Tree: 62e84b27e7 liuyanjie.github.io / source / _posts / 5-libuv源码分析（六）流（Stream）.md

Find file Copy path

liuyanjie add articles 2c06277 on Apr 24

1 contributor

530 lines (413 sloc) 17.9 KB

Raw Blame History

title	date	updated	tags	categories

title	date	updated	tags	categories
libuv源码分析（六）流（Stream）	2019-04-23 15:00:06 UTC	2019-04-24 01:31:28 UTC	libuv, node.js, eventloop	源码分析

Stream 提供一个全双工的通信信道的抽象，`uv_stream_t` 是一个抽象数据类型，libuv 提供了 `uv_tcp_t`、`uv_pipe_t`、`uv_tty_t` `3` 个 `Stream` 实现。

uv_stream_t

`uv_stream_t` 并未直接提供初始化函数，如同 `uv_handle_t` 一样，`uv_stream_t` 是在派生类型初始化的时候间接初始化的。派生类型的初始化函数中都调用了 `uv__stream_init` 函数对 `uv_stream_t` 进行初始化。

先介绍一下 `uv_stream_t` 的各个字段的含义

https://github.com/libuv/libuv/blob/view-v1.28.0/include/uv.h#L470

```
/*
 * uv_stream_t is a subclass of uv_handle_t.
 *
 * uv_stream is an abstract class.
 *
 * uv_stream_t is the parent class of uv_tcp_t, uv_pipe_t and uv_tty_t.
 */
struct uv_stream_s {
  UV_HANDLE_FIELDS
  UV_STREAM_FIELDS
};
```

https://github.com/libuv/libuv/blob/view-v1.28.0/include/uv.h#L462

```
#define UV_STREAM_FIELDS                          \
  /* number of bytes queued for writing */        \ 共有字段：
  size_t write_queue_size;                        \   等待写的字节数
  uv_alloc_cb alloc_cb;                           \   用于分配空间的函数指针
  uv_read_cb read_cb;                             \   读取数据完成之后的回调函数
  /* private */                                   \
  UV_STREAM_PRIVATE_FIELDS
```

https://github.com/libuv/libuv/blob/view-v1.28.0/include/uv/unix.h#L283

```
#define UV_STREAM_PRIVATE_FIELDS                   \ 私有字段：
  uv_connect_t *connect_req;                       \   连接请求
  uv_shutdown_t *shutdown_req;                     \   关闭请求
  uv__io_t io_watcher;                             \   I/O观察者（has-a）
  void* write_queue[2];                            \   写数据队列
  void* write_completed_queue[2];                  \   完成的写数据队列
  uv_connection_cb connection_cb;                  \   有新连接时的回调函数
  int delayed_error;                               \   延迟的错误
  int accepted_fd;                                 \   对端的fd
  void* queued_fds;                                \   排队的文件描述符列表
  UV_STREAM_PRIVATE_PLATFORM_FIELDS                \
```

Init

https://github.com/libuv/libuv/blob/v1.x/src/unix/stream.c#L84

```
void uv__stream_init(uv_loop_t* loop,
                     uv_stream_t* stream,
```

```
                    uv_handle_type type) {
  int err;

  uv__handle_init(loop, (uv_handle_t*)stream, type);
  stream->read_cb = NULL;
  stream->alloc_cb = NULL;
  stream->close_cb = NULL;
  stream->connection_cb = NULL;
  stream->connect_req = NULL;
  stream->shutdown_req = NULL;
  stream->accepted_fd = -1;
  stream->queued_fds = NULL;
  stream->delayed_error = 0;
  QUEUE_INIT(&stream->write_queue);
  QUEUE_INIT(&stream->write_completed_queue);
  stream->write_queue_size = 0;

  if (loop->emfile_fd == -1) {
    err = uv__open_cloexec("/dev/null", O_RDONLY);
    if (err < 0)
        /* In the rare case that "/dev/null" isn't mounted open "/"
         * instead.
         */
        err = uv__open_cloexec("/", O_RDONLY);
    if (err >= 0)
      loop->emfile_fd = err;
  }

#if defined(__APPLE__)
  stream->select = NULL;
#endif /* defined(__APPLE_) */

  uv__io_init(&stream->io_watcher, uv__stream_io, -1);
}
```

`uv__stream_init` 的整体工作逻辑如下：

1. 首先调用基类（`uv_handle_t`）初始化函数 `uv__handle_init` 对基类进行初始化；
2. 对 `stream` 结构进行初始化；
    i. 初始化相关字段；
    ii. 初始化 `stream->write_queue` 写队列；
    iii. 初始化 `stream->write_completed_queue` 写完成队列；为什么有两个写相关的队列？写操作为了实现异步非阻塞，上层的写操作并不能直接写，而是丢到队列中，当下层I/O观察者触发可写事件时，在进行写入操作。
3. 最后调用I/O观察者初始化函数 `uv__io_init` 对 `stream->io_watcher` 进行初始化，初始化传递了异步回调函数 `uv__stream_io`。

`uv__stream_init` 在 `uv_stream_t` 的派生类型的初始化函数 `uv_tcp_init`、`uv_pipe_init`、`uv_tty_init` 中被调用。

接下来看看 `uv__stream_io` 都做了什么

uv__stream_io

`uv__stream_io` 是 `uv_stream_t` I/O事件的处理函数，实现如下：

https://github.com/libuv/libuv/blob/v1.28.0/src/unix/stream.c#L1281

```
static void uv__stream_io(uv_loop_t* loop, uv__io_t* w, unsigned int events) {
  uv_stream_t* stream;

  // 取得原 stream 实例
  stream = container_of(w, uv_stream_t, io_watcher);

  // 断言
```

```
  assert(stream->type == UV_TCP ||
         stream->type == UV_NAMED_PIPE ||
         stream->type == UV_TTY);
  assert(!(stream->flags & UV_HANDLE_CLOSING));

  // 如果 stream 上存在 连接请求，则首选需要建立连接
  if (stream->connect_req) {
    uv__stream_connect(stream);
    return;
  }

  // 断言存在文件描述符
  assert(uv__stream_fd(stream) >= 0);

  // 满足读数据条件，进行数据读取，读取成功后继续向下执行，读取需要多久？
  /* Ignore POLLHUP here. Even if it's set, there may still be data to read. */
  if (events & (POLLIN | POLLERR | POLLHUP))
    uv__read(stream);

  // read_cb 可能会关闭 stream
  if (uv__stream_fd(stream) == -1)
    return;  /* read_cb closed stream. */

  /* Short-circuit iff POLLHUP is set, the user is still interested in read
   * events and uv__read() reported a partial read but not EOF. If the EOF
   * flag is set, uv__read() called read_cb with err=UV_EOF and we don't
   * have to do anything. If the partial read flag is not set, we can't
   * report the EOF yet because there is still data to read.
   */
  if ((events & POLLHUP) &&
      (stream->flags & UV_HANDLE_READING) &&
      (stream->flags & UV_HANDLE_READ_PARTIAL) &&
      !(stream->flags & UV_HANDLE_READ_EOF)) {
    uv_buf_t buf = { NULL, 0 };
    uv__stream_eof(stream, &buf);
```

```
  }

  // read_cb 可能会关闭 stream
  if (uv__stream_fd(stream) == -1)
    return;  /* read_cb closed stream. */

  // 满足写数据条件，进行数据写入，写入成功后继续向下执行，读取需要多久？
  if (events & (POLLOUT | POLLERR | POLLHUP)) {
    uv__write(stream);
    uv__write_callbacks(stream);

    /* Write queue drained. */
    if (QUEUE_EMPTY(&stream->write_queue))
      uv__drain(stream);
  }
}
```

在调用 `uv__stream_io` 时，传递了事件循环对象、I/O观察者对象、事件类型等信息。

执行逻辑如下：

1. 首先，通过 `container_of` 将I/O观察者对象地址换算成 `stream` 对象地址，再进行强制类型转换，进而还原出 `stream` 类型；
2. 验证 `stream` 类型已经状态是否正常；
3. 如果 `stream->connect_req` 存在，说明 该 `stream` 需要 进行 `connect`，于是调用 `uv__stream_connect`；
4. 如果 满足 可读条件 调用 `uv__read` 进行数据读操作，读的数据来源于对应的文件描述符，内部调用 `stream->alloc_cb` 分配 `uv_buf_t` 进行数据存储空间分配，然后进行数据读取，读取完成后调用读完成回调 `stream->read_cb`；
5. 如果 满足 流结束条件 调用 `uv__stream_eof` 进行相关处理；
6. 如果 满足 可写条件 调用 `uv__write` 进行数据写操作，写数据需要事先准备好，这些数据被放到了 `stream->write_queue` 写队列上，当底层描述符可写时，将队列上的数据写入。

后续，继续分析 `uv__read` `uv__write` `uv__stream_eof` 的相关实现逻辑，因为不影响大的逻辑，所以暂时可以先留空。

uv__read

https://github.com/libuv/libuv/blob/view-v1.28.0/src/unix/stream.c#L1110

当I/O观察者存在可读事件时，函数 `uv__read` 会被调用，当 `uv__read` 调用时，会通过 `read` 从底层文件描述符读取数据，读取的数据写到由 `stream->alloc_cb` 分配到内存中，并在完成读取后由 `stream->read_cb` 回调给用户层代码。因为可读数据已经由底层准备好，所以读取速度是非常快的，不需要等待。

默认情况下，当底层没有数据的情况时，`read` 系统调用会阻塞，但是此处因为文件描述符工作在非阻塞模式下，所有即使没有数据，`read` 也会立即返回。所以事件循环不好因为 `uv__read` 调用而耗时过长。

uv__write

https://github.com/libuv/libuv/blob/view-v1.28.0/src/unix/stream.c#L801

当I/O观察者存在可写事件时，函数 `uv__write` 会被调用，当 `uv__write` 调用时，数据已经在 `stream->write_queue` 队列上排好了，这个队列是 `uv_write_t` 类型的数据，如果队列为空没有数据可以写。用户在进行 `uv_write()` API 调用时，因为是异步操作，所以数据并不会直接执行真正的写操作，而是丢到写请求队列中后直接返回了，待到 `stream` 处于可写状态，事件处理含数 `uv__stream_io` 被调用，开始调用系统API进行真正的数据写入。

默认情况下，当底层没有更多内存缓冲区可用时，`write` 系统调用会阻塞，但是此处因为文件描述符工作在非阻塞模式下，所有即使缓冲区用完，`write` 也会立即返回。所以事件循环不好因为 `uv__write` 调用而耗时过长。

uv__write_callbacks

清理 `stream->write_completed_queue` 已完成写请求的队列，清理空间，并调用回调函数。

https://github.com/libuv/libuv/blob/view-v1.28.0/src/unix/stream.c#L926

```
static void uv__write_callbacks(uv_stream_t* stream) {
  uv_write_t* req;
  QUEUE* q;
  QUEUE pq;

  if (QUEUE_EMPTY(&stream->write_completed_queue))
    return;

  QUEUE_MOVE(&stream->write_completed_queue, &pq);

  while (!QUEUE_EMPTY(&pq)) {
    /* Pop a req off write_completed_queue. */
    q = QUEUE_HEAD(&pq);
    req = QUEUE_DATA(q, uv_write_t, queue);
    QUEUE_REMOVE(q);
    uv__req_unregister(stream->loop, req);

    if (req->bufs != NULL) {
      stream->write_queue_size -= uv__write_req_size(req);
      if (req->bufs != req->bufsml)
        uv__free(req->bufs);
      req->bufs = NULL;
    }

    /* NOTE: call callback AFTER freeing the request data. */
    if (req->cb)
      req->cb(req, req->error);
  }
}
```

uv__drain

```
static void uv__drain(uv_stream_t* stream) {
  uv_shutdown_t* req;
  int err;

  assert(QUEUE_EMPTY(&stream->write_queue));
  uv__io_stop(stream->loop, &stream->io_watcher, POLLOUT);
  uv__stream_osx_interrupt_select(stream);

  /* Shutdown? */
  if ((stream->flags & UV_HANDLE_SHUTTING) &&
      !(stream->flags & UV_HANDLE_CLOSING) &&
      !(stream->flags & UV_HANDLE_SHUT)) {
    assert(stream->shutdown_req);

    req = stream->shutdown_req;
    stream->shutdown_req = NULL;
    stream->flags &= ~UV_HANDLE_SHUTTING;
    uv__req_unregister(stream->loop, req);

    err = 0;
    if (shutdown(uv__stream_fd(stream), SHUT_WR))
      err = UV__ERR(errno);

    if (err == 0)
      stream->flags |= UV_HANDLE_SHUT;

    if (req->cb != NULL)
      req->cb(req, err);
  }
}
```

Read

不同于其他类型的 `handle`，提供了 `uv_timer_start` 等方法，Stream 的 Start 在命名上略有不同，对 Stream 来说，有 uv_read_start 和 uv_write 以及其他的 Start 方式。

Start：`uv_read_start`

https://github.com/libuv/libuv/blob/view-v1.28.0/src/win/stream.c#L67

```
int uv_read_start(uv_stream_t* stream,
                  uv_alloc_cb alloc_cb,
                  uv_read_cb read_cb) {
  assert(stream->type == UV_TCP || stream->type == UV_NAMED_PIPE ||
      stream->type == UV_TTY);

  if (stream->flags & UV_HANDLE_CLOSING)
    return UV_EINVAL;

  if (!(stream->flags & UV_HANDLE_READABLE))
    return -ENOTCONN;

  /* The UV_HANDLE_READING flag is irrelevant of the state of the tcp - it just
   * expresses the desired state of the user.
   */
  stream->flags |= UV_HANDLE_READING;

  /* TODO: try to do the read inline? */
  /* TODO: keep track of tcp state. If we've gotten a EOF then we should
   * not start the IO watcher.
   */
  assert(uv__stream_fd(stream) >= 0);
  assert(alloc_cb);

  stream->read_cb = read_cb;
  stream->alloc_cb = alloc_cb;
```

```
  uv__io_start(stream->loop, &stream->io_watcher, POLLIN);
  uv__handle_start(stream);
  uv__stream_osx_interrupt_select(stream);

  return 0;
}
```

uv_read_start 有三个参数：

1. stream，数据源；
2. alloc_cb，读取数据时调用该函数分配内存空间；
3. read_cb，读取成功后触发异步回调。

可以看到，启动过程同样没做什么特别的事情，将I/O观察者加入到队列中后，以便在事件循环的特定阶段进行处理。

Stop：`uv_read_stop`

https://github.com/libuv/libuv/blob/view-v1.28.0/src/unix/stream.c#L1584

```
int uv_read_stop(uv_stream_t* stream) {
  if (!(stream->flags & UV_HANDLE_READING))
    return 0;

  stream->flags &= ~UV_HANDLE_READING;
  uv__io_stop(stream->loop, &stream->io_watcher, POLLIN);
  if (!uv__io_active(&stream->io_watcher, POLLOUT))
    uv__handle_stop(stream);
  uv__stream_osx_interrupt_select(stream);

  stream->read_cb = NULL;
  stream->alloc_cb = NULL;
```

```
  return 0;
}
```

Write

https://github.com/libuv/libuv/blob/view-v1.28.0/src/unix/stream.c#L1483

```
/* The buffers to be written must remain valid until the callback is called.
 * This is not required for the uv_buf_t array.
 */
int uv_write(uv_write_t* req,
             uv_stream_t* handle,
             const uv_buf_t bufs[],
             unsigned int nbufs,
             uv_write_cb cb) {
  return uv_write2(req, handle, bufs, nbufs, NULL, cb);
}
```

https://github.com/libuv/libuv/blob/view-v1.28.0/src/unix/stream.c#L1387

```
int uv_write2(uv_write_t* req,
              uv_stream_t* stream,
              const uv_buf_t bufs[],
              unsigned int nbufs,
              uv_stream_t* send_handle,
              uv_write_cb cb) {
  int empty_queue;

  assert(nbufs > 0);
  assert((stream->type == UV_TCP ||
          stream->type == UV_NAMED_PIPE ||
```

```
           stream->type == UV_TTY) &&
         "uv_write (unix) does not yet support other types of streams");

  if (uv__stream_fd(stream) < 0)
    return UV_EBADF;

  if (!(stream->flags & UV_HANDLE_WRITABLE))
    return -EPIPE;

  if (send_handle) {
    if (stream->type != UV_NAMED_PIPE || !((uv_pipe_t*)stream)->ipc)
      return UV_EINVAL;

    /* XXX We abuse uv_write2() to send over UDP handles to child processes.
     * Don't call uv__stream_fd() on those handles, it's a macro that on OS X
     * evaluates to a function that operates on a uv_stream_t with a couple of
     * OS X specific fields. On other Unices it does (handle)->io_watcher.fd,
     * which works but only by accident.
     */
    if (uv__handle_fd((uv_handle_t*) send_handle) < 0)
      return UV_EBADF;

#if defined(__CYGWIN__) || defined(__MSYS__)
    /* Cygwin recvmsg always sets msg_controllen to zero, so we cannot send it.
       See https://github.com/mirror/newlib-cygwin/blob/86fc4bf0/winsup/cygwin/fhandler_socket.cc#l
    return UV_ENOSYS;
#endif
  }

  /* It's legal for write_queue_size > 0 even when the write_queue is empty;
   * it means there are error-state requests in the write_completed_queue that
   * will touch up write_queue_size later, see also uv__write_req_finish().
   * We could check that write_queue is empty instead but that implies making
   * a write() syscall when we know that the handle is in error mode.
   */
```

```
  empty_queue = (stream->write_queue_size == 0);

  /* Initialize the req */
  uv__req_init(stream->loop, req, UV_WRITE);
  req->cb = cb;
  req->handle = stream;
  req->error = 0;
  req->send_handle = send_handle;
  QUEUE_INIT(&req->queue);

  req->bufs = req->bufsml;
  if (nbufs > ARRAY_SIZE(req->bufsml))
    req->bufs = uv__malloc(nbufs * sizeof(bufs[0]));

  if (req->bufs == NULL)
    return UV_ENOMEM;

  memcpy(req->bufs, bufs, nbufs * sizeof(bufs[0]));
  req->nbufs = nbufs;
  req->write_index = 0;
  stream->write_queue_size += uv__count_bufs(bufs, nbufs);

  /* Append the request to write_queue. */
  QUEUE_INSERT_TAIL(&stream->write_queue, &req->queue);

  /* If the queue was empty when this function began, we should attempt to
   * do the write immediately. Otherwise start the write_watcher and wait
   * for the fd to become writable.
   */
  if (stream->connect_req) {
    /* Still connecting, do nothing. */
  }
  else if (empty_queue) {
    uv__write(stream);
  }
```

```
  else {
    /*
     * blocking streams should never have anything in the queue.
     * if this assert fires then somehow the blocking stream isn't being
     * sufficiently flushed in uv__write.
     */
    assert(!(stream->flags & UV_HANDLE_BLOCKING_WRITES));
    uv__io_start(stream->loop, &stream->io_watcher, POLLOUT);
    uv__stream_osx_interrupt_select(stream);
  }

  return 0;
}
```

https://github.com/libuv/libuv/blob/view-v1.28.0/src/unix/stream.c#L1501

```
int uv_try_write(uv_stream_t* stream,
                 const uv_buf_t bufs[],
                 unsigned int nbufs) {
  int r;
  int has_pollout;
  size_t written;
  size_t req_size;
  uv_write_t req;

  /* Connecting or already writing some data */
  if (stream->connect_req != NULL || stream->write_queue_size != 0)
    return UV_EAGAIN;

  has_pollout = uv__io_active(&stream->io_watcher, POLLOUT);

  r = uv_write(&req, stream, bufs, nbufs, uv_try_write_cb);
  if (r != 0)
```

```
    return r;

  /* Remove not written bytes from write queue size */
  written = uv__count_bufs(bufs, nbufs);
  if (req.bufs != NULL)
    req_size = uv__write_req_size(&req);
  else
    req_size = 0;
  written -= req_size;
  stream->write_queue_size -= req_size;

  /* Unqueue request, regardless of immediateness */
  QUEUE_REMOVE(&req.queue);
  uv__req_unregister(stream->loop, &req);
  if (req.bufs != req.bufsml)
    uv__free(req.bufs);
  req.bufs = NULL;

  /* Do not poll for writable, if we wasn't before calling this */
  if (!has_pollout) {
    uv__io_stop(stream->loop, &stream->io_watcher, POLLOUT);
    uv__stream_osx_interrupt_select(stream);
  }

  if (written == 0 && req_size != 0)
    return UV_EAGAIN;
  else
    return written;
}
```

源文件地址：https://github.com/liuyanjie/knowledge/tree/master/node.js/libuv/6-libuv-stream.md