

## 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:

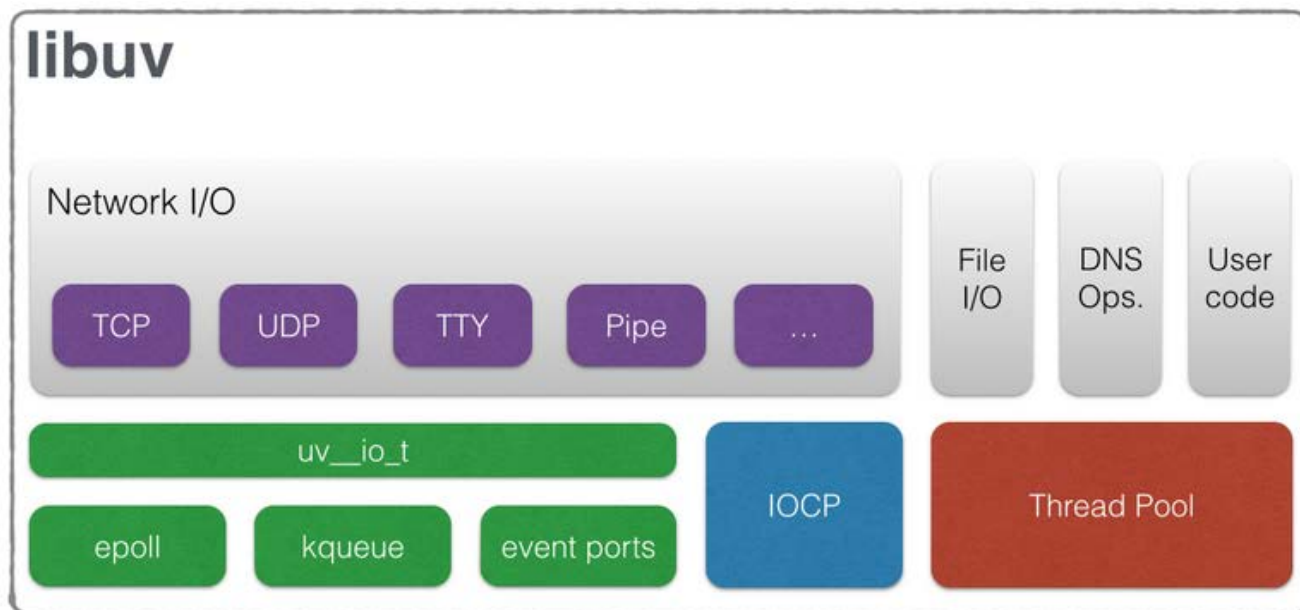
- 设计概述
- 错误处理
- 版本检测宏与函数
- uv\_loop\_t——事件循环
- uv\_handle\_t——基础句柄
- uv\_req\_t——基础请求
- uv\_timer\_t——定时器句柄
- uv\_prepare\_t——预处理handle
- uv\_check\_t——check handle
- uv\_idle\_t——空转handle
- uv\_async\_t——异步handle
- uv\_poll\_t——poll handle
- uv\_signal\_t——信号量handle
- uv\_process\_t——进程handle
- uv\_stream\_t——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。

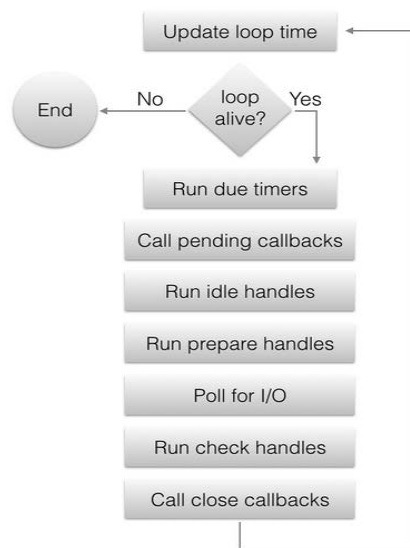
请求(Requests)通常是代表操作的短寿命对象。这些操作能够通过一个句柄(handle)进行——写请求被用来在一个handle上写入数据；也可以是独立的——getaddrinfo请求不需要handle，直接运行在循环上。

## 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时间，以减少时间相关的系统调用次数。
- 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 handle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回调没有被调用，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。这意味着新的APIs可以在主版本更新中被引进。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的支持

### 宏

```
UV_VERSION_MAJOR
    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_RELEASE
    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\_init()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\_DEFAULT：loop一直运行到所有的被使用的句柄以及请求。如果uv\_stop()被调用，有未完成的handle和request返回非0值，否则返回0  
UV\_RUN\_ONCE：只运行一次。注意，如果有等待的回调函数，本函数会阻塞。如果有未完成的handles和requests返回0，否则返回非0值。  
UV\_RUN\_NOWAIT：只运行一次，但是如果有等待的回调函数不回阻塞。返回值UV\_RUN\_ONCE

**int uv\_loop\_alive(const uv\_loop\_t\* loop)**  
如果loop有未完成的句柄或者request返回非0值

**void uv\_stop(uv\_loop\_t\* loop)**  
停止事件循环，会uv\_run()函数返回。不会在下一个事件代开始起作用。如果在i/o阻塞用了一个句柄，不会立即阻塞

**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)**  
获取后端文件描述符，只有kqueue, epoll and event ports支持

int **uv\_backend\_timeout**(const uv\_loop\_t\* loop)  
获取 的 时。返回值是 ，或者 1代表 有 时设置

uint\_t **uv\_now**(const uv\_loop\_t\* loop)  
返回当 时 ， 是 。时 在 代的开始时被缓存 来。  
时 一个 意的时 增。不要对开始的时 设。  
注意： 用 uv\_hrtime()来获取 的 度

void **uv\_update\_time**(uv\_loop\_t\* loop)  
更新loop定 的 “当 时 ”。libuv在 代开始的时 缓存当 时 。  
一 来 不需要 用 数， 非 的回 数需要阻塞 一段长的时

void **uv\_walk**(uv\_loop\_t\* loop, uv\_walk\_cb walk\_cb, void\* arg)  
alk the list of handles walk\_cb将会被 用 传 arg参数



# uv\_handle\_t——句柄 (Base handle)

uv\_handle\_t是libuv的 有句柄类型的

于结 是一 的， 以libuv的 有句柄结 为uv\_handle\_t。

## 数据类型

uv\_handle\_t  
libuv的 句柄类型

uv\_any\_handle  
有句柄类型的

void (\*uv\_alloc\_cb)(uv\_handle\_t\* handle, size\_t suggested\_size, uv\_buf\_t\* buf)  
传 uv\_read\_start()以及uv\_udp\_rcv\_start() 数的回 数的 数指 的定 。用 必须提供uv\_buf\_t结 的 。 的大小 是 3 ，但是 不一定需要 。将大小设为0将会在uv\_udp\_rcv\_cb以及uv\_read\_cb回 中 UV\_ENOBUFS的错误

void (\*uv\_close\_cb)(uv\_handle\_t\* handle)  
传 uv\_close() 数的回 数的 数指 的定

## 公有成员

uv\_loop\_t\* uv\_handle\_t.loop  
行当 handle的loop的指 ，只 。

void\* uv\_handle\_t.data  
指 用 定 的数据，libuv 不 用 成员

## API

int **uv\_is\_active**(const uv\_handle\_t\* handle)  
如果handle是 的， 返回非0值，否 返回0。 代表的意 据handle的类型 有 不  
uv\_async\_t handle 是 的， 不是 的， 非 (uv\_close)  
uv\_pipe\_t, uv\_tcp\_t, uv\_udp\_t等handle—— 本 i o的句柄——当 及到i o 时是 的， 如 链  
接 接 链接等  
uv\_check\_t, uv\_idle\_t, uv\_timer\_t等句柄，当uv\_check\_start() uv\_idle\_start()等被 用时，是 的  
经 ：如果一个uv\_foo\_t有一个对 的uv\_foo\_start() 数， 在 数被 用时，handle是 的， ， uv\_foo\_stop()  
将会 再 为非

int **uv\_is\_closing**(const uv\_handle\_t\* handle)  
如果handle已经被 或者 在被 ，返回非0值，否 返回0  
注意：本 数只 在handle初始化以及close回 用 被 用。

void **uv\_close**(uv\_handle\_t\* handle, uv\_close\_cb close\_cb)  
handle。close\_cb将会在此 数 被 用。本 数必须在handle 存 被 用  
文件描述的handle将会被 ，但是close\_cb 会在下一个 代中 用，以此来 用 会 的  
。  
对于 在 求的handle， 如uv\_connect\_t或者uv\_write\_t， 求将会取 ， 回 数会被 用，传  
UV\_E AN E E 。

void **uv\_ref**(uv\_handle\_t\* handle)  
用传 的handle。 用是 等的， 表 如果一个handle已经被 用了， 用 个 数将不 作用。  
参考 eference counting

void **uv\_unref**(uv\_handle\_t\* handle)  
用传 的handle。如果 有被 用， 不 作用  
参考 eference counting

int **uv\_has\_ref**(const uv\_handle\_t\* handle)  
如果被 用了，返回非0值，否 返回0

size\_t **uv\_handle\_size**(uv\_handle\_type type)  
返回传 的handle类型的大小

## API 函数

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)

获取或者设置用于 socket 的数据缓存的大小

如果 \*value 为 0，将会返回当前设置的大小，否则设置新的大小

本函数用于 unix 平台的 P U P 管道 handle，以及 windows 平台的 P U P

注意：linux 将会设置为 value 的，返回来的大小

int **uv\_recv\_buffer\_size**(uv\_handle\_t\* handle, int\* value)

获取或者设置用于 socket 的接收缓存的大小

如果 \*value 为 0，将会返回当前设置的大小，否则设置新的大小

本函数用于 unix 平台的 P U P 管道 handle，以及 windows 平台的 P U P

注意：linux 将会设置为 value 的，返回来的大小

int **uv\_fileno**(const uv\_handle\_t\* handle, uv\_os\_fd\_t\* fd)

获取平台的文件描述符

支持以下句柄：P pipes U P 以及 poll。传 handle 将会 UV\_EINVAL

如果一个 handle 有一个文件描述符或者已被，本函数会返回 UV\_EBADF

：用本函数必须非常小心，为 libuv 设置文件描述符的，可以

错误

## 用数

libuv 的 timer 将会一行到有的以及被用的 handle。通常用已被用的 handle，用可以 loop 提，例如在 `uv_timer_start()` 用 `uv_unref()`

一个 handle 可以是被动用或有被动用，用数不用计数器，以有的作用是等的

当被动式时，有的 handles 是被用的，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\_timer\_t——定时器句柄( timer 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\_async\_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 是unix的poll(2) 不 是， 要是为了 文件描述符的可 以可 以及断开 。

poll handle的 的是 于 件 的 ( 如c-ares或者libssh2)可以 的 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 用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\_E\_READ UV\_E\_WRITE UV\_E\_READ\_WRITE和UV\_E\_DISCONNECT 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\_E\_DISCONNECT 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\_E\_DISCONNECT 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\_E\_DISCONNECT 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

句柄实现了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\_signal\_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

UV\_P\_O ESS\_SE UI (1 0),  
Set the child process user id. 设置子进程的用 I

UV\_P\_O ESS\_SE GI (1 1),  
Set the child process group id. 设置子进程的 I

\*  
\* 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_IGNORE 0x00,
        UV_EPIPE 0x01,
        UV_EOF 0x02,
        UV_EINTR 0x04,
        *
        * 当 UV_EPIPE被 定, UV_EPIPE和 UV_EINTR
        * 以子进程的 度 定 的 ( )。 一种 会指定要 一个 工数据 B
        *
        UV_EPIPE 0x10,
        UV_EINTR 0x20
    uv_stdio_flags

```

## 公有成员

uv\_process\_t.pid  
成的新进程的进程ID。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 CreateProcess, 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\_PROCESSES\_IN\_OVERLAPPED\_AGUENS

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\_spawn() 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 uid以及 gid。只有当 字段设置了 的 可行 windows平台不可用。uv\_spawn() 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)`

子进程继承父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：本数大可为行：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\_ O U\_ 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。将等待在等待的请求。handle必须是指已经初始化的。req必须是指有初始化的请求。回调函数将会在请求完成时回调。

int **uv\_listen**(uv\_stream\_t\* stream, int backlog, uv\_connection\_cb cb)  
开始监听将来的连接，backlog表示可接受的连接数，listen(2)。当一个新的连接被接受到，回调函数将会被调用。

int **uv\_accept**(uv\_stream\_t\* server, uv\_stream\_t\* client)  
此函数uv\_listen()的结束用来接受连接。在uv\_connection\_cb回调函数中用该函数来接受连接。在本函数中，client必须被初始化。返回值0代表错误。  
当uv\_connection\_cb被回调时，本函数在下一行时将完成。如果成功，可能会。  
uv\_connection\_cb回调中只使用一个。  
注意：server和client必须在同一个loop中。

int **uv\_read\_start**(uv\_stream\_t\* stream, uv\_alloc\_cb alloc\_cb, uv\_read\_cb read\_cb)  
进来的连接中读取数据。uv\_read\_cb可能会被回调到有数据去读取，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。  
该函数是阻塞的，在已经停止的stream上使用是安全的。

int **uv\_write**(uv\_write\_t\* req, uv\_stream\_t\* handle, const uv\_buf\_t bufs, unsigned int nbufs, uv\_write\_cb cb)  
将数据写入。buffers。例如：

```
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 1234567890123456 */
uv_write(&req1, stream, a, 2, cb)
uv_write(&req2, stream, b, 2, cb)
```

int **uv\_write2**(uv\_write\_t\* req, uv\_stream\_t\* handle, const uv\_buf\_t bufs, unsigned int nbufs, uv\_stream\_t\* send\_handle, uv\_write\_cb cb)  
的函数，用来通过管道。管道必须用ipc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或者管道将被绑定为服务。

int **uv\_try\_write**(uv\_stream\_t\* handle, const uv\_buf\_t bufs, unsigned int nbufs)  
uv\_write()函数。但是如果不会阻塞。  
返回值0：写入的字节，可预期的。  
非0，错误代码（如果数据不，返回UV\_EAGAIN）

int **uv\_is\_readable**(const uv\_stream\_t\* handle)  
如果可读，返回1，否则返回0

int **uv\_is\_writable**(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)  
用或者阻塞模式。  
如果用，有的操作将会完成。the interface remains unchanged otherwise。例如：操作完成或者通过一个回调。  
：不度个API，可能会在来版本被。在windows对uv\_pipe\_t作用，在UNI对uv\_handle作用。  
libuv也不在请求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, size\_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:

- 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**(uv\_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

提供一个函数获取当前 libuv 的版本:

1. uv\_version, 获取数字版本
2. uv\_version\_string 获取字符串版本

## tree.h

定义 (splay tree) 和 (B tree)

1. 在 windows 平台 用
2. 定义
  - a. define B\_EA (name, type) 结构体, 包含一个节点指针 (如 uvwin.h 中的 uv\_timer\_s, 其中 uv\_timer\_s 中包含 UV\_I E \_P I V A E \_F I E 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) 初始化节点
 

```
define B_INI (root) do
    (root) rbh_root NU
    while ( * ONS ON * 0)
```
  - c. define B\_EN (type) 节点, 包含子节点, 节点以及节点
  - d. 节点的定义, 要包含一个节点 下一个节点 大节点 小节点 查找等, 可以参考 stl 的 (stl 通常模式支持节点包含不同类型的数据, 用一定的, libuv 通常来实, 不同类型的节点包含 B\_EN, 的操作本是对定的结)

## uv win.h

定义平台 的

1. 定义模型 (Overlapped I O) 用到的 socket 数指
  - a. typedef int ( SA API \* PFN\_ SA E V )
  - b. typedef int ( SA API \* PFN\_ SA E VF O )
2. typedef struct uv\_buf\_t 结构体, 够 为 SABUF
3. B\_EA (uv\_timer\_tree\_s, uv\_timer\_s) 定义 一 uv\_timer\_tree\_s, 节点 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 msec. *
    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 \_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. 在32平台下, 如果定义了BUILD\_SHARED\_LIBS 是 数 (libuv工程在工程配置 c c 编译器 中定)

```
if defined(BUILD_SHARED_LIBS)
    * Building shared library. *
    define UV_EXPORT __declspec(dllexport)
```

如果定义了BUILD\_SHARED\_LIBS (用户定义), 是 数

2. define UV\_EXPORT\_NO\_API() 定义了错误名以及对错误的表的, 中 可以是一个, 如在uv\_common.h中的:

```
define UV_EXPORT_NO_API(name,_) case UV_EXPORT name return name
switch 句 用, 返回错误名。 中错误的代码通 定 :
typedef enum
    define (code,_) UV_EXPORT code UV_EXPORT code,
    UV_EXPORT_NO_API()
    undef
    UV_EXPORT_NO_API UV_EXPORT_EOF 1
    uv_errno_t
```

3. define UV\_EXPORT\_NO\_API() , 句柄类型 表, 时参考:

```
typedef enum
    UV_EXPORT_NO_API UV_EXPORT_0,
    define (uc,lc) UV_EXPORT uc,
    UV_EXPORT_NO_API UV_EXPORT_0
    undef
    UV_EXPORT_0,
    UV_EXPORT_NO_API UV_EXPORT_0
    uv_handle_type
```

4. define UV\_EXPORT\_NO\_API() , 定 求类型 表, 时参考:

```
typedef enum
    UV_EXPORT_NO_API UV_EXPORT_0,
    define (uc,lc) UV_EXPORT uc,
    UV_EXPORT_NO_API UV_EXPORT_0
    undef
    UV_EXPORT_NO_API UV_EXPORT_0
    UV_EXPORT_NO_API UV_EXPORT_0
    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_EXPORT_NO_API UV_EXPORT_0
    uv_loop_option

typedef enum
    UV_EXPORT_NO_API UV_EXPORT_0,
    UV_EXPORT_NO_API UV_EXPORT_0,
    UV_EXPORT_NO_API UV_EXPORT_0
    uv_run_mode
```

5. 数 明, 以及 明 种 数指

8. 定 种结



