&wa->io_watcher, uv__async_io, pipefd[0]);  // 注册 async io 事件的 callback 为 uv__asy
  uv__io_start(loop, &wa->io_watcher, POLLIN);  // 将该 io_watcher 添加到 loop->watcher_queue，参考上
  wa->wfd = pipefd[1];
  wa->cb = cb;  // 注册 uv__async 的 cb 为 uv__async_event

  return 0;
}
```