Tree: 62e84b27e7 liuyanjie.github.io / source / _posts / 2-libuv源码分析（三）资源抽象：Handle 和 Request.md

Find file Copy path

liuyanjie add articles 2c06277 on Apr 24

1 contributor

768 lines (563 sloc) | 31.6 KB

Raw Blame History

title	date	updated	tags	categories

title	date	updated	tags	categories
libuv源码分析（三）资源抽象：Handle 和 Request	2019-04-23 15:00:03 UTC	2019-04-24 01:31:28 UTC	libuv, node.js, eventloop	源码分析

Handle 是 libuv 设计实现的核心部分之一，根据官方描述：**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。

libuv 采用了组合的方式实现代码复用，并且达到了面向对象编程中的继承的效果。

Handle 有很多种不同的类型，这些类型有一个共同的、公共的基础结构 `uv_handle_s`，因结构体内存布局字节对齐所有子类型都可以强制类型转换成 `uv_handle_t` 类型，所以所有能够应用在 `uv_handle_t` 上的基础API都可用于子类型的 `handle`。

在开始对不同类型的 Handle 开始分析之前，将会对 Handle 进行整体分析。

首先，看一下 libuv 中的 Handle 相关的类型声明和定义：

https://github.com/libuv/libuv/blob/v1.28.0/include/uv.h#L201

```
/* 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;
```

以上是 libuv 中所有的 `handle` 声明，并且都起了类型别名，命名规律显而易见，`uv_loop_t` 也在其中，`uv_loop_t` 作为所有资源的统一入口同样也是一种资源，而且是生命周期最长的资源。

下面来开一下 `uv_handle_s` 的结构定义：

https://github.com/libuv/libuv/blob/v1.28.0/include/uv.h#L411

```
#define UV_HANDLE_FIELDS     \
  /* public */               \ 公有字段：
  void* data;                \    数据指针，指向实际的数据，可用于指向任意用户自定义数据地址
  /* read-only */            \ 只读字段：
  uv_loop_t* loop;           \    事件循环对象指针，指向事件循环对象
  uv_handle_type type;       \    类型
  /* private */              \ 私有字段：
  uv_close_cb close_cb;      \    关闭回调，Handle关闭时触发的回调函数
  void* handle_queue[2];     \    队列节点，用于链接前后节点，形成双向队列，该字段是含有两个成员的数组，分别表示
  union {                    \    联合体字段，TODO：待补充作用
    int fd;                  \
    void* reserved[4];       \
  } u;                       \
  UV_HANDLE_PRIVATE_FIELDS   \  平台相关字段，libuv在各个不同平台上的实现是不同的，为支持不同平台，需针对不同平
```

宏 `UV_HANDLE_FIELDS` 被放到了 `struct uv_handle_s`：

https://github.com/libuv/libuv/blob/view-v1.28.0/include/uv.h#L426

```
/* The abstract base class of all handles. */
struct uv_handle_s {
  UV_HANDLE_FIELDS
};
```

如注释所言，`struct uv_handle_s` 是所有 `handle` 的抽象基类。

在 `*nix` 平台下，`UV_HANDLE_PRIVATE_FIELDS` 宏定义如下：

https://github.com/libuv/libuv/blob/v1.x/src/uv-common.h#L62

```
#define UV_HANDLE_PRIVATE_FIELDS     \
  uv_handle_t* next_closing;         \ 下一个 处于 closing 状态的 handle 的地址，形成一个单向的链表
  unsigned int flags;                \ handle 的 标识，handle 存在很多标识，通过位运算获取。
```

`flags` 可以使用的标识如下：

```
/* Handle flags. Some flags are specific to Windows or UNIX. */
enum {
  /* Used by all handles. */
  UV_HANDLE_CLOSING                 = 0x00000001,
  UV_HANDLE_CLOSED                  = 0x00000002,
  UV_HANDLE_ACTIVE                  = 0x00000004,
  UV_HANDLE_REF                     = 0x00000008,
  UV_HANDLE_INTERNAL                = 0x00000010,
  UV_HANDLE_ENDGAME_QUEUED          = 0x00000020,
```

```
  /* Used by streams. */
  UV_HANDLE_LISTENING                 = 0x00000040,
  UV_HANDLE_CONNECTION                = 0x00000080,
  UV_HANDLE_SHUTTING                  = 0x00000100,
  UV_HANDLE_SHUT                      = 0x00000200,
  UV_HANDLE_READ_PARTIAL              = 0x00000400,
  UV_HANDLE_READ_EOF                  = 0x00000800,

  // ... 其他标识省略标识
};
```

所有 `handle` 都具备 `UV_HANDLE_ACTIVE` `UV_HANDLE_CLOSED` 等几个公共状态，还有一些特地 `handle` 特定的状态。

以上为 `uv_handle_s` 定义，其中字段是通过 `UV_HANDLE_FIELDS` 宏定义和引入的，这样做的目的是为了复用字段定义部分的代码，能有效降低代码量，提升可维护性。相关字段的功能描述见字段后的说明。

`uv_handle_t` 实际上就是作为所有其他 `handle` 的 `基类` 存在的，其他 `handle` 通过 `组合` 的方式集成了 `uv_handle_t` 字段，通过强制类型转换，可以转换为 `uv_handle_t`，之后在其上应用 `uv_handle_t` 的相关方法。

以 `stream` 为例看一下其类型定义：

https://github.com/libuv/libuv/blob/v1.x/include/uv.h#L461

```
#define UV_STREAM_FIELDS                        \
  /* number of bytes queued for writing */      \
  size_t write_queue_size;                      \
  uv_alloc_cb alloc_cb;                         \
  uv_read_cb read_cb;                           \
  /* private */                                 \
  UV_STREAM_PRIVATE_FIELDS
```

```
/*
 * uv_stream_t is a subclass of uv_handle_t.
 *
 * uv_stream is an abstract class.
 *
 * uv_stream_t is the parent class of uv_tcp_t, uv_pipe_t and uv_tty_t.
 */
struct uv_stream_s {
  UV_HANDLE_FIELDS
  UV_STREAM_FIELDS
};
```

在 `uv_stream_s` 结构体中，包含了 `uv_handle_s` 的宏定义 `UV_HANDLE_FIELDS` 和 `uv_stream_s` 类型特定宏定义 `UV_STREAM_FIELDS`，`uv_stream_s` 和 `uv_handle_s` 在结构体内存布局上存在公共的部分且是以起始地址对齐的，`uv_stream_s` 比 `uv_handle_s` 多出一块特有的部分，可以通过强制类型转换将 `uv_stream_s` 转换为 `uv_handle_s`。

`uv_handle_t` 定义了所有 `handle` 公共的部分，作为一个抽象基类存在。`uv_handle_t` 是不直接使用的，因为它并不能支持用户需求，无实际意义，实际上，在使用其他派生类型时，会间接使用 `uv_handle_t`。所有派生类型在初始化的时候，也进行了 `uv_handle_t` 的初始化，这类似于高级语言构造函数在执行时常常需要调用基类构造函数一样。除初始化操作以外，同样还有其他操作需要调用 `uv_handle_t` 函数的相关操作。

一般来说，派生类型具备如下几个操作：

- `uv_{handle}_init`：初始化 `handle` 结构，把各个字段设置成合理值，并插入 `loop->handle_queue` 队列；
- `uv_{handle}_start`：启动 `handle` 使其处于 `UV_HANDLE_ACTIVE` 状态；
- `uv_{handle}_stop`：停止 `handle` 使其处于 `UV_HANDLE_CLOSED` 状态，并移出 `loop->handle_queue` 队列。

以上各派生类型的公共操作，提现了 `handle` 的声明周期，和 `loop` 生命周期类似，除此之外还包括一些特定 `handle` 特定处理逻辑。

因为各个派生类型的初始化/启动/停止逻辑都有不同，所以并没有公共的初始化/启动/停止方法，每个派生类型根据需要提供特定的初始化/启动/停止函数，它们都在内部初始化/启动/停止 `uv_handle_t` 。对应的方法为：

- `uv__handle_init`
- `uv__handle_start`
- `uv__handle_stop`

从命名可以看出这些都是不对外暴露的方法。

接下来我就来看一下 `handle` 的初始化。

Init：`uv__handle_init`

`uv_handle_t` 的初始化代码是用宏 `uv__handle_init` 定义的宏函数，实现如下：

https://github.com/libuv/libuv/blob/v1.28.0/src/uv-common.h#L282

```
#define uv__handle_init(loop_, h, type_)                                \
  do {                                                                  \
    (h)->loop = (loop_);                                                \
    (h)->type = (type_);                                                \
    (h)->flags = UV_HANDLE_REF;  /* Ref the loop when active. */        \
    QUEUE_INSERT_TAIL(&(loop_)->handle_queue, &(h)->handle_queue);      \
    uv__handle_platform_init(h);                                        \
  }                                                                     \
  while (0)

#if defined(_WIN32)
# define uv__handle_platform_init(h) ((h)->u.fd = -1)
#else
```

```
# define uv__handle_platform_init(h) ((h)->next_closing = NULL)
#endif
```

`uv__handle_init` 是一个使用宏定义的宏函数，`do{}while(0)` 是一种巧妙用法，形成代码块，且调用时后边必须加分号，会被编译器优化掉。

这段代码主要完成以下几个工作：

1. 关联 `loop` 到 `handle`，可以通过 `handle` 找到对应的 `loop`；
2. 设置 `handle` 类型；
3. 设置 `handle` 标识为 `UV_HANDLE_REF`，这个标识位决定了 `handle` 是否计入引用计数。后续 Start Stop 会看到其用途；
4. 将 `handle` 插入 `loop->handle_queue` 队列的尾部，所有初始化的 `handle` 就将被插入到这个队列中；
5. 通过 `uv__handle_platform_init` 平台特定初始化函数将 `handle` 的 `next_closing` 设置为 `NULL`，这是一个连接了所有关闭的 `handle` 的单链表。

如下是 `uv_timer_t` 的初始化函数 `uv_timer_init`，它直接引用了 `uv__handle_init` 初始化 `uv_handle_t`，其他派生类型也是如此。

https://github.com/libuv/libuv/blob/v1.28.0/src/timer.c#L62

```
int uv_timer_init(uv_loop_t* loop, uv_timer_t* handle) {
  uv__handle_init(loop, (uv_handle_t*)handle, UV_TIMER);
  handle->timer_cb = NULL;
  handle->repeat = 0;
  return 0;
}
```

这样初始化工作就完成了，各个派生结构特定的初始化部分可能很简单，也可能很复杂。

Start：`uv__handle_start`

`uv_handle_t` 的启动代码是用宏 `uv__handle_start` 定义的宏函数，实现如下：

https://github.com/libuv/libuv/blob/v1.28.0/src/uv-common.h#L239

```
#define uv__handle_start(h)                                               \
  do {                                                                    \
    if (((h)->flags & UV_HANDLE_ACTIVE) != 0) break;                      \
    (h)->flags |= UV_HANDLE_ACTIVE;                                       \
    if (((h)->flags & UV_HANDLE_REF) != 0) uv__active_handle_add(h);      \
  }                                                                       \
  while (0)
```

`uv__handle_start` 将 handle 设置为 `UV_HANDLE_ACTIVE` 状态，并通过 `uv__active_handle_add` 更新活动的 `handle` 引用计数。如果不存在 `UV_HANDLE_REF` 标志位，则不会增加引用计数。

虽然对 handle 进行了 `Start` 操作，但是实际仅仅是设置了个标志位和增加了一个引用计数而已，看不到任何的 `Start`，实际上是告诉 libuv 该 `handle` 准备好了，可以 `Go` 了。因为更新引用计数间接影响了事件循环的活动状态。

`uv_run` 才是真正的启动操作，向 libuv 表明 `Ready` 了之后，`uv_run` 的时候才会处理这个 `handle`。

Stop：`uv__handle_stop`

handle 的 Stop： 操作 由 `uv__handle_stop` 宏实现：

```
#define uv__handle_stop(h)                                                \
  do {                                                                    \
```

```
    if (((h)->flags & UV_HANDLE_ACTIVE) == 0) break;                      \
    (h)->flags &= ~UV_HANDLE_ACTIVE;                                      \
    if (((h)->flags & UV_HANDLE_REF) != 0) uv__active_handle_rm(h);       \
  }                                                                       \
  while (0)
```

`uv__handle_stop` 将 handle 设置为 `~UV_HANDLE_ACTIVE` 状态，并通过 `uv__active_handle_rm` 更新活动的 `handle` 引用计数。如果不存在 `UV_HANDLE_REF` 标志位，则不会减少引用计数。

`Stop` 是 `Start` 的反向操作，将 handle 修改为`非准备`状态。

Close：`uv_close`

对于 handle 来说，还有一个 Close 方法 `uv_close`，`Close` 可以认为是 `Init` 的反向操作，它将 `handle` 从 `loop->handle_queue` 移除，清理资源并触发回调。

不同于上面三个方法，`uv_close` 是对外开放的，适用于所有类型 `handle` 的方法，在 `uv_close` 内部根据不同的类型，调用对应的函数处理。

https://github.com/libuv/libuv/blob/v1.28.0/src/unix/core.c#L107

```
void uv_close(uv_handle_t* handle, uv_close_cb close_cb) {
  assert(!uv__is_closing(handle));

  handle->flags |= UV_HANDLE_CLOSING;
  handle->close_cb = close_cb;

  switch (handle->type) {
    // ...
  }
```

```
  uv__make_close_pending(handle);
}
```

https://github.com/libuv/libuv/blob/v1.28.0/src/unix/core.c#L209

```
void uv__make_close_pending(uv_handle_t* handle) {
  assert(handle->flags & UV_HANDLE_CLOSING);
  assert(!(handle->flags & UV_HANDLE_CLOSED));
  handle->next_closing = handle->loop->closing_handles;
  handle->loop->closing_handles = handle;
}
```

在 `loop` 上有一个 `closing_handles` 字段，这是一个单向链表，关联了处于关闭进行中的 handle，这个字段的类型是 `uv_handle_t*`，指向了 `uv_handle_t`，而 `uv_handle_s` 存在了一个 `uv_handle_t*` 类型的指针 `next_closing` 指向下一个 `handle`, 这样就形成一个单向链表。

如下 `closing_handles` 的声明和初始化：

```
#define UV_LOOP_PRIVATE_FIELDS                                       \
  uv_handle_t* closing_handles;                                      \
```

```
#define UV_HANDLE_PRIVATE_FIELDS                                     \
  uv_handle_t* next_closing;                                         \
```

```
int uv_loop_init(uv_loop_t* loop) {
  // ...
  loop->closing_handles = NULL;
```

```
  // ...
}
```

`uv_close` 通过调用 `uv__make_close_pending` 将待关闭的 `handle` 放到 `loop->closing_handles` 链表末尾，`panding` 的含义是延迟到下次事件循环处理。

在 `uv_run` 的 `Call close callbacks` 阶段，通过函数 `uv__run_closing_handles` 专门负责处理 `loop->closing_handles`：

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) {
  while (r != 0 && loop->stop_flag == 0) {
    uv__run_closing_handles(loop);
  }
}
```

https://github.com/libuv/libuv/blob/v1.28.0/src/unix/core.c#L286

```
static void uv__run_closing_handles(uv_loop_t* loop) {
  uv_handle_t* p;
  uv_handle_t* q;

  p = loop->closing_handles;
  loop->closing_handles = NULL;

  while (p) {
    q = p->next_closing;
    uv__finish_close(p);
    p = q;
```

```
  }
}
```

https://github.com/libuv/libuv/blob/v1.28.0/src/unix/core.c#L236

```
static void uv__finish_close(uv_handle_t* handle) {
  assert(handle->flags & UV_HANDLE_CLOSING);
  assert(!(handle->flags & UV_HANDLE_CLOSED));
  handle->flags |= UV_HANDLE_CLOSED;

  switch (handle->type) {
    //
  }

  uv__handle_unref(handle);
  QUEUE_REMOVE(&handle->handle_queue);

  if (handle->close_cb) {
    handle->close_cb(handle);
  }
}
```

首先将 `closing_handles` 从 `loop` 摘除，然后遍历 `closing_handles`，通过 `uv__finish_close` 对每个 `handle` 进行最后的 `close`，`handle` 被移除 `loop->handle_queue` 并调用其关联的 `close_cb`，至此 `handle` 彻底没有了和 `loop` 的关联走完了一个完整的生命周期。

`uv_close` 的处理过程被拆分成了两段，一段是调用 `uv__make_close_pending`，另一段是在事件循环中调用 `uv__run_closing_handles`，关闭的过程是异步的，用户程序无法仅仅是通过 `uv_close` 返回判断关闭是否完成，需要在 `close_cb` 中接收异步操作结果。那么问题来了，为什么要拆分成两段而不是一次性处理完呢？

`uv_close` 一般来说都是在异步回调中被调用的，因为一个 `handle` 的关闭在逻辑上依赖于 `handle` 完成相关工作，而异步的逻辑中，完成工作后会调用相应的回调，所以只有在回调中调用 `uv_close` 才能使逻辑上是同步。

`Close` 阶段可以看做是 `Init` 阶段的反向操作。

`handle` 就这样伴随着事件循环经历了 `Init` -> `Start` -> `Stop` -> `Close` 等生命周期。

Reference counting：Ref & Unref

上文已经遇到过，`handle` 有个 `UV_HANDLE_REF` 标志位，这个状态用于控制 `handle` 是否计入 `loop->active_handles` 引用计数，因为 `handle` 的引用计数影响 `loop` 活动状态，所以 `UV_HANDLE_REF` 状态会间接影响 `loop` 的状态。

接下来，我们看下引用计数相关API：

https://github.com/libuv/libuv/blob/view-v1.28.0/src/uv-common.c#L502

```
void uv_ref(uv_handle_t* handle) {
  uv__handle_ref(handle);
}


void uv_unref(uv_handle_t* handle) {
  uv__handle_unref(handle);
}


int uv_has_ref(const uv_handle_t* handle) {
  return uv__has_ref(handle);
}
```

https://github.com/libuv/libuv/blob/v1.28.0/src/uv-common.h#L255

```
#define uv__handle_ref(h)                                                   \
  do {                                                                      \
    if (((h)->flags & UV_HANDLE_REF) != 0) break;                           \
    (h)->flags |= UV_HANDLE_REF;                                            \
    if (((h)->flags & UV_HANDLE_CLOSING) != 0) break;                       \
    if (((h)->flags & UV_HANDLE_ACTIVE) != 0) uv__active_handle_add(h);     \
  }                                                                         \
  while (0)

#define uv__handle_unref(h)                                                 \
  do {                                                                      \
    if (((h)->flags & UV_HANDLE_REF) == 0) break;                           \
    (h)->flags &= ~UV_HANDLE_REF;                                           \
    if (((h)->flags & UV_HANDLE_CLOSING) != 0) break;                       \
    if (((h)->flags & UV_HANDLE_ACTIVE) != 0) uv__active_handle_rm(h);      \
  }                                                                         \
  while (0)

#define uv__has_ref(h)                                                      \
  (((h)->flags & UV_HANDLE_REF) != 0)
```

https://github.com/libuv/libuv/blob/v1.28.0/src/uv-common.h#L221

```
#define uv__active_handle_add(h)                                            \
  do {                                                                      \
    (h)->loop->active_handles++;                                            \
  }                                                                         \
  while (0)

#define uv__active_handle_rm(h)                                             \
  do {                                                                      \
```

```
    (h)->loop->active_handles--;                                          \
  }                                                                       \
  while (0)
```

存在引用计数标志 `UV_HANDLE_REF`，会计入引用计数，否则不会引用计数。

实现非常简单，在条件满足的情况下，更新 `loop->active_handles` 值。

在事件循环初始化函数 `uv_loop_init` 中，`loop->child_watcher`、`loop->wq_async` 都被 `Unref` 了，避免影响 `loop` 的存活状态。

Status：Active & Closing

https://github.com/libuv/libuv/blob/v1.28.0/src/unix/core.c#L400

```
int uv_is_active(const uv_handle_t* handle) {
  return uv__is_active(handle);
}
```

https://github.com/libuv/libuv/blob/v1.28.0/src/unix/core.c#L301

```
int uv_is_closing(const uv_handle_t* handle) {
  return uv__is_closing(handle);
}
```

https://github.com/libuv/libuv/blob/v1.28.0/src/uv-common.h#L233

```
#define uv__is_active(h)                                                      \
  (((h)->flags & UV_HANDLE_ACTIVE) != 0)

#define uv__is_closing(h)                                                     \
  (((h)->flags & (UV_HANDLE_CLOSING | UV_HANDLE_CLOSED)) != 0)
```

实现简单，无需解释。

handle 和 loop 的关联关系

`handle` 和 `loop` 之间的关联是最为重要的，`handle` 必须注册到 `loop` 中的各种结构中才有意义，脱离 `loop` 的 `handle` 是毫无用途的，只有关联到 `loop` 上的 `handle` 才能在事件循环的过程中被处理。以上 `handle` 生命周期的核心就是在管理这种关系。除了以上基本的关联之外，`handle` 和 `loop` 还有其他关联。

在 `Init` 和 `Close` 操作中，`handle` 被插入/移除 `loop->handle_queue` 队列，`uv__active_handle_add`、`uv__active_handle_rm` 这两个宏函数修改 `handle` 的引用计数，进而间接修改了 `loop` 的状态。

除 `loop->handle_queue` 外，`loop` 中还有多个 `handle` 有关的队列，`handle` 除了被插入 `loop->handle_queue` 队列外，还会被插入到类型特定的结构中（如：队列、链表、堆），在 `uv_run` 的各个阶段，libuv 依赖这些结构完成工作中，下面将逐个来介绍一下都有哪些关联以及都是什么用途。

在 `uv_loop_t` 中，有多个相关 `handle` 队列：

uv_idle_t uv_check_t uv_check_t

https://github.com/libuv/libuv/blob/v1.28.0/include/uv/unix.h#L231

```
#define UV_LOOP_PRIVATE_FIELDS                                                \
  void* process_handles[2];                                                   \
```

```
  void* prepare_handles[2];                                     \
  void* check_handles[2];                                       \
  void* idle_handles[2];                                        \
```

- `uv_idle_t` 还会被插入到 `loop->idle_handles` 队列头部，队列节点为 `handle->queue`；
- `uv_check_t` 还会被插入到 `loop->check_handles` 队列头部，队列节点为 `handle->queue`；
- `uv_check_t` 还会被插入到 `loop->prepare_handles` 队列头部，队列节点为 `handle->queue`。

队列初始化：

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

```
int uv_loop_init(uv_loop_t* loop) {
  // ...
  QUEUE_INIT(&loop->idle_handles);
  QUEUE_INIT(&loop->async_handles);
  QUEUE_INIT(&loop->check_handles);
  QUEUE_INIT(&loop->prepare_handles);
  // ...
}
```

队列插入节点：

https://github.com/libuv/libuv/blob/v1.28.0/src/unix/loop-watcher.c#L24

```
#define UV_LOOP_WATCHER_DEFINE(name, type)                      \
  ...
  int uv_##name##_start(uv_##name##_t* handle, uv_##name##_cb cb) {  \
    ...
    QUEUE_INSERT_HEAD(&handle->loop->name##_handles, &handle->queue);  \
    ...
```

```
  }                                                                     \
  ...
```

uv_async_t

```
#define UV_LOOP_PRIVATE_FIELDS                                          \
  void* async_handles[2];                                               \
```

- `uv_async_t` 还会被插入到 `loop->async_handles` 队列尾部，队列节点为 `handle->queue`

队列初始化：

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

```
int uv_loop_init(uv_loop_t* loop) {
  // ...
  QUEUE_INIT(&loop->async_handles);
  // ...
}
```

队列插入节点：

https://github.com/libuv/libuv/blob/v1.x/src/unix/async.c#L40

```
int uv_async_init(uv_loop_t* loop, uv_async_t* handle, uv_async_cb async_cb) {
  QUEUE_INSERT_TAIL(&loop->async_handles, &handle->queue);
}
```

uv_process_t

```
#define UV_LOOP_PRIVATE_FIELDS                                       \
  void* process_handles[2];                                          \
```

`uv_process_t` 还会被插入到 `loop->process_handles` 队列尾部，队列节点为 `handle->queue`。

队列初始化：

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

```
int uv_loop_init(uv_loop_t* loop) {
  // ...
  uv__handle_unref(&loop->child_watcher);
  loop->child_watcher.flags |= UV_HANDLE_INTERNAL;
  QUEUE_INIT(&loop->process_handles);
  // ...
}
```

队列插入节点：

https://github.com/libuv/libuv/blob/v1.28.0/src/unix/process.c#L410

```
int uv_spawn(uv_loop_t* loop,
             uv_process_t* process,
             const uv_process_options_t* options) {
  // ...
  /* Only activate this handle if exec() happened successfully */
  if (exec_errorno == 0) {
    QUEUE_INSERT_TAIL(&loop->process_handles, &process->queue);
    uv__handle_start(process);
```

```
  }
  // ...
}
```

uv_timer_t

```
#define UV_LOOP_PRIVATE_FIELDS                                       \
  struct {                                                           \
    void* min;                                                       \
    unsigned int nelts;                                              \
  } timer_heap;                                                      \
```

`uv_timer_t` 还会被插入到 `loop->timer_heap` 堆（最小堆）中，堆节点为 `handle->heap_node`。

以上这些针对于不同类型的`队列/链表/堆`结构，都是为了方便的找到并统一处理相同类型的 `handle`。除了以上这些类型的 `handle` 之外，在 `loop` 上并没有这种特定类型的直接入口，而是通过其他链间接访问到 `handle` 的。

uv__io_t

在 `loop` 中，存在一个 `watchers` 数组，这个数组的每一项都是一个指向 `uv__io_t` 结构的指针，`uv__io_t` 是一个I/O观察者被内嵌到多个IO相关的 `handle` 结构中，所以所有的内嵌I/O观察者的 `handle` 都通过 `watchers` 被关联到 `loop` 上了。`uv__io_t` 存在多个子类型，这些子类型都可以被放到 `watchers` 数组中。

另外，还存在两个I/O观察者队列：

- `watcher_queue`：所有的IO观察者都会被插入到这个队列中。
- `pending_queue`：所有的挂起IO观察者都会被插入到这个队列中。

`uv__io_t` 相关字段声明：

https://github.com/libuv/libuv/blob/v1.28.0/include/uv/unix.h#L218

```
#define UV_LOOP_PRIVATE_FIELDS                                       \
  void* pending_queue[2];                                            \
  void* watcher_queue[2];                                            \
  uv__io_t** watchers;                                               \
  unsigned int nwatchers;                                            \
  unsigned int nfds;                                                 \
```

uv__work

`uv__work` 是 libuv 中任务的抽象，任务有线程池处理，任务在发起 `uv_work_t` request 时创建。

在 `loop` 中，存在一个任务队列，这个队列中记录了所以经线程池处理完成的任务，队列入口为：`loop->wq`

https://github.com/libuv/libuv/blob/v1.28.0/include/uv/unix.h#L218

```
#define UV_LOOP_PRIVATE_FIELDS                                       \
  void* wq[2];                                                       \
```

队列初始化：

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

```
int uv_loop_init(uv_loop_t* loop) {
  // ...
  QUEUE_INIT(&loop->wq);
  // ...
}
```

任务被处理完成或者任务取消后，都会被插入到该队里中。

closing_handles

`Close` 中提到过 `closing_handles` 关联了所有正在关闭的 `handle`，这也是一个关联的入口。

`handle` 可能同时存在于 `loop->handle_queue` 队列、`loop->closing_handles` 链表 以及 其他某一个特定类型的 `handle` 队列中。

通过上面的描述，可以在大脑中勾勒出一幅数据结构图。

除了 通过 `loop` 可以找到每一个 Handle，每一个 Handle 也可以通过其 `loop` 字段找到其所在的 `loop`。

以上这些 `handle` 通过各种方式关联到 `loop` 上除了 `loop` 作为资源的统一入口需要管理注册记录所有资源外，还需要使 `事件循环` 在运行的时候，能够方便的高效的处理这些 `handle`。

Request

Requests 一般代表一个短生命周期的 `操作`，有些 Request 需要通过在 Handle 上执行，有些 Request 则可以直接执行。

在 libuv 中，有如下 Request：

```
/* 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;
```

`uv_getaddrinfo_t` `uv_getnameinfo_t` `uv_fs_t` `uv_work_t` 可以直接执行，不依赖于 `handle`，直接关联到 `loop` 上。

`uv_connect_t` `uv_write_t` `uv_udp_send_t` `uv_shutdown_t` 都是跟读写相关的 `request`，其操作依赖于 `uv_stream_t`，也就是这些操作作用于 `uv_stream_t`，这些 `request` 通过 `handle` 关联到 `loop` 上。

同 `handle` 相似，`request` 也有一个基础结构 `uv_req_s`，其他 `request` 都通过组合复用 `uv_req_s` 的字段。

`uv_req_s` 结构定义如下：

```
#define UV_REQ_FIELDS                                 \
  /* public */                                        \
  void* data;                                         \
  /* read-only */                                     \
  uv_req_type type;                                   \
  /* private */                                       \
  void* reserved[6];                                  \
  UV_REQ_PRIVATE_FIELDS                               \

/* Abstract base class of all requests. */
struct uv_req_s {
  UV_REQ_FIELDS
};
```

`request` 并不需要在用户代码中显示的初始化，初始过程在相关的实现代码中由核心处理，通用的初始化部分 由 `uv__req_init` 函数处理，如下代码实现：

https://github.com/libuv/libuv/blob/v1.28.0/src/uv-common.h#L310

```
#define uv__req_init(loop, req, typ)                                          \
  do {                                                                        \
    UV_REQ_INIT(req, typ);                                                    \
    uv__req_register(loop, req);                                              \
  }                                                                           \
  while (0)
```

```
# define UV_REQ_INIT(req, typ)                                                \
  do {                                                                        \
    (req)->type = (typ);                                                      \
  }                                                                           \
  while (0)
#endif
```

https://github.com/libuv/libuv/blob/v1.28.0/src/uv-common.h#L205

```
#define uv__req_register(loop, req)                                           \
  do {                                                                        \
    (loop)->active_reqs.count++;                                              \
  }                                                                           \
  while (0)

#define uv__req_unregister(loop, req)                                         \
  do {                                                                        \
    assert(uv__has_active_reqs(loop));                                        \
    (loop)->active_reqs.count--;                                              \
  }                                                                           \
  while (0)
```

`uv__req_init` 初始化了 `request` 的类型，并通过 `uv__req_register` 更新 `request` 的引用计数，也可以通过 `uv__req_unregister` 反注册。

`request` 的更多内容，将在后续的分析中结合相关功能继续介绍。

源文件地址：https://github.com/liuyanjie/knowledge/tree/master/node.js/libuv/3-libuv-handle-and-request.md