Libuv 源码分析 v0.0.1

微信：theratliter

公众号：编程杂技

2020.05.30 于深圳

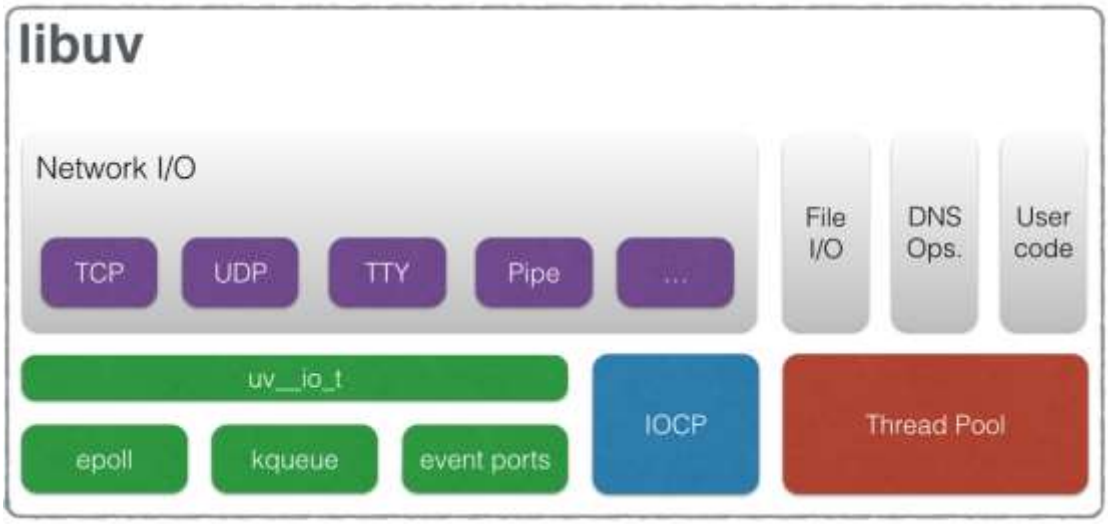
本文档是我在学习 nodejs 和 libuv 源码过程中的一些理解和心得。是对之前文章的整理，本文档非科普文，也写得比较粗糙。因为时间有限而代码过多，无法做到通俗易懂。需要读者有一定的操作系统和网络基础，如作者有理解错误之处或读者有不解之处，欢迎交流。有些还没有读完或者没有理解，后续有时间继续更新。如果你也喜欢 nodejs、libuv、操作系统、网络，也欢迎交流。

目录

一、Libuv 介绍

1.1 Libuv 架构

Libuv 是一个跨平台的的基于事件驱动的异步 io 库。但是他提供的功能不仅仅是 io，包括进程、线程、信号、定时器、进程间通信等。下面是来自官网对 Libuv 架构的介绍图。

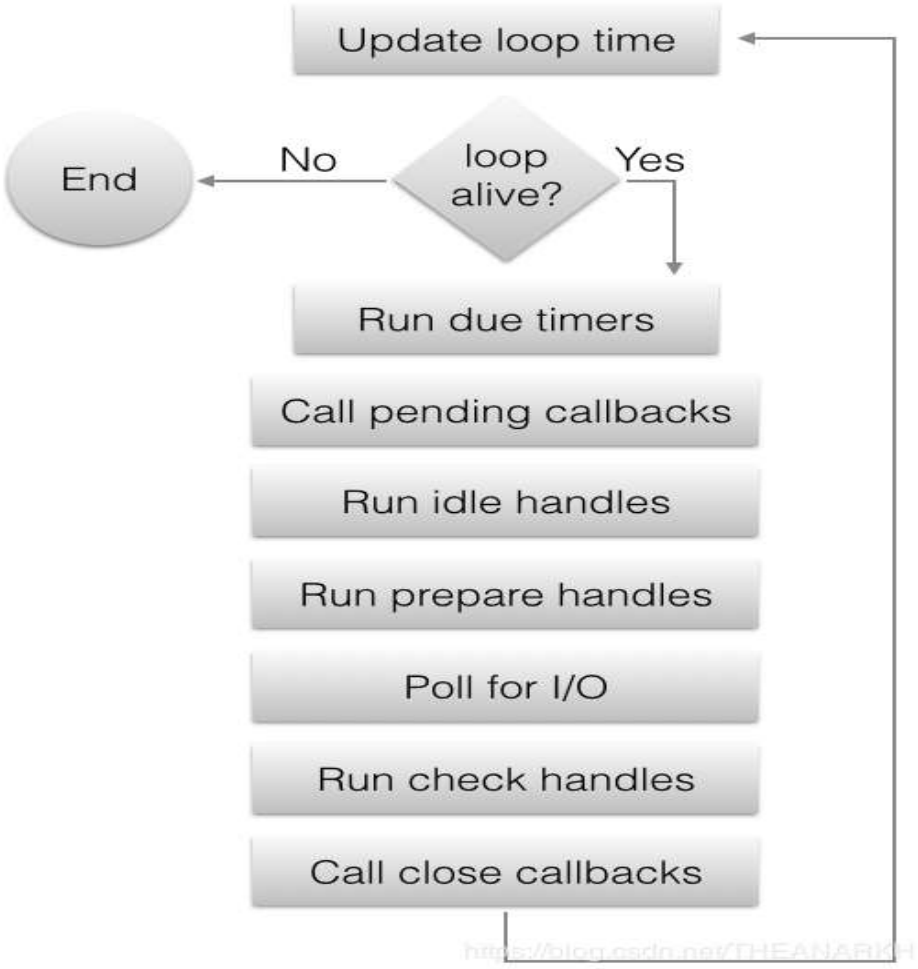


从上图中我们看到

1 Libuv 使用各平台提供的事件驱动模块实现异步（epoll，kqueue，IOCP，event ports）。他用来支持上层非文件 io 的模块。

2 Libuv 实现一个线程池用来支持上层文件 io、dns 以及用户层耗 cpu 的任务。

Libuv 的整体执行架构



从上图中我们大致了解到，Libuv 分为几个阶段，然后在一个循环里不断执行每个阶段里的任务。下面我们具体看一下每个阶段。

1 更新当前事件，在每次事件循环开始的时候，libuv 会更新当前事件到变量中，这一轮循环的剩下操作可以使用这个变量获取当前时间，避免过多的系统调用影响性能。额外的影响就是时间不是那么精确。但是在一轮事件循环中，libuv 在必要的时候，会主动更新这个时间，比如在 epoll 中阻塞了 timeout 时间后返回时，会再次更新当前时间变量。

2 如果时间循环是处于 alive 状态，则开始处理事件循环的每个阶段。否则退出这个事件循环。alive 状态是什么意思呢？如果有 active 和 ref 状态的 handle，active 状

态的 request 或者 closing 状态的 handle 则认为事件循环是 alive 的（具体的后续会讲到）。

3 timer 阶段：判断最小堆中的节点哪个节点超时了，执行他的回调。

4 pending 阶段：执行 pending 回调。一般来说，所有的 io 回调（网络，文件，dns）都会在 poll io 阶段执行。但是有的情况下，poll io 阶段的回调会延迟到下一次循环执行，那么这种回调就是在 pending 阶段执行的。

5 idle 阶段：如果节点处理 avtive 状态，注入每次事件循环都会被执行（idle 不是说事件循环空闲的时候才执行）。

6 prepare 阶段：和 idle 阶段一样。

7 poll io 阶段：计算最长等待时间 timeout，计算规则：

如果时间循环是以 UV_RUN_NOWAIT 模式运行的，则 timeout 是 0。

如果时间循环即将退出（调用了 uv_stop），则 timeout 是 0。

如果没有 active 状态的 handle 或者 request，timeout 是 0。

如果有 dile 阶段的队列里有节点，则 timeout 是 0。

如果有 handle 等待被关闭的（即调了 uv_close），timeout 是 0。

如果上面的都不满足，则取 timer 阶段中最快超时的节点作为 timeout，

如果没有则 timeout 等于-1，即永远阻塞，直到满足条件。

8 poll io 阶段：调用各平台提供的 io 多路复用接口，最多等待 timeout 时间。返回的时候，执行对应的回调。（比如 linux 下就是 epoll 模式）

9 check 阶段：和 idle prepare 一样。

10 closing 阶段：处理调用了 uv_close 函数的 handle 的回调。

11 如果 libuv 是以 UV_RUN_ONCE 模式运行的，那事件循环即将退出。但是有一种情况是，poll io 阶段的 timeout 的值是 timer 阶段的节点的值。并且 poll io 阶段是因为超时返回的，即没有任何事件发生，也没有执行任何 io 回调。这时候需要在执行一次 timer 阶段。因为有节点超时了。

12 一轮事件循环结束，如果 libuv 以 UV_RUN_NOWAIT 或 UV_RUN_ONCE 模式运行的，则退出事件循环。如果是以 UV_RUN_DEFAULT 模式运行的并且状态是 alive，则开始下一轮循环。否则退出事件循环。

1.2 Libuv 源码获取及编译

1.2.1 获取源码

Libuv 是开源项目，其开源项目地址为：https://github.com/libuv/libuv

1.2.2 编译和使用（linux 系统）

1 安装包：apt install automake，apt install libtool

2 执行 sh autogen.sh，make，make install

3 安装完后 libuv 的头文件和库在/usr/local/libuv 文件夹下。编译测试代码。

```
#include <stdio.h>

#include <stdlib.h>

#include <uv.h>

 int main() {

 uv_loop_t *loop = malloc(sizeof(uv_loop_t));

 uv_loop_init(loop);



 printf("Now quitting.\n");

 uv_run(loop, UV_RUN_DEFAULT);



 uv_loop_close(loop);

 free(loop);

 return 0;

}
```

使用 gcc 编译，gcc test.c -I /usr/local/libuv/include -L/usr/local/libuv/lib –luv

-I（大写 i） 是指定头文件的路径 -L 是指定链接库的路径 -l 是需要链接的动态库

编译完后执行会报错找不到动态链接库。

我们要为 gcc 指定路径。

1. 方法 1： 在用户目录下（比如/home/user）编辑.bashrc 文件，最后加入`export LD_LIBRARY_PATH="/usr/local/libuv/lib"。执行 source .bashrc`

2. 方法 2： 执行`sudo vi /etc/ld.so.conf.d/libc.conf` ，在最后一行加`/usr/local/libuv/lib`（在/etc/ld.so.conf.d 下新增一个.conf 文件应该也是 ok 的，见/etc/ld.so.conf

）。

再次执行就成功了。

二、Libuv 数据结构

2.1 核心结构体 uv_loop_s

uv_loop_s 是 Libuv 的核心数据结构，每个一个 Libuv 实例对应一个 uv_loop_s 结构体。他记录了整个事件循环中的核心数据。我们来分析每一个字段的意义。

void* data;

这个是用户自定义数据的字段

unsigned int active_handles;

活跃的 handle 个数

void* handle_queue[2];

handle 队列

union { void* unused[2]; unsigned int count; } active_reqs;

request 个数（主要用于文件操作）

unsigned int stop_flag;

事件循环是否结束的标记

unsigned long flags;

libuv 运行的一些标记 ,目前只有 UV_LOOP_BLOCK_SIGPROF ,主要是用于 epoll_wait 的时候屏蔽 SIGPROF 信号，提高性能，SIGPROF 是调操作系统 settimer 函数设置从而触发的信号

int backend_fd;

epoll 的 fd

void* pending_queue[2];

pending 阶段的队列

void* watcher_queue[2];

需要在 epoll 中注册的结构体队列，上下文是 uv__io_t

uv__io_t** watchers;

watcher_queue 队列的节点中有一个 fd 字段，watchers 以 fd 为索引，记录 fd 所在的 uv__io_t 结构体

unsigned int nwatchers;

watchers 相关的数量，在 maybe_resize 函数里设置

unsigned int nfds;

watchers 里 fd 个数，一般为 watcher_queue 队列的节点

数

void* wq[2];

线程池的线程处理完任务后把对应的结构体插入到 wq 队

列

uv_mutex_t wq_mutex;

控制 wq 队列互斥访问

uv_async_t wq_async;

用于线程池和主线程通信

uv_rwlock_t cloexec_lock;

用于读写锁的互斥变量

uv_handle_t* closing_handles;

closing 阶段的队列。由 uv_close 产生

void* process_handles[2];

fork 出来的进程队列

void* prepare_handles[2];

libuv 的 prepare 阶段对应的任务队列

void* check_handles[2];

libuv 的 check 阶段对应的任务队列

void* idle_handles[2];

libuv 的 idle 阶段对应的任务队列

void* async_handles[2];

async_handles 队列，在线程池中发送就绪信号给主线程的时候，主线程在 poll io 阶段执行 uv__async_io 中遍历 async_handles 队列处理里面 pending 为 1 的节点。

uv__io_t async_io_watcher;

保存了线程通信管道的读端和回调，用于接收线程池的消息，调用 uv__async_io 回调处理 async_handle 队列的节点

int async_wfd;

用于保存线程池和主线程通信的写端 fd

struct { void* min; unsigned int nelts;} timer_heap;

保存定时器二叉堆结构

uint64_t timer_counter;

管理定时器节点的 id，不断叠加

uint64_t time;

当前时间，Libuv 会在每次事件循环的开始和 poll io 阶段会更新当前时间，然后在后续的各个阶段使用，减少对系统调用。

int signal_pipefd[2];

用于 fork 出来的进程和主进程通信的管道，用于非主进程收到信号的时候通知主进程，然后主进程执行非主进程节点注册的回调

uv__io_t signal_io_watcher; uv_signal_t child_watcher;

类似 async_handle，signal_io_watcher 保存了管道读端 fd 和回调，然后注册到 epoll 中，在非主进程收到信号的时候，通过 write 写到管道，最后在 poll io 阶段执行回调。

int emfile_fd;

备用的 fd

2.1.1 基类 uv_handle_t

在 Libuv 中，uv_handle_t 是一个基类，有很多子类继承于他（类似 c++的继承）。handle 代表生命周期比较长的对象。例如

一个处于 active 状态的 prepare handle，他的回调会在每次事件循环化的时候被执行。

一个 tcp handle 在每次有连接到来时，执行他的回调。

我们看一下 uv_handle_t 的定义

```
// 自定义的数据
void* data;
// 所属的事件循环
  uv_loop_t* loop;
 // handle 类型
  uv_handle_type type;
 // handle 被关闭后被执行的回调
  uv_close_cb close_cb;
 // 用于组织 handle 队列的前置后置指针
  void* handle_queue[2];
// 文件描述符
  union {
    int fd;
    void* reserved[4];
  } u;
 // 用于插入 close 阶段队列
uv_handle_t* next_closing;
// handle 的状态和标记
unsigned int flags;
```

2.1.2 uv_handle_t 族结构体之 uv_stream_s

uv_stream_s 是表示流的结构体。除了继承 uv_handle_t 的字段外，他额外定义下面字段

```
// 等待发送的字节数
size_t write_queue_size;
// 分配内存的函数
  uv_alloc_cb alloc_cb;
// 读取数据成功时执行的回调
  uv_read_cb read_cb;
 // 连接成功后，执行 connect_req 的回调（connect_req 在 uv__xxx_connect 中赋值）
   uv_connect_t *connect_req;
 // 关闭写端的时候，发送完缓存的数据，回调 shutdown_req 的回调( shutdown_req 在 uv_shutdown 的时候赋值）
  uv_shutdown_t *shutdown_req;
 // 用于插入 epoll，注册读写事件
  uv__io_t io_watcher;
 // 待发送队列
  void* write_queue[2];
 // 发送完成的队列
  void* write_completed_queue[2];
```

```
// 收到连接，并且 accept 后执行 connection_cb 回调
uv_connection_cb connection_cb;
// socket 操作失败的错误码
int delayed_error;
// accept 返回的 fd
int accepted_fd;
// 已经 accept 了一个 fd，又有新的 fd，暂存起来
void* queued_fds;
```

2.1.3 uv_handle_t 族结构体之 uv_tcp_s

uv_tcp_s 继承 uv_handle_s 和 uv_stream_s。

2.1.4 uv_handle_t 族结构体之 uv_udp_s

```
// 发送字节数
size_t send_queue_size;
// 写队列节点的个数
size_t send_queue_count;
// 分配接收数据的内存
uv_alloc_cb alloc_cb;
// 接收完数据后执行的回调
uv_udp_recv_cb recv_cb;
// 插入 epoll 里的 io 观察者，实现数据读写
```

uv__io_t io_watcher;

// 待发送队列

void* write_queue[2];

// 发送完成的队列（发送成功或失败），和待发送队列相关

void* write_completed_queue[2];

2.1.5 uv_handle_t 族结构体之 uv_tty_s

uv_tty_s 继承于 uv_handle_t 和 uv_stream_t。额外定义了下面字段。

// 终端的参数

struct termios orig_termios;

// 终端的工作模式

int mode;

2.1.6 uv_handle_t 族结构体之 uv_pipe_s

uv_pipe_s 继承于 uv_handle_t 和 uv_stream_t。额外定义了下面字段。

// 标记管道是否能在进程间传递

int ipc;

// 用于 unix 域通信的文件路径

const char* pipe_fname;

2.1.2 uv_handle_t 族结构体之 uv_poll_s

uv_poll_s 继承于 uv_handle_t，额外定义了下面字段。

```
// 监听的 fd 有感兴趣的事件时执行的回调
uv_poll_cb poll_cb;
// 保存了 fd 和回调的 io 观察者，注册到 epoll 中
uv__io_t io_watcher;
```

2.1.7 uv_handle_t 族结构体之 uv_prepare_s、uv_check_s、uv_idle_s

上面三个结构体定义是类似的，他们都继承 uv_handle_t，额外定义了两个字段。

```
//  prepare、check、idle 阶段回调
uv_xxx_cb xxx_cb;
// 用于插入 prepare、check、idle 队列
void* queue[2];
```

2.1.8 uv_handle_t 族结构体之 uv_timer_s

Uv_timer_s 继承 uv_handle_t，额外定义了下面几个字段。

```
// 超时回调
uv_timer_cb timer_cb;
// 插入二叉堆的字段
 void* heap_node[3];
// 超时时间
 uint64_t timeout;
// 超时后是否继续开始重新计时，是的话重新插入二叉堆
 uint64_t repeat;
```

// id 标记，用于插入二叉堆的时候对比

uint64_t start_id

2.1.9 uv_handle_t 族结构体之 uv_process_s

uv_process_s 继承 uv_handle_t，额外定义了

// 进程退出时执行的回调

uv_exit_cb exit_cb;

// 进程 id

int pid;

// 用于插入队列，进程队列或者 pending 队列

void* queue[2];

// 退出码，进程退出时设置

int status;

2.1.10 uv_handle_t 族结构体之 uv_fs_event_s

uv_fs_event_s 用于监听文件改动。uv_fs_event_s 继承 uv_handle_t，额外定义了

// 监听的文件路径(文件或目录)

char* path;

// 文件改变时执行的回调

uv_fs_event_cb cb;

2.1.11 uv_handle_t 族结构体之 uv_fs_poll_s

uv_fs_poll_s 继承 uv_handle_t，额外定义了

```
// 上下文
void* poll_ctx;
```

poll_ctx 指向一个结构体

```
struct poll_ctx {
  // 对应的 handle
  uv_fs_poll_t* parent_handle;
  // 标记是否开始轮询和轮询时的失败原因
  int busy_polling;
  // 多久检测一次文件内容是否改变
  unsigned int interval;
  // 每一轮轮询时的开始时间
  uint64_t start_time;
  // 所属事件循环
  uv_loop_t* loop;
  // 文件改变时回调
  uv_fs_poll_cb poll_cb;
// 定时器，用于定时超时后轮询
  uv_timer_t timer_handle;
  // 记录轮询的一下上下文信息，文件路径、回调等
```

```
  uv_fs_t fs_req;
 // 轮询时保存操作系统返回的文件信息
  uv_stat_t statbuf;
 // 监听的文件路径，字符串的值追加在结构体后面
  char path[1]; /* variable length */
};
```

2.1.12 uv_handle_t 族结构体之 uv_signal_s

uv_signal_s 解除 uv_handle_t，额外定义了以下字段

```
//  收到信号时的回调
uv_signal_cb signal_cb;
// 注册的信号
int signum;
/*
```

用于插入红黑树，进程把感兴趣的信号和回调封装成 uv_signal_s，然后插入到红黑树，信号到来时，进程在信号处理号中把通知写入管道，通知 libuv。libuv 在 poll io 阶段会执行进程对应的回调。

```
*/
struct {
  struct uv_signal_s* rbe_left;
  struct uv_signal_s* rbe_right;
  struct uv_signal_s* rbe_parent;
```

```
  int rbe_color;
} tree_entry;
// 收到的信号个数
unsigned int caught_signals;
// 已经处理的信号个数
unsigned int dispatched_signals;
```

2.1.13 uv_handle_t 族结构体之 uv_async_s

uv_async_s 是 Libuv 中实现主进程和其他进程线程异步通信的结构体。继承于 uv_handle_t，并额外定义了以下字段。

```
// 异步事件触发时执行的回调
uv_async_cb async_cb;
// 用于插入 async-handles 队列
void* queue[2];
/*
async_handles 队列中的节点 pending 字段为 1 说明对应的事件触发了，
主要用于线程池和主线程的通信
*/
int pending;
```

2.1.14 基类 uv_req_s

在 Libuv 中，uv_req_s 是一个基类，有很多子类继承于他，request 代表生命周期比较

短的请求。比如读写一个文件，读写 socket，查询 dns。任务完成后这个 request 就结束了。request 可以和 handle 结合使用，比如在一个 tcp 服务器上（handle）写一个数据（request），也可以单独使用一个 request，比如 dns 查询或者文件读写。

2.1.15 uv_req_s 族结构体之 uv_shutdown_s

额外定义的字段

```
// 要关闭的流，比如 tcp
uv_stream_t* handle;
// 关闭流的写端后执行的回调
uv_shutdown_cb cb;
```

2.1.16 uv_req_s 族结构体之 uv_write_s

```
// 写完后的回调
uv_write_cb cb;
// 需要传递的文件描述符，在 send_handle 中
  uv_stream_t* send_handle;
// 关联的 handle
  uv_stream_t* handle;
 // 用于插入队列
   void* queue[2];
// 保存需要写的数据相关的字段
  unsigned int write_index;
```

```
  uv_buf_t* bufs;
  unsigned int nbufs;
  uv_buf_t bufsml[4];
// 写出错的错误码
  int error;
```

2.1.17 uv_req_s 族结构体之 uv_connect_s

```
// 连接成功后执行的回调
uv_connect_cb cb;
// 对应的流，比如 tcp
uv_stream_t* handle;
// 用于插入队列
void* queue[2];
```

2.1.18 uv_req_s 族结构体之 uv_udp_send_s

```
// 所属 udp 的 handle，udp_send_s 代表一次发送，需要对应一个 udp handle
uv_udp_t* handle;
// 没用到
uv_udp_send_cb cb;
// 用于插入待发送队列
void* queue[2];
// 发送的目的地址
```

```
struct sockaddr_storage addr;
// 保存了发送数据的缓冲区和个数
unsigned int nbufs;
uv_buf_t* bufs;
uv_buf_t bufsml[4];
// 发送状态或成功发送的字节数
ssize_t status;
// 发送完执行的回调（发送成功或失败）
uv_udp_send_cb send_cb;
```

2.1.19 uv_req_s 族结构体之 uv_getaddrinfo_s

```
// 所属事件循环
uv_loop_t* loop;
// 用于异步 dns 解析时插入线程池任务队列的节点
struct uv__work work_req;
// dns 解析完后执行的回调
uv_getaddrinfo_cb cb;
// dns 查询的配置
struct addrinfo* hints;
char* hostname;
char* service;
// dns 解析结果
```

struct addrinfo* addrinfo;

// dns 解析的返回码

int retcode;

2.1.20 uv_req_s 族结构体之 uv_getnameinfo_s

uv_loop_t* loop;

// 用于异步 dns 解析时插入线程池任务队列的节点

struct uv__work work_req;

// socket 转域名完成的回调

uv_getnameinfo_cb getnameinfo_cb;

// 需要转域名的 socket 结构体

struct sockaddr_storage storage;

// 指示查询返回的信息

int flags;

// 查询返回的信息

char host[NI_MAXHOST];

char service[NI_MAXSERV];

// 查询返回码

int retcode;

2.1.21 uv_req_s 族结构体之 uv_work_s

uv_loop_t* loop;

```
// 处理任务的函数
uv_work_cb work_cb;
// 处理完任务后执行的函数
uv_after_work_cb after_work_cb;
/*
 封装一个work插入到线程池队列 ,work_req的work和done函数是对上面work_cb
和 after_work_cb 的封装
*/
struct uv__work work_req;
```

2.1.22 uv_req_s 族结构体之 uv_fs_s

```
// 文件操作类型
uv_fs_type fs_type;
  uv_loop_t* loop;
 // 文件操作完成的回调
  uv_fs_cb cb;
 // 文件操作的返回码
  ssize_t result;
 // 文件操作返回的数据
  void* ptr;
// 文件操作路径
  const char* path;
```

```
  // 文件的 stat 信息
  uv_stat_t statbuf;
// 文件操作涉及到两个路径时，保存目的路径
const char *new_path;
// 文件描述符
uv_file file;
// 文件标记
int flags;
// 操作模式
mode_t mode;
// 写文件时传入的数据和个数
unsigned int nbufs;
uv_buf_t* bufs;
// 文件偏移
off_t off;
// 保存需要设置的 uid 和 gid，例如 chmod 的时候
uv_uid_t uid;
uv_gid_t gid;
// 保存需要设置的文件修改、访问时间，例如 fs.utimes 的时候
double atime;
double mtime;
// 异步的时候用于插入任务队列，保存工作函数，回调函数
```

struct uv__work work_req;

// 保存读取数据或者长度。例如 read 和 sendfile

uv_buf_t bufsml[4];

2.2 queue

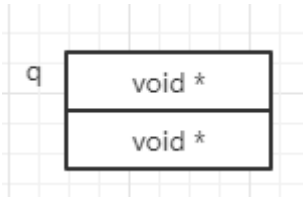
Libuv 中的 queue 实现非常复杂。队列在 Libuv 中到处可见，所以理解队列的实现，才能更容易读懂 Libuv 其他代码。因为他是 Libuv 中的一个非常通用的数据结构。首先我们看一下他的定义。

typedef void *QUEUE[2];

这个是 c 语言中定义类型别名的一种方式。比如我们定义一个变量

QUEUE q 相当于 void *q[2];

即一个数组，他每个元素是 void 型的指针。



下面我们接着分析四个举足轻重的宏定义，理解他们就相当于理解了 libuv 的队列。在分析之前，我们先来回顾一下数组指针和二维数组的知识。

int a[2];

// 数组指针

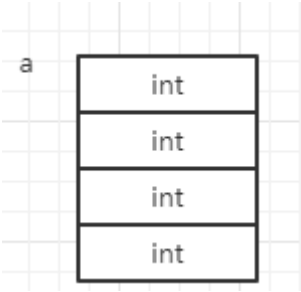
int (*p)[2] = a;

// *(*(p+0)+1)取元素的值

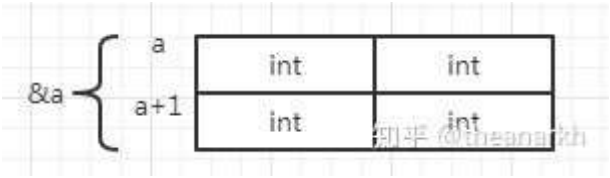
二维数组

int a[2][2];

我们知道二维数组在内存中的布局是一维。



但是为了方便理解我们画成二维的。



1. &a 代表二维数组的首地址。类型是 int (*)[2][2]，他是一个指针，他指向的元素是一个二维数组。假设 int 是四个字节。数组首地址是 0，那么&a + 1 等于 16.

2. a 代表第一行的首地址，类型是 int (*)[2]，他是一个指针，指向的元素是一个一维数组。a+1 等于 8。

3. a[0]也是第一行的首地址，类型是 int *。

4. &a[0]也是第一行的首地址，类型是 int (*)[2];

5. 如果 int (p) = &a[0]，那么我们想取数组某个值的时候，可以使用((p+i) + j)的方式。(p+i)即把范围固定到第一行（这时候的指针类型是 init ），(*(p+i) + j)即在第一行的范围内定位到某一列，然后通过解引用取得内存的值。

下面开始分析 libuv 的具体实现

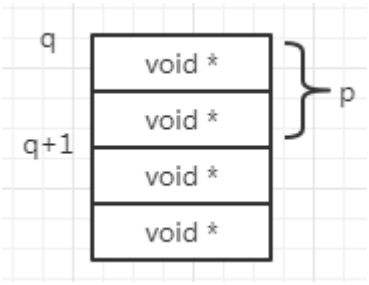
2.3.1 QUEUE_NEXT

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

QUEUE_NEXT 看起来是获取当前节点的 next 字段的地址。但是他的实现非常巧妙。我们逐步分析这个宏定义。首先我们先看一下 QUEUE_NEXT 是怎么使用的。

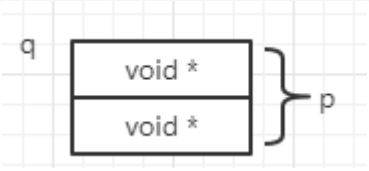
```
void *p[2][2];

QUEUE* q = &p[0]; // void *(*q)[2] = &p[0];

QUEUE_NEXT(q);
```

我们看到 QUEUE_NEXT 的参数是一个指针，他指向一个大小为 2 的数组，数组里的每个元素是 void 。内存布局如下。



因为 libuv 的数组只有两个元素。相当于 p[2][2]变成了*p[2][1]。所以上面的代码简化为。

```
void *p[2];

QUEUE* q = &p; // void *(*q)[2] = &p;

QUEUE_NEXT(q);
```



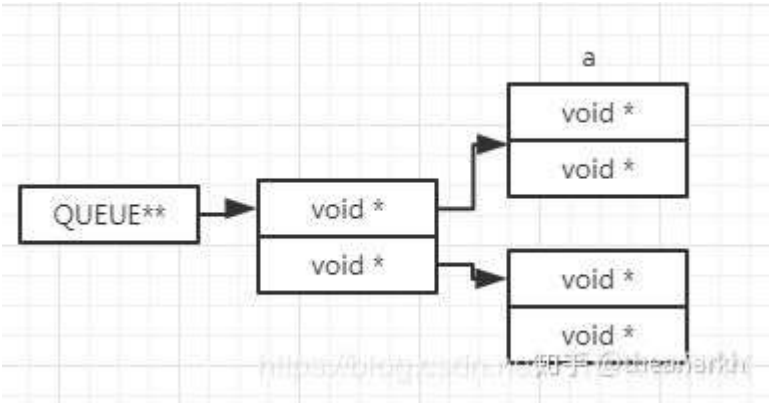
根据上面的代码我们逐步展开宏定义。

q 指向整个数组 p 的首地址，*(q)还指向数组第一行的首地址（这时候指针类型为 void *，见上面二维数组的分析 5 ）。

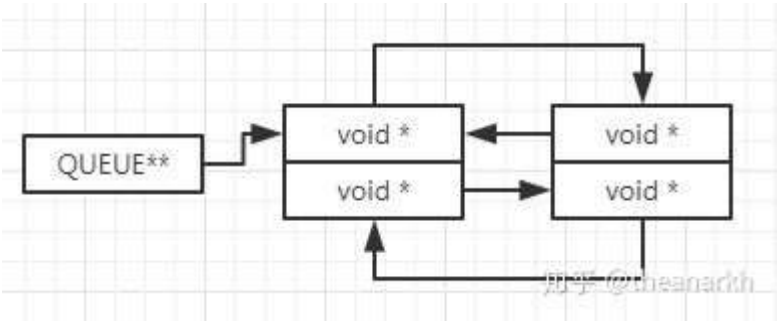
(*(q))[0]即把指针定位到第一行第一列的内存地址（这时候指针类型还是 void *，见上

面二维数组的分析 5 ）。

&((*(q))[0])把 2 中的结果（即 void * ）转成二级指针（void ** ），然后强制转换类型(QUEUE **) 。为什么需要强制转成等于 QUEUE **呢？因为需要保持类型。转成 QUEUE **后（即 void * (**)[2] ）。说明他是一个二级指针，他指向一个指针数组，每个元素指向一个大小为 2 的数组。这个大小为 2 的数组就是下一个节点的地址。



在 libuv 中如下



*(QUEUE *) &(((q))[0])解引用取得 q 下一个节点的地址（作为右值），或者修改当前节点的 next 域内存里的值（作为左值）,类型是 void (*)[2]。

2.3.2 QUEUE_PREV

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

prev 的宏和 next 是类似的，区别是 prev 得到的是当前节点的上一个节点的地址。不再分析。

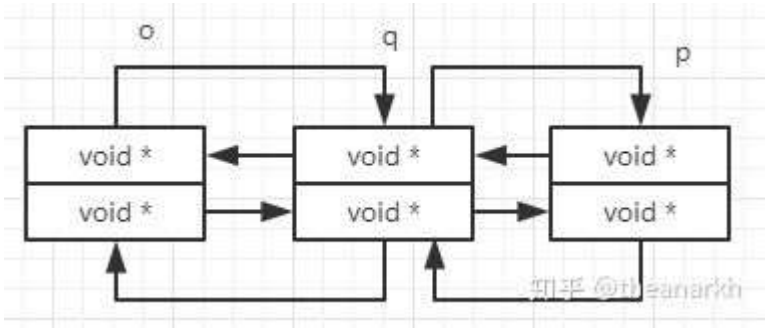
2.3.3 QUEUE_PREV_NEXT、QUEUE_NEXT_PREV

```
#define QUEUE_PREV_NEXT(q)  (QUEUE_NEXT(QUEUE_PREV(q))

#define QUEUE_NEXT_PREV(q)  (QUEUE_PREV(QUEUE_NEXT(q))
```

这两个宏就是取当前节点的前一个节点的下一个节点和取当前节点的后一个节点的前一个节点。那不就是自己吗？这就是 libuv 队列的亮点了。下面我们看一下这些宏的使用。

2.3.4 删除节点 QUEUE_REMOVE

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



1 QUEUE_NEXT(q); 拿到 q 下一个节点的地址，即 p

2 QUEUE_PREV_NEXT(q)分为两步，第一步拿到 q 前一个节点的地址。即 o。然后再执行 QUEUE_NEXT(o),分析之前我们先看一下关于指针变量作为左值和右值的问题。

int zym = 9297;

int *cyb = &zym;

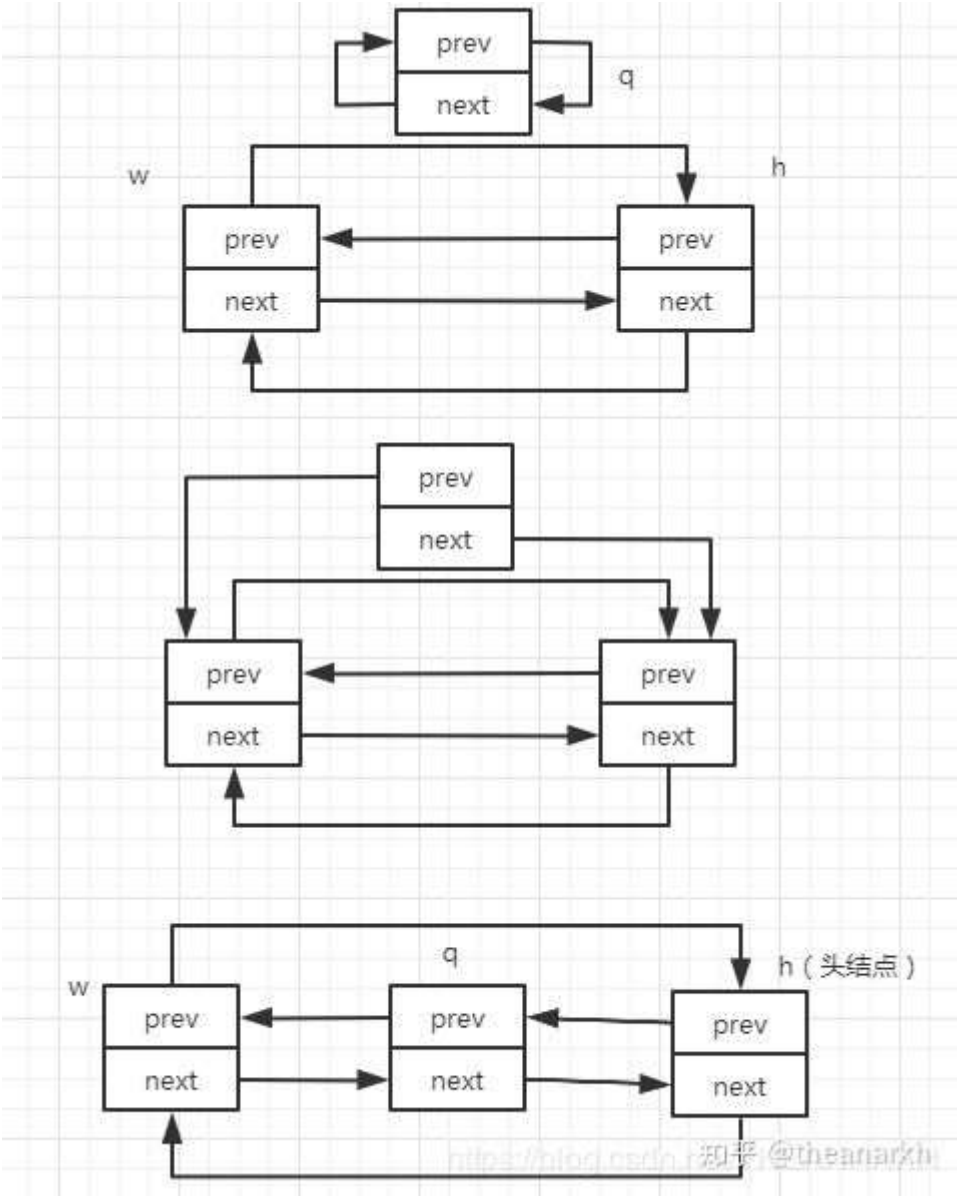
int hello = *cyb; // hello 等于 9297

int *cyb = 1101;

我们看到一个指针变量，如果他在右边，对他解引用（p）的时候，得到的值是他指向内存里的值。而如果他在左边的时候，p 就是修改他自己内存里的值。我们回顾对 QUEUE_NEXT 宏的分析。他返回的是一个指针 void (*)[2]。所以 QUEUE_PREV_NEXT(q) = QUEUE_NEXT(q); 的效果其实是修改 q 的前置节点（o）的 next 指针的内存。让他指向 q 的下一个节点（p），就这样完成了 q 的删除。

2.3.5 插入队列 QUEUE_INSERT_TAIL

```
// q 插入 h，h 是头节点
#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)
```



2.3 io 观察者

io 观察者是 Libuv 中的核心概念和数据结构。我们看一下他的定义

```
struct uv__io_s {
  // 事件触发后的回调
  uv__io_cb cb;
  // 用于插入队列
```

```
  void* pending_queue[2];
  void* watcher_queue[2];
  // 保存本次感兴趣的事件，在插入 io 观察者队列时设置
  unsigned int pevents; /* Pending event mask i.e. mask at next tick. */
  // 保存当前感兴趣的事件
  unsigned int events;   /* Current event mask. */
  int fd;
};
```

io 观察者就是封装了事件和回调的结构体，然后插入到 loop 维护的 io 观察者队列，在 poll io 阶段，libuv 会根据 io 观察者描述的信息，往底层的事件驱动模块注册相应的信息。当注册的事件触发的时候，io 观察者的回调就会被执行。我们看如何初 io 观察者的一些逻辑。

1 初始化 io 观察者

```
void uv__io_init(uv__io_t* w, uv__io_cb cb, int fd) {
  // 初始化队列，回调，需要监听的 fd
  QUEUE_INIT(&w->pending_queue);
  QUEUE_INIT(&w->watcher_queue);
  w->cb = cb;
  w->fd = fd;
  // 上次加入 epoll 时感兴趣的事件，在执行完 epoll 操作函数后设置
  w->events = 0;
  // 当前感兴趣的事件，在再次执行 epoll 函数之前设置
```

```
  w->pevents = 0;
}
```

2 注册一个 io 观察到 libuv。

```
void uv__io_start(uv_loop_t* loop, uv__io_t* w, unsigned int events) {
  // 设置当前感兴趣的事件
  w->pevents |= events;
  // 可能需要扩容
  maybe_resize(loop, w->fd + 1);
  if (w->events == w->pevents)
    return;
  // io 观察者没有挂载在其他地方则插入 libuv 的 io 观察者队列
  if (QUEUE_EMPTY(&w->watcher_queue))
    QUEUE_INSERT_TAIL(&loop->watcher_queue, &w->watcher_queue);
  // 保存映射关系
  if (loop->watchers[w->fd] == NULL) {
    loop->watchers[w->fd] = w;
    loop->nfds++;
  }
}
```

uv__io_start 函数就是把一个 io 观察者插入到 libuv 的观察者队列中 ,并且在 watchers 数组中保存一个映射关系。libuv 在 poll io 阶段会处理 io 观察者队列。

3 撤销 io 观察者或者事件

uv__io_stop 修改 io 观察者感兴趣的事件，如果还有感兴趣的事件的话，io 观察者还会在队列里，否则移出

```
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;
  // 清除之前注册的事件，保存在 pevents 里，表示当前感兴趣的事件
  w->pevents &= ~events;
  // 对所有事件都不感兴趣了
  if (w->pevents == 0) {
    // 移出 io 观察者队列
    QUEUE_REMOVE(&w->watcher_queue);
    // 重置
    QUEUE_INIT(&w->watcher_queue);
    // 重置
    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;
      }
    }
    /*
          之前还没有插入 io 观察者队列，则插入，
          等到 poll io 时处理，否则不需要处理
    */
    else if (QUEUE_EMPTY(&w->watcher_queue))
      QUEUE_INSERT_TAIL(&loop->watcher_queue, &w->watcher_queue);
}
```

三、Libuv 的实现

3.1 通用逻辑

1. uv__handle_init

uv__handle_init 初始化 handle 的类型，设置 REF 标记，插入 handle 队列。

```
#define uv__handle_init(loop_, h, type_)
  do {
```

```
    (h)->loop = (loop_);
    (h)->type = (type_);
    (h)->flags = UV_HANDLE_REF;
    QUEUE_INSERT_TAIL(&(loop_)->handle_queue, &(h)->handle_queue);
    (h)->next_closing = NULL
  }
  while (0)
```

2. uv__handle_start

uv__handle_start 设置标记 handle 为 ACTIVE，如果设置了 REF 标记，则 active handle 的个数加一，active handle 数会影响事件循环的退出。

```
#define uv__handle_start(h)
  do {
    if (((h)->flags & UV_HANDLE_ACTIVE) != 0) break;
    (h)->flags |= UV_HANDLE_ACTIVE;
    if (((h)->flags & UV_HANDLE_REF) != 0)
    (h)->loop->active_handles++;
  }
  while (0)
```

3. uv__handle_stop

uv__handle_stop 和 uv__handle_start 相反。

```
#define uv__handle_stop(h)
  do {
```

```
    if (((h)->flags & UV_HANDLE_ACTIVE) == 0) break;
    (h)->flags &= ~UV_HANDLE_ACTIVE;
    if (((h)->flags & UV_HANDLE_REF) != 0) uv__active_handle_rm(h);
  }
  while (0)
```

libuv 中 handle 有 REF 和 ACTIVE 两个状态。当一个 handle 调用 xxx_init 函数的时候，他首先被打上 REF 标记，并且插入 loop->handle 队列。当 handle 调用 xxx_start 函数的时候，他首先被打上 ACTIVE 标记，并且记录 active handle 的个数加一。只有 ACTIVE 状态的 handle 才会影响事件循环的退出。

4. uv__req_init

uv__req_init 初始化请求的类型，记录请求的个数

```
#define uv__req_init(loop, req, typ)
  do {
    (req)->type = (typ);
    (loop)->active_reqs.count++;
  }
  while (0)
```

5. uv__req_register

uv__req_register 记录请求（request）的个数加一

```
#define uv__req_register(loop, req)
  do {
    (loop)->active_reqs.count++;
```

```
  }
  while (0)
```

6. uv__req_unregister

uv__req_unregister 记录请求（request）的个数减一

```
#define uv__req_unregister(loop, req)
  do {
    assert(uv__has_active_reqs(loop));
    (loop)->active_reqs.count--;
  }
  while (0)
```

7. uv__req_init

uv_req_init 初始化一个 request 类的 handle

```
#define uv__req_init(loop, req, typ)
  do {
    UV_REQ_INIT(req, typ);
    uv__req_register(loop, req);
  }
  while (0)
```

8. uv__handle_ref

uv__handle_ref 标记 handle 为 REF 状态，如果 handle 是 ACTIVE 状态，则 active handle 数加一

```
#define uv__handle_ref(h)
```

```
  do {
    if (((h)->flags & UV_HANDLE_REF) != 0) break;
    (h)->flags |= UV_HANDLE_REF;
    if (((h)->flags & UV_HANDLE_CLOSING) != 0) break;
    if (((h)->flags & UV_HANDLE_ACTIVE) != 0) uv__active_handle_add(h);
  }
  while (0)
```

9. uv__handle_unref

uv__handle_unref 去掉 handle 的 REF 状态 ,如果 handle 是 ACTIVE 状态 ,则 active handle 数减一

```
#define uv__handle_unref(h)
  do {
    if (((h)->flags & UV_HANDLE_REF) == 0) break;
    (h)->flags &= ~UV_HANDLE_REF;
    if (((h)->flags & UV_HANDLE_CLOSING) != 0) break;
    if (((h)->flags & UV_HANDLE_ACTIVE) != 0) uv__active_handle_rm(h);
  }
  while (0)
```

3.2 事件循环

实现循环由 libuv 的 uv_run 函数实现。在该函数中执行 while 循环，然后处理各种阶段（phase）的事件回调。事件循环的处理相当于一个消费者，消费由各业务代码生产的任务。下面看一下代码。

```
int uv_run(uv_loop_t* loop, uv_run_mode mode) {
  int timeout;
  int r;
  int ran_pending;


  r = uv__loop_alive(loop);
  if (!r)
    uv__update_time(loop);


  while (r != 0 && loop->stop_flag == 0) {
    // 更新 loop 的 time 字段
    uv__update_time(loop);
    // 执行超时回调
    uv__run_timers(loop);
    // 执行 pending 回调，ran_pending 代表 pending 队列是否为空，即没有节点可以执行
    ran_pending = uv__run_pending(loop);
    // 继续执行各种队列
```

```
    uv__run_idle(loop);
    uv__run_prepare(loop);


    timeout = 0;
    // UV_RUN_ONCE 并且有 pending 节点的时候，会阻塞式 poll io，默认模式也是
    if ((mode == UV_RUN_ONCE && !ran_pending) || mode == UV_RUN_DEFAULT)
      timeout = uv_backend_timeout(loop);
    // poll io timeout 是 epoll_wait 的超时时间
    uv__io_poll(loop, timeout);
    uv__run_check(loop);
    uv__run_closing_handles(loop);
    // 还有一次执行超时回调的机会
    if (mode == UV_RUN_ONCE) {
      uv__update_time(loop);
      uv__run_timers(loop);
    }


    r = uv__loop_alive(loop);
    if (mode == UV_RUN_ONCE || mode == UV_RUN_NOWAIT)
      break;
  }
  if (loop->stop_flag != 0)
```

```
    loop->stop_flag = 0;
  return r;
}
```

libuv 分为几个阶段，下面分别分析各个阶段的相关代码。

3.2.1 事件循环之 close

close 是 libuv 每轮事件循环中最后的一个阶段。我们看看怎么使用。我们知道对于一个 handle ,他的使用一般是 init ,start ,stop。但是如果我们在 stop 一个 handle 之后，还有些事情需要处理怎么办？这时候就可以使用 close 阶段。close 阶段可以用来关闭一个 handle，并且执行一个回调。比如用于释放动态申请的内存。close 阶段的任务由 uv_close 产生。

```
void uv_close(uv_handle_t* handle, uv_close_cb close_cb) {
  // 正在关闭，但是还没执行回调等后置操作
  handle->flags |= UV_HANDLE_CLOSING;
  handle->close_cb = close_cb;

  switch (handle->type) {

  case UV_PREPARE:
    uv__prepare_close((uv_prepare_t*)handle);
    break;
```

```
  case UV_CHECK:
    uv__check_close((uv_check_t*)handle);
    break;
    ...
  default:
    assert(0);
  }

  uv__make_close_pending(handle);
}
```

uv_close 设置回调和状态，然后根据 handle 类型调对应的 close 函数，一般就是 stop 这个 handle。比如 prepare 的 close 函数。

```
void uv__prepare_close(uv_prepare_t* handle) {
    uv_prepare_stop(handle);
}
```

接着执行 uv__make_close_pending 往 close 队列追加节点。

```
// 头插法插入 closing 队列，在 closing 阶段被执行
void uv__make_close_pending(uv_handle_t* handle) {
  handle->next_closing = handle->loop->closing_handles;
```

```
  handle->loop->closing_handles = handle;
}
```

产生的节点在 closing_handles 队列中保存，然后在 close 节点逐个处理。

```
// 执行 closing 阶段的的回调
static void uv__run_closing_handles(uv_loop_t* loop) {
  uv_handle_t* p;
  uv_handle_t* q;

  p = loop->closing_handles;
  loop->closing_handles = NULL;

  while (p) {
    q = p->next_closing;
    uv__finish_close(p);
    p = q;
  }
}

// 执行 closing 阶段的回调
static void uv__finish_close(uv_handle_t* handle) {
  handle->flags |= UV_HANDLE_CLOSED;
```

```
  ...
  uv__handle_unref(handle);
  QUEUE_REMOVE(&handle->handle_queue);
  if (handle->close_cb) {
    handle->close_cb(handle);
  }
}
```

逐个执行回调，close 和 stop 有一点不同的是，stop 一个 handle，他不会从事件循环中被移除，但是 close 一个 handle，他会从事件循环的 handle 队列中移除。

我们看一个使用了 uv_close 的例子（省略部分代码）。

```
int uv_fs_poll_start(uv_fs_poll_t* handle,
                     uv_fs_poll_cb cb,
                     const char* path,
                     unsigned int interval) {
  struct poll_ctx* ctx;
  // 分配一块堆内存存上下文结构体和 path 对应的字符串
  ctx = uv__calloc(1, sizeof(*ctx) + len);
  // 挂载上下文到 handle
  handle->poll_ctx = ctx;
```

```
}
```

uv_fs_poll_start 是用于监听文件是否有改变的函数。他在 handle 里挂载了一个基于堆结构体。当结束监听的时候，他需要释放掉这块内存。

```
// 停止 poll
int uv_fs_poll_stop(uv_fs_poll_t* handle) {
  struct poll_ctx* ctx;
  ctx = handle->poll_ctx;
  handle->poll_ctx = NULL;
  uv_close((uv_handle_t*)&ctx->timer_handle, timer_close_cb);
}
```

uv_fs_poll_stop 通过 uv_close 函数关闭 handle，传的回调是 timer_close_cb。

```
// 释放上下文结构体的内存
static void timer_close_cb(uv_handle_t* handle) {
  uv__free(container_of(handle, struct poll_ctx, timer_handle));
}
```

所以在 close 阶段就会释放这块内存。

3.2.2 事件循环之 poll io

poll io 是 libuv 非常重要的一个阶段，文件 io、网络 io、信号处理等都在这个阶段处理。这也是最复杂的一个阶段。处理逻辑在 uv__io_poll 这个函数。这个函数比较复杂，我们分开分析。

开始说 poll io 之前，先了解一下他相关的一些数据结构。

1 io 观察者 uv__io_t。这个结构体是 poll io 阶段核心结构体。他主要是保存了 io 相关的文件描述符、回调、感兴趣的事件等信息。

2 watcher_queue 观察者队列。所有需要 libuv 处理的 io 观察者都挂载在这个队列里。libuv 会逐个处理。

下面我们开始分析 poll io 阶段。先看第一段逻辑。

```
// 没有 io 观察者，则直接返回
if (loop->nfds == 0) {
    assert(QUEUE_EMPTY(&loop->watcher_queue));
    return;
 }
 // 遍历 io 观察者队列
 while (!QUEUE_EMPTY(&loop->watcher_queue)) {
       // 取出当前头节点
    q = QUEUE_HEAD(&loop->watcher_queue);
    // 脱离队列
```

```
    QUEUE_REMOVE(q);
    // 初始化（重置）节点的前后指针
    QUEUE_INIT(q);
    // 通过结构体成功获取结构体首地址
    w = QUEUE_DATA(q, uv__io_t, watcher_queue);
    // 设置当前感兴趣的事件
    e.events = w->pevents;
    // 这里使用了 fd 字段，事件触发后再通过 fd 从 watchs 字段里找到对应的 io 观察者，没有使用 ptr 指向 io 观察者的方案
    e.data.fd = w->fd;
    // w->events 初始化的时候为 0，则新增，否则修改
    if (w->events == 0)
      op = EPOLL_CTL_ADD;
    else
      op = EPOLL_CTL_MOD;
    // 修改 epoll 的数据
    epoll_ctl(loop->backend_fd, op, w->fd, &e)
    // 记录当前加到 epoll 时的状态
    w->events = w->pevents;
  }
```

第一步首先遍历 io 观察者，修改 epoll 的数据，即感兴趣的事件。然后准备进入等待，如果设置了 UV_LOOP_BLOCK_SIGPROF 的话。libuv 会做一个优化。如果调 setitimer(ITIMER_PROF,...)设置了定时触发 SIGPROF 信号，则到期后，并且每隔一段时间后会触发 SIGPROF 信号，这里如果设置了 UV_LOOP_BLOCK_SIGPROF 救护屏蔽这个信号。否则会提前唤醒 epoll_wait。

```
  psigset = NULL;
  if (loop->flags & UV_LOOP_BLOCK_SIGPROF) {
    sigemptyset(&sigset);
    sigaddset(&sigset, SIGPROF);
    psigset = &sigset;
  }
    /*
       http://man7.org/linux/man-pages/man2/epoll_wait.2.html
       pthread_sigmask(SIG_SETMASK, &sigmask, &origmask);
       ready = epoll_wait(epfd, &events, maxevents, timeout);
       pthread_sigmask(SIG_SETMASK, &origmask, NULL);
       即屏蔽 SIGPROF 信号，避免 SIGPROF 信号唤醒 epoll_wait，但是却没有就绪的事件
    */
    nfds = epoll_pwait(loop->backend_fd,
```

```
                    events,
                    ARRAY_SIZE(events),
                    timeout,
                    psigset);
  // epoll 可能阻塞，这里需要更新事件循环的时间
  uv__update_time(loop)
```

在 epoll_wait 可能会引起主线程阻塞，具体要根据 libuv 当前的情况。所以 wait 返回后需要更新当前的时间，否则在使用的时候时间差会比较大。因为 libuv 会在每轮时间循环开始的时候缓存当前时间这个值。其他地方直接使用，而不是每次都去获取。下面我们接着看 epoll 返回后的处理（假设有事件触发）。

```
  // 保存 epoll_wait 返回的一些数据，maybe_resize 申请空间的时候+2 了
  loop->watchers[loop->nwatchers] = (void*) events;
  loop->watchers[loop->nwatchers + 1] = (void*) (uintptr_t) nfds;
  for (i = 0; i < nfds; i++) {
    // 触发的事件和文件描述符
    pe = events + i;
    fd = pe->data.fd;
    // 根据 fd 获取 io 观察者，见上面的图
    w = loop->watchers[fd];
    // 会其他回调里被删除了，则从 epoll 中删除
```

```
    if (w == NULL) {
      epoll_ctl(loop->backend_fd, EPOLL_CTL_DEL, fd, pe);
      continue;
    }
    if (pe->events != 0) {
      // 用于信号处理的 io 观察者感兴趣的事件触发了，即有信号发生。
      if (w == &loop->signal_io_watcher)
        have_signals = 1;
      else
        // 一般的 io 观察者指向回调
        w->cb(loop, w, pe->events);
      nevents++;
    }
  }
  // 有信号发生，触发回调
  if (have_signals != 0)
    loop->signal_io_watcher.cb(loop, &loop->signal_io_watcher, POLLIN);
```

这里开始处理 io 事件，执行 io 观察者里保存的回调。但是有一个特殊的地方就是信号处理的 io 观察者需要单独判断。他是一个全局的 io 观察者，和一般动态申请和销毁的 io 观察者不一样，他是存在于 libuv 运行的整个生命周期。

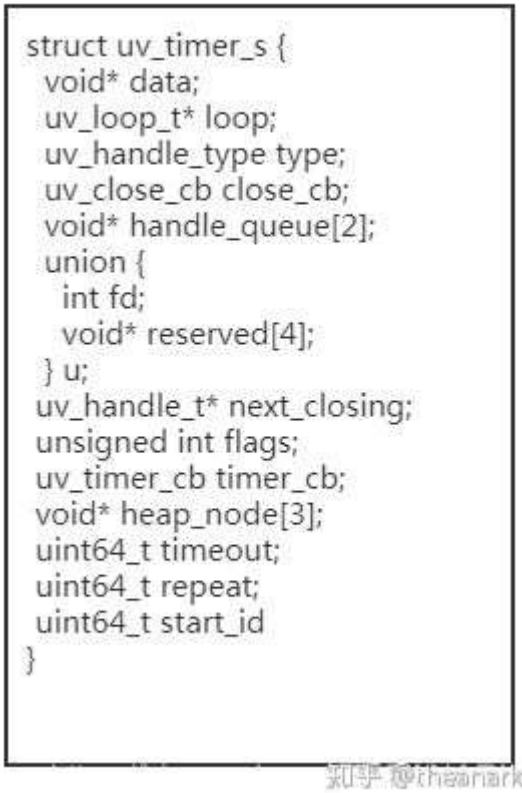
async io 也是。这就是 poll io 的整个过程。最后看一下 epoll_wait 阻塞时间的计算规则。

```
// 计算 epoll 使用的 timeout
int uv_backend_timeout(const uv_loop_t* loop) {
  // 下面几种情况下返回 0，即不阻塞在 epoll_wait
  if (loop->stop_flag != 0)
    return 0;
  // 没有东西需要处理，则不需要阻塞 poll io 阶段
  if (!uv__has_active_handles(loop) && !uv__has_active_reqs(loop))
    return 0;
  // idle 阶段有任务，不阻塞，尽快返回直接 idle 任务
  if (!QUEUE_EMPTY(&loop->idle_handles))
    return 0;
  // 同上
  if (!QUEUE_EMPTY(&loop->pending_queue))
    return 0;
  // 同上
  if (loop->closing_handles)
    return 0;
  // 返回下一个最早过期的时间，即最早超时的节点
  return uv__next_timeout(loop);
```

```
}
```

3.2.3 事件循环之定时器

libuv 中，定时器是以最小堆实现的。即最快过期的节点是根节点。我看看看定时器的数据结构。



看一下定时器的使用。

```
int main()
  v_timer_t once;
  uv_timer_init(uv_default_loop(), &once);
  uv_timer_start(&once, once_cb, 10, 0);
```

```
  uv_run(uv_default_loop(), UV_RUN_DEFAULT);
  return 0;
}
```

我们从 uv_timer_init 函数开始分析。

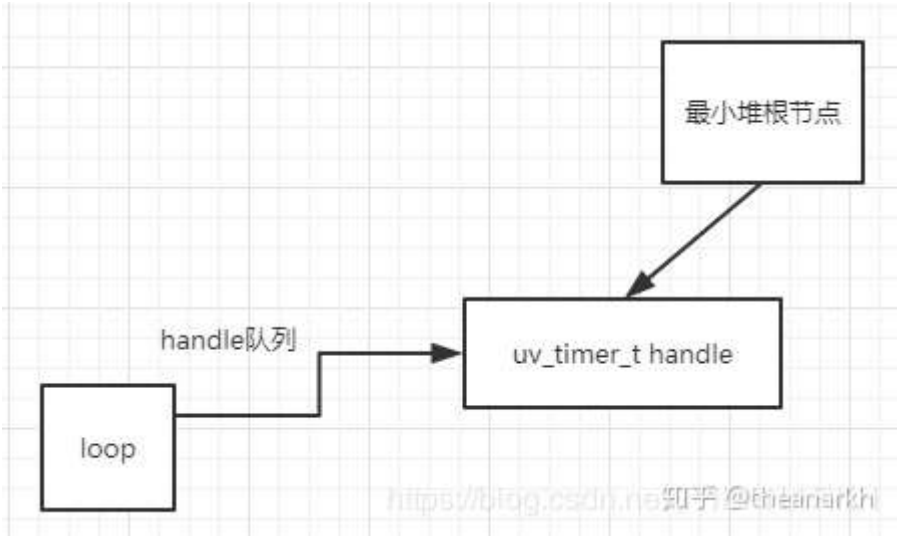
```
// 初始化 uv_timer_t 结构体
int uv_timer_init(uv_loop_t* loop, uv_timer_t* handle) {
  uv__handle_init(loop, (uv_handle_t*)handle, UV_TIMER);
  handle->timer_cb = NULL;
  handle->repeat = 0;
  return 0;
}
```

init 函数和其他阶段的 init 函数一样，初始化 handle 和私有的一些字段。接着我们看 start 函数。该函数是启动一个定时器（省略部分代码）。

```
// 启动一个计时器
int uv_timer_start(
    uv_timer_t* handle,
    uv_timer_cb cb,
    uint64_t timeout,
    uint64_t repeat
 ) {
```

```
  uint64_t clamped_timeout;
  // 重新执行 start 的时候先把之前的停掉
  if (uv__is_active(handle))
    uv_timer_stop(handle);
  // 超时时间，为绝对值
  clamped_timeout = handle->loop->time + timeout;
  // 初始化回调，超时时间，是否重复计时，赋予一个独立无二的 id
  handle->timer_cb = cb;
  handle->timeout = clamped_timeout;
  handle->repeat = repeat;
  /* start_id is the second index to be compared in uv__timer_cmp() */
  handle->start_id = handle->loop->timer_counter++;
  // 插入最小堆
  heap_insert(timer_heap(handle->loop),
              (struct heap_node*) &handle->heap_node,
              timer_less_than);
  // 激活该 handle
  uv__handle_start(handle);
  return 0;
}
```

start 函数首先初始化 handle 里的某些字段，包括超时回调，是否重复启动定时器、超时的绝对时间等。接着把 handle 节点插入到最小堆中。最后给这个 handle 打上标记，激活这个 handle。这时候的结构体如下。



这时候到了事件循环的 timer 阶段。

```
// 找出已经超时的节点，并且执行里面的回调
void uv__run_timers(uv_loop_t* loop) {
  struct heap_node* heap_node;
  uv_timer_t* handle;

  for (;;) {
    heap_node = heap_min(timer_heap(loop));
    if (heap_node == NULL)
      break;
```

```
    handle = container_of(heap_node, uv_timer_t, heap_node);
    // 如果当前节点的时间大于当前时间则返回，说明后面的节点也没有超时
    if (handle->timeout > loop->time)
      break;
    // 移除该计时器节点，重新插入最小堆，如果设置了 repeat 的话
    uv_timer_stop(handle);
    uv_timer_again(handle);
    // 执行超时回调
    handle->timer_cb(handle);
  }
}
```

libuv 在每次事件循环开始的时候都会缓存当前的时间，在整个一轮的事件循环中，使用的都是这个缓存的时间。缓存了当前最新的时间后，就执行 uv__run_timers，该函数的逻辑很明了，就是遍历最小堆，找出当前超时的节点。因为堆的性质是父节点肯定比孩子小。所以如果找到一个节点，他没有超时，则后面的节点也不会超时。对于超时的节点就知道他的回调。执行完回调后，还有两个关键的操作。第一就是 stop，第二就是 again。

```
// 停止一个计时器
int uv_timer_stop(uv_timer_t* handle) {
  if (!uv__is_active(handle))
    return 0;
```

```
  // 从最小堆中移除该计时器节点
  heap_remove(timer_heap(handle->loop),
              (struct heap_node*) &handle->heap_node,
              timer_less_than);
  // 清除激活状态和 handle 的 active 数减一
  uv__handle_stop(handle);
  return 0;
}
```

stop 的逻辑很简单，其实就是把 handle 从二叉堆中删除。并且取消激活状态。那么 againt 又是什么呢？again 是为了支持 setInterval 这种场景。

```
// 重新启动一个计时器，需要设置 repeat 标记
int uv_timer_again(uv_timer_t* handle) {
  // 如果设置了 repeat 标记说明计时器是需要重复触发的
  if (handle->repeat) {
    // 先把旧的计时器节点从最小堆中移除，然后再重新开启一个计时器
    uv_timer_stop(handle);
    uv_timer_start(handle, handle->timer_cb, handle->repeat,
handle->repeat);
  }

  return 0;
```

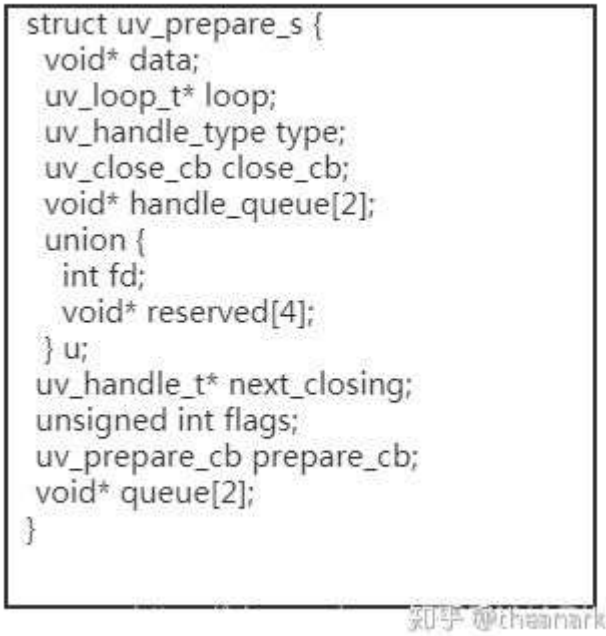
```
}
```

如果 handle 设置了 repeat 标记，则该 handle 在超时后，每 repeat 的时间后，就会继续执行超时回调。对于 setInterval，就是超时时间是 x，每 x 的时间后，执行回调。这就是 nodejs 里定时器的底层原理。但 nodejs 不是每次调 setTimeout 的时候都往最小堆插入一个节点。nodejs 里，只有一个关于 uv_timer_s 的 handle。他在 js 层维护了一个数据结构，每次计算出最早到期的节点，然后修改 handle 的超时时间。具体原理在之前的一篇文章已经分析过。

timer 阶段和 poll io 阶段也有一些联系，因为 poll io 可能会导致主线程阻塞，为了保证主线程可以尽快执行定时器的回调，poll io 不能一直阻塞，所以这时候，阻塞的时长就是最快到期的定时器节点的时长。

3.2.4 事件循环之 prepare,check,idle

prepare 是 libuv 事件循环中属于比较简单的一个阶段。我们知道 libuv 中分为 handle 和 request。而 prepare 阶段的任务是属于 handle。我们看一下他的定义。



下面我们看看怎么使用它

```
void prep_cb(uv_prepare_t *handle) {
    printf("Prep callback\n");
}

int main() {
    uv_prepare_t prep;
    // uv_default_loop 是 libuv 事件循环的核心结构体
    uv_prepare_init(uv_default_loop(), &prep);
    uv_prepare_start(&prep, prep_cb);
    uv_run(uv_default_loop(), UV_RUN_DEFAULT);
    return 0;
```

```
}
```

执行 main 函数，libuv 就会在 prepare 阶段执行回调 prep_cb。我们分析一下这个过程。

```
int uv_prepare_init(uv_loop_t* loop, uv_prepare_t* handle) {
    uv__handle_init(loop, (uv_handle_t*)handle, UV_PREPARE);
    handle->prepare_cb = NULL;
    return 0;
  }
```

1 uv__handle_init 是初始化 libuv 中 handle 的一个通用函数，他主要做了下面几个事情。

1 初始化 handle 的类型，所属 loop

2 打上 UV_HANDLE_REF，该标记影响事件循环的退出和 poll io 阶段超时时间的计算。具体在 start 函数的时候分析。

3 handle 插入 loop->handle_queue 队列的队尾，每个 handle 在 init 的时候都会插入 libuv 的 handle 队列。

2 初始化 prepare 节点的回调。

init 函数主要是做一些初始化操作。我们继续要看 start 函数。

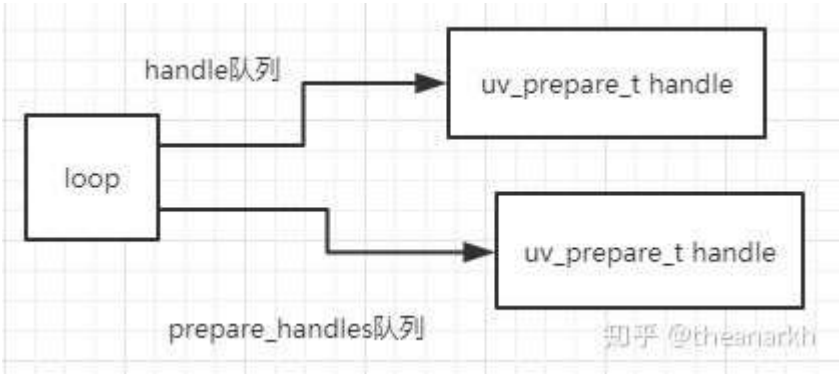
```
int uv_prepare_start(uv_prepare_t* handle, uv_prepare_cb cb) {
```

```
        // 如果已经执行过 start 函数则直接返回
    if (uv__is_active(handle)) return 0;
    if (cb == NULL) return UV_EINVAL;
    QUEUE_INSERT_HEAD(&handle->loop->prepare_handles,
&handle->queue);
    handle->prepare_cb = cb;
    uv__handle_start(handle);
    return 0;
  }
```

1 设置回调，把 handle 插入 loop 中的 prepare_handles 队列，prepare_handles 保存 prepare 阶段的任务。在事件循环的 prepare 阶段会逐个执行里面的节点的回调。

2 设置 UV_HANDLE_ACTIVE 标记位，如果这 handle 还打了 UV_HANDLE_REF 标记(在 init 阶段设置的）,则事件循环中的活 handle 数加一。UV_HANDLE_ACTIVE 标记这个 handle 是活的，影响事件循环的退出和 poll io 阶段超时时间的计算。有活的 handle 的话，libuv 如果运行在默认模式下，则不会退出，如果是其他模式，会退出。

执行完 start 函数，libuv 的结构体大概如下。



然后我们看看 libuv 在事件循环的 prepare 阶段是如何处理的。

```
void uv__run_prepare(uv_loop_t* loop) {
    uv_prepare_t* h;
    QUEUE queue;
    QUEUE* q;

    /*
        把该类型对应的队列中所有节点摘下来挂载到 queue 变量，
        相当于清空 prepare_handles 队列，因为如果直接遍历
        prepare_handles 队列，在执行回调的时候一直往 prepare_handles
        队列加节点，会导致下面的 while 循环无法退出。
        先移除的话，新插入的节点在下一轮事件循环才会被处理。
    */
     QUEUE_MOVE(&loop->prepare_handles, &queue);
    // 遍历队列，执行每个节点里面的函数
    while (!QUEUE_EMPTY(&queue)) {
```

```
        // 取下当前待处理的节点，即队列的头
        q = QUEUE_HEAD(&queue);
        /*
           取得该节点对应的整个结构体的基地址，
           即通过结构体成员取得结构体首地址
        */
        h = QUEUE_DATA(q, uv_prepare_t, queue);
        // 把该节点移出当前队列
        QUEUE_REMOVE(q);
       // 重新插入原来的队列
        QUEUE_INSERT_TAIL(&loop->prepare_handles, q);
       // 执行回调函数
        h->prepare_cb(h);
    }
}
```

run 函数的逻辑很明了，就是逐个执行 prepare_handles 队列的节点。我们回顾一开始的测试代码。因为他设置了 libuv 的运行模式是默认模式。又因为有或者的 handle（prepare 节点），所以他是不会退出的。他会一直执行回调。那如果我们要退出怎么办呢？或者说不要执行 prepare 队列的某个节点了。我们只需要 stop 一下就可以了。

```
int uv_prepare_stop(uv_prepare_t* handle) {
```

```
    if (!uv__is_active(handle)) return 0;
    // 把 handle 从 prepare 队列中移除，但是还挂载到 handle_queue 中
    QUEUE_REMOVE(&handle->queue);
  // 清除 active 标记位并且减去 loop 中 handle 的 active 数
    uv__handle_stop(handle);
    return 0;
}
```

stop 函数和 start 函数是相反的作用 ,就不分析了。这就是 nodejs 中 prepare 阶段的过程。

3.3 主进程和子进程/线程的通信

其他进程/线程和主进程的通信是使用 uv_async_t 结构体实现的。Libuv 使用 loop->async_handles 记录所有的 uv_async_t 结构体，使用 loop->async_io_watcher 作为所有 uv_async_t 结构体的 io 观察者。即 loop-> async_handles 队列上所有的 handle 都是共享 async_io_watcher 这个 io 观察者。第一次插入一个 uv_async_t 结构体到 async_handle 队列时，会初始化 io 观察者。如果再次注册一个 async_handle，只会在 loop->async_handle 队列和 handle 队列插入一个节点 ,而不是新增一个 io 观察者。

我们看一下 uv_async_init 的实现。

```
int uv_async_init(
```

```
uv_loop_t* loop,
uv_async_t* handle,
uv_async_cb async_cb
) {
  int err;
  // 给 libuv 注册一个用于异步通信的 io 观察者
  err = uv__async_start(loop);
  if (err)
    return err;
  // 设置相关字段，给 libuv 插入一个 async_handle
  uv__handle_init(loop, (uv_handle_t*)handle, UV_ASYNC);
  handle->async_cb = async_cb;
  // 标记是否有任务完成了
  handle->pending = 0;
  // 插入 async 队列，poll io 阶段判断是否有任务与完成
  QUEUE_INSERT_TAIL(&loop->async_handles, &handle->queue);
  // 激活 handle 为 active 状态
  uv__handle_start(handle);

  return 0;
}
```

由上面的代码，我们看到 uv_async_init 的逻辑非常简单，就是初始化 async handle 的一些字段。然后把 handle 插入 async_handle 队列中。不过我们发现还有个 uv__async_start 函数。我们看看他的代码。

```
/*
  初始化异步通信的 io 观察者
*/
static int uv__async_start(uv_loop_t* loop) {
  int pipefd[2];
  int err;
  /*
    因为 libuv 在初始化的时候会主动注册一个
    用于主线程和子线程通信的 async handle。
    从而初始化了 async_io_watcher。所以如果后续
    再注册 async handle，则不需要处理了。

    父子线程通信时，libuv 是优先使用 eventfd，如果不支持会回退到匿名管道。
    如果是匿名管道
      fd 是管道的读端，loop->async_wfd 是管道的写端
    如果是 eventfd
      fd 是读端也是写端。async_wfd 是-1
```

```
   所以这里判断 loop->async_io_watcher.fd 而不是 async_wfd 的值
*/
if (loop->async_io_watcher.fd != -1)
   return 0;
// 获取一个用于进程间通信的 fd
err = uv__async_eventfd();
// 成功则保存起来，不支持则使用管道通信作为进程间通信
if (err >= 0) {
   pipefd[0] = err;
   pipefd[1] = -1;
}
else if (err == UV_ENOSYS) {
   err = uv__make_pipe(pipefd, UV__F_NONBLOCK);
   // 通过打开进程在 proc 文件系统的
   #if defined(__linux__)
        if (err == 0) {
           char buf[32];
           int fd;
           /*
                  /proc/self/fd/目录下为当前进程所有打开的文件描述符。
                   通过这种方式可以使用一个文件描述符得到读写效果
           */
```

```
            snprintf(buf, sizeof(buf), "/proc/self/fd/%d", pipefd[0]);
            // 通过 fd 就可以实现对管道的读写
            fd = uv__open_cloexec(buf, O_RDWR);
            if (fd >= 0) {
              // 关掉旧的
              uv__close(pipefd[0]);
              uv__close(pipefd[1]);
              // 赋值新的
              pipefd[0] = fd;
              pipefd[1] = fd;
            }
          }
    #endif
  }
  if (err < 0)
    return err;
  // 初始化 io 观察者 async_io_watcher
  uv__io_init(&loop->async_io_watcher, uv__async_io, pipefd[0]);
  // 注册 io 观察者到 loop 里，并注册需要监听的事件 POLLIN，读
  uv__io_start(loop, &loop->async_io_watcher, POLLIN);
  // 用于主线程和子线程通信的 fd，管道的写端，子线程使用
  loop->async_wfd = pipefd[1];
```

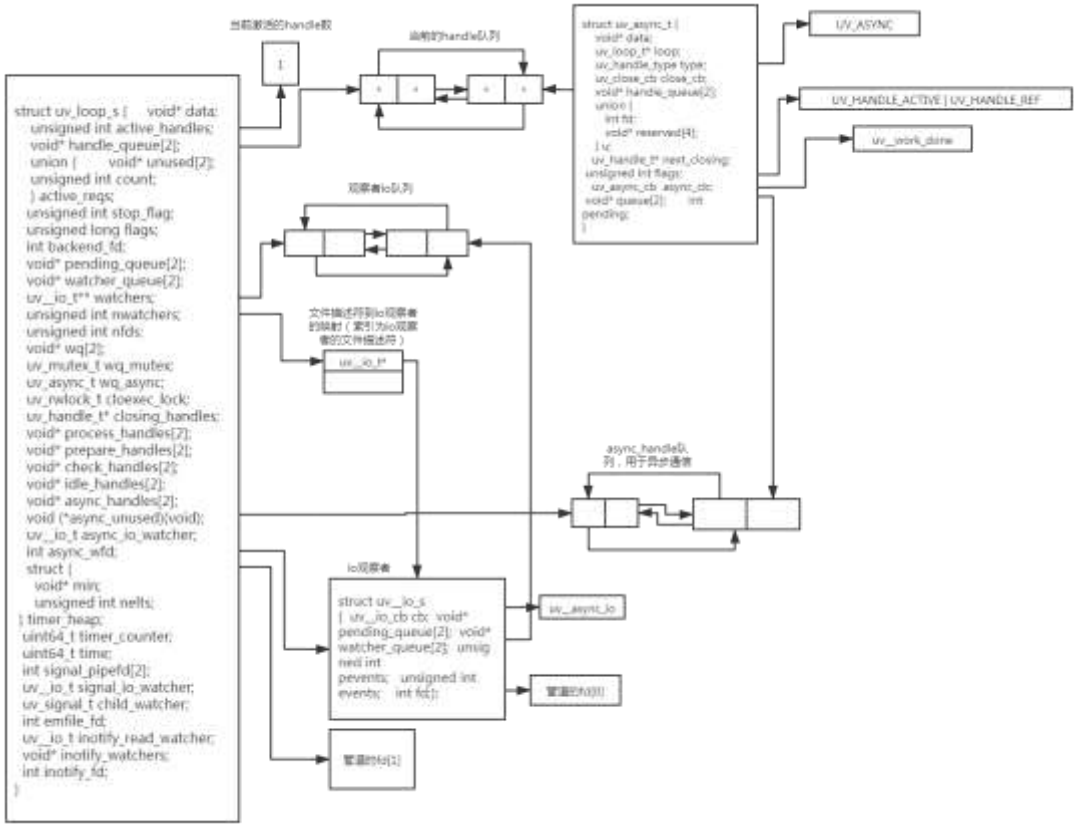
```
return 0;
```

uv__async_start 只会执行一次。主要逻辑是

1 申请用于通信的文件描述符，然后把读端和回调封装到 loop->async_io_watcher，写端保存在 loop->async_wfd。

2 插入 loop 的 io 观察者队列并建立 fd（io 观察者中的 fd）到 io 观察者的映射关系。

插入第一个 async_handle 时架构图如下



根据上面的代码，如果该 io 观察者有事件触发的时候，就会执行 uv__async_io。我们首先看一下当条件满足时，如何触发事件。

```
// 通知主线程有任务完成
int uv_async_send(uv_async_t* handle) {
  /* Do a cheap read first. */
  if (ACCESS_ONCE(int, handle->pending) != 0)
    return 0;
    /*
        设置 async handle 的 pending 标记
        如果 pending 是 0，则设置为 1，返回 0，如果是 1 则返回 1，
        所以同一个 handle 如果多次调用该函数是会被合并的
  */
  if (cmpxchgi(&handle->pending, 0, 1) == 0)
        // 设置 io 观察者有事件触发
    uv__async_send(handle->loop);
  return 0;
}
```

该函数最终会调用 uv__async_send。

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

```
  buf = "";
  len = 1;
  // 用于异步通信的管道的写端
  fd = loop->async_wfd;
#if defined(__linux__)
  // 说明用的是 eventfd 而不是管道
  if (fd == -1) {
    static const uint64_t val = 1;
    buf = &val;
    len = sizeof(val);
    // 见 uv__async_start
    fd = loop->async_io_watcher.fd;  /* eventfd */
  }
#endif
  // 通知读端
  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();
}
```

最后调用 write 函数往写端写入信息。主线程在 poll io 阶段就会发现有事件触发。我们继续来看一下事件触发后执行的回调。

```
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;
  // 用于异步通信的 io 观察者
  assert(w == &loop->async_io_watcher);
  for (;;) {
    // 判断通信内容
    r = read(w->fd, buf, sizeof(buf));
    // 如果数据大于 buf 的长度，接着读，清空这一轮写入的数据
    if (r == sizeof(buf))
      continue;
    // 不等于-1，说明读成功，失败的时候返回-1，errno 是错误码
```

```
    if (r != -1)
      break;
    if (errno == EAGAIN || errno == EWOULDBLOCK)
      break;
    // 被信号中断，继续读
    if (errno == EINTR)
      continue;
    // 出错，发送 abort 信号
    abort();
  }
  // 把 async_handles 队列里的所有节点都移到 queue 变量中
  QUEUE_MOVE(&loop->async_handles, &queue);
  while (!QUEUE_EMPTY(&queue)) {
    // 逐个取出节点
    q = QUEUE_HEAD(&queue);
    // 根据结构体字段获取结构体首地址
    h = QUEUE_DATA(q, uv_async_t, queue);
    // 从队列中移除该节点
    QUEUE_REMOVE(q);
    // 重新插入 async_handles 队列，等待下次事件
    QUEUE_INSERT_TAIL(&loop->async_handles, q);
    /*
```

```
        将第一个参数和第二个参数进行比较，如果相等，
        则将第三参数写入第一个参数，返回第二个参数的值，
        如果不相等，则返回第一个参数的值。
      */
       /*
          判断哪些 async 被触发了。pending 在 uv_async_send
          里设置成 1，如果 pending 等于 1，则清 0，返回 1.如果
          pending 等于 0，则返回 0
      */
      if (cmpxchgi(&h->pending, 1, 0) == 0)
        continue;
      if (h->async_cb == NULL)
        continue;
      // 执行上层回调
      h->async_cb(h);
    }
}
```

uv__async_io 会遍历 loop->async_handles 队里中所有的 uv_async_t。然后判断该 uv_async_t 是否有事件触发（通过 uv_async_t->pending 字段）。如果有的话，则执行该 uv_async_t 对应的回调。

3.4 线程池

Libuv 是基于事件驱动的异步库。对于耗时的操作。如果在 Libuv 的主循环里执行的话，就会阻塞后面的任务执行。所以 Libuv 里维护了一个线程池。他负责处理 Libuv 中耗时的操作，比如文件 io、dns、用户自定义的耗时任务（文件 io 因为存在跨平台兼容的问题。无法很好地在事件驱动模块实现异步 io）。我们先看线程池的初始化（在 libuv 初始化时）然后再看他的使用。

```
static void init_threads(void) {
  unsigned int i;
  const char* val;
  uv_sem_t sem;
  // 默认线程数 4 个，static uv_thread_t default_threads[4];
  nthreads = ARRAY_SIZE(default_threads);
  // 判断用户是否在环境变量中设置了线程数，是的话取用户定义的
  val = getenv("UV_THREADPOOL_SIZE");
  if (val != NULL)
    nthreads = atoi(val);
  if (nthreads == 0)
    nthreads = 1;
  // #define MAX_THREADPOOL_SIZE 128 最多 128 个线程
  if (nthreads > MAX_THREADPOOL_SIZE)
    nthreads = MAX_THREADPOOL_SIZE;
```

```
threads = default_threads;
// 超过默认大小，重新分配内存
if (nthreads > ARRAY_SIZE(default_threads)) {
  threads = uv__malloc(nthreads * sizeof(threads[0]));
  // 分配内存失败，回退到默认
  if (threads == NULL) {
    nthreads = ARRAY_SIZE(default_threads);
    threads = default_threads;
  }
}
// 初始化条件变量
if (uv_cond_init(&cond))
  abort();
// 初始化互斥变量
if (uv_mutex_init(&mutex))
  abort();

// 初始化三个队列
QUEUE_INIT(&wq);
QUEUE_INIT(&slow_io_pending_wq);
QUEUE_INIT(&run_slow_work_message);
```

```
  // 初始化信号量变量，值为 0
  if (uv_sem_init(&sem, 0))
    abort();
  // 创建多个线程，工作函数为 worker，sem 为 worker 入参
  for (i = 0; i < nthreads; i++)
    if (uv_thread_create(threads + i, worker, &sem))
      abort();
  // 为 0 则阻塞，非 0 则减一，这里等待所有线程启动成功再往下执行
  for (i = 0; i < nthreads; i++)
    uv_sem_wait(&sem);

  uv_sem_destroy(&sem);
}
```

线程池的初始化主要是初始化一些数据结构，然后创建多个线程。接着在每个线程里执行 worker 函数。worker 是消费者，在分析消费者之前，我们先看一下生产者的逻辑。

```
// 给线程池提交一个任务
void uv__work_submit(uv_loop_t* loop,
                     struct uv__work* w,
                     enum uv__work_kind kind,
                     void (*work)(struct uv__work* w),
```

```
                                void (*done)(struct uv__work* w, int status)) {
 /*
      保证已经初始化线程，并只执行一次，所以线程池是在提交第一个
      任务的时候才被初始化
  */
  uv_once(&once, init_once);
  w->loop = loop;
  w->work = work;
  w->done = done;
  post(&w->wq, kind);
}
```

这里把业务相关的函数和任务完成后的回调函数封装到 uv__work 结构体中。uv__work 结构定义如下。

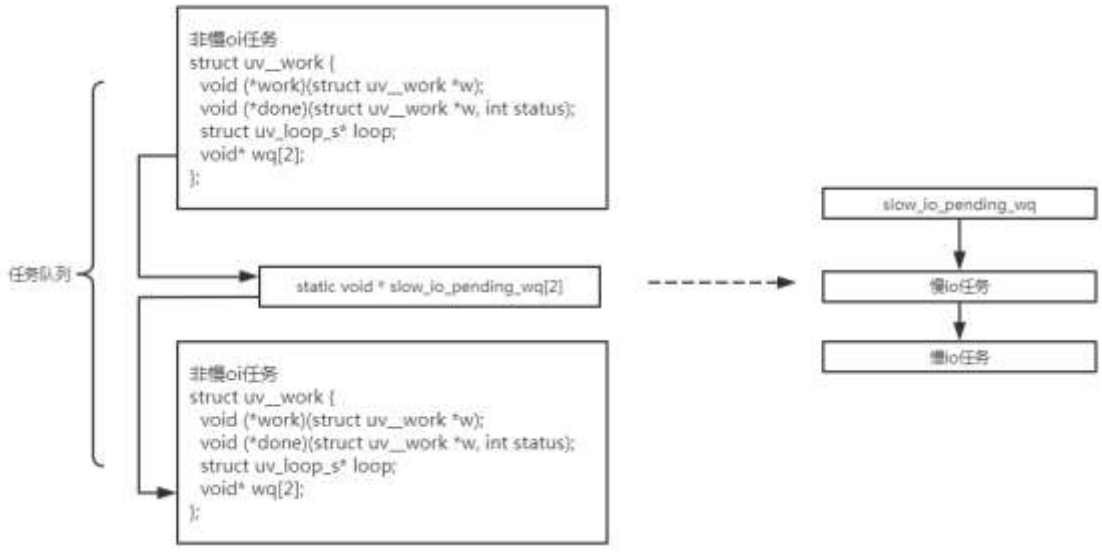
```
struct uv__work {
  void (*work)(struct uv__work *w);
  void (*done)(struct uv__work *w, int status);
  struct uv_loop_s* loop;
  void* wq[2];
};
```

然后调 post 往线程池的队列中加入一个新的任务。Libuv 把任务分为三种类型，慢 io（dns 解析）、快 io（文件操作）、cpu 密集型等，kind 就是说明任务的类型的。我们接着看 post 函数。

```
// 把任务插入队列等待线程处理
static void post(QUEUE* q, enum uv__work_kind kind) {
  // 加锁访问任务队列，因为这个队列是线程池共享的
  uv_mutex_lock(&mutex);
  // 类型是慢 IO
  if (kind == UV__WORK_SLOW_IO) {
    /*
        插入慢 IO 对应的队列，llibuv 这个版本把任务分为几种类型，
        对于慢 io 类型的任务，libuv 是往任务队列里面插入一个特殊的节点
        run_slow_work_message，然后用 slow_io_pending_wq 维护了一个慢 io 任务的队列，
        当处理到 run_slow_work_message 这个节点的时候，libuv 会从 slow_io_pending_wq
        队列里逐个取出任务节点来执行。
    */
    QUEUE_INSERT_TAIL(&slow_io_pending_wq, q);
    /*
      有慢 IO 任务的时候，需要给主队列 wq 插入一个消息节点 run_slow_work_message,
      说明有慢 IO 任务，所以如果 run_slow_work_message 是空，说明还没有插入主队列。
      需要进行 q = &run_slow_work_message;赋值，然后把 run_slow_work_message 插入
      主队列。如果 run_slow_work_message 非空，说明已经插入线程池的任务队列了。
     解锁然后直接返回。
    */
    if (!QUEUE_EMPTY(&run_slow_work_message)) {
```

```
        uv_mutex_unlock(&mutex);
        return;
      }
      // 说明 run_slow_work_message 还没有插入队列，准备插入队列
      q = &run_slow_work_message;
    }
    // 把节点插入主队列，可能是慢 IO 消息节点或者一般任务
    QUEUE_INSERT_TAIL(&wq, q);
    // 有空闲线程则唤醒他，如果大家都在忙，则等到他忙完后就会重新判断是否还有新任务
    if (idle_threads > 0)
      uv_cond_signal(&cond);
    uv_mutex_unlock(&mutex);
}
```

这就是 libuv 中线程池的生产者逻辑。架构如下。



除了上面提到的，libuv 还提供了另外一种生产者。即 uv_queue_work 函数。他只针对 cpu 密集型的。从实现来看，他和第一种生产模式的区别是，通过 uv_queue_work 提交的任务，是对应一个 request 的。如果该 request 对应的任务没有执行完，则事件循环不会退出。而通过 uv__work_submit 方式提交的任务就算没有执行完，也不会影响事件循环的退出。下面我们看 uv_queue_work 的实现。

```
int uv_queue_work(uv_loop_t* loop,
                  uv_work_t* req,
                  uv_work_cb work_cb,
                  uv_after_work_cb after_work_cb) {
  if (work_cb == NULL)
    return UV_EINVAL;

  uv__req_init(loop, req, UV_WORK);
  req->loop = loop;
  req->work_cb = work_cb;
  req->after_work_cb = after_work_cb;
  uv__work_submit(loop,
                  &req->work_req,
                  UV__WORK_CPU,
                  uv__queue_work,
                  uv__queue_done);
  return 0;
```

```
}
```

uv_queue_work 函数其实也没有太多的逻辑 ,他保存用户的工作函数和回调到 request 中。然后提交任务，然后把 uv__queue_work 和 uv__queue_done 封装到 uv__work 中，接着提交任务。所以当这个任务被执行的时候。他会执行工作函数 uv__queue_work。

```
static void uv__queue_work(struct uv__work* w) {
  // 通过结构体某字段拿到结构体地址
  uv_work_t* req = container_of(w, uv_work_t, work_req);
  req->work_cb(req);
}
```

我们看到 uv__queue_work 其实就是对用户定义的任务函数进行了封装。这时候我们可以猜到，uv__queue_done 也只是对用户回调的简单封装，即他会执行用户的回调。至此，我们分析完了 libuv 中，线程池的两种生产任务的方式。下面我们开始分析消费者。消费者由 worker 函数实现。

```
static void worker(void* arg) {
  struct uv__work* w;
  QUEUE* q;
  int is_slow_work;
  // 线程启动成功
  uv_sem_post((uv_sem_t*) arg);
  arg = NULL;
  // 加锁互斥访问任务队列
```

```
uv_mutex_lock(&mutex);
for (;;) {
  /*
    1 队列为空，
    2 队列不为空，但是队列里只有慢 IO 任务且正在执行的慢 IO 任务个数达到阈值
    则空闲线程加一，防止慢 IO 占用过多线程，导致其他快的任务无法得到执行
  */
  while (QUEUE_EMPTY(&wq) ||
         (QUEUE_HEAD(&wq) == &run_slow_work_message &&
          QUEUE_NEXT(&run_slow_work_message) == &wq &&
          slow_io_work_running >= slow_work_thread_threshold())) {
    idle_threads += 1;
    // 阻塞，等待唤醒
    uv_cond_wait(&cond, &mutex);
    // 被唤醒，开始干活，空闲线程数减一
    idle_threads -= 1;
  }
  // 取出头结点，头指点可能是退出消息、慢 IO，一般请求
  q = QUEUE_HEAD(&wq);
  // 如果头结点是退出消息，则结束线程
  if (q == &exit_message) {
    // 唤醒其他因为没有任务正阻塞等待任务的线程，告诉他们准备退出
```

```
    uv_cond_signal(&cond);
    uv_mutex_unlock(&mutex);
    break;
  }
  // 移除节点
  QUEUE_REMOVE(q);
  // 重置前后指针
  QUEUE_INIT(q);
  is_slow_work = 0;
  /*
      如果当前节点等于慢 IO 节点，上面的 while 只判断了是不是只有慢 io 任务且达到
      阈值，这里是任务队列里肯定有非慢 io 任务，可能有慢 io，如果有慢 io 并且正在
      执行的个数达到阈值，则先不处理该慢 io 任务，继续判断是否还有非慢 io 任务可
      执行。
  */

  if (q == &run_slow_work_message) {
    // 遇到阈值，重新入队
    if (slow_io_work_running >= slow_work_thread_threshold()) {
      QUEUE_INSERT_TAIL(&wq, q);
      continue;
    }
```

```
    // 没有慢 IO 任务则继续
    if (QUEUE_EMPTY(&slow_io_pending_wq))
      continue;
    // 有慢 io，开始处理慢 IO 任务
    is_slow_work = 1;
    // 正在处理慢 IO 任务的个数累加，用于其他线程判断慢 IO 任务个数是否达到阈值
    slow_io_work_running++;
    // 摘下一个慢 io 任务
    q = QUEUE_HEAD(&slow_io_pending_wq);
    QUEUE_REMOVE(q);
    QUEUE_INIT(q);
    /*
        取出一个任务后，如果还有慢 IO 任务则把慢 IO 标记节点重新入队，
        表示还有慢 IO 任务，因为上面把该标记节点出队了
    */
    if (!QUEUE_EMPTY(&slow_io_pending_wq)) {
      QUEUE_INSERT_TAIL(&wq, &run_slow_work_message);
      // 有空闲线程则唤醒他，因为还有任务处理
      if (idle_threads > 0)
        uv_cond_signal(&cond);
    }
  }
```

```
    // 不需要操作队列了，尽快释放锁
    uv_mutex_unlock(&mutex);
    // q 是慢 IO 或者一般任务
    w = QUEUE_DATA(q, struct uv__work, wq);
    // 执行业务的任务函数，该函数一般会阻塞
    w->work(w);
    // 准备操作 loop 的任务完成队列，加锁
    uv_mutex_lock(&w->loop->wq_mutex);
    // 置空说明指向完了，不能被取消了，见 cancel 逻辑
    w->work = NULL;
    // 执行完任务,插入到 loop 的 wq 队列,在 uv__work_done 的时候会执行该队列的节点
    QUEUE_INSERT_TAIL(&w->loop->wq, &w->wq);
    // 通知 loop 的 wq_async 节点
    uv_async_send(&w->loop->wq_async);
    uv_mutex_unlock(&w->loop->wq_mutex);
    // 为下一轮操作任务队列加锁
    uv_mutex_lock(&mutex);
    // 执行完慢 IO 任务，记录正在执行的慢 IO 个数变量减 1，上面加锁保证了互斥访问这个变量
    if (is_slow_work) {
      slow_io_work_running--;
    }
  }
```

```
}
```

我们看到消费者的逻辑似乎比较复杂，主要是把任务分为三种。并且对于慢 io 类型的任务，还限制了线程数。其余的逻辑和一般的线程池类型，就是互斥访问任务队列，然后取出节点执行，最后执行回调。不过 libuv 这里不是直接回调用户的函数。而是通知主进程。由主进程处理，我们看一下这块的逻辑。

一切要从 libuv 的初始化开始。

```
uv_default_loop();
```

该函数调用

```
uv_loop_init();
```

进行初始化。uv_loop_init 有以下代码。

```
uv_async_init(loop, &loop->wq_async, uv__work_done);
```

wq_async 是用于线程池和主线程通信的 async handle。他对应的回调是 uv__work_done。所以当一个线程池的线程任务完成时，通过 uv_async_send(&w->loop->wq_async)设置 loop->wq_async.pending = 1，然后通知 io 观察者。Libuv 在 poll io 阶段就会执行该 handle 对应的回调。该 io 观察者的回调是 uv__work_done 函数。那么我们就看看这个函数的逻辑。

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

```
  int err;
  // 通过结构体字段获得结构体首地址
  loop = container_of(handle, uv_loop_t, wq_async);
  // 准备处理队列，加锁
  uv_mutex_lock(&loop->wq_mutex);
  //
        把 loop->wq 队列的节点全部移到 wp 变量中，这样一来可以尽快释放锁
*/
  QUEUE_MOVE(&loop->wq, &wq);
  // 不需要使用了，解锁
  uv_mutex_unlock(&loop->wq_mutex);
  // wq 队列的节点来源是在线程的 worker 里插入
  while (!QUEUE_EMPTY(&wq)) {
    q = QUEUE_HEAD(&wq);
    QUEUE_REMOVE(q);

    w = container_of(q, struct uv__work, wq);
    err = (w->work == uv__cancelled) ? UV_ECANCELED : 0;
    // 执行回调
    w->done(w, err);
  }
}
```

逐个处理已完成的任务节点，执行回调。这就是整个消费者的逻辑。最后顺带提一下 w->work == uv__cancelled。这个处理的用处是为了支持取消一个任务。Libuv 提供了 uv__work_cancel 函数支持用户取消提交的任务。我们看一下他的逻辑。

```
static int uv__work_cancel(uv_loop_t* loop, uv_req_t* req, struct uv__work* w) {
  int cancelled;
  // 加锁，为了把节点移出队列
  uv_mutex_lock(&mutex);
  // 加锁，为了判断 w->wq 是否为空
  uv_mutex_lock(&w->loop->wq_mutex);
  /*
     w 在在任务队列中并且任务函数 work 不为空，则可取消，
     在 work 函数中，如果执行完了任务，会把 work 置 NULL，
     所以一个任务可以取消的前提是他还没执行完。或者说还没执行过
  */
  cancelled = !QUEUE_EMPTY(&w->wq) && w->work != NULL;
  // 从任务队列中删除该节点
  if (cancelled)
     QUEUE_REMOVE(&w->wq);

  uv_mutex_unlock(&w->loop->wq_mutex);
  uv_mutex_unlock(&mutex);
  // 不能取消
```

```
  if (!cancelled)
     return UV_EBUSY;
  // 重置回调函数
  w->work = uv__cancelled;

  uv_mutex_lock(&loop->wq_mutex);
   /*
      插入 loop 的 wq 队列，对于取消的动作，libuv 认为是任务执行完了。
      所以插入已完成的队列，不过他的回调是 uv__cancelled 函数，
      而不是用户设置的回调
   */
  QUEUE_INSERT_TAIL(&loop->wq, &w->wq);
  // 通知主线程有任务完成
  uv_async_send(&loop->wq_async);
  uv_mutex_unlock(&loop->wq_mutex);

  return 0;
}
```

3.5 dns

Libuv 提供了一个异步 dns 解析的能力。包括通过域名查询 ip 和 ip 查询域名两个功能。

Libuv 对操作系统提供的函数进行了封装，配合线程池和事件循环实现异步的能力。

```
// 通过域名找 ip
int uv_getaddrinfo(uv_loop_t* loop,
                   // 上层传进来的 req
                    uv_getaddrinfo_t* req,
                    // 解析完后的上层回调
                    uv_getaddrinfo_cb cb,
                    // 需要解析的名字
                    const char* hostname,
                    /*
                        查询的过滤条件：服务名。比如 http smtp。
                        也可以是一个端口。见下面注释
                     */
                    const char* service,
                    // 其他查询过滤条件
                    const struct addrinfo* hints) {

  size_t hostname_len;
  size_t service_len;
  size_t hints_len;
  size_t len;
  char* buf;
```

```
hostname_len = hostname ? strlen(hostname) + 1 : 0;

service_len = service ? strlen(service) + 1 : 0;

hints_len = hints ? sizeof(*hints) : 0;

buf = uv__malloc(hostname_len + service_len + hints_len);

uv__req_init(loop, req, UV_GETADDRINFO);

req->loop = loop;

// 设置请求的回调

req->cb = cb;

req->addrinfo = NULL;

req->hints = NULL;

req->service = NULL;

req->hostname = NULL;

req->retcode = 0;

len = 0;


if (hints) {

  req->hints = memcpy(buf + len, hints, sizeof(*hints));

  len += sizeof(*hints);

}


if (service) {

  req->service = memcpy(buf + len, service, service_len);
```

```
    len += service_len;
  }


  if (hostname)
    req->hostname = memcpy(buf + len, hostname, hostname_len);
  // 传了 cb 是异步
  if (cb) {
    uv__work_submit(loop,
                    &req->work_req,
                    UV__WORK_SLOW_IO,
                    uv__getaddrinfo_work,
                    uv__getaddrinfo_done);
    return 0;
  } else {
    // 阻塞式查询，然后执行回调
    uv__getaddrinfo_work(&req->work_req);
    uv__getaddrinfo_done(&req->work_req, 0);
    return req->retcode;
  }
}
```

我们看到这个函数首先是对一个 request 进行初始化，然后根据是否传了回调，决定走异步还是同步的模式。同步的方式比较简单，就是直接阻塞 libuv 事件循环，直到解析

完成。如果是异步，则给线程池提交一个慢 io 的任务。其中工作函数是 uv__getaddrinfo_work。回调是 uv__getaddrinfo_done。我们看一下这两个函数。

```
// 解析的工作函数
static void uv__getaddrinfo_work(struct uv__work* w) {
  uv_getaddrinfo_t* req;
  int err;
  // 根据结构体的字段获取结构体首地址
  req = container_of(w, uv_getaddrinfo_t, work_req);
  // 阻塞在这
  err = getaddrinfo(req->hostname, req->service, req->hints, &req->addrinfo);
  req->retcode = uv__getaddrinfo_translate_error(err);
}
```

工作函数主要是调用了操作系统提供的 getaddrinfo 去做解析。然后会导致阻塞。结果返回后，执行 uv__getaddrinfo_done。

```
// dns 解析完执行的函数
static void uv__getaddrinfo_done(struct uv__work* w, int status) {
  uv_getaddrinfo_t* req;


  req = container_of(w, uv_getaddrinfo_t, work_req);
  uv__req_unregister(req->loop, req);
  // 释放初始化时申请的内存
  if (req->hints)
```

```
    uv__free(req->hints);
  else if (req->service)
    uv__free(req->service);
  else if (req->hostname)
    uv__free(req->hostname);
  else
    assert(0);

  req->hints = NULL;
  req->service = NULL;
  req->hostname = NULL;
  // 解析请求被用户取消了
  if (status == UV_ECANCELED) {
    assert(req->retcode == 0);
    req->retcode = UV_EAI_CANCELED;
  }
  // 执行上层回调
  if (req->cb)
    req->cb(req, req->retcode, req->addrinfo);

}
```

3.6 unix 域

Unix 域一种进程间通信的方式，他类似 socket 通信，但是他是基于单主机的。可以说是单机上的 socket 通信。下面我们来看一下他的实现。在 libuv 中 ,unix 域用 uv_pipe_t 表示

首先初始化一个表示 unix 域的结构体 uv_pipe_t。

```
struct uv_pipe_s {
  // uv_handle_s 的字段
  void* data;
  //  所属事件循环
  uv_loop_t* loop;
  // handle 类型
  uv_handle_type type;
  //  关闭 handle 时的回调
  uv_close_cb close_cb;
  //  用于插入事件循环的 handle 队列
  void* handle_queue[2];
  union {
    int fd;
    void* reserved[4];
  } u;
  //  用于插入事件循环的 closing 阶段对应的队列
```

```
uv_handle_t* next_closing;
// 各种标记
unsigned int flags;
// 流拓展的字段
// 用户写入流的字节大小，流缓存用户的输入，然后等到可写的时候才做真正的写
size_t write_queue_size;
// 分配内存的函数，内存由用户定义，主要用来保存读取的数据
uv_alloc_cb alloc_cb;
// 读取数据的回调
uv_read_cb read_cb;
// 连接成功后，执行 connect_req 的回调（connect_req 在 uv__xxx_connect 中赋值）
uv_connect_t *connect_req;
/*
   关闭写端的时候，发送完缓存的数据，执行
   shutdown_req 的回调（shutdown_req 在 uv_shutdown 的时候赋值）
*/
uv_shutdown_t *shutdown_req;
// 流对应的 io 观察者，即文件描述符+一个文件描述符事件触发时执行的回调
uv__io_t io_watcher;
// 流缓存下来的，待写的数据
void* write_queue[2];
// 已经完成了数据写入的队列
```

```
  void* write_completed_queue[2];
  // 完成三次握手后，执行的回调
  uv_connection_cb connection_cb;
  // 操作流时出错码
  int delayed_error;
  // accept 返回的通信 socket 对应的文件描述符
  int accepted_fd;
  // 同上，用于缓存更多的通信 socket 对应的文件描述符
  void* queued_fds;
  // 标记管道是否能在进程间传递
  int ipc;
  // 用于 unix 域通信的文件路径
  const char* pipe_fname;
}
```

unix 域继承域 handle 和 stream。下面看一下他的具体实现逻辑。

```
int uv_pipe_init(uv_loop_t* loop, uv_pipe_t* handle, int ipc) {
  uv__stream_init(loop, (uv_stream_t*)handle, UV_NAMED_PIPE);
  handle->shutdown_req = NULL;
  handle->connect_req = NULL;
  handle->pipe_fname = NULL;
  handle->ipc = ipc;
  return 0;
```

```
}
```

uv_pipe_init 逻辑很简单，就是初始化 uv_pipe_t 结构体。刚才已经见过 uv_pipe_t 继承于 stream，uv__stream_init 就是初始化 stream（父类）的字段。文章开头说过，unix 域的实现类似 tcp 的实现。遵循网络 socket 编程那一套。服务端使用 bind ,listen 等函数启动服务。

```
// name 是 unix 路径名称
int uv_pipe_bind(uv_pipe_t* handle, const char* name) {
  struct sockaddr_un saddr;
  const char* pipe_fname;
  int sockfd;
  int err;


  pipe_fname = NULL;


  pipe_fname = uv__strdup(name);
  name = NULL;
  // unix 域套接字
  sockfd = uv__socket(AF_UNIX, SOCK_STREAM, 0);
  memset(&saddr, 0, sizeof saddr);
  strncpy(saddr.sun_path, pipe_fname, sizeof(saddr.sun_path) - 1);
  saddr.sun_path[sizeof(saddr.sun_path) - 1] = '\0';
  saddr.sun_family = AF_UNIX;
```

```
  // 绑定到路径，tcp 是绑定到 ip 和端口
  if (bind(sockfd, (struct sockaddr*)&saddr, sizeof saddr)) {
    // ...
  }

  // 已经绑定
  handle->flags |= UV_HANDLE_BOUND;
  handle->pipe_fname = pipe_fname;
  // 保存 socket fd，用于后面监听
  handle->io_watcher.fd = sockfd;
  return 0;
}
```

uv_pipe_bind 函数的逻辑也比较简单，就是类似 tcp 的 bind 行为。

1 申请一个 socket 套接字。

2 绑定 unix 域路径到 socket 中。

绑定了路径后，就可以调用 listen 函数开始监听。

```
int uv_pipe_listen(uv_pipe_t* handle, int backlog, uv_connection_cb cb) {
  if (uv__stream_fd(handle) == -1)
    return UV_EINVAL;
  // uv__stream_fd(handle)得到 bind 函数中获取的 socket
  if (listen(uv__stream_fd(handle), backlog))
    return UV__ERR(errno);
```

```
  // 保存回调，有进程调用 connect 的时候时触发，由 uv__server_io 函数触发
  handle->connection_cb = cb;
  // io 观察者的回调，有进程调用 connect 的时候时触发（io 观察者的 fd 在 init 函数里设置了）
  handle->io_watcher.cb = uv__server_io;
  // 注册 io 观察者到 libuv，等待连接，即读事件到来
  uv__io_start(handle->loop, &handle->io_watcher, POLLIN);
  return 0;
}
```

uv_pipe_listen 执行 listen 函数使得 socket 成为监听型的套接字。然后把 socket 对应的文件描述符和回调封装成 io 观察者。注册到 libuv。等到有读事件到来(有连接到来)。就会执行 uv__server_io 函数 ,摘下对应的客户端节点。最后执行 connection_cb 回调。这时候，使用 unix 域成功启动了一个服务。接下来就是看客户端的逻辑。

```
void uv_pipe_connect(uv_connect_t* req,
                     uv_pipe_t* handle,
                     const char* name,
                     uv_connect_cb cb) {
  struct sockaddr_un saddr;
  int new_sock;
  int err;
  int r;
  // 判断是否已经有 socket 了，没有的话需要申请一个，见下面
  new_sock = (uv__stream_fd(handle) == -1);
```

```
// 客户端还没有对应的 socket fd
if (new_sock) {
  err = uv__socket(AF_UNIX, SOCK_STREAM, 0);
  if (err < 0)
    goto out;
  // 保存 socket 对应的文件描述符到 io 观察者
  handle->io_watcher.fd = err;
}
// 需要连接的服务器信息。主要是 unix 域路径信息
memset(&saddr, 0, sizeof saddr);
strncpy(saddr.sun_path, name, sizeof(saddr.sun_path) - 1);
saddr.sun_path[sizeof(saddr.sun_path) - 1] = '\0';
saddr.sun_family = AF_UNIX;
// 连接服务器，unix 域路径是 name
do {
  r = connect(uv__stream_fd(handle),(struct sockaddr*)&saddr, sizeof saddr);
}
while (r == -1 && errno == EINTR);
// 忽略错误处理逻辑
err = 0;
// 设置 socket 的可读写属性
if (new_sock) {
```

```
    err = uv__stream_open((uv_stream_t*)handle,
                          uv__stream_fd(handle),
                          UV_HANDLE_READABLE | UV_HANDLE_WRITABLE);
  }
  // 把 io 观察者注册到 libuv，等到连接成功或者可以发送请求
  if (err == 0)
    uv__io_start(handle->loop, &handle->io_watcher, POLLIN | POLLOUT);

out:
  // 记录错误码，如果有的话
  handle->delayed_error = err;
  // 连接成功时的回调
  handle->connect_req = req;

  uv__req_init(handle->loop, req, UV_CONNECT);
  req->handle = (uv_stream_t*)handle;
  req->cb = cb;
  QUEUE_INIT(&req->queue);
  // 如果连接出错，在 pending 节点会执行 req 对应的回调。错误码是 delayed_error
  if (err)
    uv__io_feed(handle->loop, &handle->io_watcher);
}
```

Unix 域大致的流程和网络编程一样。分为服务端和客户端两面。libuv 在操作系统提供的 api 的基础上。和 libuv 的异步非阻塞结合。在 libuv 中为进程间提供了一种通信方式。下面看一下如何使用。

```
void remove_sock(int sig) {
    uv_fs_t req;
    // 删除 unix 域对应的路径
    uv_fs_unlink(loop, &req, PIPENAME, NULL);
    // 退出进程
    exit(0);
}
int main() {
    loop = uv_default_loop();
    uv_pipe_t server;
    uv_pipe_init(loop, &server, 0);
    // 注册 SIGINT 信号的信号处理函数是 remove_sock
    signal(SIGINT, remove_sock);

    int r;
    // 绑定 unix 路径到 socket
    if ((r = uv_pipe_bind(&server, PIPENAME))) {
        fprintf(stderr, "Bind error %s\n", uv_err_name(r));
        return 1;
```

```
    }
    /*
        把 unix 域对应的文件文件描述符设置为 listen 状态。
        开启监听请求的到来，连接的最大个数是 128。有连接时的回调是 on_new_connection
    */
    if ((r = uv_listen((uv_stream_t*) &server, 128, on_new_connection))) {
        fprintf(stderr, "Listen error %s\n", uv_err_name(r));
        return 2;
    }
    // 启动事件循环
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

对于了解网络编程的同学来说。上面的代码看起来会比较简单。所以就不具体分析。他执行完后就是启动了一个服务。同主机的进程可以访问（连接）他。之前说过 unix 域的实现和 tcp 的实现类似。都是基于连接的模式。服务器启动等待连接，客户端去连接。然后服务器逐个摘下连接的节点进行处理。我们从处理连接的函数 on_new_connection 开始分析整个流程。

```
// 有连接到来时的回调
void on_new_connection(uv_stream_t *server, int status) {
    // 有连接到来，申请一个结构体表示他
    uv_pipe_t *client = (uv_pipe_t*) malloc(sizeof(uv_pipe_t));
    uv_pipe_init(loop, client, 0);
```

```
    // 把 accept 返回的 fd 记录到 client，client 是用于和客户端通信的结构体
    if (uv_accept(server, (uv_stream_t*) client) == 0) {
        /*
            注册读事件，等待客户端发送信息过来,
            alloc_buffer 分配内存保存客户端的发送过来的信息,
            echo_read 是回调
        */
        uv_read_start((uv_stream_t*) client, alloc_buffer, echo_read);
    }
    else {
        uv_close((uv_handle_t*) client, NULL);
    }
}
```

分析 on_new_connection 之前，我们先看一下该函数的执行时机。该函数是在 uv__server_io 函数中被执行，而 uv__server_io 是在监听的 socket（即 listen 的那个）有可读事件时触发的回调。我们看看 uv__server_io 的部分逻辑。

```
// 有连接到来，进行 accept
    err = uv__accept(uv__stream_fd(stream));
    // 保存通信 socket 对应的文件描述符
    stream->accepted_fd = err;
    /*
```

```
        有连接，执行上层回调，connection_cb 一般会调用 uv_accept 消费 accepted_fd。
        然后重新注册等待可读事件
    */
    stream->connection_cb(stream, 0);
```

当有连接到来时，服务器调用 uv__accept 摘取一个连接节点（实现上，操作系统会返回一个文件描述符，作用类似一个 id）。然后把文件描述符保存到 accepted_fd 字段，接着执行 connection_cb 回调。就是我们设置的 on_new_connection。

uv__stream_fd(stream)是我们启动的服务器对应的文件描述符。stream 就是表示服务器的结构体。在 unix 域里，他实际上是一个 uv_pipe_s 结构体。uv_stream_s 是 uv_pipe_s 的父类。类似 c++的继承。

我们回头看一下 on_new_connection 的代码。主要逻辑如下。

1 申请一个 uv_pipe_t 结构体用于保存和客户端通信的信息。

2 执行 uv_accept

3 执行 uv_read_start 开始等待数据的到来，然后读取数据。

我们分析一下 2 和 3。我们看一下 uv_accept 的主要逻辑。

```
switch (client->type) {
    case UV_NAMED_PIPE:
```

```
        // 设置流的标记，保存文件描述符到流上
        uv__stream_open(
            client,server->accepted_fd,
            UV_HANDLE_READABLE | UV_HANDLE_WRITABLE
        );
}
```

uv_accept 中把刚才 accept 到的文件描述符保存到 client 中。这样我们后续就可以通过 client 和客户端通信。至于 uv_read_start，之前在 stream 的文章中已经分析过。就不再深入分析。我们主要分析 echo_read。echo_read 在客户端给服务器发送信息时被触发。

```
void echo_read(uv_stream_t *client, ssize_t nread, const uv_buf_t *buf) {
    // 有数据，则回写
    if (nread > 0) {
        write_req_t *req = (write_req_t*) malloc(sizeof(write_req_t));
        // 指向客户端发送过来的数据
        req->buf = uv_buf_init(buf->base, nread);
        // 回写给客户端，echo_write 是写成功后的回调
        uv_write((uv_write_t*) req, client, &req->buf, 1, echo_write);
        return;
    }
    // 没有数据了，关闭
```

```
    if (nread < 0) {
        if (nread != UV_EOF)
            fprintf(stderr, "Read error %s\n", uv_err_name(nread));
        // 销毁和客户端通信的结构体，即关闭通信
        uv_close((uv_handle_t*) client, NULL);
    }

    free(buf->base);
}
```

没有数据的时候，直接销毁和客户端通信的结构体和撤销结构体对应的读写事件。我们主要分析有数据时的处理逻辑。当有数据到来时，服务器调用 uv_write 对数据进行回写。我们看到 uv_write 的第二个参数是 client。即往 client 对应的文件描述符中写数据。也就是往客户端写。uv_write 的逻辑在 stream 中已经分析过，所以也不打算深入分析。主要逻辑就是在 client 对应的 stream 上写入数据，缓存起来，然后等待可写时，再写到对端。写完成后执行 echo_write 释放数据占据的内存。这就是使用 unix 域通信的整个过程。unix 域还有一个复杂的应用是涉及到传递描述符。即 uv_pipe_s 的 ipc 字段。

3.7 文件

由于兼容性问题，Libuv 的文件操作是以线程池实现的，操作文件的时候，会阻塞在某个线程。这里以打开文件操作为例子。

```
// 下面代码是宏展开后的效果
int uv_fs_open(
              uv_loop_t* loop,
               uv_fs_t* req,
               const char* path,
               int flags,
               int mode,
               uv_fs_cb cb
) {
     // 初始化一些字段
UV_REQ_INIT(req, UV_FS);
     req->fs_type = UV_FS_ ## subtype;
     req->result = 0;
     req->ptr = NULL;
     req->loop = loop;
     req->path = NULL;
     req->new_path = NULL;
     req->bufs = NULL;
req->cb = cb;
// 同步
     if (cb == NULL) {
        req->path = path;
```

```
} else {
    req->path = uv__strdup(path);
}
req->flags = flags;
req->mode = mode;
if (cb != NULL) {
    uv__req_register(loop, req);
    /* 异步*/
    uv__work_submit(
        loop,
        &req->work_req,
        UV__WORK_FAST_IO,
        uv__fs_work,
        uv__fs_done
        );
    return 0;
}
else {
     /* 同步 */
    uv__fs_work(&req->work_req);
    return req->result;
}
```

我们从上往下看，没有太多的逻辑，open（大部分的文件操作）分为同步和异步两种模式。同步直接导致 libuv 阻塞，不涉及到线程池，这里只看异步模式。我们看到异步模式下是调用 uv__work_submit 函数给线程池提交一个任务。设置的工作函数和回调函数分别是 uv__fs_work，uv__fs_done。所以我们看一下这两函数。uv__fs_work 函数主要是调用操作系统提供的函数。比如 open。他会引起线程的阻塞，等到执行完后，他会把返回结果保存到 request 结构体中。接着执行就是遵从线程池的处理流程。执行回调 uv__fs_done。

```
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);
  // 取消了
  if (status == UV_ECANCELED) {
    req->result = UV_ECANCELED;
  }
  // 执行用户设置的回调，比如 nodejs
  req->cb(req);
}
```

没有太多逻辑，直接执行回调。

3.8 信号处理

libuv 初始化的时候会初始化信号处理相关的逻辑。

```
uv__signal_global_once_init();

uv_signal_init(loop, &loop->child_watcher)

uv__handle_unref(&loop->child_watcher);

loop->child_watcher.flags |= UV_HANDLE_INTERNAL;
```

我们逐个分析上面的逻辑

```
// 保证只执行 uv__signal_global_init 一次

void uv__signal_global_once_init(void) {

  uv_once(&uv__signal_global_init_guard, uv__signal_global_init);

}



static void uv__signal_global_init(void) {

  if (uv__signal_lock_pipefd[0] == -1)

// 注册 fork 之后，在子进程执行的函数

if (pthread_atfork(NULL, NULL, &uv__signal_global_reinit))

        abort();

  uv__signal_global_reinit();

}
```

```
static void uv__signal_global_reinit(void) {
  // 清除原来的（如果有的话）
  uv__signal_global_fini();
  // 新建一个管道用于互斥控制
  if (uv__make_pipe(uv__signal_lock_pipefd, 0))
    abort();
  // 先往管道写入数据，即解锁。后续才能顺利 lock，unlock 配对使用
  if (uv__signal_unlock())
    abort();
}

UV_DESTRUCTOR(static void uv__signal_global_fini(void)) {
  if (uv__signal_lock_pipefd[0] != -1) {
    uv__close(uv__signal_lock_pipefd[0]);
    uv__signal_lock_pipefd[0] = -1;
  }

  if (uv__signal_lock_pipefd[1] != -1) {
    uv__close(uv__signal_lock_pipefd[1]);
    uv__signal_lock_pipefd[1] = -1;
  }
}
```

经过一系列的操作后，主要是申请了一个用于互斥控制的管道，然后往管道里写数据。后面就可以使用 lock 和 unlock 进行加锁解锁。接着在第一个注册信号的时候，还会做一些初始化的工作。

```
int uv_signal_init(uv_loop_t* loop, uv_signal_t* handle) {
  int err;
  // 申请和 libuv 的通信管道并且注册 io 观察者
  err = uv__signal_loop_once_init(loop);
  if (err)
    return err;

  uv__handle_init(loop, (uv_handle_t*) handle, UV_SIGNAL);
  handle->signum = 0;
  handle->caught_signals = 0;
  handle->dispatched_signals = 0;

  return 0;
}

static int uv__signal_loop_once_init(uv_loop_t* loop) {
  int err;
  // 初始化过了
```

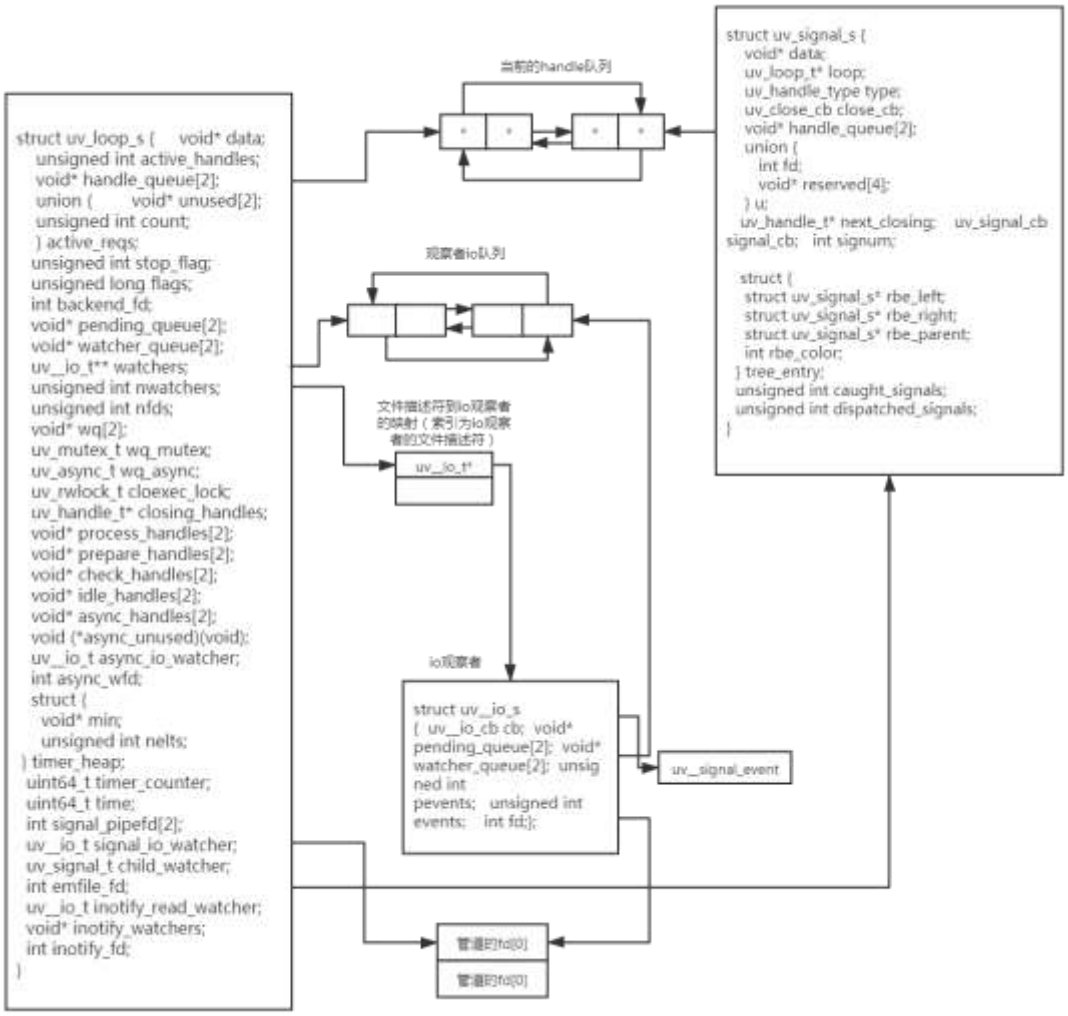
```
  if (loop->signal_pipefd[0] != -1)
    return 0;
  // 申请一个管道，用于其他进程和 libuv 主进程通信，并设置非阻塞标记
  err = uv__make_pipe(loop->signal_pipefd, UV__F_NONBLOCK);
  if (err)
    return err;
  /*
设置信号 io 观察者的处理函数和文件描述符，
libuv 在 poll io 时，发现管道读端 loop->signal_pipefd[0]可读，
则执行 uv__signal_event
*/
  uv__io_init(&loop->signal_io_watcher,
              uv__signal_event,
              loop->signal_pipefd[0]);
  /*
插入 libuv 的 io 观察者队列，并注册感兴趣的事件，即可读的时候，
执行 uv__signal_event
*/
  uv__io_start(loop, &loop->signal_io_watcher, POLLIN);

  return 0;
}
```

上面的代码主要的工作有两个

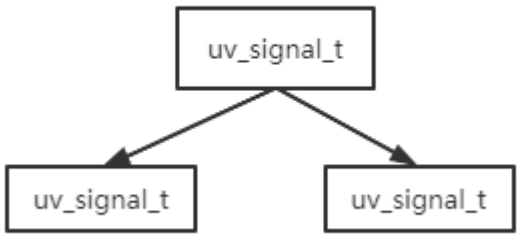
1 申请一个管道，用于其他进程（libuv 进程或 fork 出来的进程）和 libuv 进程通信。然后往 libuv 的 io 观察者队列注册一个观察者，libuv 在 poll io 阶段会把观察者加到 epoll 中。io 观察者里保存了管道读端的文件描述符 loop->signal_pipefd[0]和回调函数 uv__signal_event。uv__signal_event 是任意信号触发时的回调，他会继续根据触发的信号进行逻辑分发。

2 初始化信号信号 handle 的字段。

执行完的内容架构为



libuv 用黑红树维护信号是数据，插入的规则是根据信号的大小和 flags 等信息。



通过 uv_signal_start 注册一个信号处理函数。比如 fork 一个子进程时。libuv 会调 uv_signal_start 设置对 SIGCHLD 信号的处理函数为 uv__chld，主进程在收到这个信号的时候，会执行 uv_chld。

```
uv_signal_start(&loop->child_watcher, uv__chld, SIGCHLD);
```

我们看看这个函数的逻辑

```
static int uv__signal_start(uv_signal_t* handle,
                            uv_signal_cb signal_cb,
                            int signum,
                            int oneshot) {
  sigset_t saved_sigmask;
  int err;
  uv_signal_t* first_handle;


  assert(!uv__is_closing(handle));
  if (signum == 0)
    return UV_EINVAL;
  // 注册过了，重新设置处理函数就行
  if (signum == handle->signum) {
    handle->signal_cb = signal_cb;
    return 0;
  }
  // 这个 handle 之前已经设置了信号和处理函数，则先解除
  if (handle->signum != 0) {
    uv__signal_stop(handle);
  }
```

```
  // 屏蔽所有信号
  uv__signal_block_and_lock(&saved_sigmask);
  /*
注册了该信号的第一个 handle，
优先返回设置了 UV_SIGNAL_ONE_SHOT flag 的，
见 compare 函数
*/
  first_handle = uv__signal_first_handle(signum);
  /*
      1 之前没有注册过该信号的处理函数则直接设置
      2 之前设置过，但是是 one shot，但是现在需要
        设置的规则不是 one shot，需要修改。否则第
        二次不会不会触发。因为一个信号只能对应一
        个信号处理函数，所以，以规则宽的为准备，在回调
        里再根据 flags 判断是不是真的需要执行
          3 如果注册过信号和处理函数，则直接插入红黑树就行。
*/
if (
first_handle == NULL ||
      (!oneshot && (first_handle->flags & UV_SIGNAL_ONE_SHOT))
) {
    // 注册信号和处理函数
```

```
    err = uv__signal_register_handler(signum, oneshot);
    if (err) {
      uv__signal_unlock_and_unblock(&saved_sigmask);
      return err;
    }
  }
  // 记录感兴趣的信号
  handle->signum = signum;
  // 只处理一次该信号
  if (oneshot)
    handle->flags |= UV_SIGNAL_ONE_SHOT;
  // 插入红黑树
  RB_INSERT(uv__signal_tree_s, &uv__signal_tree, handle);

  uv__signal_unlock_and_unblock(&saved_sigmask);

  // 信号触发时的业务层回调
handle->signal_cb = signal_cb;
  uv__handle_start(handle);

  return 0;
}
```

上面的代码比较多，大致的逻辑如下

1 给进程注册一个信号和信号处理函数。主要是调用操作系统的函数来处理的，代码如下

```
// 给当前进程注册信号处理函数，会覆盖之前设置的 signum 对应的处理函数
static int uv__signal_register_handler(int signum, int oneshot) {
  struct sigaction sa;


  memset(&sa, 0, sizeof(sa));
  // 全置一，说明收到 signum 信号的时候，暂时屏蔽其他信号
  if (sigfillset(&sa.sa_mask))
abort();
  // 所有信号都由该函数处理
  sa.sa_handler = uv__signal_handler;
  sa.sa_flags = SA_RESTART;
  // 设置了 oneshot，说明信号处理函数只执行一次，然后被恢复为系统的默认处理函数
  if (oneshot)
    sa.sa_flags |= SA_RESETHAND;


  // 注册
  if (sigaction(signum, &sa, NULL))
    return UV__ERR(errno);
```

```
  return 0;
}
```

2 进程注册的信号和回调是在一棵红黑树管理的，每次注册的时候会往红黑树插入一个节点。我们发现，在 uv__signal_register_handler 函数中有这样一句代码。

```
sa.sa_handler = uv__signal_handler;
```

我们发现，不管注册什么信号，他的处理函数都是这个。我们自己的业务回调函数，是保存在 handle 里的。那么当任意信号到来的时候。uv__signal_handler 就会被调用。下面我们看看 uv__signal_handler 函数。

```
/*
信号处理函数，signum 为收到的信号，
每个子进程收到信号的时候都由该函数处理，
然后通过管道通知 libuv
*/
static void uv__signal_handler(int signum) {
  uv__signal_msg_t msg;
  uv_signal_t* handle;
  int saved_errno;
  // 保持上一个系统调用的错误码
  saved_errno = errno;
  memset(&msg, 0, sizeof msg);
```

```
if (uv__signal_lock()) {
  errno = saved_errno;
  return;
}

for (handle = uv__signal_first_handle(signum);
       handle != NULL && handle->signum == signum;
       handle = RB_NEXT(uv__signal_tree_s, &uv__signal_tree, handle)) {
  int r;

  msg.signum = signum;
  msg.handle = handle;

  do {
    // 通知 libuv，哪些 handle 需要处理该信号，在 poll io 阶段处理
    r = write(handle->loop->signal_pipefd[1], &msg, sizeof msg);
  } while (r == -1 && errno == EINTR);

  assert(r == sizeof msg ||
         (r == -1 && (errno == EAGAIN || errno == EWOULDBLOCK)));
  // 该 handle 收到信号的次数
  if (r != -1)
```

```
        handle->caught_signals++;
    }

    uv__signal_unlock();
    errno = saved_errno;
}
```

该函数遍历红黑树，找到注册了该信号的 handle，然后封装一个 msg 写入管道（即可 libuv 通信的管道）。信号的处理就完成了。接下来在 libuv 的 poll io 阶段才做真正的处理。我们知道在 poll io 阶段。epoll 会检测到管道 loop->signal_pipefd[0]可读，然后会执行 uv__signal_event 函数。我们看看这个函数的代码。

```
// 如果收到信号,libuv poll io 阶段,会执行该函数
static void uv__signal_event(uv_loop_t* loop,
                             uv__io_t* w,
                             unsigned int events) {
    uv__signal_msg_t* msg;
    uv_signal_t* handle;
    char buf[sizeof(uv__signal_msg_t) * 32];
    size_t bytes, end, i;
    int r;

    bytes = 0;
    end = 0;
```

```
do {
  // 独处所有的 uv__signal_msg_t
  r = read(loop->signal_pipefd[0], buf + bytes, sizeof(buf) - bytes);

  if (r == -1 && errno == EINTR)
    continue;

  if (r == -1 && (errno == EAGAIN || errno == EWOULDBLOCK)) {
    if (bytes > 0)
      continue;
    return;
  }

  if (r == -1)
    abort();

  bytes += r;

  /* `end` is rounded down to a multiple of sizeof(uv__signal_msg_t). */
  end = (bytes / sizeof(uv__signal_msg_t)) * sizeof(uv__signal_msg_t);
```

```
  for (i = 0; i < end; i += sizeof(uv__signal_msg_t)) {
    msg = (uv__signal_msg_t*) (buf + i);
    handle = msg->handle;
    // 收到的信号和 handle 感兴趣的信号一致，执行回调
    if (msg->signum == handle->signum) {
      assert(!(handle->flags & UV_HANDLE_CLOSING));
      handle->signal_cb(handle, handle->signum);
    }
    // 处理信号个数
    handle->dispatched_signals++;
    // 只执行一次，恢复系统默认的处理函数
    if (handle->flags & UV_SIGNAL_ONE_SHOT)
      uv__signal_stop(handle);

    // 处理完了关闭
    if ((handle->flags & UV_HANDLE_CLOSING) &&
        (handle->caught_signals == handle->dispatched_signals)) {
      uv__make_close_pending((uv_handle_t*) handle);
    }
  }

  bytes -= end;
```

```
      if (bytes) {
        memmove(buf, buf + end, bytes);
        continue;
      }
    } while (end == sizeof buf);
}
```

分支逻辑很多，我们只需要关注主要的。该函数从管道独处刚才写入的一个个 msg。从 msg 中取出 handle，然后执行里面保存的回调函数（即我们设置的回调函数）。至此。整个信号注册和处理的流程就完成了。整个流程总结如下：

1 libuv 初始化的时候，申请一个管道，用于互斥控制，然后执行往里面写一个数据，保存后续的 lock 和 unlock 可以顺利执行。

2 执行 uv_signal_init 的时候，初始化 handle 的字段。如果是第一次调用，则申请一个管道，然后把管道的读端 fd 和回调封装成一个观察者 oi，插入 libuv 的观察者队列。libuv 会在 poll io 阶段往 epoll 里插入。

3 执行 uv_signal_start 的时候，给进程注册一个信号和处理函数（固定是 uv__signal_handler）。往红黑树插入一个节点，或者修改里面的节点。

4 如果收到信号，在 uv__signal_handler 函数中会往管道（和 libuv 通信的）写入数据，即哪些 handle 注册的信号触发了。

5 在 libuv 的 poll io 阶段，从管道读端读出数据，遍历数据，是一个个 msg，取出 msg 里的 handle，然后取出 handle 里的回调函数执行。

3.9 inotify 文件监听

inotify 是 linux 系统提供用于监听文件系统的机制。inotify 机制的逻辑大致是

1 init_inotify 创建一个 inotify 机制的实例，返回一个文件描述符。类似 epoll。

2 inotify_add_watch 往 inotify 实例注册一个需监听的文件（inotify_rm_watch 是移除）。

3 read((inotify 实例对应的文件描述符, &buf, sizeof(buf)))，如果没有事件触发，则阻塞（除非设置了非阻塞）。否则返回待读取的数据长度。buf 就是保存了触发事件的信息。

libuv 在 inotify 机制的基础上做了一层封装。我们看一下 libuv 中的实现。我们从一个使用例子开始。

```
int main(int argc, char **argv) {
    // 实现循环核心结构体 loop
    loop = uv_default_loop();
    fprintf(stderr, "Adding watch on %s\n", argv[0]);
    uv_fs_event_t *fs_event_req = malloc(sizeof(uv_fs_event_t));
    // 初始化 fs_event_req 结构体的类型为 UV_FS_EVENT
    uv_fs_event_init(loop, fs_event_req);
/*
argv[argc]是文件路径，uv_fs_event_start
向底层注册监听文件 argv[argc],cb 是事件触发时的回调
*/
```

```
    uv_fs_event_start(fs_event_req, cb, argv[argc], UV_FS_EVENT_RECURSIVE);
    // 开启事件循环
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

libuv 在第一次监听文件的时候，会创建一个 inotify 实例。

```
static int init_inotify(uv_loop_t* loop) {
  int err;
  // 初始化过了则直接返回
  if (loop->inotify_fd != -1)
    return 0;
  // 调用操作系统的 inotify_init 函数申请一个 inotify 实例，并设置 UV__IN_NONBLOCK，
UV__IN_CLOEXEC 标记
  err = new_inotify_fd();
  if (err < 0)
    return err;
  // 记录 inotify 实例对应的文件描述符
  loop->inotify_fd = err;
  // inotify_read_watcher 是一个 io 观察者，uv__io_init 设置 io 观察者的文件描述符（待观察的
文件）和回调
  uv__io_init(&loop->inotify_read_watcher, uv__inotify_read, loop->inotify_fd);
  // 往 libuv 中注册该 io 观察者，感兴趣的事件为可读
  uv__io_start(loop, &loop->inotify_read_watcher, POLLIN);
```

```
  return 0;
}
```

分析完 libuv 申请 inotify 实例的逻辑，我们回到 main 函数看看 uv_fs_event_start 函数。

```
uv_fs_event_start(fs_event_req, cb, argv[argc], UV_FS_EVENT_RECURSIVE);
```

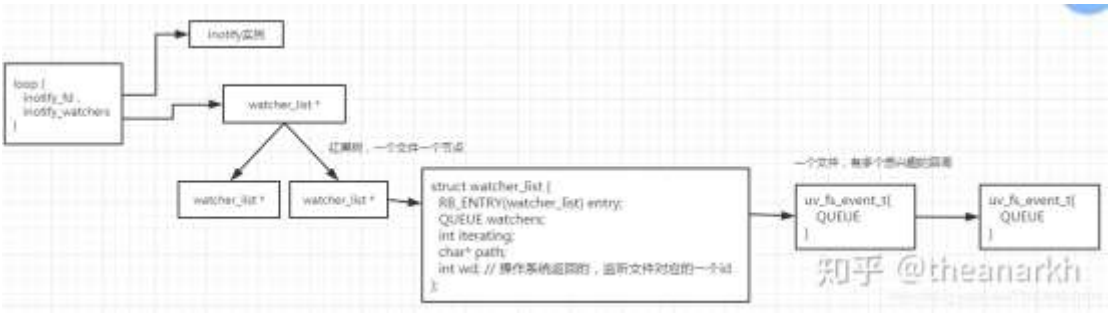
用户使用 uv_fs_event_start 函数来往 libuv 注册一个待监听的文件。我们看看实现。

```
int uv_fs_event_start(uv_fs_event_t* handle,
                      uv_fs_event_cb cb,
                      const char* path,
                      unsigned int flags) {
  struct watcher_list* w;
  int events;
  int err;
  int wd;

  if (uv__is_active(handle))
    return UV_EINVAL;
  // 申请一个 inotify 实例
  err = init_inotify(handle->loop);
  if (err)
    return err;
```

```
// 监听的事件
events = UV__IN_ATTRIB
        | UV__IN_CREATE
        | UV__IN_MODIFY
        | UV__IN_DELETE
        | UV__IN_DELETE_SELF
        | UV__IN_MOVE_SELF
        | UV__IN_MOVED_FROM
        | UV__IN_MOVED_TO;
// 调用操作系统的函数注册一个待监听的文件，返回一个对应于该文件的 id
wd = uv__inotify_add_watch(handle->loop->inotify_fd, path, events);
if (wd == -1)
  return UV__ERR(errno);
// 判断该文件是不是已经注册过了
w = find_watcher(handle->loop, wd);
// 已经注册过则跳过插入的逻辑
if (w)
  goto no_insert;
// 还没有注册过则插入 libuv 维护的红黑树
w = uv__malloc(sizeof(*w) + strlen(path) + 1);
if (w == NULL)
  return UV_ENOMEM;
```

```
  w->wd = wd;
  w->path = strcpy((char*)(w + 1), path);
  QUEUE_INIT(&w->watchers);
  w->iterating = 0;
  // 插入 libuv 维护的红黑树,inotify_watchers 是根节点
  RB_INSERT(watcher_root, CAST(&handle->loop->inotify_watchers), w);

no_insert:
  // 激活该 handle
  uv__handle_start(handle);
  // 同一个文件可能注册了很多个回调，w 对应一个文件，注册在用一个文件的回调排成队
  QUEUE_INSERT_TAIL(&w->watchers, &handle->watchers);
  // 保存信息和回调
  handle->path = w->path;
  handle->cb = cb;
  handle->wd = wd;

  return 0;
}
```

我们先看一下架构图，然后再来具体分析。



下面我们逐步分析上面的函数逻辑。

1 如果是首次调用该函数则新建一个 inotify 实例。并且往 libuv 插入一个观察者 io，libuv 会在 poll io 阶段注册到 epoll 中。

2 往操作系统注册一个待监听的文件。返回一个 id。

3 libuv 判断该 id 是不是在自己维护的红黑树中。不在红黑树中，则插入红黑树。返回一个红黑树中对应的节点。把本次请求的信息封装到 handle 中（回调时需要）。然后把 handle 插入刚才返回的节点的队列中。见上图。

这时候注册过程就完成了。libuv 在 poll io 阶段如果检测到有文件发生变化，则会执行回调 uv__inotify_read。

```
static void uv__inotify_read(uv_loop_t* loop,
                             uv__io_t* dummy,
                             unsigned int events) {
  const struct uv__inotify_event* e;
  struct watcher_list* w;
  uv_fs_event_t* h;
  QUEUE queue;
  QUEUE* q;
  const char* path;
```

```
ssize_t size;
const char *p;
/* needs to be large enough for sizeof(inotify_event) + strlen(path) */
char buf[4096];
// 一次可能没有读完
while (1) {
  do
    // 读取触发的事件信息，size 是数据大小，buffer 保存数据
    size = read(loop->inotify_fd, buf, sizeof(buf));
  while (size == -1 && errno == EINTR);
  // 没有数据可取了
  if (size == -1) {
    assert(errno == EAGAIN || errno == EWOULDBLOCK);
    break;
  }
  // 处理 buffer 的信息
  for (p = buf; p < buf + size; p += sizeof(*e) + e->len) {
    // buffer 里是多个 uv__inotify_event 结构体，里面保存了事件信息和文件对应的 id（wd
字段）
    e = (const struct uv__inotify_event*)p;

    events = 0;
```

```
if (e->mask & (UV__IN_ATTRIB|UV__IN_MODIFY))
  events |= UV_CHANGE;
if (e->mask & ~(UV__IN_ATTRIB|UV__IN_MODIFY))
  events |= UV_RENAME;
// 通过文件对应的 id（wd 字段）从红黑树中找到对应的节点
w = find_watcher(loop, e->wd);


path = e->len ? (const char*) (e + 1) : uv__basename_r(w->path);
w->iterating = 1;
// 把红黑树中，wd 对应节点的 handle 队列移到 queue 变量，准备处理
QUEUE_MOVE(&w->watchers, &queue);
while (!QUEUE_EMPTY(&queue)) {
    // 头结点
  q = QUEUE_HEAD(&queue);
  // 通过结构体偏移拿到首地址
  h = QUEUE_DATA(q, uv_fs_event_t, watchers);
  // 从处理队列中移除
  QUEUE_REMOVE(q);
  // 放回原队列
  QUEUE_INSERT_TAIL(&w->watchers, q);
  // 执行回调
  h->cb(h, path, events, 0);
```

```
      }
    }
  }
}
```

uv__inotify_read 函数的逻辑就是从操作系统中把数据读取出来，这些数据中保存了哪些文件触发了用户感兴趣的事件。然后遍历每个触发了事件的文件。从红黑树中找到该文件对应的红黑树节点。再取出红黑树节点中维护的一个 handle 队列，最后执行 handle 队列中每个节点的回调。

3.10 poll 文件监听

poll 方式监听文件是通过定时检查文件的 stat 信息是否变化。

```
#include "uv.h"
#include "uv-common.h"

#include <assert.h>
#include <stdlib.h>
#include <string.h>

struct poll_ctx {
  uv_fs_poll_t* parent_handle; /* NULL if parent has been stopped or closed */
  int busy_polling;
```

```
  unsigned int interval;
  uint64_t start_time;
  uv_loop_t* loop;
  uv_fs_poll_cb poll_cb;
  uv_timer_t timer_handle;
  uv_fs_t fs_req; /* TODO(bnoordhuis) mark fs_req internal */
  uv_stat_t statbuf;
  // 字符串的值追加在结构体后面
  char path[1]; /* variable length */
};

static int statbuf_eq(const uv_stat_t* a, const uv_stat_t* b);
static void poll_cb(uv_fs_t* req);
static void timer_cb(uv_timer_t* timer);
static void timer_close_cb(uv_handle_t* handle);

static uv_stat_t zero_statbuf;

// 初始化 uv_fs_poll_t 结构
int uv_fs_poll_init(uv_loop_t* loop, uv_fs_poll_t* handle) {
  uv__handle_init(loop, (uv_handle_t*)handle, UV_FS_POLL);
  return 0;
```

```
}
int uv_fs_poll_start(uv_fs_poll_t* handle,
                     uv_fs_poll_cb cb,
                     const char* path,
                     unsigned int interval) {
  struct poll_ctx* ctx;
  uv_loop_t* loop;
  size_t len;
  int err;

  if (uv__is_active(handle))
    return 0;

  loop = handle->loop;
  len = strlen(path);
  // 分配一块内存存上下文结构体和 path 对应的字符串
  ctx = uv__calloc(1, sizeof(*ctx) + len);

  if (ctx == NULL)
    return UV_ENOMEM;
  // 初始化上下文结构
  ctx->loop = loop;
```

```
// 内容改变时的回调
ctx->poll_cb = cb;
// 多久检测一次内容是否变化
ctx->interval = interval ? interval : 1;
// 开始的时间点
ctx->start_time = uv_now(loop);
// 上下文对应的 handle 结构
ctx->parent_handle = handle;
memcpy(ctx->path, path, len + 1);
// 初始化一个定时器
err = uv_timer_init(loop, &ctx->timer_handle);
if (err < 0)
  goto error;
// 设置 UV_HANDLE_INTERNAL 标记位
ctx->timer_handle.flags |= UV_HANDLE_INTERNAL;
//清除 UV_HANDLE_REF 标记
uv__handle_unref(&ctx->timer_handle);
// 异步获取 path 对应的文件的信息，获取到后执行 poll_cb
err = uv_fs_stat(loop, &ctx->fs_req, ctx->path, poll_cb);
if (err < 0)
  goto error;
// 挂载上下文到 handle
```

```
  handle->poll_ctx = ctx;
  // 激活该 handle，但不增加 handle 的 active 数
  uv__handle_start(handle);

  return 0;

error:
  // 出错则释放分配的内存
  uv__free(ctx);
  return err;
}

// 停止 poll
int uv_fs_poll_stop(uv_fs_poll_t* handle) {
  struct poll_ctx* ctx;

  if (!uv__is_active(handle))
    return 0;

  ctx = handle->poll_ctx;
  assert(ctx != NULL);
  assert(ctx->parent_handle != NULL);
```

```
  // 解除关联
  ctx->parent_handle = NULL;
  handle->poll_ctx = NULL;

  /* Close the timer if it's active. If it's inactive, there's a stat request
   * in progress and poll_cb will take care of the cleanup.
   */
  // 停止定时器，设置回调为 time_close_cb，设置状态为 closing
  if (uv__is_active(&ctx->timer_handle))
    uv_close((uv_handle_t*)&ctx->timer_handle, timer_close_cb);

  uv__handle_stop(handle);

  return 0;
}

// 获取 path
int uv_fs_poll_getpath(uv_fs_poll_t* handle, char* buffer, size_t* size) {
  struct poll_ctx* ctx;
  size_t required_len;

  if (!uv__is_active(handle)) {
```

```
    *size = 0;
    return UV_EINVAL;
  }

  ctx = handle->poll_ctx;
  assert(ctx != NULL);

  required_len = strlen(ctx->path);
  if (required_len >= *size) {
    *size = required_len + 1;
    return UV_ENOBUFS;
  }

  memcpy(buffer, ctx->path, required_len);
  *size = required_len;
  buffer[required_len] = '\0';

  return 0;
}


void uv__fs_poll_close(uv_fs_poll_t* handle) {
```

```
    uv_fs_poll_stop(handle);
}

// 定时器到期执行的回调
static void timer_cb(uv_timer_t* timer) {
    struct poll_ctx* ctx;

    ctx = container_of(timer, struct poll_ctx, timer_handle);
    assert(ctx->parent_handle != NULL);
    assert(ctx->parent_handle->poll_ctx == ctx);
    ctx->start_time = uv_now(ctx->loop);
    // 再次获取 stat 信息
    if (uv_fs_stat(ctx->loop, &ctx->fs_req, ctx->path, poll_cb))
        abort();
}

// 获取到 stat 后执行的回调
static void poll_cb(uv_fs_t* req) {
    uv_stat_t* statbuf;
    struct poll_ctx* ctx;
    uint64_t interval;
```

```
  ctx = container_of(req, struct poll_ctx, fs_req);

  if (ctx->parent_handle == NULL) { /* handle has been stopped or closed */
    uv_close((uv_handle_t*)&ctx->timer_handle, timer_close_cb);
    uv_fs_req_cleanup(req);
    return;
  }

  if (req->result != 0) {
    if (ctx->busy_polling != req->result) {
      ctx->poll_cb(ctx->parent_handle,
                   req->result,
                   &ctx->statbuf,
                   &zero_statbuf);
      ctx->busy_polling = req->result;
    }
    goto out;
  }

  statbuf = &req->statbuf;
  // 第一次不执行回调，因为没有可对比的 stat，第二次及后续的操作才可能执行回
调，因为第一次执行的时候置 busy_polling=1
```

```
  if (ctx->busy_polling != 0)
    // 出错或者 stat 发生了变化则执行回调
    if (ctx->busy_polling < 0 || !statbuf_eq(&ctx->statbuf, statbuf))
      ctx->poll_cb(ctx->parent_handle, 0, &ctx->statbuf, statbuf);
  // 保存当前获取到的 stat 信息，置 1
  ctx->statbuf = *statbuf;
  ctx->busy_polling = 1;

out:
  uv_fs_req_cleanup(req);

  if (ctx->parent_handle == NULL) { /* handle has been stopped by callback */
    uv_close((uv_handle_t*)&ctx->timer_handle, timer_close_cb);
    return;
  }

  /* Reschedule timer, subtract the delay from doing the stat(). */
  /*
    假设在开始时间点为 1，interval 为 10 的情况下执行了 stat，stat 完成执行并执
    行 poll_cb 回调的时间点是 3，那么定时器的超时时间则为 10-3=7，即 7 个单位
    后就要触发超时，而不是 10，是因为 stat 阻塞消耗了 3 个单位的时间，所以下次
    执行超时回调函数时说明从 start 时间点开始算，已经经历了 x 单位各 interval，
```

```
    然后超时回调里又执行了 stat 函数，再到执行 stat 回调，这个时间点即
    now=start+x 单位个 interval+stat 消耗的时间。得出 now-start
    为 interval 的 x 倍+stat 消耗，即对 interval 取余可得到 stat 消耗，所以
    当前轮，定时器的超时时间为 interval - ((now-start) % interval)
  */
  interval = ctx->interval;
  interval -= (uv_now(ctx->loop) - ctx->start_time) % interval;

  if (uv_timer_start(&ctx->timer_handle, timer_cb, interval, 0))
    abort();
}

// 释放上下文结构体的内存
static void timer_close_cb(uv_handle_t* handle) {
  uv__free(container_of(handle, struct poll_ctx, timer_handle));
}

static int statbuf_eq(const uv_stat_t* a, const uv_stat_t* b) {
  return a->st_ctim.tv_nsec == b->st_ctim.tv_nsec
      && a->st_mtim.tv_nsec == b->st_mtim.tv_nsec
      && a->st_birthtim.tv_nsec == b->st_birthtim.tv_nsec
      && a->st_ctim.tv_sec == b->st_ctim.tv_sec
```

```
        && a->st_mtim.tv_sec == b->st_mtim.tv_sec
        && a->st_birthtim.tv_sec == b->st_birthtim.tv_sec
        && a->st_size == b->st_size
        && a->st_mode == b->st_mode
        && a->st_uid == b->st_uid
        && a->st_gid == b->st_gid
        && a->st_ino == b->st_ino
        && a->st_dev == b->st_dev
        && a->st_flags == b->st_flags
        && a->st_gen == b->st_gen;
}
```

文件监听的原理是，第一次先执行 stat 函数获取文件基本信息，然后在 stat 的回调函数里设置定时器，定时器超时后会执行 stat，然后获取 stat 信息，再次执行 stat 回调函数重新设置定时器，如此反复，如果 stat 不一样就执行用户的回调。使用

3.11 流

流的实现在 libuv 里占了很大篇幅。首先看数据结构。流在 libuv 里用 uv_stream_s 表示，他属于 handle 族。继承于 uv_handle_s。

```
struct uv_stream_s {
  // uv_handle_s 的字段
  void* data;
```

```
// 所属事件循环
uv_loop_t* loop;
// handle 类型
uv_handle_type type;
// 关闭 handle 时的回调
uv_close_cb close_cb;
// 用于插入事件循环的 handle 队列
void* handle_queue[2];
union {
  int fd;
  void* reserved[4];
} u;
// 用于插入事件循环的 closing 阶段对应的队列
uv_handle_t* next_closing;
// 各种标记
unsigned int flags;
// 流拓展的字段
// 用户写入流的字节大小，流缓存用户的输入，然后等到可写的时候才做真正的写
size_t write_queue_size;
// 分配内存的函数，内存由用户定义，主要用来保存读取的数据
uv_alloc_cb alloc_cb;
// 读取数据的回调
```

```
  uv_read_cb read_cb;
  // 连接成功后，执行 connect_req 的回调（connect_req 在 uv__xxx_connect 中赋值）
  uv_connect_t *connect_req;
  // 关闭写端的时候，发送完缓存的数据，执行 shutdown_req 的回调（shutdown_req 在 uv_shutdown 的时候赋值）
  uv_shutdown_t *shutdown_req;
  // 流对应的 io 观察者，即文件描述符+一个文件描述符事件触发时执行的回调
  uv__io_t io_watcher;
  // 流缓存下来的，待写的数据
  void* write_queue[2];
  // 已经完成了数据写入的队列
  void* write_completed_queue[2];
  // 完成三次握手后，执行的回调
  uv_connection_cb connection_cb;
  // 操作流时出错码
  int delayed_error;
  // accept 返回的通信 socket 对应的文件描述符
  int accepted_fd;
  // 同上，用于缓存更多的通信 socket 对应的文件描述符
  void* queued_fds;
}
```

流的实现中，最核心的字段是 io 观察者，其余的字段是和流的性质相关的。io 观察者封装了流对应的文件描述符和文件描述符事件触发时的回调。比如读一个流，写一个流，关闭一个流，连接一个流，监听一个流，在 uv_stream_s 中都有对应的字段去支持。但是本质上是靠 io 观察者去驱动的。

1 读一个流，就是 io 观察者中的文件描述符。可读事件触发时，执行用户的读回调。

2 写一个流，先把数据写到流中，然后 io 观察者中的文件描述符。可写事件触发时，执行最后的写入，并执行用户的写完成回调。

3 关闭一个流，就是 io 观察者中的文件描述符。可写事件触发时，如果待写的数据已经写完（比如发送完），然后执行关闭流的写端。接着执行用户的回调。

4 连接一个流，比如作为客户端去连接服务器。就是 io 观察者中的文件描述符。可读事件触发时（建立三次握手成功），执行用户的回调。

5 监听一个流，就是 io 观察者中的文件描述符。可读事件触发时（有完成三次握手的连接），执行用户的回调。

今天我们具体分析一下流读写操作的实现。首先我们看一下如何初始化一个流。

```
// 初始化流
void uv__stream_init(uv_loop_t* loop,
                     uv_stream_t* stream,
                     uv_handle_type type) {
```

```
  int err;
  // 记录 handle 的类型
  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;
  // 这个逻辑看起来是为了拿到一个备用的文件描述符 ,如果以后触发 UV_EMFILE 错
误（打开的文件太多）时，使用这个备用的 fd
  if (loop->emfile_fd == -1) {
    err = uv__open_cloexec("/dev/null", O_RDONLY);
    if (err < 0)
        err = uv__open_cloexec("/", O_RDONLY);
    if (err >= 0)
```

```
        loop->emfile_fd = err;
    }
    // 初始化 io 观察者，把文件描述符（这里还没有，所以是-1）和回调 uv__stream_io
记录在 io_watcher 上
    uv__io_init(&stream->io_watcher, uv__stream_io, -1);
}
```

我们看到流的初始化没有太多逻辑。主要是初始化一些字段。接着我们看一下如何打开（激活）一个流。

```
// 关闭 nagle，开启长连接，保存 fd
int uv__stream_open(uv_stream_t* stream, int fd, int flags) {

    // 还没有设置 fd 或者设置的同一个 fd 则继续，否则返回 busy
    if (!(stream->io_watcher.fd == -1 || stream->io_watcher.fd == fd))
        return UV_EBUSY;

    // 设置流的标记
    stream->flags |= flags;

    if (stream->type == UV_TCP) {
        // 关闭 nagle 算法
```

```
    if ((stream->flags & UV_HANDLE_TCP_NODELAY) && uv__tcp_nodelay(fd,
1))
        return UV__ERR(errno);

    // 开启 SO_KEEPALIVE，使用 tcp 长连接，一定时间后没有收到数据包会发送心
跳包
    if ((stream->flags & UV_HANDLE_TCP_KEEPALIVE) &&
        uv__tcp_keepalive(fd, 1, 60)) {
        return UV__ERR(errno);
    }
  }
  // 保存 socket 对应的文件描述符到 io 观察者中，libuv 会在 io poll 阶段监听该文
件描述符
  stream->io_watcher.fd = fd;

  return 0;
}
```

打开一个流 ,本质上就是给这个流关联一个文件描述符。还有一些属性的设置。有了文件描述符，后续就可以操作这个流了。 下面我们逐个操作分析。

3.2.13.1 读

我们在一个流上执行 uv_read_start。流的数据（如果有的话）就会源源不断地流向调用方。

```
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);
  // 流已经关闭，不能读
  if (stream->flags & UV_HANDLE_CLOSING)
    return UV_EINVAL;
  // 流不可读，说明可能是只写流
  if (!(stream->flags & UV_HANDLE_READABLE))
    return -ENOTCONN;
  // 标记正在读
  stream->flags |= UV_HANDLE_READING;
  // 记录读回调，有数据的时候会执行这个回调
  stream->read_cb = read_cb;
  // 分配内存函数，用于存储读取的数据
  stream->alloc_cb = alloc_cb;
  // 注册读事件
  uv__io_start(stream->loop, &stream->io_watcher, POLLIN);
```

```
  // 激活 handle，有激活的 handle，事件循环不会退出
  uv__handle_start(stream);

  return 0;
}
```

执行 uv_read_start 本质上是给流对应的文件描述符在 epoll 中注册了一个可读事件。并且给一些字段赋值，比如读回调函数，分配内存的函数。打上正在做读取操作的标记。然后在可读事件触发的时候，读回调就会被执行，这个逻辑我们后面分析。除了开始读取数据，还有一个读操作就是停止读取。对应的函数是 uv_read_stop。

```
int uv_read_stop(uv_stream_t* stream) {
  // 是否正在执行读取操作，如果不是，则没有必要停止
  if (!(stream->flags & UV_HANDLE_READING))
    return 0;
  // 清除 正在读取 的标记
  stream->flags &= ~UV_HANDLE_READING;
  // 撤销 等待读事件
  uv__io_stop(stream->loop, &stream->io_watcher, POLLIN);
  // 对写事件也不感兴趣，停掉 handle。允许事件循环退出
  if (!uv__io_active(&stream->io_watcher, POLLOUT))
    uv__handle_stop(stream);
```

```
  stream->read_cb = NULL;
  stream->alloc_cb = NULL;
  return 0;
}
```

和 start 相反，start 是注册等待可读事件和打上正在读取这个标记，stop 就是撤销等待可读事件和清除这个标记。另外还有一个辅助函数，判断流是否设置了可读属性。

```
int uv_is_readable(const uv_stream_t* stream) {
  return !!(stream->flags & UV_HANDLE_READABLE);
}
```

3.2.13.2 写

我们在流上执行 uv_write 就可以往流中写入数据。

```
int uv_write(
        // 一个写请求，记录了需要写入的数据和信息。数据来自下面的 const
uv_buf_t bufs[]
          uv_write_t* req,
          // 往哪个流写
        uv_stream_t* handle,
        // 需要写入的数据
        const uv_buf_t bufs[],
```

```
        // 个数
        unsigned int nbufs,
        // 写完后执行的回调
        uv_write_cb cb
) {
  return uv_write2(req, handle, bufs, nbufs, NULL, cb);
}
```

uv_write 是直接调用 uv_write2。第四个参数是 NULL。代表是一般的写数据，不传递文件描述符。

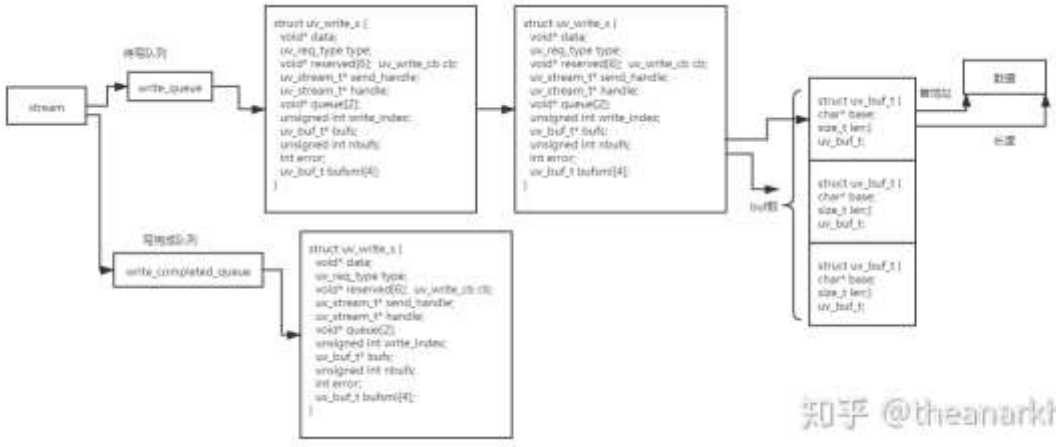
```
int uv_write2(
      uv_write_t* req,
      uv_stream_t* stream,
      const uv_buf_t bufs[],
      unsigned int nbufs,
      // 需要传递的文件描述符所在的流，这里不分析，在分析 unix 的时候再分析
      uv_stream_t* send_handle,
      uv_write_cb cb
)
{
  int empty_queue;
  // 是不可写流
```

```
if (!(stream->flags & UV_HANDLE_WRITABLE))
   return -EPIPE;
// 流中缓存的数据大小是否为 0
empty_queue = (stream->write_queue_size == 0);


// 初始化一个写请求
uv__req_init(stream->loop, req, UV_WRITE);
// 写完后执行的回调
req->cb = cb;
// 往哪个流写
req->handle = stream;
// 写出错的错误码，初始化为 0
req->error = 0;
QUEUE_INIT(&req->queue);
// 默认 buf
req->bufs = req->bufsml;
// 不够则扩容
if (nbufs > ARRAY_SIZE(req->bufsml))
   req->bufs = uv__malloc(nbufs * sizeof(bufs[0]));
// 把需要写入的数据填充到 req 中
memcpy(req->bufs, bufs, nbufs * sizeof(bufs[0]));
// 需要写入的 buf 个数
```

```
  req->nbufs = nbufs;
  // 目前写入的 buf 个数，初始化是 0
  req->write_index = 0;
  // 更新流中待写数据的总长度，就是每个 buf 的数据大小加起来
  stream->write_queue_size += uv__count_bufs(bufs, nbufs);

  // 插入待写队列
  QUEUE_INSERT_TAIL(&stream->write_queue, &req->queue);
  /*
    stream->connect_req 非空说明是作为客户端，并且正在建立三次握手，建立成功会置 connect_req 为 NULL。
    这里非空说明还没有建立成功或者不是作为客户端（不是连接流）。即没有用到 connect_req 这个字段。
  */
  if (stream->connect_req) {
    /* Still connecting, do nothing. */
  }
  else if (empty_queue) {
    // 待写队列为空，则直接触发写动作，即操作文件描述符
    uv__write(stream);
  }
  else {
```

```
    /*
        队列非空，说明往底层写，uv__write 中不一样会注册等待可写事件，所以这里注册一下
        给流注册等待可写事件，触发的时候，把数据消费掉
    */
    uv__io_start(stream->loop, &stream->io_watcher, POLLOUT);
  }

  return 0;
}
```

uv_write2 的主要逻辑就是封装一个写请求，插入到流的待写队列。然后根据当前流的情况。看是直接写入还是等待会再写入。架构大致如下。



我们继续看真正的写操作。

```
static void uv__write(uv_stream_t* stream) {
```

```
  struct iovec* iov;
  QUEUE* q;
  uv_write_t* req;
  int iovmax;
  int iovcnt;
  ssize_t n;
  int err;


start:
  // 待写队列为空，没得写
  if (QUEUE_EMPTY(&stream->write_queue))
    return;
  // 遍历待写队列，把每个节点的数据写入底层
  q = QUEUE_HEAD(&stream->write_queue);
  req = QUEUE_DATA(q, uv_write_t, queue);
  /*
    struct iovec {
        ptr_t iov_base; // 数据首地址
        size_t iov_len; // 数据长度
    };
    iovec 和 bufs 结构体的定义一样
  */
```

```
// 转成 iovec 格式发送
iov = (struct iovec*) &(req->bufs[req->write_index]);
// 待写的 buf 个数，nbufs 是总数，write_index 是当前已写的个数
iovcnt = req->nbufs - req->write_index;
// 最多能写几个
iovmax = uv__getiovmax();

// 取最小值
if (iovcnt > iovmax)
  iovcnt = iovmax;

// 有需要传递的描述符
if (req->send_handle) {
  // 需要传递文件描述符的逻辑，分析 unix 域的时候再分析
} else { // 单纯发送数据，则直接写
  do {
    if (iovcnt == 1) {
      n = write(uv__stream_fd(stream), iov[0].iov_base, iov[0].iov_len);
    } else {
      n = writev(uv__stream_fd(stream), iov, iovcnt);
    }
  } while (n == -1 && errno == EINTR);
```

```
}
// 发送出错
if (n < 0) {
  // 发送失败的逻辑，我们不具体分析
} else {
  // 写成功，n 是写成功的字节数
  while (n >= 0) {
    // 本次待写数据的首地址
    uv_buf_t* buf = &(req->bufs[req->write_index]);
    // 某个 buf 的数据长度
    size_t len = buf->len;
    // len 如果大于 n 说明本 buf 的数据部分被写入
    if ((size_t)n < len) {
      // 更新指针，指向下次待发送的数据首地址
      buf->base += n;
      // 更新待发送数据的长度
      buf->len -= n;
      // 更新待写数据的总长度
      stream->write_queue_size -= n;
      n = 0;
      // 设置了一直写标记，则继续写
      if (stream->flags & UV_HANDLE_BLOCKING_WRITES) {
```

```
            goto start;
          } else {
            // 否则等待可写事件触发的时候再写
            break;
          }
        } else {
          // 本 buf 的数据完成被写入，更新下一个待写入的 buf 位置
          req->write_index++;
          n -= len;
          // 更新待写数据总长度
          stream->write_queue_size -= len;
          // 如果写完了全部 buf，触发回调
          if (req->write_index == req->nbufs) {
            // 写完了本请求的数据，做后续处理
            uv__write_req_finish(req);
            return;
          }
        }
      }
    }
    // 到这说明数据还没有完全被写入，注册等待可写事件，等待继续写
    uv__io_start(stream->loop, &stream->io_watcher, POLLOUT);
```

```
  return;


error:
  // 写出错
  req->error = err;
  uv__write_req_finish(req);
  // 撤销等待可写事件
  uv__io_stop(stream->loop, &stream->io_watcher, POLLOUT);
  // 没有注册了等待可读事件，则停掉流
  if (!uv__io_active(&stream->io_watcher, POLLIN))
    uv__handle_stop(stream);
}
```

我们看一下写完一个请求后 ,libuv 如何处理他。逻辑在 uv__write_req_finish 函数。

```
// 把 buf 的数据写入完成或写出错后触发的回调
static void uv__write_req_finish(uv_write_t* req) {
  uv_stream_t* stream = req->handle;
  // 移出队列
  QUEUE_REMOVE(&req->queue);
  // 写入成功了
  if (req->error == 0) {
```

```
    /*
       bufsml 是默认的 buf 数，如果不够，则 bufs 指向新的内存，
       然后再储存数据。两者不等说明申请了额外的内存，需要 free 掉
    */
    if (req->bufs != req->bufsml)
      uv__free(req->bufs);
    req->bufs = NULL;
  }
  // 插入写完成队列
  QUEUE_INSERT_TAIL(&stream->write_completed_queue, &req->queue);
  // 插入 pending 队列，在 pending 阶段执行回调
  uv__io_feed(stream->loop, &stream->io_watcher);
}
```

uv__write_req_finish 的逻辑比较简单，就是把节点从待写队列中移除。然后插入写完成队列。最后把 io
观察者插入 pending 队列。在 pending 节点会知道 io 观察者的回调（uv__stream_io）。流模块的逻辑比较多，今天先分析到这里。后续继续分析其他操作。
上一篇分析了流的基础知识和读写操作的实现。今天继续分析。

3.2.13.3 关闭流的写端

```
// 关闭流的写端
int uv_shutdown(uv_shutdown_t* req, uv_stream_t* stream, uv_shutdown_cb cb)
{

  // 流是可写的，并且还没关闭写端，也不是处于正在关闭状态
  if (!(stream->flags & UV_HANDLE_WRITABLE) ||
        stream->flags & UV_HANDLE_SHUT ||
        stream->flags & UV_HANDLE_SHUTTING ||
        uv__is_closing(stream)) {
    return UV_ENOTCONN;
  }

  // 初始化一个关闭请求，关联的 handle 是 stream
  uv__req_init(stream->loop, req, UV_SHUTDOWN);
  req->handle = stream;
  // 关闭后执行的回调
  req->cb = cb;
  stream->shutdown_req = req;
  // 设置正在关闭的标记
  stream->flags |= UV_HANDLE_SHUTTING;
```

```
  // 注册等待可写事件
  uv__io_start(stream->loop, &stream->io_watcher, POLLOUT);


  return 0;
}
```

关闭流的写端就是相当于给流发送一个关闭请求，把请求挂载到流中，然后注册等待可写事件，在可写事件触发的时候就会执行关闭操作。这个我们后面分析。

2 关闭流

```
void uv__stream_close(uv_stream_t* handle) {
  unsigned int i;
  uv__stream_queued_fds_t* queued_fds;
  // 从事件循环中删除 io 观察者，移出 pending 队列
  uv__io_close(handle->loop, &handle->io_watcher);
  // 停止读
  uv_read_stop(handle);
  // 停掉 handle
  uv__handle_stop(handle);
  // 不可读、写
  handle->flags &= ~(UV_HANDLE_READABLE | UV_HANDLE_WRITABLE);
```

```
// 关闭非标准流的文件描述符
if (handle->io_watcher.fd != -1) {
  /* Don't close stdio file descriptors.  Nothing good comes from it. */
  if (handle->io_watcher.fd > STDERR_FILENO)
    uv__close(handle->io_watcher.fd);
  handle->io_watcher.fd = -1;
}
// 关闭通信 socket 对应的文件描述符
if (handle->accepted_fd != -1) {
  uv__close(handle->accepted_fd);
  handle->accepted_fd = -1;
}

/* Close all queued fds */
// 同上，这是在排队等待处理的通信 socket
if (handle->queued_fds != NULL) {
  queued_fds = handle->queued_fds;
  for (i = 0; i < queued_fds->offset; i++)
    uv__close(queued_fds->fds[i]);
  uv__free(handle->queued_fds);
  handle->queued_fds = NULL;
}
```

```
  assert(!uv__io_active(&handle->io_watcher, POLLIN | POLLOUT));
}
```

3 连接流

连接流是针对 tcp 的，连接即建立三次握手。所以我们首先介绍一下一些网络编程相关的内容。想要发起三次握手，首先我们先要有一个 socket。我们看 libuv 中如何新建一个 socket。

```
/*
1 获取一个新的 socket fd
2 把 fd 保存到 handle 里，并根据 flag 进行相关设置
3 绑定到本机随意的地址（如果设置了该标记的话）
*/
static int new_socket(uv_tcp_t* handle, int domain, unsigned long flags) {
  struct sockaddr_storage saddr;
  socklen_t slen;
  int sockfd;
  int err;
  // 获取一个 socket
  err = uv__socket(domain, SOCK_STREAM, 0);
  if (err < 0)
```

```
    return err;
  // 申请的 fd
  sockfd = err;
  // 设置选项和保存 socket 的文件描述符到 io 观察者中
  err = uv__stream_open((uv_stream_t*) handle, sockfd, flags);
  if (err) {
    uv__close(sockfd);
    return err;
  }
  // 设置了需要绑定标记 UV_HANDLE_BOUND
  if (flags & UV_HANDLE_BOUND) {
    slen = sizeof(saddr);
    memset(&saddr, 0, sizeof(saddr));
    // 获取 fd 对应的 socket 信息，比如 ip，端口，可能没有
    if (getsockname(uv__stream_fd(handle), (struct sockaddr*) &saddr, &slen))
{
      uv__close(sockfd);
      return UV__ERR(errno);
    }
    // 绑定到 socket 中，如果没有则绑定到系统随机选择的地址
    if (bind(uv__stream_fd(handle), (struct sockaddr*) &saddr, slen)) {
      uv__close(sockfd);
```

```
        return UV__ERR(errno);
      }
    }

    return 0;
}
```

上面的代码就是在 libuv 申请一个 socket 的逻辑，他还支持新建的 socket，可以绑定到一个用户设置的，或者操作系统随机选择的地址。不过 libuv 并不直接使用这个函数。而是又封装了一层。

```
// 如果流还没有对应的 fd，则申请一个新的，如果有则修改流的配置
static int maybe_new_socket(uv_tcp_t* handle, int domain, unsigned long flags)
{
    struct sockaddr_storage saddr;
    socklen_t slen;

    if (domain == AF_UNSPEC) {
      handle->flags |= flags;
      return 0;
    }
    // 已经有 socket fd 了
    if (uv__stream_fd(handle) != -1) {
```

```
    // 该流需要绑定到一个地址
    if (flags & UV_HANDLE_BOUND) {
      /*
          流是否已经绑定到一个地址了。handle 的 flag 是在 new_socket 里设置的，
          如果有这个标记说明已经执行过绑定了，直接更新 flags 就行。
      */
      if (handle->flags & UV_HANDLE_BOUND) {
        handle->flags |= flags;
        return 0;
      }
      // 有 socket fd，但是可能还没绑定到一个地址
      slen = sizeof(saddr);
      memset(&saddr, 0, sizeof(saddr));
      // 获取 socket 绑定到的地址
      if (getsockname(uv__stream_fd(handle), (struct sockaddr*) &saddr,
&slen))
        return UV__ERR(errno);
      // 绑定过了 socket 地址，则更新 flags 就行
      if ((saddr.ss_family == AF_INET6 &&
          ((struct sockaddr_in6*) &saddr)->sin6_port != 0) ||
          (saddr.ss_family == AF_INET &&
          ((struct sockaddr_in*) &saddr)->sin_port != 0)) {
```

```
            /* Handle is already bound to a port. */
            handle->flags |= flags;
            return 0;
          }
          // 没绑定则绑定到随机地址，bind 中实现
          if (bind(uv__stream_fd(handle), (struct sockaddr*) &saddr, slen))
            return UV__ERR(errno);
        }

        handle->flags |= flags;
        return 0;
      }
      // 申请一个新的 fd 关联到流
      return new_socket(handle, domain, flags);
}
```

maybe_new_socket 函数的逻辑分支很多

1 如果流还没有关联到 fd，则申请一个新的 fd 关联到流上。如果设置了绑定标记，fd 还会和一个地址进行绑定。

2 如果流已经关联了一个 fd

1. 如果流设置了绑定地址的标记，但是已经通过 libuv 绑定了一个地址（Libuv 会设置 UV_HANDLE_BOUND 标记，用户也可能是直接调 bind 函数绑定了）。则不需要再次绑定，更新 flags 就行。
2. 如果流设置了绑定地址的标记，但是还没有通过 libuv 绑定一个地址，这时候通过 getsocketname 判断用户是否自己通过 bind 函数绑定了一个地址，是的话则不需要再次执行绑定操作。否则随机绑定到一个地址。

以上两个函数的逻辑主要是申请一个 socket 和给 socket 绑定一个地址。下面我们开看一下连接流的实现。

```
int uv__tcp_connect(uv_connect_t* req,
                    uv_tcp_t* handle,
                    const struct sockaddr* addr,
                    unsigned int addrlen,
                    uv_connect_cb cb) {
  int err;
  int r;

  // 已经发起了 connect 了
  if (handle->connect_req != NULL)
    return UV_EALREADY;
  // 申请一个 socket 和绑定一个地址，如果还没有的话
  err = maybe_new_socket(handle,
```

```
                                addr->sa_family,
                                UV_HANDLE_READABLE | UV_HANDLE_WRITABLE);
if (err)
  return err;

handle->delayed_error = 0;

do {
  // 清除全局错误变量的值
  errno = 0;
  // 发起三次握手
  r = connect(uv__stream_fd(handle), addr, addrlen);
} while (r == -1 && errno == EINTR);

if (r == -1 && errno != 0) {
  // 三次握手还没有完成
  if (errno == EINPROGRESS)
    ; /* not an error */
  else if (errno == ECONNREFUSED)
    // 对方拒绝建立连接，延迟报错
    handle->delayed_error = UV__ERR(errno);
  else
```

```
        // 直接报错
        return UV__ERR(errno);
    }
    // 初始化一个连接型 request，并设置某些字段
    uv__req_init(handle->loop, req, UV_CONNECT);
    req->cb = cb;
    req->handle = (uv_stream_t*) handle;
    QUEUE_INIT(&req->queue);
    handle->connect_req = req;
    // 注册到 libuv 观察者队列
    uv__io_start(handle->loop, &handle->io_watcher, POLLOUT);
    // 连接出错，插入 pending 队尾
    if (handle->delayed_error)
      uv__io_feed(handle->loop, &handle->io_watcher);

    return 0;
}
```

连接流的逻辑，大致如下

1 申请一个 socket，绑定一个地址。

2 根据给定的服务器地址，发起三次握手，非阻塞的，会直接返回继续执行，不会等到三次握手完成。

3 往流上挂载一个 connect 型的请求。

4 设置 io 观察者感兴趣的事件为可写。然后把 io 观察者插入事件循环的 io 观察者队列。等待可写的时候时候（完成三次握手），就会执行 cb 回调。

4 监听流

```
int uv_tcp_listen(uv_tcp_t* tcp, int backlog, uv_connection_cb cb) {
  static int single_accept = -1;
  unsigned long flags;
  int err;


  if (tcp->delayed_error)
    return tcp->delayed_error;
  // 是否设置了不连续 accept。默认是连续 accept。
  if (single_accept == -1) {
    const char* val = getenv("UV_TCP_SINGLE_ACCEPT");
    single_accept = (val != NULL && atoi(val) != 0);  /* Off by default. */
  }
  // 设置不连续 accept
  if (single_accept)
    tcp->flags |= UV_HANDLE_TCP_SINGLE_ACCEPT;


  flags = 0;
```

```
  /*
     可能还没有用于 listen 的 fd，socket 地址等。
     这里申请一个 socket 和绑定到一个地址（如果调 listen 之前没有调 bind 则绑定到随机地址）
  */
  err = maybe_new_socket(tcp, AF_INET, flags);
  if (err)
     return err;
  // 设置 fd 为 listen 状态
  if (listen(tcp->io_watcher.fd, backlog))
     return UV__ERR(errno);
  // 建立连接后的业务回调
  tcp->connection_cb = cb;
  tcp->flags |= UV_HANDLE_BOUND;
  // 有连接到来时的 libuv 层回调
  tcp->io_watcher.cb = uv__server_io;
  // 注册读事件，等待连接到来
  uv__io_start(tcp->loop, &tcp->io_watcher, POLLIN);

  return 0;
}
```

监听流的逻辑看起来逻辑很多，但是主要的逻辑是把流对的 fd 改成 listen 状态，这样流就可以接收请求了。然后设置连接到来时执行的回调。最后注册 io 观察者到事件循环。等待连接到来。就会执行 uv__server_io。uv__server_io 再执行 connection_cb。监听流和其他流的一个区别是，当 io 观察者的事件触发时，监听流执行的回调是 uv__server_io 函数。而其他流是在 uv__stream_io 里统一处理。

流的类型分析得差不多了，最后分析一下监听流的处理函数 uv__server_io，统一处理其他流的函数是 uv__stream_io，这个下次分析。

刚才已经说到有连接到来的时候，libuv 会执行 uv__server_io，下面看一下他做了什么事情。

```
// 有 tcp 连接到来时执行该函数
void uv__server_io(uv_loop_t* loop, uv__io_t* w, unsigned int events) {
  uv_stream_t* stream;
  int err;
  // 拿到 io 观察者所在的流
  stream = container_of(w, uv_stream_t, io_watcher);
  // 继续注册事件,等待连接
  uv__io_start(stream->loop, &stream->io_watcher, POLLIN);

  /* connection_cb can close the server socket while we're
```

```
   * in the loop so check it on each iteration.
   */
  while (uv__stream_fd(stream) != -1) {
    // 有连接到来，进行 accept
    err = uv__accept(uv__stream_fd(stream));
    if (err < 0) {
      // 忽略出错处理
      // accept 出错，触发回调
      stream->connection_cb(stream, err);
      continue;
    }
    // 保存通信 socket 对应的文件描述符
    stream->accepted_fd = err;
    /*
        有连接，执行上层回调，connection_cb 一般会调用 uv_accept 消费
accepted_fd。
        然后重新注册等待可读事件
    */
    stream->connection_cb(stream, 0);
    /*
        用户还没有消费 accept_fd。先解除 io 的事件，
        等到用户调用 uv_accept 消费了 accepted_fd 再重新注册事件
```

```
    */
    if (stream->accepted_fd != -1) {
      uv__io_stop(loop, &stream->io_watcher, POLLIN);
      return;
    }
    // 定时睡眠一会（可被信号唤醒），分点给别的进程 accept
    if (stream->type == UV_TCP &&
        (stream->flags & UV_HANDLE_TCP_SINGLE_ACCEPT)) {
      struct timespec timeout = { 0, 1 };
      nanosleep(&timeout, NULL);
    }
  }
}
```

整个函数的逻辑如下

1 调用 accept 摘下一个完成了三次握手的节点。

2 然后执行上层回调。上层回调会调用 uv_accept 消费 accept 返回的 fd。然后再次注册等待可读事件（当然也可以不消费）。

3 如果 2 没有消费调 fd。则撤销等待可读事件，即处理完一个 fd 后，再 accept 下一个。如果 2 中消费了 fd。再判断有没有设置 UV_HANDLE_TCP_SINGLE_ACCEPT 标记，如果有则休眠一会，分点给别的进程 accept。否则继续 accept。

3.12 进程

这章以 nodejs 为背景进行分析。我们知道 nodejs 是单进程（单线程）的，但是 nodejs 也为用户实现了多进程的能力，下面我们看一下 nodejs 里多进程的架构是怎么样的。

nodejs 提供同步和异步创建进程的方式。我们首先看一下异步的方式 ,nodejs 创建进程的方式由很多种。但是归根到底是通过 spawn 函数。所以我们从这个函数开始，看一下整个流程。

```
var spawn = exports.spawn = function(/*file, args, options*/) {


  var opts = normalizeSpawnArguments.apply(null, arguments);
  var options = opts.options;
  var child = new ChildProcess();


  debug('spawn', opts.args, options);


  child.spawn({
    file: opts.file,
    args: opts.args,
    cwd: options.cwd,
    windowsHide: !!options.windowsHide,
    windowsVerbatimArguments: !!options.windowsVerbatimArguments,
```

```
    detached: !!options.detached,
    envPairs: opts.envPairs,
    stdio: options.stdio,
    uid: options.uid,
    gid: options.gid
  });

  return child;
};
```

我们看到 spawn 函数只是对 ChildProcess 函数的封装。然后调用他的 spawn 函数（只列出核心代码）。

```
const { Process } = process.binding('process_wrap');

function ChildProcess() {
  EventEmitter.call(this);
  this._handle = new Process();
}

ChildProcess.prototype.spawn = function(options) {
    this._handle.spawn(options);
}
```

ChildProcess 也是对 Process 的封装。Process 是 js 层和 c++层的桥梁，我们找到他对应的 c++模块。

```
NODE_BUILTIN_MODULE_CONTEXT_AWARE(process_wrap,
node::ProcessWrap::Initialize)
```

即在 js 层调用 process.binding('process_wrap')的时候，拿到的是 node::ProcessWrap::Initialize 导出的对象。

```
  static void Initialize(Local<Object> target,
                         Local<Value> unused,
                         Local<Context> context
  ) {
    Environment* env = Environment::GetCurrent(context);
    // 定义一个构造函数，值是 New
    Local<FunctionTemplate> constructor =
env->NewFunctionTemplate(New);
    constructor->InstanceTemplate()->SetInternalFieldCount(1);
    //   拿到一个字符串
    Local<String> processString =
        FIXED_ONE_BYTE_STRING(env->isolate(), "Process");
    constructor->SetClassName(processString);
```

```
    AsyncWrap::AddWrapMethods(env, constructor);
    // 设置这个构造函数的原型方法
    env->SetProtoMethod(constructor, "close", HandleWrap::Close);


    env->SetProtoMethod(constructor, "spawn", Spawn);
    env->SetProtoMethod(constructor, "kill", Kill);
    ...
    /*
        类似 js 里的 module.exports = {Process: New}
        js 层 new Process 的时候会相对于执行 new New
    */
    target->Set(processString, constructor->GetFunction());
  }
```

上面的代码翻译成 js 大概如下。

```
function New() {}
New.prototype = {
    spawn: Spawn,
    kill: Kill,
    close: Close
    ...
}
```

```
module.exports = {Process: New}
```

所以 new Process 的时候，执行的是 new New。

```
static void New(const FunctionCallbackInfo<Value>& args) {
    Environment* env = Environment::GetCurrent(args);
    new ProcessWrap(env, args.This());
}


 ProcessWrap(Environment* env, Local<Object> object)
       : HandleWrap(
                env,
                object,
                reinterpret_cast<uv_handle_t*>(&process_),
                AsyncWrap::PROVIDER_PROCESSWRAP
      )
{
}
```

我们看到 new New 就是 new ProcessWrap，但是 New 函数没有返回一个值。继续往下看。

```
HandleWrap::HandleWrap(...) {
  Wrap(object, this);
```

```
}


void Wrap(v8::Local<v8::Object> object, TypeName* pointer) {
  object->SetAlignedPointerInInternalField(0, pointer);
}
```

v8 的套路有点复杂，大致就是在 FunctionCallbackInfo 对象里保存了 ProcessWrap 类的对象。后续调用的时候会取出来。然后给 js 返回一个对象。接着

```
this._handle.spawn(options);
```

这时候会执行

```
  static void Spawn(const FunctionCallbackInfo<Value>& args) {
    Environment* env = Environment::GetCurrent(args);
    Local<Context> context = env->context();
    ProcessWrap* wrap;
    /*
        取出刚才的 ProcessWrap 对象
        wrap = args->GetAlignedPointerFromInternalField(0);
    */
    ASSIGN_OR_RETURN_UNWRAP(&wrap, args.Holder());
```

```
    int err = uv_spawn(env->event_loop(), &wrap->process_, &options);


    args.GetReturnValue().Set(err);
}
```

接着我们通过 uv_spawn 来到了 c 语言层。uv_spawn 总的来说做了下面几个事情。

1 主进程注册 SIGCHLD 信号，处理函数为 uv__chld。SIGCHLD 信号是子进程退出时发出的。

2 处理进程间通信、标准输入、输出。

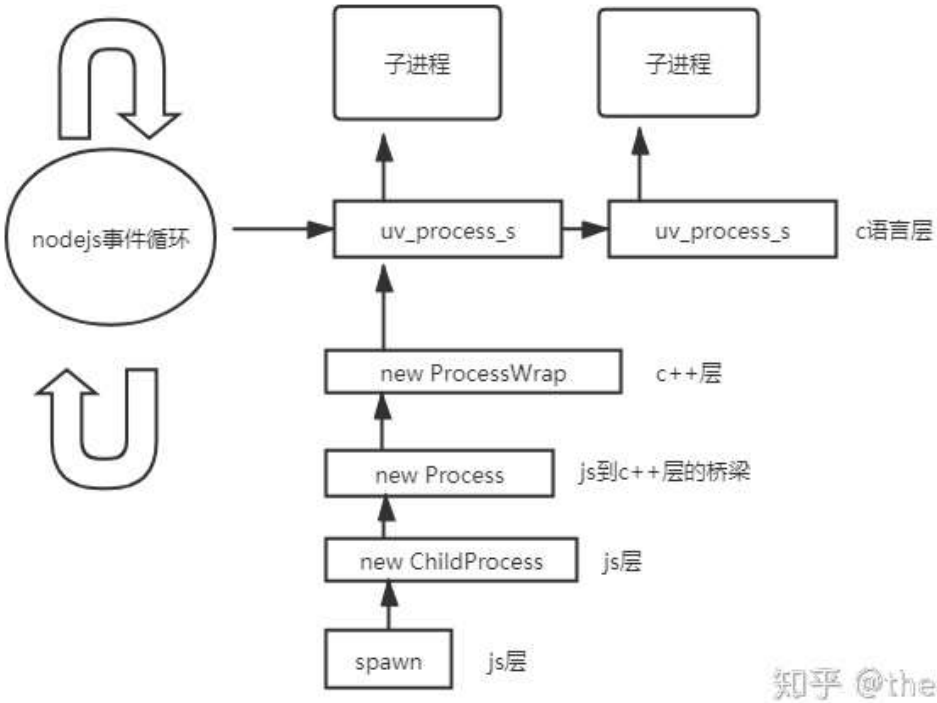
3 fork 出子进程

4 在 uv_process_t 结构图中保存子进程信息，uv_process_t 是 c++层和 c 层的联系。

5 把 uv_process_t 插入 libuv 事件循环的 process_handles 队列

6 主进程和子进程各自运行。

整个流程下来，大致形成如图所示的架构。



当进程退出的时候。nodejs主进程会收到SIGCHLD信号。然后执行uv__chld。该函数遍历 libuv 进程队列中的节点，通过 waitpid 判断该节点对应的进程是否已经退出后，从而收集已退出的节点，然后移出 libuv 队列，最后执行已退出进程的回调。

```
static void uv__chld(uv_signal_t* handle, int signum) {
  uv_process_t* process;
  uv_loop_t* loop;
  int exit_status;
  int term_signal;
  int status;
  pid_t pid;
```

```
QUEUE pending;
QUEUE* q;
QUEUE* h;
// 保存进程（已退出的状态）的队列
QUEUE_INIT(&pending);
loop = handle->loop;

h = &loop->process_handles;
q = QUEUE_HEAD(h);
//   收集已退出的进程
while (q != h) {
   process = QUEUE_DATA(q, uv_process_t, queue);
   q = QUEUE_NEXT(q);

   do
      // WNOHANG 非阻塞等待子进程退出，其实就是看哪个子进程退出了，没有的话就直接返回，而不是阻塞
      pid = waitpid(process->pid, &status, WNOHANG);
   while (pid == -1 && errno == EINTR);

   if (pid == 0)
      continue;
```

```
    // 进程退出了，保存退出状态，移出队列，插入 peding 队列，等待处理
    process->status = status;
    QUEUE_REMOVE(&process->queue);
    QUEUE_INSERT_TAIL(&pending, &process->queue);
}

h = &pending;
q = QUEUE_HEAD(h);
// 是否有退出的进程
while (q != h) {
    process = QUEUE_DATA(q, uv_process_t, queue);
    q = QUEUE_NEXT(q);
    QUEUE_REMOVE(&process->queue);
    QUEUE_INIT(&process->queue);
    uv__handle_stop(process);

    if (process->exit_cb == NULL)
      continue;

    exit_status = 0;
    // 获取退出信息，执行上传回调
    if (WIFEXITED(process->status))
```

```
      exit_status = WEXITSTATUS(process->status);


    term_signal = 0;
    if (WIFSIGNALED(process->status))
      term_signal = WTERMSIG(process->status);


    process->exit_cb(process, exit_status, term_signal);
  }
}
```

这就是 nodejs 中进程的整个生命周期。

接下来看看如何以同步的方式创建进程。入口函数是 spawnSync。对应的 c++ 模块是 spawn_sync。过程就不详细说明了，直接看核心代码。

```
void SyncProcessRunner::TryInitializeAndRunLoop(Local<Value> options) {
  int r;


  uv_loop_ = new uv_loop_t;
 // ExitCallback 会把子进程的结构体从 libuv 中移除
  uv_process_options_.exit_cb = ExitCallback;
  r = uv_spawn(uv_loop_, &uv_process_, &uv_process_options_);
  r = uv_run(uv_loop_, UV_RUN_DEFAULT);
}
```

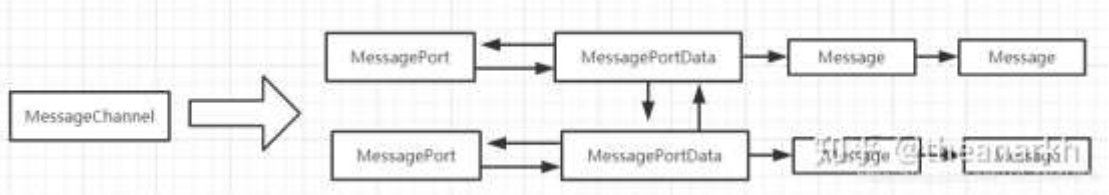
我们看到，对于同步创建进程，nodejs 没有使用 waitpid 这种方式阻塞自己，从而等待子进程退出。而是重新开启了一个事件循环。我们知道 uv_run 是一个死循环，所以这时候，nodejs 主进程会阻塞在上面的 uv_run。直到子进程退出，uv_run 才会退出循环，从而再次回到 nodejs 原来的事件循环。

3.13 线程

这一章以 nodejs 为背景，不分析 libuv 的线程实现，因为他本质是对系统线程库的封装，可以参考后面提供的资料，直接看线程的源码。

nodejs 支持了进程之后，又支持了线程。类似浏览器端的 web worker。因为 nodejs 是单线程的，但是底层又实现了一个线程池，接着实现了进程，又实现了线程。一下变得混乱起来，我们要了解这些功能的实现原理，才能更好地使用他。只有了解线程的实现，才能知道什么时候应该用线程，为什么可以用线程。

线程的实现也非常复杂。虽然底层只是对线程库的封装，但是把它和 nodejs 原本的架构结合起来似乎就变得麻烦起来。下面开始分析创建线程的过程。分析线程实现之前，我们先看一下线程通信的实现，因为线程实现中会用到。通俗来说，他的实现类似一个管道。



1 Message 代表一个消息。

2 MessagePortData 是对 Message 操作的一个封装和对消息的承载。

3 MessagePort 是代表通信的端点。

1 MessageChannel 类似 socket 通信，他包括两个端点。定义一个 MessageChannel 相当于建立一个 tcp 连接，他首先申请两个端点（MessagePort），然后把他们关联起来。

分析完线程通信的实现，我们开始分析线程的实现。nodejs 中 node_worker.cc 实现了线程模块的功能。我们看一下这个模块的定义。

```
NODE_MODULE_CONTEXT_AWARE_INTERNAL(worker,
node::worker::InitWorker)
```

这意味着我们在 js 里执行下面的代码时

```
const exportSomething = internalBinding('worker');
```

拿到的对象是由 node::worker::InitWorker 导出的结果。所以我们看看他导出了什么（只列出核心代码）。

```
void InitWorker(...) {
  Environment* env = Environment::GetCurrent(context);


  {
```

```
    // 定义一个函数模板，模板函数是 Worker::New
  Local<FunctionTemplate> w = env->NewFunctionTemplate(Worker::New);
  // 设置一些原型方法
  env->SetProtoMethod(w, "startThread", Worker::StartThread);
  env->SetProtoMethod(w, "stopThread", Worker::StopThread);
  // 导出这个函数
  Local<String> workerString =
      FIXED_ONE_BYTE_STRING(env->isolate(), "Worker");
  w->SetClassName(workerString);
  target->Set(env->context(),
              workerString,
              w->GetFunction(env->context()).ToLocalChecked()).Check();
}
// 导出额外的一些变量
env->SetMethod(target, "getEnvMessagePort", GetEnvMessagePort);


target
    ->Set(env->context(),
          FIXED_ONE_BYTE_STRING(env->isolate(), "isMainThread"),
          Boolean::New(env->isolate(), env->is_main_thread()))
    .Check();
```

```
}
```

翻译成 js 大概是

```
function New() {}
New.prototype = {
    startThread,StartThread,
    StopThread: StopThread,
    ...
}
module.exports = {
    Worker: New,
    getEnvMessagePort: GetEnvMessagePort,
    isMainThread: true | false
}
```

了解了 c++导出的变量，我们看看 js 层的封装。

```
class Worker extends EventEmitter {
  constructor(filename, options = {}) {
    super();

    this[kHandle] = new Worker(url,...));
    this[kPort] = this[kHandle].messagePort;
```

```
    const { port1, port2 } = new MessageChannel();
    this[kPublicPort] = port1;
    this[kPort].postMessage({
      type: messageTypes.LOAD_SCRIPT,
      filename,
      workerData: options.workerData,
      publicPort: port2,
    }, [port2]);

    this[kHandle].startThread();
  }
}
```

下面我们逐步分析上面的代码。

1 this[kHandle] = new Worker(url,…));

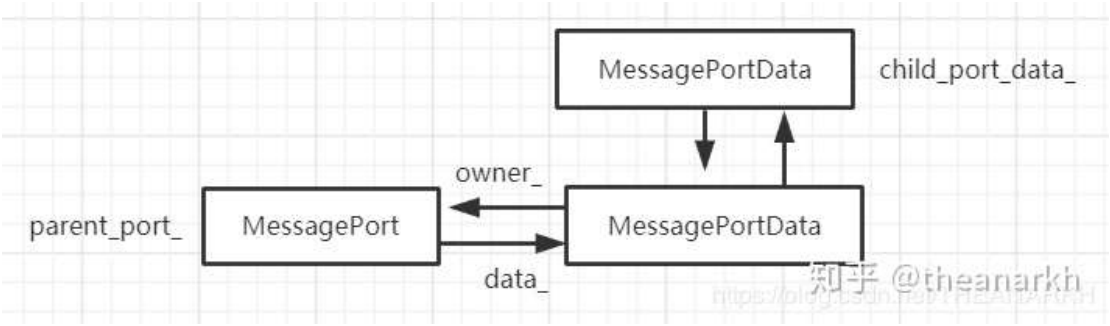
根据上面的分析我们知道 Worker 函数对应的是 c++层的 New 函数。所以我们看看 New 函数做了什么 。

```
void Worker::New(const FunctionCallbackInfo<Value>& args) {
  Environment* env = Environment::GetCurrent(args);
  Isolate* isolate = args.GetIsolate();
  //  忽略一系列参数处理
```

```
  Worker* worker = new Worker(env,
                                        args.This(),
                                        url,
                                        per_isolate_opts,
                                        std::move(exec_argv_out),
                                        env_vars);
}
```

是对 Worker 类的封装。

```
Worker::Worker(...) {
 // 申请一个端点
  parent_port_ = MessagePort::New(env, env->context());
  //   申请一个 MessagePortData 类对象
  child_port_data_ = std::make_unique<MessagePortData>(nullptr);
  //   使得两个 MessagePortDatat 互相关联
  MessagePort::Entangle(parent_port_, child_port_data_.get());
  // 设置 messagePort 属性为 parent_port_ 的值
  object()->Set(env->context(),
                env->message_port_string(),
                parent_port_->object()).Check();

}
```



所以 new Worker 就是定义了一个对象，并初始化了三个属性。

2 this[kPort] = this[kHandle].messagePort;

我们看到 new Worker 的时候定义了 messagePort 这个属性。他对应 c++ 的 parent_port_ 属性。是初始化时，父线程和子线程通信的一端。另一端在子线程中维护。

3 const { port1, port2 } = new MessageChannel();

申请两个可以互相通信的端点。用户后面的线程间通信。

4 保存主线程端的端点，后续用于通信。并且给子线程发送一些信息。告诉子线程通信的端口（子线程端的）和执行的 js 文件名。

```
this[kPublicPort] = port1;
this[kPort].postMessage({
```

```
        type: messageTypes.LOAD_SCRIPT,
        filename,
        workerData: options.workerData,
        publicPort: port2,
    }, [port2]);
```

在通信中，至少要存在两个端，假设 x 和 y。那么 x.postMessage(...)，就是给 y 发信息。但是根据图二，我们发现只有一个 MessagePort。所以上面代码中的 postMessage 只是把消息缓存到消息队列里（MessagePortData 中）。

5 this[kHandle].startThread();

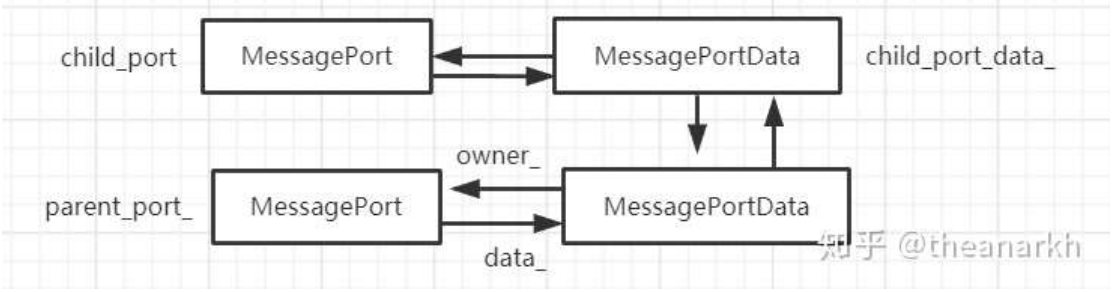
开始启动线程。根据 c++层导出的变量我们知道 startThread 对应函数是 StartThread。

```
void Worker::StartThread(const FunctionCallbackInfo<Value>& args) {

        Worker* w;

    uv_thread_create_ex(&w->tid_, &thread_options, [](void* arg) {
          Worker* w = static_cast<Worker*>(arg);
           w->Run();
    }, static_cast<void*>(w)), 0
```

```
}
```

创建一个线程，然后在里面执行 Run 函数。Run 函数非常复杂，下面列出几个步骤。

1 创建一个通信端点。和图二的完成关联。这样子线程就可以处理刚才缓存的消息了。



2 处理刚才缓存的消息。把 parentPort 设置成父线程传过来的端点。子线程就可以和父线程通信了。

```
const { Worker, isMainThread, parentPort } = require('worker_threads');
if (isMainThread) {
  const worker = new Worker(__filename);
  worker.once('message', (message) => {
    ...
  });
  worker.postMessage('Hello, world!');
} else {
```

```
  parentPort.once('message', (message) => {
    parentPort.postMessage(message);
  });
}
```

3 执行主线程的 js 文件（即 new Worker 时传进来的参数，有很多种形式）。

4 开启新的事件循环，即子线程和主线程是分开的两个事件循环。

以上就是 nodejs 中线程的大致原理。

3.14 网络

本章主要介绍 tcp，按照 socket 网络编程的顺序，分析整个过程。Libuv 中，tcp 相关功能使用 uv_tcp_t 结构体表示。看一下怎么初始化一个 uv_tcp_t 结构体。

1 申请一个 socket

```
// 初始化 uv_tcp_t 结构体
int uv_tcp_init(uv_loop_t* loop, uv_tcp_t* tcp) {
  // 未指定协议
  return uv_tcp_init_ex(loop, tcp, AF_UNSPEC);
}
// 省略一些不重要的逻辑
int uv_tcp_init_ex(uv_loop_t* loop, uv_tcp_t* tcp, unsigned int flags) {
```

```
  // 初始化流部分，参考分析流章节
  uv__stream_init(loop, (uv_stream_t*)tcp, UV_TCP);
  // 获取一个 socket
  uv__socket(domain, SOCK_STREAM, 0);
  // 设置选项和保存 socket 的文件描述符到 io 观察者中
  uv__stream_open((uv_stream_t*) tcp, sockfd, flags);
}
```

uv_tcp_init_ex 函数主要是申请一个 socket，然后把 socket 对应的文件描述符和回调封装到 io 观察者中。但是还没有插入事件循环的观察者队列。

2 绑定地址

```
int uv__tcp_bind(uv_tcp_t* tcp,
                 const struct sockaddr* addr,
                 unsigned int addrlen,
                 unsigned int flags) {
  int err;
  int on;
  // 设置流可读写
  err = maybe_new_socket(tcp,
                         addr->sa_family,
                         UV_STREAM_READABLE | UV_STREAM_WRITABLE);
  // 设置 SO_REUSEADDR 属性
```

```
  on = 1;
  if (setsockopt(tcp->io_watcher.fd, SOL_SOCKET, SO_REUSEADDR, &on,
sizeof(on)))
    return -errno;

  errno = 0;
  // 绑定地址到 socket
  if (bind(tcp->io_watcher.fd, addr, addrlen) && errno != EADDRINUSE) {
    // ...
  }
  tcp->delayed_error = -errno;
  // 设置已经绑定标记
  tcp->flags |= UV_HANDLE_BOUND;

  return 0;
}
```

3 listen

```
int uv_tcp_listen(uv_tcp_t* tcp, int backlog, uv_connection_cb cb) {
  static int single_accept = -1;
  unsigned long flags;
  int err;
```

```
  // 是否 accept 后，等待一段时间再 accept，否则就是连续 accept
  if (single_accept == -1) {
    const char* val = getenv("UV_TCP_SINGLE_ACCEPT");
    single_accept = (val != NULL && atoi(val) != 0);   /* Off by default. */
  }

  if (single_accept)
    tcp->flags |= UV_TCP_SINGLE_ACCEPT;
  // 设置 socket 为监听状态
  if (listen(tcp->io_watcher.fd, backlog))
    return -errno;
  // 设置有连接完成时的回调
  tcp->connection_cb = cb;
  tcp->flags |= UV_HANDLE_BOUND;
  // 有连接到来时 libuv 层的回调，connection_cb 为业务回调，被 libuv 调用
  tcp->io_watcher.cb = uv__server_io;
  // 注册 io 观察者到事件循环的 io 观察者队列，设置等待读事件
  uv__io_start(tcp->loop, &tcp->io_watcher, POLLIN);

  return 0;
}
```

Listen 函数的逻辑主要是

1 设置 socket 为监听状态，即可以等待建立连接

2 封装 io 观察者，然后注册到事件循环

3 保存相关上下文，比如业务回调函数

这时候服务器已经启动，如果这时候有一个连接到来（已完成三次握手）。就会执行 uv__server_io。

```
void uv__server_io(uv_loop_t* loop, uv__io_t* w, unsigned int events) {
          uv_stream_t* stream;
          int err;
    // 摘下一个已完成三次握手的连接
    err = uv__accept(uv__stream_fd(stream));
    // 记录拿到的通信 socket 对应的 fd
    stream->accepted_fd = err;
    // 执行上传回调
    stream->connection_cb(stream, 0);
    // accept 成功，则等待用户消费 accepted_fd 再 accept，这里撤销事件
    if (stream->accepted_fd != -1) {
      uv__io_stop(loop, &stream->io_watcher, POLLIN);
      return;
    }
  }
}
```

当有连接到来，libuv 会调用 accept 摘下一个连接，然后通知用户，并且停

止 accept。等到用户消费 accepted_fd。用户可以通过 uv_accept 消费 accepted_fd。比如

```
uv_tcp_t *client = (uv_tcp_t*) malloc(sizeof(uv_tcp_t));
uv_tcp_init(loop, client);
uv_accept(server, (uv_stream_t*) client);
```

首先申请一个新的 uv_tcp_t 结构体，该新的结构体用来表示和客户端通信，之前的那个是用于处理连接，新的用来处理通信。

```
int uv_accept(uv_stream_t* server, uv_stream_t* client) {
int err;
// 设置流的标记，保存文件描述符到流上
uv__stream_open(client,server->accepted_fd,UV_HANDLE_READABLE|UV_HAN
DLE_WRITABLE);
// 消费完了，恢复为-1
server->accepted_fd = -1;
// 重新注册时间，处理下一个连接，有的话
uv__io_start(server->loop, &server->io_watcher, POLLIN);
  return err;
}
```

libuv 作为客户端的过程也是类似的，首先申请一个 socket，然后把 socket 对应的 fd 和回调封装成 io 观察者，然后注册到 libuv，等三次握手成功后，会执行回调函数。

四、参考资料

1 nodejs 源码，文档

2 libuv 源码、文档

3 linux，线程早期源码

五、资源

https://github.com/theanarkh

https://www.processon.com/view/link/5e115e7ce4b0bcfb7332f36b

https://www.processon.com/view/link/5e10a603e4b009af4a5c3da6

欢迎关注编程杂技，分享技术，交流技术。