title	date	updated	tags	categories

title	date	updated	tags	categories
libuv源码分析（八）线程池（Threadpool）	2019-04-23 15:00:08 UTC	2019-04-24 01:31:28 UTC	libuv, node.js, eventloop	源码分析

- http://docs.libuv.org/en/v1.x/threadpool.html
- http://docs.libuv.org/en/v1.x/dns.html

线程池在程序设计中是常用的提升并发计算能力、提升吞吐量的常用手段，在 libnv 也不例外，并且结合事件循环，实现了异步支持。

libuv 提供可用于执行用户代码的线程池，并且能够在任务完成时，向事件循环线程发送消息通知主线程完成收尾工作。

默认情况下，线程池的大小是 `4`，但是可以在启动阶段通过设置 `UV_THREADPOOL_SIZE` 环境变量进行修改，最大为 `128`。

初始化

线程池全局的并且跨所有事件循环共享，当特定的函数使用线程池时（例如，调用 `uv_queue_work()`），libuv 通过 `init_threads` 函数预分配和初始化一定数量的线程，初始化函数只会被调用一次，这会带来一定的内存开销，但是可以提升运行时性能。

线程池初始化

线程池是由 `init_threads` 函数初始化的：

```
static void init_threads(void) {
  unsigned int i;
  const char* val;
  uv_sem_t sem;

  nthreads = ARRAY_SIZE(default_threads);
  val = getenv("UV_THREADPOOL_SIZE");
  if (val != NULL)
    nthreads = atoi(val);
  if (nthreads == 0)
    nthreads = 1;
  if (nthreads > MAX_THREADPOOL_SIZE)
    nthreads = MAX_THREADPOOL_SIZE;

  threads = default_threads;
  if (nthreads > ARRAY_SIZE(default_threads)) {
    threads = uv__malloc(nthreads * sizeof(threads[0]));
    if (threads == NULL) {
      nthreads = ARRAY_SIZE(default_threads);
      threads = default_threads;
    }
  }

  if (uv_cond_init(&cond))
    abort();

  if (uv_mutex_init(&mutex))
    abort();

  QUEUE_INIT(&wq);
  QUEUE_INIT(&slow_io_pending_wq);
  QUEUE_INIT(&run_slow_work_message);

  if (uv_sem_init(&sem, 0))
```

```
    abort();

  for (i = 0; i < nthreads; i++)
    if (uv_thread_create(threads + i, worker, &sem))
      abort();

  for (i = 0; i < nthreads; i++)
    uv_sem_wait(&sem);

  uv_sem_destroy(&sem);
}
```

初始化逻辑如下：

1. 线程池中线程数量，并分配用于存储线程信息的内存空间；
2. 初始化静态全局的 `线程锁` 和 `线程条件变量` ；
3. 初始化静态全局 `uv__work` 队列；
    i. `wq` 待执行的任务队列，未执行完毕， `loop->wq` 同为任务队列，但是保持的是执行完毕的任务；
    ii. `slow_io_pending_wq` 慢IO延迟任务队列；
    iii. `run_slow_work_message` 慢IO延迟任务队列代表，当存在慢IO延迟任务队列时，`run_slow_work_message` 被插入到 `wq` 中代替所有慢IO任务排队；
4. 创建一定数量的线程；
5. 等待所以线程创建完成。

在创建线程的时候，线程执行的函数是 `worker` ，该函数负责在线程中处理 `wq` 上的任务。

`worker` 实现如下：

```
/* To avoid deadlock with uv_cancel() it's crucial that the worker
 * never holds the global mutex and the loop-local mutex at the same time.
```

```
 */
static void worker(void* arg) {
  struct uv__work* w;
  QUEUE* q;
  int is_slow_work;

  uv_sem_post((uv_sem_t*) arg);
  arg = NULL;

  // 加锁 mutex
  // 因为只有一个线程能抢占锁，所以多个线程也只能一个接一个的进入循环
  // 因为整个线程池中线程创建过程中不会出现其他线程在其他位置抢占并锁定 mutex 的情形出现，
  // 所以只有该位置会抢占加锁，而后很快释放锁，所以线程池中的线程之后短暂的阻塞在这里。
  // 工作线程需要不断的等待处理任务，所以需要进入死循环
  uv_mutex_lock(&mutex);
  for (;;) {
    /* `mutex` should always be locked at this point. */

    /* Keep waiting while either no work is present or only slow I/O
       and we're at the threshold for that. */
    // 条件满足时，没有任务需要处理，线程进入挂起等待状态，等待被唤醒。
    while (
      // 任务队列为空
      QUEUE_EMPTY(&wq) ||
      // 任务队列非空，但是
           // 队列头部被标记为慢速IO任务
           // 且该队列中只有run_slow_work_message一个数据节点
           // 且正在处理的慢IO任务超过阈值（默认2）
           //   该一个条件避免太多线程同时都在处理慢IO操作
           //   达到阈值后空闲的线程不再接慢IO任务而是挂起，等待非慢IO操作任务 能有机会尽快得到处理
           //   正在进行的慢IO任务完成后，阈值限制解除，可以接慢IO任务
           //   最终，保证了最多只有 `(nthreads + 1) / 2` 个线程处理慢IO
           //   区分了快车道和慢车道后，能有效避免慢车堵快车，提升性能
           (QUEUE_HEAD(&wq) == &run_slow_work_message
           && QUEUE_NEXT(&run_slow_work_message) == &wq
```

```
           && slow_io_work_running >= slow_work_thread_threshold())) {
      // 进入休息区，注意某线程在执行 while 循环时该线程一定抢占了 mutex，不论是首次还是后续执行
      // 线程挂起，等待唤醒
      // uv_cond_wait 会使线程挂起等待cond上的信号，为防止多线程同时调用 uv_cond_wait，必须提前加锁
      // uv_cond_wait 在挂起前会释放 mutex，其他阻塞在 mutex 上的线程会在 mutex 释放时被唤醒，并在唤醒时重新抢
      // 所以，阻塞在for循环外的多个线程中的某一个会重新抢占 mutex 执行到达此处挂起，又继续唤醒其他线程
      //   也可能唤醒 阻塞在 uv__work_submit -> post 函数提交任务的抢占锁的位置的线程（通常为主事件循环线程）
      // 挂起的线程都是空闲的线程，被唤醒后为非空闲的线程，所以需要更新空闲线程计数
      idle_threads += 1;
      uv_cond_wait(&cond, &mutex);
      idle_threads -= 1;
      // 挂起的线程在被唤醒后，一定不满足再次进入循环的条件，会继续向下执行
    }

    // 进入工作区，一共有三个区间，前后两个区间都有锁，中间的区间执行用户代码无锁

    // 线程被唤醒，开始干活

    // 以下操作因线程被唤醒时会自动对mutex上锁
    // 所以以下解锁前的区域对共享变量的操作都是安全的
    // 锁定区间代码同一时段只能有一个线程在执行
    // 因并无耗时任务，所以不会影响性能

    // 获取任务
    q = QUEUE_HEAD(&wq);
    // 如果收到线程退出消息，跳出循环，线程声明周期结束
    // 在外部发送消息通知线程主动退出，也可在外部kill线程
    if (q == &exit_message) {
      uv_cond_signal(&cond);
      uv_mutex_unlock(&mutex);
      break;
    }

    // 将任务摘出来
    QUEUE_REMOVE(q);
```

```
      QUEUE_INIT(q);  /* Signal uv_cancel() that the work req is executing. */

      // 初始化慢IO操作标记为0，即非慢IO操作
      is_slow_work = 0;
      if (q == &run_slow_work_message) {
        // 该任务为慢IO任务
        // 通常情况下，while 的第二个条件成立才能进入此段代码
        // 此时 q 只是一个慢IO任务标记，真正的任务在 slow_io_pending_wq 中
        // 所以需要特殊处理，获取真正的任务 q

        /* If we're at the slow I/O threshold, re-schedule until after all
           other work in the queue is done. */
        // 如果当前运行的慢IO操作的线程数达到阈值（2个线程）
        // 则将这些操作插入到 wq 队列末尾，延迟处理
        // 避免多个线程同时处理慢IO
        // 临界状态：已经有达到阈值限制个数的线程进入工作区处理慢IO任务，但是还没执行更新慢IO线程计数器代码，
        //          后续被慢IO任务唤醒的线程线程可能因为慢IO线程计数器未更新而满足进入条件。
        //          但是，因为该区间锁定了 mutex，阻塞在 uv_cond_wait 处的代码无法抢占锁无法执行，也就是无法跳出 w
        //          到 mutex 释放时，被唤醒的线程能够抢占锁时，计数器已经被更新了，前面所说的进入条件不再满足了。
        //    所以，条件满足时不能动，能动了条件又不满足了，本质上，两次判断在同一段锁定区间，所以以下情形应该难以出现
        if (slow_io_work_running >= slow_work_thread_threshold()) {
          QUEUE_INSERT_TAIL(&wq, q);
          continue;
        }

        /* If we encountered a request to run slow I/O work but there is none
           to run, that means it's cancelled => Start over. */
        // 如果慢IO队列为空，可能任务被取消
        if (QUEUE_EMPTY(&slow_io_pending_wq))
          continue;

        // 注意以上两处不需要 uv_mutex_unlock(&mutex)

        // 标记该线程正在处理慢IO操作，同时增加慢IO线程计数器
        is_slow_work = 1;
```

```
      slow_io_work_running++;

      // 从慢IO队列中重新获取任务
      q = QUEUE_HEAD(&slow_io_pending_wq);
      QUEUE_REMOVE(q);
      QUEUE_INIT(q);

      /* If there is more slow I/O work, schedule it to be run as well. */
      // 如果还有更多的慢IO操作，则将这些任务插入到 wq 队列末尾，本次只能处理 q 这一个任务
      if (!QUEUE_EMPTY(&slow_io_pending_wq)) {
        QUEUE_INSERT_TAIL(&wq, &run_slow_work_message);
        // 如果有空闲线程，唤醒
        if (idle_threads > 0)
          uv_cond_signal(&cond);
      }
    }

    // 解锁 mutex
    uv_mutex_unlock(&mutex);

    // 只有以下两行不涉及竞态资源读写，不需要加锁，实际也不能锁
    // 慢IO任务还是非慢IO任务，指的是w->work
    w = QUEUE_DATA(q, struct uv__work, wq);
    w->work(w);

    // 因为 loop 在多线程中共享，所以访问 loop 需要加锁
    uv_mutex_lock(&w->loop->wq_mutex);
    w->work = NULL;  /* Signal uv_cancel() that the work req is done
                        executing. */
    // 将完成的任务插入到 loop->wq 队列中，在主事件循环线程中处理
    QUEUE_INSERT_TAIL(&w->loop->wq, &w->wq);
    // 发送完成信号，唤醒事件询线程并处理
    uv_async_send(&w->loop->wq_async);
    uv_mutex_unlock(&w->loop->wq_mutex);
```

```
      /* Lock `mutex` since that is expected at the start of the next
       * iteration. */
      uv_mutex_lock(&mutex);
      if (is_slow_work) {
        /* `slow_io_work_running` is protected by `mutex`. */
        slow_io_work_running--;
      }
    }
  }
```

`uv_async_send` 已经分析过了，它向事件循环线程发送消息唤醒事件循环线程

主线程中的初始化工作

主线程中的初始化工作是先于线程池初始化的，这部分初始化完成了用于接收 work 线程消息的 AsyncHandle 的初始化工作。

`uv_async_send` 通过 `loop->wq_async` Handle 发送了消息，字段定义如下：

```
#define UV_LOOP_PRIVATE_FIELDS                                       \
  uv_mutex_t wq_mutex;                                               \
  uv_async_t wq_async;                                               \
```

`loop->wq_async` 是在 `uv_loop_init` 初始化的，如下：

```
int uv_loop_init(uv_loop_t* loop) {
  ...
  err = uv_async_init(loop, &loop->wq_async, uv__work_done);
  if (err)
    goto fail_async_init;
```

```
  uv__handle_unref(&loop->wq_async);
  loop->wq_async.flags |= UV_HANDLE_INTERNAL;
  ...
}
```

`loop->wq_async` 被解引用了，所以并不会影响 `loop` 的活动状态。

`loop->wq_async` 的事件处理函数是 `uv__work_done`，该函数在事件循环线程中执行，实现如下：

```
void uv__work_done(uv_async_t* handle) {
  struct uv__work* w;
  uv_loop_t* loop;
  QUEUE* q;
  QUEUE wq;
  int err;

  // 取出所有已完成的work，因与其他线程共享此变量，所以需要同步，因此此处可能会导致事件循环线程短暂阻塞
  loop = container_of(handle, uv_loop_t, wq_async);
  uv_mutex_lock(&loop->wq_mutex);
  QUEUE_MOVE(&loop->wq, &wq);
  uv_mutex_unlock(&loop->wq_mutex);

  // 遍历所有已完成的work，调用 w->done，done 函数由用户提供
  while (!QUEUE_EMPTY(&wq)) {
    q = QUEUE_HEAD(&wq);
    QUEUE_REMOVE(q);

    w = container_of(q, struct uv__work, wq);
    err = (w->work == uv__cancelled) ? UV_ECANCELED : 0;
    w->done(w, err);
  }
}
```

至此，从线程池初始化到线程处理任务再到线程与事件循环线程通信最后事件循环线程清理已完成的任务的整个流程已经分析完成。

下面，该了解一下，如何向线程池提交任务任务了。

任务提交

向线程池提交任务的 API 是 `uv_queue_work`，也实现线程池唯一对外暴露的 API，下面我们看它的具体实现：

```
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` 初始化了一个 `uv_work_t` 类型的 `request`，`work_cb` 为线程池中线程执行的函数，`after_work_cb` 为 `work_cb` 执行完成之后在事件循环线程中执行的函数，`req->work_req` 是队列节点。最后通过 `uv__work_submit` 向线程池中提交任务。

最后通过调用 `uv__work_submit` 向线程池中提交任务，`uv__work_submit` 的两个实参 `uv__queue_work` 和 `uv__queue_done` 分别对 `work_cb` 和 `after_work_cb` 进行简单的封装。实现如下：

```
static void uv__queue_work(struct uv__work* w) {
  uv_work_t* req = container_of(w, uv_work_t, work_req);

  req->work_cb(req);
}
```

```
static void uv__queue_done(struct uv__work* w, int err) {
  uv_work_t* req;

  req = container_of(w, uv_work_t, work_req);
  uv__req_unregister(req->loop, req);

  if (req->after_work_cb == NULL)
    return;

  req->after_work_cb(req, err);
}
```

`uv__work_submit` 的实现如下：

```
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)) {
  uv_once(&once, init_once);
  w->loop = loop;
  w->work = work;
```

```
  w->done = done;
  post(&w->wq, kind);
}
```

`uv__work_submit` 通过调用 `init_once` 初始化线程池，`uv_once` 确保线程池初始化函数 `init_once` 只会被调用一次。

然后对 `uv__work` 进行初始化，`w->work` 在工作线程 `worker` 中调用，`w->done` 在事件循环线程 `uv__work_done` 中调用

最后通过调用 `post` 提交任务，`post` 实现如下：

```
static void post(QUEUE* q, enum uv__work_kind kind) {
  uv_mutex_lock(&mutex);
  if (kind == UV__WORK_SLOW_IO) {
    /* Insert into a separate queue. */
    QUEUE_INSERT_TAIL(&slow_io_pending_wq, q);
    if (!QUEUE_EMPTY(&run_slow_work_message)) {
      /* Running slow I/O tasks is already scheduled => Nothing to do here.
         The worker that runs said other task will schedule this one as well. */
      uv_mutex_unlock(&mutex);
      return;
    }
    q = &run_slow_work_message;
  }

  QUEUE_INSERT_TAIL(&wq, q);
  if (idle_threads > 0)
    uv_cond_signal(&cond);
  uv_mutex_unlock(&mutex);
}
```

因为任务队列会被线程池中的多个线程并发访问，所以在操作队列之前需要先加锁，完成之后需要解锁。如果有空闲的线程，则立即唤醒它们进行工作。

在 `post` 中，慢IO任务被插入到 `slow_io_pending_wq` 队列中，如果 `run_slow_work_message` 不在 `wq` 中，则需要将 `run_slow_work_message` 插入 `wq` 队列尾部，标识 `slow_io_pending_wq` 中存在任务，当 `run_slow_work_message` 得到被处理机会时，处理慢任务队列中的任务。

在 `uv_queue_work` 中的 `uv__work_submit` 调用时，传递的是 `UV__WORK_CPU` 表示 CPU 密集型任务。

任务可能在任意一个线程中提交，通常是在事件循环线程中提交，但是也有可能在work线程中提交，即，`w->work` 和 `w->done` 这两个函数中都有可能调用 `uv__work_submit`，这取决于实现。

将任务提交到工作队列中，这一阶段的工作就已经完成了，线程池中的线程可以开始工作了。

至此，整个线程池的工作原理已经分析完成，整个工作流程大致可分为三个阶段：

1. 提交任务；
2. work线程处理任务，完成后通知事件循环线程；
3. 事件循环线程收到通知后完成收尾工作。

在接口使用中，是不需要太关心以上流程和工作原理的，更应该关系 `work_cb` 和 `after_work_cb` 以及其他逻辑的实现。

Example

线程池在 libuv 内部用于完成所有文件系统操作（`requests`），也用于实现 `getaddrinfo` 和 `getnameinfo` 等 DNS 相关的操作（`requests`）。搜索 `uv_queue_work` 可找到相关使用位置。可以这些内部实现作为使用示例，在内部，并不通过 `uv_queue_work` 提交任务，而是直接调用 `uv__work_submit`，因为它们都有各自不同的 `uv__x_work` 和 `uv__x_done` 实现。

源文件地址：https://github.com/liuyanjie/knowledge/tree/master/node.js/libuv/8-libuv-threadpool.md