libuv documentation

发布 *1.26.0*

libuv contributors

2019 年 **02** 月 **28** 日

Contents

CHAPTER 1

摘要

libuv是一个强调异步I/O的多平台支持库。 开发它 主要是用于 Node.js， 但它也被用在 Luvit 、 Julia 、 pyuv 和 其他项目 。

注解: 如果你在这份翻译中发现问题你可以发送 pull requests 来帮助！

CHAPTER 2

特点

- 全功能的事件循环基于epoll、kqueue、IOCP、event ports
- 异步的TCP和UDP套接字
- 异步的DNS解析
- 异步的文件和文件系统操作
- 文件系统事件
- ANSI转义代码控制的TTY
- IPC包括套接字共享，使用Unix域套接字或有名管道（Windows）
- 子进程
- 线程池
- 信号处理
- 高分辨率时钟
- 线程和同步原语

CHAPTER 3

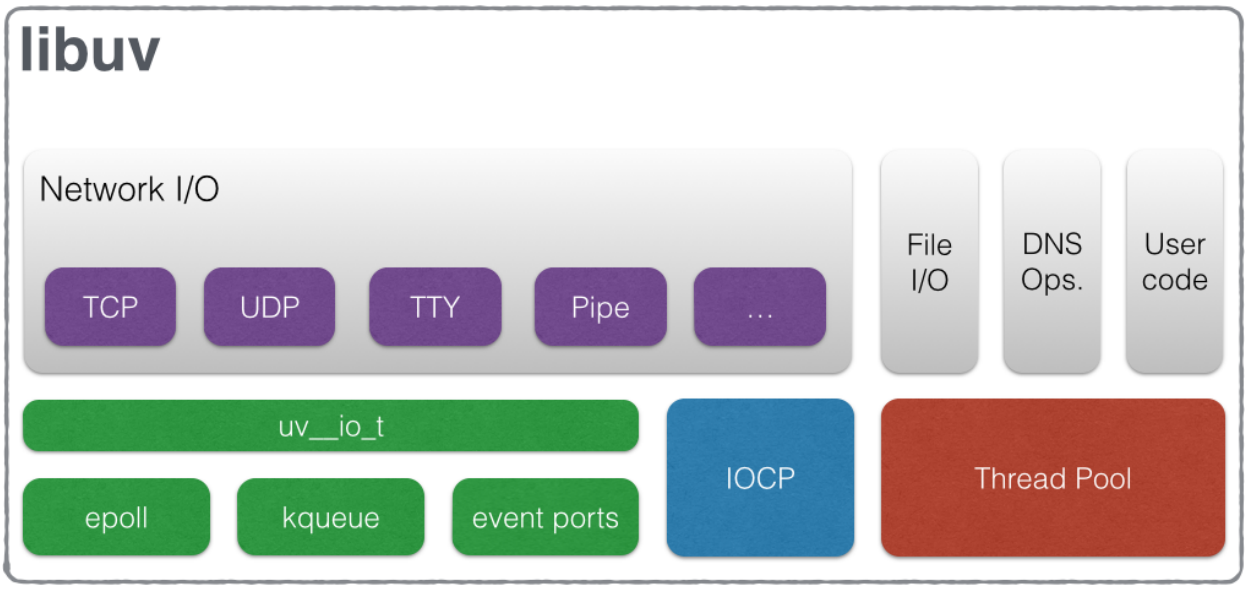
文档

3.1 设计摘要

libuv 是一个跨平台支持库，原先为 NodeJS 而写。它围绕着事件驱动的异步I/O模型而设计。

这个库提供不仅仅是对不同I/O轮询机制的简单抽象，还包括： ‘句柄’和‘流’对套接字和其他实体提供了高级别的抽象； 也提供了跨平台的文件I/O和线程功能，以及其他一些东西。

这是一份图表解释了组成libuv的不同组件和它们相关联的子系统：



3.1.1 句柄和请求

libuv 提供给用户使用两个抽象，与事件循环相配合： 句柄 和 请求。

句柄 表示能够在活动时执行特定操作的长期存在的对象。 比方说:

- 一个准备句柄当活动时每一次循环迭代调用它的回调函数。
- 一个TCP服务器句柄每次有新连接时调用它的connection回调函数。

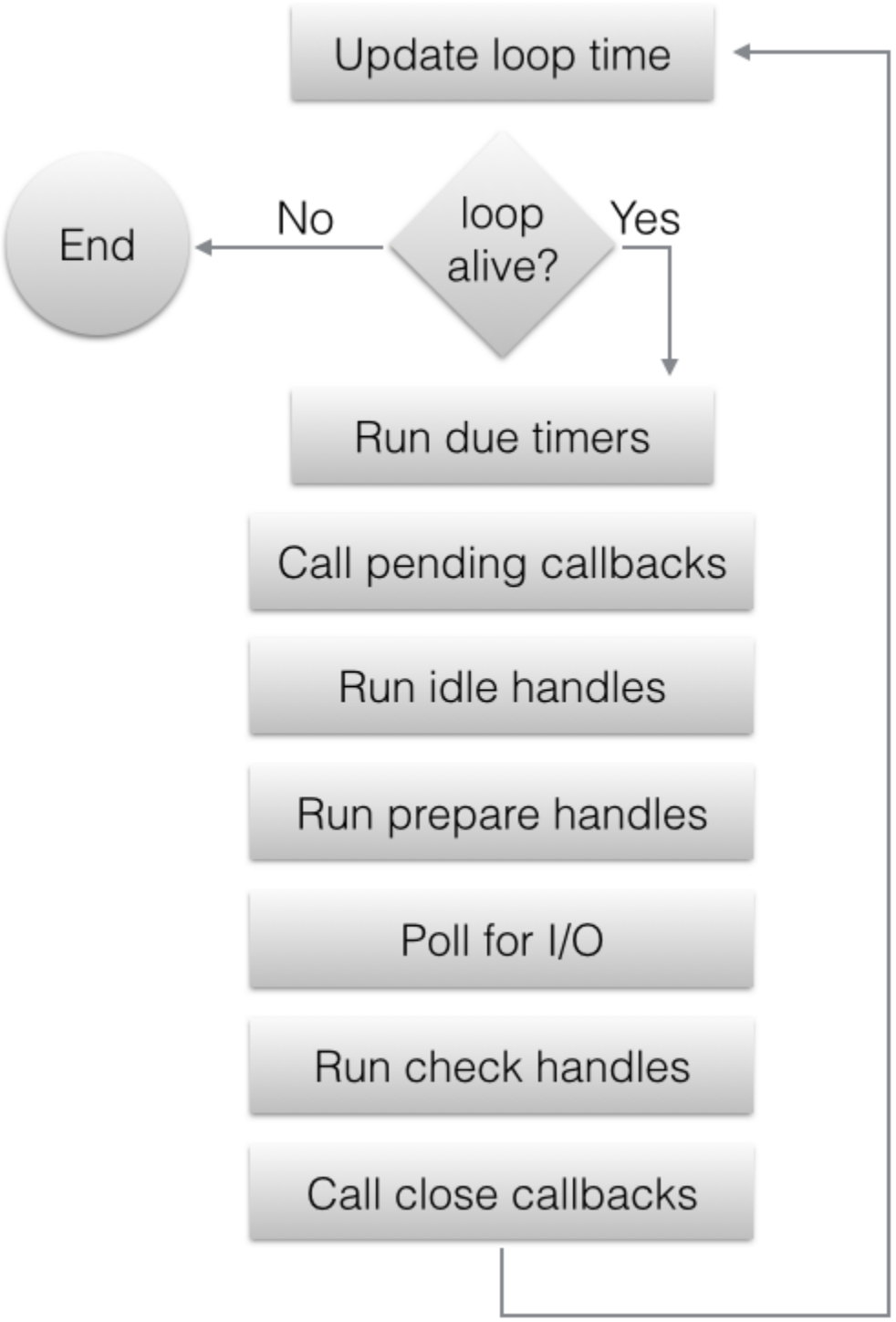
请求代表着（通常是）短期的操作。 这些操作可以通过一个句柄执行： 写请求用于在句柄上写数据；或是独立不需要句柄的：getaddrinfo 请求 不需要句柄，它们直接在循环上运行。

3.1.2 I/O 循环

I/O（或 事件）循环是 libuv 的核心组件。 它为全部I/O操作建立内容， 并且它必须关联到单个线程。 可以运行多个事件循环 只要每个运行在不同的线程。 libuv 事件循环（或任何其他涉及循环或句柄的API，就此而言） **不是线程安全的** 除非另行说明。

事件循环遵循很常见的单线程异步I/O方法：全部（网络） I/O 在非阻塞的套接字上执行，在给定平台上使用最好的可用的机制来轮询： epoll在Linux上、kqueue在OSX和其他BSD上，event ports 在SunOS上和IOCP在Windows上。 作为循环迭代的一部分，循环将阻塞等待套接字上已被添加到轮询器的I/O活动， 并且回调函数将被触发以指示套接字状态 （可读、可写挂起），这样句柄能够读、写或执行所需要的I/O操作。

为了更好地理解事件循环怎么运作，下面的图表显示了一次循环迭代的所有阶段:



1. 循环概念 'now' 被更新。 在开始事件循环计的时候事件循环缓存当前的时间以减少时间相关的系统调用的数目。
2. 如果循环处于 活动 状态的话一次迭代开始，否则的话循环立刻终止。 那么， 何时一个循环确定是活动 的？ 如果一个循环有活动的和被引用的句柄、 活动的请求或正在关闭的句柄，它被确定为 活动的。
3. 运行适当的计时器。 所有在循环概念 *now* 之前到期的活动的计时器的回调函数被调用。

4. 待处理的回调函数被调用。 大多数情况下，在I/O轮询之后所有的I/O回调函数会被调用。 然而有些情况下，这些回调推延到下一次迭代中。 如果前一次的迭代推延了任何的I/O回调函数的调用，回调函数将在此刻运行。
5. 空转句柄的回调函数被调用。 虽有不恰当的名字，当其活动时空转句柄在每次循环迭代时都会运行。
6. 准备句柄的回调函数被调用。 在循环将为I/O阻塞前，准备句柄的回调函数被调用。
7. 计算轮询时限。 在为I/O阻塞前，循环计算出它应该阻塞多长时间。 这些是计算时限的规则：
   - 如果循环使用 UV_RUN_NOWAIT 标志运行，时限是0。
   - 如果循环即将被终止（*uv_stop()* 被调用），时限是0。
   - 如果没有活动的句柄或请求，时限是0。
   - 如果没有任何活动的空转句柄，时限是0。
   - 如果有任何待处理的句柄，时限是0。
   - 如果以上均不符合，采用最近的计时器的时限，如果没有活动计时器的话，为无穷大。
8. 循环为I/O阻塞。 此刻循环将按上一步计算的时限为I/O阻塞。 对于所有监视给定文件描述符读写操作的I/O相关的句柄， 在此刻它们的回调函数被调用。
9. 检查句柄的回调函数被调用。 在循环为I/O阻塞之后，检查句柄的回调函数被调用。 检查句柄基本上与准备句柄相辅相成。
10. 关闭 回调函数被调用。 如果一个句柄通过调用 *uv_close()* 被关闭， 它的回调函数将被调用。
11. 当循环以 UV_RUN_ONCE 的特别情况下，它意味着前移。 在I/O阻塞之后也许没有触发I/O回调函数，但是已经过去了一些时间， 所以可能有到期的计时器，这些计时器的回调函数被调用。
12. 迭代结束。 如果循环以 UV_RUN_NOWAIT 或 UV_RUN_ONCE 模式运行则 迭代结束且 *uv_run()* 将返回。 如果循环以 UV_RUN_DEFAULT 运行， 它将继续从头开始如果它仍是 活动 的，否则的话它也会结束。

重要： libuv 使用了一个线程池来使得异步文件I/O操作可实现， 但是网络I/O **总是** 在单线程中执行，即每个循环的线程。

注解： 虽然轮询机制不同，libuv 提供了在Unix系统和Windows下一致的执行模型。

3.1.3 文件 I/O

不像是网络 I/O，libuv 没有平台特定的文件I/O原语可以依靠， 因此目前的方案是在线程池中运行阻塞的文件I/O操作。

对跨平台文件I/O规划的详尽介绍，参考 这篇文章 。

libuv 目前使用一个全局的线程池，各类循环都能够在它上面排队执行。 目前在这个池上运作的有3种操作：

- 文件系统操作
- DNS功能 (getaddrinfo 和 getnameinfo)
- 用户定义的代码于 *uv_queue_work()*

警告： 见 *Thread pool work scheduling* 部分获取更多详细信息，但是记好了此线程池的大小非常有限。

3.2 API文档

3.2.1 错误处理

在libuv中，错误是负数常量。 根据经验， 不管何时有一个状态参数，或是一个API函数返回一个整数的时候， 一个负数意味着一个错误。

当一个使用回调函数的函数返回了一个错误， 这个回调函数将永远不会被调用。

注解： 实现的细节：在 Unix 中错误代码是负的 *errno* （或者说 *-errno*），当在 Windows 上它们由libuv定义为任意的负数。

错误常量

UV_E2BIG
参数列表太长了

UV_EACCES
权限被拒绝

UV_EADDRINUSE
地址已经被使用

UV_EADDRNOTAVAIL
地址不可用

UV_EAFNOSUPPORT
不支持的地址族

UV_EAGAIN
资源临时不可用

UV_EAI_ADDRFAMILY
不支持的地址族

UV_EAI_AGAIN
临时性失败

UV_EAI_BADFLAGS
错的 ai_flags 值

UV_EAI_BADHINTS
无效的指示值

UV_EAI_CANCELED
请求取消了

UV_EAI_FAIL
永久性失败

UV_EAI_FAMILY
ai_family 不支持

UV_EAI_MEMORY
内存用完

UV_EAI_NODATA
没有地址

UV_EAI_NONAME
未知的代码或服务

UV_EAI_OVERFLOW
参数缓存越界

UV_EAI_PROTOCOL
解析的协议未知

UV_EAI_SERVICE
对套接字类型，服务不可用

UV_EAI_SOCKTYPE
套接字类型不支持

UV_EALREADY
连接已经在进行中

UV_EBADF
错的文件描述符

UV_EBUSY
资源忙或是锁定了

UV_ECANCELED
操作取消了

UV_ECHARSET
非法的 Unicode 字符

UV_ECONNABORTED
软件导致的连接中止

UV_ECONNREFUSED
连接被拒绝

UV_ECONNRESET
连接被远端重置

UV_EDESTADDRREQ
需要目的地址

UV_EEXIST
文件已经存在

UV_EFAULT
在系统调用参数里有错的地址

UV_EFBIG
文件太大了

UV_EHOSTUNREACH
主机不可达

UV_EINTR
中断的系统调用

UV_EINVAL
非法参数

UV_EIO
I/O 错误

UV_EISCONN
套接字已连接

UV_EISDIR
在目录上的非法操作

UV_ELOOP
遇到了太多符号链接

UV_EMFILE
打开的文件太多了

UV_EMSGSIZE
消息太长了

UV_ENAMETOOLONG
名字太长了

UV_ENETDOWN
网络停机

UV_ENETUNREACH
网络不可达

UV_ENFILE
文件表溢出

UV_ENOBUFS
没有可用的缓存空间

UV_ENODEV
没有这样的设备

UV_ENOENT
没哟这样的文件或目录

UV_ENOMEM
内存不够

UV_ENONET
机器不在网络上

UV_ENOPROTOOPT
协议不可用

UV_ENOSPC
设备上没有剩余空间

UV_ENOSYS
未被实现的函数

UV_ENOTCONN
套接字未连接

UV_ENOTDIR
不是一个目录

UV_ENOTEMPTY
目录非空

UV_ENOTSOCK
在非套接字上进行套接字操作

UV_ENOTSUP
套接字不支持的操作

UV_EPERM
不允许的操作

UV_EPIPE
破碎的管道

UV_EPROTO
协议错误

UV_EPROTONOSUPPORT
协议不支持

UV_EPROTOTYPE
对套接字的错误的协议类型

UV_ERANGE
结果太大了

UV_EROFS
只读的文件系统

UV_ESHUTDOWN
不能在传输终点关机后发送

UV_ESPIPE
非法查寻

UV_ESRCH
没有这样的进程

UV_ETIMEDOUT
连接超时

UV_ETXTBSY
文本文件忙

UV_EXDEV
不允许跨设备链接

UV_UNKNOWN
未知错误

UV_EOF
文件结尾

UV_ENXIO
没有这样的设备或地址

UV_EMLINK
太多的链接

API

UV_ERRNO_MAP(iter_macro)
对以上每个错误常量扩展出一系列的 *iter_macro* 调用的宏。 *iter_macro* 以两个参数调用：不带 *UV_* 前缀的错误常量名， 和错误信息字符串字面量。

const char* **uv_strerror**(int *err*)
返回对应给定错误代码的错误信息。 泄漏一些字节的内存，当你以未知的错误代码调用它时。

char* **uv_strerror_r**(int *err*, char* *buf*, size_t *buflen*)
返回对应给定错误代码的错误信息。 以零结尾的信息存储在用户提供的缓冲区 *buf* 里，不超过 *buflen* 字节。

1.22.0 新版功能.

const char* **uv_err_name**(int *err*)
返回对应给定错误代码的错误名。 泄漏一些字节的内存，当你以未知的错误代码调用它时。

char* **uv_err_name_r**(int *err*, char* *buf*, size_t *buflen*)
返回对应给定错误代码的错误名。 以零结尾的名称存储在用户提供的缓冲区 *buf* 里，不超过 *buflen* 字节。

1.22.0 新版功能.

int **uv_translate_sys_error**(int *sys_errno*)
返回等同于给定平台相关错误代码的libuv错误代码： POSIX 错误代码在 Unix 上（存储于 *errno* ），和Win32错误代码在Windows上（ *GetLastError()* 或 *WSAGetLastError()* 返回的）。

如果 *sys_errno* 已经是一个libuv错误，则直接返回。

在 1.10.0 版更改: function declared public.

3.2.2 用于版本检测的宏和函数

始于版本1.0.0，libuv遵循 semantic versioning 模式。这意味着新的API能在一个主版本发布的任何时期引入。 在这个部分你将会了解所有允许你有条件地编写或编译代码的宏和函数， 用于跟多个libuv版本打交道。

宏

UV_VERSION_MAJOR
libuv版本中的主编号。

UV_VERSION_MINOR
libuv版本中的次编号。

UV_VERSION_PATCH
libuv版本中的补丁编号。

UV_VERSION_IS_RELEASE
设置成 1 代表着一个libuv的发布版，0 表示开发版快照。

UV_VERSION_SUFFIX
libuv版本后缀。特定的开发发布版本例如 Release Candidates 可能有个后缀比如 "rc" 。

UV_VERSION_HEX
返回libuv的版本信息打包进单个整数中。每个部分占8位， 补丁版本在最低8位。 比方说，libuv 1.2.3 就是 0x010203 。

1.7.0 新版功能.

函数

unsigned int **uv_version**(void)
返回 *UV_VERSION_HEX*.

const char* **uv_version_string**(void)
以字符串形式返回libuv版本号。 对于非发布版本，版本后缀也包括在内。

3.2.3 uv_loop_t — 事件循环

事件循环是libuv功能的核心。 它负责处理I/O轮询，和基于不同事件源的将被运行的回调函数的调度。

数据类型

uv_loop_t
循环 数据类型。

uv_run_mode
用在以 *uv_run()* 运行循环的模式。

```
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*)
传递给 *uv_walk()* 的回调函数的类型定义。

公共成员

void* **uv_loop_t.data**
用户定义的任意数据的空间。 libuv不使用且不触及这个字段。

API

int **uv_loop_init**(*uv_loop_t** *loop*)
初始化给定的 *uv_loop_t* 结构体。

int **uv_loop_configure**(*uv_loop_t** *loop*, uv_loop_option *option*, ...)
1.0.2 新版功能.

设置额外的循环选项。 你通常应该在第一次调用 *uv_run()* 之前先调用这个，除非另行说明。

返回 0 若成功， 或一个UV_E* 错误代码若失败。 准备好处理 UV_ENOSYS ； 它意味着循环选项不被平台所支持。

支持的选项:

- UV_LOOP_BLOCK_SIGNAL: 当轮询新事件时阻塞信号。 给 *uv_loop_configure()* 的第二个参数是信号编号。

  这个选项当前仅实现了SIGPROF信号， 用于当使用抽样性能分析器时压制不必要的唤醒。 请求其他的信号将会以 UV_EINVAL 失败。

int **uv_loop_close**(*uv_loop_t* loop*)
释放所有的内部循环资源。 调用这个函数仅当循环已经结束执行并且所有开放的句柄和请求已经被关闭， 否则的话它将返回 UV_EBUSY 。 在这个函数返回后， 用户可以释放为循环分配的内存。

*uv_loop_t** **uv_default_loop**(void)
返回初始化过的默认循环。 它可能返回 NULL 如若内存分配失败。

这个函数就是对于需要一个贯穿应用程序的全局循环的一个简单方式， 默认循环与 `uv_loop_init()` 初始化的循环并无差异。 就其本身而言， 默认循环可以（并且应该）被 `uv_loop_close()` 关闭， 以便与它关联的资源被释放。

> **警告:** 这个函数不是线程安全。

int **uv_run**(*uv_loop_t* loop*, *uv_run_mode mode*)
这个函数运行事件循环。 它将依指定的模式而采取不同的行为：

- UV_RUN_DEFAULT：运行事件循环直到没有更多的活动的和被引用到的句柄或请求。 返回非零如果 `uv_stop()` 被调用且仍有活动的句柄或请求。 在所有其他情况下返回零。
- UV_RUN_ONCE：轮询I/O一次。 注意如若没有待处理的回调函数这个函数阻塞。 返回零当完成时（没有剩余的活动的句柄或请求）， 或者非零值如果期望更多回调函数时 （意味着你应该在未来某时再次运行这个事件循环）。
- UV_RUN_NOWAIT：轮询I/O一次但不会阻塞，如若没有待处理的回调函数时。 返回零当完成时（没有剩余的活动的句柄或请求）， 或者非零值如果期望更多回调函数时 （意味着你应该在未来某时再次运行这个事件循环）。

int **uv_loop_alive**(const *uv_loop_t* loop*)
如果有被引用的活动句柄、活动请求或者循环里的关闭句柄时返回非零值。

void **uv_stop**(*uv_loop_t* loop*)
停止事件循环，致使 `uv_run()` 尽快结束。 这不快于下次循环迭代。 如果这个函数在I/O阻塞前被调用， 这次循环将不会为I/O阻塞。

size_t **uv_loop_size**(void)
返回 *uv_loop_t* 结构体的大小。 对不想知道结构体布局的FFI绑定作者有用。

int **uv_backend_fd**(const *uv_loop_t* loop*)
获取后端文件描述符。 仅支持kqueue、epoll和event ports。

这可以跟 *uv_run(loop, UV_RUN_NOWAIT)* 协同使用， 来在一个线程内轮询同时在另一个线程内运行事件循环的回调函数。 详见 test/test-embed.c 获取示例。

> **注解:** 在另一个kqueue轮询集合里嵌入一个kqueue文件描述符不在所有平台有效。 不是一个添加描述符导致的错误，而是不会产生事件。

int **uv_backend_timeout**(const *uv_loop_t* loop*)
获取轮询时限。 返回值是微秒，或者 -1 当没有时限的时候。

uint64_t **uv_now**(const *uv_loop_t* loop*)
以微秒返回当前的时间戳。 时间戳在事件循环计时开始的时候缓存， 详见 `uv_update_time()` 获取细节和原理。

这个时间戳在一些任意的时间点单调增加。 不要对开始点作出假设，你将只会感到沮丧。

> **注解:** 用 `uv_hrtime()` 若你需要亚毫秒粒度。

void **uv_update_time**(*uv_loop_t* loop*)
更新事件循环概念 "now" 。 Libuv在事件循环计时开始时缓存当前时间， 以减少时间相关的系统调用数目。

你通常将不会需要调用这个函数， 除非你有在更长时间周期内阻塞事件循环的回调函数， 此处 "更长" 有点主观但大概一毫秒或更多。

void **uv_walk**(*uv_loop_t* loop*, *uv_walk_cb walk_cb*, void* *arg*)
遍历句柄列表： *walk_cb* 将以给定的 *arg* 被执行。

int **uv_loop_fork**(*uv_loop_t* loop*)
1.12.0 新版功能.

fork(2) 系统调用后在子进程内重新初始化任何必要的内核状态。

先前开启的监视器将继续在子进程内开启。

有必要在每次在父线程里建立事件循环时显式调用这个函数， 如若你计划在子进程里继续使用这个循环， 包括默认循环（即便你不在父线程里继续使用它）。 这个函数必须在调用 *uv_run()* 或任何其他使用到子进程里的循环的API函数之前调用。 这么做失败了将导致未定义的行为， 可能包括发给父子进程重复的事件或是中止子进程。

如果可能，优先在子进程建立一个新循环而不是复用父进程创建的循环。 fork后在子进程新建的循环不应该使用这个函数。

这个函数未在 Windows 上实现，这里返回 UV_ENOSYS 。

警告： 这个函数是实验性的。 它可能包含bug，且可能修改或删除。 无法保证 API 和 ABI 稳定性。

注解： 在 Mac OS X 上，如果目录FS事件句柄用在父进程的 任何事件循环 中， 子进程将不再能够使用最有效率的FSEvent实现。 相反，在子进程使用目录FS事件句柄将回退到用于文件和其他基于kqueue系统的同等实现。

警告： 在 AIX 和 SunOS 上，在fork时已经在父进程开启的FS事件句柄将 不会 在子进程里分发事件； 它们必须被关闭且重启。 在所有其他系统上，它们继续正常工作无需任何进一步介入。

警告： 任何之前从 *uv_backend_fd()* 返回的值现在无效了。 那个函数必须再次调用以确定正确的后端文件描述符。

void* **uv_loop_get_data**(const *uv_loop_t* loop*)
返回 *loop->data* 。

1.19.0 新版功能.

void* **uv_loop_set_data**(*uv_loop_t* loop*, void* *data*)
设置 *loop->data* 为 *data* 。

1.19.0 新版功能.

3.2.4 uv_handle_t — 基础句柄

uv_handle_t 是所有libuv句柄类型的基类型。

结构体是对齐的以便任何libuv句柄能转化成 *uv_handle_t*。 这里定义的所有API函数适用于任意句柄类型。

libuv句柄无法移动。 传递给函数的句柄结构体指针在请求的操作期间必须保持有效。 当使用栈分配的句柄时请小心。

数据类型

uv_handle_t
基础的libuv句柄类型。

uv_handle_type
libuv句柄的种类。

```
typedef enum {
  UV_UNKNOWN_HANDLE = 0,
  UV_ASYNC,
  UV_CHECK,
  UV_FS_EVENT,
  UV_FS_POLL,
  UV_HANDLE,
  UV_IDLE,
  UV_NAMED_PIPE,
  UV_POLL,
  UV_PREPARE,
  UV_PROCESS,
  UV_STREAM,
  UV_TCP,
  UV_TIMER,
  UV_TTY,
  UV_UDP,
  UV_SIGNAL,
  UV_FILE,
  UV_HANDLE_TYPE_MAX
} uv_handle_type;
```

uv_any_handle
所有句柄类型的并集。

void **(*uv_alloc_cb)**(*uv_handle_t** *handle*, size_t *suggested_size*, *uv_buf_t** *buf*)
传递给 *uv_read_start()* 和 *uv_udp_recv_start()* 的回调函数的类型定义。 用户必须分配内存并填充 *uv_buf_t* 结构体。 如果赋值NULL作为缓冲区的基址或0作为缓冲区长度， 将会在 *uv_udp_recv_cb* 或 *uv_read_cb* 的回调函数里触发一个 UV_ENOBUFS 错误。

提供了建议的大小（目前大多数情况是65536），但这只是个标示， 与待读取的数据没有任何关系。用户自行决定分配多少内存。

例如，使用了像freelists、分配池或slab分配器这类自定义分配模式的程序可能决定使用一个不同的大小以匹配于它们所用的memory chunks。

例子：

```
static void my_alloc_cb(uv_handle_t* handle, size_t suggested_size, uv_buf_t*␣
↪buf) {
  buf->base = malloc(suggested_size);
```

(continues on next page)

(续上页)

```
  buf->len = suggested_size;
}
```

void **(*uv_close_cb)**(*uv_handle_t** *handle*)
传递给 *uv_close()* 的回调函数的类型定义。

公共成员

*uv_loop_t** **uv_handle_t.loop**
指向句柄所运行在的 *uv_loop_t* 。 只读。

uv_handle_type **uv_handle_t.type**
uv_handle_type ，指向潜在句柄的类型。 只读。

void* **uv_handle_t.data**
用户定义的任意数据的空间。 libuv不使用且不触及这个字段。

API

UV_HANDLE_TYPE_MAP(iter_macro)
对每个句柄类型扩展出一系列的 *iter_macro* 调用的宏。 *iter_macro* 以两个参数调用：不带 *UV_* 前缀的 *uv_handle_type* 元素名， 和不带 *uv_* 前缀和 *_t* 后缀的对应的结构体类型名。

int **uv_is_active**(const *uv_handle_t** *handle*)
如果句柄活动返回非零值，如果不活动返回零。 "活动" 的意思依赖于句柄类型：

- uv_async_t 句柄总是活动的且无法不活动， 除非以 uv_close() 关闭它。
- uv_pipe_t、uv_tcp_t、uv_udp_t 等句柄 —— 基本上任何处理I/O的句柄 —— 当做牵涉到I/O的事情时是活动的， 像读、写、连接、接受新连接等等。
- uv_check_t, uv_idle_t, uv_timer_t 等句柄是活动的当以 uv_check_start(), uv_idle_start() 等调用开始时。

经验法则：如果一个 *uv_foo_t* 类型的句柄有个 *uv_foo_start()* 函数， 则它从那个函数调用起是活动的。同样地， *uv_foo_stop()* 使句柄再次不活动。

int **uv_is_closing**(const *uv_handle_t** *handle*)
如果句柄正在关闭或已经关闭返回非零值，否则零。

注解： 这个函数只应该在句柄初始化和关闭回调函数到来前这段时间内被使用。

---

void **uv_close**(*uv_handle_t** *handle*, *uv_close_cb* *close_cb*)
请求句柄关闭。 *close_cb* 将在这个调用之后被异步调用。 这个函数必须在每个句柄释放内存前调用。此外，内存只能在 *close_cb* 里或者返回后释放。

封装文件描述符的句柄立即关闭， 但是 *close_cb* 将会被推迟到下次事件循环迭代。 给你释放关联于句柄的任何资源的机会。

进行中请求，像 uv_connect_t 或 uv_write_t， 被取消并且它们的回调函数以 status=UV_ECANCELED 被异步调用。

void **uv_ref**(*uv_handle_t** *handle*)
引用给定的句柄。 引用是幂等的，也就是说， 如果已经被引用的句柄再调用这个函数无效果。

See 引用计数.

void **uv_unref**(*uv_handle_t* handle*)
反引用给定的句柄。 引用是幂等的，也就是说， 如果没有被引用的句柄再调用这个函数无效果。

See 引用计数.

int **uv_has_ref**(const *uv_handle_t* handle*)
如果句柄被引用了，返回非零，否则是零。

See 引用计数.

size_t **uv_handle_size**(*uv_handle_type type*)
返回给定句柄类型的大小。 对不想知道结构体布局的FFI绑定作者有用。

杂项 API函数

下面的API函数用到一个 `uv_handle_t` 类型的参数，但是它们只适用于某些句柄类型。

int **uv_send_buffer_size**(*uv_handle_t* handle*, int* *value*)
获取或是设置操作系统用于套接字的发送缓存的大小。

如果 **value* == 0 ，将返回当前发送缓存大小， 否则它将用 **value* 设置新的发送缓存大小。

此函数在Unix上对TCP、管道和UDP句柄有效， 在Windows上对TCP和UDP句柄有效。

注解： Linux将会设置双倍的大小，并且返回的是原先设置值的双倍大小。

---

int **uv_recv_buffer_size**(*uv_handle_t* handle*, int* *value*)
获取或是设置操作系统用于套接字的接收缓存的大小。

如果 **value* == 0 ，将返回当前接收缓存大小， 否则它将用 **value* 设置新的发送接收大小。

此函数在Unix上对TCP、管道和UDP句柄有效， 在Windows上对TCP和UDP句柄有效。

注解： Linux将会设置双倍的大小，并且返回的是原先设置值的双倍大小。

---

int **uv_fileno**(const *uv_handle_t* handle*, *uv_os_fd_t* fd*)
获取平台相关的等效文件描述符。

以下句柄被支持：TCP、管道、TTY、UDP和轮询。 传递其他任何句柄类型将会以 *UV_EINVAL* 失败。

如果一个句柄尚未有依附的文件描述符或句柄被关闭了， 这个返回将返回 *UV_EBADF* 。

警告： 使用这个函数时请小心。 libuv假定它控制着文件描述符，所以对文件描述符的任何改变可能引发失灵。

*uv_loop_t** **uv_handle_get_loop**(const *uv_handle_t* handle*)
返回 *handle->loop* 。

1.19.0 新版功能.

void* **uv_handle_get_data**(const *uv_handle_t* handle*)
返回 *handle->data* 。

1.19.0 新版功能.

void* **uv_handle_set_data**(*uv_handle_t** *handle*, void* *data*)
设置 *handle->data* 为 *data* 。

1.19.0 新版功能.

uv_handle_type **uv_handle_get_type**(const *uv_handle_t** *handle*)
返回 *handle->type* 。

1.19.0 新版功能.

const char* **uv_handle_type_name**(*uv_handle_type* *type*)
返回给定句柄类型等效的结构体名称， 例如对 *UV_NAMED_PIPE* 是 *"pipe"* （即 `uv_pipe_t` ）。

如果不存在这样的句柄类型，它返回 *NULL* 。

1.19.0 新版功能.

引用计数

libuv事件循环（如果运行在默认模式）运行直至没有剩下活动的 和 被引用的句柄。用户能够通过对活动句柄解引用来强制循环提前退出， 例如调用 `uv_unref()` 在调用 `uv_timer_start()` 之后。

句柄可以被引用和解引用，引用计数模式没有用到计数器， 所以两种操作是幂等的。

所有活动句柄默认是被引用的，参见 `uv_is_active()` 获取在 活动的 关联上更详尽的解释。

3.2.5 uv_req_t — 基础请求

uv_req_t 是所有libuv请求类型的基类型。

结构体是对齐的以便任何libuv请求能转化成 *uv_req_t*。 这里定义的所有API函数适用于任意请求类型。

数据类型

uv_req_t
基础libuv请求结构体。

uv_any_req
所有请求类型的并集。

公共成员

void* **uv_req_t.data**
用户定义的任意数据的空间。 libuv不使用且不触及这个字段。

uv_req_type **uv_req_t.type**
指向请求的类型。 只读。

```
typedef enum {
    UV_UNKNOWN_REQ = 0,
    UV_REQ,
    UV_CONNECT,
    UV_WRITE,
    UV_SHUTDOWN,
    UV_UDP_SEND,
    UV_FS,
```

(continues on next page)

(续上页)

```
    UV_WORK,
    UV_GETADDRINFO,
    UV_GETNAMEINFO,
    UV_REQ_TYPE_MAX,
} uv_req_type;
```

API

UV_REQ_TYPE_MAP (iter_macro)
对每个请求类型扩展出一系列的 *iter_macro* 调用的宏。 *iter_macro* 以两个参数调用：不带 *UV_* 前缀的 *uv_req_type* 元素名， 和不带 *uv_* 前缀和 *_t* 后缀的对应的结构体类型名。

int **uv_cancel** (*uv_req_t** *req*)
取消待处理的请求。 如果请求在执行中或已经执行完毕时失败。

返回 0 当成功时，或者一个 < 0 的错误代码当失败时。

当前仅支持取消 *uv_fs_t* 、 *uv_getaddrinfo_t* 、 *uv_getnameinfo_t* 和 *uv_work_t* 请求。

取消的请求的回调函数在未来某时被调用。 释放关联于请求的内存是 不 安全的直到回调函数调用之后。

这是取消如何报告给回调函数的方式：

- 一个 *uv_fs_t* 请求的 req->result 字段设为 *UV_ECANCELED* 。
- 一个 *uv_work_t* 、 *uv_getaddrinfo_t* 或 *uv_getnameinfo_t* 请求的回调函数以 status == *UV_ECANCELED* 被调用。

size_t **uv_req_size** (uv_req_type *type*)
返回给定请求类型的大小。 对不想知道结构体布局的FFI绑定作者有用。

void* **uv_req_get_data** (const *uv_req_t** *req*)
返回 *req->data*.

1.19.0 新版功能.

void* **uv_req_set_data** (*uv_req_t** *req*, void* *data*)
设置 *req->data* 为 *data*.

1.19.0 新版功能.

uv_req_type **uv_req_get_type** (const *uv_req_t** *req*)
返回 *req->type*.

1.19.0 新版功能.

const char* **uv_req_type_name** (uv_req_type *type*)
返回给定请求类型等效的结构体名称， 例如对 *UV_CONNECT* 是 *"connect"* （即 *uv_connect_t* ）。

如果不存在这样的请求类型，它返回 *NULL* 。

1.19.0 新版功能.

3.2.6 uv_timer_t — 计时器句柄

计时器句柄用于调度回调函数在将来被调用。

数据类型

uv_timer_t
计时器句柄类型。

void **(*uv_timer_cb)** (*uv_timer_t* handle*)
传递给 *uv_timer_start()* 的回调函数的类型定义。

公共成员

N/A

参见:

uv_handle_t 的成员也适用。

API

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

int **uv_timer_start** (*uv_timer_t* handle*, *uv_timer_cb* cb*, uint64_t *timeout*, uint64_t *repeat*)
开启计时器。 *timeout* 和 *repeat* 以微秒计。

如果 *timeout* 是零，回调函数在下一次事件循环迭代时运行。 如果 *repeat* 是非零值，回调函数首次在 *timeout* 毫秒后运行，之后每隔 *repeat* 毫秒重复运行。

注解: 不要更新事件循环概念"now"。 详见 *uv_update_time()* 获取更多信息。

如果计时器已经活动，它简单地被更新。

---

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

int **uv_timer_again** (*uv_timer_t* handle*)
停止计时器，并且如果它是重复的则用repeat值作为timeout重启它。 如果计时器之前从未被启用，返回 UV_EINVAL 。

void **uv_timer_set_repeat** (*uv_timer_t* handle*, uint64_t *repeat*)
以微秒设置重复间隔值。 计时器将以给定的间隔被调度运行， 不管回调函数的执行周期，并且在时间片超出时遵循正常计时器语义。

比方说，如果一个50ms重复的计时器首次运行于17ms， 它将在33ms之后再次运行。如果在首次回调函数之后其他的任务花费了超过33ms， 则回调函数将会尽可能快运行。

注解: 如果repeat值从一个计时器回调函数里被设置，它不会立即生效。 如果计时器之前不是重复的，之前的计时将被停止。 如果它是重复的，则旧的repeat值将被用来调度下一次时限。

---

uint64_t **uv_timer_get_repeat** (const *uv_timer_t* handle*)
获取计时器 repeat 值。

参见:

uv_handle_t 的API函数也适用。

3.2.7 uv_prepare_t — 准备句柄

准备句柄将在每次循环迭代时运行给定的回调函数一次， 在I/O轮询前一刻。

数据类型

uv_prepare_t
准备句柄类型。

void **(*uv_prepare_cb)**(*uv_prepare_t* handle*)
传递给 *uv_prepare_start()* 的回调函数的类型定义。

公共成员

N/A

参见:

uv_handle_t 的成员也适用。

API

int **uv_prepare_init**(*uv_loop_t* loop*, *uv_prepare_t* prepare*)
初始化句柄。

int **uv_prepare_start**(*uv_prepare_t* prepare*, *uv_prepare_cb cb*)
以给定的回调函数开始句柄。

int **uv_prepare_stop**(*uv_prepare_t* prepare*)
停止句柄，回调函数将不会再被调用。

参见:

uv_handle_t 的API函数也适用。

3.2.8 uv_check_t — 检查句柄

检查句柄将在每次循环迭代时运行给定的回调函数一次， 在I/O轮询后一刻。

数据类型

uv_check_t
检查句柄类型。

void **(*uv_check_cb)**(*uv_check_t* handle*)
传递给 *uv_check_start()* 的回调函数的类型定义。

公共成员

N/A

参见:

uv_handle_t 的成员也适用。

API

int **uv_check_init**(*uv_loop_t* loop*, *uv_check_t* check*)
初始化句柄。

int **uv_check_start**(*uv_check_t* check*, *uv_check_cb cb*)
以给定的回调函数开始句柄。

int **uv_check_stop**(*uv_check_t* check*)
停止句柄，回调函数将不会再被调用。

参见:

uv_handle_t 的API函数也适用。

3.2.9 uv_idle_t — 空转句柄

空转句柄将在每次循环迭代时运行给定的回调函数一次， 在 *uv_prepare_t* 句柄前一刻。

注解: 与准备句柄的显著差异在于当有活动空转句柄时， 循环将以零时限轮询而不是为I/O阻塞。

警告: 虽有不恰当的名字，空转句柄的回调函数在每次循环迭代时都会被调用， 而不仅仅在循环确实"空转的" 时候。

数据类型

uv_idle_t
空转句柄类型。

void **(*uv_idle_cb)**(*uv_idle_t* handle*)
传递给 *uv_idle_start()* 的回调函数的类型定义。

公共成员

N/A

参见:

uv_handle_t 的成员也适用。

API

int **uv_idle_init**(*uv_loop_t* loop*, *uv_idle_t* idle*)
初始化句柄。

int **uv_idle_start**(*uv_idle_t* idle*, *uv_idle_cb cb*)
以给定的回调函数开始句柄。

int **uv_idle_stop**(*uv_idle_t* idle*)
停止句柄，回调函数将不会再被调用。

参见:

uv_handle_t 的API函数也适用。

3.2.10 uv_async_t — 异步句柄

异步句柄允许用户 "唤醒" 事件循环并且从另一个线程调用回调函数。

数据类型

uv_async_t
异步句柄类型。

void **(*uv_async_cb)**(*uv_async_t* handle*)
传递给 *uv_async_init()* 的回调函数的类型定义。

公共成员

N/A

参见:

uv_handle_t 的成员也适用。

API

int **uv_async_init**(*uv_loop_t* loop*, *uv_async_t* async*, *uv_async_cb* async_cb*)
初始化句柄。 允许回调函数为NULL。

返回 0 当成功时，或者一个 < 0 的错误代码当失败时。

注解: 不同于其他句柄初始化函数，句柄立刻开始。

int **uv_async_send**(*uv_async_t* async*)
唤醒事件循环并且调用异步句柄的回调函数。

返回 0 当成功时，或者一个 < 0 的错误代码当失败时。

注解: 从任何线程调用这个函数都是安全的。 回调函数将从循环的线程上被调用。

警告: libuv将会合并对 *uv_async_send()* 的调用，那就是说，不是对它的每个调用会 yield 回调函数的执行。 例如：如果在回调函数被调用前一连调用 *uv_async_send()* 5 次，回调函数将只会调用一次。 如果在回调函数被调用后再次调用 *uv_async_send()* ，回调函数将会再次被调用。

参见:

uv_handle_t 的API函数也适用。

3.2.11 uv_poll_t — 轮询句柄

轮询句柄用于监视文件描述符的可读性、可写性和连接断开， 和 poll(2) 的目的类似。

轮询句柄的目的是允许集成的依赖于事件循环的外部库可以发出关于套接字状态变化的信号， 像 c-ares 或是 libssh2。 于任何其他目的使用 uv_poll_t 不被推荐； *uv_tcp_t* 、 *uv_udp_t* 等等提供了更快的和比起 *uv_poll_t* 更具伸缩性的实现， 尤其是在Windows上。

轮询句柄偶尔可能会发出文件描述符可读或可写的信号即使并不是的时候。 用户因此应该总是准备好处理 EAGAIN 或者同类错误，当尝试从文件描述符读取或写入的时候。

对同个套接字有多个活动的轮询句柄是不可以的， 这会导致libuv死循环或者别的故障。

用户不应该关闭文件描述符，当其被一个活动的轮询句柄轮询的时候。 这可能导致句柄报错，但是也可能开始轮询另外一个套接字。 然而在调用 *uv_poll_stop()* 或 *uv_close()* 之后，文件描述符立即能被安全地关闭。

注解： 在Windows上，只有套接字能被轮询句柄轮询。 在Unix上，任何被 poll(2) 接受的文件描述符都可以用。

注解： 在AIX上，不支持监视连接断开。

数据类型

uv_poll_t
轮询句柄类型。

void **(*uv_poll_cb)**(*uv_poll_t** *handle*, int *status*, int *events*)
传递给 *uv_poll_start()* 的回调函数的类型定义。

uv_poll_event
轮询事件类型

```
enum uv_poll_event {
    UV_READABLE = 1,
    UV_WRITABLE = 2,
    UV_DISCONNECT = 4,
    UV_PRIORITIZED = 8
};
```

公共成员

N/A

参见：

uv_handle_t 的成员也适用。

API

int **uv_poll_init**(*uv_loop_t** *loop*, *uv_poll_t** *handle*, int *fd*)
使用一个文件描述符初始化句柄。

在 1.2.2 版更改: 文件描述符设为非阻塞模式。

int **uv_poll_init_socket**(*uv_loop_t* loop*, *uv_poll_t* handle*, *uv_os_sock_t socket*)
使用一个套接字描述符初始化句柄。 在Linux上这与 `uv_poll_init()` 一样。在Windows上它使用一个SOCKET句柄。

在 1.2.2 版更改: 套接字设为非阻塞模式。

int **uv_poll_start**(*uv_poll_t* handle*, int *events*, *uv_poll_cb cb*)
开始轮询文件描述符。 *events* 是一个由 UV_READABLE、UV_WRITABLE、UV_PRIORITIZED和UV_DISCONNECT组成的位掩码。 当事件一被检测到，回调函数将以 *status* 为 0 被调用， 并且检测的事件设置于 *events* 字段。

UV_PRIORITIZED 事件用来监视sysfs中断或是TCP带外信息。

UV_DISCONNECT 事件就某种意义而言也许不该被报告并且用户可以自由地忽略它， 但是它可以帮助优化停机过程，因为可能避免额外的读或写调用。

如果当轮询中发生了错误， *status* 将 < 0 并且对应于一种 UV_E* 错误代码（详见 错误处理 ）。 用户不应该当句柄活动时关闭套接字。 如果用户无论如何关闭了套接字，回调函数 也许 会被调用以报告错误状态，但这 **无法** 保证。

注解: 在已经活动的句柄上调用 `uv_poll_start()` 是可以的。 这么做将更新正监视中的事件掩码。

注解: 虽然可以设置UV_DISCONNECT，它不支持AIX并且在这种情况下不会出现于回调函数的 *events* 字段。

在 1.9.0 版更改: 新增 UV_DISCONNECT 事件。

在 1.14.0 版更改: 新增 UV_PRIORITIZED 事件。

int **uv_poll_stop**(*uv_poll_t* poll*)
停止轮询文件描述符，回调函数将不会再被调用。

参见:

`uv_handle_t` 的API函数也适用。

3.2.12 uv_signal_t — 信号句柄

信号句柄在按事件循环的基础上实现了Unix风格的信号处理。

在Windows上模拟了一些信号的接收:

- SIGINT 通常在用户按 CTRL+C 时被发送。 然而，如同在Unix上，当终端原始模式开启时这无法保证。
- SIGBREAK 在用户按 CTRL + BREAK 时被发送。
- SIGHUP 在用户关闭终端窗口时被生成。 在SIGHUP时给予程序大约10秒执行清理。 在那之后Windows将会无条件地终止程序。
- 对其他类型的信号的监视器能被成功地创建，但是这些信号永远不会被接收到。 这些信号是： *SIGILL* 、 *SIGABRT* 、 *SIGFPE* 、 *SIGSEGV* 、 *SIGTERM* 和 *SIGKILL* 。
- 通过编程方式调用 raise() 或 abort() 发出一个信号不会被libuv检测到； 这些信号将不会触发信号监视器。

注解: 在Linux上 SIGRT0 和 SIGRT1 （信号32和33）用于NPTL线程库来管理线程。 对这些信号安装监视器将会导致无法预测的行为并且强烈不推荐。 libuv未来的版本中可能会直接拒绝它们。

在 1.15.0 版更改: 在Windows上改进对 SIGWINCH 的支持。

数据类型

uv_signal_t
信号句柄类型。

void **(*uv_signal_cb)**(*uv_signal_t* handle*, int *signum*)
传递给 `uv_signal_start()` 的回调函数的类型定义。

公共成员

int **uv_signal_t.signum**
被这个句柄监视的信号。 只读。

参见:

`uv_handle_t` 的成员也适用。

API

int **uv_signal_init**(*uv_loop_t* loop*, *uv_signal_t* signal*)
初始化句柄。

int **uv_signal_start**(*uv_signal_t* signal*, *uv_signal_cb cb*, int *signum*)
以给定的回调函数开始句柄，监视给定的信号。

int **uv_signal_start_oneshot**(*uv_signal_t* signal*, *uv_signal_cb cb*, int *signum*)
1.12.0 新版功能.

与 `uv_signal_start()` 同样的功能，但是信号句柄在接受到信号的时刻被重置。

int **uv_signal_stop**(*uv_signal_t* signal*)
停止句柄，回调函数将不会再被调用。

参见:

`uv_handle_t` 的API函数也适用。

3.2.13 uv_process_t — 进程句柄

进程句柄将会生成新进程，并且允许用户控制它和使用流与它建立通信通道。

数据类型

uv_process_t
进程句柄类型。

uv_process_options_t
生成进程的选项（传递给 `uv_spawn()` 的）

```
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**, int64_t *exit_status*, int *term_signal*)
传递给 *uv_process_options_t* 的回调函数的类型定义， 将指明退出状态和引发进程终止的信号，如果有的话。

uv_process_flags
设置在 *uv_process_options_t* 标志字段的标志。

```
enum uv_process_flags {
    /*
    * 设置子进程的用户ID。
    */
    UV_PROCESS_SETUID = (1 << 0),
    /*
    * 设置子进程的组ID。
    */
    UV_PROCESS_SETGID = (1 << 1),
    /*
    * 当将参数列表转换为一个命令行字符串时，不要用引号包围参数，抑或是任何其它转义。
    * 这个选项只在Windows系统上有效，在Unix上被忽略。
    */
    UV_PROCESS_WINDOWS_VERBATIM_ARGUMENTS = (1 << 2),
    /*
    * 以分离状态生成子进程----这将使得它成为进程组的组长，
    * 将有效地允许子进程在父进程退出之后继续运行。
    * 注意子进程将保持父进程的事件循环处于活动状态，
    * 除非父进程在子进程句柄上调用 uv_unref() 。
    */
    UV_PROCESS_DETACHED = (1 << 3),
    /*
    * 隐藏子进程默认被创建的窗口。
    * 这个选项只在Windows系统上有效，在Unix上被忽略。
    */
    UV_PROCESS_WINDOWS_HIDE = (1 << 4),
    /*
    * 隐藏子进程默认被创建的终端窗口。
    * 这个选项只在Windows系统上有效，在Unix上被忽略。
    */
    UV_PROCESS_WINDOWS_HIDE_CONSOLE = (1 << 5),
    /*
    * 隐藏子进程默认被创建的GUI窗口。
    * 这个选项只在Windows系统上有效，在Unix上被忽略。
    */
    UV_PROCESS_WINDOWS_HIDE_GUI = (1 << 6)
};
```

uv_stdio_container_t
传递给子进程的每个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如何被传送到子进程的标志。

```
typedef enum {
    UV_IGNORE = 0x00,
    UV_CREATE_PIPE = 0x01,
    UV_INHERIT_FD = 0x02,
    UV_INHERIT_STREAM = 0x04,
    /*
    * 当指定UV_CREATE_PIPE时，UV_READABLE_PIPE和UV_WRITABLE_PIPE决定了从子进程视角的数据
流方向，
    * 可以两个被同时指定，以创建双工数据流。
    */
    UV_READABLE_PIPE = 0x10,
    UV_WRITABLE_PIPE = 0x20
    /*
     * 在Windows上以重叠模式打开子进程管道句柄。
     * 在Unix上被忽略。
     */
    UV_OVERLAPPED_PIPE = 0x40
} uv_stdio_flags;
```

公共成员

uv_process_t.pid
生成的进程的PID。 在 *uv_spawn()* 调用后被设置。

注解: *uv_handle_t* 的成员也适用。

---

uv_process_options_t.exit_cb
进程退出时调用的回调函数。

uv_process_options_t.file
指向被执行程序的路径。

uv_process_options_t.args
命令行参数。 args[0] 应该是程序路径。 在Windows上这使用了 *CreateProcess* 连接参数为一个字符串，可能导致一些奇怪的错误。 详见 *uv_process_flags* 上的 UV_PROCESS_WINDOWS_VERBATIM_ARGUMENTS 标志。

uv_process_options_t.env
新进程的环境。 如果为空使用父进程的环境。

uv_process_options_t.cwd
子进程的当前工作目录。

uv_process_options_t.flags
控制 *uv_spawn()* 行为的各种标志。 详见 *uv_process_flags* 。

uv_process_options_t.stdio_count

uv_process_options_t.stdio
指向 *uv_stdio_container_t* 结构体数组的 *stdio* 指针字段，描述了将对子进程生效的文件描述符。 惯例是文件描述符0（stdio[0]指向的）用于stdin，文件描述符1用于stdout，文件描述符2用于stderr。

> **注解:** 在Windows上文件描述符大于2只对使用MSVCRT运行时的子进程有效。

uv_process_options_t.uid

uv_process_options_t.gid
libuv能够改变子进程的用户/组ID。 这只发生于在标志字段设置恰当的比特时。

> **注解:** 在Windows上这不被支持， *uv_spawn()* 将会失败并且设置错误为 `UV_ENOTSUP` 。

uv_stdio_container_t.flags
标志位指明stdio容器应该怎样被传递给子进程。 详见 *uv_stdio_flags* 。

uv_stdio_container_t.data
包括传递给子进程的流或文件描述符的共同体。

API

void **uv_disable_stdio_inheritance**(void)
禁用继承来自父进程的文件描述符/句柄。 效果是此进程生成的子进程不会意外地继承这些句柄。

在继承的文件描述符能被关闭和复制前，推荐在你的程序中尽早调用这个函数。

> **注解:** 这个函数工作在尽力而为的基础上：不保证libuv能够发现所有继承的文件描述符。 通常这在Windows上比在Unix上工作地更好。

int **uv_spawn**(*uv_loop_t* loop*, *uv_process_t* handle*, const *uv_process_options_t* options*)
初始化进程描述符并且启动进程。 如果进程成功生成，这个函数返回0。 否则，返回对应于无法生成原因的负数错误代码。

生成失败的可能原因包括（但不限于） 要执行的文件不存在、没权限使用指定的setuid或setgid 或者没有足够的内存分配给新进程。

在 1.24.0 版更改: 新增 *UV_PROCESS_WINDOWS_HIDE_CONSOLE* 和 *UV_PROCESS_WINDOWS_HIDE_GUI* 标志。

int **uv_process_kill**(*uv_process_t* handle*, int *signum*)
发送指定的信号到给定的进程句柄。 检查 *uv_signal_t — 信号句柄* 上的文档对于信号的支持，特别是在Windows上。

int **uv_kill**(int *pid*, int *signum*)
发送指定的信号到给定的PID。 检查 *uv_signal_t — 信号句柄* 上的文档对于信号的支持，特别是在Windows上。

uv_pid_t **uv_process_get_pid**(const *uv_process_t* handle*)
返回 *handle->pid* 。

1.19.0 新版功能.

参见:

uv_handle_t 的API函数也适用。

3.2.14 uv_stream_t — 流句柄

流句柄提供对双工通信通道的一种抽象。 *uv_stream_t* 是一个抽象类型，libuv提供了3种流的实现以 *uv_tcp_t* 、 *uv_pipe_t* 和 *uv_tty_t* 的形式。

数据类型

uv_stream_t
流句柄类型。

uv_connect_t
连接请求类型。

uv_shutdown_t
停机请求类型。

uv_write_t
写请求类型。 当复用这种对象时必须小心注意。 当一个流处在非阻塞模式，用 `uv_write` 发送的写请求将被排队。 在此刻复用对象是未定义的行为。 仅在传递给 `uv_write` 的回调函数执行完毕后才能安全地复用 `uv_write_t` 对象。

void **(*uv_read_cb)**(*uv_stream_t** *stream*, ssize_t *nread*, const *uv_buf_t** *buf*)
数据在流上读取时的回调函数。

nread 是 > 0 如果有可用的数据，或 < 0 当错误时。 当我们已经到达EOF， *nread* 将被设置为 `UV_EOF` 。 当 *nread* < 0 时 *buf* 参数可能并不指向一个合法的缓冲区； 在那种情况下 *buf.len* 和 *buf.base* 都被设为0。

注解: *nread* 可能是0，并不预示着一个错误或EOF。 这等同于在 `read(2)` 下的 `EAGAIN` 或 `EWOULDBLOCK` 。

调用者负责在错误发生时通过调用 *uv_read_stop()* 或 *uv_close()* 停止/关闭流。 尝试从流再次读取是未定义的。

调用者负责释放缓冲区，libuv不会复用它。 在错误时缓冲区可能是一个空缓冲区（这里 buf->base=NULL 且 buf->len=0）。

void **(*uv_write_cb)**(*uv_write_t** *req*, int *status*)
数据已经在流上写后的回调函数。 *status* 若成功将是0，否则 < 0。

void **(*uv_connect_cb)**(*uv_connect_t** *req*, int *status*)
以 `uv_connect()` 开启连接完成后的回调函数。 *status* 若成功将是0，否则 < 0。

void **(*uv_shutdown_cb)**(*uv_shutdown_t** *req*, int *status*)
停机请求完成后的回调函数。 *status* 若成功将是0，否则 < 0。

void **(*uv_connection_cb)**(*uv_stream_t** *server*, int *status*)
当流服务器接收到新来的连接时的回调函数。 用户能够通过调用 *uv_accept()* 来接受连接。 *status* 若成功将是0，否则 < 0。

公共成员

size_t **uv_stream_t.write_queue_size**
包含等待发送的排队字节的数量。 只读。

*uv_stream_t** **uv_connect_t.handle**
指向此连接请求所运行于的流的指针。

*uv_stream_t** **uv_shutdown_t.handle**
指向此停机请求所运行于的流的指针。

*uv_stream_t** **uv_write_t.handle**
指向此写请求所运行于的流的指针。

*uv_stream_t** **uv_write_t.send_handle**
指向使用此连接请求被发送的的流的指针。

参见:

uv_handle_t 的成员也适用。

API

int **uv_shutdown**(*uv_shutdown_t** *req*, *uv_stream_t** *handle*, *uv_shutdown_cb cb*)
停机双工流的向外（写）端。 它等待未处理的写请求完成。 *handle* 应该指向已初始化的流。 *req* 应该是一个未初始化的停机请求结构体。 *cb* 在停机完成后被调用。

int **uv_listen**(*uv_stream_t** *stream*, int *backlog*, *uv_connection_cb cb*)
开始侦听新来的连接。 *backlog* 指内核可能排队的连接数，与 *:man:'listen(2)* 相同。 当接受到新来的连接时，调用 *uv_connection_cb* 回调函数。

int **uv_accept**(*uv_stream_t** *server*, *uv_stream_t** *client*)
调用用来配合 *uv_listen()* 接受新来的连接。 在接收到 *uv_connection_cb* 后调用这个函数以接受连接。 在调用这个函数前，客户端句柄必须被初始化。 < 0 返回值表示错误。

当 *uv_connection_cb* 回调函数被调用时，保证这个函数将会成功第一次。 如果你尝试使用超过一次，它可能失败。 建议每个 *uv_connection_cb* 调用只调用这个函数一次。

注解: *server* 和 *client* 必须是运行在同一个循环之上的句柄。、

---

int **uv_read_start**(*uv_stream_t** *stream*, *uv_alloc_cb alloc_cb*, *uv_read_cb read_cb*)
从内向的流读取数据。 将会调用 *uv_read_cb* 回调函数几次直到没有更多数据可读或是调用了 *uv_read_stop()* 。

int **uv_read_stop**(*uv_stream_t**)
停止从流读取数据。 *uv_read_cb* 回调函数将不再被调用。

这个函数是幂等的且可以在已停止的流上安全地被调用。

int **uv_write**(*uv_write_t** *req*, *uv_stream_t** *handle*, const *uv_buf_t bufs[]*, unsigned int *nbufs*, *uv_write_cb cb*)
写数据到流。 缓冲区按序写入。例如:

```
void cb(uv_write_t* req, int status) {
    /* 处理写结果的逻辑 */
}

uv_buf_t a[] = {
```

(continues on next page)

(续上页)

```
    { .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;

/* 写 "1234" */
uv_write(&req1, stream, a, 2, cb);
uv_write(&req2, stream, b, 2, cb);
```

注解: 被缓冲区指向的内存必须保持有效直到回调函数执行完。 这也适用于 *uv_write2()* 。

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*)
扩展的写函数用于在管道上发送句柄。 管道必须以 *ipc* == 1 初始化。

注解: *send_handle* 必须是一个TCP套接字或者管道，且为一个服务器或一个连接（侦听或已连接状态）。 绑定的套接字或管道将被假设是服务器。

int **uv_try_write**(*uv_stream_t** *handle*, const *uv_buf_t* *bufs[]*, unsigned int *nbufs*)
与 *uv_write()* 相同，但是如果无法立刻完成时不会排队写请求。

将返回以下之一：

- > 0: 已写字节数（可能小于提供的缓冲区大小）。
- < 0: 负的错误代码（返回 UV_EAGAIN 如果没有数据能立刻发送）。

int **uv_is_readable**(const *uv_stream_t** *handle*)
如果流可读返回1，否则0。

int **uv_is_writable**(const *uv_stream_t** *handle*)
如果流可写返回1，否则0。

int **uv_stream_set_blocking**(*uv_stream_t** *handle*, int *blocking*)
启用或禁用流的阻塞模式。

当阻塞模式开启时所有的写都是同步完成的。 别的界面保持不变，比如操作完成或失败将仍然通过回调函数异步被报告。

警告: 太依赖于这个API是不推荐的。 它可能在未来明显地变化。

当前在Windows上只工作于 *uv_pipe_t* 句柄。 在UNIX平台，所有的 *uv_stream_t* 句柄都被支持。

另外当前libuv当阻塞模式在已经提交写请求之后改变时没有作顺序保证。 因此推荐在打开或创建流之后立即设置阻塞模式。

在 1.4.0 版更改: 新增UNIX实现。

size_t **uv_stream_get_write_queue_size**(const *uv_stream_t** *stream*)
返回 *stream->write_queue_size* 。

1.19.0 新版功能.

参见:

uv_handle_t 的API函数也适用。

3.2.15 uv_tcp_t — TCP句柄

TCP句柄用于表示TCP流和服务器。

uv_tcp_t 是 *uv_stream_t* 的一个 '子类型' 。

数据类型

uv_tcp_t
TCP句柄类型。

公共成员

N/A

参见:

uv_stream_t 的成员也适用。

API

int **uv_tcp_init**(*uv_loop_t** *loop*, *uv_tcp_t** *handle*)
初始化句柄。 迄今为止没有创建套接字。

int **uv_tcp_init_ex**(*uv_loop_t** *loop*, *uv_tcp_t** *handle*, unsigned int *flags*)
以指定的标志来初始化句柄。 在此刻只有 *flags* 参数的低8位用于套接字域。一个套接字将为给定的域创建。 如果指定的域是 AF_UNSPEC 则没有套接字被创建，就像 *uv_tcp_init()* 一样。

1.7.0 新版功能.

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套接字作为一个TCP句柄。

在 1.2.1 版更改: 文件描述符设为非阻塞模式。

注解: 被传递的文件描述符或套接字没有类型检查，但是需要它代表一个合法的流套接字。

---

int **uv_tcp_nodelay**(*uv_tcp_t** *handle*, int *enable*)
开启 *TCP_NODELAY*，禁用Nagle算法。

int **uv_tcp_keepalive**(*uv_tcp_t** *handle*, int *enable*, unsigned int *delay*)
开启/禁用TCP keep-alive。 *delay* 是以秒计的起始延迟，当 *enable* 是零时被忽略。

int **uv_tcp_simultaneous_accepts**(*uv_tcp_t** *handle*, int *enable*)
开启/禁用同时异步的接受请求被操作系统排队当侦听新来的TCP连接时。

这个设置用来调整TCP服务器达到满意的性能。 拥有同时接受能显著地提高接受连接的速率（这就是为什么它默认开启）， 但是可能导致在多进程设置下不均衡的负载。

int **uv_tcp_bind**(*uv_tcp_t** *handle*, const struct sockaddr* *addr*, unsigned int *flags*)
绑定句柄到一个地址和端口。 *addr* 应该指向一个初始化了的 `struct sockaddr_in` 或者 `struct sockaddr_in6` 。

当端口已经被占用时，你会预期见到一个 `UV_EADDRINUSE` 错误来自于 *uv_tcp_bind()* 、*uv_listen()* 或 *uv_tcp_connect()* 之一。 那就是说，一个对此函数成功的调用并不保证对 *uv_listen()* 或 *uv_tcp_connect()* 的调用也会成功。

flags 能包括 `UV_TCP_IPV6ONLY` ，在这种情况下禁用双栈支持且只使用IPv6。

int **uv_tcp_getsockname**(const *uv_tcp_t** *handle*, struct sockaddr* *name*, int* *namelen*)
获取句柄当前绑定的地址。 *name* 必须指向一个合法且足够大的内存块， 推荐使用 `struct sockaddr_storage` 获得IPv4和IPv6支持。

int **uv_tcp_getpeername**(const *uv_tcp_t** *handle*, struct sockaddr* *name*, int* *namelen*)
连接到句柄的远端的地址。 *name* 必须指向一个合法且足够大的内存块， 推荐使用 `struct sockaddr_storage` 获得IPv4和IPv6支持。

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IPv6的TCP连接。 提供一个已初始化的TCP句柄和一个未初始化的 *uv_connect_t* 。 *addr* 应该指向一个已初始化的 `struct sockaddr_in` 或 `struct sockaddr_in6` 。

在Windows上如果 *addr* 被初始化指向一个未指定的地址 （`0.0.0.0` 或者 `::` ），它将被改变以指向 `localhost` 。 这么做是为了符合Linux系统的行为。

这个回调函数当连接已建立或当连接发生错误时被调用。

在 1.19.0 版更改: 新增 `0.0.0.0` 和 `::` 到 `localhost` 的映射

参见:

uv_stream_t 的API函数也适用。

3.2.16 uv_pipe_t — 管道句柄

管道句柄对Unix上的本地域套接字和Windows上的有名管道提供一个抽象。

uv_pipe_t 是 *uv_stream_t* 的一个'子类型'。

数据类型

uv_pipe_t
管道句柄类型。

公共成员

int **uv_pipe_t.ipc**
是否这个管道适合在不同进程间传递句柄。

参见:

uv_stream_t 的成员也适用。

API

int **uv_pipe_init**(*uv_loop_t* loop*, *uv_pipe_t* handle*, int *ipc*)
初始化一个管道句柄。 *ipc* 参数是一个布尔值指明是否管道将用于在不同进程间传递句柄。

int **uv_pipe_open**(*uv_pipe_t* handle*, *uv_file file*)
打开一个已存在的文件描述符或者句柄作为一个管道。

在 1.2.1 版更改: 文件描述符设为非阻塞模式。

注解: 被传递的文件描述符或句柄没有类型检查，但是需要它代表一个合法的管道。

---

int **uv_pipe_bind**(*uv_pipe_t* handle*, const char* *name*)
绑定管道到一个文件路径（Unix）或者名字（Windows）。

注解: Unix上的路径被截取到 `sizeof(sockaddr_un.sun_path)` 字节，通常在 92 到 108 字节之间。

---

void **uv_pipe_connect**(*uv_connect_t* req*, *uv_pipe_t* handle*, const char* *name*, *uv_connect_cb cb*)
连接到Unix域套接字或者有名管道。

注解: Unix上的路径被截取到 `sizeof(sockaddr_un.sun_path)` 字节，通常在 92 到 108 字节之间。

---

int **uv_pipe_getsockname**(const *uv_pipe_t* handle*, char* *buffer*, size_t* *size*)
获取Unix域套接字或者有名管道的名字。

必须提供一个预分配的缓冲区。 size参数存有缓冲区的大小并设为在输出上写到缓冲区的字节数。 如果缓冲区不够大，将返回 `UV_ENOBUFS` 并且这个参数将包含需要的大小。

在 1.3.0 版更改: 返回的长度不再包括终止的空字节，且缓冲区不以空字节终止。

int **uv_pipe_getpeername**(const *uv_pipe_t* handle*, char* *buffer*, size_t* *size*)
获取被句柄连接的Unix域套接字或者有名管道的名字。

必须提供一个预分配的缓冲区。 size参数存有缓冲区的大小并设为在输出上写到缓冲区的字节数。 如果缓冲区不够大，将返回 `UV_ENOBUFS` 并且这个参数将包含需要的大小。

1.3.0 新版功能.

void **uv_pipe_pending_instances**(*uv_pipe_t* handle*, int *count*)
设置当管道服务器等待连接时未处理的管道实例句柄的数目。

注解: 这个设置只应用于Windows。

---

int **uv_pipe_pending_count**(*uv_pipe_t* handle*)

uv_handle_type **uv_pipe_pending_type**(*uv_pipe_t* handle*)
用来通过IPC管道接收句柄。

首先——调用 *`uv_pipe_pending_count()`*， 如果 > 0 则初始化给定 *type* 的一个句柄， 通过 *`uv_pipe_pending_type()`* 返回再调用 `uv_accept(pipe, handle)`。

参见:

`uv_stream_t` 的API函数也适用。

int **uv_pipe_chmod**(*uv_pipe_t* handle*, int *flags*)
修改管道的权限，允许它被不同用户运行的进程访问。 使得管道被所有用户可写和可读。 模式可以是 `UV_WRITABLE` 、 `UV_READABLE` 或 `UV_WRITABLE | UV_READABLE` 。 这个函数是阻塞的。

1.16.0 新版功能.

3.2.17 uv_tty_t — TTY句柄

TTY句柄代表对终端的一个流。

uv_tty_t 是 *uv_stream_t* 的一个 '子类型' 。

数据类型

uv_tty_t
TTY句柄类型。

uv_tty_mode_t
1.2.0 新版功能.

TTY模式类型:

```
typedef enum {
    /* 初始/正常终端模式 */
    UV_TTY_MODE_NORMAL,
    /* 原始输入模式（在Windows上，也启用了ENABLE_WINDOW_INPUT） */
    UV_TTY_MODE_RAW,
    /* 用于IPC的二进制安全的I/O模式（仅Unix） */
    UV_TTY_MODE_IO
} uv_tty_mode_t;
```

公共成员

N/A

参见:

uv_stream_t 的成员也适用。

API

int **uv_tty_init**(*uv_loop_t* loop*, *uv_tty_t* handle*, *uv_file fd*, int *unused*)
以给定的文件描述符初始化一个新的TTY流。 通常文件描述符是:

- 0 = stdin
- 1 = stdout
- 2 = stderr

在Unix上这个函数将会使用 ttyname_r(3) 决定终端文件描述符的路径， 打开它，如果被传递的文件描述符指向一个TTY再使用它。 这允许libuv将TTY放入非阻塞模式而不影响共享这个TTY的其他进程。

这个函数在不支持ioctl的TIOCGPTN或TIOCPTYGNAME的系统上不是线程安全的， 例如OpenBSD和Solaris。

注解: 如果重新打开TTY失败，libuv回退到阻塞写。

在 1.23.1: 版更改: *readable* 参数现在没用且被忽略。 正确的值现在将由内核自动检测。

在 1.9.0: 版更改: TTY的路径由 ttyname_r(3) 决定。而在之前的版本中libuv打开 */dev/tty*。

在 1.5.0: 版更改: 尝试以一个指向一个文件的文件描述符初始化一个TTY流在UNIX上返回 *UV_EINVAL* 。

int **uv_tty_set_mode**(*uv_tty_t* handle*, *uv_tty_mode_t mode*)
在 1.2.0: 版更改: 模式由 *uv_tty_mode_t* 值指定。

使用指定的终端模式设置TTY。

int **uv_tty_reset_mode**(void)
当程序退出时将被调用。 重设TTY设置到默认值以便被接下来的进程接管。

这个函数在Unix平台上是异步线程安全的，但是可能以错误代码 UV_EBUSY 而失败， 如果你当执行于 *uv_tty_set_mode()* 中间调用它的时候。

int **uv_tty_get_winsize**(*uv_tty_t* handle*, int* *width*, int* *height*)
获取当前的窗口大小。 成功时返回0。

参见:

uv_stream_t 的API函数也适用。

3.2.18 uv_udp_t — UDP句柄

UDP句柄封装了对客户端和服务端的UDP通信。

数据类型

uv_udp_t
UDP句柄类型。

uv_udp_send_t
UDP发送请求类型。

uv_udp_flags
用在 *uv_udp_bind()* 和 *uv_udp_recv_cb* 的标志。

```
enum uv_udp_flags {
    /* 禁用双栈模式。 */
    UV_UDP_IPV6ONLY = 1,
    /*
    * 指明消息被截断由于读缓冲区太小。剩余部分被操作系统丢弃。
    * 用于 uv_udp_recv_cb 。
    */
    UV_UDP_PARTIAL = 2,
    /*
    * 指明当以uv_udp_bind绑定句柄时是否 SO_REUSEADDR 将被设置。
    * 在BSDs和OS X上这设置SO_REUSEPORT套接字标志。
    * 在其他Unix平台上，它设置SO_REUSEADDR标志。
    * 这意味着多个线程或进程能够无错误地绑定到同一个地址（假如它们都设置了此标志）
    * 但是只有最后一个绑定的将会接收到任何流量，以从之前侦听器"偷走"端口的效果。
    */
```

(continues on next page)

(续上页)

```
    UV_UDP_REUSEADDR = 4
};
```

void **(*uv_udp_send_cb)**(*uv_udp_send_t** *req*, int *status*)
传递给 *uv_udp_send()* 的回调函数的类型定义，在数据发送之后被调用。

void **(*uv_udp_recv_cb)**(*uv_udp_t** *handle*, ssize_t *nread*, const *uv_buf_t** *buf*, const struct sockaddr* *addr*, unsigned *flags*)
传递给 *uv_udp_recv_start()* 的回调函数的类型定义，在终点接收了数据时被调用。

- *handle*： UDP句柄。
- *nread*： 已接收字节数。0 如果没有更多数据可读。你可以丢弃读缓冲区或赋予其新的用途。注意 0 也意味着收到了空的数据报（在这种情况下 *addr* 是非空的）。< 0 如果检测到了一个传输错误。
- *buf*： *uv_buf_t* 带有接收到的数据。
- *addr*： `struct sockaddr*` 包含发送者地址。可能为 NULL。只在回调函数的期间内有效。
- *flags*: 一个或更多被或的UV_UDP_* 常量。当前仅 `UV_UDP_PARTIAL` 被使用。

注解： 接收回调函数将被以 *nread* == 0 和 *addr* == NULL 调用当没有数据可读，且以 *nread* == 0 和 *addr* != NULL 当收到了一个空的UDP数据报。

---

uv_membership
对多播地址的成员类型。

```
typedef enum {
    UV_LEAVE_GROUP = 0,
    UV_JOIN_GROUP
} uv_membership;
```

公共成员

size_t **uv_udp_t.send_queue_size**
排队发送的字节数目。这个字段严格显示当前排队的消息量。

size_t **uv_udp_t.send_queue_count**
当前排队等待处理的发送请求的数目。

*uv_udp_t** **uv_udp_send_t.handle**
此发送请求所发生在的UDP句柄。

参见:

uv_handle_t 的成员也适用。

API

int **uv_udp_init**(*uv_loop_t** *loop*, *uv_udp_t** *handle*)
初始化一个新的UDP句柄。实际的套接字是惰性创建的。返回 0 当成功时。

int **uv_udp_init_ex**(*uv_loop_t** *loop*, *uv_udp_t** *handle*, unsigned int *flags*)
以指定的标志初始化句柄。在此刻只有 *flags* 参数的低8位用于套接字域。一个套接字将为给定的域创建。如果指定的域是 `AF_UNSPEC` 则没有套接字被创建，就像 *uv_udp_init()* 一样。

1.7.0 新版功能.

int **uv_udp_open**(*uv_udp_t* handle*, *uv_os_sock_t sock*)
打开一个已存在的文件描述符或者Windows套接字作为一个UDP句柄。

仅对Unix： *sock* 参数的唯一需求是遵循数据报合约（工作在无连接模式、支持sendmsg()/recvmsg()、等等）。 换句话说，其他数据报类型的套接字像原始套接字或者Netlink套接字也能被传递给这个函数。

在 1.2.1 版更改: 文件描述符设为非阻塞模式。

注解： 被传递的文件描述符或套接字没有类型检查，但是需要它代表一个合法的数据报套接字。

---

int **uv_udp_bind**(*uv_udp_t* handle*, const struct sockaddr* *addr*, unsigned int *flags*)
绑定UDP句柄到一个IP地址和端口。

参数

- **handle** – UDP句柄。 应该以 `uv_udp_init()` 被初始化。
- **addr** – 带地址和端口绑定的 *struct sockaddr_in* 或 *struct sockaddr_in6* 。
- **flags** – 指明套接字将被如何绑定， `UV_UDP_IPV6ONLY` 和 `UV_UDP_REUSEADDR` 被支持。

返回 0 若成功，或一个 < 0 的错误代码若失败。

int **uv_udp_getsockname**(const *uv_udp_t* handle*, struct sockaddr* *name*, int* *namelen*)
获取UDP句柄的本地的IP和端口。

参数

- **handle** – UDP句柄。 应该以 `uv_udp_init()` 被初始化并且被绑定。
- **name** – 被地址数据填充的结构体的指针。 为了支持IPv4和IPv6 *struct sockaddr_storage* 应被使用。
- **namelen** – 在输入上它指明 *name* 字段的数据。 在输出上它指明它被填充了多少。

返回 0 若成功，或一个 < 0 的错误代码若失败。

int **uv_udp_set_membership**(*uv_udp_t* handle*, const char* *multicast_addr*, const char* *interface_addr*, *uv_membership membership*)
对一个多播地址设置成员。

参数

- **handle** – UDP句柄。 应该以 `uv_udp_init()` 被初始化。
- **multicast_addr** – 设置成员的多播地址。
- **interface_addr** – 接口地址。
- **membership** – 应该为 `UV_JOIN_GROUP` 或 `UV_LEAVE_GROUP` 。

返回 0 若成功，或一个 < 0 的错误代码若失败。

int **uv_udp_set_multicast_loop**(*uv_udp_t* handle*, int *on*)
设置IP多播循环标志。 使得多播包循环回本地套接字。

参数

- **handle** – UDP句柄。 应该以 `uv_udp_init()` 被初始化。
- **on** – 1 为开，0 为关。

返回 0 若成功，或一个 < 0 的错误代码若失败。

int **uv_udp_set_multicast_ttl**(*uv_udp_t* handle*, int *ttl*)
设置多播TTL。

参数

- **handle** – UDP句柄。 应该以 `uv_udp_init()` 被初始化。
- **ttl** – 1 到 255 。

返回 0 若成功，或一个 < 0 的错误代码若失败。

int **uv_udp_set_multicast_interface**(*uv_udp_t* handle*, const char* *interface_addr*)
设置发送和接收数据所在的多播接口。

参数

- **handle** – UDP句柄。 应该以 `uv_udp_init()` 被初始化。
- **interface_addr** – 接口地址。

返回 0 若成功，或一个 < 0 的错误代码若失败。

int **uv_udp_set_broadcast**(*uv_udp_t* handle*, int *on*)
设置多播开关。

参数

- **handle** – UDP句柄。 应该以 `uv_udp_init()` 被初始化。
- **on** – 1 为开，0 为关。

返回 0 若成功，或一个 < 0 的错误代码若失败。

int **uv_udp_set_ttl**(*uv_udp_t* handle*, int *ttl*)
设置生存时间（TTL）。

参数

- **handle** – UDP句柄。 应该以 `uv_udp_init()` 被初始化。
- **ttl** – 1 到 255 。

返回 0 若成功，或一个 < 0 的错误代码若失败。

int **uv_udp_send**(*uv_udp_send_t* req*, *uv_udp_t* handle*, const *uv_buf_t bufs[]*, unsigned int *nbufs*, const struct sockaddr* *addr*, *uv_udp_send_cb send_cb*)
通过UDP套接字发送数据。 如果套接字之前没有用 `uv_udp_bind()` 绑定，它将绑定到 0.0.0.0（IPv4地址“所有接口”）和一个随机的端口号。

在Windows上如果 *addr* 被初始化指向一个未指定的地址（`0.0.0.0` 或者 `::`），它将被改变以指向 `localhost` 。 这么做是为了符合Linux系统的行为。

参数

- **req** – UDP请求句柄。 不需要初始化。
- **handle** – UDP句柄。 应该以 `uv_udp_init()` 被初始化。
- **bufs** – 发送缓冲区的列表。
- **nbufs** – *bufs* 中的缓冲区个数。
- **addr** – 带远端地址和端口的 *struct sockaddr_in* 或 *struct sockaddr_in6* 。
- **send_cb** – 当数据已发出时调用的回调函数。

返回 0 若成功，或一个 < 0 的错误代码若失败。

在 1.19.0 版更改: 新增 `0.0.0.0` 和 `::` 到 `localhost` 的映射

int **uv_udp_try_send**(*uv_udp_t* handle*, const *uv_buf_t bufs[]*, unsigned int *nbufs*, const struct sockaddr* *addr*)
与 *uv_udp_send()* 相同，但是如果无法立刻完成不会排队一个发送请求。

返回 >= 0： 已发送的字节数（匹配给定缓冲区的大小）。< 0： 负的错误代码（返回 `UV_EAGAIN` 当无法立刻发送消息时）。

int **uv_udp_recv_start**(*uv_udp_t* handle*, *uv_alloc_cb alloc_cb*, *uv_udp_recv_cb recv_cb*)
准备接受数据。 如果套接字之前没有用 *uv_udp_bind()* 绑定，它将绑定到 0.0.0.0 （IPv4地址“所有接口”）和一个随机的端口号。

参数

- **handle** – UDP句柄。 应该以 *uv_udp_init()* 被初始化。
- **alloc_cb** – 当需要临时存储时调用的回调函数。
- **recv_cb** – 接收数据调用的回调函数。

返回 0 若成功，或一个 < 0 的错误代码若失败。

int **uv_udp_recv_stop**(*uv_udp_t* handle*)
停止侦听新来的数据报。

参数

- **handle** – UDP句柄。 应该以 *uv_udp_init()* 被初始化。

返回 0 若成功，或一个 < 0 的错误代码若失败。

size_t **uv_udp_get_send_queue_size**(const *uv_udp_t* handle*)
返回 *handle->send_queue_size* 。

1.19.0 新版功能.

size_t **uv_udp_get_send_queue_count**(const *uv_udp_t* handle*)
返回 *handle->send_queue_count* 。

1.19.0 新版功能.

参见:

uv_handle_t 的API函数也适用。

3.2.19 `uv_fs_event_t` — FS Event handle

FS Event handles allow the user to monitor a given path for changes, for example, if the file was renamed or there was a generic change in it. This handle uses the best backend for the job on each platform.

注解: For AIX, the non default IBM bos.ahafs package has to be installed. The AIX Event Infrastructure file system (ahafs) has some limitations:

- ahafs tracks monitoring per process and is not thread safe. A separate process must be spawned for each monitor for the same event.
- Events for file modification (writing to a file) are not received if only the containing folder is watched.

See documentation for more details.

The z/OS file system events monitoring infrastructure does not notify of file creation/deletion within a directory that is being monitored. See the IBM Knowledge centre for more details.

Data types

uv_fs_event_t
FS Event handle type.

void **(*uv_fs_event_cb)**(*uv_fs_event_t** *handle*, const char* *filename*, int *events*, int *status*)
Callback passed to *uv_fs_event_start()* which will be called repeatedly after the handle is started. If the handle was started with a directory the *filename* parameter will be a relative path to a file contained in the directory. The *events* parameter is an ORed mask of *uv_fs_event* elements.

uv_fs_event
Event types that *uv_fs_event_t* handles monitor.

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

uv_fs_event_flags
Flags that can be passed to *uv_fs_event_start()* to control its behavior.

```
enum uv_fs_event_flags {
    /*
    * By default, if the fs event watcher is given a directory name, we will
    * watch for all events in that directory. This flags overrides this behavior
    * and makes fs_event report only changes to the directory entry itself. This
    * flag does not affect individual files watched.
    * This flag is currently not implemented yet on any backend.
    */
    UV_FS_EVENT_WATCH_ENTRY = 1,
    /*
    * By default uv_fs_event will try to use a kernel interface such as inotify
    * or kqueue to detect events. This may not work on remote file systems such
    * as NFS mounts. This flag makes fs_event fall back to calling stat() on a
    * regular interval.
    * This flag is currently not implemented yet on any backend.
    */
    UV_FS_EVENT_STAT = 2,
    /*
    * By default, event watcher, when watching directory, is not registering
    * (is ignoring) changes in its subdirectories.
    * This flag will override this behaviour on platforms that support it.
    */
    UV_FS_EVENT_RECURSIVE = 4
};
```

Public members

N/A

参见:

The *uv_handle_t* members also apply.

API

int **uv_fs_event_init**(*uv_loop_t* loop*, *uv_fs_event_t* handle*)
Initialize the handle.

int **uv_fs_event_start**(*uv_fs_event_t* handle*, *uv_fs_event_cb cb*, const char* *path*, unsigned int *flags*)
Start the handle with the given callback, which will watch the specified *path* for changes. *flags* can be an ORed mask of `uv_fs_event_flags`.

> **注解:** Currently the only supported flag is `UV_FS_EVENT_RECURSIVE` and only on OSX and Windows.

int **uv_fs_event_stop**(*uv_fs_event_t* handle*)
Stop the handle, the callback will no longer be called.

int **uv_fs_event_getpath**(*uv_fs_event_t* handle*, char* *buffer*, size_t* *size*)
Get the path being monitored by the handle. The buffer must be preallocated by the user. Returns 0 on success or an error code < 0 in case of failure. On success, *buffer* will contain the path and *size* its length. If the buffer is not big enough *UV_ENOBUFS* will be returned and *size* will be set to the required size, including the null terminator.

在 1.3.0 版更改: the returned length no longer includes the terminating null byte, and the buffer is not null terminated.

在 1.9.0 版更改: the returned length includes the terminating null byte on *UV_ENOBUFS*, and the buffer is null terminated on success.

参见:

The `uv_handle_t` API functions also apply.

3.2.20 uv_fs_poll_t — FS Poll handle

FS Poll handles allow the user to monitor a given path for changes. Unlike `uv_fs_event_t`, fs poll handles use *stat* to detect when a file has changed so they can work on file systems where fs event handles can't.

Data types

uv_fs_poll_t
FS Poll handle type.

void **(*uv_fs_poll_cb)**(*uv_fs_poll_t* handle*, int *status*, const *uv_stat_t* prev*, const *uv_stat_t* curr*)
Callback passed to `uv_fs_poll_start()` which will be called repeatedly after the handle is started, when any change happens to the monitored path.

The callback is invoked with *status* < *0* if *path* does not exist or is inaccessible. The watcher is *not* stopped but your callback is not called again until something changes (e.g. when the file is created or the error reason changes).

When *status* == *0*, the callback receives pointers to the old and new `uv_stat_t` structs. They are valid for the duration of the callback only.

Public members

N/A

参见:

The uv_handle_t members also apply.

API

int **uv_fs_poll_init**(*uv_loop_t* loop*, *uv_fs_poll_t* handle*)
Initialize the handle.

int **uv_fs_poll_start**(*uv_fs_poll_t* handle*, *uv_fs_poll_cb poll_cb*, const char* path*, unsigned int *interval*)
Check the file at *path* for changes every *interval* milliseconds.

注解: For maximum portability, use multi-second intervals. Sub-second intervals will not detect all changes on many file systems.

int **uv_fs_poll_stop**(*uv_fs_poll_t* handle*)
Stop the handle, the callback will no longer be called.

int **uv_fs_poll_getpath**(*uv_fs_poll_t* handle*, char* *buffer*, size_t* *size*)
Get the path being monitored by the handle. The buffer must be preallocated by the user. Returns 0 on success or an error code < 0 in case of failure. On success, *buffer* will contain the path and *size* its length. If the buffer is not big enough *UV_ENOBUFS* will be returned and *size* will be set to the required size.

在 1.3.0 版更改: the returned length no longer includes the terminating null byte, and the buffer is not null terminated.

在 1.9.0 版更改: the returned length includes the terminating null byte on *UV_ENOBUFS*, and the buffer is null terminated on success.

参见:

The uv_handle_t API functions also apply.

3.2.21 File system operations

libuv provides a wide variety of cross-platform sync and async file system operations. All functions defined in this document take a callback, which is allowed to be NULL. If the callback is NULL the request is completed synchronously, otherwise it will be performed asynchronously.

All file operations are run on the threadpool. See *Thread pool work scheduling* for information on the threadpool size.

注解: On Windows *uv_fs_** functions use utf-8 encoding.

Data types

uv_fs_t
File system request type.

uv_timespec_t
Portable equivalent of `struct timespec`.

```
typedef struct {
    long tv_sec;
    long tv_nsec;
} uv_timespec_t;
```

uv_stat_t

Portable equivalent of `struct stat`.

```
typedef struct {
    uint64_t st_dev;
    uint64_t st_mode;
    uint64_t st_nlink;
    uint64_t st_uid;
    uint64_t st_gid;
    uint64_t st_rdev;
    uint64_t st_ino;
    uint64_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

File system request type.

```
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,
```

(continues on next page)

(续上页)

```
    UV_FS_READLINK,
    UV_FS_CHOWN,
    UV_FS_FCHOWN,
    UV_FS_REALPATH,
    UV_FS_COPYFILE,
    UV_FS_LCHOWN
} uv_fs_type;
```

uv_dirent_t

Cross platform (reduced) equivalent of `struct dirent`. Used in *uv_fs_scandir_next()*.

```
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;
```

Public members

*uv_loop_t** **uv_fs_t.loop**

Loop that started this request and where completion will be reported. Readonly.

uv_fs_type **uv_fs_t.fs_type**

FS request type.

const char* **uv_fs_t.path**

Path affecting the request.

ssize_t **uv_fs_t.result**

Result of the request. < 0 means error, success otherwise. On requests such as *uv_fs_read()* or *uv_fs_write()* it indicates the amount of data that was read or written, respectively.

uv_stat_t **uv_fs_t.statbuf**

Stores the result of *uv_fs_stat()* and other stat requests.

void* **uv_fs_t.ptr**

Stores the result of *uv_fs_readlink()* and *uv_fs_realpath()* and serves as an alias to *statbuf*.

参见:

The *uv_req_t* members also apply.

API

void **uv_fs_req_cleanup**(*uv_fs_t** *req*)

Cleanup request. Must be called after a request is finished to deallocate any memory libuv might have allocated.

int **uv_fs_close**(*uv_loop_t* loop*, *uv_fs_t* req*, *uv_file file*, uv_fs_cb *cb*)
Equivalent to 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*)
Equivalent to open(2).

注解: On Windows libuv uses *CreateFileW* and thus the file is always opened in binary mode. Because of this the O_BINARY and O_TEXT flags are not supported.

---

int **uv_fs_read**(*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*)
Equivalent to preadv(2).

int **uv_fs_unlink**(*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, uv_fs_cb *cb*)
Equivalent to 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*)
Equivalent to pwritev(2).

int **uv_fs_mkdir**(*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, int *mode*, uv_fs_cb *cb*)
Equivalent to mkdir(2).

注解: *mode* is currently not implemented on Windows.

---

int **uv_fs_mkdtemp**(*uv_loop_t* loop*, *uv_fs_t* req*, const char* *tpl*, uv_fs_cb *cb*)
Equivalent to mkdtemp(3).

注解: The result can be found as a null terminated string at *req->path*.

---

int **uv_fs_rmdir**(*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, uv_fs_cb *cb*)
Equivalent to 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*)
Equivalent to scandir(3), with a slightly different API. Once the callback for the request is called, the user can use `uv_fs_scandir_next()` to get *ent* populated with the next directory entry data. When there are no more entries `UV_EOF` will be returned.

注解: Unlike *scandir(3)*, this function does not return the "." and ".." entries.

注解: On Linux, getting the type of an entry is only supported by some file systems (btrfs, ext2, ext3 and ext4 at the time of this writing), check the getdents(2) man page.

---

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*)
Equivalent to stat(2), fstat(2) and lstat(2) respectively.

int **uv_fs_rename**(*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, const char* *new_path*, uv_fs_cb *cb*)
Equivalent to rename(2).

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

> **注解:** For AIX, *uv_fs_fsync* returns *UV_EBADF* on file descriptors referencing non regular files.

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

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

int **uv_fs_copyfile**(*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, const char* *new_path*, int *flags*, uv_fs_cb *cb*)
Copies a file from *path* to *new_path*. Supported *flags* are described below.

- *UV_FS_COPYFILE_EXCL*: If present, *uv_fs_copyfile()* will fail with *UV_EEXIST* if the destination path already exists. The default behavior is to overwrite the destination if it exists.
- *UV_FS_COPYFILE_FICLONE*: If present, *uv_fs_copyfile()* will attempt to create a copy-on-write reflink. If the underlying platform does not support copy-on-write, then a fallback copy mechanism is used.
- *UV_FS_COPYFILE_FICLONE_FORCE*: If present, *uv_fs_copyfile()* will attempt to create a copy-on-write reflink. If the underlying platform does not support copy-on-write, then an error is returned.

> **警告:** If the destination path is created, but an error occurs while copying the data, then the destination path is removed. There is a brief window of time between closing and removing the file where another process could access the file.

1.14.0 新版功能.

在 1.20.0 版更改: *UV_FS_COPYFILE_FICLONE* and *UV_FS_COPYFILE_FICLONE_FORCE* are supported.

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*)
Limited equivalent to sendfile(2).

int **uv_fs_access**(*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, int *mode*, uv_fs_cb *cb*)
Equivalent to access(2) on Unix. Windows uses `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*)
Equivalent to chmod(2) and fchmod(2) respectively.

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*)
Equivalent to utime(2) and futime(2) respectively.

> **注解:** AIX: This function only works for AIX 7.1 and newer. It can still be called on older versions but will return `UV_ENOSYS`.

在 1.10.0 版更改: sub-second precission is supported on Windows

int **uv_fs_link**(*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, const char* *new_path*, uv_fs_cb *cb*)
Equivalent to 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*)
Equivalent to symlink(2).

注解: On Windows the *flags* parameter can be specified to control how the symlink will be created:

- `UV_FS_SYMLINK_DIR`: indicates that *path* points to a directory.
- `UV_FS_SYMLINK_JUNCTION`: request that the symlink is created using junction points.

---

int **uv_fs_readlink**(*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, uv_fs_cb *cb*)
Equivalent to readlink(2). The resulting string is stored in *req->ptr*.

int **uv_fs_realpath**(*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, uv_fs_cb *cb*)
Equivalent to realpath(3) on Unix. Windows uses GetFinalPathNameByHandle. The resulting string is stored in *req->ptr*.

警告: This function has certain platform-specific caveats that were discovered when used in Node.

- macOS and other BSDs: this function will fail with UV_ELOOP if more than 32 symlinks are found while resolving the given path. This limit is hardcoded and cannot be sidestepped.
- Windows: while this function works in the common case, there are a number of corner cases where it doesn't:
  - Paths in ramdisk volumes created by tools which sidestep the Volume Manager (such as ImDisk) cannot be resolved.
  - Inconsistent casing when using drive letters.
  - Resolved path bypasses subst'd drives.

While this function can still be used, it's not recommended if scenarios such as the above need to be supported.

The background story and some more details on these issues can be checked here.

注解: This function is not implemented on Windows XP and Windows Server 2003. On these systems, UV_ENOSYS is returned.

1.8.0 新版功能.

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*)

int **uv_fs_lchown**(*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, uv_uid_t *uid*, uv_gid_t *gid*, uv_fs_cb *cb*)
Equivalent to chown(2), fchown(2) and lchown(2) respectively.

注解: These functions are not implemented on Windows.

在 1.21.0 版更改: implemented uv_fs_lchown

uv_fs_type **uv_fs_get_type**(const *uv_fs_t** *req*)
Returns *req->fs_type*.

1.19.0 新版功能.

ssize_t **uv_fs_get_result**(const *uv_fs_t** *req*)
Returns *req->result*.

1.19.0 新版功能.

void* **uv_fs_get_ptr**(const *uv_fs_t** *req*)
Returns *req->ptr*.

1.19.0 新版功能.

const char* **uv_fs_get_path**(const *uv_fs_t** *req*)
Returns *req->path*.

1.19.0 新版功能.

*uv_stat_t** **uv_fs_get_statbuf**(*uv_fs_t** *req*)
Returns *&req->statbuf*.

1.19.0 新版功能.

参见:

The `uv_req_t` API functions also apply.

Helper functions

uv_os_fd_t **uv_get_osfhandle**(int *fd*)
For a file descriptor in the C runtime, get the OS-dependent handle. On UNIX, returns the `fd` intact. On Windows, this calls _get_osfhandle. Note that the return value is still owned by the C runtime, any attempts to close it or to use it after closing the fd may lead to malfunction.

1.12.0 新版功能.

int **uv_open_osfhandle**(*uv_os_fd_t os_fd*)
For a OS-dependent handle, get the file descriptor in the C runtime. On UNIX, returns the `os_fd` intact. On Windows, this calls _open_osfhandle. Note that the return value is still owned by the CRT, any attempts to close it or to use it after closing the handle may lead to malfunction.

1.23.0 新版功能.

File open constants

UV_FS_O_APPEND
The file is opened in append mode. Before each write, the file offset is positioned at the end of the file.

UV_FS_O_CREAT
The file is created if it does not already exist.

UV_FS_O_DIRECT
File I/O is done directly to and from user-space buffers, which must be aligned. Buffer size and address should be a multiple of the physical sector size of the block device.

注解: *UV_FS_O_DIRECT* is supported on Linux, and on Windows via FILE_FLAG_NO_BUFFERING. *UV_FS_O_DIRECT* is not supported on macOS.

UV_FS_O_DIRECTORY
If the path is not a directory, fail the open.

注解：*UV_FS_O_DIRECTORY* is not supported on Windows.

UV_FS_O_DSYNC
The file is opened for synchronous I/O. Write operations will complete once all data and a minimum of metadata are flushed to disk.

注解：*UV_FS_O_DSYNC* is supported on Windows via FILE_FLAG_WRITE_THROUGH.

UV_FS_O_EXCL
If the *O_CREAT* flag is set and the file already exists, fail the open.

注解：In general, the behavior of *O_EXCL* is undefined if it is used without *O_CREAT*. There is one exception: on Linux 2.6 and later, *O_EXCL* can be used without *O_CREAT* if pathname refers to a block device. If the block device is in use by the system (e.g., mounted), the open will fail with the error *EBUSY*.

UV_FS_O_EXLOCK
Atomically obtain an exclusive lock.

注解：*UV_FS_O_EXLOCK* is only supported on macOS and Windows.

在 1.17.0 版更改: support is added for Windows.

UV_FS_O_NOATIME
Do not update the file access time when the file is read.

注解：*UV_FS_O_NOATIME* is not supported on Windows.

UV_FS_O_NOCTTY
If the path identifies a terminal device, opening the path will not cause that terminal to become the controlling terminal for the process (if the process does not already have one).

注解：*UV_FS_O_NOCTTY* is not supported on Windows.

UV_FS_O_NOFOLLOW
If the path is a symbolic link, fail the open.

注解：*UV_FS_O_NOFOLLOW* is not supported on Windows.

UV_FS_O_NONBLOCK
Open the file in nonblocking mode if possible.

注解：*UV_FS_O_NONBLOCK* is not supported on Windows.

UV_FS_O_RANDOM
Access is intended to be random. The system can use this as a hint to optimize file caching.

注解：*UV_FS_O_RANDOM* is only supported on Windows via FILE_FLAG_RANDOM_ACCESS.

UV_FS_O_RDONLY
Open the file for read-only access.

UV_FS_O_RDWR
Open the file for read-write access.

UV_FS_O_SEQUENTIAL
Access is intended to be sequential from beginning to end. The system can use this as a hint to optimize file caching.

注解：*UV_FS_O_SEQUENTIAL* is only supported on Windows via FILE_FLAG_SEQUENTIAL_SCAN.

UV_FS_O_SHORT_LIVED
The file is temporary and should not be flushed to disk if possible.

注解：*UV_FS_O_SHORT_LIVED* is only supported on Windows via FILE_ATTRIBUTE_TEMPORARY.

UV_FS_O_SYMLINK
Open the symbolic link itself rather than the resource it points to.

UV_FS_O_SYNC
The file is opened for synchronous I/O. Write operations will complete once all data and all metadata are flushed to disk.

注解：*UV_FS_O_SYNC* is supported on Windows via FILE_FLAG_WRITE_THROUGH.

UV_FS_O_TEMPORARY
The file is temporary and should not be flushed to disk if possible.

注解：*UV_FS_O_TEMPORARY* is only supported on Windows via FILE_ATTRIBUTE_TEMPORARY.

UV_FS_O_TRUNC
If the file exists and is a regular file, and the file is opened successfully for write access, its length shall be truncated to zero.

UV_FS_O_WRONLY
Open the file for write-only access.

3.2.22 Thread pool work scheduling

libuv provides a threadpool which can be used to run user code and get notified in the loop thread. This thread pool is internally used to run all file system operations, as well as getaddrinfo and getnameinfo requests.

Its default size is 4, but it can be changed at startup time by setting the `UV_THREADPOOL_SIZE` environment variable to any value (the absolute maximum is 128).

The threadpool is global and shared across all event loops. When a particular function makes use of the threadpool (i.e. when using *uv_queue_work()*) libuv preallocates and initializes the maximum number of threads allowed by UV_THREADPOOL_SIZE. This causes a relatively minor memory overhead (~1MB for 128 threads) but increases the performance of threading at runtime.

注解: Note that even though a global thread pool which is shared across all events loops is used, the functions are not thread safe.

Data types

uv_work_t
Work request type.

void **(*uv_work_cb)**(*uv_work_t** *req*)
Callback passed to *uv_queue_work()* which will be run on the thread pool.

void **(*uv_after_work_cb)**(*uv_work_t** *req*, int *status*)
Callback passed to *uv_queue_work()* which will be called on the loop thread after the work on the threadpool has been completed. If the work was cancelled using *uv_cancel()* *status* will be UV_ECANCELED.

Public members

*uv_loop_t** **uv_work_t.loop**
Loop that started this request and where completion will be reported. Readonly.

参见:

The *uv_req_t* members also apply.

API

int **uv_queue_work**(*uv_loop_t** *loop*, *uv_work_t** *req*, *uv_work_cb* *work_cb*, *uv_after_work_cb* *after_work_cb*)
Initializes a work request which will run the given *work_cb* in a thread from the threadpool. Once *work_cb* is completed, *after_work_cb* will be called on the loop thread.

This request can be cancelled with *uv_cancel()*.

参见:

The *uv_req_t* API functions also apply.

3.2.23 DNS utility functions

libuv provides asynchronous variants of *getaddrinfo* and *getnameinfo*.

Data types

uv_getaddrinfo_t
getaddrinfo request type.

void **(*uv_getaddrinfo_cb)**(*uv_getaddrinfo_t** *req*, int *status*, struct addrinfo* *res*)
Callback which will be called with the getaddrinfo request result once complete. In case it was cancelled, *status* will have a value of `UV_ECANCELED`.

uv_getnameinfo_t
getnameinfo request type.

void **(*uv_getnameinfo_cb)**(*uv_getnameinfo_t** *req*, int *status*, const char* *hostname*, const char* *service*)
Callback which will be called with the getnameinfo request result once complete. In case it was cancelled, *status* will have a value of `UV_ECANCELED`.

Public members

*uv_loop_t** **uv_getaddrinfo_t.loop**
Loop that started this getaddrinfo request and where completion will be reported. Readonly.

struct addrinfo* **uv_getaddrinfo_t.addrinfo**
Pointer to a *struct addrinfo* containing the result. Must be freed by the user with *uv_freeaddrinfo()*.

在 1.3.0 版更改: the field is declared as public.

*uv_loop_t** **uv_getnameinfo_t.loop**
Loop that started this getnameinfo request and where completion will be reported. Readonly.

char[NI_MAXHOST] **uv_getnameinfo_t.host**
Char array containing the resulting host. It's null terminated.

在 1.3.0 版更改: the field is declared as public.

char[NI_MAXSERV] **uv_getnameinfo_t.service**
Char array containing the resulting service. It's null terminated.

在 1.3.0 版更改: the field is declared as public.

参见:

The *uv_req_t* members also apply.

API

int **uv_getaddrinfo**(*uv_loop_t** *loop*, *uv_getaddrinfo_t** *req*, *uv_getaddrinfo_cb* *getaddrinfo_cb*, const char* *node*, const char* *service*, const struct addrinfo* *hints*)
Asynchronous getaddrinfo(3).

Either node or service may be NULL but not both.

hints is a pointer to a struct addrinfo with additional address type constraints, or NULL. Consult *man -s 3 getaddrinfo* for more details.

Returns 0 on success or an error code < 0 on failure. If successful, the callback will get called sometime in the future with the lookup result, which is either:

- status == 0, the res argument points to a valid *struct addrinfo*, or
- status < 0, the res argument is NULL. See the UV_EAI_* constants.

Call *uv_freeaddrinfo()* to free the addrinfo structure.

在 1.3.0 版更改: the callback parameter is now allowed to be NULL, in which case the request will run **synchronously**.

void **uv_freeaddrinfo**(struct addrinfo* *ai*)
Free the struct addrinfo. Passing NULL is allowed and is a no-op.

int **uv_getnameinfo**(*uv_loop_t** *loop*, *uv_getnameinfo_t** *req*, *uv_getnameinfo_cb* *getnameinfo_cb*, const struct sockaddr* *addr*, int *flags*)
Asynchronous getnameinfo(3).

Returns 0 on success or an error code < 0 on failure. If successful, the callback will get called sometime in the future with the lookup result. Consult *man -s 3 getnameinfo* for more details.

在 1.3.0 版更改: the callback parameter is now allowed to be NULL, in which case the request will run **synchronously**.

参见:

The `uv_req_t` API functions also apply.

3.2.24 Shared library handling

libuv provides cross platform utilities for loading shared libraries and retrieving symbols from them, using the following API.

Data types

uv_lib_t
Shared library data type.

Public members

N/A

API

int **uv_dlopen**(const char* *filename*, *uv_lib_t** *lib*)
Opens a shared library. The filename is in utf-8. Returns 0 on success and -1 on error. Call `uv_dlerror()` to get the error message.

void **uv_dlclose**(*uv_lib_t** *lib*)
Close the shared library.

int **uv_dlsym**(*uv_lib_t** *lib*, const char* *name*, void** *ptr*)
Retrieves a data pointer from a dynamic library. It is legal for a symbol to map to NULL. Returns 0 on success and -1 if the symbol was not found.

const char* **uv_dlerror**(const *uv_lib_t** *lib*)
Returns the last uv_dlopen() or uv_dlsym() error message.

3.2.25 Threading and synchronization utilities

libuv provides cross-platform implementations for multiple threading and synchronization primitives. The API largely follows the pthreads API.

Data types

uv_thread_t
Thread data type.

void **(*uv_thread_cb)**(void* *arg*)
Callback that is invoked to initialize thread execution. *arg* is the same value that was passed to *uv_thread_create()*.

uv_key_t
Thread-local key data type.

uv_once_t
Once-only initializer data type.

uv_mutex_t
Mutex data type.

uv_rwlock_t
Read-write lock data type.

uv_sem_t
Semaphore data type.

uv_cond_t
Condition data type.

uv_barrier_t
Barrier data type.

API

Threads

uv_thread_options_t
Options for spawning a new thread (passed to *uv_thread_create_ex()*).

```
typedef struct uv_process_options_s {
  enum {
    UV_THREAD_NO_FLAGS = 0x00,
    UV_THREAD_HAS_STACK_SIZE = 0x01
  } flags;
  size_t stack_size;
} uv_process_options_t;
```

More fields may be added to this struct at any time, so its exact layout and size should not be relied upon.

1.26.0 新版功能.

int **uv_thread_create**(*uv_thread_t** *tid*, *uv_thread_cb* *entry*, void* *arg*)
在 1.4.1 版更改: returns a UV_E* error code on failure

int **uv_thread_create_ex**(*uv_thread_t** *tid*, const *uv_thread_options_t** *params*, *uv_thread_cb* *entry*, void* *arg*)
Like *uv_thread_create()*, but additionally specifies options for creating a new thread.

If *UV_THREAD_HAS_STACK_SIZE* is set, *stack_size* specifies a stack size for the new thread. *0* indicates that the default value should be used, i.e. behaves as if the flag was not set. Other values will be rounded up to the nearest page boundary.

1.26.0 新版功能.

uv_thread_t **uv_thread_self**(void)

int **uv_thread_join**(*uv_thread_t *tid*)

int **uv_thread_equal**(const *uv_thread_t* t1*, const *uv_thread_t* t2*)

Thread-local storage

注解: The total thread-local storage size may be limited. That is, it may not be possible to create many TLS keys.

---

int **uv_key_create**(*uv_key_t* key*)

void **uv_key_delete**(*uv_key_t* key*)

void* **uv_key_get**(*uv_key_t* key*)

void **uv_key_set**(*uv_key_t* key*, void* value*)

Once-only initialization

Runs a function once and only once. Concurrent calls to `uv_once()` with the same guard will block all callers except one (it's unspecified which one). The guard should be initialized statically with the UV_ONCE_INIT macro.

void **uv_once**(*uv_once_t* guard*, void (**callback*)(void))

Mutex locks

Functions return 0 on success or an error code < 0 (unless the return type is void, of course).

int **uv_mutex_init**(*uv_mutex_t* handle*)

int **uv_mutex_init_recursive**(*uv_mutex_t* handle*)

void **uv_mutex_destroy**(*uv_mutex_t* handle*)

void **uv_mutex_lock**(*uv_mutex_t* handle*)

int **uv_mutex_trylock**(*uv_mutex_t* handle*)

void **uv_mutex_unlock**(*uv_mutex_t* handle*)

Read-write locks

Functions return 0 on success or an error code < 0 (unless the return type is void, of course).

int **uv_rwlock_init**(*uv_rwlock_t* rwlock*)

void **uv_rwlock_destroy**(*uv_rwlock_t* rwlock*)

void **uv_rwlock_rdlock**(*uv_rwlock_t* rwlock*)

int **uv_rwlock_tryrdlock**(*uv_rwlock_t* rwlock*)

void **uv_rwlock_rdunlock**(*uv_rwlock_t* rwlock*)

void **uv_rwlock_wrlock**(*uv_rwlock_t* rwlock*)

int **uv_rwlock_trywrlock**(*uv_rwlock_t* rwlock*)

void **uv_rwlock_wrunlock**(*uv_rwlock_t* rwlock*)

Semaphores

Functions return 0 on success or an error code < 0 (unless the return type is void, of course).

int **uv_sem_init**(*uv_sem_t* sem*, unsigned int *value*)

void **uv_sem_destroy**(*uv_sem_t* sem*)

void **uv_sem_post**(*uv_sem_t* sem*)

void **uv_sem_wait**(*uv_sem_t* sem*)

int **uv_sem_trywait**(*uv_sem_t* sem*)

Conditions

Functions return 0 on success or an error code < 0 (unless the return type is void, of course).

注解:

1. Callers should be prepared to deal with spurious wakeups on `uv_cond_wait()` and `uv_cond_timedwait()`.
2. The timeout parameter for `uv_cond_timedwait()` is relative to the time at which function is called.
3. On z/OS, the timeout parameter for `uv_cond_timedwait()` is converted to an absolute system time at which the wait expires. If the current system clock time passes the absolute time calculated before the condition is signaled, an ETIMEDOUT error results. After the wait begins, the wait time is not affected by changes to the system clock.

---

int **uv_cond_init**(*uv_cond_t* cond*)

void **uv_cond_destroy**(*uv_cond_t* cond*)

void **uv_cond_signal**(*uv_cond_t* cond*)

void **uv_cond_broadcast**(*uv_cond_t* cond*)

void **uv_cond_wait**(*uv_cond_t* cond*, *uv_mutex_t* mutex*)

int **uv_cond_timedwait**(*uv_cond_t* cond*, *uv_mutex_t* mutex*, uint64_t *timeout*)

Barriers

Functions return 0 on success or an error code < 0 (unless the return type is void, of course).

注解: `uv_barrier_wait()` returns a value > 0 to an arbitrarily chosen "serializer" thread to facilitate cleanup, i.e.

```
if (uv_barrier_wait(&barrier) > 0)
    uv_barrier_destroy(&barrier);
```

---

int **uv_barrier_init**(*uv_barrier_t* barrier*, unsigned int *count*)

void **uv_barrier_destroy**(*uv_barrier_t* barrier*)

int **uv_barrier_wait**(*uv_barrier_t* barrier*)

3.2.26 Miscellaneous utilities

This section contains miscellaneous functions that don't really belong in any other section.

Data types

uv_buf_t
Buffer data type.

char* **uv_buf_t.base**
Pointer to the base of the buffer.

size_t **uv_buf_t.len**
Total bytes in the buffer.

> **注解:** On Windows this field is ULONG.

void* **(*uv_malloc_func)**(size_t *size*)
Replacement function for malloc(3). See *uv_replace_allocator()*.

void* **(*uv_realloc_func)**(void* *ptr*, size_t *size*)
Replacement function for realloc(3). See *uv_replace_allocator()*.

void* **(*uv_calloc_func)**(size_t *count*, size_t *size*)
Replacement function for calloc(3). See *uv_replace_allocator()*.

void **(*uv_free_func)**(void* *ptr*)
Replacement function for free(3). See *uv_replace_allocator()*.

uv_file
Cross platform representation of a file handle.

uv_os_sock_t
Cross platform representation of a socket handle.

uv_os_fd_t
Abstract representation of a file descriptor. On Unix systems this is a *typedef* of *int* and on Windows a *HANDLE*.

uv_pid_t
Cross platform representation of a *pid_t*.

1.16.0 新版功能.

uv_rusage_t
Data type for resource usage results.

```
typedef struct {
    uv_timeval_t ru_utime; /* user CPU time used */
    uv_timeval_t ru_stime; /* system CPU time used */
    uint64_t ru_maxrss; /* maximum resident set size */
    uint64_t ru_ixrss; /* integral shared memory size (X) */
    uint64_t ru_idrss; /* integral unshared data size (X) */
```

(continues on next page)

(续上页)

```
    uint64_t ru_isrss; /* integral unshared stack size (X) */
    uint64_t ru_minflt; /* page reclaims (soft page faults) (X) */
    uint64_t ru_majflt; /* page faults (hard page faults) */
    uint64_t ru_nswap; /* swaps (X) */
    uint64_t ru_inblock; /* block input operations */
    uint64_t ru_oublock; /* block output operations */
    uint64_t ru_msgsnd; /* IPC messages sent (X) */
    uint64_t ru_msgrcv; /* IPC messages received (X) */
    uint64_t ru_nsignals; /* signals received (X) */
    uint64_t ru_nvcsw; /* voluntary context switches (X) */
    uint64_t ru_nivcsw; /* involuntary context switches (X) */
} uv_rusage_t;
```

Members marked with *(X)* are unsupported on Windows. See getrusage(2) for supported fields on Unix

uv_cpu_info_t

Data type for CPU information.

```
typedef struct uv_cpu_info_s {
    char* model;
    int speed;
    struct uv_cpu_times_s {
        uint64_t user;
        uint64_t nice;
        uint64_t sys;
        uint64_t idle;
        uint64_t irq;
    } cpu_times;
} uv_cpu_info_t;
```

uv_interface_address_t

Data type for interface addresses.

```
typedef struct uv_interface_address_s {
    char* name;
    char phys_addr[6];
    int is_internal;
    union {
        struct sockaddr_in address4;
        struct sockaddr_in6 address6;
    } address;
    union {
        struct sockaddr_in netmask4;
        struct sockaddr_in6 netmask6;
    } netmask;
} uv_interface_address_t;
```

uv_passwd_t

Data type for password file information.

```
typedef struct uv_passwd_s {
    char* username;
    long uid;
    long gid;
    char* shell;
    char* homedir;
} uv_passwd_t;
```

uv_utsname_t

Data type for operating system name and version information.

```
typedef struct uv_utsname_s {
    char sysname[256];
    char release[256];
    char version[256];
    char machine[256];
} uv_utsname_t;
```

API

uv_handle_type **uv_guess_handle**(*uv_file file*)

Used to detect what type of stream should be used with a given file descriptor. Usually this will be used during initialization to guess the type of the stdio streams.

For isatty(3) equivalent functionality use this function and test for `UV_TTY`.

int **uv_replace_allocator**(*uv_malloc_func malloc_func*, *uv_realloc_func realloc_func*, *uv_calloc_func calloc_func*, *uv_free_func free_func*)

1.6.0 新版功能.

Override the use of the standard library's malloc(3), calloc(3), realloc(3), free(3), memory allocation functions.

This function must be called before any other libuv function is called or after all resources have been freed and thus libuv doesn't reference any allocated memory chunk.

On success, it returns 0, if any of the function pointers is NULL it returns UV_EINVAL.

> **警告:** There is no protection against changing the allocator multiple times. If the user changes it they are responsible for making sure the allocator is changed while no memory was allocated with the previous allocator, or that they are compatible.

uv_buf_t **uv_buf_init**(char* *base*, unsigned int *len*)

Constructor for *uv_buf_t*.

Due to platform differences the user cannot rely on the ordering of the *base* and *len* members of the uv_buf_t struct. The user is responsible for freeing *base* after the uv_buf_t is done. Return struct passed by value.

char** **uv_setup_args**(int *argc*, char** *argv*)

Store the program arguments. Required for getting / setting the process title.

int **uv_get_process_title**(char* *buffer*, size_t *size*)

Gets the title of the current process. You *must* call *uv_setup_args* before calling this function. If *buffer* is *NULL* or *size* is zero, *UV_EINVAL* is returned. If *size* cannot accommodate the process title and terminating *NULL* character, the function returns *UV_ENOBUFS*.

在 1.18.1 版更改: now thread-safe on all supported platforms.

int **uv_set_process_title**(const char* *title*)

Sets the current process title. You *must* call *uv_setup_args* before calling this function. On platforms with a fixed size buffer for the process title the contents of *title* will be copied to the buffer and truncated if larger than the available space. Other platforms will return *UV_ENOMEM* if they cannot allocate enough space to duplicate the contents of *title*.

在 1.18.1 版更改: now thread-safe on all supported platforms.

int **uv_resident_set_memory**(size_t* *rss*)
Gets the resident set size (RSS) for the current process.

int **uv_uptime**(double* *uptime*)
Gets the current system uptime.

int **uv_getrusage**(*uv_rusage_t** *rusage*)
Gets the resource usage measures for the current process.

注解: On Windows not all fields are set, the unsupported fields are filled with zeroes. See `uv_rusage_t` for more details.

---

uv_pid_t **uv_os_getpid**(void)
Returns the current process ID.

1.18.0 新版功能.

uv_pid_t **uv_os_getppid**(void)
Returns the parent process ID.

1.16.0 新版功能.

int **uv_cpu_info**(*uv_cpu_info_t*** *cpu_infos*, int* *count*)
Gets information about the CPUs on the system. The *cpu_infos* array will have *count* elements and needs to be freed with `uv_free_cpu_info()`.

void **uv_free_cpu_info**(*uv_cpu_info_t** *cpu_infos*, int *count*)
Frees the *cpu_infos* array previously allocated with `uv_cpu_info()`.

int **uv_interface_addresses**(*uv_interface_address_t*** *addresses*, int* *count*)
Gets address information about the network interfaces on the system. An array of *count* elements is allocated and returned in *addresses*. It must be freed by the user, calling `uv_free_interface_addresses()`.

void **uv_free_interface_addresses**(*uv_interface_address_t** *addresses*, int *count*)
Free an array of `uv_interface_address_t` which was returned by `uv_interface_addresses()`.

void **uv_loadavg**(double *avg[3]*)
Gets the load average. See: http://en.wikipedia.org/wiki/Load_(computing)

注解: Returns [0,0,0] on Windows (i.e., it's not implemented).

---

int **uv_ip4_addr**(const char* *ip*, int *port*, struct sockaddr_in* *addr*)
Convert a string containing an IPv4 addresses to a binary structure.

int **uv_ip6_addr**(const char* *ip*, int *port*, struct sockaddr_in6* *addr*)
Convert a string containing an IPv6 addresses to a binary structure.

int **uv_ip4_name**(const struct sockaddr_in* *src*, char* *dst*, size_t *size*)
Convert a binary structure containing an IPv4 address to a string.

int **uv_ip6_name**(const struct sockaddr_in6* *src*, char* *dst*, size_t *size*)
Convert a binary structure containing an IPv6 address to a string.

int **uv_inet_ntop**(int *af*, const void* *src*, char* *dst*, size_t *size*)

int **uv_inet_pton**(int *af*, const char* *src*, void* *dst*)
Cross-platform IPv6-capable implementation of inet_ntop(3) and inet_pton(3). On success they return 0. In case of error the target *dst* pointer is unmodified.

UV_IF_NAMESIZE
Maximum IPv6 interface identifier name length. Defined as *IFNAMSIZ* on Unix and *IF_NAMESIZE* on Linux and Windows.

1.16.0 新版功能.

int **uv_if_indextoname**(unsigned int *ifindex*, char* *buffer*, size_t* *size*)
IPv6-capable implementation of if_indextoname(3). When called, **size* indicates the length of the *buffer*, which is used to store the result. On success, zero is returned, *buffer* contains the interface name, and **size* represents the string length of the *buffer*, excluding the NUL terminator byte from **size*. On error, a negative result is returned. If *buffer* is not large enough to hold the result, *UV_ENOBUFS* is returned, and **size* represents the necessary size in bytes, including the NUL terminator byte into the **size*.

On Unix, the returned interface name can be used directly as an interface identifier in scoped IPv6 addresses, e.g. *fe80::abc:def1:2345%en0*.

On Windows, the returned interface cannot be used as an interface identifier, as Windows uses numerical interface identifiers, e.g. *fe80::abc:def1:2345%5*.

To get an interface identifier in a cross-platform compatible way, use *uv_if_indextoiid()*.

Example:

```
char ifname[UV_IF_NAMESIZE];
size_t size = sizeof(ifname);
uv_if_indextoname(sin6->sin6_scope_id, ifname, &size);
```

1.16.0 新版功能.

int **uv_if_indextoiid**(unsigned int *ifindex*, char* *buffer*, size_t* *size*)
Retrieves a network interface identifier suitable for use in an IPv6 scoped address. On Windows, returns the numeric *ifindex* as a string. On all other platforms, *uv_if_indextoname()* is called. The result is written to *buffer*, with **size* indicating the length of *buffer*. If *buffer* is not large enough to hold the result, then *UV_ENOBUFS* is returned, and **size* represents the size, including the NUL byte, required to hold the result.

See *uv_if_indextoname* for further details.

1.16.0 新版功能.

int **uv_exepath**(char* *buffer*, size_t* *size*)
Gets the executable path.

int **uv_cwd**(char* *buffer*, size_t* *size*)
Gets the current working directory, and stores it in *buffer*. If the current working directory is too large to fit in *buffer*, this function returns *UV_ENOBUFS*, and sets *size* to the required length, including the null terminator.

在 1.1.0 版更改: On Unix the path no longer ends in a slash.

在 1.9.0 版更改: the returned length includes the terminating null byte on *UV_ENOBUFS*, and the buffer is null terminated on success.

int **uv_chdir**(const char* *dir*)
Changes the current working directory.

int **uv_os_homedir**(char* *buffer*, size_t* *size*)
Gets the current user's home directory. On Windows, *uv_os_homedir()* first checks the *USERPROFILE* environment variable using *GetEnvironmentVariableW()*. If *USERPROFILE* is not set, *GetUserProfileDirectoryW()* is called. On all other operating systems, *uv_os_homedir()* first checks the *HOME* environment variable using getenv(3). If *HOME* is not set, getpwuid_r(3) is called. The user's home directory is stored in *buffer*. When *uv_os_homedir()* is called, *size* indicates the maximum size of *buffer*. On success *size* is set to the string length of *buffer*. On *UV_ENOBUFS* failure *size* is set to the required length for *buffer*, including the null byte.

> **警告：** *uv_os_homedir()* is not thread safe.

1.6.0 新版功能.

int **uv_os_tmpdir**(char* *buffer*, size_t* *size*)
Gets the temp directory. On Windows, *uv_os_tmpdir()* uses *GetTempPathW()*. On all other operating systems, *uv_os_tmpdir()* uses the first environment variable found in the ordered list *TMPDIR*, *TMP*, *TEMP*, and *TEMPDIR*. If none of these are found, the path *"/tmp"* is used, or, on Android, *"/data/local/tmp"* is used. The temp directory is stored in *buffer*. When *uv_os_tmpdir()* is called, *size* indicates the maximum size of *buffer*. On success *size* is set to the string length of *buffer* (which does not include the terminating null). On *UV_ENOBUFS* failure *size* is set to the required length for *buffer*, including the null byte.

> **警告：** *uv_os_tmpdir()* is not thread safe.

1.9.0 新版功能.

int **uv_os_get_passwd**(*uv_passwd_t** *pwd*)
Gets a subset of the password file entry for the current effective uid (not the real uid). The populated data includes the username, euid, gid, shell, and home directory. On non-Windows systems, all data comes from getpwuid_r(3). On Windows, uid and gid are set to -1 and have no meaning, and shell is *NULL*. After successfully calling this function, the memory allocated to *pwd* needs to be freed with `uv_os_free_passwd()`.

1.9.0 新版功能.

void **uv_os_free_passwd**(*uv_passwd_t** *pwd*)
Frees the *pwd* memory previously allocated with `uv_os_get_passwd()`.

1.9.0 新版功能.

uint64_t **uv_get_free_memory**(void)
Gets memory information (in bytes).

uint64_t **uv_get_total_memory**(void)
Gets memory information (in bytes).

uint64_t **uv_hrtime**(void)
Returns the current high-resolution real time. This is expressed in nanoseconds. It is relative to an arbitrary time in the past. It is not related to the time of day and therefore not subject to clock drift. The primary use is for measuring performance between intervals.

> **注解：** Not every platform can support nanosecond resolution; however, this value will always be in nanoseconds.

void **uv_print_all_handles**(*uv_loop_t** *loop*, FILE* *stream*)
Prints all handles associated with the given *loop* to the given *stream*.

Example:

```
uv_print_all_handles(uv_default_loop(), stderr);
/*
[--I] signal   0x1a25ea8
[-AI] async    0x1a25cf0
[R--] idle     0x1a7a8c8
*/
```

The format is *[flags] handle-type handle-address*. For *flags*:

- *R* is printed for a handle that is referenced
- *A* is printed for a handle that is active
- *I* is printed for a handle that is internal

> **警告：** This function is meant for ad hoc debugging, there is no API/ABI stability guarantees.

1.8.0 新版功能.

void **uv_print_active_handles**(*uv_loop_t* loop*, FILE* *stream*)
This is the same as `uv_print_all_handles()` except only active handles are printed.

> **警告：** This function is meant for ad hoc debugging, there is no API/ABI stability guarantees.

1.8.0 新版功能.

int **uv_os_getenv**(const char* *name*, char* *buffer*, size_t* *size*)
Retrieves the environment variable specified by *name*, copies its value into *buffer*, and sets *size* to the string length of the value. When calling this function, *size* must be set to the amount of storage available in *buffer*, including the null terminator. If the environment variable exceeds the storage available in *buffer*, *UV_ENOBUFS* is returned, and *size* is set to the amount of storage required to hold the value. If no matching environment variable exists, *UV_ENOENT* is returned.

> **警告：** This function is not thread safe.

1.12.0 新版功能.

int **uv_os_setenv**(const char* *name*, const char* *value*)
Creates or updates the environment variable specified by *name* with *value*.

> **警告：** This function is not thread safe.

1.12.0 新版功能.

int **uv_os_unsetenv**(const char* *name*)
Deletes the environment variable specified by *name*. If no such environment variable exists, this function returns successfully.

> **警告：** This function is not thread safe.

1.12.0 新版功能.

int **uv_os_gethostname**(char* *buffer*, size_t* *size*)
Returns the hostname as a null-terminated string in *buffer*, and sets *size* to the string length of the hostname. When calling this function, *size* must be set to the amount of storage available in *buffer*, including the null terminator. If the hostname exceeds the storage available in *buffer*, *UV_ENOBUFS* is returned, and *size* is set to the amount of storage required to hold the value.

1.12.0 新版功能.

在 1.26.0 版更改: *UV_MAXHOSTNAMESIZE* is available and represents the maximum *buffer* size required to store a hostname and terminating *nul* character.

int **uv_os_getpriority**(*uv_pid_t pid*, int* *priority*)
Retrieves the scheduling priority of the process specified by *pid*. The returned value of *priority* is between -20 (high priority) and 19 (low priority).

注解: On Windows, the returned priority will equal one of the *UV_PRIORITY* constants.

1.23.0 新版功能.

int **uv_os_setpriority**(*uv_pid_t pid*, int *priority*)
Sets the scheduling priority of the process specified by *pid*. The *priority* value range is between -20 (high priority) and 19 (low priority). The constants *UV_PRIORITY_LOW*, *UV_PRIORITY_BELOW_NORMAL*, *UV_PRIORITY_NORMAL*, *UV_PRIORITY_ABOVE_NORMAL*, *UV_PRIORITY_HIGH*, and *UV_PRIORITY_HIGHEST* are also provided for convenience.

注解: On Windows, this function utilizes *SetPriorityClass()*. The *priority* argument is mapped to a Windows priority class. When retrieving the process priority, the result will equal one of the *UV_PRIORITY* constants, and not necessarily the exact value of *priority*.

注解: On Windows, setting *PRIORITY_HIGHEST* will only work for elevated user, for others it will be silently reduced to *PRIORITY_HIGH*.

1.23.0 新版功能.

int **uv_os_uname**(*uv_utsname_t** *buffer*)
Retrieves system information in *buffer*. The populated data includes the operating system name, release, version, and machine. On non-Windows systems, *uv_os_uname()* is a thin wrapper around uname(3). Returns zero on success, and a non-zero error value otherwise.

1.25.0 新版功能.

3.3 用户指南

警告: 这份指南的内容最近并入libuv文档了， 并且它没有经过完全的审查。 如果你发现了错误请给官方libuv项目提 issue，或更好的话，开启一个pull request！ 如果是翻译问题，则到翻译项目提出。

3.3.1 Introduction

This 'book' is a small set of tutorials about using libuv as a high performance evented I/O library which offers the same API on Windows and Unix.

It is meant to cover the main areas of libuv, but is not a comprehensive reference discussing every function and data structure. The official libuv documentation may be consulted for full details.

This book is still a work in progress, so sections may be incomplete, but I hope you will enjoy it as it grows.

Who this book is for

If you are reading this book, you are either:

1. a systems programmer, creating low-level programs such as daemons or network services and clients. You have found that the event loop approach is well suited for your application and decided to use libuv.
2. a node.js module writer, who wants to wrap platform APIs written in C or C++ with a set of (a)synchronous APIs that are exposed to JavaScript. You will use libuv purely in the context of node.js. For this you will require some other resources as the book does not cover parts specific to v8/node.js.

This book assumes that you are comfortable with the C programming language.

Background

The node.js project began in 2009 as a JavaScript environment decoupled from the browser. Using Google's V8 and Marc Lehmann's libev, node.js combined a model of I/O – evented – with a language that was well suited to the style of programming; due to the way it had been shaped by browsers. As node.js grew in popularity, it was important to make it work on Windows, but libev ran only on Unix. The Windows equivalent of kernel event notification mechanisms like kqueue or (e)poll is IOCP. libuv was an abstraction around libev or IOCP depending on the platform, providing users an API based on libev. In the node-v0.9.0 version of libuv libev was removed.

Since then libuv has continued to mature and become a high quality standalone library for system programming. Users outside of node.js include Mozilla's Rust programming language, and a variety of language bindings.

This book and the code is based on libuv version v1.3.0.

Code

All the code from this book is included as part of the source of the book on Github. Clone/Download the book, then build libuv:

```
cd libuv
./autogen.sh
./configure
make
```

There is no need to `make install`. To build the examples run `make` in the `code/` directory.

3.3.2 Basics of libuv

libuv enforces an **asynchronous**, **event-driven** style of programming. Its core job is to provide an event loop and callback based notifications of I/O and other activities. libuv offers core utilities like timers, non-blocking networking support, asynchronous file system access, child processes and more.

Event loops

In event-driven programming, an application expresses interest in certain events and respond to them when they occur. The responsibility of gathering events from the operating system or monitoring other sources of events is handled by libuv, and the user can register callbacks to be invoked when an event occurs. The event-loop usually keeps running *forever*. In pseudocode:

```
while there are still events to process:
    e = get the next event
    if there is a callback associated with e:
        call the callback
```

Some examples of events are:

- File is ready for writing
- A socket has data ready to be read
- A timer has timed out

This event loop is encapsulated by `uv_run()` – the end-all function when using libuv.

The most common activity of systems programs is to deal with input and output, rather than a lot of number-crunching. The problem with using conventional input/output functions (`read`, `fprintf`, etc.) is that they are **blocking**. The actual write to a hard disk or reading from a network, takes a disproportionately long time compared to the speed of the processor. The functions don't return until the task is done, so that your program is doing nothing. For programs which require high performance this is a major roadblock as other activities and other I/O operations are kept waiting.

One of the standard solutions is to use threads. Each blocking I/O operation is started in a separate thread (or in a thread pool). When the blocking function gets invoked in the thread, the processor can schedule another thread to run, which actually needs the CPU.

The approach followed by libuv uses another style, which is the **asynchronous, non-blocking** style. Most modern operating systems provide event notification subsystems. For example, a normal `read` call on a socket would block until the sender actually sent something. Instead, the application can request the operating system to watch the socket and put an event notification in the queue. The application can inspect the events at its convenience (perhaps doing some number crunching before to use the processor to the maximum) and grab the data. It is **asynchronous** because the application expressed interest at one point, then used the data at another point (in time and space). It is **non-blocking** because the application process was free to do other tasks. This fits in well with libuv's event-loop approach, since the operating system events can be treated as just another libuv event. The non-blocking ensures that other events can continue to be handled as fast as they come in[1].

注解: How the I/O is run in the background is not of our concern, but due to the way our computer hardware works, with the thread as the basic unit of the processor, libuv and OSes will usually run background/worker threads and/or polling to perform tasks in a non-blocking manner.

Bert Belder, one of the libuv core developers has a small video explaining the architecture of libuv and its background. If you have no prior experience with either libuv or libev, it is a quick, useful watch.

libuv's event loop is explained in more detail in the documentation.

Hello World

With the basics out of the way, lets write our first libuv program. It does nothing, except start a loop which will exit immediately.

helloworld/main.c

[1] Depending on the capacity of the hardware of course.

```c
#include <stdio.h>
#include <stdlib.h>
#include <uv.h>

int main() {
    uv_loop_t *loop = malloc(sizeof(uv_loop_t));
    uv_loop_init(loop);

    printf("Now quitting.\n");
    uv_run(loop, UV_RUN_DEFAULT);

    uv_loop_close(loop);
    free(loop);
    return 0;
}
```

This program quits immediately because it has no events to process. A libuv event loop has to be told to watch out for events using the various API functions.

Starting with libuv v1.0, users should allocate the memory for the loops before initializing it with `uv_loop_init(uv_loop_t *)`. This allows you to plug in custom memory management. Remember to de-initialize the loop using `uv_loop_close(uv_loop_t *)` and then delete the storage. The examples never close loops since the program quits after the loop ends and the system will reclaim memory. Production grade projects, especially long running systems programs, should take care to release correctly.

Default loop

A default loop is provided by libuv and can be accessed using `uv_default_loop()`. You should use this loop if you only want a single loop.

注解： node.js uses the default loop as its main loop. If you are writing bindings you should be aware of this.

Error handling

Initialization functions or synchronous functions which may fail return a negative number on error. Async functions that may fail will pass a status parameter to their callbacks. The error messages are defined as `UV_E*` constants.

You can use the `uv_strerror(int)` and `uv_err_name(int)` functions to get a `const char *` describing the error or the error name respectively.

I/O read callbacks (such as for files and sockets) are passed a parameter `nread`. If `nread` is less than 0, there was an error (UV_EOF is the end of file error, which you may want to handle differently).

Handles and Requests

libuv works by the user expressing interest in particular events. This is usually done by creating a **handle** to an I/O device, timer or process. Handles are opaque structs named as `uv_TYPE_t` where type signifies what the handle is used for.

libuv watchers

```
  UV_REQ_TYPE_PRIVATE
  UV_REQ_TYPE_MAX
} uv_req_type;


/* Handle types. */
typedef struct uv_loop_s uv_loop_t;
typedef struct uv_handle_s uv_handle_t;
typedef struct uv_stream_s uv_stream_t;
typedef struct uv_tcp_s uv_tcp_t;
typedef struct uv_udp_s uv_udp_t;
typedef struct uv_pipe_s uv_pipe_t;
typedef struct uv_tty_s uv_tty_t;
typedef struct uv_poll_s uv_poll_t;
typedef struct uv_timer_s uv_timer_t;
typedef struct uv_prepare_s uv_prepare_t;
typedef struct uv_check_s uv_check_t;
typedef struct uv_idle_s uv_idle_t;
typedef struct uv_async_s uv_async_t;
typedef struct uv_process_s uv_process_t;
typedef struct uv_fs_event_s uv_fs_event_t;
typedef struct uv_fs_poll_s uv_fs_poll_t;
typedef struct uv_signal_s uv_signal_t;

/* Request types. */
typedef struct uv_req_s uv_req_t;
typedef struct uv_getaddrinfo_s uv_getaddrinfo_t;
typedef struct uv_getnameinfo_s uv_getnameinfo_t;
typedef struct uv_shutdown_s uv_shutdown_t;
typedef struct uv_write_s uv_write_t;
typedef struct uv_connect_s uv_connect_t;
typedef struct uv_udp_send_s uv_udp_send_t;
typedef struct uv_fs_s uv_fs_t;
typedef struct uv_work_s uv_work_t;
```

Handles represent long-lived objects. Async operations on such handles are identified using **requests**. A request is short-lived (usually used across only one callback) and usually indicates one I/O operation on a handle. Requests are used to preserve context between the initiation and the callback of individual actions. For example, an UDP socket is represented by a `uv_udp_t`, while individual writes to the socket use a `uv_udp_send_t` structure that is passed to the callback after the write is done.

Handles are setup by a corresponding:

```
uv_TYPE_init(uv_loop_t *, uv_TYPE_t *)
```

function.

Callbacks are functions which are called by libuv whenever an event the watcher is interested in has taken place. Application specific logic will usually be implemented in the callback. For example, an IO watcher's callback will receive the data read from a file, a timer callback will be triggered on timeout and so on.

Idling

Here is an example of using an idle handle. The callback is called once on every turn of the event loop. A use case for idle handles is discussed in *Utilities*. Let us use an idle watcher to look at the watcher life cycle and see how

`uv_run()` will now block because a watcher is present. The idle watcher is stopped when the count is reached and `uv_run()` exits since no event watchers are active.

idle-basic/main.c

```
#include <stdio.h>
#include <uv.h>

int64_t counter = 0;

void wait_for_a_while(uv_idle_t* handle) {
    counter++;

    if (counter >= 10e6)
        uv_idle_stop(handle);
}

int main() {
    uv_idle_t idler;

    uv_idle_init(uv_default_loop(), &idler);
    uv_idle_start(&idler, wait_for_a_while);

    printf("Idling...\n");
    uv_run(uv_default_loop(), UV_RUN_DEFAULT);

    uv_loop_close(uv_default_loop());
    return 0;
}
```

Storing context

In callback based programming style you'll often want to pass some 'context' – application specific information – between the call site and the callback. All handles and requests have a `void* data` member which you can set to the context and cast back in the callback. This is a common pattern used throughout the C library ecosystem. In addition `uv_loop_t` also has a similar data member.

3.3.3 Filesystem

Simple filesystem read/write is achieved using the `uv_fs_*` functions and the `uv_fs_t` struct.

注解: The libuv filesystem operations are different from *socket operations*. Socket operations use the non-blocking operations provided by the operating system. Filesystem operations use blocking functions internally, but invoke these functions in a thread pool and notify watchers registered with the event loop when application interaction is required.

All filesystem functions have two forms - *synchronous* and *asynchronous*.

The *synchronous* forms automatically get called (and **block**) if the callback is null. The return value of functions is a *libuv error code*. This is usually only useful for synchronous calls. The *asynchronous* form is called when a callback is passed and the return value is 0.

Reading/Writing files

A file descriptor is obtained using

```
int uv_fs_open(uv_loop_t* loop, uv_fs_t* req, const char* path, int flags, int mode,␣
↪uv_fs_cb cb)
```

`flags` and `mode` are standard Unix flags. libuv takes care of converting to the appropriate Windows flags.

File descriptors are closed using

```
int uv_fs_close(uv_loop_t* loop, uv_fs_t* req, uv_file file, uv_fs_cb cb)
```

Filesystem operation callbacks have the signature:

```
void callback(uv_fs_t* req);
```

Let's see a simple implementation of `cat`. We start with registering a callback for when the file is opened:

uvcat/main.c - opening a file

```
void on_open(uv_fs_t *req) {
    // The request passed to the callback is the same as the one the call setup
    // function was passed.
    assert(req == &open_req);
    if (req->result >= 0) {
        iov = uv_buf_init(buffer, sizeof(buffer));
        uv_fs_read(uv_default_loop(), &read_req, req->result,
                   &iov, 1, -1, on_read);
    }
    else {
        fprintf(stderr, "error opening file: %s\n", uv_strerror((int)req->result));
    }
}
```

The `result` field of a `uv_fs_t` is the file descriptor in case of the `uv_fs_open` callback. If the file is successfully opened, we start reading it.

uvcat/main.c - read callback

```
void on_read(uv_fs_t *req) {
    if (req->result < 0) {
        fprintf(stderr, "Read error: %s\n", uv_strerror(req->result));
    }
    else if (req->result == 0) {
        uv_fs_t close_req;
        // synchronous
        uv_fs_close(uv_default_loop(), &close_req, open_req.result, NULL);
    }
    else if (req->result > 0) {
        iov.len = req->result;
        uv_fs_write(uv_default_loop(), &write_req, 1, &iov, 1, -1, on_write);
    }
}

```

In the case of a read call, you should pass an *initialized* buffer which will be filled with data before the read callback is triggered. The `uv_fs_*` operations map almost directly to certain POSIX functions, so EOF is indicated in this case by `result` being 0. In the case of streams or pipes, the `UV_EOF` constant would have been passed as a status instead.

Here you see a common pattern when writing asynchronous programs. The `uv_fs_close()` call is performed synchronously. *Usually tasks which are one-off, or are done as part of the startup or shutdown stage are performed synchronously, since we are interested in fast I/O when the program is going about its primary task and dealing with multiple I/O sources*. For solo tasks the performance difference usually is negligible and may lead to simpler code.

Filesystem writing is similarly simple using `uv_fs_write()`. *Your callback will be triggered after the write is complete*. In our case the callback simply drives the next read. Thus read and write proceed in lockstep via callbacks.

uvcat/main.c - write callback

```
1
2  void on_write(uv_fs_t *req) {
3      if (req->result < 0) {
4          fprintf(stderr, "Write error: %s\n", uv_strerror((int)req->result));
5      }
6      else {
7          uv_fs_read(uv_default_loop(), &read_req, open_req.result, &iov, 1, -1, on_
   ↪read);
8      }
9  }
```

警告： Due to the way filesystems and disk drives are configured for performance, a write that 'succeeds' may not be committed to disk yet.

We set the dominos rolling in `main()`:

uvcat/main.c

```
1  int main(int argc, char **argv) {
2      uv_fs_open(uv_default_loop(), &open_req, argv[1], O_RDONLY, 0, on_open);
3      uv_run(uv_default_loop(), UV_RUN_DEFAULT);
4
5      uv_fs_req_cleanup(&open_req);
6      uv_fs_req_cleanup(&read_req);
7      uv_fs_req_cleanup(&write_req);
8      return 0;
9  }
```

警告： The `uv_fs_req_cleanup()` function must always be called on filesystem requests to free internal memory allocations in libuv.

Filesystem operations

All the standard filesystem operations like `unlink`, `rmdir`, `stat` are supported asynchronously and have intuitive argument order. They follow the same patterns as the read/write/open calls, returning the result in the `uv_fs_t`.

`result` field. The full list:

Filesystem operations

```
UV_EXTERN char** uv_setup_args(int argc, char** argv);
UV_EXTERN int uv_get_process_title(char* buffer, size_t size);
UV_EXTERN int uv_set_process_title(const char* title);
UV_EXTERN int uv_resident_set_memory(size_t* rss);
UV_EXTERN int uv_uptime(double* uptime);
UV_EXTERN uv_os_fd_t uv_get_osfhandle(int fd);
UV_EXTERN int uv_open_osfhandle(uv_os_fd_t os_fd);

typedef struct {
  long tv_sec;
  long tv_usec;
} uv_timeval_t;

typedef struct {
   uv_timeval_t ru_utime; /* user CPU time used */
   uv_timeval_t ru_stime; /* system CPU time used */
   uint64_t ru_maxrss;    /* maximum resident set size */
   uint64_t ru_ixrss;     /* integral shared memory size */
   uint64_t ru_idrss;     /* integral unshared data size */
   uint64_t ru_isrss;     /* integral unshared stack size */
   uint64_t ru_minflt;    /* page reclaims (soft page faults) */
   uint64_t ru_majflt;    /* page faults (hard page faults) */
   uint64_t ru_nswap;     /* swaps */
   uint64_t ru_inblock;   /* block input operations */
   uint64_t ru_oublock;   /* block output operations */
   uint64_t ru_msgsnd;    /* IPC messages sent */
   uint64_t ru_msgrcv;    /* IPC messages received */
   uint64_t ru_nsignals;  /* signals received */
   uint64_t ru_nvcsw;     /* voluntary context switches */
   uint64_t ru_nivcsw;    /* involuntary context switches */
} uv_rusage_t;

UV_EXTERN int uv_getrusage(uv_rusage_t* rusage);

UV_EXTERN int uv_os_homedir(char* buffer, size_t* size);
UV_EXTERN int uv_os_tmpdir(char* buffer, size_t* size);
UV_EXTERN int uv_os_get_passwd(uv_passwd_t* pwd);
UV_EXTERN void uv_os_free_passwd(uv_passwd_t* pwd);
UV_EXTERN uv_pid_t uv_os_getpid(void);
UV_EXTERN uv_pid_t uv_os_getppid(void);

#define UV_PRIORITY_LOW 19
#define UV_PRIORITY_BELOW_NORMAL 10
#define UV_PRIORITY_NORMAL 0
#define UV_PRIORITY_ABOVE_NORMAL -7
#define UV_PRIORITY_HIGH -14
#define UV_PRIORITY_HIGHEST -20

UV_EXTERN int uv_os_getpriority(uv_pid_t pid, int* priority);
UV_EXTERN int uv_os_setpriority(uv_pid_t pid, int priority);

UV_EXTERN int uv_cpu_info(uv_cpu_info_t** cpu_infos, int* count);
```

(continues on next page)

(续上页)

```
UV_EXTERN void uv_free_cpu_info(uv_cpu_info_t* cpu_infos, int count);

UV_EXTERN int uv_interface_addresses(uv_interface_address_t** addresses,
                                     int* count);
UV_EXTERN void uv_free_interface_addresses(uv_interface_address_t* addresses,
                                           int count);

UV_EXTERN int uv_os_getenv(const char* name, char* buffer, size_t* size);
UV_EXTERN int uv_os_setenv(const char* name, const char* value);
UV_EXTERN int uv_os_unsetenv(const char* name);

#ifdef MAXHOSTNAMELEN
# define UV_MAXHOSTNAMESIZE (MAXHOSTNAMELEN + 1)
#else
  /*
    Fallback for the maximum hostname size, including the null terminator. The
    Windows gethostname() documentation states that 256 bytes will always be
    large enough to hold the null-terminated hostname.
  */
# define UV_MAXHOSTNAMESIZE 256
#endif

UV_EXTERN int uv_os_gethostname(char* buffer, size_t* size);

UV_EXTERN int uv_os_uname(uv_utsname_t* buffer);


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,
```

(continues on next page)

(续上页)

```
  UV_FS_FCHOWN,
  UV_FS_REALPATH,
  UV_FS_COPYFILE,
  UV_FS_LCHOWN
```

Buffers and Streams

The basic I/O handle in libuv is the stream (`uv_stream_t`). TCP sockets, UDP sockets, and pipes for file I/O and IPC are all treated as stream subclasses.

Streams are initialized using custom functions for each subclass, then operated upon using

```
int uv_read_start(uv_stream_t*, uv_alloc_cb alloc_cb, uv_read_cb read_cb);
int uv_read_stop(uv_stream_t*);
int uv_write(uv_write_t* req, uv_stream_t* handle,
             const uv_buf_t bufs[], unsigned int nbufs, uv_write_cb cb);
```

The stream based functions are simpler to use than the filesystem ones and libuv will automatically keep reading from a stream when `uv_read_start()` is called once, until `uv_read_stop()` is called.

The discrete unit of data is the buffer – `uv_buf_t`. This is simply a collection of a pointer to bytes (`uv_buf_t.base`) and the length (`uv_buf_t.len`). The `uv_buf_t` is lightweight and passed around by value. What does require management is the actual bytes, which have to be allocated and freed by the application.

> **错误：** THIS PROGRAM DOES NOT ALWAYS WORK, NEED SOMETHING BETTER**

To demonstrate streams we will need to use `uv_pipe_t`. This allows streaming local files[2]. Here is a simple tee utility using libuv. Doing all operations asynchronously shows the power of evented I/O. The two writes won't block each other, but we have to be careful to copy over the buffer data to ensure we don't free a buffer until it has been written.

The program is to be executed as:

```
./uvtee <output_file>
```

We start off opening pipes on the files we require. libuv pipes to a file are opened as bidirectional by default.

uvtee/main.c - read on pipes

```
1
2  int main(int argc, char **argv) {
3      loop = uv_default_loop();
4
5      uv_pipe_init(loop, &stdin_pipe, 0);
6      uv_pipe_open(&stdin_pipe, 0);
7
8      uv_pipe_init(loop, &stdout_pipe, 0);
9      uv_pipe_open(&stdout_pipe, 1);
10
11     uv_fs_t file_req;
```

(continues on next page)

[2] see *Pipes*

(续上页)

```
    int fd = uv_fs_open(loop, &file_req, argv[1], O_CREAT | O_RDWR, 0644, NULL);
    uv_pipe_init(loop, &file_pipe, 0);
    uv_pipe_open(&file_pipe, fd);

    uv_read_start((uv_stream_t*)&stdin_pipe, alloc_buffer, read_stdin);

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

The third argument of `uv_pipe_init()` should be set to 1 for IPC using named pipes. This is covered in *Processes*. The `uv_pipe_open()` call associates the pipe with the file descriptor, in this case `0` (standard input).

We start monitoring `stdin`. The `alloc_buffer` callback is invoked as new buffers are required to hold incoming data. `read_stdin` will be called with these buffers.

uvtee/main.c - reading buffers

```
void alloc_buffer(uv_handle_t *handle, size_t suggested_size, uv_buf_t *buf) {
    *buf = uv_buf_init((char*) malloc(suggested_size), suggested_size);
}

void read_stdin(uv_stream_t *stream, ssize_t nread, const uv_buf_t *buf) {
    if (nread < 0){
        if (nread == UV_EOF){
            // end of file
            uv_close((uv_handle_t *)&stdin_pipe, NULL);
            uv_close((uv_handle_t *)&stdout_pipe, NULL);
            uv_close((uv_handle_t *)&file_pipe, NULL);
        }
    } else if (nread > 0) {
        write_data((uv_stream_t *)&stdout_pipe, nread, *buf, on_stdout_write);
        write_data((uv_stream_t *)&file_pipe, nread, *buf, on_file_write);
    }

    // OK to free buffer as write_data copies it.
    if (buf->base)
        free(buf->base);
}
```

The standard `malloc` is sufficient here, but you can use any memory allocation scheme. For example, node.js uses its own slab allocator which associates buffers with V8 objects.

The read callback `nread` parameter is less than 0 on any error. This error might be EOF, in which case we close all the streams, using the generic close function `uv_close()` which deals with the handle based on its internal type. Otherwise `nread` is a non-negative number and we can attempt to write that many bytes to the output streams. Finally remember that buffer allocation and deallocation is application responsibility, so we free the data.

The allocation callback may return a buffer with length zero if it fails to allocate memory. In this case, the read callback is invoked with error UV_ENOBUFS. libuv will continue to attempt to read the stream though, so you must explicitly call `uv_close()` if you want to stop when allocation fails.

The read callback may be called with `nread = 0`, indicating that at this point there is nothing to be read. Most applications will just ignore this.

uvtee/main.c - Write to pipe

```
typedef struct {
    uv_write_t req;
    uv_buf_t buf;
} write_req_t;

void free_write_req(uv_write_t *req) {
    write_req_t *wr = (write_req_t*) req;
    free(wr->buf.base);
    free(wr);
}

void on_stdout_write(uv_write_t *req, int status) {
    free_write_req(req);
}

void on_file_write(uv_write_t *req, int status) {
    free_write_req(req);
}

void write_data(uv_stream_t *dest, size_t size, uv_buf_t buf, uv_write_cb cb) {
    write_req_t *req = (write_req_t*) malloc(sizeof(write_req_t));
    req->buf = uv_buf_init((char*) malloc(size), size);
    memcpy(req->buf.base, buf.base, size);
    uv_write((uv_write_t*) req, (uv_stream_t*)dest, &req->buf, 1, cb);
}
```

`write_data()` makes a copy of the buffer obtained from read. This buffer does not get passed through to the write callback trigged on write completion. To get around this we wrap a write request and a buffer in `write_req_t` and unwrap it in the callbacks. We make a copy so we can free the two buffers from the two calls to `write_data` independently of each other. While acceptable for a demo program like this, you'll probably want smarter memory management, like reference counted buffers or a pool of buffers in any major application.

警告： If your program is meant to be used with other programs it may knowingly or unknowingly be writing to a pipe. This makes it susceptible to aborting on receiving a SIGPIPE. It is a good idea to insert:

```
signal(SIGPIPE, SIG_IGN)
```

in the initialization stages of your application.

File change events

All modern operating systems provide APIs to put watches on individual files or directories and be informed when the files are modified. libuv wraps common file change notification libraries[1]. This is one of the more inconsistent parts of libuv. File change notification systems are themselves extremely varied across platforms so getting everything working everywhere is difficult. To demonstrate, I'm going to build a simple utility which runs a command whenever any of the watched files change:

```
./onchange <command> <file1> [file2] ...
```

The file change notification is started using `uv_fs_event_init()`:

[1] inotify on Linux, FSEvents on Darwin, kqueue on BSDs, ReadDirectoryChangesW on Windows, event ports on Solaris, unsupported on Cygwin

onchange/main.c - The setup

```
int main(int argc, char **argv) {
    if (argc <= 2) {
        fprintf(stderr, "Usage: %s <command> <file1> [file2 ...]\n", argv[0]);
        return 1;
    }

    loop = uv_default_loop();
    command = argv[1];

    while (argc-- > 2) {
        fprintf(stderr, "Adding watch on %s\n", argv[argc]);
        uv_fs_event_t *fs_event_req = malloc(sizeof(uv_fs_event_t));
        uv_fs_event_init(loop, fs_event_req);
        // The recursive flag watches subdirectories too.
        uv_fs_event_start(fs_event_req, run_command, argv[argc], UV_FS_EVENT_
↪RECURSIVE);
    }

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

The third argument is the actual file or directory to monitor. The last argument, `flags`, can be:

```
                            uv_fs_t* req,
                               uv_fs_t* req,
                               uv_fs_cb cb);
```

`UV_FS_EVENT_WATCH_ENTRY` and `UV_FS_EVENT_STAT` don't do anything (yet). `UV_FS_EVENT_RECURSIVE` will start watching subdirectories as well on supported platforms.

The callback will receive the following arguments:

1. `uv_fs_event_t *handle` - The handle. The `path` field of the handle is the file on which the watch was set.
2. `const char *filename` - If a directory is being monitored, this is the file which was changed. Only non-`null` on Linux and Windows. May be `null` even on those platforms.
3. **`int flags` - one of `UV_RENAME` or `UV_CHANGE`, or a bitwise OR of** both.
4. `int status` - Currently 0.

In our example we simply print the arguments and run the command using `system()`.

onchange/main.c - file change notification callback

```
void run_command(uv_fs_event_t *handle, const char *filename, int events, int status)
↪{
    char path[1024];
    size_t size = 1023;
    // Does not handle error if path is longer than 1023.
    uv_fs_event_getpath(handle, path, &size);
    path[size] = '\0';

    fprintf(stderr, "Change detected in %s: ", path);
```

(continues on next page)

(续上页)

```
    if (events & UV_RENAME)
        fprintf(stderr, "renamed");
    if (events & UV_CHANGE)
        fprintf(stderr, "changed");

    fprintf(stderr, " %s\n", filename ? filename : "");
    system(command);
}
```

3.3.4 Networking

Networking in libuv is not much different from directly using the BSD socket interface, some things are easier, all are non-blocking, but the concepts stay the same. In addition libuv offers utility functions to abstract the annoying, repetitive and low-level tasks like setting up sockets using the BSD socket structures, DNS lookup, and tweaking various socket parameters.

The `uv_tcp_t` and `uv_udp_t` structures are used for network I/O.

注解： The code samples in this chapter exist to show certain libuv APIs. They are not examples of good quality code. They leak memory and don't always close connections properly.

TCP

TCP is a connection oriented, stream protocol and is therefore based on the libuv streams infrastructure.

Server

Server sockets proceed by:

1. `uv_tcp_init` the TCP handle.
2. `uv_tcp_bind` it.
3. Call `uv_listen` on the handle to have a callback invoked whenever a new connection is established by a client.
4. Use `uv_accept` to accept the connection.
5. Use *stream operations* to communicate with the client.

Here is a simple echo server

tcp-echo-server/main.c - The listen socket

```
        uv_close((uv_handle_t*) client, on_close);
    }
}

int main() {
```

(continues on next page)

(续上页)

```
    loop = uv_default_loop();

    uv_tcp_t server;
    uv_tcp_init(loop, &server);

    uv_ip4_addr("0.0.0.0", DEFAULT_PORT, &addr);

    uv_tcp_bind(&server, (const struct sockaddr*)&addr, 0);
    int r = uv_listen((uv_stream_t*) &server, DEFAULT_BACKLOG, on_new_connection);
    if (r) {
        fprintf(stderr, "Listen error %s\n", uv_strerror(r));
        return 1;
    }
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

You can see the utility function `uv_ip4_addr` being used to convert from a human readable IP address, port pair to the sockaddr_in structure required by the BSD socket APIs. The reverse can be obtained using `uv_ip4_name`.

注解: There are `uv_ip6_*` analogues for the ip4 functions.

Most of the setup functions are synchronous since they are CPU-bound. `uv_listen` is where we return to libuv's callback style. The second arguments is the backlog queue – the maximum length of queued connections.

When a connection is initiated by clients, the callback is required to set up a handle for the client socket and associate the handle using `uv_accept`. In this case we also establish interest in reading from this stream.

tcp-echo-server/main.c - Accepting the client

```

    free(buf->base);
}

void on_new_connection(uv_stream_t *server, int status) {
    if (status < 0) {
        fprintf(stderr, "New connection error %s\n", uv_strerror(status));
        // error!
        return;
    }

    uv_tcp_t *client = (uv_tcp_t*) malloc(sizeof(uv_tcp_t));
    uv_tcp_init(loop, client);
    if (uv_accept(server, (uv_stream_t*) client) == 0) {
        uv_read_start((uv_stream_t*) client, alloc_buffer, echo_read);
    }
```

The remaining set of functions is very similar to the streams example and can be found in the code. Just remember to call `uv_close` when the socket isn't required. This can be done even in the `uv_listen` callback if you are not interested in accepting the connection.

Client

Where you do bind/listen/accept on the server, on the client side it's simply a matter of calling `uv_tcp_connect`. The same `uv_connect_cb` style callback of `uv_listen` is used by `uv_tcp_connect`. Try:

```
uv_tcp_t* socket = (uv_tcp_t*)malloc(sizeof(uv_tcp_t));
uv_tcp_init(loop, socket);

uv_connect_t* connect = (uv_connect_t*)malloc(sizeof(uv_connect_t));

struct sockaddr_in dest;
uv_ip4_addr("127.0.0.1", 80, &dest);

uv_tcp_connect(connect, socket, (const struct sockaddr*)&dest, on_connect);
```

where `on_connect` will be called after the connection is established. The callback receives the `uv_connect_t` struct, which has a member `.handle` pointing to the socket.

UDP

The User Datagram Protocol offers connectionless, unreliable network communication. Hence libuv doesn't offer a stream. Instead libuv provides non-blocking UDP support via the *uv_udp_t* handle (for receiving) and *uv_udp_send_t* request (for sending) and related functions. That said, the actual API for reading/writing is very similar to normal stream reads. To look at how UDP can be used, the example shows the first stage of obtaining an IP address from a DHCP server – DHCP Discover.

注解: You will have to run *udp-dhcp* as **root** since it uses well known port numbers below 1024.

udp-dhcp/main.c - Setup and send UDP packets

```
1
2  uv_loop_t *loop;
3  uv_udp_t send_socket;
4  uv_udp_t recv_socket;
5
6  int main() {
7      loop = uv_default_loop();
8
9      uv_udp_init(loop, &recv_socket);
10     struct sockaddr_in recv_addr;
11     uv_ip4_addr("0.0.0.0", 68, &recv_addr);
12     uv_udp_bind(&recv_socket, (const struct sockaddr *)&recv_addr, UV_UDP_REUSEADDR);
13     uv_udp_recv_start(&recv_socket, alloc_buffer, on_read);
14
15     uv_udp_init(loop, &send_socket);
16     struct sockaddr_in broadcast_addr;
17     uv_ip4_addr("0.0.0.0", 0, &broadcast_addr);
18     uv_udp_bind(&send_socket, (const struct sockaddr *)&broadcast_addr, 0);
19     uv_udp_set_broadcast(&send_socket, 1);
20
21     uv_udp_send_t send_req;
22     uv_buf_t discover_msg = make_discover_msg();
```

(continues on next page)

(续上页)

```
23
24      struct sockaddr_in send_addr;
25      uv_ip4_addr("255.255.255.255", 67, &send_addr);
26      uv_udp_send(&send_req, &send_socket, &discover_msg, 1, (const struct sockaddr *)&
    ↪send_addr, on_send);
27
28      return uv_run(loop, UV_RUN_DEFAULT);
29  }
```

注解: The IP address `0.0.0.0` is used to bind to all interfaces. The IP address `255.255.255.255` is a broadcast address meaning that packets will be sent to all interfaces on the subnet. port `0` means that the OS randomly assigns a port.

First we setup the receiving socket to bind on all interfaces on port 68 (DHCP client) and start a read on it. This will read back responses from any DHCP server that replies. We use the UV_UDP_REUSEADDR flag to play nice with any other system DHCP clients that are running on this computer on the same port. Then we setup a similar send socket and use `uv_udp_send` to send a *broadcast message* on port 67 (DHCP server).

It is **necessary** to set the broadcast flag, otherwise you will get an `EACCES` error[1]. The exact message being sent is not relevant to this book and you can study the code if you are interested. As usual the read and write callbacks will receive a status code of < 0 if something went wrong.

Since UDP sockets are not connected to a particular peer, the read callback receives an extra parameter about the sender of the packet.

`nread` may be zero if there is no more data to be read. If `addr` is NULL, it indicates there is nothing to read (the callback shouldn't do anything), if not NULL, it indicates that an empty datagram was received from the host at `addr`. The `flags` parameter may be `UV_UDP_PARTIAL` if the buffer provided by your allocator was not large enough to hold the data. *In this case the OS will discard the data that could not fit* (That's UDP for you!).

udp-dhcp/main.c - Reading packets

```
1   void on_read(uv_udp_t *req, ssize_t nread, const uv_buf_t *buf, const struct sockaddr␣
    ↪*addr, unsigned flags) {
2       if (nread < 0) {
3           fprintf(stderr, "Read error %s\n", uv_err_name(nread));
4           uv_close((uv_handle_t*) req, NULL);
5           free(buf->base);
6           return;
7       }
8
9       char sender[17] = { 0 };
10      uv_ip4_name((const struct sockaddr_in*) addr, sender, 16);
11      fprintf(stderr, "Recv from %s\n", sender);
12
13      // ... DHCP specific code
14      unsigned int *as_integer = (unsigned int*)buf->base;
15      unsigned int ipbin = ntohl(as_integer[4]);
16      unsigned char ip[4] = {0};
17      int i;
18      for (i = 0; i < 4; i++)
```

(continues on next page)

[1] http://beej.us/guide/bgnet/output/html/multipage/advanced.html#broadcast

(续上页)

```
        ip[i] = (ipbin >> i*8) & 0xff;
    fprintf(stderr, "Offered IP %d.%d.%d.%d\n", ip[3], ip[2], ip[1], ip[0]);

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

UDP Options

Time-to-live

The TTL of packets sent on the socket can be changed using `uv_udp_set_ttl`.

IPv6 stack only

IPv6 sockets can be used for both IPv4 and IPv6 communication. If you want to restrict the socket to IPv6 only, pass the `UV_UDP_IPV6ONLY` flag to `uv_udp_bind`[2].

Multicast

A socket can (un)subscribe to a multicast group using:

```
};

typedef void (*uv_udp_send_cb)(uv_udp_send_t* req, int status);
typedef void (*uv_udp_recv_cb)(uv_udp_t* handle,
```

where `membership` is `UV_JOIN_GROUP` or `UV_LEAVE_GROUP`.

The concepts of multicasting are nicely explained in this guide.

Local loopback of multicast packets is enabled by default[3], use `uv_udp_set_multicast_loop` to switch it off.

The packet time-to-live for multicast packets can be changed using `uv_udp_set_multicast_ttl`.

Querying DNS

libuv provides asynchronous DNS resolution. For this it provides its own `getaddrinfo` replacement[4]. In the callback you can perform normal socket operations on the retrieved addresses. Let's connect to Freenode to see an example of DNS resolution.

dns/main.c

[2] on Windows only supported on Windows Vista and later.

[3] http://www.tldp.org/HOWTO/Multicast-HOWTO-6.html#ss6.1

[4] libuv use the system `getaddrinfo` in the libuv threadpool. libuv v0.8.0 and earlier also included c-ares as an alternative, but this has been removed in v0.9.0.

```
int main() {
    loop = uv_default_loop();

    struct addrinfo hints;
    hints.ai_family = PF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;
    hints.ai_flags = 0;

    uv_getaddrinfo_t resolver;
    fprintf(stderr, "irc.freenode.net is... ");
    int r = uv_getaddrinfo(loop, &resolver, on_resolved, "irc.freenode.net", "6667", &
↪hints);

    if (r) {
        fprintf(stderr, "getaddrinfo call error %s\n", uv_err_name(r));
        return 1;
    }
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

If `uv_getaddrinfo` returns non-zero, something went wrong in the setup and your callback won't be invoked at all. All arguments can be freed immediately after `uv_getaddrinfo` returns. The *hostname*, *servname* and *hints* structures are documented in the getaddrinfo man page. The callback can be `NULL` in which case the function will run synchronously.

In the resolver callback, you can pick any IP from the linked list of `struct addrinfo(s)`. This also demonstrates `uv_tcp_connect`. It is necessary to call `uv_freeaddrinfo` in the callback.

dns/main.c

```
void on_resolved(uv_getaddrinfo_t *resolver, int status, struct addrinfo *res) {
    if (status < 0) {
        fprintf(stderr, "getaddrinfo callback error %s\n", uv_err_name(status));
        return;
    }

    char addr[17] = {'\0'};
    uv_ip4_name((struct sockaddr_in*) res->ai_addr, addr, 16);
    fprintf(stderr, "%s\n", addr);

    uv_connect_t *connect_req = (uv_connect_t*) malloc(sizeof(uv_connect_t));
    uv_tcp_t *socket = (uv_tcp_t*) malloc(sizeof(uv_tcp_t));
    uv_tcp_init(loop, socket);

    uv_tcp_connect(connect_req, socket, (const struct sockaddr*) res->ai_addr, on_
↪connect);

    uv_freeaddrinfo(res);
}
```

libuv also provides the inverse uv_getnameinfo.

Network interfaces

Information about the system's network interfaces can be obtained through libuv using `uv_interface_addresses`. This simple program just prints out all the interface details so you get an idea of the fields that are available. This is useful to allow your service to bind to IP addresses when it starts.

interfaces/main.c

```
#include <stdio.h>
#include <uv.h>

int main() {
    char buf[512];
    uv_interface_address_t *info;
    int count, i;

    uv_interface_addresses(&info, &count);
    i = count;

    printf("Number of interfaces: %d\n", count);
    while (i--) {
        uv_interface_address_t interface = info[i];

        printf("Name: %s\n", interface.name);
        printf("Internal? %s\n", interface.is_internal ? "Yes" : "No");

        if (interface.address.address4.sin_family == AF_INET) {
            uv_ip4_name(&interface.address.address4, buf, sizeof(buf));
            printf("IPv4 address: %s\n", buf);
        }
        else if (interface.address.address4.sin_family == AF_INET6) {
            uv_ip6_name(&interface.address.address6, buf, sizeof(buf));
            printf("IPv6 address: %s\n", buf);
        }

        printf("\n");
    }

    uv_free_interface_addresses(info, count);
    return 0;
}
```

`is_internal` is true for loopback interfaces. Note that if a physical interface has multiple IPv4/IPv6 addresses, the name will be reported multiple times, with each address being reported once.

3.3.5 Threads

Wait a minute? Why are we on threads? Aren't event loops supposed to be **the way** to do *web-scale programming*? Well... no. Threads are still the medium in which processors do their jobs. Threads are therefore mighty useful sometimes, even though you might have to wade through various synchronization primitives.

Threads are used internally to fake the asynchronous nature of all of the system calls. libuv also uses threads to allow you, the application, to perform a task asynchronously that is actually blocking, by spawning a thread and collecting the result when it is done.

Today there are two predominant thread libraries: the Windows threads implementation and POSIX's pthreads. libuv's thread API is analogous to the pthreads API and often has similar semantics.

A notable aspect of libuv's thread facilities is that it is a self contained section within libuv. Whereas other features intimately depend on the event loop and callback principles, threads are complete agnostic, they block as required, signal errors directly via return values, and, as shown in the *first example*, don't even require a running event loop.

libuv's thread API is also very limited since the semantics and syntax of threads are different on all platforms, with different levels of completeness.

This chapter makes the following assumption: **There is only one event loop, running in one thread (the main thread)**. No other thread interacts with the event loop (except using `uv_async_send`).

Core thread operations

There isn't much here, you just start a thread using `uv_thread_create()` and wait for it to close using `uv_thread_join()`.

thread-create/main.c

```
int main() {
    int tracklen = 10;
    uv_thread_t hare_id;
    uv_thread_t tortoise_id;
    uv_thread_create(&hare_id, hare, &tracklen);
    uv_thread_create(&tortoise_id, tortoise, &tracklen);

    uv_thread_join(&hare_id);
    uv_thread_join(&tortoise_id);
    return 0;
}
```

小技巧: `uv_thread_t` is just an alias for `pthread_t` on Unix, but this is an implementation detail, avoid depending on it to always be true.

The second parameter is the function which will serve as the entry point for the thread, the last parameter is a `void *` argument which can be used to pass custom parameters to the thread. The function `hare` will now run in a separate thread, scheduled pre-emptively by the operating system:

thread-create/main.c

```
void hare(void *arg) {
    int tracklen = *((int *) arg);
    while (tracklen) {
        tracklen--;
        sleep(1);
        fprintf(stderr, "Hare ran another step\n");
    }
    fprintf(stderr, "Hare done running!\n");
}
```

Unlike `pthread_join()` which allows the target thread to pass back a value to the calling thread using a second parameter, `uv_thread_join()` does not. To send values use *Inter-thread communication*.

Synchronization Primitives

This section is purposely spartan. This book is not about threads, so I only catalogue any surprises in the libuv APIs here. For the rest you can look at the pthreads man pages.

Mutexes

The mutex functions are a **direct** map to the pthread equivalents.

libuv mutex functions

```
/*
 * This flag can be used with uv_fs_symlink() on Windows to specify whether
 * path argument points to a directory.
 */
#define UV_FS_SYMLINK_DIR          0x0001
```

The `uv_mutex_init()`, `uv_mutex_init_recursive()` and `uv_mutex_trylock()` functions will return 0 on success, and an error code otherwise.

If *libuv* has been compiled with debugging enabled, `uv_mutex_destroy()`, `uv_mutex_lock()` and `uv_mutex_unlock()` will `abort()` on error. Similarly `uv_mutex_trylock()` will abort if the error is anything *other than* `EAGAIN` or `EBUSY`.

Recursive mutexes are supported, but you should not rely on them. Also, they should not be used with `uv_cond_t` variables.

The default BSD mutex implementation will raise an error if a thread which has locked a mutex attempts to lock it again. For example, a construct like:

```
uv_mutex_init(a_mutex);
uv_mutex_lock(a_mutex);
uv_thread_create(thread_id, entry, (void *)a_mutex);
uv_mutex_lock(a_mutex);
// more things here
```

can be used to wait until another thread initializes some stuff and then unlocks `a_mutex` but will lead to your program crashing if in debug mode, or return an error in the second call to `uv_mutex_lock()`.

注解： Mutexes on Windows are always recursive.

Locks

Read-write locks are a more granular access mechanism. Two readers can access shared memory at the same time. A writer may not acquire the lock when it is held by a reader. A reader or writer may not acquire a lock when a writer is holding it. Read-write locks are frequently used in databases. Here is a toy example.

locks/main.c - simple rwlocks

```c
#include <stdio.h>
#include <uv.h>

uv_barrier_t blocker;
uv_rwlock_t numlock;
int shared_num;

void reader(void *n)
{
    int num = *(int *)n;
    int i;
    for (i = 0; i < 20; i++) {
        uv_rwlock_rdlock(&numlock);
        printf("Reader %d: acquired lock\n", num);
        printf("Reader %d: shared num = %d\n", num, shared_num);
        uv_rwlock_rdunlock(&numlock);
        printf("Reader %d: released lock\n", num);
    }
    uv_barrier_wait(&blocker);
}

void writer(void *n)
{
    int num = *(int *)n;
    int i;
    for (i = 0; i < 20; i++) {
        uv_rwlock_wrlock(&numlock);
        printf("Writer %d: acquired lock\n", num);
        shared_num++;
        printf("Writer %d: incremented shared num = %d\n", num, shared_num);
        uv_rwlock_wrunlock(&numlock);
        printf("Writer %d: released lock\n", num);
    }
    uv_barrier_wait(&blocker);
}

int main()
{
    uv_barrier_init(&blocker, 4);

    shared_num = 0;
    uv_rwlock_init(&numlock);

    uv_thread_t threads[3];

    int thread_nums[] = {1, 2, 1};
    uv_thread_create(&threads[0], reader, &thread_nums[0]);
    uv_thread_create(&threads[1], reader, &thread_nums[1]);

    uv_thread_create(&threads[2], writer, &thread_nums[2]);

    uv_barrier_wait(&blocker);
    uv_barrier_destroy(&blocker);

    uv_rwlock_destroy(&numlock);
```

(continues on next page)

(续上页)

```
56      return 0;
57  }
```

Run this and observe how the readers will sometimes overlap. In case of multiple writers, schedulers will usually give them higher priority, so if you add two writers, you'll see that both writers tend to finish first before the readers get a chance again.

We also use barriers in the above example so that the main thread can wait for all readers and writers to indicate they have ended.

Others

libuv also supports semaphores, condition variables and barriers with APIs very similar to their pthread counterparts.

In addition, libuv provides a convenience function `uv_once()`. Multiple threads can attempt to call `uv_once()` with a given guard and a function pointer, **only the first one will win, the function will be called once and only once**:

```
/* Initialize guard */
static uv_once_t once_only = UV_ONCE_INIT;

int i = 0;

void increment() {
    i++;
}

void thread1() {
    /* ... work */
    uv_once(once_only, increment);
}

void thread2() {
    /* ... work */
    uv_once(once_only, increment);
}

int main() {
    /* ... spawn threads */
}
```

After all threads are done, `i == 1`.

libuv v0.11.11 onwards also added a `uv_key_t` struct and api for thread-local storage.

libuv work queue

`uv_queue_work()` is a convenience function that allows an application to run a task in a separate thread, and have a callback that is triggered when the task is done. A seemingly simple function, what makes `uv_queue_work()` tempting is that it allows potentially any third-party libraries to be used with the event-loop paradigm. When you use event loops, it is *imperative to make sure that no function which runs periodically in the loop thread blocks when performing I/O or is a serious CPU hog*, because this means that the loop slows down and events are not being handled at full capacity.

However, a lot of existing code out there features blocking functions (for example a routine which performs I/O under the hood) to be used with threads if you want responsiveness (the classic 'one thread per client' server model), and getting them to play with an event loop library generally involves rolling your own system of running the task in a separate thread. libuv just provides a convenient abstraction for this.

Here is a simple example inspired by node.js is cancer. We are going to calculate fibonacci numbers, sleeping a bit along the way, but run it in a separate thread so that the blocking and CPU bound task does not prevent the event loop from performing other activities.

queue-work/main.c - lazy fibonacci

```
void fib(uv_work_t *req) {
    int n = *(int *) req->data;
    if (random() % 2)
        sleep(1);
    else
        sleep(3);
    long fib = fib_(n);
    fprintf(stderr, "%dth fibonacci is %lu\n", n, fib);
}

void after_fib(uv_work_t *req, int status) {
    fprintf(stderr, "Done calculating %dth fibonacci\n", *(int *) req->data);
}
```

The actual task function is simple, nothing to show that it is going to be run in a separate thread. The `uv_work_t` structure is the clue. You can pass arbitrary data through it using the `void* data` field and use it to communicate to and from the thread. But be sure you are using proper locks if you are changing things while both threads may be running.

The trigger is `uv_queue_work`:

queue-work/main.c

```
int main() {
    loop = uv_default_loop();

    int data[FIB_UNTIL];
    uv_work_t req[FIB_UNTIL];
    int i;
    for (i = 0; i < FIB_UNTIL; i++) {
        data[i] = i;
        req[i].data = (void *) &data[i];
        uv_queue_work(loop, &req[i], fib, after_fib);
    }

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

The thread function will be launched in a separate thread, passed the `uv_work_t` structure and once the function returns, the *after* function will be called on the thread the event loop is running in. It will be passed the same structure.

For writing wrappers to blocking libraries, a common *pattern* is to use a baton to exchange data.

Since libuv version *0.9.4* an additional function, `uv_cancel()`, is available. This allows you to cancel tasks on the libuv work queue. Only tasks that *are yet to be started* can be cancelled. If a task has *already started executing, or it*

has finished executing, `uv_cancel()` **will fail**.

`uv_cancel()` is useful to cleanup pending tasks if the user requests termination. For example, a music player may queue up multiple directories to be scanned for audio files. If the user terminates the program, it should quit quickly and not wait until all pending requests are run.

Let's modify the fibonacci example to demonstrate `uv_cancel()`. We first set up a signal handler for termination.

queue-cancel/main.c

```
int main() {
    loop = uv_default_loop();

    int data[FIB_UNTIL];
    int i;
    for (i = 0; i < FIB_UNTIL; i++) {
        data[i] = i;
        fib_reqs[i].data = (void *) &data[i];
        uv_queue_work(loop, &fib_reqs[i], fib, after_fib);
    }

    uv_signal_t sig;
    uv_signal_init(loop, &sig);
    uv_signal_start(&sig, signal_handler, SIGINT);

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

When the user triggers the signal by pressing `Ctrl+C` we send `uv_cancel()` to all the workers. `uv_cancel()` will return `0` for those that are already executing or finished.

queue-cancel/main.c

```
void signal_handler(uv_signal_t *req, int signum)
{
    printf("Signal received!\n");
    int i;
    for (i = 0; i < FIB_UNTIL; i++) {
        uv_cancel((uv_req_t*) &fib_reqs[i]);
    }
    uv_signal_stop(req);
}
```

For tasks that do get cancelled successfully, the *after* function is called with `status` set to `UV_ECANCELED`.

queue-cancel/main.c

```
void after_fib(uv_work_t *req, int status) {
    if (status == UV_ECANCELED)
        fprintf(stderr, "Calculation of %d cancelled.\n", *(int *) req->data);
}
```

`uv_cancel()` can also be used with `uv_fs_t` and `uv_getaddrinfo_t` requests. For the filesystem family of functions, `uv_fs_t.errorno` will be set to `UV_ECANCELED`.

小技巧: A well designed program would have a way to terminate long running workers that have already started executing. Such a worker could periodically check for a variable that only the main process sets to signal termination.

Inter-thread communication

Sometimes you want various threads to actually send each other messages *while* they are running. For example you might be running some long duration task in a separate thread (perhaps using `uv_queue_work`) but want to notify progress to the main thread. This is a simple example of having a download manager informing the user of the status of running downloads.

progress/main.c

```
uv_loop_t *loop;
uv_async_t async;
}

int main() {
    loop = uv_default_loop();

    uv_work_t req;
    int size = 10240;
    req.data = (void*) &size;

    uv_async_init(loop, &async, print_progress);
    uv_queue_work(loop, &req, fake_download, after);

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

The async thread communication works *on loops* so although any thread can be the message sender, only threads with libuv loops can be receivers (or rather the loop is the receiver). libuv will invoke the callback (`print_progress`) with the async watcher whenever it receives a message.

警告: It is important to realize that since the message send is *async*, the callback may be invoked immediately after `uv_async_send` is called in another thread, or it may be invoked after some time. libuv may also combine multiple calls to `uv_async_send` and invoke your callback only once. The only guarantee that libuv makes is – The callback function is called *at least once* after the call to `uv_async_send`. If you have no pending calls to `uv_async_send`, the callback won't be called. If you make two or more calls, and libuv hasn't had a chance to run the callback yet, it *may* invoke your callback *only once* for the multiple invocations of `uv_async_send`. Your callback will never be called twice for just one event.

progress/main.c

```
double percentage;

void fake_download(uv_work_t *req) {
    int size = *((int*) req->data);
    int downloaded = 0;
```

(continues on next page)

(续上页)

```
    while (downloaded < size) {
        percentage = downloaded*100.0/size;
        async.data = (void*) &percentage;
        uv_async_send(&async);

        sleep(1);
        downloaded += (200+random())%1000; // can only download max 1000bytes/sec,
                                           // but at least a 200;
    }
```

In the download function, we modify the progress indicator and queue the message for delivery with `uv_async_send`. Remember: `uv_async_send` is also non-blocking and will return immediately.

progress/main.c

```

void print_progress(uv_async_t *handle) {
    double percentage = *((double*) handle->data);
    fprintf(stderr, "Downloaded %.2f%%\n", percentage);
```

The callback is a standard libuv pattern, extracting the data from the watcher.

Finally it is important to remember to clean up the watcher.

progress/main.c

```

void after(uv_work_t *req, int status) {
    fprintf(stderr, "Download complete\n");
    uv_close((uv_handle_t*) &async, NULL);
```

After this example, which showed the abuse of the `data` field, bnoordhuis pointed out that using the `data` field is not thread safe, and `uv_async_send()` is actually only meant to wake up the event loop. Use a mutex or rwlock to ensure accesses are performed in the right order.

注解: mutexes and rwlocks **DO NOT** work inside a signal handler, whereas `uv_async_send` does.

One use case where `uv_async_send` is required is when interoperating with libraries that require thread affinity for their functionality. For example in node.js, a v8 engine instance, contexts and its objects are bound to the thread that the v8 instance was started in. Interacting with v8 data structures from another thread can lead to undefined results. Now consider some node.js module which binds a third party library. It may go something like this:

1. In node, the third party library is set up with a JavaScript callback to be invoked for more information:

```
var lib = require('lib');
lib.on_progress(function() {
    console.log("Progress");
});

lib.do();

// do other stuff
```

2. `lib.do` is supposed to be non-blocking but the third party lib is blocking, so the binding uses `uv_queue_work`.
3. The actual work being done in a separate thread wants to invoke the progress callback, but cannot directly call into v8 to interact with JavaScript. So it uses `uv_async_send`.
4. The async callback, invoked in the main loop thread, which is the v8 thread, then interacts with v8 to invoke the JavaScript callback.

3.3.6 Processes

libuv offers considerable child process management, abstracting the platform differences and allowing communication with the child process using streams or named pipes.

A common idiom in Unix is for every process to do one thing and do it well. In such a case, a process often uses multiple child processes to achieve tasks (similar to using pipes in shells). A multi-process model with messages may also be easier to reason about compared to one with threads and shared memory.

A common refrain against event-based programs is that they cannot take advantage of multiple cores in modern computers. In a multi-threaded program the kernel can perform scheduling and assign different threads to different cores, improving performance. But an event loop has only one thread. The workaround can be to launch multiple processes instead, with each process running an event loop, and each process getting assigned to a separate CPU core.

Spawning child processes

The simplest case is when you simply want to launch a process and know when it exits. This is achieved using `uv_spawn`.

spawn/main.c

```
uv_loop_t *loop;
uv_process_t child_req;
uv_process_options_t options;
int main() {
    loop = uv_default_loop();

    char* args[3];
    args[0] = "mkdir";
    args[1] = "test-dir";
    args[2] = NULL;

    options.exit_cb = on_exit;
    options.file = "mkdir";
    options.args = args;

    int r;
    if ((r = uv_spawn(loop, &child_req, &options))) {
        fprintf(stderr, "%s\n", uv_strerror(r));
        return 1;
    } else {
        fprintf(stderr, "Launched process with ID %d\n", child_req.pid);
    }
```

(continues on next page)

(续上页)

```
23
24      return uv_run(loop, UV_RUN_DEFAULT);
25  }
```

注解: `options` is implicitly initialized with zeros since it is a global variable. If you change `options` to a local variable, remember to initialize it to null out all unused fields:

```
uv_process_options_t options = {0};
```

The `uv_process_t` struct only acts as the handle, all options are set via `uv_process_options_t`. To simply launch a process, you need to set only the `file` and `args` fields. `file` is the program to execute. Since `uv_spawn` uses execvp internally, there is no need to supply the full path. Finally as per underlying conventions, **the arguments array has to be one larger than the number of arguments, with the last element being NULL**.

After the call to `uv_spawn`, `uv_process_t.pid` will contain the process ID of the child process.

The exit callback will be invoked with the *exit status* and the type of *signal* which caused the exit.

spawn/main.c

```
1
2   void on_exit(uv_process_t *req, int64_t exit_status, int term_signal) {
3       fprintf(stderr, "Process exited with status %" PRId64 ", signal %d\n", exit_
    ↪status, term_signal);
4       uv_close((uv_handle_t*) req, NULL);
```

It is **required** to close the process watcher after the process exits.

Changing process parameters

Before the child process is launched you can control the execution environment using fields in `uv_process_options_t`.

Change execution directory

Set `uv_process_options_t.cwd` to the corresponding directory.

Set environment variables

`uv_process_options_t.env` is a null-terminated array of strings, each of the form `VAR=VALUE` used to set up the environment variables for the process. Set this to `NULL` to inherit the environment from the parent (this) process.

Option flags

Setting `uv_process_options_t.flags` to a bitwise OR of the following flags, modifies the child process behaviour:

- `UV_PROCESS_SETUID` - sets the child's execution user ID to `uv_process_options_t.uid`.

- UV_PROCESS_SETGID - sets the child's execution group ID to uv_process_options_t.gid.

Changing the UID/GID is only supported on Unix, uv_spawn will fail on Windows with UV_ENOTSUP.

- UV_PROCESS_WINDOWS_VERBATIM_ARGUMENTS - No quoting or escaping of uv_process_options_t.args is done on Windows. Ignored on Unix.
- UV_PROCESS_DETACHED - Starts the child process in a new session, which will keep running after the parent process exits. See example below.

Detaching processes

Passing the flag UV_PROCESS_DETACHED can be used to launch daemons, or child processes which are independent of the parent so that the parent exiting does not affect it.

detach/main.c

```
int main() {
    loop = uv_default_loop();

    char* args[3];
    args[0] = "sleep";
    args[1] = "100";
    args[2] = NULL;

    options.exit_cb = NULL;
    options.file = "sleep";
    options.args = args;
    options.flags = UV_PROCESS_DETACHED;

    int r;
    if ((r = uv_spawn(loop, &child_req, &options))) {
        fprintf(stderr, "%s\n", uv_strerror(r));
        return 1;
    }
    fprintf(stderr, "Launched sleep with PID %d\n", child_req.pid);
    uv_unref((uv_handle_t*) &child_req);

    return uv_run(loop, UV_RUN_DEFAULT);
```

Just remember that the handle is still monitoring the child, so your program won't exit. Use uv_unref() if you want to be more *fire-and-forget*.

Sending signals to processes

libuv wraps the standard kill(2) system call on Unix and implements one with similar semantics on Windows, with *one caveat*: all of SIGTERM, SIGINT and SIGKILL, lead to termination of the process. The signature of uv_kill is:

```
uv_err_t uv_kill(int pid, int signum);
```

For processes started using libuv, you may use uv_process_kill instead, which accepts the uv_process_t watcher as the first argument, rather than the pid. In this case, **remember to call** uv_close on the watcher.

Signals

libuv provides wrappers around Unix signals with some Windows support as well.

Use `uv_signal_init()` to initialize a handle and associate it with a loop. To listen for particular signals on that handler, use `uv_signal_start()` with the handler function. Each handler can only be associated with one signal number, with subsequent calls to `uv_signal_start()` overwriting earlier associations. Use `uv_signal_stop()` to stop watching. Here is a small example demonstrating the various possibilities:

signal/main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <uv.h>

uv_loop_t* create_loop()
{
    uv_loop_t *loop = malloc(sizeof(uv_loop_t));
    if (loop) {
      uv_loop_init(loop);
    }
    return loop;
}

void signal_handler(uv_signal_t *handle, int signum)
{
    printf("Signal received: %d\n", signum);
    uv_signal_stop(handle);
}

// two signal handlers in one loop
void thread1_worker(void *userp)
{
    uv_loop_t *loop1 = create_loop();

    uv_signal_t sig1a, sig1b;
    uv_signal_init(loop1, &sig1a);
    uv_signal_start(&sig1a, signal_handler, SIGUSR1);

    uv_signal_init(loop1, &sig1b);
    uv_signal_start(&sig1b, signal_handler, SIGUSR1);

    uv_run(loop1, UV_RUN_DEFAULT);
}

// two signal handlers, each in its own loop
void thread2_worker(void *userp)
{
    uv_loop_t *loop2 = create_loop();
    uv_loop_t *loop3 = create_loop();

    uv_signal_t sig2;
    uv_signal_init(loop2, &sig2);
    uv_signal_start(&sig2, signal_handler, SIGUSR1);

```

(continues on next page)

(续上页)

```
    uv_signal_t sig3;
    uv_signal_init(loop3, &sig3);
    uv_signal_start(&sig3, signal_handler, SIGUSR1);

    while (uv_run(loop2, UV_RUN_NOWAIT) || uv_run(loop3, UV_RUN_NOWAIT)) {
    }
}

int main()
{
    printf("PID %d\n", getpid());

    uv_thread_t thread1, thread2;

    uv_thread_create(&thread1, thread1_worker, 0);
    uv_thread_create(&thread2, thread2_worker, 0);

    uv_thread_join(&thread1);
    uv_thread_join(&thread2);
    return 0;
}
```

注解: `uv_run(loop, UV_RUN_NOWAIT)` is similar to `uv_run(loop, UV_RUN_ONCE)` in that it will process only one event. UV_RUN_ONCE blocks if there are no pending events, while UV_RUN_NOWAIT will return immediately. We use NOWAIT so that one of the loops isn't starved because the other one has no pending activity.

Send `SIGUSR1` to the process, and you'll find the handler being invoked 4 times, one for each `uv_signal_t`. The handler just stops each handle, so that the program exits. This sort of dispatch to all handlers is very useful. A server using multiple event loops could ensure that all data was safely saved before termination, simply by every loop adding a watcher for `SIGINT`.

Child Process I/O

A normal, newly spawned process has its own set of file descriptors, with 0, 1 and 2 being `stdin`, `stdout` and `stderr` respectively. Sometimes you may want to share file descriptors with the child. For example, perhaps your applications launches a sub-command and you want any errors to go in the log file, but ignore `stdout`. For this you'd like to have `stderr` of the child be the same as the stderr of the parent. In this case, libuv supports *inheriting* file descriptors. In this sample, we invoke the test program, which is:

proc-streams/test.c

```
#include <stdio.h>

int main()
{
    fprintf(stderr, "This is stderr\n");
    printf("This is stdout\n");
    return 0;
}
```

The actual program `proc-streams` runs this while sharing only `stderr`. The file descriptors of the child process are set using the `stdio` field in `uv_process_options_t`. First set the `stdio_count` field to the number of

file descriptors being set. `uv_process_options_t.stdio` is an array of `uv_stdio_container_t`, which is:

```
  /* struct addrinfo* addrinfo is marked as private, but it really isn't. */
  UV_GETADDRINFO_PRIVATE_FIELDS
};


UV_EXTERN int uv_getaddrinfo(uv_loop_t* loop,
                             uv_getaddrinfo_t* req,
                             uv_getaddrinfo_cb getaddrinfo_cb,
                             const char* node,
```

where flags can have several values. Use `UV_IGNORE` if it isn't going to be used. If the first three `stdio` fields are marked as `UV_IGNORE` they'll redirect to `/dev/null`.

Since we want to pass on an existing descriptor, we'll use `UV_INHERIT_FD`. Then we set the `fd` to `stderr`.

proc-streams/main.c

```
1
2  int main() {
3      loop = uv_default_loop();
4
5      /* ... */
6
7      options.stdio_count = 3;
8      uv_stdio_container_t child_stdio[3];
9      child_stdio[0].flags = UV_IGNORE;
10     child_stdio[1].flags = UV_IGNORE;
11     child_stdio[2].flags = UV_INHERIT_FD;
12     child_stdio[2].data.fd = 2;
13     options.stdio = child_stdio;
14
15     options.exit_cb = on_exit;
16     options.file = args[0];
17     options.args = args;
18
19     int r;
20     if ((r = uv_spawn(loop, &child_req, &options))) {
21         fprintf(stderr, "%s\n", uv_strerror(r));
22         return 1;
23     }
24
25     return uv_run(loop, UV_RUN_DEFAULT);
26 }
```

If you run `proc-stream` you'll see that only the line "This is stderr" will be displayed. Try marking `stdout` as being inherited and see the output.

It is dead simple to apply this redirection to streams. By setting `flags` to `UV_INHERIT_STREAM` and setting `data.stream` to the stream in the parent process, the child process can treat that stream as standard I/O. This can be used to implement something like CGI.

A sample CGI script/executable is:

cgi/tick.c

```
#include <stdio.h>
#include <unistd.h>

int main() {
    int i;
    for (i = 0; i < 10; i++) {
        printf("tick\n");
        fflush(stdout);
        sleep(1);
    }
    printf("BOOM!\n");
    return 0;
}
```

The CGI server combines the concepts from this chapter and *Networking* so that every client is sent ten ticks after which that connection is closed.

cgi/main.c

```

void on_new_connection(uv_stream_t *server, int status) {
    if (status == -1) {
        // error!
        return;
    }

    uv_tcp_t *client = (uv_tcp_t*) malloc(sizeof(uv_tcp_t));
    uv_tcp_init(loop, client);
    if (uv_accept(server, (uv_stream_t*) client) == 0) {
        invoke_cgi_script(client);
    }
    else {
        uv_close((uv_handle_t*) client, NULL);
    }
```

Here we simply accept the TCP connection and pass on the socket (*stream*) to `invoke_cgi_script`.

cgi/main.c

```

    args[1] = NULL;

    /* ... finding the executable path and setting up arguments ... */

    options.stdio_count = 3;
    uv_stdio_container_t child_stdio[3];
    child_stdio[0].flags = UV_IGNORE;
    child_stdio[1].flags = UV_INHERIT_STREAM;
    child_stdio[1].data.stream = (uv_stream_t*) client;
    child_stdio[2].flags = UV_IGNORE;
    options.stdio = child_stdio;
```

(continues on next page)

(续上页)

```
13
14      options.exit_cb = cleanup_handles;
15      options.file = args[0];
16      options.args = args;
17
18      // Set this so we can close the socket after the child process exits.
19      child_req.data = (void*) client;
20      int r;
21      if ((r = uv_spawn(loop, &child_req, &options))) {
22          fprintf(stderr, "%s\n", uv_strerror(r));
```

The `stdout` of the CGI script is set to the socket so that whatever our tick script prints, gets sent to the client. By using processes, we can offload the read/write buffering to the operating system, so in terms of convenience this is great. Just be warned that creating processes is a costly task.

Pipes

libuv's `uv_pipe_t` structure is slightly confusing to Unix programmers, because it immediately conjures up `|` and pipe(7). But `uv_pipe_t` is not related to anonymous pipes, rather it is an IPC mechanism. `uv_pipe_t` can be backed by a Unix Domain Socket or Windows Named Pipe to allow multiple processes to communicate. This is discussed below.

Parent-child IPC

A parent and child can have one or two way communication over a pipe created by settings `uv_stdio_container_t.flags` to a bit-wise combination of `UV_CREATE_PIPE` and `UV_READABLE_PIPE` or `UV_WRITABLE_PIPE`. The read/write flag is from the perspective of the child process.

Arbitrary process IPC

Since domain sockets[1] can have a well known name and a location in the file-system they can be used for IPC between unrelated processes. The D-BUS system used by open source desktop environments uses domain sockets for event notification. Various applications can then react when a contact comes online or new hardware is detected. The MySQL server also runs a domain socket on which clients can interact with it.

When using domain sockets, a client-server pattern is usually followed with the creator/owner of the socket acting as the server. After the initial setup, messaging is no different from TCP, so we'll re-use the echo server example.

pipe-echo-server/main.c

```
1  void remove_sock(int sig) {
2      uv_fs_t req;
3      uv_fs_unlink(loop, &req, PIPENAME, NULL);
4      exit(0);
5  }
6
7  int main() {
8      loop = uv_default_loop();
```

(continues on next page)

[1] In this section domain sockets stands in for named pipes on Windows as well.

(续上页)

```
9
10      uv_pipe_t server;
11      uv_pipe_init(loop, &server, 0);
12
13      signal(SIGINT, remove_sock);
14
15      int r;
16      if ((r = uv_pipe_bind(&server, PIPENAME))) {
17          fprintf(stderr, "Bind error %s\n", uv_err_name(r));
18          return 1;
19      }
20      if ((r = uv_listen((uv_stream_t*) &server, 128, on_new_connection))) {
21          fprintf(stderr, "Listen error %s\n", uv_err_name(r));
22          return 2;
23      }
24      return uv_run(loop, UV_RUN_DEFAULT);
25  }
```

We name the socket `echo.sock` which means it will be created in the local directory. This socket now behaves no different from TCP sockets as far as the stream API is concerned. You can test this server using socat:

```
$ socat - /path/to/socket
```

A client which wants to connect to a domain socket will use:

```
void uv_pipe_connect(uv_connect_t *req, uv_pipe_t *handle, const char *name, uv_
↪connect_cb cb);
```

where `name` will be `echo.sock` or similar. On Unix systems, `name` must point to a valid file (e.g. `/tmp/echo.sock`). On Windows, `name` follows a `\\?\pipe\echo.sock` format.

Sending file descriptors over pipes

The cool thing about domain sockets is that file descriptors can be exchanged between processes by sending them over a domain socket. This allows processes to hand off their I/O to other processes. Applications include load-balancing servers, worker processes and other ways to make optimum use of CPU. libuv only supports sending **TCP sockets or other pipes** over pipes for now.

To demonstrate, we will look at a echo server implementation that hands of clients to worker processes in a round-robin fashion. This program is a bit involved, and while only snippets are included in the book, it is recommended to read the full code to really understand it.

The worker process is quite simple, since the file-descriptor is handed over to it by the master.

multi-echo-server/worker.c

```
1
2   uv_loop_t *loop;
3   uv_pipe_t queue;
4   int main() {
5       loop = uv_default_loop();
6
7       uv_pipe_init(loop, &queue, 1 /* ipc */);
```

(continues on next page)

(续上页)

```
    uv_pipe_open(&queue, 0);
    uv_read_start((uv_stream_t*)&queue, alloc_buffer, on_new_connection);
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

`queue` is the pipe connected to the master process on the other end, along which new file descriptors get sent. It is important to set the `ipc` argument of `uv_pipe_init` to 1 to indicate this pipe will be used for inter-process communication! Since the master will write the file handle to the standard input of the worker, we connect the pipe to `stdin` using `uv_pipe_open`.

multi-echo-server/worker.c

```
void on_new_connection(uv_stream_t *q, ssize_t nread, const uv_buf_t *buf) {
    if (nread < 0) {
        if (nread != UV_EOF)
            fprintf(stderr, "Read error %s\n", uv_err_name(nread));
        uv_close((uv_handle_t*) q, NULL);
        return;
    }

    uv_pipe_t *pipe = (uv_pipe_t*) q;
    if (!uv_pipe_pending_count(pipe)) {
        fprintf(stderr, "No pending count\n");
        return;
    }

    uv_handle_type pending = uv_pipe_pending_type(pipe);
    assert(pending == UV_TCP);

    uv_tcp_t *client = (uv_tcp_t*) malloc(sizeof(uv_tcp_t));
    uv_tcp_init(loop, client);
    if (uv_accept(q, (uv_stream_t*) client) == 0) {
        uv_os_fd_t fd;
        uv_fileno((const uv_handle_t*) client, &fd);
        fprintf(stderr, "Worker %d: Accepted fd %d\n", getpid(), fd);
        uv_read_start((uv_stream_t*) client, alloc_buffer, echo_read);
    }
    else {
        uv_close((uv_handle_t*) client, NULL);
    }
}
```

First we call `uv_pipe_pending_count()` to ensure that a handle is available to read out. If your program could deal with different types of handles, `uv_pipe_pending_type()` can be used to determine the type. Although `accept` seems odd in this code, it actually makes sense. What `accept` traditionally does is get a file descriptor (the client) from another file descriptor (The listening socket). Which is exactly what we do here. Fetch the file descriptor (`client`) from `queue`. From this point the worker does standard echo server stuff.

Turning now to the master, let's take a look at how the workers are launched to allow load balancing.

multi-echo-server/main.c

```
struct child_worker {
    uv_process_t req;
    uv_process_options_t options;
    uv_pipe_t pipe;
} *workers;
```

The `child_worker` structure wraps the process, and the pipe between the master and the individual process.

multi-echo-server/main.c

```
void setup_workers() {
    round_robin_counter = 0;

    // ...

    // launch same number of workers as number of CPUs
    uv_cpu_info_t *info;
    int cpu_count;
    uv_cpu_info(&info, &cpu_count);
    uv_free_cpu_info(info, cpu_count);

    child_worker_count = cpu_count;

    workers = calloc(sizeof(struct child_worker), cpu_count);
    while (cpu_count--) {
        struct child_worker *worker = &workers[cpu_count];
        uv_pipe_init(loop, &worker->pipe, 1);

        uv_stdio_container_t child_stdio[3];
        child_stdio[0].flags = UV_CREATE_PIPE | UV_READABLE_PIPE;
        child_stdio[0].data.stream = (uv_stream_t*) &worker->pipe;
        child_stdio[1].flags = UV_IGNORE;
        child_stdio[2].flags = UV_INHERIT_FD;
        child_stdio[2].data.fd = 2;

        worker->options.stdio = child_stdio;
        worker->options.stdio_count = 3;

        worker->options.exit_cb = close_process_handle;
        worker->options.file = args[0];
        worker->options.args = args;

        uv_spawn(loop, &worker->req, &worker->options);
        fprintf(stderr, "Started worker %d\n", worker->req.pid);
    }
}
```

In setting up the workers, we use the nifty libuv function `uv_cpu_info` to get the number of CPUs so we can launch an equal number of workers. Again it is important to initialize the pipe acting as the IPC channel with the third argument as 1. We then indicate that the child process' `stdin` is to be a readable pipe (from the point of view of the child). Everything is straightforward till here. The workers are launched and waiting for file descriptors to be written to their standard input.

It is in `on_new_connection` (the TCP infrastructure is initialized in `main()`), that we accept the client socket

and pass it along to the next worker in the round-robin.

multi-echo-server/main.c

```
void on_new_connection(uv_stream_t *server, int status) {
    if (status == -1) {
        // error!
        return;
    }

    uv_tcp_t *client = (uv_tcp_t*) malloc(sizeof(uv_tcp_t));
    uv_tcp_init(loop, client);
    if (uv_accept(server, (uv_stream_t*) client) == 0) {
        uv_write_t *write_req = (uv_write_t*) malloc(sizeof(uv_write_t));
        dummy_buf = uv_buf_init("a", 1);
        struct child_worker *worker = &workers[round_robin_counter];
        uv_write2(write_req, (uv_stream_t*) &worker->pipe, &dummy_buf, 1, (uv_stream_
↪t*) client, NULL);
        round_robin_counter = (round_robin_counter + 1) % child_worker_count;
    }
    else {
        uv_close((uv_handle_t*) client, NULL);
    }
}
```

The `uv_write2` call handles all the abstraction and it is simply a matter of passing in the handle (`client`) as the right argument. With this our multi-process echo server is operational.

Thanks to Kyle for pointing out that `uv_write2()` requires a non-empty buffer even when sending handles.

3.3.7 Advanced event loops

libuv provides considerable user control over event loops, and you can achieve interesting results by juggling multiple loops. You can also embed libuv's event loop into another event loop based library – imagine a Qt based UI, and Qt's event loop driving a libuv backend which does intensive system level tasks.

Stopping an event loop

`uv_stop()` can be used to stop an event loop. The earliest the loop will stop running is *on the next iteration*, possibly later. This means that events that are ready to be processed in this iteration of the loop will still be processed, so `uv_stop()` can't be used as a kill switch. When `uv_stop()` is called, the loop **won't** block for i/o on this iteration. The semantics of these things can be a bit difficult to understand, so let's look at `uv_run()` where all the control flow occurs.

src/unix/core.c - uv_run

`stop_flag` is set by `uv_stop()`. Now all libuv callbacks are invoked within the event loop, which is why invoking `uv_stop()` in them will still lead to this iteration of the loop occurring. First libuv updates timers, then runs pending timer, idle and prepare callbacks, and invokes any pending I/O callbacks. If you were to call `uv_stop()` in any of them, `stop_flag` would be set. This causes `uv_backend_timeout()` to return `0`, which is why the loop does

not block on I/O. If on the other hand, you called `uv_stop()` in one of the check handlers, I/O has already finished and is not affected.

`uv_stop()` is useful to shutdown a loop when a result has been computed or there is an error, without having to ensure that all handlers are stopped one by one.

Here is a simple example that stops the loop and demonstrates how the current iteration of the loop still takes places.

uvstop/main.c

```
#include <stdio.h>
#include <uv.h>

int64_t counter = 0;

void idle_cb(uv_idle_t *handle) {
    printf("Idle callback\n");
    counter++;

    if (counter >= 5) {
        uv_stop(uv_default_loop());
        printf("uv_stop() called\n");
    }
}

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

int main() {
    uv_idle_t idler;
    uv_prepare_t prep;

    uv_idle_init(uv_default_loop(), &idler);
    uv_idle_start(&idler, idle_cb);

    uv_prepare_init(uv_default_loop(), &prep);
    uv_prepare_start(&prep, prep_cb);

    uv_run(uv_default_loop(), UV_RUN_DEFAULT);

    return 0;
}
```

3.3.8 Utilities

This chapter catalogues tools and techniques which are useful for common tasks. The libev man page already covers some patterns which can be adopted to libuv through simple API changes. It also covers parts of the libuv API that don't require entire chapters dedicated to them.

Timers

Timers invoke the callback after a certain time has elapsed since the timer was started. libuv timers can also be set to invoke at regular intervals instead of just once.

Simple use is to init a watcher and start it with a `timeout`, and optional `repeat`. Timers can be stopped at any time.

```
uv_timer_t timer_req;

uv_timer_init(loop, &timer_req);
uv_timer_start(&timer_req, callback, 5000, 2000);
```

will start a repeating timer, which first starts 5 seconds (the `timeout`) after the execution of `uv_timer_start`, then repeats every 2 seconds (the `repeat`). Use:

```
uv_timer_stop(&timer_req);
```

to stop the timer. This can be used safely from within the callback as well.

The repeat interval can be modified at any time with:

```
uv_timer_set_repeat(uv_timer_t *timer, int64_t repeat);
```

which will take effect **when possible**. If this function is called from a timer callback, it means:

- If the timer was non-repeating, the timer has already been stopped. Use `uv_timer_start` again.
- If the timer is repeating, the next timeout has already been scheduled, so the old repeat interval will be used once more before the timer switches to the new interval.

The utility function:

```
int uv_timer_again(uv_timer_t *)
```

applies **only to repeating timers** and is equivalent to stopping the timer and then starting it with both initial `timeout` and `repeat` set to the old `repeat` value. If the timer hasn't been started it fails (error code `UV_EINVAL`) and returns -1.

An actual timer example is in the *reference count section*.

Event loop reference count

The event loop only runs as long as there are active handles. This system works by having every handle increase the reference count of the event loop when it is started and decreasing the reference count when stopped. It is also possible to manually change the reference count of handles using:

```
void uv_ref(uv_handle_t*);
void uv_unref(uv_handle_t*);
```

These functions can be used to allow a loop to exit even when a watcher is active or to use custom objects to keep the loop alive.

The latter can be used with interval timers. You might have a garbage collector which runs every X seconds, or your network service might send a heartbeat to others periodically, but you don't want to have to stop them along all clean exit paths or error scenarios. Or you want the program to exit when all your other watchers are done. In that case just unref the timer immediately after creation so that if it is the only watcher running then `uv_run` will still exit.

This is also used in node.js where some libuv methods are being bubbled up to the JS API. A `uv_handle_t` (the superclass of all watchers) is created per JS object and can be ref/unrefed.

ref-timer/main.c

```
uv_loop_t *loop;
uv_timer_t gc_req;
uv_timer_t fake_job_req;

int main() {
    loop = uv_default_loop();

    uv_timer_init(loop, &gc_req);
    uv_unref((uv_handle_t*) &gc_req);

    uv_timer_start(&gc_req, gc, 0, 2000);

    // could actually be a TCP download or something
    uv_timer_init(loop, &fake_job_req);
    uv_timer_start(&fake_job_req, fake_job, 9000, 0);
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

We initialize the garbage collector timer, then immediately `unref` it. Observe how after 9 seconds, when the fake job is done, the program automatically exits, even though the garbage collector is still running.

Idler pattern

The callbacks of idle handles are invoked once per event loop. The idle callback can be used to perform some very low priority activity. For example, you could dispatch a summary of the daily application performance to the developers for analysis during periods of idleness, or use the application's CPU time to perform SETI calculations :) An idle watcher is also useful in a GUI application. Say you are using an event loop for a file download. If the TCP socket is still being established and no other events are present your event loop will pause (**block**), which means your progress bar will freeze and the user will face an unresponsive application. In such a case queue up and idle watcher to keep the UI operational.

idle-compute/main.c

```
uv_loop_t *loop;
uv_fs_t stdin_watcher;
uv_idle_t idler;
char buffer[1024];

int main() {
    loop = uv_default_loop();

    uv_idle_init(loop, &idler);

    uv_buf_t buf = uv_buf_init(buffer, 1024);
    uv_fs_read(loop, &stdin_watcher, 0, &buf, 1, -1, on_type);
    uv_idle_start(&idler, crunch_away);
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

Here we initialize the idle watcher and queue it up along with the actual events we are interested in. `crunch_away` will now be called repeatedly until the user types something and presses Return. Then it will be interrupted for a brief amount as the loop deals with the input data, after which it will keep calling the idle callback again.

idle-compute/main.c

```
void crunch_away(uv_idle_t* handle) {
    // Compute extra-terrestrial life
    // fold proteins
    // computer another digit of PI
    // or similar
    fprintf(stderr, "Computing PI...\n");
    // just to avoid overwhelming your terminal emulator
    uv_idle_stop(handle);
}

```

Passing data to worker thread

When using `uv_queue_work` you'll usually need to pass complex data through to the worker thread. The solution is to use a `struct` and set `uv_work_t.data` to point to it. A slight variation is to have the `uv_work_t` itself as the first member of this struct (called a baton[1]). This allows cleaning up the work request and all the data in one free call.

```
struct ftp_baton {
    uv_work_t req;
    char *host;
    int port;
    char *username;
    char *password;
}
```

```
ftp_baton *baton = (ftp_baton*) malloc(sizeof(ftp_baton));
baton->req.data = (void*) baton;
baton->host = strdup("my.webhost.com");
baton->port = 21;
// ...

uv_queue_work(loop, &baton->req, ftp_session, ftp_cleanup);
```

Here we create the baton and queue the task.

Now the task function can extract the data it needs:

```
void ftp_session(uv_work_t *req) {
    ftp_baton *baton = (ftp_baton*) req->data;

    fprintf(stderr, "Connecting to %s\n", baton->host);
}

void ftp_cleanup(uv_work_t *req) {
    ftp_baton *baton = (ftp_baton*) req->data;

    free(baton->host);
    // ...
    free(baton);
}
```

[1] I was first introduced to the term baton in this context, in Konstantin Käfer's excellent slides on writing node.js bindings – http://kkaefer.github.com/node-cpp-modules/#baton

We then free the baton which also frees the watcher.

External I/O with polling

Usually third-party libraries will handle their own I/O, and keep track of their sockets and other files internally. In this case it isn't possible to use the standard stream I/O operations, but the library can still be integrated into the libuv event loop. All that is required is that the library allow you to access the underlying file descriptors and provide functions that process tasks in small increments as decided by your application. Some libraries though will not allow such access, providing only a standard blocking function which will perform the entire I/O transaction and only then return. It is unwise to use these in the event loop thread, use the libuv-work-queue instead. Of course, this will also mean losing granular control on the library.

The `uv_poll` section of libuv simply watches file descriptors using the operating system notification mechanism. In some sense, all the I/O operations that libuv implements itself are also backed by `uv_poll` like code. Whenever the OS notices a change of state in file descriptors being polled, libuv will invoke the associated callback.

Here we will walk through a simple download manager that will use libcurl to download files. Rather than give all control to libcurl, we'll instead be using the libuv event loop, and use the non-blocking, async multi interface to progress with the download whenever libuv notifies of I/O readiness.

uvwget/main.c - The setup

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <uv.h>
#include <curl/curl.h>

uv_loop_t *loop;
CURLM *curl_handle;
uv_timer_t timeout;
}

int main(int argc, char **argv) {
    loop = uv_default_loop();

    if (argc <= 1)
        return 0;

    if (curl_global_init(CURL_GLOBAL_ALL)) {
        fprintf(stderr, "Could not init cURL\n");
        return 1;
    }

    uv_timer_init(loop, &timeout);

    curl_handle = curl_multi_init();
    curl_multi_setopt(curl_handle, CURLMOPT_SOCKETFUNCTION, handle_socket);
    curl_multi_setopt(curl_handle, CURLMOPT_TIMERFUNCTION, start_timeout);

    while (argc-- > 1) {
        add_download(argv[argc], argc);
    }

    uv_run(loop, UV_RUN_DEFAULT);
```

(continues on next page)

(续上页)

```
    curl_multi_cleanup(curl_handle);
    return 0;
}
```

The way each library is integrated with libuv will vary. In the case of libcurl, we can register two callbacks. The socket callback `handle_socket` is invoked whenever the state of a socket changes and we have to start polling it. `start_timeout` is called by libcurl to notify us of the next timeout interval, after which we should drive libcurl forward regardless of I/O status. This is so that libcurl can handle errors or do whatever else is required to get the download moving.

Our downloader is to be invoked as:

```
$ ./uvwget [url1] [url2] ...
```

So we add each argument as an URL

uvwget/main.c - Adding urls

```

void add_download(const char *url, int num) {
    char filename[50];
    sprintf(filename, "%d.download", num);
    FILE *file;

    file = fopen(filename, "w");
    if (file == NULL) {
        fprintf(stderr, "Error opening %s\n", filename);
        return;
    }

    CURL *handle = curl_easy_init();
    curl_easy_setopt(handle, CURLOPT_WRITEDATA, file);
    curl_easy_setopt(handle, CURLOPT_URL, url);
    curl_multi_add_handle(curl_handle, handle);
    fprintf(stderr, "Added download %s -> %s\n", url, filename);
}
```

We let libcurl directly write the data to a file, but much more is possible if you so desire.

`start_timeout` will be called immediately the first time by libcurl, so things are set in motion. This simply starts a libuv timer which drives `curl_multi_socket_action` with `CURL_SOCKET_TIMEOUT` whenever it times out. `curl_multi_socket_action` is what drives libcurl, and what we call whenever sockets change state. But before we go into that, we need to poll on sockets whenever `handle_socket` is called.

uvwget/main.c - Setting up polling

```

void start_timeout(CURLM *multi, long timeout_ms, void *userp) {
    if (timeout_ms <= 0)
        timeout_ms = 1; /* 0 means directly call socket_action, but we'll do it in a␣
↪bit */
    uv_timer_start(&timeout, on_timeout, timeout_ms, 0);
}
```

(continues on next page)

(续上页)

```
int handle_socket(CURL *easy, curl_socket_t s, int action, void *userp, void
↪*socketp) {
    curl_context_t *curl_context;
    if (action == CURL_POLL_IN || action == CURL_POLL_OUT) {
        if (socketp) {
            curl_context = (curl_context_t*) socketp;
        }
        else {
            curl_context = create_curl_context(s);
            curl_multi_assign(curl_handle, s, (void *) curl_context);
        }
    }

    switch (action) {
        case CURL_POLL_IN:
            uv_poll_start(&curl_context->poll_handle, UV_READABLE, curl_perform);
            break;
        case CURL_POLL_OUT:
            uv_poll_start(&curl_context->poll_handle, UV_WRITABLE, curl_perform);
            break;
        case CURL_POLL_REMOVE:
            if (socketp) {
                uv_poll_stop(&((curl_context_t*)socketp)->poll_handle);
                destroy_curl_context((curl_context_t*) socketp);
                curl_multi_assign(curl_handle, s, NULL);
            }
            break;
        default:
            abort();
    }

    return 0;
}
```

We are interested in the socket fd `s`, and the `action`. For every socket we create a `uv_poll_t` handle if it doesn't exist, and associate it with the socket using `curl_multi_assign`. This way `socketp` points to it whenever the callback is invoked.

In the case that the download is done or fails, libcurl requests removal of the poll. So we stop and free the poll handle.

Depending on what events libcurl wishes to watch for, we start polling with `UV_READABLE` or `UV_WRITABLE`. Now libuv will invoke the poll callback whenever the socket is ready for reading or writing. Calling `uv_poll_start` multiple times on the same handle is acceptable, it will just update the events mask with the new value. `curl_perform` is the crux of this program.

uvwget/main.c - Driving libcurl.

```
void curl_perform(uv_poll_t *req, int status, int events) {
    uv_timer_stop(&timeout);
    int running_handles;
    int flags = 0;
    if (status < 0)                      flags = CURL_CSELECT_ERR;
    if (!status && events & UV_READABLE) flags |= CURL_CSELECT_IN;
    if (!status && events & UV_WRITABLE) flags |= CURL_CSELECT_OUT;
```

(continues on next page)

(续上页)

```
8
9       curl_context_t *context;
10
11      context = (curl_context_t*)req;
12
13      curl_multi_socket_action(curl_handle, context->sockfd, flags, &running_handles);
14      check_multi_info();
15  }
```

The first thing we do is to stop the timer, since there has been some progress in the interval. Then depending on what event triggered the callback, we set the correct flags. Then we call `curl_multi_socket_action` with the socket that progressed and the flags informing about what events happened. At this point libcurl does all of its internal tasks in small increments, and will attempt to return as fast as possible, which is exactly what an evented program wants in its main thread. libcurl keeps queueing messages into its own queue about transfer progress. In our case we are only interested in transfers that are completed. So we extract these messages, and clean up handles whose transfers are done.

uvwget/main.c - Reading transfer status.

```
1   void check_multi_info(void) {
2       char *done_url;
3       CURLMsg *message;
4       int pending;
5
6       while ((message = curl_multi_info_read(curl_handle, &pending))) {
7           switch (message->msg) {
8           case CURLMSG_DONE:
9               curl_easy_getinfo(message->easy_handle, CURLINFO_EFFECTIVE_URL,
10                                &done_url);
11              printf("%s DONE\n", done_url);
12
13              curl_multi_remove_handle(curl_handle, message->easy_handle);
14              curl_easy_cleanup(message->easy_handle);
15              break;
16
17          default:
18              fprintf(stderr, "CURLMSG default\n");
19              abort();
20          }
21      }
22  }
```

Check & Prepare watchers

TODO

Loading libraries

libuv provides a cross platform API to dynamically load shared libraries. This can be used to implement your own plugin/extension/module system and is used by node.js to implement `require()` support for bindings. The usage is quite simple as long as your library exports the right symbols. Be careful with sanity and security checks when

loading third party code, otherwise your program will behave unpredictably. This example implements a very simple plugin system which does nothing except print the name of the plugin.

Let us first look at the interface provided to plugin authors.

plugin/plugin.h

```
#ifndef UVBOOK_PLUGIN_SYSTEM
#define UVBOOK_PLUGIN_SYSTEM

// Plugin authors should use this to register their plugins with mfp.
void mfp_register(const char *name);

#endif
```

You can similarly add more functions that plugin authors can use to do useful things in your application[2]. A sample plugin using this API is:

plugin/hello.c

```
#include "plugin.h"

void initialize() {
    mfp_register("Hello World!");
}
```

Our interface defines that all plugins should have an `initialize` function which will be called by the application. This plugin is compiled as a shared library and can be loaded by running our application:

```
$ ./plugin libhello.dylib
Loading libhello.dylib
Registered plugin "Hello World!"
```

注解： The shared library filename will be different depending on platforms. On Linux it is `libhello.so`.

This is done by using `uv_dlopen` to first load the shared library `libhello.dylib`. Then we get access to the `initialize` function using `uv_dlsym` and invoke it.

plugin/main.c

```
#include "plugin.h"

typedef void (*init_plugin_function)();

void mfp_register(const char *name) {
    fprintf(stderr, "Registered plugin \"%s\"\n", name);
}

int main(int argc, char **argv) {
```

(continues on next page)

[2] mfp is My Fancy Plugin

(续上页)

```
    if (argc == 1) {
        fprintf(stderr, "Usage: %s [plugin1] [plugin2] ...\n", argv[0]);
        return 0;
    }

    uv_lib_t *lib = (uv_lib_t*) malloc(sizeof(uv_lib_t));
    while (--argc) {
        fprintf(stderr, "Loading %s\n", argv[argc]);
        if (uv_dlopen(argv[argc], lib)) {
            fprintf(stderr, "Error: %s\n", uv_dlerror(lib));
            continue;
        }

        init_plugin_function init_plugin;
        if (uv_dlsym(lib, "initialize", (void **) &init_plugin)) {
            fprintf(stderr, "dlsym error: %s\n", uv_dlerror(lib));
            continue;
        }

        init_plugin();
    }

    return 0;
}
```

`uv_dlopen` expects a path to the shared library and sets the opaque `uv_lib_t` pointer. It returns 0 on success, -1 on error. Use `uv_dlerror` to get the error message.

`uv_dlsym` stores a pointer to the symbol in the second argument in the third argument. `init_plugin_function` is a function pointer to the sort of function we are looking for in the application's plugins.

TTY

Text terminals have supported basic formatting for a long time, with a pretty standardised command set. This formatting is often used by programs to improve the readability of terminal output. For example `grep --colour`. libuv provides the `uv_tty_t` abstraction (a stream) and related functions to implement the ANSI escape codes across all platforms. By this I mean that libuv converts ANSI codes to the Windows equivalent, and provides functions to get terminal information.

The first thing to do is to initialize a `uv_tty_t` with the file descriptor it reads/writes from. This is achieved with:

```
int uv_tty_init(uv_loop_t*, uv_tty_t*, uv_file fd, int unused)
```

The `unused` parameter is now auto-detected and ignored. It previously needed to be set to use `uv_read_start()` on the stream.

It is then best to use `uv_tty_set_mode` to set the mode to *normal* which enables most TTY formatting, flow-control and other settings. Other modes are also available.

Remember to call `uv_tty_reset_mode` when your program exits to restore the state of the terminal. Just good manners. Another set of good manners is to be aware of redirection. If the user redirects the output of your command to a file, control sequences should not be written as they impede readability and `grep`. To check if the file descriptor is indeed a TTY, call `uv_guess_handle` with the file descriptor and compare the return value with `UV_TTY`.

Here is a simple example which prints white text on a red background:

tty/main.c

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <uv.h>

uv_loop_t *loop;
uv_tty_t tty;
int main() {
    loop = uv_default_loop();

    uv_tty_init(loop, &tty, STDOUT_FILENO, 0);
    uv_tty_set_mode(&tty, UV_TTY_MODE_NORMAL);

    if (uv_guess_handle(1) == UV_TTY) {
        uv_write_t req;
        uv_buf_t buf;
        buf.base = "\033[41;37m";
        buf.len = strlen(buf.base);
        uv_write(&req, (uv_stream_t*) &tty, &buf, 1, NULL);
    }

    uv_write_t req;
    uv_buf_t buf;
    buf.base = "Hello TTY\n";
    buf.len = strlen(buf.base);
    uv_write(&req, (uv_stream_t*) &tty, &buf, 1, NULL);
    uv_tty_reset_mode();
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

The final TTY helper is `uv_tty_get_winsize()` which is used to get the width and height of the terminal and returns `0` on success. Here is a small program which does some animation using the function and character position escape codes.

tty-gravity/main.c

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <uv.h>

uv_loop_t *loop;
uv_tty_t tty;
uv_timer_t tick;
uv_write_t write_req;
int width, height;
int pos = 0;
char *message = "  Hello TTY  ";

void update(uv_timer_t *req) {
    char data[500];

    uv_buf_t buf;
```

(continues on next page)

(续上页)

```
    buf.base = data;
    buf.len = sprintf(data, "\033[2J\033[H\033[%dB\033[%luC\033[42;37m%s",
                              pos,
                              (unsigned long) (width-strlen(message))/2,
                              message);
    uv_write(&write_req, (uv_stream_t*) &tty, &buf, 1, NULL);

    pos++;
    if (pos > height) {
        uv_tty_reset_mode();
        uv_timer_stop(&tick);
    }
}

int main() {
    loop = uv_default_loop();

    uv_tty_init(loop, &tty, STDOUT_FILENO, 0);
    uv_tty_set_mode(&tty, 0);

    if (uv_tty_get_winsize(&tty, &width, &height)) {
        fprintf(stderr, "Could not get TTY information\n");
        uv_tty_reset_mode();
        return 1;
    }

    fprintf(stderr, "Width %d, height %d\n", width, height);
    uv_timer_init(loop, &tick);
    uv_timer_start(&tick, update, 200, 200);
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

The escape codes are:

Code	Meaning
2 J	Clear part of the screen, 2 is entire screen
H	Moves cursor to certain position, default top-left
n B	Moves cursor down by n lines
n C	Moves cursor right by n columns
m	Obeys string of display settings, in this case green background (40+2), white text (30+7)

As you can see this is very useful to produce nicely formatted output, or even console based arcade games if that tickles your fancy. For fancier control you can try ncurses.

在 1.23.1: 版更改: the *readable* parameter is now unused and ignored. The appropriate value will now be auto-detected from the kernel.

3.3.9 About

Nikhil Marathe started writing this book one afternoon (June 16, 2012) when he didn't feel like programming. He had recently been stung by the lack of good documentation on libuv while working on node-taglib. Although reference documentation was present, there were no comprehensive tutorials. This book is the output of that need and tries to be accurate. That said, the book may have mistakes. Pull requests are encouraged.

Nikhil is indebted to Marc Lehmann's comprehensive man page about libev which describes much of the semantics of the two libraries.

This book was made using Sphinx and vim.

注解: In 2017 the libuv project incorporated the Nikhil's work into the official documentation and it's maintained there henceforth.

3.4 版本升级

对不同libuv版本的迁移指南，开始于1.0。

3.4.1 libuv 0.10 -> 1.0.0 迁移指南

一些API在1.0.0开发过程当中发生了很大的变化。 这是一份迁移指南，关于在0.10发布后发生的最重要的变化。

循环初始化和关闭

在libuv 0.10（和任何之前的版本中），循环通过 *uv_loop_new* 创建， 为新的循环分配内存且初始化它；同时通过 *uv_loop_delete*， 销毁循环且释放内存。 从1.0开始，这些被废弃了 并且用户负责分配内存和初始化循环。

libuv 0.10

```
uv_loop_t* loop = uv_loop_new();
...
uv_loop_delete(loop);
```

libuv 1.0

```
uv_loop_t* loop = malloc(sizeof *loop);
uv_loop_init(loop);
...
uv_loop_close(loop);
free(loop);
```

注解: 错误处理为了简洁被省略了。 查看文档 *uv_loop_init()* 和 *uv_loop_close()*.

错误处理

错误处理在libuv 1.0中有巨大的翻修。 总的来说，在libuv 0.10中，函数和状态参数里 0 代表成功和 -1 代表失败，用户不得不使用 *uv_last_error* 获取错误代码，这是个正数。

在1.0中，函数和状态参数包括了确切的错误代码，0 代表成功， 错误的时候为一个负数。

libuv 0.10

```
... 假如 'server' 是一个已经在侦听中的TCP服务器
r = uv_listen((uv_stream_t*) server, 511, NULL);
if (r == -1) {
  uv_err_t err = uv_last_error(uv_default_loop());
  /* err.code 包括 UV_EADDRINUSE */
}
```

libuv 1.0

```
... 假如 'server' 是一个已经在侦听中的TCP服务器
r = uv_listen((uv_stream_t*) server, 511, NULL);
if (r < 0) {
  /* r 包括 UV_EADDRINUSE */
}
```

线程池变化

在libuv 0.10当中，Unix下用了一个默认是4个线程的线程池，而Windows下用了 *QueueUserWorkItem* API，这使用了一个Windows内部的线程池，默认是每个进程 512 个线程。

在1.0中，我们统一了两种实现，这样Windows现在使用和Unix同样的实现。 线程池的大小可以被外部环境变量 `UV_THREADPOOL_SIZE` 设置。详见 *Thread pool work scheduling*。

分配回调函数 API变化

在libuv 0.10中，回调函数必须返回一个 *uv_buf_t* 类型的填充值:

```
uv_buf_t alloc_cb(uv_handle_t* handle, size_t size) {
    return uv_buf_init(malloc(size), size);
}
```

在libuv 1.0中，一个指向一个buffer的指针传递给回调函数，用户需要填充它:

```
void alloc_cb(uv_handle_t* handle, size_t size, uv_buf_t* buf) {
    buf->base = malloc(size);
    buf->len = size;
}
```

IPv4 / IPv6 API的统一

libuv 1.0统一了IPv4和IPv6 API。 再也没有 *uv_tcp_bind* 和 *uv_tcp_bind6* 这样的二元性， 现在只有 *uv_tcp_bind()* 。

IPv4函数采用 `struct sockaddr_in` 结构体的值，IPv6函数采用 `struct sockaddr_in6`。现在，函数采用 `struct sockaddr*` （注意，它是一个指针）。 它能在栈上分配。

libuv 0.10

```
struct sockaddr_in addr = uv_ip4_addr("0.0.0.0", 1234);
...
uv_tcp_bind(&server, addr)
```

libuv 1.0

```
struct sockaddr_in addr;
uv_ip4_addr("0.0.0.0", 1234, &addr)
...
uv_tcp_bind(&server, (const struct sockaddr*) &addr, 0);
```

IPv4和IPv6结构创建函数（ *uv_ip4_addr()* 和 *uv_ip6_addr()* ） 也变了，确保使用前你查看了文档。

注解： 这些变化适用于所有区分IPv4和IPv6地址的函数。

流 / UDP数据接收回调函数 API变化

流和UDP数据接收回调函数现在接受一个指向 *uv_buf_t* 缓冲区的指针， 而不是一个结构体值。

libuv 0.10

```
void on_read(uv_stream_t* handle,
             ssize_t nread,
             uv_buf_t buf) {
    ...
}

void recv_cb(uv_udp_t* handle,
             ssize_t nread,
             uv_buf_t buf,
             struct sockaddr* addr,
             unsigned flags) {
    ...
}
```

libuv 1.0

```
void on_read(uv_stream_t* handle,
             ssize_t nread,
             const uv_buf_t* buf) {
    ...
}

void recv_cb(uv_udp_t* handle,
             ssize_t nread,
             const uv_buf_t* buf,
             const struct sockaddr* addr,
             unsigned flags) {
    ...
}
```

通过管道的接收句柄 API变化

在libuv 0.10（和之前版本）中， *uv_read2_start* 函数被用来开始在管道上读数据， 可能致使在其上接收句柄。 这类函数的回调函数看起来像这样:

```
void on_read(uv_pipe_t* pipe,
             ssize_t nread,
             uv_buf_t buf,
```

(continues on next page)

(续上页)

```
                uv_handle_type pending) {
    ...
}
```

在libuv 1.0中， *uv_read2_start* 被移除了，并且在读回调函数当中， 用户需要检查是否有待处理的句柄，使用 *uv_pipe_pending_count()* 和 *uv_pipe_pending_type()* ：

```
void on_read(uv_stream_t* handle,
             ssize_t nread,
             const uv_buf_t* buf) {
    ...
    while (uv_pipe_pending_count((uv_pipe_t*) handle) != 0) {
        pending = uv_pipe_pending_type((uv_pipe_t*) handle);
        ...
    }
    ...
}
```

从句柄里提取文件描述符

当它还不被API支持的时候，用户通常访问libuv内部变量， 以获取例如TCP句柄的文件访问符。

```
fd = handle->io_watcher.fd;
```

这现在已经正式地暴露于 *uv_fileno()* 函数。

uv_fs_readdir重命名和API变化

在libuv 0.10中，当完成时 *uv_fs_readdir* 返回一个字符串列表于 *req->ptr* 字段。 在1.0中，这个函数被重命名为 *uv_fs_scandir()* ， 因为这确实通过 `scandir(3)` 实现。

另外，用户可以使用 *uv_fs_scandir_next()* 函数一次获取一个结果，而不是分配完整的列表字符串。这个函数不需要往返于线程池，因为libuv将保持 `scandir(3)` 返回的 凹齿 列表在一旁。

CHAPTER 4

下载

libuv 可在 这儿 下载。

CHAPTER 5

安装

安装步骤详见 README 。

U