libuv API 文档

http://docs.libuv.org/en/v1.x

概述：

libuv是一个针对异步IO的跨平台库。本库住要是为Node.js设记的，不过Luvit,Julia,pyuv和一些其他的工程也使用了本库。

注意：如果您在本文档中发现了错误，请通过推送请求来帮助我们改进。

特性：

- 支持基于epoll(Unix)，kqueue，IOCP(Windows)，event ports的事件循环
- 异步TCP、UDP套接字
- 异步DNS解析
- 异步文件和文件系统操作
- 文件系统事件
- ANSI escape code controlled TTY
- 基于Unix domain sockets或者命名管道的进程间通信以及socket共享
- 子进程
- 线程池
- 信号量
- 高分辨率时钟
- 线程和线程同步

下载：

点击这里下载。

说明文档：

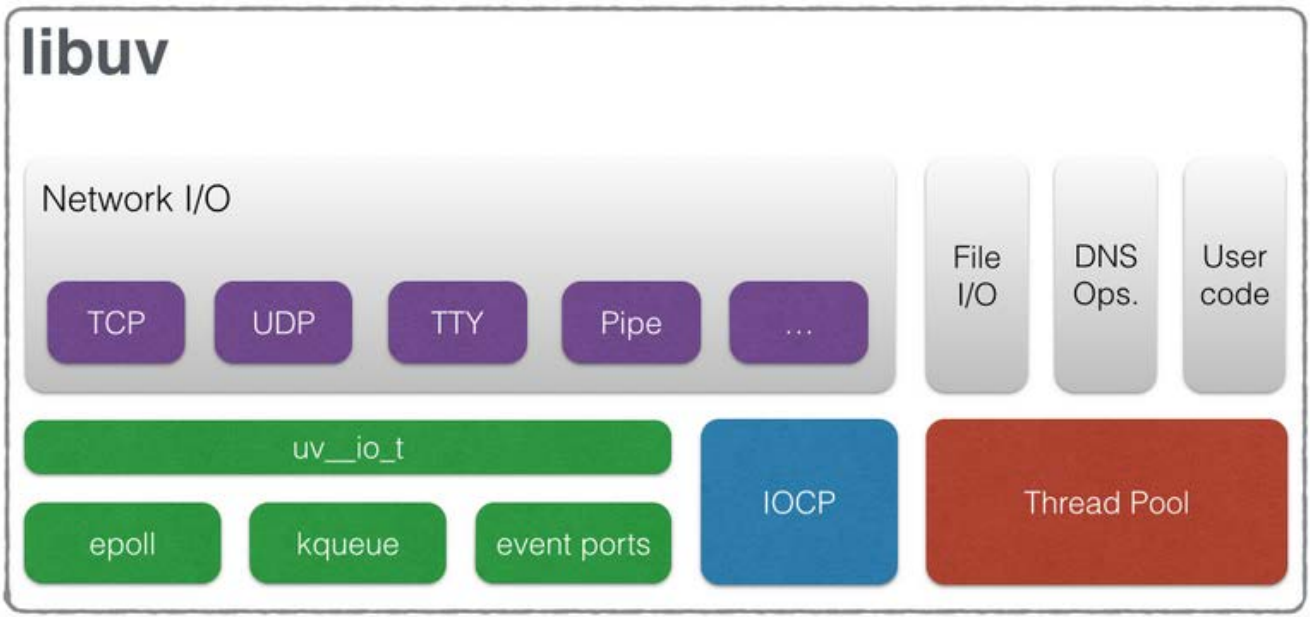
- 设计概述
- 错误处理
- 版本检测宏与函数
- uv_loop_t——事件循环
- uv_handle_t——基础句柄
- uv_req_t——基础请求
- uv_timer_t——定时器句柄
- uv_prepare_t——预处理handle
- uv_check_t——check handle
- uv_idle_t——空转handle
- uv_async_t——异步handle
- uv_poll_t——poll handle
- uv_signal_t——信号量handle
- uv_process_t——进程handle
- uv_stream_t——流handle
- uv_tcp_t——tcp handle
- uv_pipe_t——管道handle
- uv_tty_t——TTY handle
- uv_udp_t——UDP handle
- uv_fs_event_t——FS Event handle
- uv_fs_poll_t——FS Poll handle
- Filesystem operation 文件系统操作
- Thread pool work scheduling 线程池调度
- DNS utility functions DNS功能函数
- Shared library handing 动态库处理
- Threading and synchronization utility 线程同步
- Miscellaneous utility 其他工具

设计概述

libuv是一个跨平台的库，最初是为node.js设置。这个库是针对事件循环的异步IO模型而设计的。

本库不仅仅只是简单的提供各种IO轮询机制的抽象封装：‘handle’和’stream’为套接字(sockets)和其它实现(entities)提供更高层次的抽象封装；同时也提供跨平台文件IO以及线程功能。

下面的图表表明了libuv的子模块的组成



句柄和请求(Handles and requests)

libuv提供用户2个抽象的结构去结合事件循环(event loop)使用：句柄和请求。

句柄(handles)代表长寿命对象，能够在活动中进行特定的操作。例如：当一个prepare handle激活时，它的回调函数数将会在每一个循环迭代中被调用一次，而一个TCP服务handle的连接回调函数在每一个新的连接发生时会被调用。

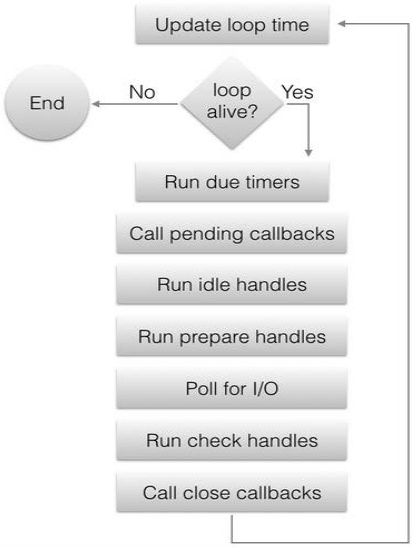
请求(Requests)通常是代表操作的短寿命对象。这些操作能够通过一个句柄(handle)进行——写请求被用来在一个handle上写入数据；也可以是独立的——getaddrinfo请求不需要handle，直接运行在循环上。

I/O循环(I/O loop)

I/O循环(或者事件循环)是libuv的核心部分。它建立了环境供所有的I/O操作，并且是绑定在单线程上的。使用者可以开多个事件循环(event loop)，不过每个循环运行在不同的线程上。libuv的事件循环(或者其他的涉及到handle或者循环的api)不是线程安全的，除非有特殊说明。

事件循环遵循通常的单线程异步I/O方法：所有的(网络)I/O通过非阻塞的sockets进行，并使用特定平台的最有效的轮询方法——unix平台的epoll，OSX平台的kqueue和其他BSDs，SunOS的event ports以及windows下的IOCP。作为循环迭代(loop iteration)的一部分，循环会为已经被添加到循环的I/O活动中断，并使用回调函数表明当前socket的状态(可读、可写、断开)，这样handles就可以执行对应的I/O操作。

下图表明了一个迭代的所有阶段：



1.循环更新当前时间。循环在计时开始时缓存当前间，以减少时间相关的系统调用次数数。

2.如果循环是存活的(alive)，一个迭代就开始了，否则会立刻退出。那么，什么情况下认为循环是alive？如果一个循环有激活、ref'd handles,激活的请求或者正在关闭的handles，就认为是alive

3.预定的定时器运行。所有活动的定时器，如果预定的时间在loop概念上的“现在”之前，它们的回调函数被调用

4.正在等待的回调被调用。大部分情况下所有的I/O回调将会在I/O轮询之后被调用。特殊情况下，回调被推迟到下一个迭代。如果上一个迭代推迟了任何的I/O回调，将在此时被调用

5.空转(Idle)handle回调被调用。尽管名字是空转，但是将会在每一个迭代中被调用

6.Prepare handle回调调用。Prepare回调在loop为I/O中断之前被调用

7.计算轮询超时。在中断之前，loop会计算中断超时。有以下的规则：
- 如果loop使用了UV_RUN_NOWAIT标记，超时是0
- 如果loop即将停止(uv_stop()被调用)，超时是0
- 如果没有活动的handle或者requests，超时是0
- 如果有任意的空转handle，超时是0
- 如果有任何handles需要被关闭，超时是0
- 以上条件不满足，使用最近定时器的超时，如果没有活动的定时器，超时是无限大

8.I/O中断。所有的I/O相关的handle，监控指定的文件描述符进行读或写操作此时得到回调。

9.check handel回调被调用。check handles通常与prepare handle匹配。

10.关闭回调被调用。被uv_close()函数关闭的handles的close回调函数将会被调用

11.在UV_RUN_ONCE的标记下。有可能I/O回调没有被掉用，而定时器回调被调用

12.迭代结束。如果是在UV_RUN_NOWAIT或者UV_RUN_ONCE模式下，迭代会结束同时uv_run()函数会返回。如果是UV_RUN_DEFAULT模式，迭代会继续从start开始，当然如果loop不是alive状态，还是会结束

重要：libuv通过线程池实现异步文件I/O，但是网络I/O则是在loop所在的单线程

注意：windows和unix平台下的轮询机制是不同的，libuv统一了二者的执模式。

文件I/O

与网络I/O不同的是，没有平台原生的文件I/O可以使用，所以libuv目前对阻塞文件I/O的实现是线程池。libuv目前用

一个全局的线程池，所有的loop都可以以队列的方式使用。3种操作在这个线程池上运行

- 文件系统操作
- DNS函数
- 用户通过uv_queue_work()执行指定的代码

警告：参考 线程池调度 部分获取跟多细节，要记住线程池的大小相当有限。

错误处理

在libuv中，错误的编号通常为负。根据经验，无论是状态参数还是API的返回值，负数总是代表错误

注意：实现细节：在Unix中错误代码是-errno，在windows中是由ibuv定义的任意负数

错误常数

UV_E2BIG
: 参数列表太长

UV_EACCES
: 权限被拒绝

UV_EADDRINUSE
: 地址已被使用

UV_EADDRNOTAVAIL
: address not available 地址不可用

UV_EAFNOSUPPORT
: address family not supported 不支持指定的地址族

UV_EAGAIN
: resource temporarily unavailable 资源暂时不可用

UV_EAI_AADDRFAMILY
: address family not supported 不支持指定的地址族

UV_EAI_AGAIN
: temporary failure 暂时失败

UV_EAI_BADFLAGS
: bad ai_flags value

UV_EAI_BADHINTS
: invalid value for hints 提示无效值

UV_EAI_CANCELED
: request canceled 请求取消

UV_EAI_FAIL
: permanent failure 永久性故障

UV_EAI_FAMILY
: ai_family not supported

UV_EAI_MEMORY
: out of memory

UV_EAI_NODATA
: no address

UV_EAI_NONAME
: unknown node or service

UV_EAI_OVERFLOW
: argument buffer overflow 溢出

UV_EAI_PROTOCOL
: resolved protocol is unknown 解决协议是未知的

UV_EAI_SERVICE
: service not available for socket type

UV_EAI_SOCKTYPE
: socket type not supported

UV_EALREADY
: connection already in progress 已经开始链接

UV_EBADF
: bad file descriptor 文件描述错误

UV_EBUSY
: resource busy or locked 资源忙或者被锁

UV_ECANCELED
: operation canceled 取消操作

UV_ECHARSET
: invalid Unicode character 无效的unicode字节

UV_ECONNABORTED
: software caused connection abort 软件造成连接中止

UV_ECONNREFUSED
: connection refused 链接被拒绝

UV_ECONNRESET
: connection reset by peer 连接复位

UV_EDESTADDRREQ
: destination address required 需要目标地址

UV_EEXIST
: file already exists

UV_EFAULT
: bad address in system call argument 系统调用参数中的错误地址

UV_EFBIG
: file too large
UV_EHOSTUNREACH
: host is unreachable 无法访问主机
UV_EINVAL
: interrupted system call 中断系统调用
UV_EINVAL
: invalid argument
UV_EIO
: i/o error
UV_EISCONN
: socket is already connected
UV_EISDIR
: illegal operation on a directory 目录上的非法操作
UV_ELOOP
: too many symbolic links encountered 符号链接冲突
UV_EMFILE
: too many open files
UV_EMSGSIZE
: message too long
UV_ENAMETOOLONG
: name too long
UV_ENETDOWN
: network is down
UV_ENETUNREACH
: network is unreachable
UV_ENFILE
: file table overflow 文件表溢出
UV_ENOBUFS
: no buffer space available
UV_ENODEV
: no such device
UV_ENOENT
: no such file or directory
UV_ENOMEM
: not enough memory
UV_ENONET
: machine is not on the network
UV_ENOPROTOOPT
: protocol not available
UV_ENOSPC
: no space left on device
UV_ENOSYS
: function not implemented
UV_ENOTCONN
: socket is not connected
UV_ENOTDIR
: not a directory
UV_ENOTEMPTY
: directory not empty
UV_ENOTSOCK
: socket operation on non-socket
UV_ENOTSUP
: operation not supported on socket
UV_EPERM
: operation not permitted 操作不允许
UV_EPIPE
: broken pipe 断开的管道
UV_EPROTO
: protocol error
UV_EPROTONOSUPPORT
: protocol not supported
UV_EPROTOTYPE
: protocol wrong type for socket
UV_ERANGE
: result too large
UV_EROFS
: read-only file system
UV_ESHUTDOWN
: cannot send after transport endpoint shutdown

UV_ESPIPE
invalid seek 无效的寻找
UV_ESRCH
no such process
UV_ETIMEDOUT
connection timed out 连接超时
UV_ETXTBSY
text file is busy
UV_EXDEV
cross-device link not permitted 不允许跨设备链接
UV_UNKNOWN
unknown error
UV_EOF
end of file
UV_ENXIO
no such device or address
UV_EMLINK
too many links

API

const char* uv_strerror(int err)

返回错误代码。如果错误代码未定义，将会导致少量的内存泄露（参考uv__unknown_err_code函数)

const char* uv_err_name(int err)

返回错误名字。如果是未知的错误代码，将会导致少量的内存泄露

版本检测宏与函数

从1.0.0版本开始，libuv遵循语义版本控制方案。这意味新的APIs可以在主版本跟新中被引进。在本节你会发现所有的宏和函数允许你有条件的编写代码，以此实现多版本libuv的的支持

宏

UV_VERSION_MAJORO

libuv的主版本号 ： 1.2.3

UV_VERSION_MINOR

libuv的小版本号 ： 1.2.3

UV_VERSION_PATCH

libuv的补丁版本号 ： 1.2.3

UV_VERSION_IS_RELEASESE

定义为1代表发布版本，0代表开发版本(snapshot)

UV_VERSION_SUFFIX

libuv版本后缀。开发版本，比如发布版本可能有‘rc’后缀

UV_VERSION_HEX

将版本以integer形式返回，每8bit代表一个版本号，eg：1.2.3 返回 0x010203

UV_VERSION_HEXNG

将数字版本转换为字符串

函数

unsigned int uv_version(void)

返回UV_VERSION_HEX

const char* uv_version_string(void)

返回UV_VERSION_STRING

uv_loop_t —事件循环

事件循环是libuv的核心功能。主要负责对不同来源事件的i/o轮询以及回调函数的调用

数据类型

uv_loop_t

loop的数据结构，在uv.h中定义

uv_run_mode

通过uv_run()函数来运行loop时的模式，在uv.h文件中定义

```
typedef    enum {
               UV_RUN_DEFAULT = 0,
               UV_RUN_ONCE,
               UV_RUN_NOWAIT
} uv_run_mode;
```

void (*uv_walk_cb)(uv_handle_t* handle, void* arg)

传给uv_walk()函数的回调函数的指针的定义

公有数据

void* uv_loop_t.data

用来存放任意的用户定义的数据。libuv不使用这个数据，在uv_loop_init()函数中将其设为null，在debug版本的uv_loop_close()函数中将其设置memset(loop, -1, sizeof(*loop))；

API

int uv_loop_init(uv_loop_t* loop)

初始化传入的uv_loop_t 结构体

int uv_loop_configure(uv_loop_t* loop, uv_loop_option option, ...)

在版本1.0.2中新增

设置附加的循环配置。使用者通常需要在uv_run()函数之前调用本函数，除非有特殊申明。

返回0代表成功，返回UV_E*错误代码代表失败。要做好处理返回值为UV_ENOSYS的准备，这说明当前平台不支持传入的配置

支持的配置如下：

· UV_LOOP_BLOCK_SIGNAL：当轮询一个新的事件时阻塞一个信号量。

int uv_loop_close(uv_loop_t* loop)

释放所有的循环内部的资源。只能在loop完成运行并且所有打开的handle以及requests已经被关闭，否则会返回UV_EBUSY。当本函数返回之后，用户可以释放为loop分配的内存了。

uv_loop_t* uv_default_loop(void)

返回已经初始化的默认loop。如果分配内存失败，就会返回null

这个函数只是方便获取一个全局的loop，获取的loop与的通过uv_loop_initial()初始化的loop没有任何区别。所以，默认的loop可以通过uv_loop_close()函数关闭，与它相关的资源也会被释放。

int uv_run(uv_loop_t* loop, uv_run_mode mode)

本函数开始一个事件循环，根据不同的模式有不同的表现

UV_RUN_DEFAUT:一直运行到没有激活的和被引用的句柄以及请求。如果uv_stop()被调用并且仍有活动的handle或者requests，返回非0值。其他情况下返回0

UN_RUN_ONCE：轮询i/o一次。注意，如果没有等待的回调函数，本函数会阻塞。如果没有活动的handles和rquests返回0，否则返回非0值。

UV_RUN_NOWAIT：轮询i/o一次，但是如果没有等待的回调函数就不阻塞。如果没有剩下的活动的handle以及requests，返回值0，否则非0，意味着需要在之后再次运行循环。

int uv_loop_alive(const uv_loop_t* loop)

如果loop没有活动的事件或者request返回非0值

void uv_stop(uv_loop_t* loop)

停止循环，会使uv_run()函数尽快返回。这不会在下一个循环迭代开始之前起作用。如果在i/o阻塞之前调用这个函数，循环不会因为i/o阻塞

size_t uv_loop_size()

返回uv_loop_t结构体的大小。

int uv_backend_fd(const uv_loop_t* loop)

获取后台文件描述符，只有kqueue, epoll and event ports支持

int **uv_backend_timeout**(const uv_loop_t* loop)

获取轮询的超时。返回值是毫秒，或者-1代表没有超时设置

uint64_t **uv_now**(const uv_loop_t* loop)

以毫秒返回当时时间。时间戳在每次循环迭代代的开始时被缓存起来。
时间戳在一个任意的时间点递增。不要对开始的时间做任何假设设。
注意：使用 uv_hrtime()来获取更高精度的时间

void **uv_update_time**(uv_loop_t* loop)

更新loop定义的“当前时间”。libuv在每次迭代开始的时候缓存当前时间。以此来减少不需要的事件相关的系统调用。
除非回调函数需要阻塞很长一段时间，否则用户通常并不需要调用此函数

void **uv_walk**(uv_loop_t* loop, uv_walk_cb walk_cb, void* arg)

Walk the list of handles： walk_cb将会被调用并传入arg参数

uv_handle_t——基础句柄（Base handle）

uv_handle_t是libuv的所有句柄类型的基础
由于结构是一致的，所以libuv的所有句柄结构都能转换为uv_handle_t。

数据类型

uv_handle_t
libuv的基础句柄类型

uv_any_handle
所有句柄类型的联合体

void (*uv_alloc_cb)(uv_handle_t* handle, size_t suggested_size, uv_buf_t* buf)
传递给uv_read_start()以及uv_udp_recv_start()函数的回调函数的函数指针类型。用户必须提供uv_buf_t结构体的空间。目前建议的大小是65536，但是并不一定需要遵守。将大小设为0将会在uv_udp_recv_cb以及uv_read_cb回调中触发UV_ENOBUFS的错误

void (*uv_close_cb)(uv_handle_t* handle)
传给uv_close()函数的回调函数的函数指针

公有成员

uv_loop_t* uv_handle_t.loop
运行当前handle的loop的指针，只读。

void* vu_handle_t.data
指向用户自定义的数据，libuv不使用该成员

API

int **uv_is_active**(const uv_handle_t* handle)
如果handle是活动的，返回非0值，否则返回0。活动代表的意思据handle的类型而有所不同
uv_async_t handle总是活动的，并且不能是非活动的，除非关闭（uv_close）
uv_pipe_t, uv_tcp_t, uv_udp_t等handle——基本所有与i/o相关的句柄——当涉及到i/o活动时是活动的，比如写、读、连接、断开连接等
uv_check_t,uv_idle_t, uv_timer_t等句柄，当uv_check_start()、uv_idle_start()等被调用时，是激活的
经验：如果一个uv_foo_t有一个对应的uv_foo_start()函数数，那么在函数被调用时，handle是激活的，同样，uv_foo_stop()将会使其再次变为非激活状态

int **uv_is_closing**(const uv_handle_t* handle)
如果handle已经被关闭或者正在被关闭，返回非0值，否则返回0
注意：本函数只能在handle初始化以及close回调函数调用期间被调用。

void **uv_close**(uv_handle_t* handle, uv_close_cb close_cb)
请求关闭handle。close_cb将会在此函数之后被异步调用。本函数必须在handle内存释放之前被调用
文件描述的handle将会被立刻关闭，但是close_cb仍然会在下一个循环迭代中调用，以此来给用机会释放所有相关的资源。
对于正在处理请求的handle，比如uv_connect_t或者uv_write_t，请求将会取消，对应回调函数会被异步调用，并将status参数设为UV_ECANCELED。

void **uv_ref**(uv_handle_t* handle)
引用传入的handle。引用是幂等的，这表示如果一个handle已经被引用了，调用这个函数将不起作用。
参考Reference counting

void **uv_unref**(uv_handle_t* handle)
取消引用传入的handle。如果没有被引用，本函数不起作用
参考Reference counting

int **uv_has_ref**(const uv_handle_t* handle)
如果被引用了，返回非0值，否 返回0

size_t **uv_handle_size**(uv_handle_type type)
返回传入的handle类型的大小

其他 API

接下来的函数接受uv_handle_t类型的参数，但是只对部分handle类型起作用int

uv_send_buffer_size(uv_handle_t* handle, int* value)

获取或者设置操作系统用于socket的发送数据缓存的大小
如果 *value==0，将会返回当前设置的大小，否则设置新的大小
本函数适用于unix平台的下的TCP,UDP,管道handle，以及windows平台的TCP,UDP
注意：linux将会设置为value的两倍，并返回原来的大小

int **uv_recv_buffer_size**(uv_handle_t* handle, int* value)

获取或者设置系统用于socket的接收缓存的大小
如果 *value==0，将会返回当前设置的大小，否则设置新的大小
本函数适用于unix平台的TCP，UDP，管道handle，以及windows平台的TCP，UDP
注意：linux将会设置为value的两倍，并返回原来的大小

int **uv_fileno**(const uv_handle_t* handle, uv_os_fd_t* fd)

获取平台相关的文件描述符
支持以下句柄：TCP，pipes，TTY，UDP以及poll。传入任何其他handle将会返回UV_EINVA
如果一个handle还没有附加一个文件描述或者已被关闭，本函数会返回UV_EBADF
警告：调用本函数必须非常小心，因为libuv假设文件描述是正确的，所以任何改变都可能导致错误

引用记数

libuv的事件循环将会一直运行行直到没有活动的以及被引用的handle。 通过解引用已被引用的handle，用户可以使loop提前退出，例如在uv_timer_start()之后调用uv_unref()

一个handle可以是被引用或没有被引用，引用记数方案并不使用记数器，所以所有的操作都是幂等的

当被默认方式激活时，所有的handles都是被引用的，参考**uv_is_active()**

uv_req_t — Base request

uv_req_t 是libuv 有 求类型的 类型

一 的结 libuv的 有 求类型 可以 为uv_req_t。下 的API 数 用于 有的 求类型。

数据类型

uv_req_t
libuv的 求结 类型

uv_any_req
有 求类型的

公有成员

void* uv_req_t.data
指 用 定 数据，libuv不 用 成员

uv_req_type uv_req_t.type
求的类型，只
typedef enum
UV_UN NO N_ E 0,
UV_ E ,
UV_ ONNE ,
UV_ I E,
UV_S U O N,
UV_U P_SEN ,
UV_FS,
UV_ O ,
UV_GE A INFO,
UV_GE NA EINFO,
UV_ E _ PE_P IVA E,
UV_ E _ PE_ A ,
uv_req_type

API

int **uv_cancel**(uv_req_t* req)
取 一个等待 的 求。如果 求 在被 或者 成了 , 数 行
成 返回0， 返回错误代码
只支持uv_fs_t,uv_getaddrinfo_t, uv_getnameinfo_t 以及 uv_work_t句柄 求的取
被取 的 求的回 数将会在 的一段时 被 用。在回 数被 用 , 求的 存是不 的。
下 取 的 是 被传到回 数的
uv_fs_t 求的req result 成员被设置为UV_E AN E E
uv_work_t, uv_getaddrinfo_t 或者 c type uv_getnameinfo_t 求的回 数将会被 用， 时status参数设置为 UV_E AN E E .

size_t **uv_req_size**(uv_req_type type)
返回对 的 求类型的大小

uv_time_t——定时器句柄(imer handle)

定时器handle用来在一段时 用预定的回 数

数据类型

uv_timer_t
定时器handle类型

void (*uv_timer_cb)(uv_timer_t* handle)
传 uv_timer_start() 数的回 数的 数指 定

公有数据

无

API

int **uv_timer_init**(uv_loop_t* loop, uv_timer_t* handle)
初始化handle

int **uv_timer_start**(uv_timer_t* handle, uv_timer_cb cb, uint _t timeout, uint _t repeat)
开 定时器。 时以及 是
如果 时是0，回 数将在下一 代 。如果repeat不是0，回 数将会在timeout 被 用，接 repeat 再 被 用。

int **uv_timer_stop**(uv_timer_t* handle)
停止定时器。回 数将不会再被 用

int **uv_timer_again**(uv_timer_t* handle)
停止定时器， and if it is repeating restart it using the repeat value as the timeout。如果定时器 有 , 返回UV_EINVA

int **uv_timer_set_repeat**(uv_timer_t* handle, uint _t repeat)
设置一个值作为 , 为 。定时器将会 定的 行，不管回 的 行 需的时 , 在时 的 下 常的定时器 。
例如，如果 0 的定时器 行了1 , 将预 再 行33 。如果 的 务 的 33 , 定时器将会 可 的 行
注意：如果 时 (repeat)是在定时器回 数中设置的， 不会 作用。如果定时器 有repeat， 将会停止。如果 在 行中， 的 时 将会被用来 行下一 。

uint _t **uv_timer_repeat**(const uv_timer_t* handle)
获取定时器的repeat值

参考：**uv_handle_t**的API 可用

uv_prepare_t——预　　handle（Prepare handle）

预　handle将会在　　代　i o　　用一

数据类型

uv_prepare_t
　预　handle结　类型

void (*uv_prepare_cb)(uv_prepare_t* handle)
　传　uv_prepare_start()　数的回　数的　数指　类型

公有成员

无

API

int **uv_prepare_init**(uv_loop_t* loop, uv_prepare_t* handle)
　初始化handle

int **uv_prepare_start**(uv_prepart_t* handle, uv_prepare_cb cb)
　开始handle，　传　回　数

int **uv_prepare_stop**(uv_prepare_t* handle)
　停止handle，　回　数将不会再被　用

参考：**uv_handle_t**的API　可用

uv_check_t—— heck handle

heck handle将会在 代 i o 用回 数

数据类型

uv_check_t
check handle结 类型

void (*uv_check_cb)(uv_check_t* handle)
传 uv_check_start() 数的回 数的 数指 定

公有成员

无

参考：**uv_handle_t**结

API

int **uv_check_init**(uv_loop_t* loop, uv_check_t* handle)
初始化handle

int **uv_check_start**(uv_check_t* handle, uv_check_cb cb)
开始handle， 传 回 数指

int **uv_check_stop**(uv_check_t* handle)
停止handle，回 数将不会再被 用

参考：**uv_handle_t**的API 用

uv_idle_t—— handle

handle将会在 代的uv_prepare_t handle 用回 数

注意: handle 预 (prepare)handle的区 是，当有 的 handle时， 将会 取 时 ，不是阻塞的i o

： 管名字是 (idle),但 不是 当 是 时 用回 数， 是在 一 代的时 会用。

数据类型

uv_idle_t
handle结 类型

void (*uv_idle_cb)(uv_idle_t* handle)
传 uv_idle_start() 数的回 数的 数指 定

公有成员

无

参考：uv_handle_t (包含uv_handle_t的成员)

API

int **uv_idle_init**(uv_loop_t* loop, uv_idle_t* handle)
初始化handle

int **uv_idle_start**(uv_idle_t* handle, uv_idle_cb cb)
开始 handle， 传 回 数指

int **uv_idle_stop**(uv_idle_t* handle)
停止 handle，回 数不会再被 用

参考：uv_handle_t的API 用

uv_async_t——　　handle

　　handle　　用　在　一个　程中通知(　　wakeup)　件

数据类型

uv_async_t
　　handle结　　类型

void(*uv_async_cb)(uv_async_t* handle)
　传　　uv_async_init()　数的回　　数的　数指　类型

公有成员

无

参考：uv_handle_t的成员

API

int uv_async_init(uv_loop_t* loop, uv_asychn_t* handle, uv_async_cb cb)

　　初始化　　handle，　传　回　　数指

　　注意：　　handle不　的是，　个初始化　数将　接　　handle

int uv_async_send(uv_async_t* handle)
　　(wakeup)　　用　　handle的回　　数

　　注意：在　意　程　用本　数　是　　的。回　　数将会在　件　　的　程被　用

　　：libuv将会　　对于uv_async_send()　数的　用，　意　　不是　　用　对　一个回　　数的　用。例如：如果　数在回　　数被　用　　用了　，　　回　　数将只会　用一　。如果uv_async_send()　数在回　　数再　被　用，　　回　　数也会再　被　用

参考：uv_handle_t的API　　用

uv_poll_t——poll handle

poll handle的作用 inux的poll(2) 不 , 要是为了 文件描述符的可 可 以及断开 。

poll handle的 的是 于 件 的 (如c ares或者libssh2)可以 socket 化的 。 于 的 用uv_poll_t 是不 的 uv_tcp_t uv_udp_t等提供 uv_poll_t更 更好的实 , 是在windows平台 。

poll handle对于文件描述符的可 可 偶 可 会 错误的 。 此在 文件描述符(fd)时用 是 好 EAGAIN或类 的 。

对 一个socket 用 个 的poll handle是不可行的, 会 libuv (busyloop)或 的 。

当一个文件描述符 在被poll handle 时, 用 不 。 为 会 handle 错, 时也有可 开始 一个socket。在 用uv_poll_stop()或者uv_close() 文件描述符可以 的

注意: 在windows平台只有socket可以被poll handle , 在unix平台, 被poll(2) 支持的文件描述符 用。

数据类型

uv_poll_t
Poll handle结 类型

void (*uv_poll_cb)(uv_poll_t* handle)
传 uv_poll_start() 数的回 数的 数指 类型

uv_poll_event
poll 件类型

公有成员

无

参考: uv_handle_t的成员

API

int uv_poll_init(uv_loop_t* loop, uv_poll_t* handle, int fd)
通 文件描述符初始化handle
1.2.2版本 : 文件描述符被设置为非阻塞模式

int uv_poll_init_socket(uv_loop_t* loop, uv_poll_t* handle, uv_os_sock_t socket)
用socket描述符初始化poll handle。在unix平台等效于 uv_poll_init(), 在windows平台, 本 数支持socket句柄
1.2.2版本 : socket被设置为非阻塞模式

int uv_poll_start(uv_poll_t* handle, int event, uv_poll_cb cb)
开始 文件描述符。event是 UV_ EA AB E UV_ I AB E和UV_ IS ONNE 成的 码。一 一个时 被检到, 回 数会被 用 status被设置为0, 检 到的 件类型会被 值 events成员 。

UV_ IS ONNE 件是可 的, 可 不会被 , 用 可以 , 不 个 件有 于 路径(shutdown path), 为可以 的 或 作。

如果在 时 错误, statue将为小于0, 对 一个UV_E*的错误码。在handle 时, 用 不 socket, 否 回数会被 用, 是错误的 值, 但是 不 一定如此。

注意: 在一个已经开始的handle 用uv_poll_start是可行的。 可以跟新 件的 码。

注意: UV_ IS ONNE 是可以设置的, 但是 不 在AI 中 用, 此回 数中 不会将events 设置为个。

在1.9.0中 了UV_ IS ONNE 件

uv_poll_stop(uv_poll_t* handle)
停止 文件描述符, 回 数也不会再被 用。

参考: uv_handle_t的API 可用。

uv_signal_t——　　handle

　　句柄实　了unix类型的　　，用来　　个　件　　的　　。

在windows平台模　了一　　的接
　　当用　　下ctrl　c时会　　。　　在unix　一　，当终　的　始模式　　时　不会如此。
　　当用　　下ctrl　break时　　中断
　　当用　　台　　时会　　SIGNUP：在SIGNUP，(　　)会　　程　大　10s的时　去进行　　。　　windows会无件的终止程　。
　　当libuv检　到　　台程　　时，会　　SIG　IN　　。当程　　用uv_tty_t handle　　台　　时，libuv会模　　SIG　IN　　。SIG　IN　　不　是及时的　　，libuv只是在　　时检　大小　　。当一个可　的uv_tty_t被用在　始模式（raw mode）下时，　　台缓冲区也将　　sigwinch　　。

对　　型　的　　也可以成　的　　，但是　　无　　到。　　是：
SIG　，SIGAB　，SIGFPE，SIGSEGV，SIG　E　，SIG　I　。

通　　程　用raise()以及abort()　数的　式　　的　　不会被libuv检　到，　　也不会　　者。

注意：在linux平台下，SIG　0以及SIG　1(　　32和33)被NP　pthreads　　用来管　　程。对　　者将　　不可预知的行为，　　不　　如此。libuv的　来版本可　会　　。

数据类型

uv_signal_t
　　结　　类型

void (*uv_singnal_cb)(uv_signal_t*　handle)
　　传　　uv_signal_start()　数的回　　数的　数指　类型

公有成员

int　uv_signal_t.signum
　　被　个handle　　的　　，只

参考：uv_handle_t的成员

API

int　**uv_signal_init**(uv_loop_t* loop,　uv_signal_t*　handle)
　　初始化handle

int　**uv_signal_start**(uv_signal_t*　handle,　uv_signal_cb　cb,　int　signum)
　　开始handle，　传　回　　数，开始　　定的

int　**uv_signal_stop**(uv_signal_t*　handle)
　　停止handle，对　的回　　数将不会再被　用

参考：**uv_handle_t**的API　　用

uv_process_t——进程handle

进程handle将会 成一个新的进程， 用 通 (streams) 道。

数据类型

uv_process_t
进程handle结 类型

uv_process_options_t
成新进程的配置(传 uv_spawn() 数)
typedef struct uv_process_options_s
uv_exit_cb exit_cb
const char* file
char** args
char** env
const char* cwd
unsigned int flags
int stdio_count
uv_stdio_container_t* stdio
uv_uid_t uid
uv_gid_t gid
uv_process_options_t

void (*uv_exit_cb)(uv_process_t* handle, int _t exit_status, int term_signal)
传 uv_process_options_t 数的回 数的 数指 定 。uv_process_options_t 数将会指 进程 的 以及

uv_process_flags
传 uv_process_options_t 成员的 类型
enum uv_process_flags

Set the child process user id. 设置子进程的用 I
UV_P O ESS_SE UI (1 0),

Set the child process group id. 设置子进程的 I
UV_P O ESS_SE GI (1 1),
*
* 当将参数 表 为命 行字符 时，不要对 参数 用 或perform any other escaping。
* 个配置只在windows 下 作用，在unix下将被 。
*
UV_P O ESS_ IN O S_VE BA I _A GU EN S (1 2),
*
* 在分 (detached state) 下 成子进程—— 将 子进程成为一个进程 的 始进程， 会 子进程在 进程
* 也 持 行。注意：子进程会 持 进程 件 alive， 非 进程对子进程handle 用uv_unref()
*
UV_P O ESS_ E A E (1 3),
*
* 子进程的 台 ，只在windows平台下有效。
*
UV_P O ESS_ IN O S_ I E (1)

uv_stdio_container_t
传 子进程的stdio句柄或文件描述符的 器
typedef struct uv_stdio_container_s
uv_stdio_flags flags
union
uv_stream_t* stream
int fd
data
uv_stdio_container_t

uv_stdio_flags
指定stdio将如 被传 子进程的
typedef enum
UV_IGNO E 0x00,
UV_ EA E_PIPE 0x01,
UV_IN E I _F 0x02,
UV_IN E I _S EA 0x0 ,
*
* 当 UV_ EA E_PIPE被 定, UV_ EA AB E_PIPE 和 UV_ I AB E_PIPE
*以子进程的 度 定 的 ( )。 一种 会指定要 一个 工数据 B
*
UV_ EA AB E_PIPE 0x10,
UV_ I AB E_PIPE 0x20
uv_stdio_flags

公有成员

uv_process_t.pid
成的新进程的进程I 。在uv_spawn() 数 用 设置

注意：uv_handle_t成员也可用

uv_process_options_t.exit_cb
进程 的回 数

uv_process_options_t.file
需要 行的进程的路径

uv_process_options_t.args
命 行参数。args 0 是进程路径。在windows平台下 用 reateProcess， 个 数会将参数连成一个字符 ， 回 一 的错误，参考uv_process_flags中的UV_P O ESS_ IN O S_VE BA I _A GU EN S

uv_process_options_t.env
新进程的 。如果为 ，将 用 进程的

uv_process_options_t.cwd
子进程的当 工作

uv_process_options_t.flags
定 uv_sqawn() 数行为的 种 ，参考uv_process_flags

uv_process_options_t.stdio_count
uv_process_options_t.stdio
stdio类型的指 ，指 一个uv_stdio_container_t结 的数 ， 是字进程可用的文件描述符。 例是stdio 0 是stdin( )，fd 1是stdout( )，fd 2是stderr

注意：在windows平台文件描述符数 只有在子进程 用msvcrt 行时(runtime) 大于等于2

uv_process_options_t.uid
uv_process_options_t.gid
libuv可以 子进程的用 id以及 id。只有当 字段设置了 的 可行
windows平台不可用。uv_sqawn() 数将会 返回UV_ENOSUP错误

uv_stdio_container_t.flag
定stdio 器如 被传到子进程。参考uv_stdio_flags

uv_stdio_container_t.data
包含传 子进程的 (stream)或者文件描述符的 。

API

void uv_disable_stdio_inheritance(void)
子进程 进程的文件描述符无 再被 。效果是本进程 成的子进程无 意 的 句柄 在文件描述符被 或 可 的 用本 数。
注意：本 数 大可 的 行： 不 libuv 够 有的 来的文件描述符。一 来 本 数在windows下效果更好。

int uv_spawn(uv_loop_t* loop, uv_process_t* handle, uv_process_options_t* options)
初始化进程handle 进程。如果进程被成 成，将会返回0，否 返回错误代码。
可 的 包 但不 于：可 行文件不存在； 有 指定setuid或setgid； 存不 。

int uv_process_kill(uv_process_t* handle, int signum)
传 的 指定的进程句柄。参考uv_signal_t， 是在windows平台下

int uv_kill(int pid, int signum)
传 的 指定的PI (进程I)。

参考：uv_handle_t的API 用。

uv_stream_t—— handle

handle提供一个抽象的 工通 通道。uv_stream_t是抽象类型，libuv提供3种 实 ：uv_tcp_t,uv_pipe_t以及uv_tty_t

数据类型

uv_stream_t
  handle类型

uv_connect_t
  链接 求类型

uv_shutdown_t
  求类型

uv_write_t
  求类型

void (*uv_read_cb)(uv_stream_t* handle, size_t nread, const uv_buf_t* buf)
  当 的数据被 取时的回 数
  当数据可用时，nread 0，错误 下nread 0。当 到 尾，nread被设为UV_EOF。当nread 0，buf参数将不会指 可用的缓存；在 种 下，buf.base以及buf.len 被设为0
  注意：nread可 是0，但 不表 错或 到 尾， 当于在read(2) 下的EAGAIN 或者 E OU B O

  当 于 用uv_read_stop()或者uv_close()停止 错误时， 被 用者 。 取数据是定 的行为。
  被 用者 buffer，libuv不会再 用 。buffer可 是 的或者错误的

void (*uv_write_cb)(uv_write_t* req, int status)
  数据被 的回 数。status为0表 成 ， 0表

void (*uv_connect_cb)(uv_connect_t* req, int status)
  当uv_connect() 数开始的一个链接 成时回 。status为0表 成 ， 0表

void (*uv_shutdown_cb)(uv_shutdown_t* req, int status)
  当一个 求被 会被回 。status为0表 成 ， 0表

void (*uv_connection_cb)(uv_stream_t* server, int status)
  当一个 服务 到一个链接 求时被回 。用 可以通 uv_accept() 数接 求。status为0表 成 ， 0表

公有成员

size_t uv_stream_t.write_queue_size
  包含等待 的 字节的数 。只

uv_stream_t* uv_connect_t.handle
  本链接 求 对的 的指

uv_stream_t* uv_shutdown_t.handle
  本 求 对的 的指

uv_stream_t* uv_write_t.handle
  本 求 对的 的指

uv_stream_t* uv_write_t.send_handle
  指 用本 求的 将要 去的

参考：uv_handle_t的成员

API

int **uv_shutdown**(uv_shutdown_t* req, uv_stream_t* handle, uv_shutdown_cb cb)
的 。 将等待 在等待的 求 成。handle必须指 已经初始化的 。req必须是 有初始化的 求结 。回 数将会在 求 成 回 。

int **uv_listen**(uv_stream_t* stream, int backlog, uv_connection_cb cb)
开始 将到来的链接，backlog表 可 的链接 , listen(2)。当一个新的链接被 到，回数将会被 用

int **uv_accept**(uv_stream_t* server, uv_strea_t* client)
此 数 uv_listen() 数结 用来接 链接。在uv_connection_cb回 数中 用 个 数来接 链接。在用本 数 ，client必须被初始化。返回值 0代表 错。
当uv_conneton_cb被回 时，本 数在 一 行时将 成 。如果 用， 可 会 。 uv_connection_cb回 中只 用一 。
注意：server 于client 必须在 一个loop

int **uv_read_start**(uv_stream_t* stream, uv_alloc_cb alloc_cb, uv_read_cb read_cb)
进来的链接中 取数据。uv_read_cb可 会被回 到 有数据去 取，或者uv_read_stop() 数被用

int **uv_read_stop**(uv_stream_t* stream)
停止 取。uv_read_cb回 数将不会再被 用
个 数是 等的，在已经停止的stream 用是 的

int **uv_write**(uv_write_t* req, uv_stream_t* handle, const uv_buf bufs , unsigned int bufs, uv_write_cb cb)
将数据 到 。buffers 。 如：

```
void cb(uv_write_t* req, int status)
        *  ogic which handles the write result *


    uv_buf_t a
            .base   1 , .len   1  ,
            .base   2 , .len   1

    uv_buf_t b
            .base   3 , .len   1  ,
            .base     , .len   1

    uv_write_t req1
    uv_write_t req2

     * writes  123        123   *
    uv_write(  req1, stream, a, 2, cb)
    uv_write(  req2, stream, b, 2, cb)
```

int **uv_write2**(uv_write_t* req, uv_steam_t* handle, const uv_buf_t bufs , unsigned int nbufs, uv_stream_t* send_handle, uv_write_cb cb)
的 数，用来通 管道 。管道必须 用ipc 1 初始化
注意：send_handle必须是tcp socket或者管道，可以是服务 或者一个链接。绑定sockets或者管道将被 定为服务 。

int uv_try_write(uv_stream_t* handle, const uv_buf_t bufs , unsigned int nbufs)
uv_write() 数 。但是如果不 的 不会 求 。
返回值 0： 的字节，可 预 的
0，错误代码（如果数据不 ，返回UV_EAGAIN）

int **uv_is_readable**(const uv_stream_t* handle)
如果可 ，返回1，否 返回0

int **uv_is_writeable**(const uv_stream_t* handle)
如果可 ，返回1，否 返回0

int **uv_stream_set_blocking**(uv_stream_t* handle, int blocking)
用或者 的阻塞模式
如果 用， 有的的 作将会 成。 he interface remains unchanged otherwise。例如： 作的 成或者 通 一个 回 。
：不 度 个API， 可 会在 来版本被 。 在windows 对uv_pipe_t 作用，在UNI 对 有handle 作用。
libuv也不 在 求提 阻塞模式一定会 作用。 此 在打开或 设置阻塞模式。

uv_tcp_t——tcp handle

tcp handle可以代表tcp 服务

uv_tcp_t是uv_stream_t的一个 子类

数据类型

uv_tcp_t
tcp handle 类型

公有成员

无
参考：uv_stream_t

API

int **uv_tcp_init**(uv_loop_t* loop, uv_tcp_t* handle)
初始化tcp handle。 有 socket

int uv_tcp_init_ex(uv_loop_t* loop, uv_tcp_t* handle, unsigned int flags)
用 定的 初始化handle。 只有 8 被 用。将会 据此参数 socket。如果 是AF_UNSPE , 不会 socket，作用 和uv_tcp_init()一
1. .0版本新增

int **uv_tcp_open**(uv_tcp_t* handle, uv_os_sock_t sock)
打开一个已存在的文件描述符或者socket作为tcp handle
在1.2.1版本中的 ：文件描述符被设置为非阻塞模式
注意：指定的文件描述符或者socket 不会进行类型检查，但是需要指 有效的stream socket

int **uv_tcp_nodelay**(uv_tcp_t* handle, int enable)
用 不 用 Nagle s

int **uv_tcp_keepalive**(uv_tcp_t* handle, int enable, unsigned int delay)
用 不 用 P的keep alive。delay是初始 ，如果enable为0 本参数

int **uv_tcp_simultaneous_accepts**(uv_tcp_t* handle, int enable)
用 不 用 新的tcp链接时有 作 的 时 的 链接
个设置被用来 P服务来 到 的 。 时接 可以提 接 链接的 度(用的)，但是会 进程设置中的不平 分配。

int **uv_tcp_bind**(uv_tcp_t* handle, const struct sockaddr* addr, unsigned int flags)
将handle绑定到一个地 以及 。addr需要指 已经初始化的sockadd_in或者sockaddr_in 结
当 已经 用时，可以 uv_tcp_bind(), uv_listen() 或者uv_tcp_connect() 数获取UV_EA INUSE错误。也 是 ，本数的成 用 不 uv_listen()或者uv_tcp_connect() 数的成 。
flags可以是UV_ P_IPV ON , 只支持IPV 。

int **uv_tcp_getsockname**(const uv_tcp_t* handle, struct sockaddr* name, int namelen)
获取绑定的handle的地 。addr必须指 可用的 够大的 存 。 用sockaddr_stroage结 , 时支持IPv 以及IPv 。

int **uv_tcp_getpeername**(const uv_tcp_t* handle, struct sockaddr*name, int namelen)
获取链接到本handle的socket的地 。addr必须指 可用的 够大的 存 。 用sockaddr_stroage结 , 时支持IPv 以及IPv 。

int **uv_tcp_connect**(uv_connect_t* req, uv_tcp_t* handle, const struct sockaddr* addr, uv_connect_cb cb)
一个IPv 或者IPv 的 P链接，提供一个初始化的tcp handle以及一个 初始化的uv_connect_t。addr需要指 一个初始化的socket_in或者sockaddr_in 结 。
当链接 了或者 了错误的时 将会 用回 数。

参考：uv_stream_t的API 用

uv_pipe_t——管道handle

管道handle提供了对于unix下本地socket以及windows平台下管道的抽象

uv_pipe_t是uv_stream_t的一个“子类”

数据类型

uv_pipe_t
管道handle类型

公有成员

无
参考：uv_stream_t的成员

API

int **uv_pipe_init**(uv_loop_t* loop, uv_pipe_t* handle, int ipc)
初始化管道handle。ipc参数是bool值，指明是否支持跨进程。

int **uv_pipe_open**(uv_pipe_t* handle, uv_file file)
打开一个已经存在的文件描述符作为pipe；
1.2.1更新：文件描述符设置为非阻塞模式。
注意：传进来的文件描述符或者handle不会进行类型检查，但是要求是可用的pipe

int **uv_pipe_bind**(uv_pipe_t* handle, const char* name)
将管道绑定到文件路径(unix平台)或者名字(windows平台)
注意：unix平台下的路径会被截断到（sockaddr_un.sun_path）长度，通常介于92和108字节

void **uv_pipe_connect**(uv_connect_t* req, uv_pipe_t* handle, const char* name, uv_connect_cb cb)
链接到unix下的本地socket或者windows下的命名管道
注意：unix平台下的路径会被截断到（sockaddr_un.sun_path）长度，通常介于92和108字节

int **uv_pipe_getsockname**(const uv_pipe_t* handle, char* buffer, size_t* size)
获取unix下本地socket或者windows下命名管道的名字。
必须提供预先分配好的缓冲区。size参数代表缓冲区的大小，it’s set to the number of bytes written to the buffer on output。如果缓冲区不大，会返回UV_ENOBUFS错误码，size？会设置为需要的大小。
1.3.0更新：返回的长度不再包含终止的null字节，缓冲区也不以null结尾。

int **uv_pipe_getpeername**(const uv_pipe_t* handle, char* buffer, szie_t* size)
Get the name of the Unix domain socket or the named pipe to which the handle is connected.
必须提供预先分配好的缓冲区。size参数代表缓冲区的大小，it’s set to the number of bytes written to the buffer on output。如果缓冲区不大，会返回UV_ENOBUFS错误码，size？会设置为需要的大小。
在1.3.0版本中新增

void **uv_pipe_pending_instance**(uv_pipe_t* handle, int count)
设置当管道服务器等待连接时等待的管道实例句柄数。
注意：只在windows平台下有效

uv_handle_type **uv_pipe_pending_type**(uv_pipe_t* handle)
IP 管道中接 链接handle
先 用uv_pipe_pending_count(),如果大于0， 初始化一个 uv_pipe_pending_type()返回的handle类型，接用uv_accept(pipe， handle)

参考：uv_stream_t的API 用

uv_tty_t——TTY handle

tty handle是一个控制台的流。
uv_tty_t是uv_stream_t的一个”子类“

数据类型

uv_tty_t
    tty handle类型

uv_tty_mode_t
    1.2.0版本新增
    TTY模式类型：
    typedef enum {
        /* Initial/normal terminal mode */ 初始/默认的终端模式
        UV_TTY_MODE_NORMAL,
        /* Raw input mode (On Windows, ENABLE_WINDOW_INPUT is also enabled) */
        //原始输入模式(在windows平台，ENABLE_WINDOW_INPUT也会启用)
        UV_TTY_MODE_RAW,
        /* Binary-safe I/O mode for IPC (Unix-only) */
        UV_TTY_MODE_IO
    } uv_tty_mode_t;

公有数据

无
参考：uv_stream_t类型

API

int **uv_tty_init**(uv_loop_t* loop, uv_tty_t* handle, uv_file fd, int readable)
    通过给定的文件描述符初始化一个新的TTY handle。通常会是：
        · 0 = stdin
        · 1 = stdout
        · 2 = stderr
    readable参数表明你是否想要使用uv_read_start()函数。stdin是可读的，stdout不是

    在unix平台下，如果传入的描述符指向一个TTY，这个函数会使用ttyname_r(3)函数确定文件路径，打开它，并使用。这让libuv可以在不影响其他共享了这个tty的进程的情况下将tty设为非阻塞模式。
    这个函数在不支持TIOCGPTN以及TIOCPTYGNAME的系统下不是线程安全的，比如OpenBSD以及SOlaris系统。
    注意：如果重新打开TTY失败，libuv falls back to blocking writes for non-readable TTY streams.
    1.9.2：TTY的路径由ttyname_r(3)函数确定。在早期版本，libuv打开dev/tty
    1.5.0：在UNIX平台尝试通过指向文件的描述符初始化TTY stream将会返回UV_EINVAL

int **uv_tty_set_mode**(uv_tty_t* handle, uv_tty_mode_t mode)
    1.2.0：mode参数为一个uv_tty_type_t类型值
    使用制定的终端类型设置TTY

int **uv_tty_reset_mode**(void)
    程序退出时被调用。将tty还原为默认设置以便下一个程序使用
    本函数在unix平台下是异步信号安全的，但是当你在uv_tty_set_mode()执行中调用时会导致失败，并返回错误码UV_EBUSY。

int **uv_tty_get_winsize**(ut_tty_t* handle, int* width, int* height)
    获取当前窗口的尺寸。成功返回0

参考：uv_stream_t的API同样适用

uv_udp_t——udp handle

udp handle封装了UDP链接的客户端以及服务端

数据类型

uv_udp_t
    udp handle类型

uv_udp_send_t
    udp发送请求类型

uv_udp_flags
    在uv_udp_bind()以及uv_udp_recv_cb中使用的类型
    enum uv_udp_flags {
        /* Disables dual stack mode. */ 双协议栈不可用(只可用IPv6)
        UV_UDP_IPV6ONLY = 1,
        /*
        * 由于读缓冲区太小导致数据被截断. 剩下的数据被系统丢弃，在uv_udp_recv_cb中使用
        */
        UV_UDP_PARTIAL = 2,
        /*
        * 表明在uv_udp_bind时SO_REUSEADDR 标记是否被指定
        * 在BSDs以及OS X系统下设置SO_REUSEPORT socket 标记 . 在其他
        * Unix 系统设置SO_REUSEADDR 标记.。这表示在多个线程或进程中
        * 绑定到相同的地址上不会报错(假如都设置了这个标记)但是只有最后
        *一个绑定的可以收到数据，就好像将端口从之前的绑定偷了过来
        * any traffic, in effect "stealing" the port from the previous listener.
        */
        UV_UDP_REUSEADDR = 4
    };

void (*uv_udp_send_cb)(uv_udp_send_t* req, int status)
    传递给uv_udp_send()的回调函数的函数指针类型。将会在数据发送之后被调用

void (*uv_udp_recv_cb)(uv_udp_t* handle, size_t nread, const uv_buf_t* buf, const struct sockaddr* addr, unsigned flags)
    传递给uv_udp_recv_start()函数的回调函数的函数指针类型，将会在收到数据的时候被调用
        · handle：UDP handle
        · nread：收到数据的大小。如果没有数据了，为0。你可以丢弃或者将缓冲区用于他用。注意，0也可能表示收到了长度为0的数据（这种情况下addr不为空），<0表示检测到发生了错误。
        · buf：指向接受到的数据的uv_buf_t
        · addr：包含发送方地址的sockaddr 结构体的指针。可以为空，有效期仅为回调执行期间。
        · flag：一个或者多个UV_TCP_*常数。目前只有UV_UDP_PARTIAL 可用
    注意：回调函数只可能会在nread==0 addr==null的情况下(不再有数据)调用,或者nread==0 addr!=0的情况下(收到了空包)调用

uv_membership
    一个多播地址的成员类型
    typedef enum {
        UV_LEAVE_GROUP = 0,
        UV_JOIN_GROUP
    } uv_membership;

公有成员

size_t uv_udp_t.send_queue_size
    在队列中等待发送的数据长度。严格的显示有多少数据在队列中

size_t uv_udp_t.send_queue_count
    队列中有多少发送请求等待处理

uv_udp_t* uv_udp_send_t.handle
    发送请求基于的udp handle指针

参考：uv_handle_t的成员

API

int uv_udp_init(uv_loop_t* loop, uv_udp_t* handle)
初始化一个新的UDP handle，真实的socket将会延时创建。如果成功返回0

int uv_udp_init_ex(uv_loop_t* loop, uv_udp_t* handle, unsigned int flags)
通过特殊额标志初始化udp handle，目前只用到flags的低8位。通过给定的domain将会创建socket

version.c

提供 个 数获取当 libuv 的版本:

1.uv_version,获取数字版本
2.uv_version_string获取字符 版本

tree.h

定 (splay tree) 和 (B tree)

1. 在windows平台 用

2. 定
a. define B_ EA (name, type) 结 , 包含一个 节 指 (如uvwin.h中的uv_timer_s, 中uv_timer_s 中 包含UV_ I E _P IVA E_FIE S , 开 有 B_EN (uv_timer_s) tree_entry)

```
define B_ EA (name, type)
struct name
        struct type *rbh_root * root of the tree *
```

b. define B_INI (root) 初始化 节

```
define B_INI (root) do
        (root) rbh_root NU
while ( * ONS ON * 0)
```

c. define B_EN (type) 节 , 包含 子 , 节 以及节

d. 数的 定 , 要包 一个节 下一个节 大节 小节 查 等 , 可以参考stl的 (stl通 模 实 支持 节 包含不 类型的数据, 用 一 的 , libuv通 定 来实 , 不 类型的 节 包含 B_EN , 的 作 本 是 对 定 的结)

uv win.h

定 平台 的
1. 定 模型 (Overlapped I O) 用到的socket 数指
a. typedef int (SAAPI* PFN_ SA E V)
b. typedef int (SAAPI* PFN_ SA E VF O)

2. typedef struct uv_buf_t结 , 够 为 SABUF

3. B_ EA (uv_timer_tree_s, uv_timer_s) 定 一 uv_timer_tree_s, 节 为uv_timer_s

```
. define UV_  OOP_P  IVA  E_FIE    S 定   loop(      )的   有
                * he loop s I O completion port *
                 iocp   成      句柄
                 AN    E iocp
                * he current time according to the event loop. in msecs. *
                    件      的当   时
               uint   _t time
                * ail of a single linked circular queue of pending reqs. If the queue *
                * is empty, tail_ is NU    . If there is only one item, *
                * tail_   next_req      tail_ *
                                的尾指
               uv_req_t* pending_reqs_tail
                *   ead of a single linked list of closed handles *
                 已经      的句柄的         表的   指
               uv_handle_t* endgame_handles
                 * he head of the timers tree *
                 定时器
               struct uv_timer_tree_s timers
                  * ists of active loop (prepare   check   idle) watchers *
                      的            者   表
                uv_prepare_t* prepare_handles
               uv_check_t* check_handles
               uv_idle_t* idle_handles
                * his pointer will refer to the prepare check idle handle whose *
                * callback is scheduled to be called next.   his is needed to allow *
                * safe removal from one of the lists above while that list being *
                * iterated over. *

               uv_prepare_t* next_prepare_handle
               uv_check_t* next_check_handle
               uv_idle_t* next_idle_handle
                * his handle holds the peer sockets for the fast variant of uv_poll_t *
               SO    E   poll_peer_sockets UV_   SAF   _P  OVI   E   _   OUN
                *   ounter to keep track of active tcp streams *
                      的tcp   数
               unsigned int active_tcp_streams
                *   ounter to keep track of active udp streams *
                      的udp   数
               unsigned int active_udp_streams
                *   ounter to started timer *
                 已开始的定时器的数
               uint   _t timer_counter
                 *   hreadpool *
                void* wq 2
               uv_mutex_t wq_mutex
               uv_async_t wq_async
```

. define UV_ E _ PE_P IVA E 定 E (求)的类型

. define UV_ E _P IVA E_FIE S 定 E 的 有

```
define UV_ E _P IVA E_FIE S
union
        * Used by I O operations *
      struct
            OVE APPE overlapped
            size_t queued_bytes
        io
 u
struct uv_req_s* next_req
```

. define UV_ I E_P IVA E_FIE S 定 write的 有

```
int ipc_header
uv_buf_t write_buffer
  AN E event_handle
  AN E wait_handle
```

8. uv_pipe_accept_s
9. uv_tcp_accept_s
10. uv_read_s
11. uv_stream_connection_fields
12. uv_stream_server_fields
13. UV_S EA _P IVA E_FIE S
1 . uv_tcp_server_fields
1 . uv_tcp_connection_fields
1 . UV_ P_P IVA E_FIE S
1 . uv_pipe_server_fields
18. uv_pipe_connection_fields
19. UV_PIPE_P IVA E_FIE S
20. UV_ _P IVA E_FIE S
21. UV_PO _P IVA E_FIE S
22. UV_ I E _P IVA E_FIE S
23. UV_AS N _P IVA E_FIE S
2 . UV_P EPA E_P IVA E_FIE S
2 . UV_ E _P IVA E_FIE S
2 . UV_I E_P IVA E_FIE S
2 . UV_ AN E_P IVA E_FIE S
28. UV_GE A INFO_P IVA E_FIE S
29. UV_GE NA EINFO_P IVA E_FIE S
30. UV_P O ESS_P IVA E_FIE S
31. UV_FS_P IVA E_FIE S
32. UV_ O _P IVA E_FIE S
33. UV_FS_EVEN _P IVA E_FIE S
3 . UV_SIGNA _P IVA E_FIE S

uv.h

要是一 定 以及 数的 明

1._ IN32平台下，如果定 了BUI ING_UV_S A E 是 数（libuv工程在工程配置 c c ——预 器 中定 ）：

```
if defined(BUI ING_UV_S A E )
* Building shared library. *
define UV_E E N __declspec(dllexport)
```

如果定 了USING_UV_S A E （ 用者定 ）， 是 数

2. define UV_E NO_ AP( ) 定 了错误名以及对 的错误 的 表， 中 可以是 一个 ， 如在uv common.h中的：

```
define UV_E _NA E_GEN(name, _) case UV_ name return name
```

配 switch 句 用，返回错误名。 中错误的代码通 定 ：

```
typedef enum
define (code, _) UV_ code UV__ code,
UV_E NO_ AP( )
undef
UV_E NO_ A UV__EOF 1
uv_errno_t
```

3. define UV_ AN E_ PE_ AP( )，句柄类型 表， 时参考：

```
typedef enum
UV_UN NO N_ AN E 0,
define (uc, lc) UV_ uc,
UV_ AN E_ PE_ AP( )
undef
UV_FI E,
UV_ AN E_ PE_ A
uv_handle_type
```

. define UV_ E _ PE_ AP( )，定 求类型 表， 时参考：

```
typedef enum
UV_UN NO N_ E 0,
define (uc, lc) UV_ uc,
UV_ E _ PE_ AP( )
undef
UV_ E _ PE_P IVA E
UV_ E _ PE_ A
uv_req_type
```

.定 一 结 类型， 如typedef struct uv_loop_s uv_loop_t ， 要有句柄结 以及 求结

.定 一 配置：

```
typedef enum
UV_ OOP_B O _SIGNA
uv_loop_option

typedef enum
UV_ UN_ EFAU 0,
UV_ UN_ON E,
UV_ UN_NO AI
uv_run_mode
```

. 数 明，以及 明 种 数指

8. 定 种结