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：

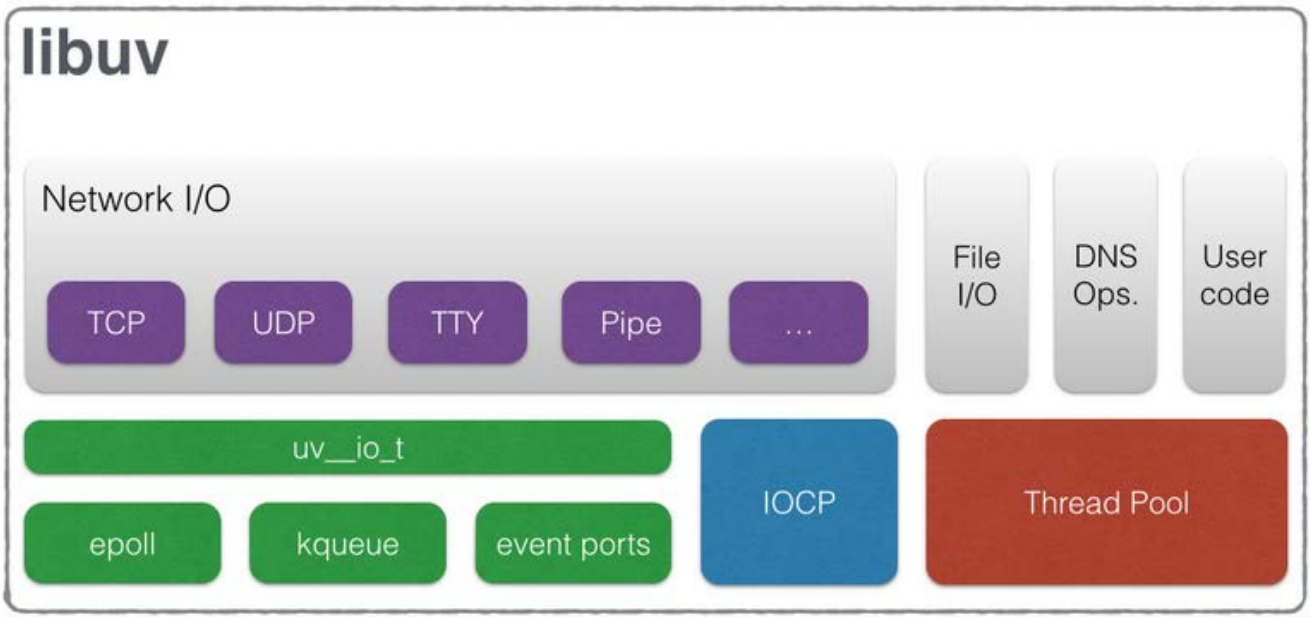
- 设计概述
- 错误处理
- 版本检测宏与函数
- uv_loop_t——事件循环
- uv_handle_t——基础句柄
- uv_req_t——基础请求
- uv_timer_t——定时器句柄
- uv_prepare_t——预处理handle
- uv_check_t——check handle
- uv_idle_t——空转handle
- uv_async_t——异步handle
- uv_poll_t——poll handle
- uv_signal_t——信号量handle
- uv_process_t——进程handle
- uv_stream_t——流handle
- uv_tcp_t——tcp 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。

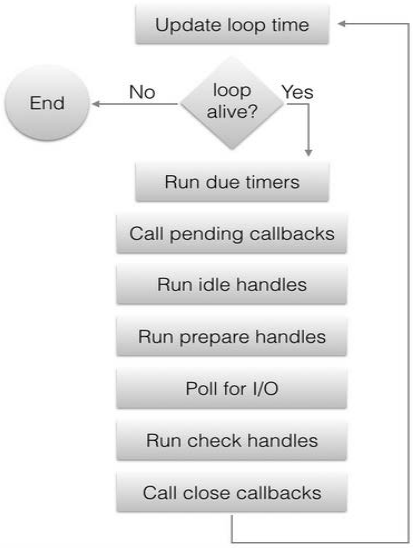
请求(Requests)通常是代表操作的短寿命对象。这些操作能够通过一个句柄(handle)进行——写请求被用来在一个handle上写入数据；也可以是独立的——getaddrinfo请求不需要handle，直接运行在循环上。

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所有请求类型的基础类型

一致的结构使得libuv的所有求类型都可以转换为uv_req_t。下面的API函数适用于所有的请求类型。

数据类型

uv_req_t

libuv的基础请求结构体类型

uv_any_req

所有请求类型的联合体

公有成员

void* uv_req_t.data

指向用户自定义数据，libuv不使用该成员

uv_req_type uv_req_t.type

请求的类型，只读

```
typedef enum {
        UV_UNKNOWN_REQ = 0,
        UV_REQ,
        UV_CONNECT
        UV_WRITE,
        UV_SHUTDOQN,
        UV_UDP_SEND,
        UV_FS,
        UV_WORK,
        UV_GETADDRINFO,
        UV_GETNAMEINFO,
        UV_REQ_TYPE_PRIVATE,
        UV_REQ_TYPE_MAX,
} uv_req_type;
```

API

int **uv_cancel**(uv_req_t* req)

取消一个等待处理的请求。如果请求正在被处理或者完成了处理，函数执行失败

成功返回0，失败返回错误代码

目前只支持uv_fs_t,uv_getaddrinfo_t, uv_getnameinfo_t 以及 uv_work_t句柄请求的取消

被取消的请求的回调函数将会在之后的一段时间被调用。在回调函数被调用之前，释放请求的内存是不安全的。

下面显示取消的请求是如何被传到回调函数的

- uv_fs_t 请求的req->result 成员被设置为UV_ECANCEL
- uv_work_t, uv_getaddrinfo_t 或者 c:type:uv_getnameinfo_t 请求的回调函数将会被调用，同时status参数设置为UV_ECANCELED.

size_t uv_req_size(uv_req_type type)

返回对应的请求类型的大小

uv_time_t——定时器句柄(Timer handle)

定时器handle用来在一段时间之后调用预定的回调函数

数据类型

uv_timer_t
　　定时器handle类型

void (*uv_timer_cb)(uv_timer_t* handle)
　　传递给uv_timer_start()函数的回调函数的函数指针定义

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, uint64_t timeout, uint64_t repeat)
　　开启定时器。超时以及重复都是毫秒
　　如果超时是0，回调函数将在下一次循环迭代处罚。如果repeat不是0，回调函数将会在timeout毫秒之后被调用，接着每隔repeat毫秒再次被调用。

int **uv_timer_stop**(uv_timer_t* handle)
　　停止定时器。回调函数将不会再被调用

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。如果定时器没有启动，那么返回UV_EINVAL

int uv_timer_set_repeat(uv_timer_t* handle, uint64_t repeat)
　　设置一个值作为重复间隔，单位为毫秒。定时器将会按照给定的间隔运行，不管回调的执行所需的时间，并且在时间片溢出的情况下遵循正常的定时器语义。
　　例如，如果50毫秒间隔的定时器运行了17毫秒，那么它将预计再运行33毫秒。如果其他的任务消费的超过33毫秒，那么定时器将会尽可能早的执行
　　注意：如果重复时间(repeat)是在定时器回调函数中设置的，那么不会立刻起作用。如果定时器之前没有repeat，那么将会停止。如果正在重复执行中，旧的重复时间将会被用来执行下一次。

uint64_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——预处理handle（Prepare handle）

预处理handle将会在每次循环迭代轮询i/o之前调用一次

数据类型

uv_prepare_t

预处理handle结构体类型

void (*uv_prepare_cb)(uv_prepare_t* handle)

传递给uv_prepare_start()函数的回调函数的函数指针类型

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,并传入回调函数

int **uv_prepare_stop**(uv_prepare_t* handle)

停止handle，并且回调函数将不会再被调用

参考：**uv_handle_t**的API同样可用

uv_check_t——Check handle

Check handle将会在每次循环迭代轮询i/o之后调用回调函数

数据类型

uv_check_t
check handle结构体类型

void (*uv_check_cb)(uv_check_t* handle)
传递给uv_check_start()函数的回调函数的函数指针定义

公有成员

无

参考：**uv_handle_t**结构体

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，并传入回调函数指针

int **uv_check_stop**(uv_check_t* handle)
停止handle，回调函数将不会再被调用

参考：**uv_handle_t**的API同样适用

uv_idle_t——空转handle

空转handle将会在每次循环迭代的uv_prepare_t handle之前调用回调函数

注意：空转handle与预处理(prepare)handle的区别是，当有活动的空转handle时，循环将会采取零超时轮询，而不是阻塞的i/o轮询

警告：尽管名字是空转(idle),但并不是说当循环是空转时才调用回调函数，而是在每一次循环迭代的时候都会调用。

数据类型

uv_idle_t
空转handle结构体类型

void (*uv_idle_cb)(uv_idle_t* handle)
传递给uv_idle_start()函数的回调函数的函数指针定义

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)
开始空转handle，并传入回调函数指针

int **uv_idle_stop**(uv_idle_t* handle)
停止空转handle，回调函数不会再被调用

参考：**uv_handle_t**的API同样适用

uv_async_t——异步handle

异步handle允许用户在另一个线程中通知(唤醒/wakeup)事件循环

数据类型

uv_async_t
异步handle结构体类型

void(*uv_async_cb)(uv_async_t* handle)
传递给uv_async_init()函数的回调函数的函数指针类型

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)
初始化异步handle，并传入回调函数指针

注意：与其他handle不同的是，这个初始化函数将直接激活异步handle

int uv_async_send(uv_async_t* handle)
唤醒(wakeup)循环并调用异步handle的回调函数

注意：在任意线程调用本函数都是安全的。回调函数将会在事件循环的线程被调用

警告：libuv将会合并对于uv_async_send()函数的调用，这意味着并不是每次调用都对应一个回调函数的调用。例如：如果函数在回调函数被调用之前调用了5次，那么回调函数将只会调用一次。如果uv_async_send()函数在回调函数之后再次被调用，那么回调函数也会再次被调用

参考：uv_handle_t的API同样适用

uv_poll_t——poll handle

poll handle的作用与Linux的poll(2)差不多，主要是为了监视文件描述符的可读性/可写性以及断开情况。

poll handle的目的是使依赖于事件循环的外部库(比如c-ares或者libssh2)可以向循环发送socket状态变化的信号。基于任何其他目的使用uv_poll_t都是不推荐的;uv_tcp_t、uv_udp_t等提供比uv_poll_t更快、伸缩性更好的实现，尤其是在windows平台上。

poll handle对于文件描述符的可读与可写偶尔可能会发送错误的信号。因此在尝试读写文件描述符(fd)时用户应该总是准备好处理EAGAIN或类似的问题。

对同一个socket使用多个活动的poll handle是不可行的，这会导致libuv忙循环(自旋busyloop)或其他的问题。

当一个文件描述符正在被poll handle轮训时，用户不应该关闭它。因为这会导致handle报错，同时也有可能开始轮询另一个socket。在调用uv_poll_stop()或者uv_close()之后文件描述符可以安全的关闭

注意：在windows平台只有socket可以被poll handle轮询，在unix平台，任何被poll(2) 支持的文件描述符都能使用。

数据类型

uv_poll_t
Poll handle结构体类型

void (*uv_poll_cb)(uv_poll_t* handle)
传递给uv_poll_start()函数的回调函数的函数指针类型

uv_poll_event
poll事件类型

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)
通过文件描述符初始化handle
1.2.2版本改动：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)
使用socket描述符初始化poll handle。在unix平台等效于与uv_poll_init()，在windows平台，本函数支持socket句柄
1.2.2版本改动：socket被设置为非阻塞模式

int uv_poll_start(uv_poll_t* handle, int event, uv_poll_cb cb)
开始轮询文件描述符。event是由UV_READABLE、UV_WRITABLE和UV_DISCONNECT 组成的掩码。一旦一个时间被检测到，回调函数会被调用并且status被设置为0，并且检测到的事件类型会被赋值给events成员变量。

UV_DISCONNECT事件是可选的，可能并不会被报告，用户可以选择忽视它，不过这个事件有助于关闭路径(shutdown path)，因为可以避免额外的读或写操作。

如果在轮询时发生错误，statue将为小于0，对应一个UV_E*的错误码。在handle依然活动时，用户不应该关闭socket，否则回调函数会被调用，并且是错误的状态值，但是并不保证一定如此。

注意：在一个已经开始的handle上调用uv_poll_start是可行的。这么做可以跟新监控事件的掩码。

注意：虽然UV_DISCONNECT是可以设置的，但是闭关不推荐在AIX中使用，并且因此回调函数中并不会将events变量设置为这个。

在1.9.0中添加了UV_DISCONNECT事件

uv_poll_stop(uv_poll_t* handle)

停止轮询文件描述符，回调函数也不会再被调用。

参考：uv_handle_t的API同样可用。

uv_signal_t——信号量handle

信号量句柄实现了unix类型的信号量，用来处理每个事件循环的基础。在windows平台模拟了一些信号量的接收

· 当用户按下ctrl+c时会发出信号。就像在unix上一样，当终端的原始模式启动时并不会如此。

· 当用户按下ctrl+break时信号中断

· 当用户关闭控制台窗口时会发生SIGNUP：在SIGNUP，(系统)会给予程序大约10s的时间去进行清理。之后windows会无条 件的终止程序。

· 当libuv检测到控制台程序改变时，会发出SIGWINCH。当程序使用uv_tty_t handle向控制台写入时，libuv会模拟SIGWINCH。SIGWINCH并不总是及时的发出，libuv只是在鼠标移动时检测大小改变。当一个可读的uv_tty_t被用在原始模式（raw mode）下时，调整控制台缓冲区也将引发sigwinch信号。

对其他型号的监视也可以成功的创建，但是这些信号无法收到。这些信号是：
SIGLL，SIGABRT，SIGFPE，SIGSEGV，SIGTERM，SIGKILL。
通过编程调用raise()以及abort()函数的方式发送的信号并不会被libuv检测到，这些也不会触发信号监视者。
注意：在linux平台下，SIGRT0以及SIGRT1(信号32和33)被NPTL pthreads动态库用来管理线程。对这些信号安装监视者将导致不可预知的行为，强烈不推荐如此。libuv的未来版本可能会拒绝它们。

数据类型

uv_signal_t
信号量结构体类型

void (*uv_singnal_cb)(uv_signal_t* handle)
传递给uv_signal_start()函数的回调函数的函数指针类型

公有成员

int uv_signal_t.signum
被这个handle监控的信号，只读

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，并传入回调函数，开始监控制定的信号

int **uv_signal_stop**(uv_signal_t* handle)
停止handle，对应的回调函数将不会再被调用

参考：**uv_handle_t**的API同样适用

uv_process_t——进程handle

进程handle将会生成一个新的进程，允许用户控制它并通过流(streams)与它建立交流渠道。

数据类型

uv_process_t

进程handle结构体类型

uv_process_options_t

生成新进程的配置(传递给uv_spawn()函数)

```
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, int64_t exit_status, int term_signal)

传递给uv_process_options_t函数的回调函数的函数指针定义。uv_process_options_t函数将会指出任何导致进程退出的信号量以及退出状态

uv_process_flags

传递给uv_process_options_t标记成员的标记类型

```
enum uv_process_flags {

        //Set the child process' user id.  设置子进程的用户ID
        UV_PROCESS_SETUID = (1 << 0),
        //Set the child process' group id. 设置子进程的组ID
        UV_PROCESS_SETGID = (1 << 1),
        /*
        * 当将参数列表转换为命令行字符串时，不要对任何参数使用引号或perform any other escaping。
        * 这个配置只在windows系统下起作用，在unix下将被默认忽略。
        */
        UV_PROCESS_WINDOWS_VERBATIM_ARGUMENTS = (1 << 2),
        /*
        * 在分离(detached state)状态下生成子进程——这将使子进程成为一个进程组的起始进程，并会使子进程在父进程
        * 退出后也能持续运行。注意：子进程会保持父进程事件循环alive，除非父进程对子进程handle调用uv_unref()
        */
        UV_PROCESS_DETACHED = (1 << 3),
        /*
        * 隐藏子进程的控制台窗口，只在windows平台下有效。
        */
        UV_PROCESS_WINDOWS_HIDE = (1 << 4)
};
```

uv_stdio_container_t

传递给子进程的stdio句柄或文件描述符的容器

```
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将如何被传给子进程的标记

```
typedef enum {
        UV_IGNORE = 0x00,
        UV_CREATE_PIPE = 0x01,
        UV_INHERIT_FD = 0x02,
        UV_INHERIT_STREAM =
        0x04,  /*
        * 当 UV_CREATE_PIPE被制定, UV_READABLE_PIPE 和 UV_WRITABLE_PIPE
        * 以子进程的角度确定流的方向(读/写)。每一种标记都会指定要创建一个双工数据流B
        */
        UV_READABLE_PIPE = 0x10,
        UV_WRITABLE_PIPE = 0x20
} uv_stdio_flags;
```

公有成员

uv_process_t.pid
生成的新进程的进程ID。在uv_spawn()函数调用之后设置

注意：uv_handle_t成员也可用

uv_process_options_t.exit_cb

进程退出后的回调函数

uv_process_options_t.file

需要运行的进程的路径

uv_process_options_t.args

命令行参数。args[0]应该是进程路径。在windows平台下使用CreateProcess，这个函数会将参数连成一个字符串，这回导致一些奇怪的错误，参考uv_process_flags中的UV_PROCESS_WINDOWS_VERBATIM_ARGUMENTS标记

uv_process_options_t.env

新进程的环境变量。如果为空，将使用父进程的环境

uv_process_options_t.cwd
子进程的当前工作目录

uv_process_options_t.flags
定义uv_sqawn()函数行为的各种标记，参考uv_process_flags

uv_process_options_t.stdio_count
uv_process_options_t.stdio

stdio类型的指针，指向一个uv_stdio_container_t结构的数组，里面是字进程可用的文件描述符。惯例是stdio[0]是stdin(输入)，fd 1是stdout(输出)，fd 2是stderr

注意：在windows平台文件描述符数量只有在子进程使用msvcrt运行时(runtime)环境才能大于等于2

uv_process_options_t.uid
uv_process_options_t.gid
libuv可以改变子进程的用户id以及组id。只有当标记字段设置了合适的位之后才可行
windows平台不可用。uv_sqawn()函数将会失败并返回UV_ENOSUP错误

uv_stdio_container_t.flag
制定stdio容器如何被传到子进程。参考uv_stdio_flags

uv_stdio_container_t.data
包含传递给子进程的流(stream)或者文件描述符的联合体。

API

void uv_disable_stdio_inheritance(void)

使子进程继承与父进程的文件描述符无法再被继承。效果是本进程生成的子进程无法意外的继承这些句柄

推荐在文件描述符被关闭或复制之前尽可能早的调用本函数。

注意：本函数尽最大可能的运行：并不能保证libuv能够发现所有的继承而来的文件描述符。一般来说本函数在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并启动进程。如果进程被成功生成，将会返回0，否则返回错误代码。

可能的失败原因包括但不局限于：可执行文件不存在；没有权限指定setuid或setgid；内存不足。

int uv_process_kill(uv_process_t* handle, int signum)

传递特殊的信号给指定的进程句柄。参考uv_signal_t，尤其是在windows平台下

int uv_kill(int pid, int signum)

传递特殊的信号给指定的PID(进程ID)。

参考：uv_handle_t的API同样适用。

uv_stream_t——流handle

流handle提供一个抽象的双工通信通道。uv_stream_t是抽象类型，libuv提供3种流测实现：uv_tcp_t,uv_pipe_t以及uv_tty_t

数据类型

uv_stream_t
流handle类型

uv_connect_t
链接请求类型

uv_shutdown_t
关闭请求类型

uv_write_t
写请求类型

void (*uv_read_cb)(uv_stream_t* handle, size_t nread, const uv_buf_t* buf)
当流上的数据被读取时的回调函数
当数据可用时，nread>0，错误情况下nread<0。当读到末尾，nread被设为UV_EOF。当nread<0，buf参数将不会指向可用的缓存；在这种情况下，buf.base以及buf.len都被设为0
注意：nread可能是0，但并不表示出错或读到末尾，这相当于在read(2)情况下的EAGAIN 或者 EWOULDBLOCK
当由于调用uv_read_stop()或者uv_close()停止、关闭流发生错误时，由被调用者负责处理处理。尝试继续从该流上读取数据是未定义的行为。
被调用者负责释放buffer，libuv不会再使用它。buffer可能是空的或者错误的

void (*uv_write_cb)(uv_write_t* req, int status)
数据被写入流之后的回调函数。status为0表示成功， <0表示失败

void (*uv_connect_cb)(uv_connect_t* req, int status)
当uv_connect()函数开始的一个链接完成时回调。status为0表示成功， <0表示失败

void (*uv_shutdown_cb)(uv_shutdown_t* req, int status)
当一个关闭请求被处理完之后会被回调。status为0表示成功， <0表示失败

void (*uv_connection_cb)(uv_stream_t* server, int status)
当一个流服务端收到一个链接请求时被回调。用户可以通过uv_accept()函数接受请求。status为0表示成功， <0表示失败

公有成员

size_t uv_stream_t.write_queue_size
包含等待发送的队列字节的数量。只读

uv_stream_t* uv_connect_t.handle
本链接请求针对的流的指针

uv_stream_t* uv_shutdown_t.handle
本关闭请求针对的流的指针

uv_stream_t* uv_write_t.handle
本写请求针对的流的指针

uv_stream_t* uv_write_t.send_handle
指向使用本写请求的流将要发送去的流

参考：uv_handle_t的成员

API

int **uv_shutdown**(uv_shutdown_t* req, uv_stream_t* handle, uv_shutdown_cb cb)

关闭双向流的写端。它将等待正在等待的写请求完成。handle必须指向已经初始化的流。req必须是没有初始化的关闭请求结构体。回调函数将会在关闭请求完成之后回调。

int **uv_listen**(uv_stream_t* stream, int backlog, uv_connection_cb cb)

开始监听即将到来的链接，backlog表示内核可能的链接队列，就像listen(2)。当一个新的链接被收到，回调函数将会被调用

int **uv_accept**(uv_stream_t* server, uv_strea_t* client)

此函数与uv_listen()函数结合使用来接受链接。在uv_connection_cb回调函数中调用这个函数来接受链接。在调用本函数之前，client必须被初始化。返回值<0代表出错。

当uv_conneton_cb被回调时，本函数在第一次执行时将保证成功。如果你尝试多次使用，那么可能会失败。建议每次uv_connection_cb回调中只调用一次。

注意：server由于client都必须在同一个loop

int **uv_read_start**(uv_stream_t* stream, uv_alloc_cb alloc_cb, uv_read_cb read_cb)

从进来的链接中读取数据。uv_read_cb可能会被回调多次直到没有数据去读取，或者uv_read_stop()函数被调用

int **uv_read_stop**(uv_stream_t* stream)

停止读取。uv_read_cb回调函数将不会再被调用

这个函数是幂等的，在已经停止的stream上调用是安全的

int **uv_write**(uv_write_t* req, uv_stream_t* handle, const uv_buf bufs[], unsigned int bufs, uv_write_cb cb)

将数据写到流。buffers按顺序写入。比如：

```
void cb(uv_write_t* req, int status) {

        /* Logic 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 = "4", .len = 1 }
        };

        uv_write_t req1;
        uv_write_t req2;
        /* writes "1234 写入1234" */
        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)

扩展的写函数，用来通过管道发送。管道必须使用ipc==1 初始化

注意：send_handle必须是tcp socket或者管道，可以是服务端或者一个链接。绑定sockets或者管道将被假定为服务端。

int **uv_try_write**(uv_stream_t* handle, const uv_buf_t bufs[], unsigned int nbufs)

与uv_write()函数相同。但是如果不能立刻写入的话不会加入请求队列。

- 返回值>0：写入的字节，可能比预期的少
- <0，错误代码（如果数据不能立刻发送，返回UV_EAGAIN）

int **uv_is_readable**(const uv_stream_t* handle)

如果可读，返回1，否则返回0

int **uv_is_writeable**(const uv_stream_t* handle)

如果可写，返回1，否则返回0

int **uv_stream_set_blocking**(uv_stream_t* handle, int blocking)

启用或者关闭流的阻塞模式

如果启用，所有的的写操作将会同步完成。The interface remains unchanged otherwise。例如：操作的完成或者失败任然通过一个异步回调报告。

警告：不建议过度依赖这个API，它可能会在未来版本被改变。目前在windows仅对uv_pipe_t起作用，在UNIX对所有handle起作用。

libuv也不保证在写请求提交后改变阻塞模式一定会起作用。因此建议在打开或创建流后立即设置阻塞模式。

uv_tcp_t——tcp handle

tcp handle可以代表tcp流与服务端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。并没有创建socket

int uv_tcp_init_ex(uv_loop_t* loop, uv_tcp_t* handle, unsigned int flags)
使用制定的标志初始化handle。目前只有低8位被使用。将会根据此参数创建socket。如果标志是AF_UNSPEC，那么不会创建socket，作用就和uv_tcp_init()一样
1.7.0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版本中的改变：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并不会进行类型检查，但是需要指向有效的stream socket

int **uv_tcp_nodelay**(uv_tcp_t* handle, int enable)
使用/不使用 Nagle's算法

int **uv_tcp_keepalive**(uv_tcp_t* handle, int enable, unsigned int delay)
使用/不使用 TCP的keep-alive。delay是初始延迟秒，如果enable为0就忽略本参数

int **uv_tcp_simultaneous_accepts**(uv_tcp_t* handle, int enable)
使用/不使用监听新的tcp链接时有操作系统维护的同时发生的异步链接队列
这个设置被用来调整TCP服务来达到期望的性能。同时接受可以提高接受链接的速度(默认启用的原因)，但是会导致多进程设置中的不平衡负载分配。

int **uv_tcp_bind**(uv_tcp_t* handle, const struct sockaddr* addr, unsigned int flags)
将handle绑定到一个地址以及端口。addr需要指向已经初始化的sockadd_in或者sockaddr_in6结构体
当端口已经使用时，可以从 uv_tcp_bind(), uv_listen() 或者uv_tcp_connect()函数获取UV_EADDRINUSE错误。也就是说，本函数的成功调用并不保证uv_listen()或者uv_tcp_connect()函数的成功。
flags可以是UV_TCP_IPV6ONLY，这样就只支持IPV6。

int **uv_tcp_getsockname**(const uv_tcp_t* handle, struct sockaddr* name, int namelen)
获取绑定的handle的地址。addr必须指向可用的足够大的内存块。推荐使用sockaddr_stroage结构体，同时支持IPv4以及IPv6。
int **uv_tcp_getpeername**(const uv_tcp_t* handle, struct sockaddr*name, int namelen)
获取链接到本handle的socket的地址。addr必须指向可用的足够大的内存块。推荐使用sockaddr_stroage结构体，同时支持IPv4以及IPv6。
int **uv_tcp_connect**(uv_connect_t* req, uv_tcp_t* handle, const struct sockaddr* addr, uv_connect_cb cb)
建立一个IPv4或者IPv6的TCP链接，提供一个初始化的tcp handle以及一个未初始化的uv_connect_t。addr需要指向一个初始化的socket_in或者sockaddr_in6结构体。

当链接建立了或者发生了错误的时候将会调用回调函数。

参考：uv_stream_t的API同样适用

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。如果标记是AF_UNSPEC，将不会创建socket。
在1.7.0版本中新增

int uv_udp_open(uv_udp_t* handle, uv_os_sock)
打开一个已经存在的文件描述符或者windows的socket作为udp handle

Unix平台下：对于sock参数的唯一要求是准寻数据报协议(比如：在断开模式下，支持sendmsg()和recvmsg())。换句话说，其他的比如原始socket以及netlink socket也可以当作参数

1.2.1：文件描述符设置为非阻塞模式

注意：并不会检测传入的文件描述符或者socket的类型，但是需要是可用的。

int uv_udp_bind(uv_udp_t* handle, const sockaddr* addr, unsigned int flags)
将udp handle绑定到一个IP 端口上
参数：· handle——UDP handle。必须是已经通过uv_udp_init()函数初始化过的
· addr——装有地址的sockaddr_in 或者sockaddr_in6结构体
· flags——制定绑定方式。支持UV_UDP_IPV6ONLY以及UV_UDP_REUSEADDR。
返回值：成功返回0，失败返回<0

int uv_udp_getsockname(const uv_udp_t* handle, struct sockaddr* name, int* namelen)
获取handle的本地IP以及端口
参数：· handle——UDP handle。必须是已经通过uv_udp_init()函数初始化过的
· name——指向地址结构体的指针。可以使用sockaddr_storage结构体来支持IPv4与IPv6
· namelen——作为输入，指定name字段；作为输出，制定所需长度
返回值：成功返回0，失败返回<0

int uv_udp_set_membership(uv_udp_t* handle, const char* multicast_addr, const char* interface_addr, uv_membership membership)
设置多播地址的成员
参数：· handle – UDP handle。必须是已经通过uv_udp_init()函数初始化过的
· multicast_addr – 多播地址
· interface_addr – 接口地址
· membership – 应该为 UV_JOIN_GROUP或者UV_LEAVE_GROUP.
返回值：成功返回0，失败返回<0

int uv_udp_set_multicast_loop(uv_udp_t* handle, int on)
设置多播循环标志。使多播数据包循环回到本地套接字
参数：· handle – UDP handle。必须是已经通过uv_udp_init()函数初始化过的
· on—— 1表示打开， 0表示关闭
返回值：成功返回0，失败返回<0

int uv_udp_set_multicast_ttl(uv_udp_t* handle, int ttl)
设置多播存活时间。
参数：· handle – UDP handle。必须是已经通过uv_udp_init()函数初始化过的
· ttl—— 1到255。
返回值：成功返回0，失败返回<0

int uv_udp_set_multicast_interface(uv_udp_t* handle, const char* interface_addr)
设置多播接口用来发送或者接收数据
参数：· handle – UDP handle。必须是已经通过uv_udp_init()函数初始化过的
· interface_addr——接口地址
返回值：成功返回0，失败返回<0

int uv_udp_set_broadcast(uv_udp_t* handle, int on)
打开或关闭广播
参数：· handle – UDP handle。必须是已经通过uv_udp_init()函数初始化过的
· on—— 1是打开 0是关闭
返回值：成功返回0，失败返回<0

int uv_udp_set_ttl(uv_udp_t* handle, int ttl)
设置udp包存活时间
参数：· handle – UDP handle。必须是已经通过uv_udp_init()函数初始化过的
· ttl—— 1到255
返回值：成功返回0，失败返回<0

int uv_udp_send(uv_udp_send_t* req, uv_udp_t* handle, const uv_buf_t ubfs[], unsigned int nbufs, const sockaddr* add, uv_udp_send_cb cb)

通过udp套接字发送数据。如果handle并没有经过绑定，那么默认绑定到0.0.0.0和一个随机的端口。

参数：· req——udp请求句柄。需要初始化
· handle – UDP handle。必须是已经通过uv_udp_init()函数初始化过的
· bufs——需要发送的buffer的列表
· nbufs——buffer数量
· addr——发送目的地地址。sockaddr_in或者sockaddr_in6
· send_cb——数据发送之后的回调函数

返回值：成功返回0，失败返回<0

int uv_udp_tey_send(uv_tcp_t* handle, const uv_buf_t bufs[], unsigned int nbufs, const struct sockaddr* addr)

与uv_udp_send()函数类似，但是如果不能立刻发送出去就不会将请求放入队列

返回值：>=0：发送的数据量(bytes，与给定的buffer大小相同)；<0：错误代码

int uv_udp_recv_start(uv_udp_t* handle, uv_alloc_cb alloc_cb, uv_udp_recv_cb recv_cb)

准备接收数据。如果socket没有绑定地址，那么就会绑定到0.0.0.0以及一个随机的端口。

参数：· handle – UDP handle。必须是已经通过uv_udp_init()函数初始化过的
· alloc_cb——需要临时空间时会回调
· recv_cb——收到数据时回调

返回值：成功返回0，失败返回<0

int uv_udp_recv_stop(uv_udp_t* handle)

停止接收数据

参数：· handle – UDP handle。必须是已经通过uv_udp_init()函数初始化过的

返回值：成功返回0，失败返回<0

参考：uv_handle_t的API同样适用

uv_fs_event_t——文件系统事件handle

文件系统事件handle允许用户监视一个给定路径的变化。例如：文件被重命名或发生了其他的改变。这个handle使用各个平台上最有效的方法实现。

注意：对于AIX，必须安装非默认的IBM bos.ahafs包
- ahafs跟踪监控每个进程，并且不是线程安全的。A separate process must be spawned for each monitor for the same event.
- 如果只监控文件夹，那么并不能收到文件改动的通知

参考文档获取跟多。

数据类型

uv_fs_event_t

文件系统事件handle类型

void (***uv_fs_event_cb**)(uv_fs_event_t* handle, const char* filename, int events, int status)

传递给uv_fs_event_start()函数的回调函数的函数指针。filename参数为相对路径。events是ORed掩码。

uv_fs_event

uv_fs_event_t监控的事件类型

```
enum uv_fs_event {
    UV_RENAME = 1,
    UV_CHANGE = 2
};
```

uv_fs_event_flags

传递给uv_fs_event_start函数的标记量

```
enum uv_fs_event_flags {
/*
* 默认情况下，如果fs事件监控的是文件夹名字，将会监控文件夹下所有的文件
* 本标记改变这个行为，并让fs_event只报告对于文件夹的改动
* 此标志不影响个别文件的监控
* 此标志目前尚未在任何后端实现
*/
UV_FS_EVENT_WATCH_ENTRY = 1,
/*
* 默认情况下uv_fs_event会尝试使用内核接口(inotify或kqueue)来发现事件
* 这在远程文件系统上行不通。这个标记让fs_event定期调用stat函数 。
* 目前本标记尚未在任何后台实现
*/
UV_FS_EVENT_STAT = 2,
/*
* 默认情况下, 对文件夹的监控并不监控子目录, 本标记在支持监控子目录的平台上使其支持监控子目录
*/
UV_FS_EVENT_RECURSIVE = 4
};
```

公有成员

无
参考：uv_handle_t的成员

API

int **uv_fs_event_init**(uv_loop_t* loop, uv_fs_event_t* handle)
初始化句柄

int **uv_fs_event_start**(uv_fs_event_t* handle, uv_fs_event_cb cb, const char* path, unsigned int flags)
开始事件，并传入回调函数。flags是uv_fs_event_flags的 ORed mask
注意：目前在OSX以及windows平台下只支持UV_FS_EVENT_RECURSIVE

int **uv_fs_event_stop**(uv_fs_event_t* handle)
停止handle。

int **uv_fs_event_getpath**(uv_fs_event_t* handle, char* buffer, size_t* size)
获取handle监控的路径。buffer的内存由用户分配。成功返回0，失败返回 <0。如果成功，buffer包含路径，size是长度。如果buffer不够大。返回UV_ENOBUFS，size将会被设为需要的长度
1.3.0：需要长度不再包含末尾的null，buffer也不再是null结尾。

参考：uv_handle_t的API同样可用

uv_fs_poll_t——文件系统轮询handle

文件系统轮询handle允许用户监控给定的路径的变化。与uv_fs_event_t不同的是，本handle使用stat检测当文件更改，因此在一些fs event无法使用的地方可以使用uv_fs_poll_t

数据类型

uv_fs_poll_t
文件轮询handle类型

void (*uv_fs_poll_cb)(uv_fs_poll_t* handle, int status, const uv_stat_t* prev, const uv_stat_t* curr)
传递给uv_fs_poll_start()函数的回调函数的函数指针。
本函数被调用时，如果status<0表示文件路径不存在或者无法访问。监控不会停止，但是回调函数也不会再被调用直到情况发生了改变(比如：文件被创建或者错误原因发生改变)
当status == 0，回调函数收到新、旧两个指向uv_stat_t结构体的指针，它们只在回调期间内有效

公有成员

无
参考：uv_handle_t的成员

API

int **uv_fs_poll_init**(uv_loop_t* loop, uv_fs_poll_t* handle)
初始化handle

int **uv_fs_poll_start**(uv_fs_poll_t* handle, uv_fs_poll_cb poll_cb, const char* path, unsigned int interval)
每隔interval秒检测一次文件
注意：为了最大的可移植性，使用多秒间隔，因为在许多文件系统，次秒级的间隔无法检测出所有的改变

int **uv_fs_poll_stop**(uv_fs_poll_t* handle)
停止handle。

int **uv_fs_poll_getpath**(uv_fs_poll_t* handle, char* buffer, size_t* size)
获取handle监控的路径。buffer空间由用户申请。成功返回0，失败返回<0。如果长度不够，返回UV_ENOBUFS，并将长度设给size。
1.3.0：长度不再包含末尾的null，buffer也不再是以null结尾的字符串

参考：uv_handle_t的API同样适用

文件系统操作(Filesystem operations)

libuv提供了多样的跨平台的同步以及异步的文件操作。本节所有的函数都需要一个可以为空的回调。如果回调函数为null，请求将会同步完成，否则异步执行

所有的操作在线程池上进行，参考**线程池调度**获取更多信息

数据类型

uv_fs_t
    文件系统请求类型

uv_timespec_t
    相当于timespec结构体
        typedef struct {
            long tv_sec;
            long tv_nsec;
        } uv_timespec_t;

uv_stat_t
    相当于stat结构体
        typedef struct {
            int64_t st_size;
            uint64_t st_blksize;
            uint64_t st_blocks;
            uint64_t st_flags;
            uint64_t st_gen;
            uv_timespec_t st_atim;
            uv_timespec_t st_mtim;
            uv_timespec_t st_ctim;
            uv_timespec_t st_birthtim;
        } uv_stat_t;

uv_fs_type
    文件系统请求类型
        typedef enum {
        UV_FS_UNKNOWN = -1,
        UV_FS_CUSTOM,
        UV_FS_OPEN,
        UV_FS_CLOSE,
        UV_FS_READ,
        UV_FS_WRITE,
        UV_FS_SENDFILE,
        UV_FS_STAT,
        UV_FS_LSTAT,
        UV_FS_FSTAT,
        UV_FS_FTRUNCATE,
        UV_FS_UTIME,
        UV_FS_FUTIME,
        UV_FS_ACCESS,
        UV_FS_CHMOD,
        UV_FS_FCHMOD,
        UV_FS_FSYNC,
        UV_FS_FDATASYNC,
        UV_FS_UNLINK,
        UV_FS_RMDIR,
        UV_FS_MKDIR,
        UV_FS_MKDTEMP,
        UV_FS_RENAME,
        UV_FS_SCANDIR,
        UV_FS_LINK,
        UV_FS_SYMLINK,
        UV_FS_READLINK,
        UV_FS_CHOWN,
        UV_FS_FCHOWN
        } uv_fs_type;

uv_dirent_t
相当于一个跨平台的dirent结构体。在uv_fs_scandir_next()中使用

```
typedef enum {
        UV_DIRENT_UNKNOWN,
        UV_DIRENT_FILE,
        UV_DIRENT_DIR,
        UV_DIRENT_LINK,
        UV_DIRENT_FIFO,
        UV_DIRENT_SOCKET,
        UV_DIRENT_CHAR,
        UV_DIRENT_BLOCK
} uv_dirent_type_t;

typedef struct uv_dirent_s {
        const char* name;
        uv_dirent_type_t type;
} uv_dirent_t;
```

公有成员

uv_loop_t* uv_fs_t.loop
指向开启请求以及完成请求之后汇报的循环的指针，只读。

uv_fs_type uv_fs_t.fs_type
文件系统请求类型

const char* uv_fs_t.path
请求相关的文件路径

ssize_t uv_fs_t.result
请求的结果。<0表示错误，否则表示成功。在像uv_fs_read()以及uv_fs_write()的请求中，这个只代表对应操作的数据量。

uv_stat_t uv_fs_t.statbuf
存储uv_fs_stat()以及其他stat请求的结果。

void* uv_fs_t.ptr
作为statbuf的别名存储uv_fs_readlink()的结果

参考：uv_req_t的成员

API

void uv_fs_req_cleanup(uv_fs_t* req)
清理请求。必须在已经释放了任何libuv可能申请的请求的内存之后调用。

int uv_fs_close(uv_loop_t* loop, uv_fs_t* req, uv_file file, uv_fs_cb cb)
等同于close(2)

int uv_fs_open(uv_loop_t* loop, uv_fs_t* req, const char* path, int flags, int mode, uv_fs_cb cb)
等同于open(2)
注意：在windows平台使用CreateFileW，因此文件总是以二进制方式打开。所以不支持O_BINARY以及O_TEXT模式

int uv_fs_read(uv_loop_tt* loop, uv_fs_t* req, uv_file file, const uv_buf_t bufs[], unsigned int nbufs, int64_t offset, uv_fs_cb cb)
等同于preadv(2)

int uv_fs_unlink(uv_loop_t* loop, uv_fs_t* req, const char* path, uv_fs_cb cb)
等同于unlink(2)

int uv_fs_write(uv_loop_t* loop, uv_fs_t* req, uv_file file, const uv_buf_t bufs[], unsigned int nbufs, int64_t offset, uv_fs_cb cb)
等同于pwritev(2).

int uv_fs_mkdir(uv_loop_t* loop, uv_fs_t* req, const char* path, int mode, uv_fs_cb cb)
等同于 mkdir(2).
注意：mode参数目前在windows平台下并未实现

int uv_fs_mkdtemp(uv_loop_t* loop, uv_fs_t* req, const char* tpl, uv_fs_cb cb)
等同于mkdtemp(3).
注意：可以通过req-->path参数获取结果，结果是一个null结尾的字符串

int uv_fs_rmdir(uv_loop_t* loop, uv_fs_t* req, const char* path, uv_fs_cb cb)
等同于rmdir(2).

int uv_fs_scandir(uv_loop_t* loop, uv_fs_t* req, const char* path, int flags, uv_fs_cb cb)
int uv_fs_scandir_next(uv_fs_t* req, uv_dirent_t* ent)
等同于scandir(3)，只有稍微的区别。一旦本请求的回调函数被调用，用户可以使用uv_fs_scandir_next函数来获取下一个入口的数据。当没有入口时，返回UV_EOF。
注意：与scandir(3)函数不同的是，本函数不回返回'.'和'..'的文件
注意：在unix平台下，只有部分文件系统支持获取一个入口类型的功能，参考 getdents(2)

int uv_fs_stat(uv_loop_t* loop, uv_fs_t* req, const char* path, uv_fs_cb cb)
int uv_fs_fstat(uv_loop_t* loop, uv_fs_t* req, uv_file file, uv_fs_cb cb)
int uv_fs_lstat(uv_loop_t* loop, uv_fs_t* req, const char* path, uv_fs_cb cb)
分别等同于stat(2), fstat(2) 以及 fstat(2)

int uv_fs_rename(uv_loop_t* loop, uv_fs_t* req, const char* new_path, uv_fs_cb cb)
等同于 rename(2).

int uv_fs_fsync(uv_loop_t* loop, uv_fs_t* req, uv_file file, uv_fs_cb cb)
等同于 fsync(2).

int uv_fs_fdatasync(uv_loop_t* loop, uv_fs_t* req, uv_file file, uv_fs_cb cb)
等同于 fdatasync(2).

int uv_fs_ftruncate(uv_loop_t* loop, uv_fs_t* req, uv_file file, int64_t offset, uv_fs_cb cb)
等同于 ftruncate(2).

int uv_fs_sendfile(uv_loop_t* loop, uv_fs_t* req, uv_file out_fd, uv_file in_fd, int64_t in_offset, size_t length, uv_fs_cb cb)
有限的等同于 sendfile(2).

int uv_fs_access(uv_loop_t* loop, uv_fs_t* req, const char* path, int mode, uv_fs_cb cb)
unix平台等同于 access(2) . Windows 平台使用 GetFileAttributesW().

int uv_fs_chmod(uv_loop_t* loop, uv_fs_t* req, const char* path, int mode, uv_fs_cb cb)
int uv_fs_fchmod(uv_loop_t* loop, uv_fs_t* req, uv_file file, int mode, uv_fs_cb cb)
分别等同于 chmod(2) 和 fchmod(2)

int uv_fs_utime(uv_loop_t* loop, uv_fs_t* req, const char* path, double atime, double mtime, uv_fs_cb cb)
int uv_fs_futime(uv_loop_t* loop, uv_fs_t* req, uv_file file, double atime, double mtime, uv_fs_cb cb)
分别等同于 utime(2) and futime(2) ？？ respectively.
注意：AIX：本函数只在AIX7.1以及更新的版本中支持。在旧版本调用会返回UV_ENOSYS
1.10.0：windows平台支持亚秒级精度

int uv_fs_link(uv_loop_t* loop, uv_fs_t* req, const char* path, const char* new_path, uv_fs_cb cb)
等同于link(2).

int uv_fs_symlink(uv_loop_t* loop, uv_fs_t* req, const char* path, const char* new_path, int flags, uv_fs_cb cb)
等同于 symlink(2).
注意：在windows平台flags参数可以制定创建symlink的方式
- UV_FS_SYMLINK_DIR：表示将路径指向文件夹
- UV_FS_SYMLINK_JUNCTION：要求使用junction point创建symlink

int uv_fs_readlink(uv_loop_t* loop, uv_fs_t* req, const char* path, uv_fs_cb cb)
等同于 readlink(2).

int uv_fs_realpath(uv_loop_t* loop, uv_fs_t* req, const char* path, uv_fs_cb cb)
unix平台下等同于 realpath(3) . Windows 平台下使用GetFinalPathNameByHandle().
注意：本函数在windowsXP以及Windows Server20003平台下未实现。
在1.8.0版本中新增

int uv_fs_chown(uv_loop_t* loop, uv_fs_t* req, const char* path, uv_uid_t uid, uv_gid_t gid, uv_fs_cb cb)
int uv_fs_fchown(uv_loop_t* loop, uv_fs_t* req, uv_file file, uv_uid_t uid, uv_gid_t gid, uv_fs_cb cb)
等同于 chown(2) 和 fchown(2)
注意：这些函数在windows平台未实现

参考：uv_req_t的API同样适用

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