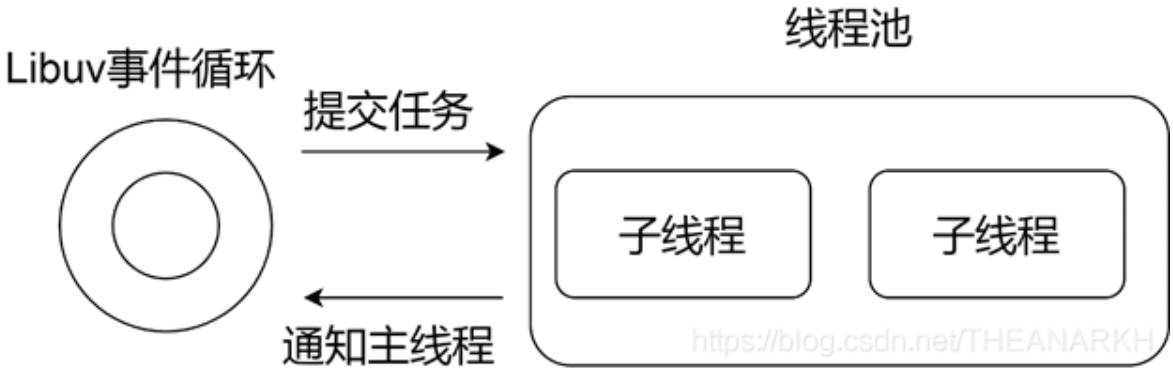
第四章 线程池

Libuv是单线程事件驱动的异步IO库，对于阻塞式或耗时的操作，如果在Libuv的主循环里执行的话，就会阻塞后面的任务执行，所以Libuv里维护了一个线程池，它负责处理Libuv中耗时或者导致阻塞的操作，比如文件IO、DNS、自定义的耗时任务。线程池在Libuv架构中的位置如图4-1所示。



Libuv主线程通过线程池提供的接口把任务提交给线程池，然后立刻返回到事件循环中继续执行，线程池维护了一个任务队列，多个子线程会互斥地从中摘下任务节点执行，当子线程执行任务完毕后会通知主线程，主线程在事件循环的Poll IO阶段就会执行对应的回调。下面我们看一下线程池在Libuv中的实现。

4.1主线程和子线程间通信

Libuv子线程和主线程的通信是使用uv_async_t结构体实现的。Libuv使用loop->async_handles队列记录所有的uv_async_t结构体，使用loop->async_io_watcher作为所有uv_async_t结构体的IO观察者，即loop->async_handles队列上所有的handle都是共享async_io_watcher这个IO观察者的。第一次插入一个uv_async_t结构体到async_handle队列时，会初始化IO观察者，如果再次注册一个async_handle，只会在loop->async_handle队列和

handle队列插入一个节点，而不会新增一个IO观察者。当uv_async_t结构体对应的任务完成时，子线程会设置IO观察者为可读。Libuv在事件循环的Poll IO阶段就会处理IO观察者。下面我们看一下uv_async_t在Libuv中的使用。

4.1.1 初始化

使用uv_async_t之前首先需要执行uv_async_init进行初始化。

```
    int uv_async_init(uv_loop_t* loop,
                      uv_async_t* handle,
                      uv_async_cb async_cb) {
      int err;
      // 给Libuv注册一个观察者io
      err = uv__async_start(loop);
      if (err)
        return err;
      // 设置相关字段，给Libuv插入一个handle
      uv__handle_init(loop, (uv_handle_t*)handle,
UV_ASYNC);
        // 设置回调
      handle->async_cb = async_cb;
        // 初始化标记字段，0表示没有任务完成
      handle->pending = 0;
      // 把uv_async_t插入async_handle队列
      QUEUE_INSERT_TAIL(&loop->async_handles, &handle-
>queue);
      uv__handle_start(handle);
      return 0;
    }
```

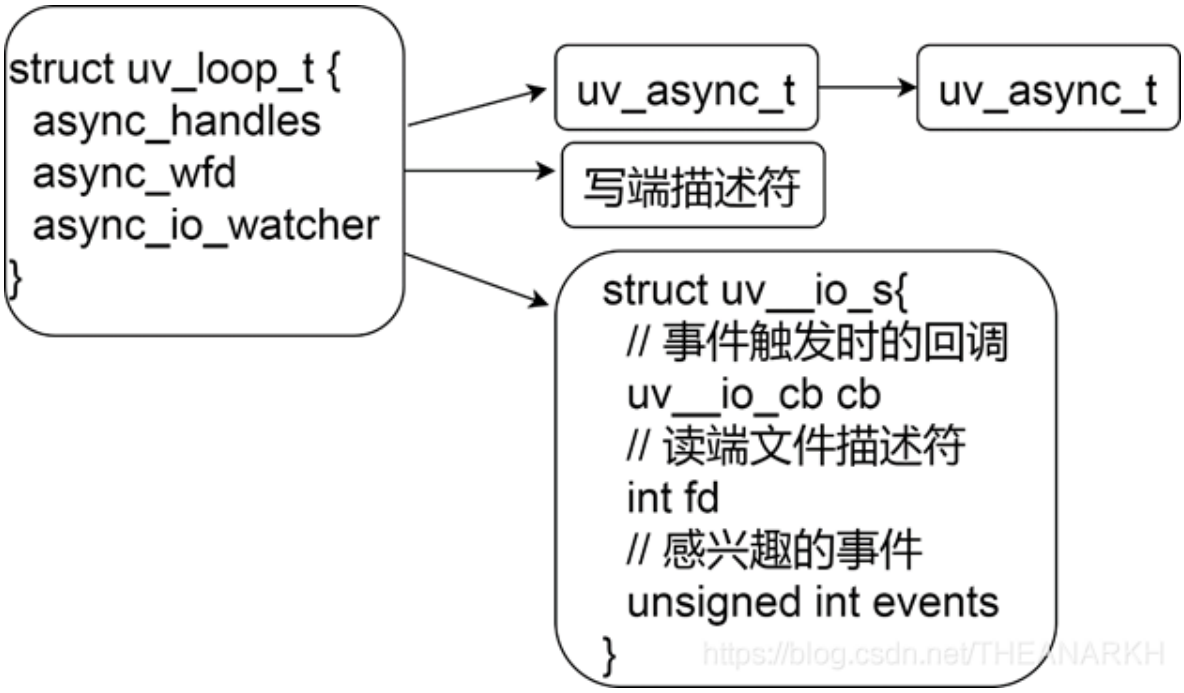
uv_async_init函数主要初始化结构体uv_async_t的一些字段，然后执行QUEUE_INSERT_TAIL给Libuv的async_handles队列追加一个节点。我们看到还有一个uv__async_start函数。我们看一下uv__async_start的实现。

```
    static int uv__async_start(uv_loop_t* loop) {
      int pipefd[2];
      int err;
```

```
4         // uv__async_start只执行一次，有fd则不需要执行了
5         if (loop->async_io_watcher.fd != -1)
6           return 0;
7         // 获取一个用于进程间通信的fd（Linux的eventfd机制）
8         err = uv__async_eventfd();
9         /*
10            成功则保存fd，失败说明不支持eventfd，
11             则使用管道通信作为进程间通信
12          */
13        if (err >= 0) {
14          pipefd[0] = err;
15          pipefd[1] = -1;
16        }
17        else if (err == UV_ENOSYS) {
18          // 不支持eventfd则使用匿名管道
19          err = uv__make_pipe(pipefd, UV__F_NONBLOCK);
20      #if defined(__Linux__)
21          if (err == 0) {
22            char buf[32];
23            int fd;
24            snprintf(buf, sizeof(buf), "/proc/self/fd/%d",
25  pipefd[0]);          // 通过一个fd就可以实现对管道的读写，高
26  级用法
27            fd = uv__open_cloexec(buf, O_RDWR);
28            if (fd >= 0) {
29              // 关掉旧的
30              uv__close(pipefd[0]);
31              uv__close(pipefd[1]);
32              // 赋值新的
33              pipefd[0] = fd;
34              pipefd[1] = fd;
35            }
36          }
37      #endif
38        }
39        // err大于等于0说明拿到了通信的读写两端
40        if (err < 0)
41          return err;
42        /*
43            初始化IO观察者async_io_watcher，
44            把读端文件描述符保存到IO观察者
```

```
45            */
46          uv__io_init(&loop->async_io_watcher, uv__async_io,
47      pipefd[0]);
48          // 注册IO观察者到loop里，并注册感兴趣的事件POLLIN，等待
49      可读
50          uv__io_start(loop, &loop->async_io_watcher,
        POLLIN);
              // 保存写端文件描述符
            loop->async_wfd = pipefd[1];
            return 0;
        }
```

uv__async_start只会执行一次，时机在第一次执行uv_async_init的时候。uv__async_start主要的逻辑如下 1 获取通信描述符（通过eventfd生成一个通信的fd（充当读写两端）或者管道生成线程间通信的两个fd表示读端和写端）。2 封装感兴趣的事件和回调到IO观察者然后追加到watcher_queue队列，在Poll IO阶段，Libuv会注册到epoll里面，如果有任务完成，也会在Poll IO阶段执行回调。 3 保存写端描述符。任务完成时通过写端fd通知主线程。 我们看到uv__async_start函数里有很多获取通信文件描述符的逻辑，总的来说，是为了完成两端通信的功能。初始化async结构体后，Libuv结构如图4-2所示。



4.1.2 通知主线程

初始化async结构体后，如果async结构体对应的任务完成后，就会通知主线程，子线程通过设置这个handle的pending为1标记任务完成，然后再往管道写端写入标记，通知主线程有任务完成了。

```
    int uv_async_send(uv_async_t* handle) {
      /* Do a cheap read first. */
      if (ACCESS_ONCE(int, handle->pending) != 0)
        return 0;
      /*
          如pending是0，则设置为1，返回0，如果是1则返回1，
          所以如果多次调用该函数是会被合并的
        */
      if (cmpxchgi(&handle->pending, 0, 1) == 0)
        uv__async_send(handle->loop);
      return 0;
    }

    static void uv__async_send(uv_loop_t* loop) {
      const void* buf;
      ssize_t len;
      int fd;
      int r;

      buf = "";
      len = 1;
      fd = loop->async_wfd;

    #if defined(__Linux__)
      // 说明用的是eventfd而不是管道,eventfd时读写两端对应同一个fd
      if (fd == -1) {
        static const uint64_t val = 1;
        buf = &val;
        len = sizeof(val);
        // 见uv__async_start
        fd = loop->async_io_watcher.fd;  /* eventfd */
      }
```

```
#endif
  // 通知读端
  do
    r = write(fd, buf, len);
  while (r == -1 && errno == EINTR);

  if (r == len)
    return;

  if (r == -1)
    if (errno == EAGAIN || errno == EWOULDBLOCK)
      return;

  abort();
}
```

uv_async_send首先拿到写端对应的fd，然后调用write函数，此时，往管道的写端写入数据，标记有任务完成。有写则必然有读。读的逻辑是在uv__io_poll中实现的。uv__io_poll函数即Libuv中Poll IO阶段执行的函数。在uv__io_poll中会发现管道可读，然后执行对应的回调uv__async_io。

4.1.3 主线程处理回调

```
static void uv__async_io(uv_loop_t* loop,
                         uv__io_t* w,
                         unsigned int events) {
  char buf[1024];
  ssize_t r;
  QUEUE queue;
  QUEUE* q;
  uv_async_t* h;

  for (;;) {
    // 消费所有的数据
    r = read(w->fd, buf, sizeof(buf));
        // 数据大小大于buf长度（1024），则继续消费
    if (r == sizeof(buf))
      continue;
```

```
            // 成功消费完毕，跳出消费的逻辑
        if (r != -1)
          break;
            // 读繁忙
        if (errno == EAGAIN || errno == EWOULDBLOCK)
          break;
            // 读被中断，继续读
        if (errno == EINTR)
          continue;
        abort();
      }
      // 把async_handles队列里的所有节点都移到queue变量中
      QUEUE_MOVE(&loop->async_handles, &queue);
      while (!QUEUE_EMPTY(&queue)) {
        // 逐个取出节点
        q = QUEUE_HEAD(&queue);
        // 根据结构体字段获取结构体首地址
        h = QUEUE_DATA(q, uv_async_t, queue);
        // 从队列中移除该节点
        QUEUE_REMOVE(q);
        // 重新插入async_handles队列，等待下次事件
        QUEUE_INSERT_TAIL(&loop->async_handles, q);
        /*
         将第一个参数和第二个参数进行比较，如果相等，
         则将第三参数写入第一个参数，返回第二个参数的值，
         如果不相等，则返回第一个参数的值。
        */
        /*
              判断触发了哪些async。pending在uv_async_send
里设置成1，
              如果pending等于1，则清0，返回1.如果pending等
于0，则返回0
            */
        if (cmpxchgi(&h->pending, 1, 0) == 0)
          continue;

        if (h->async_cb == NULL)
          continue;
        // 执行上层回调
        h->async_cb(h);
```

```
        }
      }
```

uv__async_io会遍历async_handles队列，pending等于1的话说明任务完成，然后执行对应的回调并清除标记位。

4.2 线程池的实现

了解了Libuv中子线程和主线程的通信机制后，我们来看一下线程池的实现。

4.2.1 线程池的初始化

线程池是懒初始化的，Node.js启动的时候，并没有创建子线程，而是在提交第一个任务给线程池时，线程池才开始初始化。我们先看线程池的初始化逻辑，然后再看它的使用。

```
1   static void init_threads(void) {
2     unsigned int i;
3     const char* val;
4     // 默认线程数4个，static uv_thread_t
5   default_threads[4];
6     nthreads = ARRAY_SIZE(default_threads);
7     // 判断用户是否在环境变量中设置了线程数，是的话取用户定
8   义的
9     val = getenv("UV_THREADPOOL_SIZE");
10    if (val != NULL)
11      nthreads = atoi(val);
12    if (nthreads == 0)
13      nthreads = 1;
14    // #define MAX_THREADPOOL_SIZE 128最多128个线程
15    if (nthreads > MAX_THREADPOOL_SIZE)
16      nthreads = MAX_THREADPOOL_SIZE;
17
18    threads = default_threads;
19    // 超过默认大小，重新分配内存
20    if (nthreads > ARRAY_SIZE(default_threads)) {
```

```
            threads = uv__malloc(nthreads *
sizeof(threads[0]));
        }
      // 初始化条件变量，用于有任务时唤醒子线程，没有任务时挂
起子线程
      if (uv_cond_init(&cond))
        abort();
      // 初始化互斥变量，用于多个子线程互斥访问任务队列
      if (uv_mutex_init(&mutex))
        abort();

      // 初始化三个队列
      QUEUE_INIT(&wq);
      QUEUE_INIT(&slow_io_pending_wq);
      QUEUE_INIT(&run_slow_work_message);

      // 创建多个线程，工作函数为worker，sem为worker入参
      for (i = 0; i < nthreads; i++)
        if (uv_thread_create(threads + i, worker, &sem))
          abort();
    }
```

线程池初始化时，会根据配置的子线程数创建对应数量的线程。默认是4个，最大128个子线程（不同版本的Libuv可能会不一样），我们也可以通过环境变量设置自定义的大小。线程池的初始化主要是初始化一些数据结构，然后创建多个线程，接着在每个线程里执行worker函数处理任务。后面我们会分析worker的逻辑。

4.2.2 提交任务到线程池

了解线程池的初始化之后，我们看一下如何给线程池提交任务

```
    // 给线程池提交一个任务
    void uv__work_submit(uv_loop_t* loop,
                         struct uv__work* w,
                         enum uv__work_kind kind,
                         void (*work)(struct uv__work* w),
```

```
                void (*done)(struct uv__work* w, int
status)){
       /*
         保证已经初始化线程，并只执行一次，所以线程池是在提交
第一个
        任务的时候才被初始化，init_once -> init_threads
        */
      uv_once(&once, init_once);
      w->loop = loop;
      w->work = work;
      w->done = done;
      post(&w->wq, kind);
    }
```

这里把业务相关的函数和任务完成后的回调函数封装到uv__work结构体中。uv__work结构定义如下。

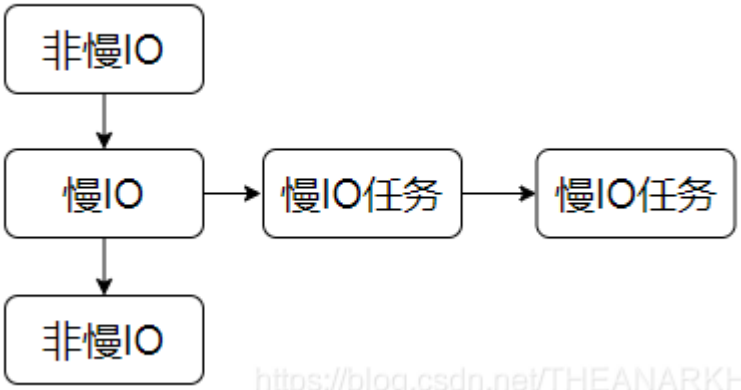
```
   struct uv__work {
     void (*work)(struct uv__work *w);
     void (*done)(struct uv__work *w, int status);
     struct uv_loop_s* loop;
     void* wq[2];
   };
```

然后调调用post函数往线程池的队列中加入一个新的任务。Libuv把任务分为三种类型，慢IO（DNS解析）、快IO（文件操作）、CPU密集型等，kind就是说明任务的类型的。我们接着看post函数。

```
    static void post(QUEUE* q, enum uv__work_kind kind)
{
      // 加锁访问任务队列，因为这个队列是线程池共享的
      uv_mutex_lock(&mutex);
      // 类型是慢IO
      if (kind == UV__WORK_SLOW_IO) {
        /*
        插入慢IO对应的队列，Libuv这个版本把任务分为几种类型，
       对于慢IO类型的任务，Libuv是往任务队列里面插入一个特殊的
节点
```

```
        run_slow_work_message，然后用slow_io_pending_wq维护了一个慢IO
          任务的队列，当处理到run_slow_work_message这个节点的时候，
          Libuv会从slow_io_pending_wq队列里逐个取出任务节点来执行。
        */
        QUEUE_INSERT_TAIL(&slow_io_pending_wq, q);
        /*
          有慢IO任务的时候，需要给主队列wq插入一个消息节点
              run_slow_work_message,说明有慢IO任务，所以如果
              run_slow_work_message是空，说明还没有插入主队列。需要进行
              q = &run_slow_work_message;赋值，然后把
              run_slow_work_message插入主队列。如果run_slow_work_message
              非空，说明已经插入线程池的任务队列了。解锁然后直接返回。
        */
        if (!QUEUE_EMPTY(&run_slow_work_message)) {
          uv_mutex_unlock(&mutex);
          return;
            }
            // 说明run_slow_work_message还没有插入队列，准备插入队列
          q = &run_slow_work_message;
      }
      // 把节点插入主队列，可能是慢IO消息节点或者一般任务
      QUEUE_INSERT_TAIL(&wq, q);
      /*
         有空闲线程则唤醒它，如果大家都在忙，
          则等到它忙完后就会重新判断是否还有新任务
         */
      if (idle_threads > 0)
        uv_cond_signal(&cond);
        // 操作完队列，解锁
      uv_mutex_unlock(&mutex);
    }
```

这就是Libuv中线程池的生产者逻辑。任务队列的架构如图4-3所示。



除了上面提到的，Libuv还提供了另外一种生产任务的方式，即uv_queue_work函数，它只提交CPU密集型的任务（在Node.js的crypto模块中使用）。下面我们看uv_queue_work的实现。

```
    int uv_queue_work(uv_loop_t* loop,
                      uv_work_t* req,
                      uv_work_cb work_cb,
                      uv_after_work_cb after_work_cb) {
      if (work_cb == NULL)
        return UV_EINVAL;

      uv__req_init(loop, req, UV_WORK);
      req->loop = loop;
      req->work_cb = work_cb;
      req->after_work_cb = after_work_cb;
      uv__work_submit(loop,
              &req->work_req,
              UV__WORK_CPU,
              uv__queue_work,
              uv__queue_done);
      return 0;
    }
```

uv_queue_work函数其实也没有太多的逻辑，它保存用户的工作函数和回调到request中。然后把uv__queue_work和uv__queue_done封装到uv__work中，接着提交任务到线程池中。所以当这个任务被执行的时候。它会执行工作函数uv__queue_work。

```
    static void uv__queue_work(struct uv__work* w) {
      // 通过结构体某字段拿到结构体地址
      uv_work_t* req = container_of(w, uv_work_t,
work_req);
      req->work_cb(req);
    }
```

我们看到uv__queue_work其实就是对用户定义的任务函数进行了封装。这时候我们可以猜到，uv__queue_done也只是对用户回调的简单封装，即它会执行用户的回调。

4.2.3 处理任务

我们提交了任务后，线程自然要处理，初始化线程池的时候我们分析过，worker函数是负责处理任务。我们看一下worker函数的逻辑。

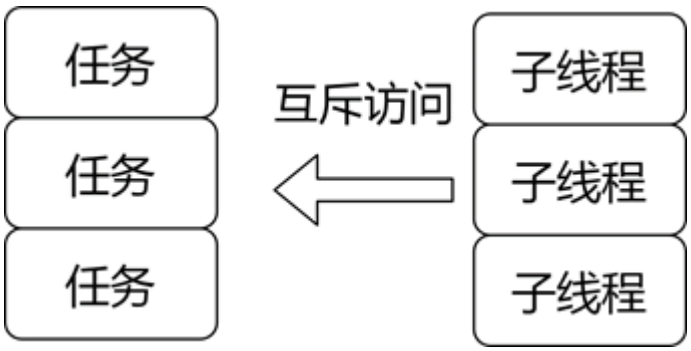
```
    static void worker(void* arg) {
      struct uv__work* w;
      QUEUE* q;
      int is_slow_work;
      // 线程启动成功
      uv_sem_post((uv_sem_t*) arg);
      arg = NULL;
      // 加锁互斥访问任务队列
      uv_mutex_lock(&mutex);
      for (;;) {
        /*
          1 队列为空
          2 队列不为空，但是队列中只有慢IO任务且正在执行的慢
IO任务
                  个数达到阈值则空闲线程加一，防止慢IO占用过多
```

```
线程，导致
                其它快的任务无法得到执行
        */
        while (QUEUE_EMPTY(&wq) ||
            (QUEUE_HEAD(&wq) == &run_slow_work_message
&&
            QUEUE_NEXT(&run_slow_work_message) == &wq
&&
            slow_io_work_running >=
slow_work_thread_threshold()))       {
          idle_threads += 1;
          // 阻塞，等待唤醒
          uv_cond_wait(&cond, &mutex);
            // 被唤醒，开始干活，空闲线程数减一
          idle_threads -= 1;
        }
        // 取出头结点，头指点可能是退出消息、慢IO，一般请求
        q = QUEUE_HEAD(&wq);
         // 如果头结点是退出消息，则结束线程
            if (q == &exit_message) {
              /*
                 唤醒其它因为没有任务正阻塞等待任务的线程，
                  告诉它们准备退出
                */
          uv_cond_signal(&cond);
          uv_mutex_unlock(&mutex);
          break;
        }
        // 移除节点
            QUEUE_REMOVE(q);
            // 重置前后指针
        QUEUE_INIT(q);
        is_slow_work = 0;
            /*
          如果当前节点等于慢IO节点，上面的while只判断了是不
是只有慢
              IO任务且达到阈值，这里是任务队列里肯定有非慢IO
任务，可能有
              慢IO，如果有慢IO并且正在执行的个数达到阈值，则
先不处理该慢
              IO任务，继续判断是否还有非慢IO任务可执行。
```

```
57              */
58             if (q == &run_slow_work_message) {
59               // 达到阈值，该节点重新入队，因为刚才被删除了
60               if (slow_io_work_running >=
61 slow_work_thread_threshold())          {
62                 QUEUE_INSERT_TAIL(&wq, q);
63                 continue;
64               }
65               /*
66                       没有慢IO任务则继续，这时候
67 run_slow_work_message
68                       已经从队列中被删除，下次有慢IO的时候重新入
69 队
70                     */
71               if (QUEUE_EMPTY(&slow_io_pending_wq))
72                 continue;
73               // 有慢IO，开始处理慢IO任务
74               is_slow_work = 1;
75               /*
76                       正在处理慢IO任务的个数累加，用于其它线程判
77 断慢IO任务个
78                       数是否达到阈值，slow_io_work_running是
79 多个线程共享的变量
80                     */
81               slow_io_work_running++;
82               // 摘下一个慢IO任务
83               q = QUEUE_HEAD(&slow_io_pending_wq);
84                     // 从慢IO队列移除
85               QUEUE_REMOVE(q);
86               QUEUE_INIT(q);
87               /*
88                   取出一个任务后，如果还有慢IO任务则把慢IO标记节
89 点重新入
90                       队，表示还有慢IO任务，因为上面把该标记节
91 点出队了
92                 */
93               if (!QUEUE_EMPTY(&slow_io_pending_wq)) {
94                 QUEUE_INSERT_TAIL(&wq,
95 &run_slow_work_message);
96                 // 有空闲线程则唤醒它，因为还有任务处理
97                 if (idle_threads > 0)
```

```
                uv_cond_signal(&cond);
            }
          }
          // 不需要操作队列了，尽快释放锁
          uv_mutex_unlock(&mutex);
          // q是慢IO或者一般任务
          w = QUEUE_DATA(q, struct uv__work, wq);
          // 执行业务的任务函数，该函数一般会阻塞
          w->work(w);
          // 准备操作loop的任务完成队列，加锁
              uv_mutex_lock(&w->loop->wq_mutex);
              // 置空说明执行完了，见cancel逻辑
          w->work = NULL;
          /*
                执行完任务,插入到loop的wq队列,在
uv__work_done的时候会
                执行该队列的节点
              */
          QUEUE_INSERT_TAIL(&w->loop->wq, &w->wq);
          // 通知loop的wq_async节点
          uv_async_send(&w->loop->wq_async);
              uv_mutex_unlock(&w->loop->wq_mutex);
              // 为下一轮操作任务队列加锁
          uv_mutex_lock(&mutex);
          /*
                执行完慢IO任务，记录正在执行的慢IO个数变量减
1,
                上面加锁保证了互斥访问这个变量
              */
          if (is_slow_work) {
            slow_io_work_running--;
          }
        }
      }
```

我们看到消费者的逻辑似乎比较复杂，对于慢IO类型的任务，Libuv限制了处理慢IO任务的线程数，避免耗时比较少的任务得不到处理。其余的逻辑和一般的线程池类似，就是互斥访问任务队列，然后取出节点执行，执行完后通知主线程。结构如图4-4所示。



4.2.4 通知主线程

线程执行完任务后，并不是直接执行用户回调，而是通知主线程，由主线程统一处理，这是Node.js单线程事件循环的要求，也避免了多线程带来的复杂问题，我们看一下这块的逻辑。一切要从Libuv的初始化开始

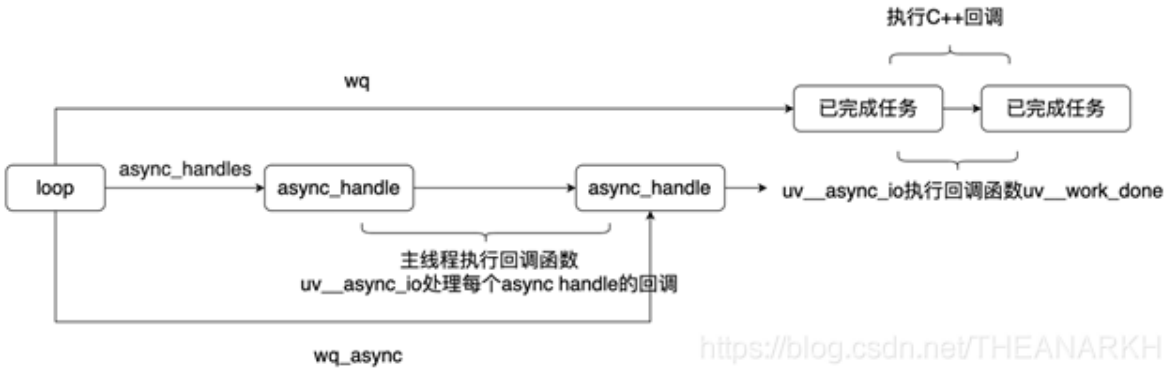
```
uv_default_loop();-> uv_loop_init();->
uv_async_init(loop, &loop->wq_async, uv__work_done);
```

刚才我们已经分析过主线程和子线程的通信机制，wq_async是用于线程池中子线程和主线程通信的async handle，它对应的回调是uv__work_done。所以当一个线程池的线程任务完成时，通过uv_async_send(&w->loop->wq_async)设置loop->wq_async.pending = 1，然后通知IO观察者，Libuv在Poll IO阶段就会执行该handle对应的回调uv__work_done函数。那么我们就看看这个函数的逻辑。

```
    void uv__work_done(uv_async_t* handle) {
      struct uv__work* w;
      uv_loop_t* loop;
      QUEUE* q;
      QUEUE wq;
      int err;
      // 通过结构体字段获得结构体首地址
      loop = container_of(handle, uv_loop_t, wq_async);
      // 准备处理队列，加锁
      uv_mutex_lock(&loop->wq_mutex);
      /*
```

```
          loop->wq是已完成的任务队列。把loop->wq队列的节点全部移到
            wp变量中，这样一来可以尽快释放锁
          */
        QUEUE_MOVE(&loop->wq, &wq);
        // 不需要使用了，解锁
        uv_mutex_unlock(&loop->wq_mutex);
        // wq队列的节点来自子线程插入
        while (!QUEUE_EMPTY(&wq)) {
          q = QUEUE_HEAD(&wq);
          QUEUE_REMOVE(q);
          w = container_of(q, struct uv__work, wq);
              // 等于uv__canceled说明这个任务被取消了
          err = (w->work == uv__cancelled) ? UV_ECANCELED : 0;
          // 执行回调
          w->done(w, err);
        }
      }
```

该函数的逻辑比较简单，逐个处理已完成的任务节点，执行回调，在Node.js中，这里的回调是C++层，然后再到JS层。结构图如图4-5所示。



4.2.5 取消任务

线程池的设计中，取消任务是一个比较重要的能力，因为在线程里执行的都是一些耗时或者引起阻塞的操作，如果能及时取消一个任务，将会减轻很多没必

要的处理。不过Libuv实现中，只有当任务还在等待队列中才能被取消，如果一个任务正在被线程处理，则无法取消了。我们先看一下Libuv中是如何实现取消任务的。Libuv提供了uv__work_cancel函数支持用户取消提交的任务。我们看一下它的逻辑。

```
    static int uv__work_cancel(uv_loop_t* loop,
uv_req_t* req, struct uv__work* w) {
      int cancelled;
      // 加锁，为了把节点移出队列
      uv_mutex_lock(&mutex);
      // 加锁，为了判断w->wq是否为空
      uv_mutex_lock(&w->loop->wq_mutex);
      /*
        cancelled为true说明任务还在线程池队列等待处理
          1 处理完，w->work == NULL
          2 处理中，QUEUE_EMPTY(&w->wq)为true，因
            为worker在摘下一个任务的时候，重置prev和next指针
          3 未处理，!QUEUE_EMPTY(&w->wq)是true 且w->work
!= NULL
      */
      cancelled = !QUEUE_EMPTY(&w->wq) && w->work !=
NULL;
      // 从线程池任务队列中删除该节点
      if (cancelled)
        QUEUE_REMOVE(&w->wq);

      uv_mutex_unlock(&w->loop->wq_mutex);
      uv_mutex_unlock(&mutex);
      // 正在执行或者已经执行完了，则不能取消
      if (!cancelled)
        return UV_EBUSY;
      // 打取消标记，Libuv执行回调的时候用到
      w->work = uv__cancelled;

      uv_mutex_lock(&loop->wq_mutex);
       /*
         插入loop的wq队列，对于取消的动作，Libuv认为是任务执
行完了。
            所以插入已完成的队列，执行回调的时候会通知用户该任
```

```
35  务的执行结果
36              是取消，错误码是UV_ECANCELED
37       */
38      QUEUE_INSERT_TAIL(&loop->wq, &w->wq);
39      // 通知主线程有任务完成
        uv_async_send(&loop->wq_async);
        uv_mutex_unlock(&loop->wq_mutex);

        return 0;
    }
```

在Libuv中，取消任务的方式就是把节点从线程池待处理队列中删除，然后打上取消的标记（w->work = uv__cancelled），接着把该节点插入已完成队列，Libuv在处理已完成队列的节点时，判断如果w->work == uv__cancelled则在执行用户回调时，传入错误码UV_ECANCELED，我们看到uv__work_cancel这个函数定义前面加了一个static，说明这个函数是只在本文件内使用的，Libuv对外提供的取消任务的接口是uv_cancel。