Tree: 62e84b27e7 liuyanjie.github.io / source / _posts / 1-libuv源码分析（二）事件循环（Eventloop）.md

Find file Copy path

liuyanjie add articles 2c06277 on Apr 24

1 contributor

432 lines (315 sloc) 15.9 KB

Raw Blame History

title	date	updated	tags	categories

title	date	updated	tags	categories
libuv源码分析（二）事件循环（Eventloop）	2019-04-23 15:00:02 UTC	2019-04-24 01:31:28 UTC	libuv, node.js, eventloop	源码分析

事件循环是 libuv 功能的核心部分。它的主要职责是对I/O进行轮询然后基于不同的事件源调度它们的回调。

事件循环主体数据结构在 libuv 中用 `struct uv_loop_s` 或类型别名 `uv_loop_t` 表示，文中统一使用 `loop` 表示其实例，它代表了事件循环，实际上它是事件循环所有资源的统一入口，所有在事件循环上运行的各类 `Handle/Request` 实例都被注册到 `uv_loop_s` 内部的各类结构中如队列、堆、伸展树等，同一实例往往被关联到多个不同的结构中，如大多数 `Handle` 都会同时存在两个队列中。

`uv_loop_t` 是一种特殊的 `Handle`，它管理了同一事件循环上的所有资源。

`uv_loop_t` 实例通常需要经历 `Init`、`Run`、`Stop`、`Close` 这几个生命周期，下面将分别分析几个阶段的实现。

Example

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

int main() {
    uv_loop_t* loop = uv_default_loop();
    uv_run(loop, UV_RUN_DEFAULT);
    uv_loop_close(loop);
    printf("quit.\n");
    return 0;
}
```

```
gcc -luv helloworld.c -o helloworld
```

```
$ ./helloworld
quit.
```

以上是一个最基本的 libuv 程序代码，通过 `uv_run` 函数启动了 libuv 事件循环，所以 `uv_run` 做为事件循环的入口一定是阅读源码的重点，可以 `uv_run` 为起点，先看看 `uv_run` 都做了什么。

程序启动之后，打印了 `quit.\n` 立刻退出了，更具体一点说就是，所以函数调用尤其是 `uv_run` 函数立即返回了，程序自然就退出了，因为我们实际什么也有做，连上文提到的异步操作以及注册的回调函数都没有。

Init：uv_loop_init

在常见的使用场景中，通常都是直接调用 `uv_default_loop` 获取已经初始了的全局 `uv_loop_t` 实例，所以在分析 `uv_run` 之前，先看一下 `uv_loop_t` 初始化。

先来看一下 `uv_default_loop`：https://github.com/libuv/libuv/blob/v1.28.0/src/unix/loop.c#L30

```
static uv_loop_t default_loop_struct;
static uv_loop_t* default_loop_ptr;


uv_loop_t* uv_default_loop(void) {
  if (default_loop_ptr != NULL)
    return default_loop_ptr;

  if (uv_loop_init(&default_loop_struct))
    return NULL;
```

```
  default_loop_ptr = &default_loop_struct;
  return default_loop_ptr;
}
```

在 libuv 中存在一个全局的、静态的 `uv_loop_t` 实例 `default_loop_struct`，首次获取的时候经过 `uv_loop_init` 进行了初始化。

`uv_default_loop` 调用 `uv_loop_init` 对 `default_loop_struct` 进行初始化并将地址赋给了 `default_loop_ptr`。

`uv_loop_init` 实现如下（含注释）：

https://github.com/libuv/libuv/blob/v1.28.0/src/unix/loop.c#L29

```
int uv_loop_init(uv_loop_t* loop) {
  void* saved_data;
  int err;

  // 数据清零
  saved_data = loop->data;
  memset(loop, 0, sizeof(*loop));
  loop->data = saved_data;

  // 定时器 uv_timer_t 相关：初始化定时器堆
  heap_init((struct heap*) &loop->timer_heap);
  // 初始化用于接收线程池中已完成任务的队列
  QUEUE_INIT(&loop->wq);
  // 初始化 uv_idle_t 队列
  QUEUE_INIT(&loop->idle_handles);
  // 初始化 uv_async_t 队列
  QUEUE_INIT(&loop->async_handles);
  // 初始化 uv_check_t 队列
```

```
    QUEUE_INIT(&loop->check_handles);
    // 初始化 uv_prepare_t 队列
    QUEUE_INIT(&loop->prepare_handles);
    // 初始化 uv_handle_t 队列，所以初始化后的 handle 都会放到此队列中
    QUEUE_INIT(&loop->handle_queue);

    // 初始化 活跃的 handle 和 request 数量
    loop->active_handles = 0;
    loop->active_reqs.count = 0;

    // 开始初始化I/O观察者相关字段
    // 文件描述符数量
    loop->nfds = 0;
    // I/O观察者数组首地址指针
    loop->watchers = NULL;
    // I/O观察者数组数量，但是 `loop->watchers` 实际长度为：nwatchers + 2
    loop->nwatchers = 0;
    // 初始化 挂起的I/O观察者队列，挂起的I/O观察者会被插入此队列延迟处理
    QUEUE_INIT(&loop->pending_queue);
    // 初始化 I/O观察者队列，所有初始化后的I/O观察者都会被插入此队列
    QUEUE_INIT(&loop->watcher_queue);

    // 关闭的 handle 队列，单向链表
    loop->closing_handles = NULL;
    // 初始化计时器 loop->time
    uv__update_time(loop);

    // uv_async_t
    // 初始化 async_io_watcher，它是一个I/O观察者，用于 uv_async_t 唤醒事件循环
    loop->async_io_watcher.fd = -1;
    // 用于写数据给 async_io_watcher
    loop->async_wfd = -1;

    // uv_signal_t
    loop->signal_pipefd[0] = -1;
```

```
  loop->signal_pipefd[1] = -1;
  // epoll_create()
  loop->backend_fd = -1;
  // EMFILE 错误相关
  loop->emfile_fd = -1;

  // 定时器计数器
  loop->timer_counter = 0;

  // 事件循环关闭标识
  loop->stop_flag = 0;

  // 平台特定初始化：UV_LOOP_PRIVATE_FIELDS
  err = uv__platform_loop_init(loop);
  if (err)
    return err;

  // uv_signal_t
  // 初始化进程信号
  uv__signal_global_once_init();

  // uv_proccess_t
  // 初始化子进程信号观察者
  err = uv_signal_init(loop, &loop->child_watcher);
  if (err)
    goto fail_signal_init;
  // 解引用loop->child_watcher
  uv__handle_unref(&loop->child_watcher);
  loop->child_watcher.flags |= UV_HANDLE_INTERNAL;
  // 初始化子进程 handle 队列
  QUEUE_INIT(&loop->process_handles);

  // 初始化线程读写锁
  err = uv_rwlock_init(&loop->cloexec_lock);
  if (err)
```

```
    goto fail_rwlock_init;

  // 初始化线程互斥量锁
  err = uv_mutex_init(&loop->wq_mutex);
  if (err)
    goto fail_mutex_init;

  // uv_work_t
  // 初始化loop->wq_async，用于结束任务完成信号，并注册处理函数
  err = uv_async_init(loop, &loop->wq_async, uv__work_done);
  if (err)
    goto fail_async_init;
  // 解引用
  uv__handle_unref(&loop->wq_async);
  loop->wq_async.flags |= UV_HANDLE_INTERNAL;

  return 0;

fail_async_init:
  uv_mutex_destroy(&loop->wq_mutex);

fail_mutex_init:
  uv_rwlock_destroy(&loop->cloexec_lock);

fail_rwlock_init:
  uv__signal_loop_cleanup(loop);

fail_signal_init:
  uv__platform_loop_delete(loop);

  return err;
}
```

`uv_loop_init` 的初始化代码是比较长的，它初始化了 libuv 运行时所有依赖的内容，这包括事件循环自身运行所需的内容，以及各类型 Handle 运行所需的共有内容和特定内容，这些都在 `uv_loop_t` 实例初始化的时候一并进行了初始化，初始化细节和很多有其他功能相关，当分析其他功能时，还会提及涉及到的该函数的部分代码块。

Run：uv_run

下面我们就来看一下 `uv_run` 函数都干了什么：

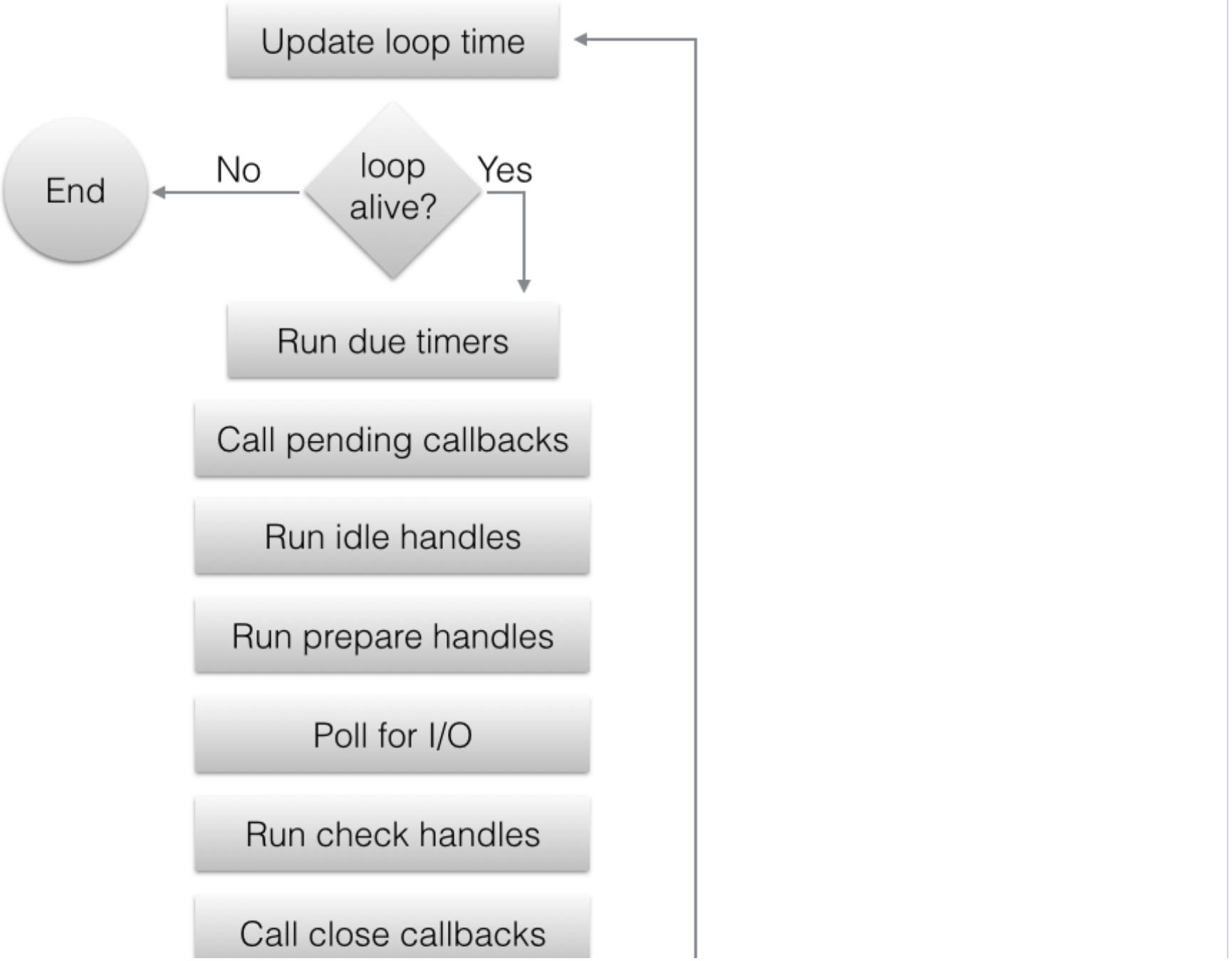
https://github.com/libuv/libuv/blob/v1.28.0/src/unix/core.c#L343

```
int uv_run(uv_loop_t* loop, uv_run_mode mode) {
  int timeout;
  int r;
  int ran_pending;

  r = uv__loop_alive(loop);
  if (!r)
    uv__update_time(loop);

  while (r != 0 && loop->stop_flag == 0) {
    uv__update_time(loop);
    uv__run_timers(loop);
    ran_pending = uv__run_pending(loop);
    uv__run_idle(loop);
    uv__run_prepare(loop);

    timeout = 0;
    if ((mode == UV_RUN_ONCE && !ran_pending) || mode == UV_RUN_DEFAULT)
      timeout = uv_backend_timeout(loop);

    uv__io_poll(loop, timeout);
    uv__run_check(loop);
```

```
    uv__run_closing_handles(loop);

    if (mode == UV_RUN_ONCE) {
      /* UV_RUN_ONCE implies forward progress: at least one callback must have
       * been invoked when it returns. uv__io_poll() can return without doing
       * I/O (meaning: no callbacks) when its timeout expires - which means we
       * have pending timers that satisfy the forward progress constraint.
       *
       * UV_RUN_NOWAIT makes no guarantees about progress so it's omitted from
       * the check.
       */
      uv__update_time(loop);
      uv__run_timers(loop);
    }

    r = uv__loop_alive(loop);
    if (mode == UV_RUN_ONCE || mode == UV_RUN_NOWAIT)
      break;
  }

  /* The if statement lets gcc compile it to a conditional store. Avoids
   * dirtying a cache line.
   */
  if (loop->stop_flag != 0)
    loop->stop_flag = 0;

  return r;
}
```

可以看到 `uv_run` 内部就是一个 `while` 循环，在 `UV_RUN_ONCE` 和 `UV_RUN_NOWAIT` 两种模式下，循环在执行一次后就会 `break`，一次性的，实际上没有循环。另外 在 `uv__loop_alive(loop) == 0` 或者 `loop->stop_flag != 0` 时 无法进入循环，同样循环结束，`uv_run` 函数返回。

该函数就如官网给出的流程图一样简单



再看看几个关键的函数调用：

1. `uv__update_time(loop)`：对应图中 `Update loop time`
2. `uv__run_timers(loop)`：对应图中 `Run due timers`，用于 `uv_timer_t`，见 Timer
3. `uv__run_pending(loop)`：对应图中 `Call pending callbacks`，用于 `uv__io_t`，见 I/O-Watcher
4. `uv__run_idle(loop)`：对应图中 `Run idle handles`，用于 `uv_idle_t`
5. `uv__run_prepare(loop)`：对应图中 `Run prepare handles`，用于 `uv_prepare_t`
6. `uv__io_poll(loop, timeout)`：对应图中 `Poll for I/O`，用于 `uv__io_t`，见 I/O-Watcher
7. `uv__run_check(loop)`：对应图中 `Run check handles`，用于 `uv_check_t`
8. `uv__run_closing_handles(loop)`：对应图中 `Call close callbacks`，用于 `uv_handle_t`，见 Handle and Requst

以上执行逻辑正好和文档中的各个执行阶段相对应，文档中描述的各个执行阶段分别对应了不同的函数调用。整个循环迭代的不同阶段，对应于不同类型/状态的 handle 处理。除了用于 `uv_timer_t`、`uv_idle_t`、`uv_prepare_t`、`uv_check_t` 这四种类型的 handle 处理的几个阶段之外，没看到其他 handle 相关内容，倒是有个 `uv__io_t` 的处理，这是前文所提到的 libuv 内部关于I/O观察者的一个基本抽象，所有其他的 `handle` 都可以当做是一个I/O观察者，类似于双重继承。

如果这个函数处于一直不断的循环状态，所在进程岂不是会一直占用CPU？实际上不会这样的，因为线程会在 `uv__io_poll(loop, timeout)` 这个函数内部因为阻塞而挂起，挂起的时间主要由下一次到来的定时器决定。在线程挂起这段时间内，不会占用CPU。

`uv_run` 启动事件循环，才使所有活动状态的 `handle` 开始工作，否则所有 `handle` 都是静止的，这一步就是 libuv 启动的按钮。

事件循环自身存在存活状态，通过 `uv__loop_alive` 判断，实现如下：

https://github.com/libuv/libuv/blob/v1.28.0/src/unix/core.c#L331

```
static int uv__loop_alive(const uv_loop_t* loop) {
  return uv__has_active_handles(loop) ||
         uv__has_active_reqs(loop) ||
         loop->closing_handles != NULL;
}
```

`uv__loop_alive` 判断 `loop` 是否是存活状态，满足以下三种条件之一即是存活状态：

- 存在活跃的 handle
- 存在活跃的 request
- 正在关闭的 handle 列表不为空

所以，若想成功事件事件循环一直不断的运行而不退出，必须在 `uv_run` 之前想事件循环里放入处于活跃状态的 `handle` 或 `request`。

在 `uv_loop_t` 结构中，存在记录处于活动状态的 `handle` 和 `request` 的计数器，所以通过简单的判断数量即可，实现如下：

```
#define uv__has_active_handles(loop)                                          \
  ((loop)->active_handles > 0)
```

```
#define uv__has_active_reqs(loop)                                             \
  ((loop)->active_reqs.count > 0)
```

另外，除了存活状态之外，`loop` 还存在一个 `stop_flag` 字段 标识 `loop` 是否处于 `关闭` 状态。

所以，当 `loop` 中没有活动的 `handle` 和 `request` 时 或者 关闭标识开启时，事件循环跳出。

libuv 在运行时 有三种模式：对应模式的用途看文档上对应的描述即可，uv_run。

在 Run 的过程中，多次调用 `uv__update_time` 来更新时间

https://github.com/libuv/libuv/blob/v1.x/src/unix/internal.h#L288

```
UV_UNUSED(static void uv__update_time(uv_loop_t* loop)) {
  /* Use a fast time source if available.  We only need millisecond precision.
   */
  loop->time = uv__hrtime(UV_CLOCK_FAST) / 1000000;
}
```

```
uint64_t uv__hrtime(uv_clocktype_t type) {
  return gethrtime();
}
```

这个函数通过调用 `gethrtime` 获取系统当前时间，精度非常高，单位是纳秒（ns），1纳秒等于十亿分之一秒。除 `1000000` 后的时间单位为 毫秒（ms）。

时间对 libuv 来说非常重要，很多机制依赖于这个时间，比如定时器，后续的分析中，我们将会看到相关的利用。

在事件循环中，还有一个 `timeout`，这个值用于控制 `uv__io_poll(loop, timeout)` 的挂起时长，这个变量的值是通过 `uv_backend_timeout` 来获取的，源码如下：

https://github.com/libuv/libuv/blob/v1.28.0/src/unix/core.c#L311

```c
int uv_backend_timeout(const uv_loop_t* loop) {
  if (loop->stop_flag != 0)
    return 0;

  if (!uv__has_active_handles(loop) && !uv__has_active_reqs(loop))
    return 0;

  if (!QUEUE_EMPTY(&loop->idle_handles))
    return 0;

  if (!QUEUE_EMPTY(&loop->pending_queue))
    return 0;

  if (loop->closing_handles)
    return 0;

  return uv__next_timeout(loop);
}
```

`uv_backend_timeout` 在多个情况下都返回 `0`，这些情况表明不需要等待超时，如果前面的条件都不满足，会通过 `uv__next_timeout` 计算 `timeout`，源码如下：

https://github.com/libuv/libuv/blob/v1.28.0/src/timer.c#L137

```c
int uv__next_timeout(const uv_loop_t* loop) {
  const struct heap_node* heap_node;
  const uv_timer_t* handle;
  uint64_t diff;

  heap_node = heap_min(timer_heap(loop));
  if (heap_node == NULL)
    return -1; /* block indefinitely */
```

```
  handle = container_of(heap_node, uv_timer_t, heap_node);
  if (handle->timeout <= loop->time)
    return 0;

  diff = handle->timeout - loop->time;
  if (diff > INT_MAX)
    diff = INT_MAX;

  return (int) diff;
}
```

`uv__next_timeout` 有两种 情况：

- 堆为空，返回 `-1`
- 堆非空，返回 `堆顶定时器` 和 `当前时间的差值`，但是差值不能越界。

综合在一起，`uv_backend_timeout` 有可能返回 `-1` `0` `正整数`。

可以看到 `timeout` 作为参数传递给了 `uv__io_poll`，而 `timeout` 正好作为 `epoll_pwait` 的超时时间，所以，这个 `timeout` 的作用主要是使 `epoll_pwait` 能够有一个合理的超时时间:

- 当 `timeout` 为 `-1` 的时候这个函数会无限期的阻塞下去；
- 当 `timeout` 为 `0` 的时候，就算没有任何事件，也会立刻返回，当没有任何需要等待的资源时，`timeout` 刚好为 `0`；
- 当 `timeout` 等于 `正整数` 的时候，将会阻塞 `timeout` 毫秒，或有I/O事件产生。

`epoll_pwait` 要在定时器时间到来时返回进入以进入下一次事件循环处理定时器，如果不能返回，将会导致定时任务不能按时得到处理，即使是按时返回，也不一定能够那是处理，因为 `uv__io_poll` 之后还有其他逻辑代码要执行，甚至是有可能是耗时计算，所以，Node.js 中定时器是不精确的，浏览器中类似。

在 `UV_RUN_ONCE` 模式下，因为循环会直接跳出，不会再次进入循环处理定时器，所以需要在这种模式下，需要处理额外处理定时器。

至此，事件循环的大逻辑已经分析完成了，后续，将会在各类型的 `handle` 的处理逻辑中展开对事件循环各阶段的内容分析。

Stop

Stop 将使事件循环在下一次循环因不满条件而无法进入，源码如下：

https://github.com/libuv/libuv/blob/v1.28.0/src/uv-common.c#L517

```
void uv_stop(uv_loop_t* loop) {
  loop->stop_flag = 1;
}
```

调用 `uv_stop` 后，事件循环同样无法进入，程序退出。

源文件地址：https://github.com/liuyanjie/knowledge/tree/master/node.js/libuv/2-libuv-event-loop.md