



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 / 6-libuv源码分析（七）异步唤醒（Async）.md

Find file

Copy path



liuyanjie add articles

2c06277 on Apr 24

1 contributor

246 lines (176 sloc) | 8.6 KB

Raw

Blame

History



title

date

updated

tags

categories

title	date	updated	tags			categories
libuv源码分析（七） 异步唤醒（Async）	2019-04-23 15:00:07 UTC	2019-04-24 01:31:28 UTC	libuv	node.js	eventloop	源码分析

Async 允许用户在其他线程中唤醒主事件循环线程并触发回调函数调用。

事件循环线程在运行到 Pool 阶段会因为 `epoll_pwait` 调用阻塞一定的时间，libuv 会根据事件循环信息预估阻塞多长时间合适，也就是 `timeout`。但是在某些情境下，libuv 是无法准确预估的，例如线程池支持的异步文件操作，这些其他线程中的任务是无法有效判断多久能够运行完成的，在 libuv 中，其他线程工作完成之后，执行结果需要交给主事件循环线程，而事件循环线程可能恰好阻塞在 `epoll_pwait` 上，这时为了能够让其他线程的执行结果能够快速得到处理，需要唤醒主事件循环线程，也就是 `epoll_pwait`，而 Async 正式用来至此唤醒主事件循环的机制，简单的调用 `uv_async_send` 即可。线程池中的线程也正是利用这个机制和主事件循环线程通讯。

通过前文中对IO观察者的分析，我们知道，让 `epoll_pwait` 返回的方式，就是让 `epoll_pwait` 轮询的文件描述符中有I/O事件发生，Async 就是这么做的，通过 `uv_async_send` 向某个固定的文件描述符发送数据，使 `epoll_pwait` 返回。

Async 的入口函数共用两个：

- `uv_async_init` 初始化 Async Handle
- `uv_async_send` 发送消息唤醒事件循环线程并触发回调函数调用

首先，看一下 Async Handle 结构 `uv_async_s` 的定义：

<https://github.com/libuv/libuv/blob/v1.28.0/include/uv.h#L789>

```
struct uv_async_s {  
    UV_HANDLE_FIELDS
```

```
UV_ASYNC_PRIVATE_FIELDS
};
```

```
#define UV_ASYNC_PRIVATE_FIELDS \
    uv_async_cb async_cb; \
    void* queue[2]; \
    int pending;
```

结构比较简单，`async_cb` 保存回调函数指针，`queue` 作为队列节点接入 `loop->async_handles`，`pending` 字段表示已发送了唤醒信号，初始化为 `0`，在调用唤醒函数之后会被设置为 `1`。

继续看 `uv_async_init`：

注意：该初始化函数不同于其他初始化函数，该函数会立即启动 `Handle`，所以没有 `Start`。

<https://github.com/libuv/libuv/blob/view-v1.28.0/src/unix/async.c#L40>

```
int uv_async_init(uv_loop_t* loop, uv_async_t* handle, uv_async_cb async_cb) {
    int err;

    err = uv__async_start(loop);
    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;
}
```

`uv__async_start` 初始化并启动了 `loop->async_io_watcher`，使事件循环能够通过 `loop->async_io_watcher` 接收到其他线程发送的唤醒消息。

在进行简单的初始化后，直接启动了 `handle`，并不需要像其他 `handle` 一样提供 `uv_async_start` 这样的方法。

我们继续看一下 `uv__async_start` 如何工作：

<https://github.com/libuv/libuv/blob/v1.28.0/src/unix/async.c#L156>

```
static int uv__async_start(uv_loop_t* loop) {
    int pipefd[2];
    int err;

    if (loop->async_io_watcher.fd != -1)
        return 0;

    err = uv__async_eventfd();
    if (err >= 0) {
        pipefd[0] = err;
        pipefd[1] = -1;
    }
    else if (err == UV_ENOSYS) {
        err = uv__make_pipe(pipefd, UV__F_NONBLOCK);
#ifdef __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(&loop->async_io_watcher, uv__async_io, pipefd[0]);
uv__io_start(loop, &loop->async_io_watcher, POLLIN);
loop->async_wfd = pipefd[1];

return 0;
}

```

函数中，初始化并启动了 `loop->async_io_watcher`，该函数中创建了管道，其本质是一个内核缓冲区（4k），有两个文件描述符引用，用于有血缘关系的进程和线程间进行数据传递（通信），`pipefd` 保存了管道的两端的文件描述符，`pipefd[0]` 用于读数据，`pipefd[1]` 用于写数据，`pipefd[1]` 被保存到了 `loop->async_wfd`，通过I/O观察者监听 `pipefd[0]` 即可接收消息，通过向 `loop->async_wfd` 写数据，即可发送消息。`uv__async_start` 在已经初始化 `loop->async_io_watcher` 的情况下，无需再次初始化。

需要注意的是，`uv_async_init` 可能调用多次用于初始化多个不同的 Async Handle，但是 `loop->async_io_watcher` 只有一个，也就是这些 Async Handle 共享了 `loop->async_io_watcher`，那么在 `loop->async_io_watcher` 上有I/O事件时，并不知道是哪个Async Handle发送的。

loop->async_io_watcher 上的I/O事件，由 uv__async_io 处理，它的实现如下：

<https://github.com/libuv/libuv/blob/view-v1.28.0/src/unix/async.c#L76>

```
static void uv__async_io(uv_loop_t* loop, uv__io_t* w, unsigned int events) {
    char buf[1024];
    ssize_t r;
    QUEUE queue;
    QUEUE* q;
    uv_async_t* h;

    assert(w == &loop->async_io_watcher);

    for (;;) {
        r = read(w->fd, buf, sizeof(buf));

        if (r == sizeof(buf))
            continue;

        if (r != -1)
            break;

        if (errno == EAGAIN || errno == EWOULDBLOCK)
            break;

        if (errno == EINTR)
            continue;

        abort();
    }

    QUEUE_MOVE(&loop->async_handles, &queue);
    while (!QUEUE_EMPTY(&queue)) {
        q = QUEUE_HEAD(&queue);
```

```

    h = QUEUE_DATA(q, uv_async_t, queue);

    QUEUE_REMOVE(q);
    QUEUE_INSERT_TAIL(&loop->async_handles, q);

    if (cmpxchg(&h->pending, 1, 0) == 0)
        continue;

    if (h->async_cb == NULL)
        continue;

    h->async_cb(h);
}
}

```

逻辑如下：

1. 不断的读取 `w->fd` 上的数据到 `buf` 中直到为空，`buf` 中的数据无实际用途；
2. 遍历 `loop->async_handles` 队列，调用所有 `h->pending` 值为 1 的 `handle` 的 `async_cb` 函数如果存在的话。

`h->pending` 是在 `uv_async_send` 中被设置为 1。因为 `h->pending` 会在多线程中被访问到，所以存在资源争抢的临界状态，`cmpxchg` 是原子操作，在这段代码中，如果 `h->pending == 1` 会被原子的修改成 0，其他线程中对 `h->pending` 的读写也通过 `cmpxchg` 进行原子操作，防止同时读写程序异常。

如上文所述，`uv_async_io` 并不知道是哪个 Async Handle 上调用的，`uv_async_io` 实际上调用了所有的 `h->pending` 值为 1 也就是发送过唤醒信号的 `handle`。实际上，Async 的设计的目的是能够唤醒主事件循环线程，所以 libuv 并不需要关心是哪个 Async Handle 发送的信号，有可能同时发送。

接下来 我们了解一下 如何唤醒事件循环，简单的调用 `uv_async_send` 即可：

<https://github.com/libuv/libuv/blob/v1.28.0/src/unix/async.c#L58>

```

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

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

    return 0;
}

```

```

static void uv__async_send(uv_loop_t* loop) {
    const void* buf;
    ssize_t len;
    int fd;
    int r;

    buf = "";
    len = 1;
    fd = loop->async_wfd;

#ifdef __linux__
    if (fd == -1) {
        static const uint64_t val = 1;
        buf = &val;
        len = sizeof(val);
        fd = loop->async_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();
}
```

`uv_async_send` 可能在多个线程中同时调用，而且有可能在同一个 Async Handle 上调用，所以要求对 `handle->pending` 进行原子性读写。

`uv__async_send` 为实际进行写操作，因为管道中存在缓存区，所以需要不断的向 `loop->async_wfd` 写入数据，直到阻塞为止。

以上，就是 Async 唤醒事件循环线程的实现方式，很简单，核心在于竞态问题的解决。

源文件地址：<https://github.com/liuyanjie/knowledge/tree/master/node.js/libuv/7-libuv-async.md>