Tree: 62e84b27e7 liuyanjie.github.io / source / _posts / 0-libuv源码分析（一）全局概览（Overview）.md

Find file Copy path

 liuyanjie add articles 2c06277 on Apr 24

1 contributor

285 lines (194 sloc) | 18.6 KB

Raw Blame History

title	date	updated	tags	categories

title	date	updated	tags	categories
libuv源码分析（一）全局概览（Overview）	2019-04-23 15:00:01 UTC	2019-04-24 01:31:28 UTC	libuv, node.js, eventloop	源码分析

简介

libuv 是一个专注于异步I/O的跨平台的程序库，它主要是用于支持 Node.js，但是也被如 Luvit、Julia、pyuv 等很多其他的库使用。它使得 异步I/O 变的简单。

libuv 对于经常接触 `c` 语言程序开发的开发人员来说，应该是非常容易的，但是对于没有没有相关经验的开发人员来说，阅读源码就比较吃力了，但是如果能阅读并理解 `libuv`，才能更深入的了解 `Node.js` 是怎样工作的，以及 `Node.js` 中的事件驱动、非阻塞异步I/O是怎么一回事儿，它是怎么解决这些问题的，它有哪些不足。

基础要求

阅读源代码需要 `c` 语言相关的知识和一定的系统编程基础，需要了解常见的系统API调用及相关机制。以下为详细说明：

1. libuv 主要是用 `c` 实现的，所以需要有一定的 `c` 语言基础，尤其是宏、结构体、指针、函数指针、数组等内容，对 `c` 语言程序的内存布局有一定了解；
2. libuv 内部包含 `堆` `队列` `树` 等基础的数据结构，需要有一定的了解；
3. libuv 内部使用了很多与操作系统交互的系统API，所以需要对相应系统平台系统编程和操作系统原理有一定基础，了解进程、线程、文件系统、网络等，了解常见的异步IO模型，最好阅读过 `APUE`。

libuv 内部实现采用的大量的 `宏` 来复用代码，并配合使用强制类型等机制实现类似于高级语言中继承的效果，这依赖于 `c` 语言结构体内存布局，`宏` 本身是非常不容易阅读的，增加了阅读源代码的难度。同时内部进行一定的抽象，

需要一定的理解能力。

libuv 支持多个平台，尤其是支持 `*nix` 和 `win` 两中设计实现差异很大的操作系统，其内部也大量使用宏进行条件编译，且部分功能在不同环境下的实现有可能完全不同，所以在阅读源码时只关注某一平台代码。本源码分析也仅针对 `linux` 环境的相关代码的分析。

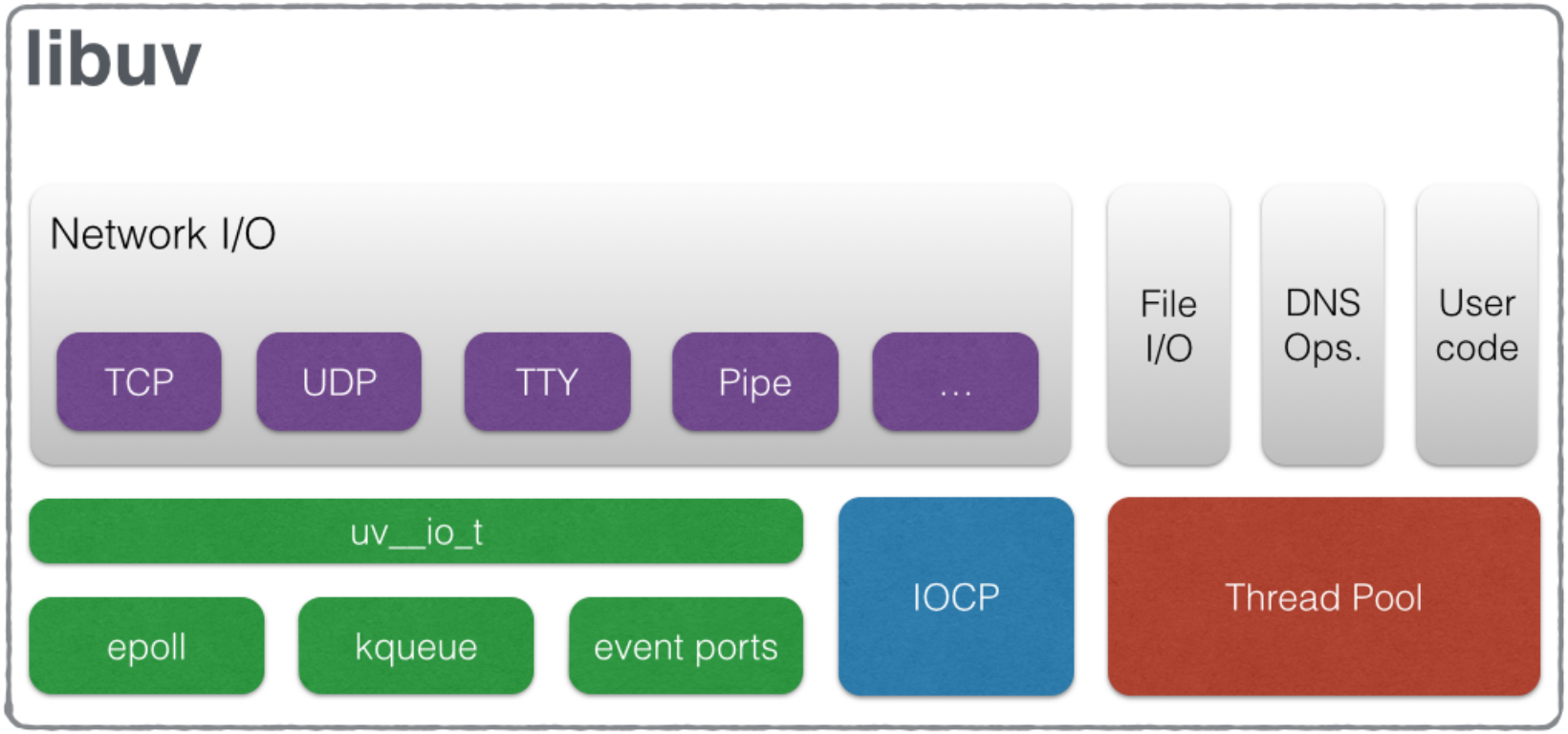
设计概述

本部分内容建议对照官方文档 Design overview 阅读。

libuv 最初是为 NodeJS 编写的跨平台支持库。它是围绕`事件驱动的异步I/O模型`设计的。提到异步非阻塞I/O模型，有经验的开发人员应该能很快意识到这是一种很常见的实现高并发的模型，这是一种不同于多进程/多线程的并发模型。并发模型可以参考知乎专栏并发模型之间的比较或其他书籍或文章。

libuv 不仅仅在多种不同I/O轮询机制之上提供了的简单抽象——I/O观察者，更通过 `handles` 和 `streams` 为套接字和其他实体提供了更高级别的抽象，同时也提供了跨平台I/O和线程管理能力，还包含一些其他功能。

下图说明了 libuv 的不同组成部分以及它们与哪些子系统相关：



从图中可以看到，libuv 主要部分都是和I/O相关的，主要包括网络I/O和文件I/O，其中文件I/O和其他少部分功能基于线程池实现，网络I/O在 `*nix` 平台基于 `uv__io_t` （内部抽象的I/O观察者）实现， `uv__io_t` 又基于不同环境采用了不同的底层机制，网络I/O在 `win` 平台基于 `IOCP` 机制实现。

Handles and requests

libuv 为用户提供了两个与实践循环结合使用的抽象： `handles` 和 `requests`

`Handles` 代表长生命周期的 `对象` 有能力执行某些操作当它们处于激活状态下。例如：

- `prepare handle` 在激活时，每次事件循环迭代都会调用一次它的回调；

- `TCP server handle` 在每一次有一个新的 `connection` 进来的时候都会调用一次它的回调。

`Requests` 一般代表一个短生命周期的`操作`。这些操作可以通过 handle 执行：`write requests` 用于在 handle 上写数据，所以 `request` 和 `handle` 可能存在一定的数据关联；或者也可以独立执行：`getaddrinfo requests` 不需要一个 `handle`，他们直接运行在事件循环中。

相关的代码分析见：Handle and Requst

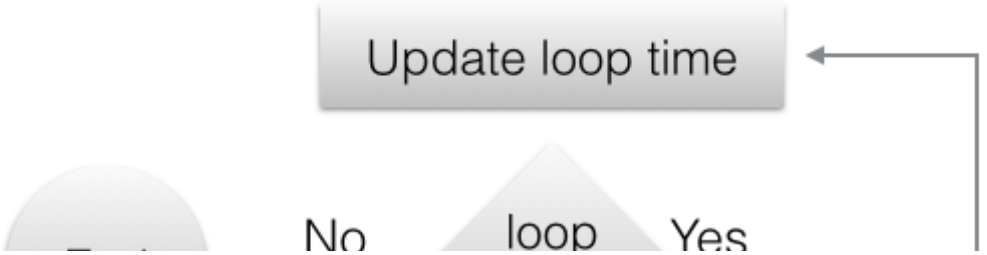
The I/O loop

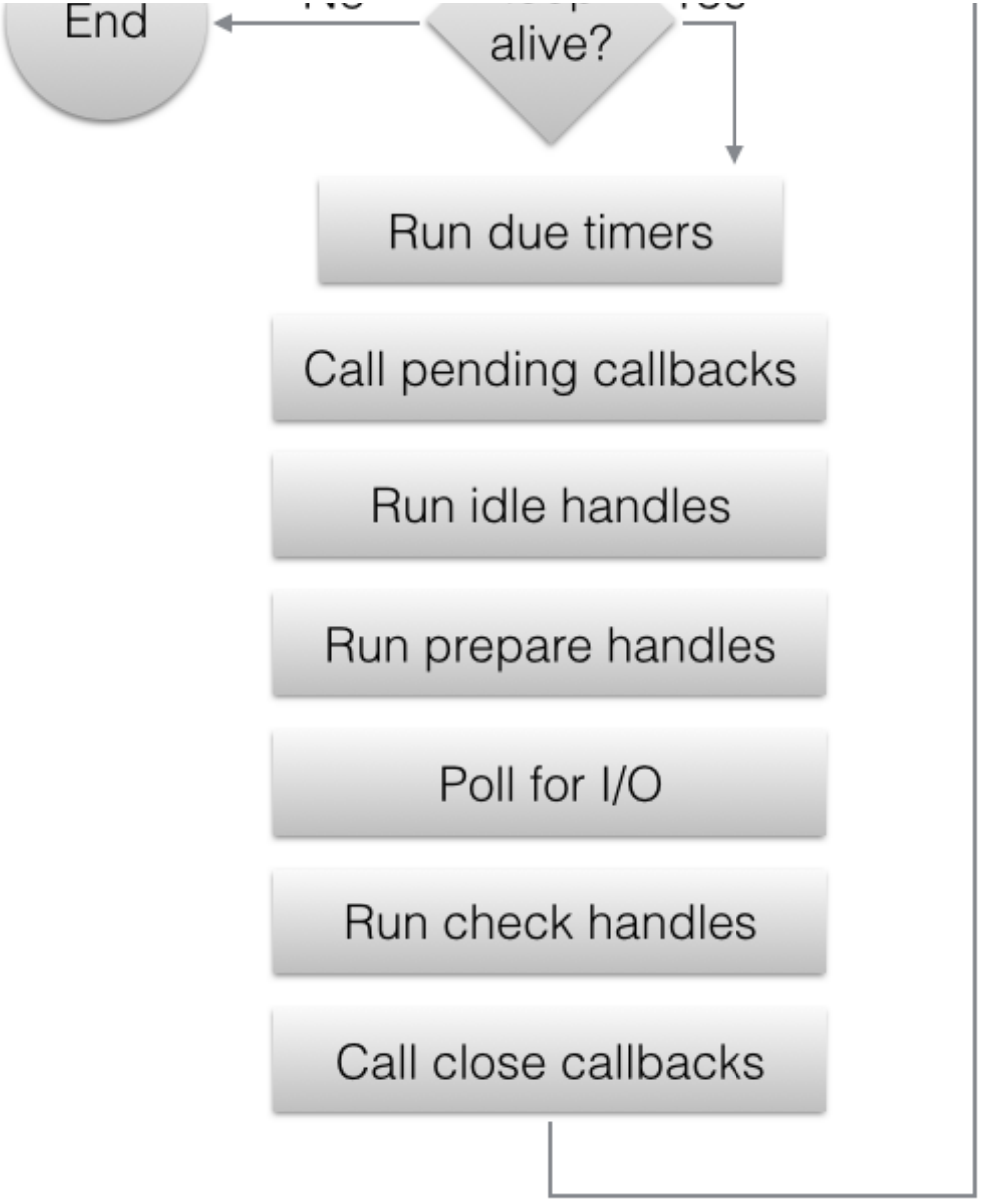
I/O loop 也就是事件循环（Event Loop）是 libuv 的核心组成部分。它为所有 I/O 操作建立内容，实际上这意味着事件循环被绑定到一个单一的线程。可以运行多个不同的事件循环只要它们在不同的线程中。除非另有说明，事件循环（任何涉及事件循环和 handle 的API）并不是线程安全的。

所有的异步操作的结果最终在事件循环中被处理，也就是通常所说的回调函数，在事件循环中被调用。

事件循环是非常常见的单线程异步I/O的处理方法：所有（网络）I/O都在非阻塞的套接字上执行，这些套接字使用给定平台上可用的最佳机制进行轮询：Linux 上使用 `epoll`，OSX 和其他 BSDs 系统上使用 `kqueue`，SunOS 使用 `event ports`，Windows 上使用 `IOCP`。以上I/O轮询作为事件循环迭代的一部分，事件循环将会被阻塞在I/O轮询（例如：linux 上的 `epoll_pwait` 调用），直到被添加到轮询器中的套接字有IO活动（事件），事件循环线程将会在有IO事件时被唤醒，关联的回调函数将会被调用表明套接字有新的连接，然后便可以在 `handles` 上进行读、写或其他想要进行的操作 `requests`。

下图显示了事件循环的所有阶段：





图中主要有七个阶段，

其中 `idle`、`prepare`、`check` 的实现完全相同，调用时间不同，类似于生命周期勾子，这几个阶段目的是允许开发者在事件循环的特定阶段执行代码，在 Node.js 用于性能信息收集。这三个阶段的实现代码比较简单，很容易理解。因源代码几乎完全使用宏实现，所以编辑器无法跳转到对应实现，搜索关键字也无法匹配，这里给出源文件路径：`src/unix/loop-watcher.c`，便于读者找到源文件。

其余剩下的阶段就主要有 `Call pending callbacks` `Poll for I/O` `Call close callbacks`，这三个阶段主要用于处理IO操作等异步操作结果，阅读源码也主要是围绕着这三个阶段的代码展开的。

各阶段用途描述：

1. Run due timers：处理定时任务；
2. Call pending callbacks：处理上一轮事件循环中因出现错误或者逻辑需要等原因挂起的任务；
3. Run idle handles；
4. Run prepare handles；
5. Poll for I/O：事件循环 则有可能 因为 `epoll_wait` 而阻塞在这里，这取决于 `timeout` 参数是否为 `0`，但是通常情况下，会阻塞到有关注的IO事件发送时回，这也直接避免了时间循环一直工作导致占用CPU的问题。这个阶段是整个 libuv 事件循环最重要的阶段。libuv 的大部分 `handle` 都依赖该阶段实现。
6. Run check handles：
7. Call close callbacks：清理被关闭的 `handles`。

更多详细描述，请直接阅读The I/O loop的详细描述。

相关的代码分析见：EventLoop

File I/O

不同于网络IO，目前没有 libuv 可以依赖的平台特定（异步）文件IO机制，所以当前的方式是在线程池中运行阻塞式的文件IO操作。

libuv 目前采用一个全局的线程池，所有事件循环都可以在向其任务队列提交任务，目前有3种类型的操作运行在线程中：

- 文件系统操作：`read`、`write` ...
- DNS功能操作：`getaddrinfo` 和 `getnameinfo`
- 通过 `uv_queue_work()` 提交的用户特定的任务

有关跨平台文件I/O现状的详尽说明：asynchronous disk I/O

线程池中的线程在工作完成之后，会向事件循环线程发送数据，然后再主线程中触发回调。

Reactor

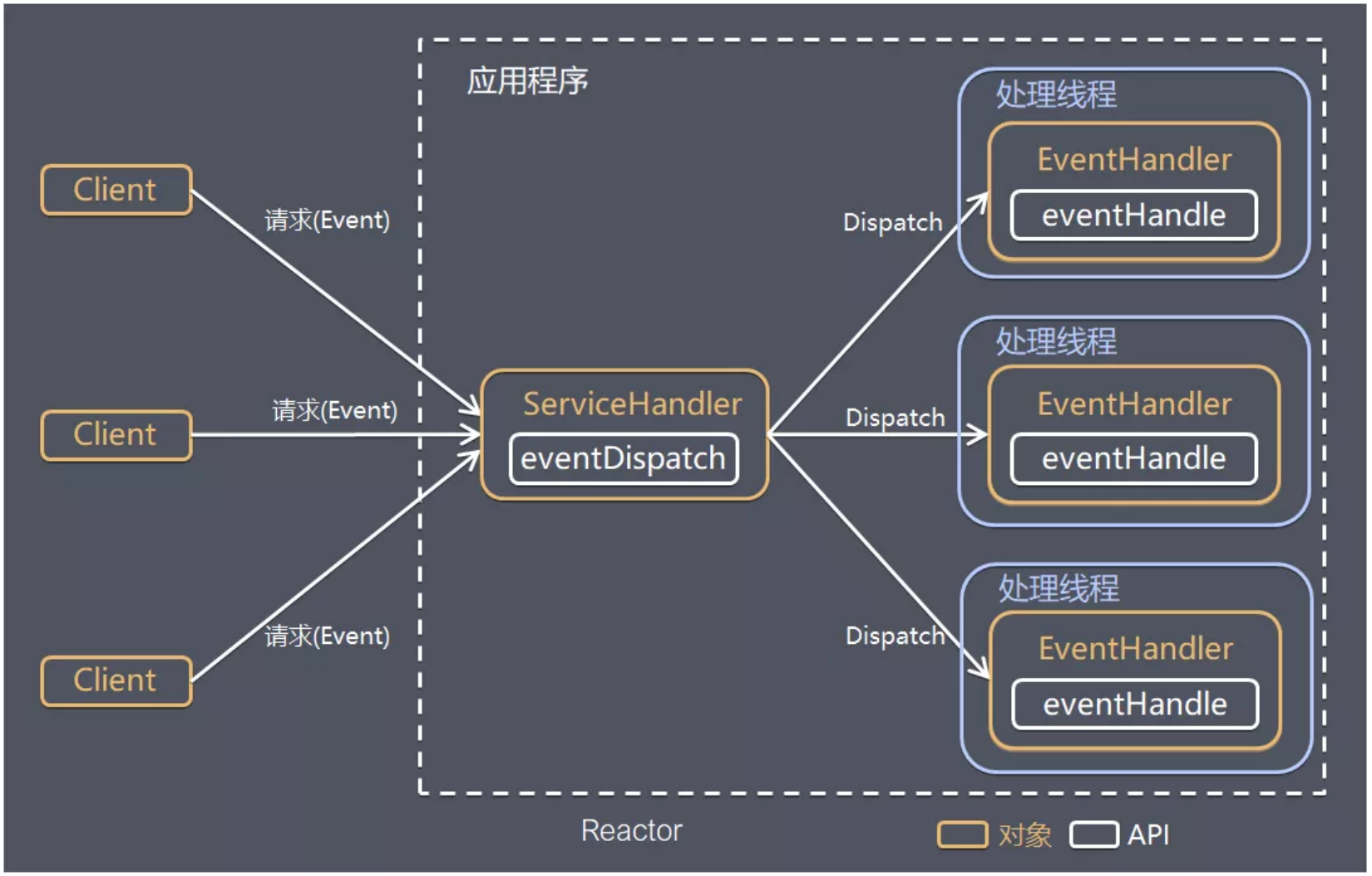
通过以上的介绍和描述，我们可以获知几个关键概念：`Event-Loop`、`Async I/O`, `Handle`、`Request`。

- `Event-Loop` 是一个程序结构，用于在程序中 等待 和 派发 消息和事件。它实际是一个运行时概念，表示一个运行时逻辑。`Event-Loop` 是实现 `Event-Driven` 编程的基本结构。
- `Async I/O` 相比 `Sync I/O`，异步I/O一个典型的特征是不会阻塞，因为IO操作通常比较慢，通常的同步IO操作，会导致程序逻辑阻塞在某一步IO操作，进而导致程序响应很慢。

`Event-Loop` 和 `Async I/O` 是实现高性能IO的常见手段，这些都始于 `C10k` 问题，就是单服务器同时服务 10000 个客户端，当然现在远不止这些了。早期的服务器是基于进程/线程模型，每新来一个连接，就分配一个进程（线程）去处理这个连接，这是一个非常大的开销，容易达到系统软硬件瓶颈，另外，进程/线程切换上下文的成本也非常高。

一般而言，大多数网络应用服务器端软件都是I/O密集型系统，服务器系统大部分的时间都花费在等待数据的输入输出上了，而不是计算，如果CPU把时间花在等待I/O操作上，就白白浪费了CPU的处理能力了，更重要的是，此时可能还有大量的客户端请求需要处理，而CPU却在等待I/O无法脱身。最终，以此方式工作的服务器吞吐量极低，需要更多的服务支撑业务，导致成本升高。

为了充分利用CPU的计算能力，就需要避免让CPU等待I/O操作完成能够抽出身来做其他工作，例如，接收更多请求，等I/O操作完成之后再来进一步处理。这就有了 非阻塞I/O。异步I/O给编程带来了一定的麻烦，因为同步思维对于人来说更自然、更容易，也不易于调试，但是实际上限时世界原本偏向异步的。如何在I/O操作完成后能够让CPU回来继续完成工作也需要更复杂的流程逻辑控制，这些都带来了一定的设计难度。不过幸好，聪明的开发者设计了`Event-Driven`编程模型解决了此问题。基于此模型，也衍生出高性能IO常见实现模式：`Reactor`模式，该模式有很多变体。Redis、Nginx、Netty、java.NIO都采用类似的模式来解决高并发的问题，libuv 自然也不例外，实现了事件驱动的异步IO。`Reactor`模式采用同步I/O，`Proactor`是一个采用异步I/O的模式。



该部分内容参考：

- https://en.wikipedia.org/wiki/Event-driven
- https://en.wikipedia.org/wiki/Event-driven_programming
- https://medium.com/@tigranbs/concurrency-vs-event-loop-vs-event-loop-concurrency-eb542ad4067b

- http://www.kegel.com/c10k.html
- https://en.wikipedia.org/wiki/Reactor_pattern
- http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf
- http://www.laputan.org/pub/sag/reactor.pdf
- http://www.dre.vanderbilt.edu/~schmidt/PDF/reactor-siemens.pdf
- https://java-design-patterns.com/patterns/reactor/
- 高性能Server - Reactor模型
- 理解高性能网络模型

libevent vs libev vs libuv

- libevent：名气最大，应用最广泛，历史悠久的跨平台事件库；
- libev：较libevent而言，设计更简练，性能更好，但对Windows支持不够好；
- libuv：开发node的过程中需要一个跨平台的事件库，他们首选了libev，但又要支持Windows，故重新封装了一套，linux下用libev实现，Windows下用IOCP实现；

libevent、libev、libuv 都是c语言实现的事件驱动（`Event-Driven`）的异步I/O（`Async I/O`）库。

异步I/O（`Async I/O`）库本质上是提供异步I/O事件通知（Asynchronous Event Notification，AEN）的。异步事件通知机制就是根据发生的事件，调用相应的回调函数进行处理。

- 事件（**Event**）：事件是通知机制的核心，比如I/O事件、定时器事件、信号事件事件。有时候也称事件为事件处理器（EventHandler），这个名称更形象，因为Handler本身表示了包含处理所需数据（或数据的地址）和处理的方法（回调函数），更像是面向对象思想中的称谓。
- 事件循环（**EventLoop**）：是事件驱动（`Event-Driven`）的核心，等待并分发事件。事件循环用于管理事件。

对于应用程序来说，这些只是异步事件库提供的API，封装了异步事件库跟操作系统的交互，异步事件库会选择一种操作系统提供的机制来实现某一种事件，比如利用Unix/Linux平台的epoll机制实现网络IO事件，在同时存在多种机制可以利用时，异步事件库会采用最优机制。

该部分内容参考：

- 网络库libevent、libev、libuv对比
- Libevent Libev Libuv ...
- whats-the-difference-between-libev-and-libevent

源码

版本

```
$ git checkout v1.28.0
Previous HEAD position was a4fc9a66 2019.03.17, Version 1.27.0 (Stable)
HEAD is now at 7bf8fabf 2019.04.16, Version 1.28.0 (Stable)
```

目录结构

以下为源码的文件结构，删掉了无关的部分

```
$ tree
.
├── README.md
├── docs/
├── include
│   ├── uv
│   │   ├── aix.h
```

```
│   │   ├── android-ifaddrs.h
│   │   ├── bsd.h
│   │   ├── darwin.h
│   │   ├── errno.h
│   │   ├── linux.h
│   │   ├── os390.h
│   │   ├── posix.h
│   │   ├── stdint-msvc2008.h
│   │   ├── sunos.h
│   │   ├── threadpool.h
│   │   ├── tree.h
│   │   ├── unix.h
│   │   ├── version.h
│   │   └── win.h
│   └── uv.h
├── src
│   ├── fs-poll.c
│   ├── heap-inl.h
│   ├── idna.c
│   ├── idna.h
│   ├── inet.c
│   ├── queue.h
│   ├── strscpy.c
│   ├── strscpy.h
│   ├── threadpool.c
│   ├── timer.c
│   ├── uv-common.c
│   ├── uv-common.h
│   ├── uv-data-getter-setters.c
│   ├── version.c
│   ├── unix/
│   └── win/
├── test/
├── samples/
```

```
├── uv.gyp
└── vcbuild.bat
```

源码主要存在于以下几个目录：

- `include`：存放 `.h` 文件，这些文件主要用于对外暴露 `c` API
  - `include/uv.h` 文件存放平台无关的头文件，该文件需要被 `include` 依赖项目的源码当中。
  - `include/uv/*.h` 路径下的文件则是针对不同平台进行的不同相关类型的声明定义等。

  `include/uv.h` 会根据不同的环境 `include` `uv/win.h` 或 `uv/unix.h`，`uv/unix.h` 再 `include` `*nix` 系的其他系统相关头文件。如同通常的c库一样，`uv.h` 不仅作为入口文件，同时还具备文档的作用，阅读源码自然适合从此文件开始。

  `include/tree.h` 是个例外，该文件内通过 宏实现了 `伸展树` 和 `红黑树`，而 同样采用`宏`实现的 `队列` 存放在 `src/queue.h` 文件中

- `src`：存放 `.c` 文件，和一些 不对外暴露的 `.h` 文件
  - `uv-common.h/uv-common.c` 包含部分公共的内部数据结构、函数的声明和实现，会被 `src` 内部大部分其他文件 包含
  - `timer.c` 对应于 定时器 的实现
  - `threadpool.c` 实现了线程池，对应的线程管理实现存在于 `src/[unix|win]/thread.c` 文件中
  - `queue.h` 基于宏实现的简单的队列
  - `heap-inl.h` 最小二叉堆实现，未采用宏实现
  - `fs-poll.c` 文件系统轮询相关实现
  - `idna.h/idna.c` IDNA Punycode 相关实现代码
  - `unix/` *nix 平台相关实现

    - `win/` win 平台相关实现

- `test`：存放一些 单元测试 代码，这里面的很多代码可以作为参考示例
- `samples`：存放 示例代码，其中 `samples/socks5-proxy` 是一个基于 `libuv` 实现的 `sock5` 代理

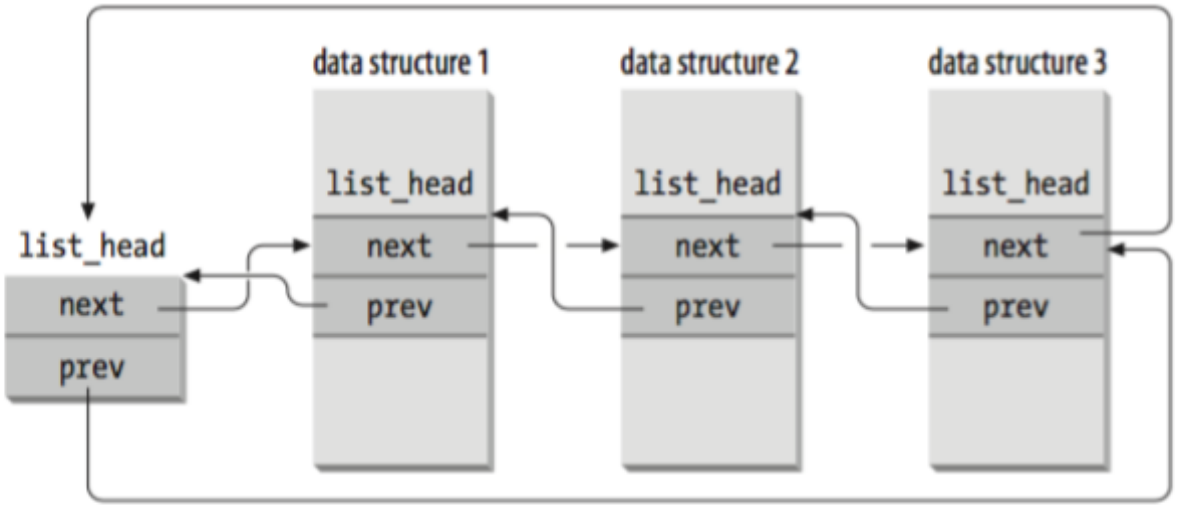
命名风格

libuv 所有函数、结构体都采用了统一的前缀 `uv_`，名称格式为：`uv_ + name`， `name` 可以以下划线开头，表示内部成员，例如：

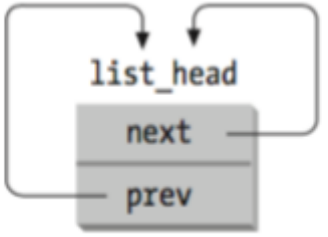
- 公开名称：`uv_loop_t = uv_ + loop_t` `uv_loop_start = uv_ + loop_start`
- 内部名称：`uv__io_t = uv_ + _io_t` `uv__io_poll = uv_ + _io_poll`

数据结构

libuv 中的数据结构（队列，堆）采用被称为 `侵入式` 的实现方式实现，下图为 Linux 内核 `/include/linux/list.h` 的实现示意图：



(a) a doubly linked listed with three elements



(b) an empty doubly linked list

该队列与通常实现最大的不同是兄弟节点指针 `prev` 和 `next` 存储的并发数据结构首地址，而是队列节点（数据结构某个成员）的地址，图中是 `list_head`，如果需要拿到完整的数据结构，需要获得数据结构的首地址，已知条件是已知 `list_head` 的地址，`c` 语言程序中结构体的内存布局是对齐的，所以可以计算出 `list_head` 相对数据结构首地址的偏移量，这样就可以算出数据结构首地址了，`container_of` 就是完成这一换算的。这种方式实现的数据结构，图中 `data structure 1`、`data structure 2`、`data structure 3` 可以是不同的类型，所以这种方式实现的数据结构可以很通用。

`container_of` 定义如下：

```
#define container_of(ptr, type, member) \
  ((type *) ((char *) (ptr) - offsetof(type, member)))
```

在 linux 内核中也定义了这个宏，但是略微不同，宏定义如下：

https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/kernel.h?id=refs/tags/v4.10.13#n842

```
/**
 * container_of - cast a member of a structure out to the containing structure
 * @ptr:        the pointer to the member.
 * @type:       the type of the container struct this is embedded in.
 * @member:     the name of the member within the struct.
 *
 */
#define container_of(ptr, type, member) ({                      \
        const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
        (type *)( (char *)__mptr - offsetof(type,member) );})
```

关于 `container_of` 的实现原理可参考：

- typeof, offsetof 和container_of

接下来，开始详细分析 libuv 源码的各个部分内容，请见下文。

源文件地址：https://github.com/liuyanjie/knowledge/tree/master/node.js/libuv/1-libuv-overview.md