
Node.js 源码剖析

作者: gc
微信: theratliter

前言

我很喜欢 JS 这门语言，感觉它和 C 语言一样，在 C 语言里，很多东西都需要自己实现，让我们可以发挥无限的创造力和想象力。在 JS 中，虽然很多东西在 V8 里已经提供，但是用 JS，依然可以创造很多好玩的东西，还有好玩的写法。另外，JS 应该我见过唯一的一门没有实现网络和文件功能的语言，网络和文件，是一个很重要的能力，对于程序员来说，也是很核心很基础的知识。很幸运，Node.js 被创造出来了，Node.js 在 JS 的基础上，使用 V8 和 Libuv 提供的能力，极大地拓展、丰富了 JS 的能力，尤其是网络和文件，这样我就不仅可以使用 JS，还可以使用网络、文件等功能，这是我逐渐转向 Node.js 方向的原因之一，也是我开始研究 Node.js 源码的原因之一。虽然 Node.js 满足了我喜好和技术上的需求，不过一开始的时候，我并没有全身心地投入代码的研究，只是偶尔会看一下某些模块的实现，真正的开始，是为了做《Node.js 是如何利用 Libuv 实现事件循环和异步》的分享，从那时候起，大部分业余时间和精力都投入源码的研究。

我首先从 Libuv 开始研究，因为 Libuv 是 Node.js 的核心之一。由于曾经研究过一些 Linux 的源码，也一直在学习操作系统的一些原理和实现，所以在阅读 Libuv 的时候，算是没有遇到太大的困难，C 语言函数的使用和原理，基本都可以看明白，重点在于需要把各个逻辑捋清楚。我使用的方法就是注释和画图，我个人比较喜欢写注释。虽然说代码是最好的注释，但是我还是愿意花时间用注释去把代码的背景和意义阐述一下，而且注释会让大部分人更快地能读懂代码的含义。读 Libuv 的时候，也穿插地读了一些 JS 和 C++ 层的代码。我阅读 Node.js 源码的方式是，选择一个模块，垂直地从 JS 层分析到 C++ 层，然后到 Libuv 层。

读完 Libuv，接下来读的是 JS 层的代码，JS 虽然容易看懂，但是 JS 层的代码非常多，而且我感觉逻辑上也非常绕，所以至今，我还有很多没有细读，这个作为后续的计划。Node.js 中，C++ 算是胶水层，很多时候，不会 C++，其实也不影响 Node.js 源码的阅读，因为 C++ 很多时候，只是一种透传的功能，它把 JS 层的请求，通过 V8，传给 Libuv，然后再反过来，所以 C++ 层我是放到最后才细读。C++ 层我觉得是最难的，这时候，我又不得不开始读 V8 的源码了，理解 V8 非常难，我选取的几乎是最早的版本 0.1.5，然后结合 8.x 版本。通过早期版本，先学习 V8 的大概原理和一些早期实现上的细节。因为后续的版本虽然变化很大，但是更多只是功能的增强和优化，有很多核心的概念

还是没有变化的，这是我选取早期版本的原因，避免一开始就陷入无穷无尽的代码中，迷失了方向，失去了动力。但是哪怕是早期的版本，有很多内容依然非常复杂，结合新版本是因为有些功能在早期版本里没有实现，这时候要明白它的原理，就只能看新版的代码，有了早期版本的经验，阅读新版的代码也有一定的好处，多少也知道了一些阅读技巧。

Node.js 的大部分代码都在 C++ 和 JS 层，所以目前仍然是在不断地阅读这两层的代码。还是按照模块垂直分析。阅读 Node.js 代码，让我更了解 Node.js 的原理，也更了解 JS。不过代码量非常大，需要源源不断的时间和精力投入。但是做技术，知其然知其所以然的感觉是非常美妙的，你靠着一门技术谋生，却对它知之甚少，这种感觉并不好。阅读源码，虽然不会为你带来直接的、迅速的收益，但是有几个好处是必然的。第一是它会决定你的高度，第二你写代码的时候，你看到的不再是一些冰冷冷、无生命的字符。这可能有点夸张，但是你了解了技术的原理，你在使用技术的时候，的确会有不同的体验，你的思维也会有了更多的变化。第三是提高了你的学习能力，当你对底层原理有了更多的了解和理解，你在学习其它技术的时候，就会更快地学会，比如你了解了 epoll 的原理，那你看 Nginx、Redis、Libuv 等源码的时候，关于事件驱动的逻辑，基本上很快就能看懂。很高兴有这些经历，也投入了很多时间和精力，希望以后对 Node.js 有更多的理解和了解，也希望在 Node.js 方向有更多的实践。

本书的目的

阅读 Node.js 源码的初衷是让自己深入理解 Node.js 的原理，但是我发现有很多同学对 Node.js 原理也非常感兴趣，因为业余时间里也一直在写一些关于 Node.js 源码分析的文章（基于 Node.js V10 和 V14），所以就打算把这些内容整理成一本有体系的书，让感兴趣的同学能系统地去了解和理解 Node.js 的原理。不过我更希望的是，读者从书中不仅学到 Node.js 的知识，而且也学到如何阅读 Node.js 源码，可以自己独立完成源码的研究。也希望更多同学分享自己的心得。本书不是 Node.js 的全部，但是尽量去讲得更多，源码非常多，错综复杂，理解上可能有不对之处，欢迎交流。因为看过 Linux 早期内核（0.11 和 1.2.13）和早期 V8（0.1.5）的一些实现，文章会引用其中的一些代码，目的在于让读者可以更了解一个知识点的大致实现原理，如果读者有兴趣，可以自行阅读相关代码。

本书结构

本书共分为二十二章，讲解的代码都是基于 Linux 系统的。

1. 主要介绍了 Node.js 的组成和整体的工作原理，另外分析了 Node.js 启动的过程，最后介绍了服务器架构的演变和 Node.js 的所选取的架构。
2. 主要介绍了 Node.js 中的基础数据结构和通用的逻辑，在后面的章节会用到。
3. 主要介绍了 Libuv 的事件循环，这是 Node.js 的核心所在，本章具体介绍了事件循环中每个阶段的实现。
4. 主要分析了 Libuv 中线程池的实现，Libuv 线程池对 Node.js 来说是非常重要的，Node.js 中很多模块都需要使用线程池，包括 crypto、fs、dns 等。如果没有线程池，Node.js 的功能将会大打折扣。同时分析了 Libuv 中子线程和主线程的通信机制。同样适合其它子线程和主线程通信。
5. 主要分析了 Libuv 中流的实现，流在 Node.js 源码中很多地方都用到，可以说是非常核心的概念。
6. 主要分析了 Node.js 中 C++ 层的一些重要模块和通用逻辑。
7. 主要分析了 Node.js 的信号处理机制，信号是进程间通信的另一种方式。
8. 主要分析了 Node.js 的 dns 模块的实现，包括 cares 的使用和原理。
9. 主要分析了 Node.js 中 pipe 模块（Unix 域）的实现和使用，Unix 域是实现进程间通信的方式，它解决了没有继承的进程无法通信的问题。而且支持传递文件描述符，极大地增强了 Node.js 的能力。
10. 主要分析了 Node.js 中定时器模块的实现。定时器是定时处理任务的利器。
11. 主要分析了 Node.js setImmediate 和 nextTick 的实现。
12. 主要介绍了 Node.js 中文件模块的实现，文件操作是我们经常会用到的功能。
13. 主要介绍了 Node.js 中进程模块的实现，多进程使得 Node.js 可以利用多核能力。
14. 主要介绍了 Node.js 中线程模块的实现，多进程和多线程有类似的功能但是也有一些差异。
15. 主要介绍了 Node.js 中 cluster 模块的使用和实现原理，cluster 模块封装了多进程能力，使得 Node.js 是可以使用多进程的服务器架构，利用了多核的能力。
16. 主要分析了 Node.js 中 UDP 的实现和相关内容。
17. 主要分析了 Node.js 中 TCP 模块的实现，TCP 是 Node.js 的核心模块，我们常用的 HTTP，HTTPS 都是基于 net 模块。
18. 主要介绍了 HTTP 模块的实现以及 HTTP 协议的一些原理。
19. 主要分析了 Node.js 中各种模块加载的原理，深入理解 Node.js 的 require 函数所做的事情。

- 20 主要介绍了一些拓展 Node.js 的方法，使用 Node.js，拓展 Node.js。
- 21 主要介绍了 JS 层 Stream 的实现，Stream 模块的逻辑很绕，大概讲解了一下。
- 22 主要介绍了 Node.js 中 event 模块的实现，event 模块虽然简单，但是是 Node.js 的核心模块。

面对的读者

本书面向有一定 Node.js 使用经验并对 Node.js 原理感兴趣的同学，因为本书是 Node.js 源码的角度去分析 Node.js 的原理，其中部分是 C、C++，所以需要读者有一定的 C、C++ 基础，另外，有一定的操作系统、计算机网络、V8 基础会更好。

阅读建议

建议首先阅读前面几种基础和通用的内容，然后再阅读单个模块的实现，最后有兴趣的话，再阅读如何拓展 Node.js 章节。如果你已经比较熟悉 Node.js，只是对某个模块或内容比较感兴趣，则可以直接阅读某个章节。刚开始阅读 Node.js 源码时，选取的是 V10.x 的版本，后来 Node.js 已经更新到了 V14，所以书中的代码有的是 V10 有的是 V14 的。

Libuv 是 V1.23。可以到我的 [github](#) 上获取。

源码阅读建议

Node.js 的源码由 JS、C++、C 组成。

1 Libuv 是 C 语言编写。理解 Libuv 除了需要了解 C 语法外，更多的是对操作系统和网络的理解，有些经典的书籍可以参考，比如《Unix 网络编程》1,2 两册，《Linux 系统编程手册》上下两册，《TCP/IP 权威指南》等等。还有 Linux 的 API 文档以及网上优秀的文章都可以参考一下。

2 C++主要是利用 V8 提供的能力对 JS 进行拓展，也有一部分功能使用 C++实现，总的来说 C++的作用更多是胶水层，利用 V8 作为桥梁，连接 Libuv 和 JS。不会 C++，也不完全影响源码的阅读，但是会 C++会更好。阅读 C++层代码，除了语法外，还需要对 V8 的概念和使用有一定的了解和理解。

3 JS 代码相信学习 Node.js 的同学都没什么问题。

其它资源

个人博客

csdn <https://blog.csdn.net/THEANARKH>

知乎 <https://www.zhihu.com/people/theanarkh>

github <https://github.com/theanarkh>

阅读 Node.js 源码时，所用到的基础知识、所作积累和记录几乎都在上面的博客中。如果你有任何问题可以到 <https://github.com/theanarkh/understand-nodejs> 提 issue 或者联系我。

目录

第一章 Node.js 组成和原理	13
1.1 Node.js 简介.....	13

1.1.1 JS 引擎 V8	14
1.1.2 Libuv	14
1.1.3 其它第三方库	20
1.2 Node.js 工作原理.....	21
1.2.1 Node.js 是如何拓展 JS 功能的?	21
1.2.2 如何在 V8 新增一个自定义的功能?	21
1.2.3 Node.js 是如何实现拓展的?.....	21
1.3 Node.js 启动过程.....	21
1.3.1 注册 C++模块.....	22
1.3.2 创建 Environment 对象	25
1.3.3 初始化 Libuv 任务	27
1.3.4 初始化 Loader 和执行上下文	28
1.3.5 执行用户 JS 文件	32
1.3.6 进入 Libuv 事件循环	33
1.4 Node.js 和其它服务器的比较.....	34
1.4.1 串行处理请求	34
1.4.2 多进程模式	35
1.4.3 多线程模式	37
1.4.4 事件驱动	38
第二章 Libuv 数据结构和通用逻辑	40
2.1 核心结构体 uv_loop_s.....	40
2.2 uv_handle_t	43
2.2.1 uv_stream_s	44
2.2.2 uv_async_s	45
2.2.3 uv_tcp_s	45
2.2.4 uv_udp_s	45
2.2.5 uv_tty_s	46
2.2.6 uv_pipe_s	46
2.2.7 uv_prepare_s、uv_check_s、uv_idle_s	46
2.2.8 uv_timer_s	46
2.2.9 uv_process_s	47
2.2.10 uv_fs_event_s	47
2.2.11 uv_fs_poll_s	47
2.2.12 uv_poll_s	48
2.1.13 uv_signal_s	48
2.3 uv_req_s	49
2.3.1 uv_shutdown_s	49
2.3.2 uv_write_s	50
2.3.3 uv_connect_s	50
2.3.4 uv_udp_send_s	50
2.3.5 uv_getaddrinfo_s	51
2.3.6 uv_getnameinfo_s	52

2.3.7 uv_work_s	52
2.3.8 uv_fs_s	53
2.4 I/O 观察者.....	54
2.4.1 初始化 I/O 观察者.....	55
2.4.2 注册一个 I/O 观察者到 Libuv。.....	55
2.4.3 撤销 I/O 观察者或者事件	56
2.5 Libuv 通用逻辑.....	57
2.5.1 uv_handle_init	57
2.5.2. uv_handle_start	57
2.5.3. uv_handle_stop	57
2.5.4. uv_req_init	58
2.5.5. uv_req_register	58
2.5.6. uv_req_unregister	58
2.5.7. uv_handle_ref	59
第三章 事件循环.....	59
3.1 事件循环之定时器	61
3.2 pending 阶段.....	62
3.3 事件循环之 prepare, check, idle.....	64
3.4 事件循环之 Poll I/O.....	66
3.5 事件循环之 close.....	69
3.6 控制事件循环	71
第四章 线程池.....	72
4.1 主线程和子线程间通信.....	72
4.1.1 初始化	73
4.1.2 通知主线程	75
4.1.3 主线程处理回调	77
4.2 线程池的实现	78
4.2.1 线程池的初始化	78
4.2.2 提交任务到线程池	79
4.2.3 处理任务	83
4.2.4 通知主线程	86
4.2.5 取消任务	87
第五章 Libuv 流	89
5.1 初始化流	90
5.2 打开流	91
5.3 读流	92
5.4 写流	100
5.5 关闭流的写端	108
5.6 关闭流	109
5.7 连接流	110
5.8 监听流	116
5.9 销毁流	120

5.10 事件触发的处理	121
第六章 C++层.....	123
6.1 BaseObject	123
6.1.1 构造函数	123
6.1.2 获取封装的对象	124
6.1.3 从对象中获取保存的 BaseObject 对象	124
6.1.4 解包	124
6.2 AsyncWrap	125
6.3 HandleWrap	126
6.3.1 新建 handle 和初始化	127
6.3.2 判断和操作 handle 状态	128
6.3.3 关闭 handle.....	129
6.4 ReqWrap	130
6.4.1 ReqWrapBase	130
6.4.2 ReqWrap	131
6.5 JS 如何使用 C++	133
6.7 C++层调用 Libuv.....	142
6.8 流封装	149
6.8.1 StreamResource	151
6.8.2 StreamBase	154
6.8.3 LibuvStreamWrap	161
6.8.4 ConnectionWrap	168
6.8.5 StreamReq	171
6.8.6 ShutdownWrap	173
6.8.7 SimpleShutdownWrap	174
6.8.8 WriteWrap	174
6.8.9 SimpleWriteWrap	175
6.8.10 StreamListener	175
6.8.11 ReportWritesToJSSStreamListener	176
6.8.12 EmitToJSSStreamListener	178
第七章 信号处理.....	182
7.1 信号的概念和实现原理	182
7.2 Libuv 信号处理的设计思想.....	183
7.3 通信机制的实现	184
7.4 信号结构体的初始化	186
7.5 信号处理的注册	186
7.6 信号的处理	190
7.7 取消/关闭信号处理	195
7.8 信号在 Node.js 中的使用	196
第八章 DNS.....	199
8.1 通过域名找 IP.....	200
8.2 cares	204

8.2.1 cares 使用和原理.....	204
8.2.2 cares_wrap.cc 的通用逻辑.....	205
8.2.3 具体实现	211
第九章 Unix 域	219
9.1 Unix 域在 Libuv 中的使用.....	219
9.1.1 初始化	220
9.1.2 绑定 Unix 域路径	220
9.1.3 启动服务	221
9.1.4 发起连接	222
9.1.5 关闭 Unix 域	223
9.2 Unix 域在 Node.js 中的使用.....	224
9.2.1 Unix 域服务器.....	224
9.2.2 Unix 域客户端.....	227
第十章 定时器.....	230
10.1 Libuv 的实现.....	231
10.1.1 Libuv 中维护定时器的数据结构.....	231
10.1.2 比较函数	231
10.1.3 初始化定时器结构体	232
10.1.4 插入一个定时器	232
10.1.5 停止一个定时器	233
10.1.6 重新设置定时器	233
10.1.7 计算二叉堆中超时时间最小值	234
10.1.8 处理定时器	234
10.2 核心数据结构	235
10.2.1 TimersList	235
10.2.2 优先队列	236
10.3 设置定时器处理函数	236
10.4 设置定时器	236
10.5 处理定时器	240
10.6 ref 和 unref	244
第十一章 setImmediate 和 nextTick.....	246
11.1 setImmediate	246
11.1.1 设置处理 immediate 任务的函数.....	246
11.1.2 注册 check 阶段的回调	247
11.1.3 setImmediate 生成任务.....	248
11.1.4 处理 setImmediate 产生的任务	251
11.1.5 Node.js 的 setTimeout(fn, 0) 和 setImmediate 谁先执行的问题	253
11.2 nextTick	254
11.2.1 初始化 nextTick.....	254
11.2.2 nextTick 生产任务	255
11.2.3 处理 tick 任务	255
11.2.4 nextTick 的使用.....	258

第十二章 文件.....	259
12.4 同步 API.....	260
12.5 异步 API.....	264
12.6 文件监听	268
12.6.1 基于轮询的文件监听机制.....	268
12.6.2 基于 inotify 的文件监听机制.....	273
12.7 Promise 化 API	280
12.8 流式 API.....	288
12.8.1 可读文件流	289
12.8.2 可写文件流	294
第十三章 进程.....	300
13.2 Node.js 主进程.....	300
13.2.1 创建 process 对象	300
13.2.2 挂载 env 属性	301
13.2.3 挂载其它属性	304
3.3 创建子进程	307
13.3.1 异步创建进程	308
13.3.2 同步创建进程	315
13.4 进程间通信	319
13.4.1 创建通信通道	319
13.4.2 主进程处理通信通道	324
13.4.3 子进程处理通信通道	325
13.5 文件描述符传递	326
13.5.1 发送文件描述符	326
13.5.2 接收文件描述符	329
第十四章 多线程.....	331
14.1 使用多线程	332
14.2 线程间通信数据结构	333
14.2.1 Message	334
14.2.2 MessagePortData	334
14.2.3 MessagePort	336
14.2.4 MessageChannel	339
14.3 多线程的实现	340
14.4 线程间通信	352
第十五章 cluster.....	356
15.1 cluster 使用例子	357
15.2 主进程初始化	357
15.3 子进程初始化	360
15.4 http.createServer 的处理	360
15.5 共享模式	363
15.6 轮询模式	365
15.7 实现自己的 cluster 模块	370

15.7.1 轮询模式	370
15.7.2 共享模式	371
第十六章 UDP	374
16.1 在 C 语言中使用 UDP	375
16.1.1 服务器流程（伪代码）	375
16.1.2 客户端流程	375
16.1.3 发送数据	376
16.1.4 接收数据	377
16.2 UDP 模块在 Node.js 中的实现	377
16.2.1 服务器	377
16.2.2 客户端	382
16.2.3 发送数据	386
16.2.4 接收数据	392
16.2.5 多播	393
16.2.6 端口复用	411
第十七章 TCP	418
17.1 TCP 客户端	420
17.1.1 建立连接	420
17.1.2 读操作	427
17.1.3 写操作	431
17.1.4 关闭写操作	433
17.1.5 销毁	435
17.2 TCP 服务器	436
17.3 keepalive	445
17.4 allowHalfOpen	451
17.5 server close	454
第十八章 HTTP	456
18.1 HTTP 解析器	457
18.2 HTTP 客户端	470
18.3 HTTP 服务器	482
18.3.1 HTTP 管道化的原理和实现	489
18.3.2 HTTP Connect 方法的原理和实现	496
18.3.3 超时管理	502
18.4 Agent	505
18.4.1 key 的计算	506
18.4.2 创建一个 socket	507
18.4.3 删 除 socket	508
18.4.4 设置 socket keepalive	509
18.4.5 复用 socket	510
18.4.6 销毁 Agent	510
18.4.7 使用连接池	510
18.4.8 测试例子	514

第十九章 模块加载.....	515
19.1 加载用户模块	517
19.1.1 加载 JSON 模块	517
19.1.2 加载 JS 模块	518
19.1.3 加载 node 模块	520
19.2 加载原生 JS 模块	525
19.3 加载内置 C++模块.....	529
第二十章 拓展 Node.js	532
20.1 修改 Node. js 内核	532
20.1.1 新增一个内置 C++模块.....	533
20.1.2 修改 Node. js 内核	536
20.2 使用 N-API 编写 C++插件.....	541
第二十一章 JS Stream	543
21.1 流基类和流通用逻辑	543
21.1.1 处理数据事件.....	543
21.1.2 流关闭/结束处理.....	544
21.1.3 错误处理	545
21.1.4 清除注册的事件	545
21.1.5 流的阈值	546
21.1.6 销毁流	547
21.2 可读流	548
21.2.1 可读流从底层资源获取数据	551
21.2.2 用户从可读流获取数据	552
21.3 可写流	553
21.3.1 WritableState	555
21.3.2 Writable	557
21.3.3 数据写入	558
21.3.4 cork 和 uncork	571
21.3.5 流结束	572
21.4 双向流	575
21.4.1 销毁	576
第二十二章 events 模块.....	577
22.1 初始化	578
22.2 订阅事件	578
22.3 触发事件	580
22.4 取消订阅	582

第一章 Node.js 组成和原理

1.1 Node.js 简介

Node.js 是基于事件驱动的单进程单线程应用，单线程具体体现在 Node.js 在单个线程中维护了一系列任务，然后在事件循环中不断消费任务队列中的节点，又不断产生新的任务，在任务的产生和消费中不断驱动着 Node.js 的执行。从另外一个角度来说，Node.js 又可以说是多线程的，因为 Node.js 底层也维护了一个线程池，该线程池主要用于处理一些文件 I/O、DNS、CPU 计算等任务。

Node.js 主要由 V8、Libuv，还有一些其它的第三方模块组成（ares 异步 DNS 解析库、HTTP 解析器、HTTP2 解析器，压缩库、加解密库等）。Node.js 源码分为三层，分别是 JS、C++、C，Libuv 是使用 C 语言编写，C++ 层主要是通过 V8 为 JS 层提供和底层交互的能力，C++ 层也实现了部分功能，JS 层是面向用户的，为用户提供调用底层的接口。

1.1.1 JS 引擎 V8

Node.js 是基于 V8 的 JS 运行时，它利用 V8 提供的能力，极大地拓展了 JS 的能力。这种拓展不是为 JS 增加了新的语言特性，而是拓展了功能模块，比如在前端，我们可以使用 Date 这个函数，但是我们不能使用 TCP 这个函数，因为 JS 中并没有内置这个函数。而在 Node.js 中，我们可以使用 TCP，这就是 Node.js 做的事情，让用户可以使用 JS 中本来不存在的功能，比如文件、网络。Node.js 中最核心的部分是 Libuv 和 V8，V8 不仅负责执行 JS，还支持自定义的拓展，实现了 JS 调用 C++ 和 C++ 调用 JS 的能力。比如我们可以写一个 C++ 模块，然后在 JS 调用，Node.js 正是利用了这个能力，完成了功能的拓展。JS 层调用的所有 C、C++ 模块都是通过 V8 来完成的。

1.1.2 Libuv

Libuv 是 Node.js 底层的异步 I/O 库，但它提供的功能不仅仅是 I/O，还包括进程、线程、信号、定时器、进程间通信等，而且 Libuv 抹平了各个操作系统之间的差异。Libuv 提供的功能大概如下

- Full-featured event loop backed by epoll, kqueue, IOCP, event ports.
- Asynchronous TCP and UDP sockets
- Asynchronous DNS resolution
- Asynchronous file and file system operations
- File system events
- ANSI escape code controlled TTY
- IPC with socket sharing, using Unix domain sockets or named pipes (Windows)
- Child processes
- Thread pool
- Signal handling
- High resolution clock
- Threading and synchronization primitives

Libuv 的实现是一个经典的生产者-消费者模型。Libuv 在整个生命周期中，每一轮循环都会处理每个阶段（phase）维护的任务队列，然后逐个执行任务队列中节点的回调，在回调中，不断生产新的任务，从而不断驱动 Libuv。图 1-1 是 Libuv 的整体执行流程

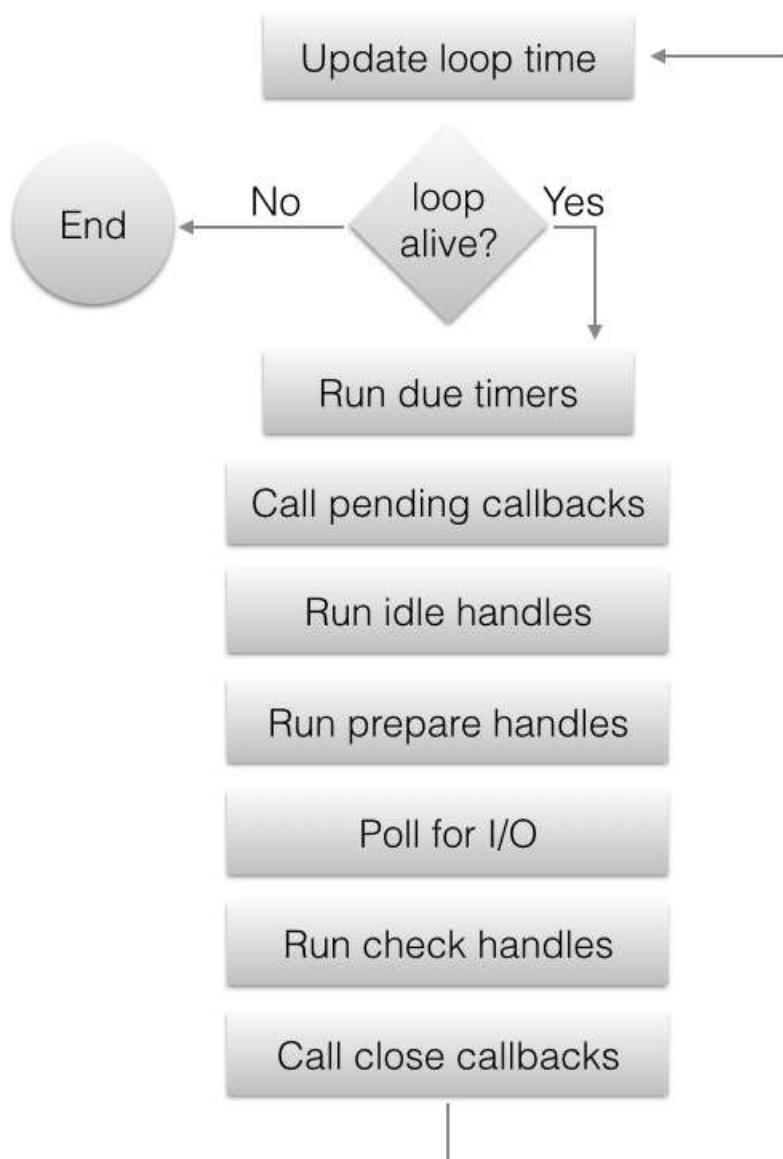


图 1-1

从图 1-1 中我们大致了解到，Libuv 分为几个阶段，然后在一个循环里不断执行每个阶段里的任务。下面我们具体看一下每个阶段

1 更新当前时间，在每次事件循环开始的时候，Libuv 会更新当前时间到变量中，这一轮循环的剩下操作可以使用这个变量获取当前时间，避免过多的系统调用影响性能，额外的影响就是时间不是那么精确。但是在一轮事件循环中，Libuv 在必要的时候，会主动更新这个时间，比如在 epoll 中阻塞了 timeout 时间后返回时，会再次更新当前时间变量。

2 如果事件循环是处于 alive 状态，则开始处理事件循环的每个阶段，否则退出这个事件循环。alive 状态是什么意思呢？如果有 active 和 ref 状态的 handle，active 状态的 request 或者 closing 状态的 handle 则认为事件循环是 alive（具体的后续会讲到）。

3 timer 阶段：判断最小堆中的节点哪个节点超时了，执行它的回调。

4 pending 阶段：执行 pending 回调。一般来说，所有的 IO 回调（网络，文件，DNS）都会在 Poll IO 阶段执行，但是有的情况下，Poll IO 阶段的回调会延迟到下一次循环执行，那么这种回调就是在 pending 阶段执行的，比如 IO 回调里出现了错误或写数据成功等等都会在下一个事件循环的 pending 阶段执行回调。

5 idle 阶段：每次事件循环都会被执行（idle 不是说事件循环空闲的时候才执行）。

6 prepare 阶段：和 idle 阶段类似。

7 Poll IO 阶段：调用各平台提供的 IO 多路复用接口（比如 Linux 下就是 epoll 模式），最多等待 timeout 时间，返回的时候，执行对应的回调。timeout 的计算规则：

如果时间循环是以 UV_RUN_NOWAIT 模式运行的，则 timeout 是 0。

如果时间循环即将退出（调用了 uv_stop），则 timeout 是 0。

如果没有 active 状态的 handle 或者 request，timeout 是 0。

如果有 idle 阶段的队列里有节点，则 timeout 是 0。

如果有 handle 等待被关闭的（即调了 uv_close），timeout 是 0。

如果上面的都不满足，则取 timer 阶段中最快超时的节点作为 timeout，

如果上面的都不满足则 timeout 等于 -1，即一直阻塞，直到满足条件。

8 check 阶段：和 idle、prepare 一样。

9 closing 阶段：执行调用 uv_close 函数时传入的回调。

10 如果 Libuv 是以 UV_RUN_ONCE 模式运行的，那事件循环即将退出。但是有一种情况是，Poll IO 阶段的 timeout 的值是 timer 阶段的节点的值，并且 Poll IO 阶段是因为超时返回的，即没有任何事件发生，也没有执行任何 IO 回调，这时候需要在执行一次 timer 阶段。因为有节点超时了。

11 一轮事件循环结束，如果 Libuv 以 UV_RUN_NOWAIT 或 UV_RUN_ONCE 模式运行的，则退出事件循环，如果是以 UV_RUN_DEFAULT 模式运行的并且状态是 alive，则开始下一轮循环。否则退出事件循环。

下面我能通过一个例子来了解 libuv 的基本原理。

```
1. #include <stdio.h>
2. #include <uv.h>
3.
4. int64_t counter = 0;
5.
6. void wait_for_a_while(uv_idle_t* handle) {
7.     counter++;
```

```

8.     if (counter >= 10e6)
9.         uv_idle_stop(handle);
10.    }
11.
12.   int main() {
13.       uv_idle_t idler;
14.       // 获取事件循环的核心结构体。并初始化一个 idle
15.       uv_idle_init(uv_default_loop(), &idler);
16.       // 往事件循环的 idle 阶段插入一个任务
17.       uv_idle_start(&idler, wait_for_a_while);
18.       // 启动事件循环
19.       uv_run(uv_default_loop(), UV_RUN_DEFAULT);
20.       // 销毁 libuv 的相关数据
21.       uv_loop_close(uv_default_loop());
22.       return 0;
23.   }

```

使用 Libuv，我们首先需要获取 Libuv 的核心结构体 `uv_loop_t`，`uv_loop_t` 是一个非常大的结构体，里面记录了 Libuv 整个生命周期的数据。`uv_default_loop` 为我们提供了一个默认已经初始化了的 `uv_loop_t` 结构体，当然我们也可以自己去分配一个，自己初始化。

```

1. uv_loop_t* uv_default_loop(void) {
2.     // 缓存
3.     if (default_loop_ptr != NULL)
4.         return default_loop_ptr;
5.
6.     if (uv_loop_init(&default_loop_struct))
7.         return NULL;
8.
9.     default_loop_ptr = &default_loop_struct;
10.    return default_loop_ptr;
11. }

```

Libuv 维护了一个全局的 `uv_loop_t` 结构体，使用 `uv_loop_init` 进行初始化，不打算展开讲解 `uv_loop_init` 函数，因为它大概就是对 `uv_loop_t` 结构体各个字段进行初始化。接着我们看一下 `uv_idle_*` 系列的函数。

1 `uv_idle_init`

```

1. int uv_idle_init(uv_loop_t* loop, uv_idle_t* handle) {
2.     /*
3.      初始化 handle 的类型，所属 loop，打上 UV_HANDLE_REF,
4.      并且把 handle 插入 loop->handle_queue 队列的队尾
5.     */
6.     uv__handle_init(loop, (uv_handle_t*)handle, UV_IDLE);

```

```
7.     handle->idle_cb = NULL;
8.     return 0;
9. }
```

执行 uv_idle_init 函数后，Libuv 的内存视图如图 1-2 所示

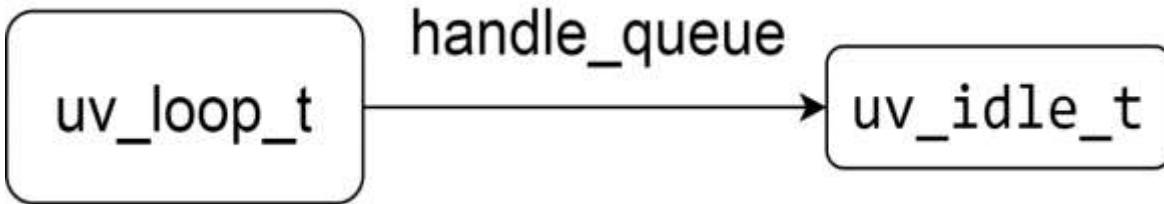


图 1-2

2 uv_idle_start

```
1. int uv_idle_start(uv_idle_t* handle, uv_idle_cb cb) {
2.     // 如果已经执行过 start 函数则直接返回
3.     if (uv__is_active(handle)) return 0;
4.     // 把 handle 插入 loop 中 idle 的队列
5.     QUEUE_INSERT_HEAD(&handle->loop->idle_handles, &handle->queue)
6.     ;
7.     // 挂载回调，下一轮循环的时候被执行
8.     handle->idle_cb = cb;
9.     /*
10.        设置 UV_HANDLE_ACTIVE 标记位，并且 loop 中的 handle 数加一，
11.        init 的时候只是把 handle 挂载到 loop，start 的时候 handle 才
12.        处于激活态
13.    */
14.    uv__handle_start(handle);
15.    return 0;
}
```

执行完 uv_idle_start 的内存视图如图 1-3 所示。

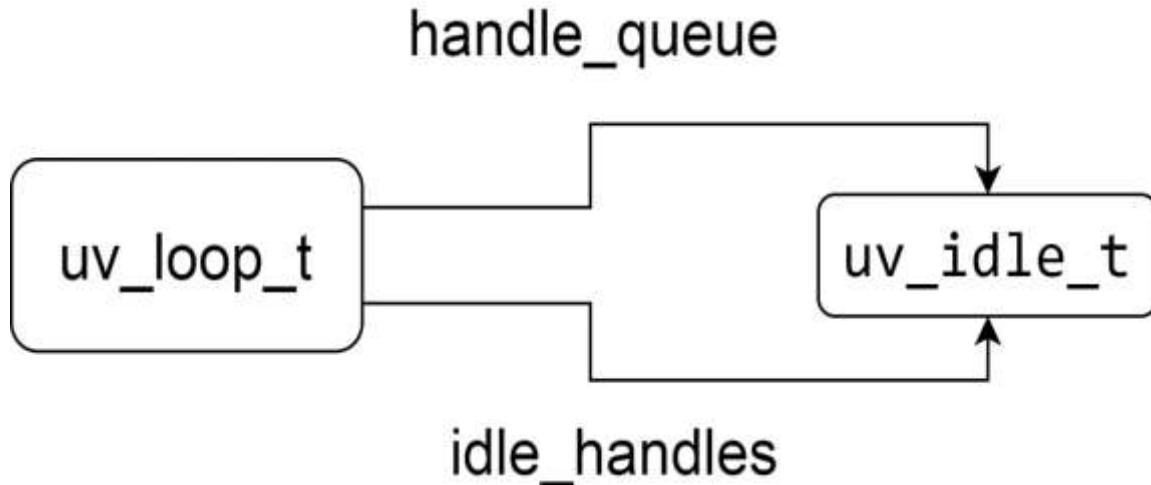


图 1-3

然后执行 uv_run 进入 Libuv 的事件循环。

```

1. int uv_run(uv_loop_t* loop, uv_run_mode mode) {
2.     int timeout;
3.     int r;
4.     int ran_pending;
5.     // 在 uv_run 之前要先提交任务到 loop
6.     r = uv_loop_alive(loop);
7.     // 没有任务需要处理或者调用了 uv_stop
8.     while (r != 0 && loop->stop_flag == 0) {
9.         // 处理 idle 队列
10.        uv_run_idle(loop);
11.    }
12.
13.    // 是因为调用了 uv_stop 退出的，重置 flag
14.    if (loop->stop_flag != 0)
15.        loop->stop_flag = 0;
16.    /*
17.     * 返回是否还有活跃的任务 (handle 或 request) ,
18.     * 业务代表可以再次执行 uv_run
19.    */
20.    return r;
21. }
```

我们看到有一个函数是 uv_run_idle，这就是处理 idle 阶段的函数。我们看一下它的实现。

```

1. // 在每一轮循环中执行该函数，执行时机见 uv_run
2. void uv_run_idle(uv_loop_t* loop) {
3.     uv_idle_t* h;
4.     QUEUE queue;
5.     QUEUE* q;
6.     /*
7.      * 把该类型对应的队列中所有节点摘下来挂载到 queue 变量,
8.      * 变量回调里不断插入新节点，导致死循环
9.     */
10.    QUEUE_MOVE(&loop->idle_handles, &queue);
11.    // 遍历队列，执行每个节点里面的函数
12.    while (!QUEUE_EMPTY(&queue)) {
13.        // 取下当前待处理的节点
14.        q = QUEUE_HEAD(&queue);
```

```
15.     // 取得该节点对应的整个结构体的基地址
16.     h = QUEUE_DATA(q, uv_idle_t, queue);
17.     // 把该节点移出当前队列, 否则循环不会结束
18.     QUEUE_REMOVE(q);
19.     // 重新插入原来的队列
20.     QUEUE_INSERT_TAIL(&loop->idle_handles, q);
21.     // 执行回调函数
22.     h->idle_cb(h);
23. }
24. }
```

我们看到 `uv_run_idle` 的逻辑并不复杂，就是遍历 `idle_handles` 队列的节点，然后执行回调，在回调里我们可以插入新的节点（产生新任务），从而不断驱动 Libuv 的运行。我们看到 `uv_run` 退出循环的条件下面的代码为 `false`。

```
1. r != 0 && loop->stop_flag == 0
```

`stop_flag` 由用户主动关闭 Libuv 事件循环。

```
1. void uv_stop(uv_loop_t* loop) {
2.   loop->stop_flag = 1;
3. }
```

`r` 是代表事件循环是否还存活，这个判断的标准是由 `uv_loop_alive` 提供

```
1. static int uv_loop_alive(const uv_loop_t* loop) {
2.   return loop->active_handles > 0 ||
3.         loop->active_reqs.count > 0 ||
4.         loop->closing_handles != NULL;
5. }
```

这时候我们有一个 `actived handles`，所以 Libuv 不会退出。当我们调用 `uv_idle_stop` 函数把 `idle` 节点移出 `handle` 队列的时候，Libuv 就会退出。后面我们会具体分析 Libuv 事件循环的原理。

1.1.3 其它第三方库

Node.js 中第三方库包括异步 DNS 解析（cares）、HTTP 解析器（旧版使用 `http_parser`，新版使用 `llhttp`）、HTTP2 解析器（`nghttp2`）、解压压缩库（`zlib`）、加密解密库（`openssl`）等等，不一一介绍。

1.2 Node.js 工作原理

1.2.1 Node.js 是如何拓展 JS 功能的？

V8 提供了一套机制，使得我们可以在 JS 层调用 C++、C 语言模块提供的功能。Node.js 正是通过这套机制，实现了对 JS 能力的拓展。Node.js 在底层做了大量的事情，实现了很多功能，然后在 JS 层暴露接口给用户使用，降低了用户成本，也提高了开发效率。

1.2.2 如何在 V8 新增一个自定义的功能？

```

1. // C++里定义
2. Handle<FunctionTemplate> Test = FunctionTemplate::New(cb);
3. global->Set(String::New("Test"), Test);
4. // JS 里使用
5. const test = new Test();

```

我们先有一个感性的认识，在后面的章节中，会具体讲解如何使用 V8 拓展 JS 的功能。

1.2.3 Node.js 是如何实现拓展的？

Node.js 并不是给每个功能都拓展一个对象，然后挂载到全局变量中，而是拓展一个 process 对象，再通过 process.binding 拓展 js 功能。Node.js 定义了一个全局的 JS 对象 process，映射到一个 C++ 对象 process，底层维护了一个 C++ 模块的链表，JS 通过调用 JS 层的 process.binding，访问到 C++ 的 process 对象，从而访问 C++ 模块（类似访问 JS 的 Object、Date 等）。不过 Node.js 14 版本已经改成 internalBinding 的方式，通过 internalBinding 就可以访问 C++ 模块，原理类似。

1.3 Node.js 启动过程

下面是 Node.js 启动的主流程图如图 1-4 所示。

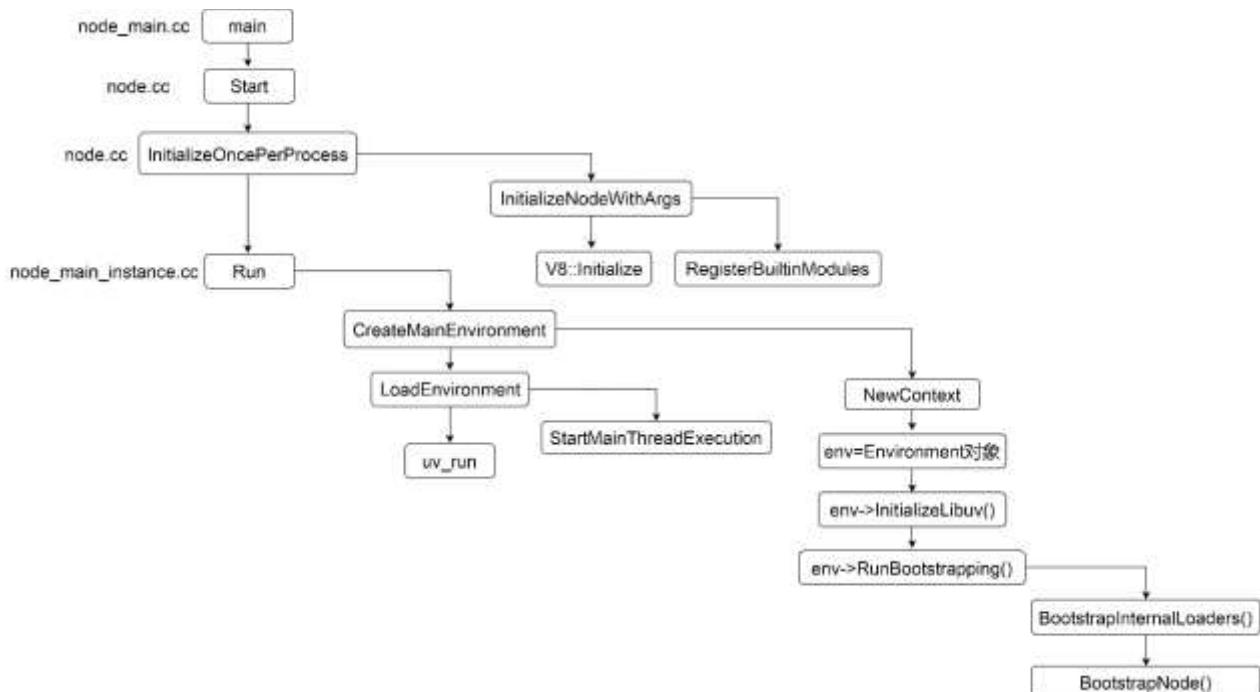


图 1-4

我们从上往下，看一下每个过程都做了些什么事情。

1.3.1 注册 C++模块

`RegisterBuiltinModules` 函数 (`node_binding.cc`) 的作用是注册 C++ 模块。

```
1. void RegisterBuiltinModules() {  
2. #define V(modname) _register_##modname();  
3. NODE_BUILTIN_MODULES(V)  
4. #undef V  
5. }
```

`NODE_BUILTIN_MODULES` 是一个 C 语言宏，宏展开后如下（省略类似逻辑）

```
1. void RegisterBuiltinModules() {  
2. #define V(modname) _register_##modname();  
3. V(tcp_wrap)  
4. V(timers)  
5. ... 其它模块  
6. #undef V  
7. }
```

再一步展开如下

```

1. void RegisterBuiltinModules() {
2.     _register_tcp_wrap();
3.     _register_timers();
4. }
```

执行了一系列_register 开头的函数，但是我们在 Node.js 源码里找不到这些函数，因为这些函数是在每个 C++ 模块定义的文件里 (.cc 文件的最后一行) 通过宏定义的。以 tcp_wrap 模块为例，看看它是怎么做的。文件 tcp_wrap.cc 的最后一句代码

NODE_MODULE_CONTEXT_AWARE_INTERNAL(tcp_wrap, node::TCPWrap::Initialize)
宏展开是

```

1. #define NODE_MODULE_CONTEXT_AWARE_INTERNAL(modname, regfunc) \
2. NODE_MODULE_CONTEXT_AWARE_CPP(modname, \
3.                                 regfunc, \
4.                                 nullptr, \
5.                                 NM_F_INTERNAL)
```

继续展开

```

6. #define NODE_MODULE_CONTEXT_AWARE_CPP(modname, regfunc, priv, flags\

7.     static node::node_module _module = { \
8.         NODE_MODULE_VERSION, \
9.         flags, \
10.        nullptr, \
11.        __FILE__, \
12.        nullptr, \
13.        (node::addon_context_register_func)(regfunc), \
14.        NODE_STRINGIFY(modname), \
15.        priv, \
16.        nullptr}; \
17. void _register_tcp_wrap() { node_module_register(&_module); }
```

我们看到每个 C++ 模块底层都定义了一个_register 开头的函数，在 Node.js 启动时，就会把这些函数逐个执行一遍。我们继续看一下这些函数都做了什么，在这之前，我们要先了解一下 Node.js 中表示 C++ 模块的数据结构。

```
1. struct node_module {  
2.     int nm_version;  
3.     unsigned int nm_flags;  
4.     void* nm_dso_handle;  
5.     const char* nm_filename;  
6.     node::addon_register_func nm_register_func;  
7.     node::addon_context_register_func nm_context_register_func;  
8.     const char* nm_modname;  
9.     void* nm_priv;  
10.    struct node_module* nm_link;  
11.};
```

我们看到_register 开头的函数调了 node_module_register，并传入一个 node_module 数据结构，所以我们看一下 node_module_register 的实现

```
1. void node_module_register(void* m) {  
2.     struct node_module* mp = reinterpret_cast<struct node_module*>(m);  
3.     if (mp->nm_flags & NM_F_INTERNAL) {  
4.         mp->nm_link = modlist_internal;  
5.         modlist_internal = mp;  
6.     } else if (!node_is_initialized) {  
7.         mp->nm_flags = NM_F_LINKED;  
8.         mp->nm_link = modlist_linked;  
9.         modlist_linked = mp;  
10.    } else {  
11.        thread_local_modpending = mp;  
12.    }  
13.}
```

C++ 内置模块的 flag 是 NM_F_INTERNAL，所以会执行第一个 if 的逻辑，modlist_internal 类似一个头指针。if 里的逻辑就是头插法建立一个单链表。C++ 内置模块在 Node.js 里是非常重要的，很多功能都会调用，后续我们会看到。

1.3.2 创建 Environment 对象

1 CreateMainEnvironment

Node.js 中 Environment 类 (env.h) 是一个很重要的类, Node.js 中, 很多数据由 Environment 对象进行管理。

```
1. context = NewContext(isolate_);
2. std::unique_ptr<Environment> env = std::make_unique<Environment>(
3.     isolate_data_.get(),
4.     context,
5.     args_,
6.     exec_args_,
7.     static_cast<Environment::Flags>(Environment::kIsMainThread |
8. Environment::kOwnsProcessState | Environment::kOwnsInspector));
```

Isolate, Context 是 V8 中的概念, Isolate 用于隔离实例间的环境, Context 用于提供 JS 执行时的上下文, kIsMainThread 说明当前运行的是主线程, 用于区分 Node.js 中的 worker_threads 子线程。Environment 类非常庞大, 我们看一下初始化的代码

```
1. Environment::Environment(IsolateData* isolate_data,
2.                             Local<Context> context,
3.                             const std::vector<std::string>& args,
4.                             const std::vector<std::string>& exec_args,
5.                             Flags flags,
6.                             uint64_t thread_id)
7. : isolate_(context->GetIsolate()),
8.   isolate_data_(isolate_data),
9.   immediate_info_(context->GetIsolate()),
10.  tick_info_(context->GetIsolate()),
11.  timer_base_(uv_now(isolate_data->event_loop())),
12.  exec_argv_(exec_args),
13.  argv_(args),
14.  exec_path_(GetExecPath(args)),
15.  should_abort_on_uncaught_toggle_(isolate_, 1),
16.  stream_base_state_(isolate_, StreamBase::kNumStreamBaseStateFields),
17.  flags_(flags),
18.  thread_id_(thread_id == kNoThreadId ? AllocateThreadId() : thread_id),
19.  fs_stats_field_array_(isolate_, kFsStatsBufferLength),
20.  fs_stats_field_bigint_array_(isolate_, kFsStatsBufferLength),
21.  context_(context->GetIsolate(), context) {
22.   // 进入当前的 context
23.   HandleScope handle_scope(isolate());
24.   Context::Scope context_scope(context);
25.   // 保存环境变量
26.   set_env_vars(per_process::system_environment);
27.   // 关联 context 和 env
28.   AssignToContext(context, ContextInfo(""));
29.   // 创建其它对象
30.   CreateProperties();
31. }
```

我们只看一下 AssignToContext 和 CreateProperties, set_env_vars 会把进程章节讲解。

1.1 AssignToContext

```
1.  inline void Environment::AssignToContext(v8::Local<v8::Context> context,
2.                                              const ContextInfo& info) {
3.      // 在 context 中保存 env 对象
4.      context->SetAlignedPointerInEmbedderData(ContextEmbedderIndex::kEnvironment, this);
5.      // Used by Environment::GetCurrent to know that we are on a node context.
6.      context->SetAlignedPointerInEmbedderData(ContextEmbedderIndex::kContextTag, Environment::kNodeContextTagPtr);
7.
8. }
```

`AssignToContext` 用于保存 `context` 和 `env` 的关系。这个逻辑非常重要，因为后续执行代码时，我们会进入 V8 的领域，这时候，我们只知道 `Isolate` 和 `context`。如果不保存 `context` 和 `env` 的关系，我们就不知道当前所属的 `env`。我们看一下如何获取对应的 `env`。

```
1.  inline Environment* Environment::GetCurrent(v8::Isolate* isolate) {
2.      v8::HandleScope handle_scope(isolate);
3.      return GetCurrent(isolate->GetCurrentContext());
4.  }
5.
6.  inline Environment* Environment::GetCurrent(v8::Local<v8::Context> context) {
7.      return static_cast<Environment*>(
8.          context->GetAlignedPointerFromEmbedderData(ContextEmbedderIndex::kEnvironment));
9.  }
```

1.2 CreateProperties

接着我们看一下 `CreateProperties` 中创建 `process` 对象的逻辑。

```
1.  Isolate* isolate = env->isolate();
2.  EscapableHandleScope scope(isolate);
3.  Local<Context> context = env->context();
4.  // 申请一个函数模板
5.  Local<FunctionTemplate> process_template = FunctionTemplate::New(isolate);
6.  process_template->SetClassName(env->process_string());
7.  // 保存函数模板生成的函数
8.  Local<Function> process_ctor;
9.  // 保存函数模块生成的函数所新建出来的对象
10. Local<Object> process;
11. if (!process_template->GetFunction(context).ToLocal(&process_ctor) ||
12.     !process_ctor->NewInstance(context).ToLocal(&process)) {
13.     return MaybeLocal<Object>();
```

`process` 所保存的对象就是我们在 JS 层用使用的 `process` 对象。`Node.js` 初始化的时候，还挂载了一些属性。

```
1.  READONLY_PROPERTY(process,
```

```

2.         "version",
3.             FIXED_ONE_BYTE_STRING(env->isolate(),
4.                                     NODE_VERSION));
5. READONLY_STRING_PROPERTY(process, "arch", per_process::metadata.arch);
.....
```

创建完 process 对象后，Node.js 把 process 保存到 env 中。

```

1. Local<Object> process_object = node::CreateProcessObject(this).FromMa
ybe(Local<Object>());
2. set_process_object(process_object)
```

1.3.3 初始化 Libuv 任务

InitializeLibuv 函数中的逻辑是往 Libuv 中提交任务。

```

1. void Environment::InitializeLibuv(bool start_profiler_idle_notifier)
{
2.     HandleScope handle_scope(isolate());
3.     Context::Scope context_scope(context());
4.     CHECK_EQ(0, uv_timer_init(event_loop(), timer_handle()));
5.     uv_unref(reinterpret_cast<uv_handle_t*>(timer_handle()));
6.     uv_check_init(event_loop(), immediate_check_handle());
7.     uv_unref(reinterpret_cast<uv_handle_t*>(immediate_check_handle()));
8.     uv_idle_init(event_loop(), immediate_idle_handle());
9.     uv_check_start(immediate_check_handle(), CheckImmediate);
10.    uv_prepare_init(event_loop(), &idle_prepare_handle_);
11.    uv_check_init(event_loop(), &idle_check_handle_);
12.    uv_async_init(
13.        event_loop(),
14.        &task_queues_async_,
15.        [] (uv_async_t* async) {
16.            Environment* env = ContainerOf(
17.                &Environment::task_queues_async_, async);
18.            env->CleanupFinalizationGroups();
19.            env->RunAndClearNativeImmediates();
20.        });
21.    uv_unref(reinterpret_cast<uv_handle_t*>(&idle_prepare_handle_));
22.    uv_unref(reinterpret_cast<uv_handle_t*>(&idle_check_handle_));
23.    uv_unref(reinterpret_cast<uv_handle_t*>(&task_queues_async_));
24.    // ...
25.}
```

这些函数都是 Libuv 提供的，分别是往 Libuv 不同阶段插入任务节点，`uv_unref` 是修改状态。

- 1 `timer_handle` 是实现 Node.js 中定时器的数据结构，对应 Libuv 的 time 阶段
- 2 `immediate_check_handle` 是实现 Node.js 中 `setImmediate` 的数据结构，对应 Libuv 的 check 阶段。
- 3 `task_queues_async_` 用于子线程和主线程通信。

1.3.4 初始化 Loader 和执行上下文

`RunBootstrapping` 里调用了 `BootstrapInternalLoaders` 和 `BootstrapNode` 函数，我们一个个分析。

1 初始化 loader

`BootstrapInternalLoaders` 用于执行 `internal/bootstrap/loaders.js`。我们看一下具体逻辑。首先定义一个变量，该变量是一个字符串数组，用于定义函数的形参列表，一会我们会看到它的作用。

```
1. std::vector<Local<String>> loaders_params = {  
2.     process_string(),  
3.     FIXED_ONE_BYTE_STRING(isolate_, "getLinkedBinding"),  
4.     FIXED_ONE_BYTE_STRING(isolate_, "getInternalBinding"),  
5.     primordials_string()};
```

然后再定义一个变量，是一个对象数组，用作执行函数时的实参。

```
1. std::vector<Local<Value>> loaders_args = {  
2.     process_object(),  
3.     NewFunctionTemplate(binding::GetLinkedBinding)  
4.         ->GetFunction(context())  
5.         .ToLocalChecked(),  
6.     NewFunctionTemplate(binding::GetInternalBinding)  
7.         ->GetFunction(context())  
8.         .ToLocalChecked(),  
9.     primordials()};
```

接着 Node.js 编译执行 `internal/bootstrap/loaders.js`，这个过程链路非常长，最后到 V8 层，就不贴出具体的代码，具体的逻辑转成 JS 如下。

```
1. function demo(process,  
2.                 getLinkedBinding,  
3.                 getInternalBinding,
```

```

4.           primordials)  {
5.     //  internal/bootstrap/loaders.js 的代码
6.   }
7. const process = {};
8. function getLinkedBinding() {}
9. function getInternalBinding() {}
10. const primordials = {};
11. const export = demo(process,
12.                       getLinkedBinding,
13.                       getInternalBinding,
14.                       primordials);

```

V8 把 internal/bootstrap/loaders.js 用一个函数包裹起来，形参就是 loaders_params 变量对应的四个字符串。然后执行这个函数，并且传入 loaders_args 里的那四个对象。internal/bootstrap/loaders.js 会导出一个对象。在看 internal/bootstrap/loaders.js 代码之前，我们先看一下 getLinkedBinding, getInternalBinding 这两个函数，Node.js 在 C++ 层对外暴露了 AddLinkedBinding 方法注册模块，Node.js 针对这种类型的模块，维护了一个单独的链表。getLinkedBinding 就是根据模块名从这个链表中找到对应的模块，但是我们一般用不到这个，所以就不深入分析。前面我们看到对于 C++ 内置模块，Node.js 同样维护了一个链表，getInternalBinding 就是根据模块名从这个链表中找到对应的模块。现在我们可以具体看一下 internal/bootstrap/loaders.js 的代码了。

```

1. let internalBinding;
2. {
3.   const bindingObj = ObjectCreate(null);
4.   internalBinding = function internalBinding(module) {
5.     let mod = bindingObj[module];
6.     if (typeof mod !== 'object') {
7.       mod = bindingObj[module] = getInternalBinding(module);
8.       moduleLoadList.push(`Internal Binding ${module}`);
9.     }
10.    return mod;
11.  };
12. }

```

Node.js 在 JS 对 getInternalBinding 进行了一个封装，主要是加了缓存处理。

```

1. const internalBindingWhitelist = new SafeSet([
2.   'tcp_wrap',
3.   // 一系列 C++ 内置模块名
4. ]);

```

```
5.  
6. {  
7.     const bindingObj = ObjectCreate(null);  
8.     process.binding = function binding(module) {  
9.         module = String(module);  
10.        if (internalBindingWhitelist.has(module)) {  
11.            return internalBinding(module);  
12.        }  
13.        throw new Error(`No such module: ${module}`);  
14.    };  
15.}
```

在 `process` 对象（就是我们平时使用的 `process` 对象）中挂载 `binding` 函数，这个函数主要用于内置的 JS 模块，后面我们会经常看到。`binding` 的逻辑就是根据模块名查找对应的 C++ 模块。上面的处理是为了 `Node.js` 能在 JS 层通过 `binding` 函数加载 C++ 模块，我们知道 `Node.js` 中还有原生的 JS 模块（`lib` 文件夹下的 JS 文件）。接下来我们看一下，对于加载原生 JS 模块的处理。`Node.js` 定义了一个 `NativeModule` 类负责原生 JS 模块的加载。还定义了一个变量保存了原生 JS 模块的名称列表。

```
static map = new Map(moduleIds.map((id) => [id, new NativeModule(id)]));
```

`NativeModule` 主要的逻辑如下

- 1 原生 JS 模块的代码是转成字符存在 `node_javascript.cc` 文件的，`NativeModule` 负责原生 JS 模块的加载，即编译和执行。
 - 2 提供一个 `require` 函数，加载原生 JS 模块，对于文件路径以 `internal` 开头的模块，是不能被用户 `require` 使用的。
- 这是原生 JS 模块加载的大概逻辑，具体的我们在 `Node.js` 模块加载章节具体分析。执行完 `internal/bootstrap/loaders.js`，最后返回三个变量给 C++ 层。

```
1. return {  
2.     internalBinding,  
3.     NativeModule,  
4.     require: nativeModuleRequire  
5.};
```

C++ 层保存其中两个函数，分别用于加载内置 C++ 模块和原生 JS 模块的函数。

```
1. set_internal_binding_loader(internal_binding_loader.As<Function>());  
2. set_native_module_require(require.As<Function>());
```

至此，`internal/bootstrap/loaders.js` 分析完了

2 初始化执行上下文

BootstrapNode 负责初始化执行上下文，代码如下

```

1. EscapableHandleScope scope(isolate_);
2. // 获取全局变量并设置 global 属性
3. Local<Object> global = context()->Global();
4. global->Set(context(), FIXED_ONE_BYTE_STRING(isolate_, "global"), global).Check();
5. /*
6.   执行 internal/bootstrap/node.js 时的参数
7.   process, require, internalBinding, primordials
8. */
9. std::vector<Local<String>> node_params = {
10.   process_string(),
11.   require_string(),
12.   internal_binding_string(),
13.   primordials_string()};
14. std::vector<Local<Value>> node_args = {
15.   process_object(),
16.   // 原生模块加载器
17.   native_module_require(),
18.   // C++模块加载器
19.   internal_binding_loader(),
20.   primordials()};
21.
22. MaybeLocal<Value> result = ExecuteBootstrapper(
23.   this, "internal/bootstrap/node", &node_params, &node_args);

```

在全局对象上设置一个 global 属性，这就是我们在 Node.js 中使用的 global 对象。接着执行 internal/bootstrap/node.js 设置一些变量（具体可以参考 internal/bootstrap/node.js）。

```

1. process.cpuUsage = wrapped.cpuUsage;
2. process.resourceUsage = wrapped.resourceUsage;
3. process.memoryUsage = wrapped.memoryUsage;
4. process.kill = wrapped.kill;
5. process.exit = wrapped.exit;

```

设置全局变量

```

1. defineOperation(global, 'clearInterval', timers.clearInterval);
2. defineOperation(global, 'clearTimeout', timers.clearTimeout);
3. defineOperation(global, 'setInterval', timers.setInterval);
4. defineOperation(global, 'setTimeout', timers.setTimeout);

```

```
5. Object.defineProperty(global, 'process', {
6.   value: process,
7.   enumerable: false,
8.   writable: true,
9.   configurable: true
10.});
```

1.3.5 执行用户 JS 文件

StartMainThreadExecution 进行一些初始化工作，然后执行用户 JS 代码。

1 给 process 对象挂载属性

执行 patchProcessObject 函数（在 node_process_methods.cc 中导出）给 process 对象挂载一些列属性，不一一列举。

```
1. // process.argv
2. process->Set(context,
3.                 FIXED_ONE_BYTE_STRING(isolate, "argv"),
4.                 ToV8Value(context, env->argv()).ToLocalChecked().C
5.                 heck());
6. READONLY_PROPERTY(process,
7.                     "pid",
8.                     Integer::New(isolate, uv_os_getpid()));
```

因为 Node.js 增加了对线程的支持，有些属性需要 hack 一下，比如在线程里使用 process.exit 的时候，退出的是单个线程，而不是整个进程，exit 等函数需要特殊处理。后面章节会详细讲解。

2 处理进程间通信

```
1. function setupChildProcessIpcChannel() {
2.   if (process.env.NODE_CHANNEL_FD) {
3.     const fd = parseInt(process.env.NODE_CHANNEL_FD, 10);
4.     delete process.env.NODE_CHANNEL_FD;
5.     const serializationMode =
6.       process.env.NODE_CHANNEL_SERIALIZATION_MODE || 'json';
7.     delete process.env.NODE_CHANNEL_SERIALIZATION_MODE;
8.     require('child_process')._forkChild(fd, serializationMode);
9.   }
10.}
```

环境变量 NODE_CHANNEL_FD 是在创建子进程的时候设置的，如果有说明当前启动的进程是子进程，则需要处理进程间通信。

3 处理 cluster 模块的进程间通信

```

1. function initializeclusterIPC() {
2.     if (process.argv[1] && process.env.NODE_UNIQUE_ID) {
3.         const cluster = require('cluster');
4.         cluster._setupWorker();
5.         delete process.env.NODE_UNIQUE_ID;
6.     }
7. }
```

4 执行用户 JS 代码

```
require('internal/modules/cjs/loader').Module.runMain(process.argv[1]);
```

internal/modules/cjs/loader.js 是负责加载用户 JS 的模块，runMain 函数在 pre_execution.js 被挂载，runMain 做的事情是加载用户的 JS，然后执行。具体的过程在后面章节详细分析。

1.3.6 进入 Libuv 事件循环

执行完所有的初始化后，Node.js 执行了用户的 JS 代码，用户的 JS 代码会往 Libuv 注册一些任务，比如创建一个服务器，最后 Node.js 进入 Libuv 的事件循环中，开始一轮又一轮的事件循环处理。如果没有需要处理的任务，Libuv 会退出，从而 Node.js 退出。

```

1. do {
2.     uv_run(env->event_loop(), UV_RUN_DEFAULT);
3.     per_process::v8_platform.DrainVMTasks(isolate_);
4.     more = uv_loop_alive(env->event_loop());
5.     if (more && !env->is_stopping()) continue;
6.     if (!uv_loop_alive(env->event_loop())) {
7.         EmitBeforeExit(env.get());
8.     }
9.     more = uv_loop_alive(env->event_loop());
10. } while (more == true && !env->is_stopping());
```

1.4 Node.js 和其它服务器的比较

服务器是现代软件中非常重要的一个组成，我们看一下服务器发展的过程中，都有哪些设计架构。一个基于 TCP 协议的服务器，基本的流程如下（伪代码）。

```
1. // 拿到一个 socket 用于监听
2. const socketfd = socket(协议类型等配置);
3. // 监听本机的地址 (ip+端口)
4. bind(socketfd, 监听地址)
5. // 标记该 socket 是监听型 socket
6. listen(socketfd)
```

执行完以上步骤，一个服务器正式开始服务。下面我们看一下基于上面的模型，分析各种各样的处理方法。

1.4.1 串行处理请求

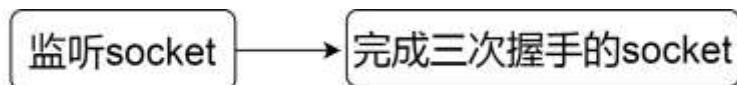
```
1. while(1) {
2.     const socketForCommunication = accept(socket);
3.     const data = read(socketForCommunication);
4.     handle(data);
5.     write(socketForCommunication, data );
6. }
```

我们看看这种模式的处理过程，假设有 n 个请求到来。那么 socket 的结构如图 1-5 所示。



图 1-5

这时候进程从 accept 中被唤醒。然后拿到一个新的 socket 用于通信。结构如图 1-6 所示。



socketForCommunication 完成三次握手的socket

图 1-6

accept 就是从已完成三次握手的连接队列里，摘下一个节点。很多同学都了解三次握手是什么，但是可能很少同学会深入思考或者看它的实现，众所周知，一个服务器启动的时候，会监听一个端口，其实这就是新建了一个 socket。那么如果有一个连接到来的时候，我们通过 accept 就能拿到这个新连接对应的 socket，那这个 socket 和监听的 socket 是不是同一个呢？其实 socket 分为监听型和通信型的，表面上，服务器用一个端口实现了多个连接，但是这个端口是用于监听的，底层用于和客户端通信的其实是另一个 socket。所以每一个连接过来，负责监听的 socket 发现是一个建立连接的包（syn 包），它就会生成一个新的 socket 与之通信（accept 的时候返回的那个）。监听 socket 里只保存了它监听的 IP 和端口，通信 socket 首先从监听 socket 中复制 IP 和端口，然后把客户端的 IP 和端口也记录下来，当下次收到一个数据包的时候，操作系统就会根据四元组从 socket 池子里找到该 socket，从而完成数据的处理。

串行这种模式就是从已完成三次握手的队列里摘下一个节点，然后处理。再摘下一个节点，再处理。如果处理的过程中有阻塞式 I/O，可想而知，效率是有多低。而且并发量比较大的时候，监听 socket 对应的队列很快就会被占满（已完成连接队列有一个最大长度）。这是最简单的模式，虽然服务器的设计中肯定不会使用这种模式，但是它让我们了解了一个服务器处理请求的整体过程。

1.4.2 多进程模式

串行模式中，所有请求都在一个进程中排队被处理，这是效率低下的原因。这时候我们可以把请求分给多个进程处理来提供效率，因为在串行处理的模式中，如果有阻塞式 I/O 操作，它就会阻塞主进程，从而阻塞后续请求的处理。在多进程的模式下，一个请求如果阻塞了进程，那么操作系统会挂起该进程，接着调度其它进程执行，那么其它进程就可以执行新的任务。多进程模式下分为几种。

1 主进程 accept，子进程处理请求

这种模式下，主进程负责摘取已完成连接的节点，然后把这个节点对应的请求交给子进程处理，逻辑如下。

```

1. while(1) {
2.     const socketForCommunication = accept(socket);
3.     if (fork() > 0) {
4.         continue;
5.         // 父进程
6.     } else {
7.         // 子进程
8.         handle(socketForCommunication);
9.     }
10. }
```

这种模式下，每次来一个请求，就会新建一个进程去处理。这种模式比串行的稍微好了一点，每个请求独立处理，假设 a 请求阻塞在文件 I/O，那么不会影响 b 请求的处理，尽可能地做到了并发。它的瓶颈就是系统的进程数有限，如果有大量的请求，系统无法扛得住，再者，进程的开销很大，对于系统来说是一个沉重的负担。

2 进程池模式

实时创建和销毁进程开销大，效率低，所以衍生了进程池模式，进程池模式就是服务器启动的时候，预先创建一定数量的进程，但是这些进程是 worker 进程。它不负责 accept 请求。它只负责处理请求。主进程负责 accept，它把 accept 返回的 socket 交给 worker 进程处理，模式如图 1-8 所示。

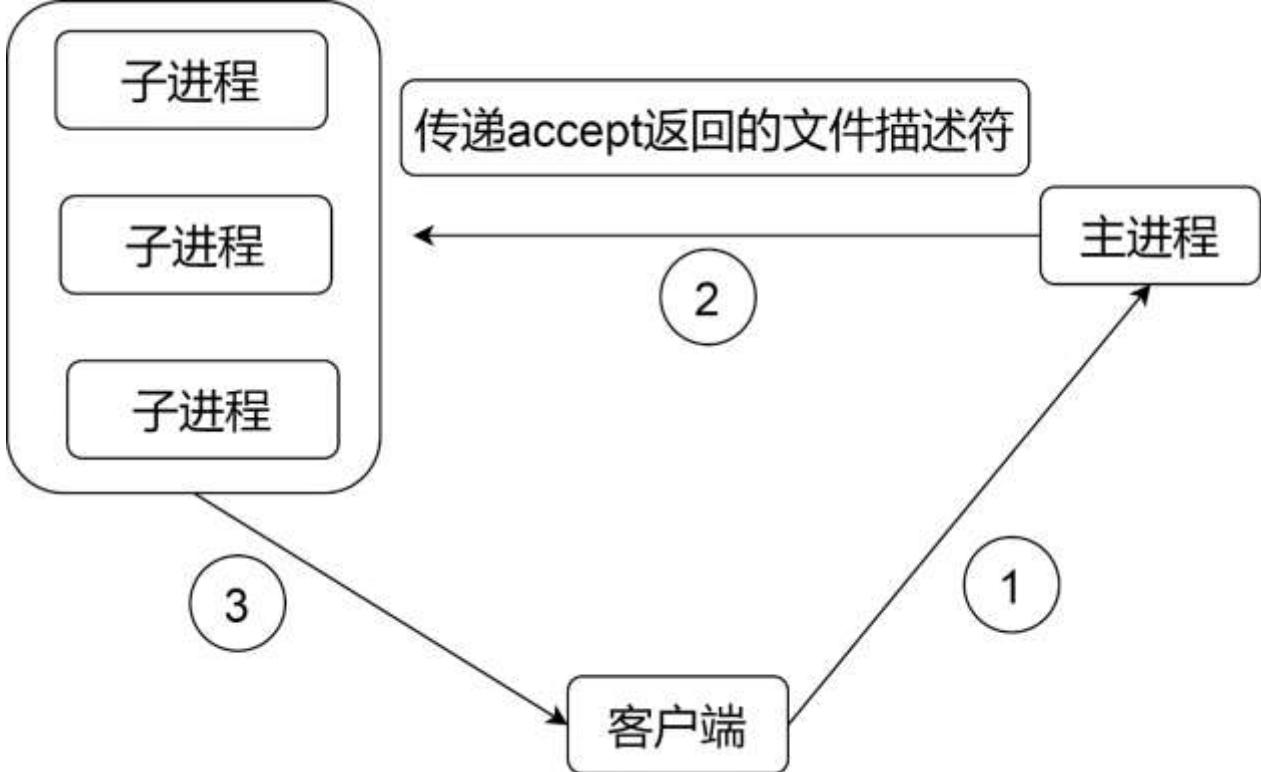


图 1-7

但是和 1 中的模式相比，进程池模式相对比较复杂，因为在模式 1 中，当主进程收到一个请求的时候，实时 fork 一个子进程，这时候，这个子进程会继承主进程中新请求对应的 fd，所以它可以直接处理该 fd 对应的请求，在进程池的模式中，子进程是预先创建的，当主进程收到一个请求的时候，子进程中是无法拿得到该请求对应的 fd 的。这时候，需要主进程使用传递文件描述符的技术把这个请求对应的 fd 传给子进程。一个进程其实就是一个结构体 task_struct，在 JS 里我们可以说是一个对象，它有一个字段记录了打开的文件描述符，当我们访问一个文件描述符的时候，操作系统就会根据 fd 的值，从 task_struct 中找到 fd 对应的底层资源，所以主进程给子进程传递文件描述符的时候，传递的不仅仅是一个数字 fd，因为如果仅仅这样做，在子进程中该 fd 可能没有对应任何资源，或者对应的资源和主进程中的是不一致的。这其中操作系统帮我们做了很多事情。让我们在子进程中可以通过 fd 访问到正确的资源，即主进程中收到的请求。

3 子进程 accept

这种模式不是等到请求来的时候再创建进程。而是在服务器启动的时候，就会创建多个进程。然后多个进程分别调用 accept。这种模式的架构如图 1-7 所示。



图 1-8

```

1. const sockfd = socket(协议类型等配置);
2. bind(sockfd, 监听地址)
3.
4. for (let i = 0 ; i < 进程个数; i++) {
5.     if (fork() > 0) {
6.         // 父进程负责监控子进程
7.     } else {
8.         // 子进程处理请求
9.         listen(sockfd);
10.        while(1) {
11.            const socketForCommunication = accept(sockfd);
12.            handle(socketForCommunication);
13.        }
14.    }
15. }
  
```

这种模式下多个子进程都阻塞在 accept。如果这时候有一个请求到来，那么所有的子进程都会被唤醒，但是首先被调度的子进程会首先摘下这个请求节点，后续的进程被唤醒后可能会遇到已经没有请求可以处理，又进入睡眠，进程被无效唤醒，这是著名的惊群现象。改进方式就是在 accpet 之前加锁，拿到锁的进程才能进行 accept，这样就保证了只有一个进程会阻塞在 accept，Nginx 解决了这个问题，但是新版操作系统已经在内核层面解决了这个问题。每次只会唤醒一个进程。通常这种模式和事件驱动配合使用。

1.4.3 多线程模式

多线程模式和多进程模式是类似的，也是分为下面几种

- 1 主进程 accept，创建子线程处理
- 2 子线程 accept
- 3 线程池

前面两种和多进程模式中是一样的，但是第三种比较特别，我们主要介绍第三种。在子进程模式时，每个子进程都有自己的 task_struct，这就意味着在 fork 之后，每个进程负责维护自己的数据，而线程则不一样，线程是共享主线程（主进程）的数据的，当主进程

从 accept 中拿到一个 fd 的时候，传给线程的话，线程是可以直接操作的。所以在线程池模式时，架构如图 1-10 所示。

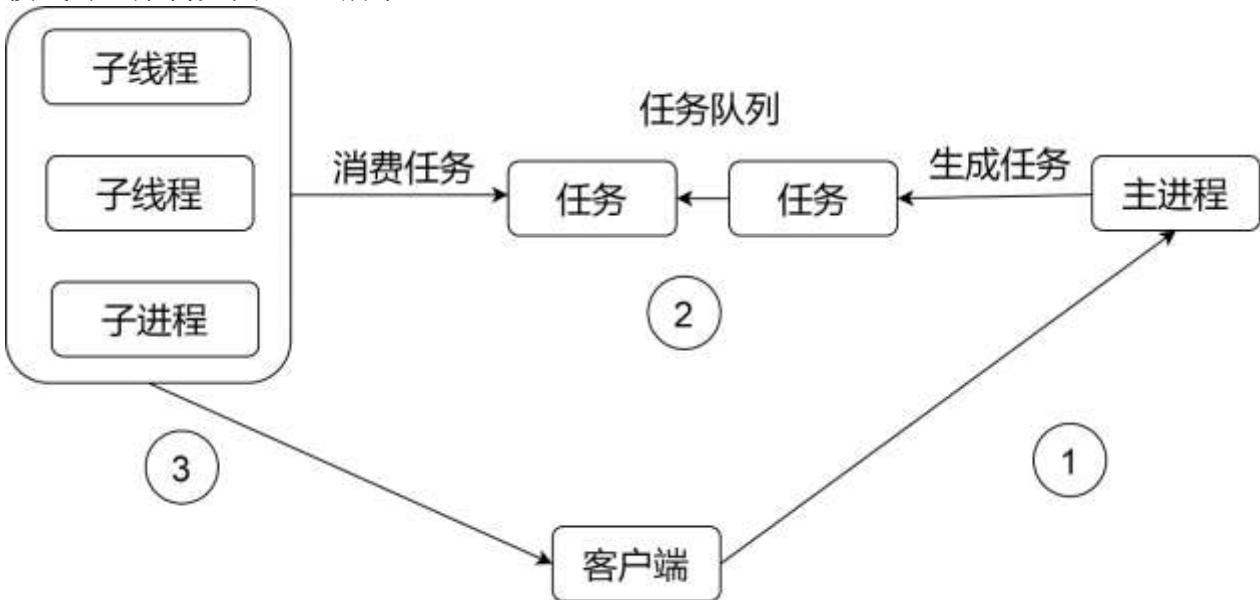


图 1-10

主进程负责 accept 请求，然后通过互斥的方式插入一个任务到共享队列中，线程池中的子线程同样是通过互斥的方式，从共享队列中摘取节点进行处理。

1.4.4 事件驱动

现在很多服务器（Nginx, Node.js, Redis）都开始使用事件驱动模式去设计。从之前的设计模式中我们知道，为了应对大量的请求，服务器需要大量的进程/线程。这个是个非常大的开销。而事件驱动模式，一般是配合单进程（单线程），再多的请求，也是在一个进程里处理的。但是因为是单进程，所以不适合 CPU 密集型，因为一个任务一直在占据 CPU 的话，后续的任务就无法执行了。它更适合 IO 密集的（一般都会提供一个线程池，负责处理 CPU 或者阻塞型的任务）。而使用多进程/线程模式的时候，一个进程/线程是无法一直占据 CPU 的，执行一定时间后，操作系统会执行任务调度。让其它线程也有机会执行，这样就不会前面的任务阻塞后面的任务，出现饥饿情况。大部分操作系统都提供了事件驱动的 API。但是事件驱动在不同系统中实现不一样。所以一般都会有一层抽象层抹平这个差异。这里以 Linux 的 epoll 为例子。

```
1. // 创建一个 epoll
2. var epollFD = epoll_create();
3. /*
4. 在 epoll 给某个文件描述符注册感兴趣的事件，这里是监听的 socket，注册可
5. 读事件，即连接到来
6. event = {
7.     event: 可读
```

```

8.     fd: 监听 socket
9.     // 一些上下文
10.    }
11.   */
12.   epoll_ctl(epollFD , EPOLL_CTL_ADD , socket, event);
13.   while(1) {
14.     // 阻塞等待事件就绪, events 保存就绪事件的信息, total 是个数
15.     var total= epoll_wait(epollFD , 保存就绪事件的结构 events, 事件
    个数, timeout);
16.     for (let i = 0; i < total; i++) {
17.       if (events[i].fd === 监听 socket) {
18.         var newSocket = accpet(socket);
19.         /*
20.           把新的 socket 也注册到 epoll, 等待可读,
21.           即可读取客户端数据
22.         */
23.         epoll_ctl(epollFD,
24.                   EPOLL_CTL_ADD,
25.                   newSocket,
26.                   可读事件);
27.     } else {
28.       // 从 events[i]中拿到一些上下文, 执行相应的回调
29.     }
30.   }
31. }
```

这就是事件驱动模式的大致过程，本质上是一个订阅/发布模式。服务器通过注册文件描述符和事件到 epoll 中，epoll 开始阻塞，等到 epoll 返回的时候，它会告诉服务器哪些 fd 的哪些事件触发了，这时候服务器遍历就绪事件，然后执行对应的回调，在回调里可以再次注册新的事件，就是这样不断驱动着。epoll 的原理其实也类似事件驱动，epoll 底层维护用户注册的事件和文件描述符，epoll 本身也会在文件描述符对应的文件 /socket/管道处注册一个回调，然后自身进入阻塞，等到别人通知 epoll 有事件发生的时候，epoll 就会把 fd 和事件返回给用户。

```

1. function epoll_wait() {
2.   for 事件个数
3.     // 调用文件系统的函数判断
4.     if (事件[i]中对应的文件描述符中有某个用户感兴趣的事件发
    生?) {
5.       插入就绪事件队列
6.     } else {
7.       /*
8.         在事件[i]中的文件描述符所对应的文件/socket/管道等 indeo 节
        点注册回调。即感兴趣的事件触发后回调 epoll，回调 epoll 后，
```

```
10.          epoll 把该 event[i] 插入就绪事件队列返回给用户
11.      */
12.  }
13. }
```

以上就是服务器设计的一些基本介绍。现在的服务器的设计中还会涉及到协程。不过目前还没有看过具体的实现，所以暂不展开介绍，有兴趣的通信可以看一下协程库 **libtask** 了解一下如何使用协程实现一个服务器。

Node.js 是基于单进程（单线程）的事件驱动模式。这也是为什么 **Node.js** 擅长处理高并发 IO 型任务而不擅长处理 CPU 型任务的原因，**Nginx**、**Redis** 也是这种模式。另外 **Node.js** 是一个及 web 服务器和应用服务器于一身的服务器，像 **Nginx** 这种属于 web 服务器，它们只处理 HTTP 协议，不具备脚本语言来处理具体的业务逻辑。它需要把请求转发到真正的 web 服务器中去处理，比如 **PHP**。而 **Node.js** 不仅可以解析 HTTP 协议，还可以处理具体的业务逻辑。

第二章 Libuv 数据结构和通用逻辑

2.1 核心结构体 uv_loop_s

`uv_loop_s` 是 Libuv 的核心数据结构，每一个事件循环对应一个 `uv_loop_s` 结构体。它记录了整个事件循环中的核心数据。我们来分析每一个字段的意义。

1 用户自定义数据的字段

```
void* data;
```

2 活跃的 handle 个数，会影响使用循环的退出

```
unsigned int active_handles;
```

3 handle 队列，包括活跃和非活跃的

```
void* handle_queue[2];
```

4 request 个数，会影响事件循环的退出

```
union { void* unused[2]; unsigned int count; } active_reqs;
```

5 事件循环是否结束的标记

unsigned int stop_flag;

6 Libuv 运行的一些标记，目前只有 UV_LOOP_BLOCK_SIGPROF，主要是用于 epoll_wait 的时候屏蔽 SIGPROF 信号，提高性能，SIGPROF 是调操作系统 settimer 函数设置从而触发的信号

unsigned long flags;

7 epoll 的 fd

int backend_fd;

8 pending 阶段的队列

void* pending_queue[2];

9 指向需要在 epoll 中注册事件的 uv__io_t 结构体队列

void* watcher_queue[2];

10 watcher_queue 队列的节点中有一个 fd 字段，watchers 以 fd 为索引，记录 fd 所在的 uv__io_t 结构体

uv__io_t** watchers;

11 watchers 相关的数量，在 maybe_resize 函数里设置

unsigned int nwatchers;

12 watchers 里 fd 个数，一般为 watcher_queue 队列的节点数

unsigned int nfds;

13 线程池的子线程处理完任务后把对应的结构体插入到 wq 队列

void* wq[2];

14 控制 wq 队列互斥访问，否则多个子线程同时访问会有问题

uv_mutex_t wq_mutex;

15 用于线程池的子线程和主线程通信

uv_async_t wq_async;

16 用于读写锁的互斥变量

uv_rwlock_t cloexec_lock;

17 事件循环 close 阶段的队列，由 uv_close 产生

uv_handle_t* closing_handles;

18 fork 出来的进程队列

void* process_handles[2];

19 事件循环的 prepare 阶段对应的任务队列

void* prepare_handles[2];

20 事件循环的 check 阶段对应的任务队列

void* check_handles[2];

21 事件循环的 idle 阶段对应的任务队列

void* idle_handles[2];

21 async_handles 队列，Poll IO 阶段执行 uv__async_io 中遍历 async_handles 队列处理里面 pending 为 1 的节点

void* async_handles[2];

22 用于监听是否有 async handle 任务需要处理

uv__io_t async_io_watcher;

23 用于保存子线程和主线程通信的写端 fd

int async_wfd;

24 保存定时器二叉堆结构

```
struct {  
    void* min;  
    unsigned int nelts;  
} timer_heap;
```

25 管理定时器节点的 id，不断叠加

uint64_t timer_counter;

26 当前时间，Libuv 会在每次事件循环的开始和 Poll IO 阶段更新当前时间，然后在后续的各个阶段使用，减少对系统调用

uint64_t time;

27 用于 fork 出来的进程和主进程通信的管道，用于子进程收到信号的时候通知主进程，然后主进程执行子进程节点注册的回调

int signal_pipefd[2];

28 类似 async_io_watcher，signal_io_watcher 保存了管道读端 fd 和回调，然后注册到 epoll 中，在子进程收到信号的时候，通过 write 写到管道，最后在 Poll IO 阶段执行回调

uv__io_t signal_io_watcher;

29 用于管理子进程退出信号的 handle

```
uv_signal_t child_watcher;
```

30 备用的 fd

```
int emfile_fd;
```

2.2 uv_handle_t

在 Libuv 中，uv_handle_t 类似 C++中的基类，有很多子类继承于它，Libuv 主要通过控制内存的布局得到继承的效果。handle 代表生命周期比较长的对象。例如

1 一个处于 active 状态的 prepare handle，它的回调会在每次事件循环化的时候被执行。

2 一个 tcp handle 在每次有连接到来时，执行它的回调。

我们看一下 uv_handle_t 的定义

1 自定义数据，用于关联一些上下文，Node.js 中用于关联 handle 所属的 C++对象

```
void* data;
```

2 所属的事件循环

```
uv_loop_t* loop;
```

3 handle 类型

```
uv_handle_type type;
```

4 handle 调用 uv_close 后，在 closing 阶段被执行的回调

```
uv_close_cb close_cb;
```

5 用于组织 handle 队列的前置后置指针

```
void* handle_queue[2];
```

6 文件描述符

```
union {
    int fd;
    void* reserved[4];
} u;
```

7 当 handle 在 close 队列时，该字段指向下一个 close 节点

```
uv_handle_t* next_closing;
```

8 handle 的状态和标记

unsigned int flags;

2.2.1 uv_stream_s

uv_stream_s 是表示流的结构体。除了继承 uv_handle_t 的字段外，它额外定义下面字段

1 等待发送的字节数

size_t write_queue_size;

2 分配内存的函数

uv_alloc_cb alloc_cb;

3 读取数据成功时执行的回调

uv_read_cb read_cb;

4 发起连接对应的结构体

uv_connect_t *connect_req;

5 关闭写端对应的结构体

uv_shutdown_t *shutdown_req;

6 用于插入 epoll，注册读写事件

uv_io_t io_watcher;

7 待发送队列

void* write_queue[2];

8 发送完成的队列

void* write_completed_queue[2];

9 收到连接时执行的回调

uv_connection_cb connection_cb;

10 socket 操作失败的错误码

int delayed_error;

11 accept 返回的 fd

int accepted_fd;

12 已经 accept 了一个 fd，又有新的 fd，暂存起来

void* queued_fds;

2.2.2 uv_async_s

uv_async_s 是 Libuv 中实现异步通信的结构体。继承于 uv_handle_t，并额外定义了以下字段。

1 异步事件触发时执行的回调

```
uv_async_cb async_cb;
```

2 用于插入 async-handles 队列

```
void* queue[2];
```

3 async_handles 队列中的节点 pending 字段为 1 说明对应的事件触发了

```
int pending;
```

2.2.3 uv_tcp_s

uv_tcp_s 继承 uv_handle_s 和 uv_stream_s。

2.2.4 uv_udp_s

1 发送字节数

```
size_t send_queue_size;
```

2 写队列节点的个数

```
size_t send_queue_count;
```

3 分配接收数据的内存

```
uv_alloc_cb alloc_cb;
```

4 接收完数据后执行的回调

```
uv_udp_recv_cb recv_cb;
```

5 插入 epoll 里的 I/O 观察者，实现数据读写

```
uv_io_t io_watcher;
```

6 待发送队列

```
void* write_queue[2];
```

7 发送完成的队列（发送成功或失败），和待发送队列相关

```
void* write_completed_queue[2];
```

2.2.5 uv_tty_s

uv_tty_s 继承于 uv_handle_t 和 uv_stream_t。额外定义了下面字段。

1 终端的参数

```
struct termios orig_termios;
```

2 终端的工作模式

```
int mode;
```

2.2.6 uv_pipe_s

uv_pipe_s 继承于 uv_handle_t 和 uv_stream_t。额外定义了下面字段。

1 标记管道是否可用于传递文件描述符

```
int ipc;
```

2 用于 Unix 域通信的文件路径

```
const char* pipe_fname;
```

2.2.7 uv_prepare_s、uv_check_s、uv_idle_s

上面三个结构体定义是类似的，它们都继承 uv_handle_t，额外定义了两个字段。

1 prepare、check、idle 阶段回调

```
uv_xxx_cb xxx_cb;
```

2 用于插入 prepare、check、idle 队列

```
void* queue[2];
```

2.2.8 uv_timer_s

uv_timer_s 继承 uv_handle_t，额外定义了下面几个字段。

1 超时回调

```
uv_timer_cb timer_cb;
```

2 插入二叉堆的字段

```
void* heap_node[3];
```

3 超时时间

```
uint64_t timeout;
```

4 超时后是否继续开始重新计时，是的话重新插入二叉堆

```
uint64_t repeat;
```

5 id 标记，用于插入二叉堆的时候对比

```
uint64_t start_id
```

2.2.9 uv_process_s

uv_process_s 继承 uv_handle_t，额外定义了

1 进程退出时执行的回调

```
uv_exit_cb exit_cb;
```

2 进程 id

```
int pid;
```

3 用于插入队列，进程队列或者 pending 队列

```
void* queue[2];
```

4 退出码，进程退出时设置

```
int status;
```

2.2.10 uv_fs_event_s

uv_fs_event_s 用于监听文件改动。uv_fs_event_s 继承 uv_handle_t，额外定义了

1 监听的文件路径(文件或目录)

```
char* path;
```

2 文件改变时执行的回调

```
uv_fs_event_cb cb;
```

2.2.11 uv_fs_poll_s

uv_fs_poll_s 继承 uv_handle_t，额外定义了

1 poll_ctx 指向 poll_ctx 结构体

```
void* poll_ctx;
```

```
struct poll_ctx {
```

```
    // 对应的 handle
```

```
uv_fs_poll_t* parent_handle;
// 标记是否开始轮询和轮询时的失败原因
int busy_polling;
// 多久检测一次文件内容是否改变
unsigned int interval;
// 每一轮轮询时的开始时间
uint64_t start_time;
// 所属事件循环
uv_loop_t* loop;
// 文件改变时回调
uv_fs_poll_cb poll_cb;
// 定时器，用于定时超时后轮询
uv_timer_t timer_handle;
// 记录轮询的一下上下文信息，文件路径、回调等
uv_fs_t fs_req;
// 轮询时保存操作系统返回的文件信息
uv_stat_t statbuf;
// 监听的文件路径，字符串的值追加在结构体后面
char path[1]; /* variable length */
};
```

2.2.12 uv_poll_s

uv_poll_s 继承于 uv_handle_t，额外定义了下面字段。

1 监听的 fd 有感兴趣的事件时执行的回调

```
uv_poll_cb poll_cb;
```

2 保存了 fd 和回调的 IO 观察者，注册到 epoll 中

```
uv_io_t io_watcher;
```

2.1.13 uv_signal_s

uv_signal_s 继承 uv_handle_t，额外定义了以下字段

1 收到信号时的回调

```
uv_signal_cb signal_cb;
```

2 注册的信号

```
int signum;
```

3 用于插入红黑树，进程把感兴趣的信号和回调封装成 uv_signal_s，然后插入到红黑树，信号到来时，进程在信号处理号中把通知写入管道，通知 Libuv。Libuv 在 Poll IO 阶段会执行进程对应的回调。红黑树节点的定义如下

```
struct {
    struct uv_signal_s* rbe_left;
    struct uv_signal_s* rbe_right;
    struct uv_signal_s* rbe_parent;
    int rbe_color;
} tree_entry;
```

4 收到的信号个数

```
unsigned int caught_signals;
```

5 已经处理的信号个数

```
unsigned int dispatched_signals;
```

2.3 uv_req_s

在 Libuv 中，uv_req_s 也类似 C++ 基类的作用，有很多子类继承于它，request 代表一次请求，比如读写一个文件，读写 socket，查询 DNS。任务完成后这个 request 就结束了。request 可以和 handle 结合使用，比如在一个 TCP 服务器上（handle）写一个数据（request），也可以单独使用一个 request，比如 DNS 查询或者文件读写。我们看一下 uv_req_s 的定义。

1 自定义数据

```
void* data;
```

2 request 类型

```
uv_req_type type;
```

3 保留字段

```
void* reserved[6];
```

2.3.1 uv_shutdown_s

uv_shutdown_s 用于关闭流的写端，额外定义的字段

1 要关闭的流，比如 TCP

```
uv_stream_t* handle;
```

2 关闭流的写端后执行的回调

```
uv_shutdown_cb cb;
```

2.3.2 uv_write_s

uv_write_s 表示一次写请求，比如在 TCP 流上发送数据，额外定义的字段

1 写完后的回调

```
uv_write_cb cb;
```

2 需要传递的文件描述符，在 send_handle 中

```
uv_stream_t* send_handle;
```

3 关联的 handle

```
uv_stream_t* handle;
```

4 用于插入队列

```
void* queue[2];
```

5 保存需要写的数据相关的字段（写入的 buffer 个数，当前写成功的位
置等）

```
unsigned int write_index;
```

```
uv_buf_t* bufs;
```

```
unsigned int nbufs;
```

```
uv_buf_t bufsml[4];
```

6 写出错的错误码

```
int error;
```

2.3.3 uv_connect_s

uv_connect_s 表示发起连接请求，比如 TCP 连接，额外定义的字段

1 连接成功后执行的回调

```
uv_connect_cb cb;
```

2 对应的流，比如 tcp

```
uv_stream_t* handle;
```

3 用于插入队列

```
void* queue[2];
```

2.3.4 uv_udp_send_s

uv_udp_send_s 表示一次发送 UDP 数据的请求

1 所属 udp 的 handle, udp_send_s 代表一次发送
uv_udp_t* handle;

2 回调
uv_udp_send_cb cb;

3 用于插入待发送队列
void* queue[2];

4 发送的目的地址
struct sockaddr_storage addr;

5 保存了发送数据的缓冲区和个数
unsigned int nbefs;
uv_buf_t* bufs;
uv_buf_t bufsml[4];

6 发送状态或成功发送的字节数
ssize_t status;

7 发送完执行的回调 (发送成功或失败)
uv_udp_send_cb send_cb;

2.3.5 uv_getaddrinfo_s

uv_getaddrinfo_s 表示一次通过域名查询 IP 的 DNS 请求，额外定义的字段

1 所属事件循环
uv_loop_t* loop;

2 用于异步 DNS 解析时插入线程池任务队列的节点
struct uv_work work_req;

3 DNS 解析完后执行的回调
uv_getaddrinfo_cb cb;

4 DNS 查询的配置
struct addrinfo* hints;
char* hostname;
char* service;

5 DNS 解析结果

```
struct addrinfo* addrinfo;
```

6 DNS 解析的返回码

```
int retcode;
```

2.3.6 uv_getnameinfo_s

uv_getnameinfo_s 表示一次通过 IP 查询域名的 DNS 查询请求，额外定义的字段

1 所属事件循环

```
uv_loop_t* loop;
```

2 用于异步 DNS 解析时插入线程池任务队列的节点

```
struct uv_work work_req;
```

3 socket 转域名完成的回调

```
uv_getnameinfo_cb getnameinfo_cb;
```

4 需要转域名的 socket 结构体

```
struct sockaddr_storage storage;
```

5 指示查询返回的信息

```
int flags;
```

6 查询返回的信息

```
char host[NI_MAXHOST];
```

```
char service[NI_MAXSERV];
```

7 查询返回码

```
int retcode;
```

2.3.7 uv_work_s

uv_work_s 用于往线程池提交任务，额外定义的字段

1 所属事件循环

```
uv_loop_t* loop;
```

2 处理任务的函数

```
uv_work_cb work_cb;
```

3 处理完任务后执行的函数

```
uv_after_work_cb after_work_cb;
```

4 封装一个 work 插入到线程池队列，work_req 的 work 和 done 函数是对上面 work_cb 和 after_work_cb 的封装

```
struct uv_work work_req;
```

2.3.8 uv_fs_s

uv_fs_s 表示一次文件操作请求，额外定义的字段

1 文件操作类型

```
uv_fs_type fs_type;
```

2 所属事件循环

```
uv_loop_t* loop;
```

3 文件操作完成的回调

```
uv_fs_cb cb;
```

4 文件操作的返回码

```
ssize_t result;
```

5 文件操作返回的数据

```
void* ptr;
```

6 文件操作路径

```
const char* path;
```

7 文件的 stat 信息

```
uv_stat_t statbuf;
```

8 文件操作涉及到两个路径时，保存目的路径

```
const char *new_path;
```

9 文件描述符

```
uv_file file;
```

10 文件标记

```
int flags;
```

11 操作模式

```
mode_t mode;
```

12 写文件时传入的数据和个数

```
unsigned int nbufs;  
uv_buf_t* bufs;
```

13 文件偏移

```
off_t off;
```

14 保存需要设置的 uid 和 gid, 例如 chmod 的时候

```
uv_uid_t uid;  
uv_gid_t gid;
```

15 保存需要设置的文件修改、访问时间, 例如 fs.utimes 的时候

```
double atime;  
double mtime;
```

16 异步的时候用于插入任务队列, 保存工作函数, 回调函数

```
struct uv_work work_req;
```

17 保存读取数据或者长度。例如 read 和 sendfile

```
uv_buf_t bufsml[4];
```

2.4 IO 观察者

IO 观察者是 Libuv 中的核心概念和数据结构。我们看一下它的定义

```
1. struct uv_io_s {  
2.     // 事件触发后的回调  
3.     uv_io_cb cb;  
4.     // 用于插入队列  
5.     void* pending_queue[2];  
6.     void* watcher_queue[2];  
7.     // 保存本次感兴趣的事件, 在插入 IO 观察者队列时设置  
8.     unsigned int pevents;  
9.     // 保存当前感兴趣的事件  
10.    unsigned int events;  
11.    int fd;  
12.};
```

IO 观察者封装了文件描述符、事件和回调，然后插入到 loop 维护的 IO 观察者队列，在 Poll I/O 阶段，Libuv 会根据 IO 观察者描述的信息，往底层的事件驱动模块注册文件描述符感兴趣的事件。当注册的事件触发的时候，IO 观察者的回调就会被执行。我们看如何初 I/O 观察者的一些逻辑。

2.4.1 初始化 I/O 观察者

```

1. void uv_io_init(uv_io_t* w, uv_io_cb cb, int fd) {
2.     // 初始化队列，回调，需要监听的 fd
3.     QUEUE_INIT(&w->pending_queue);
4.     QUEUE_INIT(&w->watcher_queue);
5.     w->cb = cb;
6.     w->fd = fd;
7.     // 上次加入 epoll 时感兴趣的事件，在执行完 epoll 操作函数后设置
8.     w->events = 0;
9.     // 当前感兴趣的事件，在再次执行 epoll 函数之前设置
10.    w->pevents = 0;
11. }
```

2.4.2 注册一个 I/O 观察者到 Libuv。

```

1. void uv_io_start(uv_loop_t* loop, uv_io_t* w, unsigned int events) {
2.     // 设置当前感兴趣的事件
3.     w->pevents |= events;
4.     // 可能需要扩容
5.     maybe_resize(loop, w->fd + 1);
6.     // 事件没有变化则直接返回
7.     if (w->events == w->pevents)
8.         return;
9.     // I/O 观察者没有挂载在其它地方则插入 Libuv 的 I/O 观察者队列
10.    if (QUEUE_EMPTY(&w->watcher_queue))
11.        QUEUE_INSERT_TAIL(&loop->watcher_queue, &w->watcher_queue);
12.    // 保存映射关系
13.    if (loop->watchers[w->fd] == NULL) {
14.        loop->watchers[w->fd] = w;
15.        loop->nfds++;
16.    }
17. }
```

`uv_io_start` 函数就是把一个 I/O 观察者插入到 Libuv 的观察者队列中，并且在 `watchers` 数组中保存一个映射关系。Libuv 在 Poll I/O 阶段会处理 I/O 观察者队列。

2.4.3 撤销 IO 观察者或者事件

uv__io_stop 修改 IO 观察者感兴趣的事件，如果还有感兴趣的事件的话，IO 观察者还会在队列里，否则移出

```
1. void uv__io_stop(uv_loop_t* loop,
2.                     uv__io_t* w,
3.                     unsigned int events) {
4.     if (w->fd == -1)
5.         return;
6.     assert(w->fd >= 0);
7.     if ((unsigned) w->fd >= loop->nwatchers)
8.         return;
9.     // 清除之前注册的事件，保存在 pevents 里，表示当前感兴趣的事件
10.    w->pevents &= ~events;
11.    // 对所有事件都不感兴趣了
12.    if (w->pevents == 0) {
13.        // 移出 IO 观察者队列
14.        QUEUE_REMOVE(&w->watcher_queue);
15.        // 重置
16.        QUEUE_INIT(&w->watcher_queue);
17.        // 重置
18.        if (loop->watchers[w->fd] != NULL) {
19.            assert(loop->watchers[w->fd] == w);
20.            assert(loop->nfds > 0);
21.            loop->watchers[w->fd] = NULL;
22.            loop->nfds--;
23.            w->events = 0;
24.        }
25.    }
26.    /*
27.     * 之前还没有插入 IO 观察者队列，则插入，
28.     * 等到 Poll IO 时处理，否则不需要处理
29.     */
30.    else if (QUEUE_EMPTY(&w->watcher_queue))
31.        QUEUE_INSERT_TAIL(&loop->watcher_queue, &w->watcher_queue);
32. }
```

2.5 Libuv 通用逻辑

2.5.1 uv_handle_init

uv_handle_init 初始化 handle 的类型，设置 REF 标记，插入 handle 队列。

```

1. #define uv_handle_init(loop_, h, type_)
2.   do {
3.     (h)->loop = (loop_);
4.     (h)->type = (type_);
5.     (h)->flags = UV_HANDLE_REF;
6.     QUEUE_INSERT_TAIL(&(loop_->handle_queue), &(h)->handle_queue);
7.     (h)->next_closing = NULL
8.   }
9.   while (0)

```

2.5.2. uv_handle_start

uv_handle_start 设置标记 handle 为 ACTIVE，如果设置了 REF 标记，则 active handle 的个数加一，active handle 数会影响事件循环的退出。

```

1. #define uv_handle_start(h)
2.   do {
3.     if (((h)->flags & UV_HANDLE_ACTIVE) != 0) break;
4.
5.     (h)->flags |= UV_HANDLE_ACTIVE;
6.     if (((h)->flags & UV_HANDLE_REF) != 0)
7.       (h)->loop->active_handles++;
8.   }
9.   while (0)

```

2.5.3. uv_handle_stop

uv_handle_stop 和 uv_handle_start 相反。

```

1. #define uv_handle_stop(h)
2.   do {
3.     if (((h)->flags & UV_HANDLE_ACTIVE) == 0) break;
4.     (h)->flags &= ~UV_HANDLE_ACTIVE;
5.     if (((h)->flags & UV_HANDLE_REF) != 0) uv_active_handle_r
6.       m(h);

```

```
6.     }
7.     while (0)
```

Libuv 中 handle 有 REF 和 ACTIVE 两个状态。当一个 handle 调用 xxx_init 函数的时候，它首先被打上 REF 标记，并且插入 loop->handle 队列。当 handle 调用 xxx_start 函数的时候，它被打上 ACTIVE 标记，并且记录 active handle 的个数加一。只有 REF 并且 ACTIVE 状态的 handle 才会影响事件循环的退出。

2.5.4. uv_req_init

uv_req_init 初始化请求的类型，记录请求的个数，会影响事件循环的退出。

```
1. #define uv_req_init(loop, req, typ)
2.     do {
3.         (req)->type = (typ);
4.         (loop)->active_reqs.count++;
5.     }
6.     while (0)
```

2.5.5. uv_req_register

请求的个数加一

```
1. #define uv_req_register(loop, req)
2.     do {
3.         (loop)->active_reqs.count++;
4.     }
5.     while (0)
```

2.5.6. uv_req_unregister

请求个数减一

```
1. #define uv_req_unregister(loop, req)
2.     do {
3.         assert(uv_has_active_reqs(loop));
4.         (loop)->active_reqs.count--;
5.     }
6.     while (0)
```

2.5.7. uv_handle_ref

uv_handle_ref 标记 handle 为 REF 状态，如果 handle 是 ACTIVE 状态，则 active handle 数加一

```

1. #define uv_handle_ref(h)
2.   do {
3.     if (((h)->flags & UV_HANDLE_REF) != 0) break;
4.     (h)->flags |= UV_HANDLE_REF;
5.     if (((h)->flags & UV_HANDLE_CLOSING) != 0) break;
6.     if (((h)->flags & UV_HANDLE_ACTIVE) != 0) uv_active_handles_add(h);
7.   }
8.   while (0)

9. uv_handle_unref

```

uv_handle_unref 去掉 handle 的 REF 状态，如果 handle 是 ACTIVE 状态，则 active handle 数减一

```

1. #define uv_handle_unref(h)
2.   do {
3.     if (((h)->flags & UV_HANDLE_REF) == 0) break;
4.     (h)->flags &= ~UV_HANDLE_REF;
5.     if (((h)->flags & UV_HANDLE_CLOSING) != 0) break;
6.     if (((h)->flags & UV_HANDLE_ACTIVE) != 0) uv_active_handles_rm(h);
7.   }
8.   while (0)

```

第三章 事件循环

Node.js 属于单线程事件循环架构，该事件循环由 Libuv 的 uv_run 函数实现，在该函数中执行 while 循环，然后不断地处理各个阶段（phase）的事件回调。事件循环的处理相当于一个消费者，消费由各种代码产生的任务。Node.js 初始化完成后就开始陷入该事件循环中，事件循环的结束也就意味着 Node.js 的结束。下面看一下事件循环的核心代码。

```

1. int uv_run(uv_loop_t* loop, uv_run_mode mode) {
2.   int timeout;
3.   int r;

```

```
4. int ran_pending;
5. // 在 uv_run 之前要先提交任务到 loop
6. r = uv_loop_alive(loop);
7. // 事件循环没有任务执行，即将退出，设置一下当前循环的时间
8. if (!r)
9.     uv_update_time(loop);
10. // 没有任务需要处理或者调用了 uv_stop 则退出事件循环
11. while (r != 0 && loop->stop_flag == 0) {
12.     // 更新 loop 的 time 字段
13.     uv_update_time(loop);
14.     // 执行超时回调
15.     uv_run_timers(loop);
16.     /*
17.         执行 pending 回调，ran_pending 代表 pending 队列是否为空，
18.         即没有节点可以执行
19.     */
20.     ran_pending = uv_run_pending(loop);
21.     // 继续执行各种队列
22.     uv_run_idle(loop);
23.     uv_run_prepare(loop);
24.
25.     timeout = 0;
26.     /*
27.         执行模式是 UV_RUN_ONCE 时，如果没有 pending 节点，
28.         才会阻塞式 Poll IO，默认模式也是
29.     */
30.     if ((mode == UV_RUN_ONCE && !ran_pending) ||
31.         mode == UV_RUN_DEFAULT)
32.         timeout = uv_backend_timeout(loop);
33.     // Poll IO timeout 是 epoll_wait 的超时时间
34.     uv_io_poll(loop, timeout);
35.     // 处理 check 阶段
36.     uv_run_check(loop);
37.     // 处理 close 阶段
38.     uv_run_closing_handles(loop);
39.     /*
40.         还有一次执行超时回调的机会，因为 uv_io_poll 可能是因为
41.         定时器超时返回的。
42.     */
43.     if (mode == UV_RUN_ONCE) {
44.         uv_update_time(loop);
45.         uv_run_timers(loop);
46.     }
47.
```

```

48.     r = uv__loop_alive(loop);
49.     /*
50.         只执行一次，退出循环，UV_RUN_NOWAIT 表示在 Poll IO 阶段
51.         不会阻塞并且循环只执行一次
52.     */
53.     if (mode == UV_RUN_ONCE || mode == UV_RUN_NOWAIT)
54.         break;
55. }
56. // 是因为调用了 uv_stop 退出的，重置 flag
57. if (loop->stop_flag != 0)
58.     loop->stop_flag = 0;
59. /*
60.     返回是否还有活跃的任务（handle 或 request），
61.     业务代表可以再次执行 uv_run
62. */
63. return r;
64. }
```

Libuv 分为几个阶段，下面从先到后，分别分析各个阶段的相关代码。

3.1 事件循环之定时器

Libuv 中，定时器阶段是第一个被处理的阶段。定时器是以最小堆实现的，最快过期的节点是根节点。Libuv 在每次事件循环开始的时候都会缓存当前的时间，在每一轮的事件循环中，使用的都是这个缓存的时间，必要的时候 Libuv 会显式更新这个时间，因为获取时间需要调用操作系统提供的接口，而频繁调用系统调用会带来一定的耗时，缓存时间可以减少操作系统的调用，提高性能。Libuv 缓存了当前最新的时间后，就执行 `uv_run_timers`，该函数就是遍历最小堆，找出当前超时的节点。因为堆的性质是父节点肯定比孩子小。并且根节点是最小的，所以如果一个根节点，它没有超时，则后面的节点也不会超时。对于超时的节点就执行它的回调。我们看一下具体的逻辑。

```

1. void uv_run_timers(uv_loop_t* loop) {
2.     struct heap_node* heap_node;
3.     uv_timer_t* handle;
4.     // 遍历二叉堆
5.     for (;;) {
6.         // 找出最小的节点
7.         heap_node = heap_min(timer_heap(loop));
8.         // 没有则退出
9.         if (heap_node == NULL)
10.             break;
11.         // 通过结构体字段找到结构体首地址
```

```
12.     handle = container_of(heap_node, uv_timer_t, heap_node);
13.     // 最小的节点都没有超时，则后面的节点也不会超时
14.     if (handle->timeout > loop->time)
15.         break;
16.     // 删除该节点
17.     uv_timer_stop(handle);
18.     /*
19.         重试插入二叉堆，如果需要的话（设置了 repeat，比如
20.         setInterval）
21.     */
22.     uv_timer_again(handle);
23.     // 执行回调
24.     handle->timer_cb(handle);
25. }
26. }
```

执行完回调后，还有两个关键的操作，第一就是 stop，第二就是 again。stop 的逻辑很简单，就是把 handle 从二叉堆中删除，并且修改 handle 的状态。那么 again 又是什么呢？again 是为了支持 setInterval 这种场景，如果 handle 设置了 repeat 标记，则该 handle 在超时后，每 repeat 的时间后，就会继续执行超时回调。对于 setInterval，就是超时时间是 x，每 x 的时间后，执行回调。这就是 Node.js 里定时器的底层原理。但 Node.js 不是每次调 setTimeout/setInterval 的时候都往最小堆插入一个节点，Node.js 里，只有一个关于 uv_timer_s 的 handle，它在 JS 层维护了一个数据结构，每次计算出最早到期的节点，然后修改 handle 的超时时间，具体在定时器章节讲解。

另外 timer 阶段和 Poll IO 阶段也有一些联系，因为 Poll IO 可能会导致主线程阻塞，为了保证主线程可以尽快执行定时器的回调，Poll IO 不能一直阻塞，所以这时候，阻塞的时长就是最快到期的定时器节点的时长（具体可参考 libuv core.c 中的 uv_backend_timeout 函数）。

3.2 pending 阶段

官网对 pending 阶段的解释是在上一轮的 Poll IO 阶段没有执行的 IO 回调，会在下一轮循环的 pending 阶段被执行。从源码来看，Poll IO 阶段处理任务时，在某些情况下，如果当前执行的操作失败需要执行回调通知调用方一些信息，该回调函数不会立刻执行，而是在下一轮事件循环的 pending 阶段执行（比如写入数据成功，或者 TCP 连接失败时回调 C++ 层），我们先看 pending 阶段的处理。

```
1. static int uv__run_pending(uv_loop_t* loop) {
2.     QUEUE* q;
3.     QUEUE pq;
4.     uv__io_t* w;
5.
6.     if (QUEUE_EMPTY(&loop->pending_queue))
```

```

7.     return 0;
8.     // 把 pending_queue 队列的节点移到 pq, 即清空了 pending_queue
9.     QUEUE_MOVE(&loop->pending_queue, &pq);
10.
11.    // 遍历 pq 队列
12.    while (!QUEUE_EMPTY(&pq)) {
13.        // 取出当前第一个需要处理的节点, 即 pq.next
14.        q = QUEUE_HEAD(&pq);
15.        // 把当前需要处理的节点移出队列
16.        QUEUE_REMOVE(q);
17.        /*
18.            重置一下 prev 和 next 指针, 因为这时候这两个指针是
19.            指向队列中的两个节点
20.        */
21.        QUEUE_INIT(q);
22.        w = QUEUE_DATA(q, uv__io_t, pending_queue);
23.        w->cb(loop, w, POLLOUT);
24.    }
25.
26.    return 1;
27. }
```

pending 阶段的处理逻辑就是把 pending 队列里的节点逐个执行。我们看一下 pending 队列的节点是如何生产出来的。

```

1. void uv__io_feed(uv_loop_t* loop, uv__io_t* w) {
2.     if (QUEUE_EMPTY(&w->pending_queue))
3.         QUEUE_INSERT_TAIL(&loop->pending_queue, &w->pending_queue);
4. }
```

Libuv 通过 uv__io_feed 函数生产 pending 任务，从 Libuv 的代码中我们看到 IO 错误的时候会调这个函数（如 tcp.c 的 uv__tcp_connect 函数）。

```

1. if (handle->delayed_error)
2.     uv__io_feed(handle->loop, &handle->io_watcher);
```

在写入数据成功后（比如 TCP、UDP），也会往 pending 队列插入一个节点，等待回调。比如发送数据成功后执行的代码（udp.c 的 uv__udp_sendmsg 函数）

```

1. // 发送完移出写队列
2. QUEUE_REMOVE(&req->queue);
3. // 加入写完成队列
4. QUEUE_INSERT_TAIL(&handle->write_completed_queue, &req->queue);
5. /*
6.     有节点数据写完了, 把 IO 观察者插入 pending 队列,
```

```
7. pending 阶段执行回调
8. */
9. uv__io_feed(handle->loop, &handle->io_watcher);
```

最后关闭 I/O 的时候（如关闭一个 TCP 连接）会从 pending 队列移除对应的节点，因为已经关闭了，自然就不需要执行回调。

```
1. void uv__io_close(uv_loop_t* loop, uv__io_t* w) {
2.     uv__io_stop(loop,
3.                  w,
4.                  POLLIN | POLLOUT | UV__POLLRDHUP | UV__POLLPRI);
5.     QUEUE_REMOVE(&w->pending_queue);
6. }
```

3.3 事件循环之 prepare, check, idle

prepare, check, idle 是 Libuv 事件循环中属于比较简单的一个阶段，它们的实现是一样的（见 `loop-watcher.c`）。本节只讲解 `prepare` 阶段，我们知道 Libuv 中分为 `handle` 和 `request`，而 `prepare` 阶段的任务是属于 `handle` 类型。这意味着除非我们显式移除，否则 `prepare` 阶段的节点在每次事件循环中都会被执行。下面我们先看看怎么使用它。

```
1. void prep_cb(uv_prepare_t *handle) {
2.     printf("Prep callback\n");
3. }
4.
5. int main() {
6.     uv_prepare_t prep;
7.     // 初始化一个 handle, uv_default_loop 是事件循环的核心结构体
8.     uv_prepare_init(uv_default_loop(), &prep);
9.     // 注册 handle 的回调
10.    uv_prepare_start(&prep, prep_cb);
11.    // 开始事件循环
12.    uv_run(uv_default_loop(), UV_RUN_DEFAULT);
13.    return 0;
14. }
```

执行 `main` 函数，Libuv 就会在 `prepare` 阶段执行回调 `prep_cb`。我们分析一下这个过程。

```
1. int uv_prepare_init(uv_loop_t* loop, uv_prepare_t* handle) {
2.     uv__handle_init(loop, (uv_handle_t*)handle, UV_PREPARE);
3.     handle->prepare_cb = NULL;
4.     return 0;
```

| 5. }

init 函数主要是做一些初始化操作。我们继续要看 start 函数。

```

1. int uv_prepare_start(uv_prepare_t* handle, uv_prepare_cb cb) {
2.     // 如果已经执行过 start 函数则直接返回
3.     if (uv_is_active(handle)) return 0;
4.     if (cb == NULL) return UV_EINVAL;
5.     QUEUE_INSERT_HEAD(&handle->loop->prepare_handles,
6.                        &handle->queue);
7.     handle->prepare_cb = cb;
8.     uv_handle_start(handle);
9.     return 0;
10. }
```

uv_prepare_start 函数主要的逻辑主要是设置回调，把 handle 插入 loop 的 prepare_handles 队列，prepare_handles 队列保存了 prepare 阶段的任务。在事件循环的 prepare 阶段会逐个执行里面的节点的回调。然后我们看看 Libuv 在事件循环的 prepare 阶段是如何处理的。

```

1. void uv_run_prepare(uv_loop_t* loop) {
2.     uv_prepare_t* h;
3.     QUEUE queue;
4.     QUEUE* q;
5.     /*
6.      把该类型对应的队列中所有节点摘下来挂载到 queue 变量,
7.      相当于清空 prepare_handles 队列, 因为如果直接遍历
8.      prepare_handles 队列, 在执行回调的时候一直往 prepare_handles
9.      队列加节点, 会导致下面的 while 循环无法退出。
10.     先移除的话, 新插入的节点在下一轮事件循环才会被处理。
11.     */
12.     QUEUE_MOVE(&loop->prepare_handles, &queue);
13.     // 遍历队列, 执行每个节点里面的函数
14.     while (!QUEUE_EMPTY(&queue)) {
15.         // 取下当前待处理的节点, 即队列的头
16.         q = QUEUE_HEAD(&queue);
17.         /*
18.          取得该节点对应的整个结构体的地址,
19.          即通过结构体成员取得结构体首地址
20.          */
21.         h = QUEUE_DATA(q, uv_prepare_t, queue);
```

```
22.          // 把该节点移出当前队列
23.          QUEUE_REMOVE(q);
24.          // 重新插入原来的队列
25.          QUEUE_INSERT_TAIL(&loop->prepare_handles, q);
26.          // 执行回调函数
27.          h->prepare_cb(h);
28.      }
29.  }
```

uv_run_prepare 函数的逻辑很简单，但是有一个重点的地方就是执行完每一个节点，Libuv 会把该节点重新插入队列中，所以 prepare（包括 idle、check）阶段的节点在每一轮事件循环中都会被执行。而像定时器、pending、closing 阶段的节点是一次性的，被执行后就会从队列里删除。

我们回顾一开始的测试代码。因为它设置了 Libuv 的运行模式是默认模式。而 prepare 队列又一直有一个 handle 节点，所以它是不会退出的。它会一直执行回调。那如果我们要退出怎么办呢？或者说不要执行 prepare 队列的某个节点了。我们只需要 stop 一下就可以了。

```
1.  int uv_prepare_stop(uv_prepare_t* handle) {
2.      if (!uv_is_active(handle)) return 0;
3.      // 把 handle 从 prepare 队列中移除，但还挂载到 handle_queue 中
4.      QUEUE_REMOVE(&handle->queue);
5.      // 清除 active 标记位并且减去 loop 中 handle 的 active 数
6.      uv_handle_stop(handle);
7.      return 0;
8.  }
```

stop 函数和 start 函数是相反的作用，这就是 Node.js 中 prepare、check、idle 阶段的原理。

3.4 事件循环之 Poll IO

Poll IO 是 Libuv 非常重要的一个阶段，文件 IO、网络 IO、信号处理等都在这个阶段处理，这也是最复杂的一个阶段。处理逻辑在 core.c 的 uv_io_poll 这个函数，这个函数比较复杂，我们分开分析。在开始分析 Poll IO 之前，先了解一下它相关的一些数据结构。

1 IO 观察者 uv_io_t。这个结构体是 Poll IO 阶段核心结构体。它主要是保存了 IO 相关的文件描述符、回调、感兴趣的事件等信息。

2 watcher_queue 观察者队列。所有需要 Libuv 处理的 IO 观察者都挂载在这个队列里，Libuv 在 Poll IO 阶段会逐个处理。

下面我们开始分析 Poll IO 阶段。先看第一段逻辑。

```

1. // 没有 IO 观察者，则直接返回
2. if (loop->nfds == 0) {
3.     assert(QUEUE_EMPTY(&loop->watcher_queue));
4.     return;
5. }
6. // 遍历 IO 观察者队列
7. while (!QUEUE_EMPTY(&loop->watcher_queue)) {
8.     // 取出当前头节点
9.     q = QUEUE_HEAD(&loop->watcher_queue);
10.    // 脱离队列
11.    QUEUE_REMOVE(q);
12.    // 初始化（重置）节点的前后指针
13.    QUEUE_INIT(q);
14.    // 通过结构体成功获取结构体首地址
15.    w = QUEUE_DATA(q, uv_io_t, watcher_queue);
16.    // 设置当前感兴趣的事件
17.    e.events = w->pevents;
18.    /*
19.        这里使用了 fd 字段，事件触发后再通过 fd 从 watchs
20.        字段里找到对应的 IO 观察者，没有使用 ptr 指向 IO 观察者的方案
21.    */
22.    e.data.fd = w->fd;
23.    // 如果 w->events 初始化的时候为 0，则新增，否则修改
24.    if (w->events == 0)
25.        op = EPOLL_CTL_ADD;
26.    else
27.        op = EPOLL_CTL_MOD;
28.    // 修改 epoll 的数据
29.    epoll_ctl(loop->backend_fd, op, w->fd, &e)
30.    // 记录当前加到 epoll 时的状态
31.    w->events = w->pevents;
32. }
```

第一步首先遍历 IO 观察者，修改 epoll 的数据。然后准备进入等待。

```

1. psigset = NULL;
2. if (loop->flags & UV_LOOP_BLOCK_SIGPROF) {
3.     sigemptyset(&sigset);
4.     sigaddset(&sigset, SIGPROF);
```

```
5.     psigset = &sigset;
6. }
7. /*
8.      http://man7.org/Linux/man-pages/man2/epoll_wait.2.html
9.      pthread_sigmask(SIG_SETMASK, &sigmask, &origmask);
10.     ready = epoll_wait(epfd, &events, maxevents, timeout);
11.     pthread_sigmask(SIG_SETMASK, &origmask, NULL);
12.     即屏蔽 SIGPROF 信号，避免 SIGPROF 信号唤醒 epoll_wait，但是却没有就绪的事件
13. */
14. nfds = epoll_pwait(loop->backend_fd,
15.                      events,
16.                      ARRAY_SIZE(events),
17.                      timeout,
18.                      psigset);
19. // epoll 可能阻塞，这里需要更新事件循环的时间
20. uv_update_time(loop)
```

epoll_wait 可能会引起主线程阻塞，所以 wait 返回后需要更新当前的时间，否则在使用的时候时间差会比较大，因为 Libuv 会在每轮时间循环开始的时候缓存当前时间这个值。其它地方直接使用，而不是每次都去获取。下面我们接着看 epoll 返回后的处理（假设有事件触发）。

```
1. // 保存 epoll_wait 返回的一些数据，maybe_resize 申请空间的时候+2 了
2. loop->watchers[loop->nwatchers] = (void*) events;
3. loop->watchers[loop->nwatchers + 1] = (void*) (uintptr_t) nf
   ds;
4. for (i = 0; i < nfds; i++) {
5.     // 触发的事件和文件描述符
6.     pe = events + i;
7.     fd = pe->data.fd;
8.     // 根据 fd 获取 IO 观察者，见上面的图
9.     w = loop->watchers[fd];
10.    // 会其它回调里被删除了，则从 epoll 中删除
11.    if (w == NULL) {
12.        epoll_ctl(loop->backend_fd, EPOLL_CTL_DEL, fd, pe);
13.        continue;
14.    }
15.    if (pe->events != 0) {
16.        /*
17.         * 用于信号处理的 IO 观察者感兴趣的事件触发了，即有信号发生。
18.     }
```

```

19.         */
20.         if (w == &loop->signal_io_watcher)
21.             have_signals = 1;
22.         else
23.             // 一般的 IO 观察者则执行回调
24.             w->cb(loop, w, pe->events);
25.             nevents++;
26.     }
27. }
28. // 有信号发生，触发回调
29. if (have_signals != 0)
30.     loop->signal_io_watcher.cb(loop,
31.                               &loop->signal_io_watcher,
32.                               POLLIN);

```

上面的代码处理 IO 事件并执行 IO 观察者里的回调，但是有一个特殊的地方就是信号处理的 IO 观察者需要单独判断，它是一个全局的 IO 观察者，和一般动态申请和销毁的 IO 观察者不一样，它是存在于 Libuv 运行的整个生命周期。这就是 Poll IO 的整个过程。

3.5 事件循环之 close

close 是 Libuv 每轮事件循环中最后的一个阶段。uv_close 用于关闭一个 handle，并且执行一个回调。uv_close 产生的任务会插入到 close 阶段的队列，然后在 close 阶段被处理。我们看一下 uv_close 函数的实现。

```

1. void uv_close(uv_handle_t* handle, uv_close_cb close_cb) {
2.     // 正在关闭，但是还没执行回调等后置操作
3.     handle->flags |= UV_HANDLE_CLOSING;
4.     handle->close_cb = close_cb;
5.
6.     switch (handle->type) {
7.     case UV_PREPARE:
8.         uv_prepare_close((uv_prepare_t*)handle);
9.         break;
10.    case UV_CHECK:
11.        uv_check_close((uv_check_t*)handle);
12.        break;
13.        ...
14.    default:
15.        assert(0);
16.    }

```

```
17.     uv__make_close_pending(handle);  
18. }
```

uv_close 设置回调和状态，然后根据 handle 类型调对应的 close 函数，一般就是 stop 这个 handle，解除 I/O 观察者注册的事件，从事件循环的 handle 队列移除该 handle 等等，比如 prepare 的 close 函数只是把 handle 从队列中移除。

```
1. void uv__prepare_close(uv_prepare_t* handle) {  
2.     uv_prepare_stop(handle);  
3. }  
4. int uv_prepare_stop(uv_prepare_t* handle) {  
5.     QUEUE_REMOVE(&handle->queue);  
6.     uv__handle_stop(handle);  
7.     return 0;  
8. }
```

根据不同的 handle 做不同的处理后，接着执行 uv__make_close_pending 往 close 队列追加节点。

```
1. // 头插法插入 closing 队列，在 closing 阶段被执行  
2. void uv__make_close_pending(uv_handle_t* handle) {  
3.     handle->next_closing = handle->loop->closing_handles;  
4.     handle->loop->closing_handles = handle;  
5. }
```

然后在 close 阶段逐个处理。我们看一下 close 阶段的处理逻辑

```
1. // 执行 closing 阶段的的回调  
2. static void uv__run_closing_handles(uv_loop_t* loop) {  
3.     uv_handle_t* p;  
4.     uv_handle_t* q;  
5.  
6.     p = loop->closing_handles;  
7.     loop->closing_handles = NULL;  
8.  
9.     while (p) {  
10.         q = p->next_closing;  
11.         uv__finish_close(p);  
12.         p = q;  
13.     }  
14. }
```

```

15.
16. // 执行 closing 阶段的回调
17. static void uv_finish_close(uv_handle_t* handle) {
18.     handle->flags |= UV_HANDLE_CLOSED;
19.     ...
20.     uv_handle_unref(handle);
21.     // 从 handle 队列里移除
22.     QUEUE_REMOVE(&handle->handle_queue);
23.     if (handle->close_cb) {
24.         handle->close_cb(handle);
25.     }
26. }
```

uv_run_closing_handles 会逐个执行每个任务节点的回调。

3.6 控制事件循环

Libuv 通过 uv_loop_alive 函数判断事件循环是否还需要继续执行。我们看看这个函数的定义。

```

1. static int uv_loop_alive(const uv_loop_t* loop) {
2.     return uv_has_active_handles(loop) ||
3.            uv_has_active_reqs(loop) ||
4.            loop->closing_handles != NULL;
5. }
```

为什么会有 `closing_handles` 的判断呢？从 `uv_run` 的代码来看，执行完 `close` 阶段后，会立刻执行 `uv_loop_alive`，正常来说，`close` 阶段的队列是空的，但是如果我们在 `close` 回调里又往 `close` 队列新增了一个节点，而该节点不会在本轮的 `close` 阶段被执行，这样会导致执行完 `close` 阶段，但是 `close` 队列依然有节点，如果直接退出，则无法执行对应的回调。

我们看到有三种情况，Libuv 认为事件循环是存活的。如果我们控制这三种条件就可以控制事件循环的退出。我们通过一个例子理解一下这个过程。

```

1. const timeout = setTimeout(() => {
2.     console.log('never console')
3. }, 5000);
4. timeout.unref();
```

上面的代码中，`setTimeout` 的回调是不会执行的。除非超时时间非常短，短到第一轮事件循环的时候就到期了，否则在第一轮事件循环之后，由于 `unref` 的影响，事件循环直接退出了。`unref` 影响的就是 `handle` 这个条件。这时候事件循环代码如下。

```
1. while (r != 0 && loop->stop_flag == 0) {  
2.     uv_update_time(loop);  
3.     uv_run_timers(loop);  
4.     // ...  
5.     // uv_loop_alive 返回 false, 直接跳出 while, 从而退出事件循环  
6.     r = uv_loop_alive(loop);  
7. }
```

第四章 线程池

Libuv 是单线程事件驱动的异步 I/O 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 I/O、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。

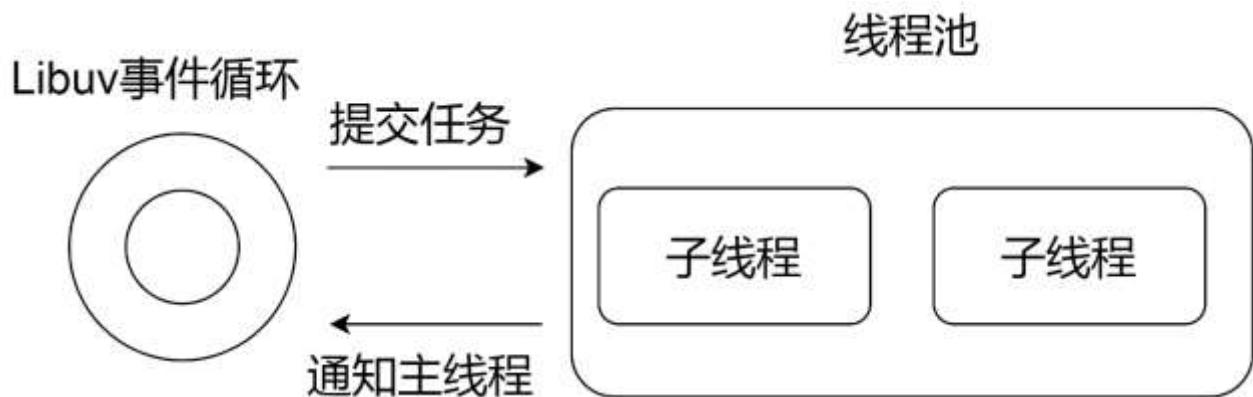


图 4-1

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 I/O 观察者，即 `loop->async_handles` 队列上所有的 handle 都是共享 `async_io_watcher` 这个 I/O 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。

```

1. int uv_async_init(uv_loop_t* loop,
2.                     uv_async_t* handle,
3.                     uv_async_cb  async_cb) {
4.     int err;
5.     // 给 Libuv 注册一个观察者 io
6.     err = uv__async_start(loop);
7.     if (err)
8.         return err;
9.     // 设置相关字段，给 Libuv 插入一个 handle
10.    uv__handle_init(loop, (uv_handle_t*)handle, UV_ASYNC);
11.    // 设置回调
12.    handle->async_cb = async_cb;
13.    // 初始化标记字段，0 表示没有任务完成
14.    handle->pending = 0;
15.    // 把 uv_async_t 插入 async_handle 队列
16.    QUEUE_INSERT_TAIL(&loop->async_handles, &handle->queue);
17.    uv__handle_start(handle);
18.    return 0;
19. }
```

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。

```

1. static int uv__async_start(uv_loop_t* loop) {
2.     int pipefd[2];
3.     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.     // 获得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", pipefd[0])
25.         ; // 通过一个 fd 就可以实现对管道的读写, 高级用法
26.         fd = uv_open_cloexec(buf, O_RDWR);
27.         if (fd >= 0) {
28.             // 关掉旧的
29.             uv_close(pipefd[0]);
30.             uv_close(pipefd[1]);
31.             // 赋值新的
32.             pipefd[0] = fd;
33.             pipefd[1] = fd;
34.         }
35.#endif
36.     }
37.     // err 大于等于 0 说明拿到了通信的读写两端
38.     if (err < 0)
39.         return err;
40. /*
41.     初始化 IO 观察者 async_io_watcher,
42.     把读端文件描述符保存到 IO 观察者
43. */
44. uv_io_init(&loop->async_io_watcher, uv_async_io, pipefd[0]);
45. // 注册 IO 观察者到 loop 里, 并注册感兴趣的事件 POLLIN, 等待可读
46. uv_io_start(loop, &loop->async_io_watcher, POLLIN);
47. // 保存写端文件描述符
48. loop->async_wfd = pipefd[1];
49. return 0;
```

|50. }

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 I/O 观察者然后追加到 watcher_queue 队列，在 Poll I/O 阶段，Libuv 会注册到 epoll 里面，如果有任务完成，也会在 Poll I/O 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。

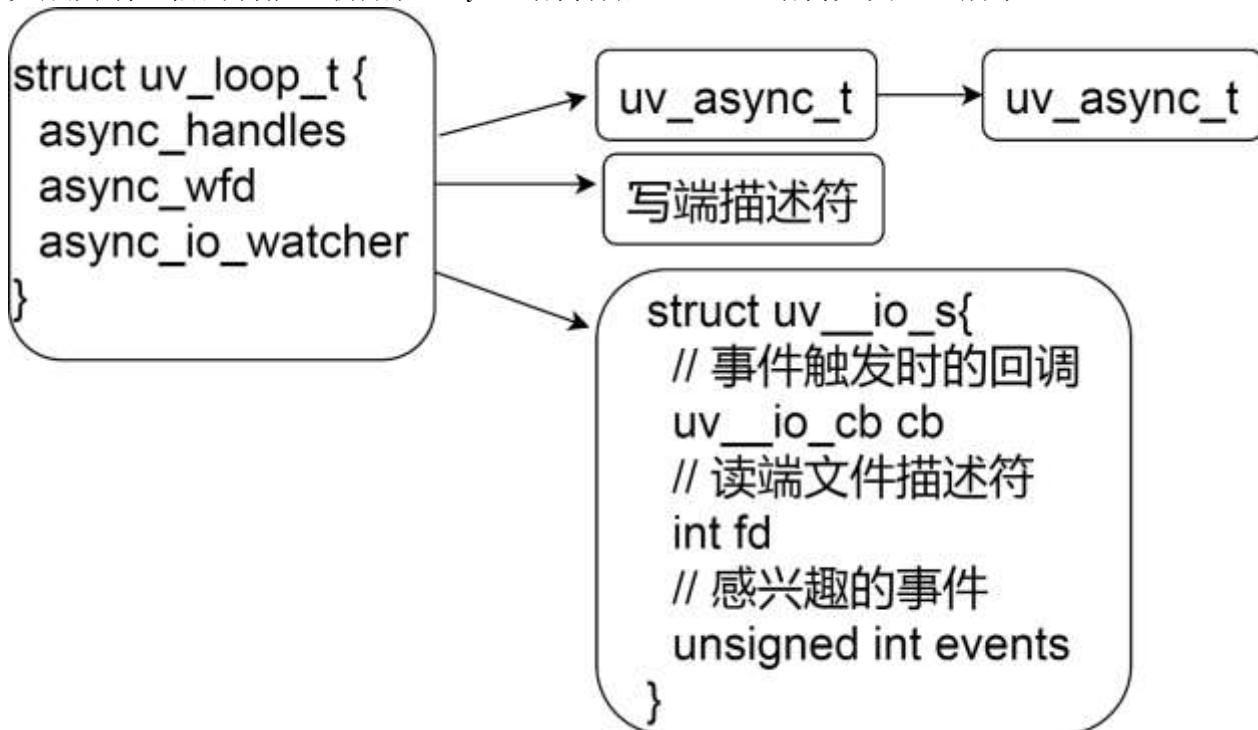


图 4-2

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。

```

1. int uv_async_send(uv_async_t* handle) {
2.     /* Do a cheap read first. */
3.     if (ACCESS_ONCE(int, handle->pending) != 0)
4.         return 0;

```

```
5.  /*
6.      如 pending 是 0, 则设置为 1, 返回 0, 如果是 1 则返回 1,
7.      所以如果多次调用该函数是会被合并的
8. */
9.  if (cmpxchgi (&handle->pending, 0, 1) == 0)
10.     uv_async_send(handle->loop);
11.    return 0;
12. }
13.
14. static void uv_async_send(uv_loop_t* loop) {
15.     const void* buf;
16.     ssize_t len;
17.     int fd;
18.     int r;
19.
20.     buf = "";
21.     len = 1;
22.     fd = loop->async_wfd;
23.
24. #if defined(_Linux_)
25.     // 说明用的是 eventfd 而不是管道, eventfd 时读写两端对应同一个 fd
26.     if (fd == -1) {
27.         static const uint64_t val = 1;
28.         buf = &val;
29.         len = sizeof(val);
30.         // 见 uv_async_start
31.         fd = loop->async_io_watcher.fd; /* eventfd */
32.     }
33. #endif
34.     // 通知读端
35.     do
36.         r = write(fd, buf, len);
37.     while (r == -1 && errno == EINTR);
38.
39.     if (r == len)
40.         return;
41.
42.     if (r == -1)
43.         if (errno == EAGAIN || errno == EWOULDBLOCK)
44.             return;
45.
46.     abort();
```

| 47. }

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

```

1. static void uv_async_io(uv_loop_t* loop,
2.                         uv_io_t* w,
3.                         unsigned int events) {
4.     char buf[1024];
5.     ssize_t r;
6.     QUEUE queue;
7.     QUEUE* q;
8.     uv_async_t* h;
9.
10.    for (;;) {
11.        // 消费所有的数据
12.        r = read(w->fd, buf, sizeof(buf));
13.        // 数据大小大于 buf 长度 (1024)，则继续消费
14.        if (r == sizeof(buf))
15.            continue;
16.        // 成功消费完毕，跳出消费的逻辑
17.        if (r != -1)
18.            break;
19.        // 读繁忙
20.        if (errno == EAGAIN || errno == EWOULDBLOCK)
21.            break;
22.        // 读被中断，继续读
23.        if (errno == EINTR)
24.            continue;
25.        abort();
26.    }
27.    // 把 async_handles 队列里的所有节点都移到 queue 变量中
28.    QUEUE_MOVE(&loop->async_handles, &queue);
29.    while (!QUEUE_EMPTY(&queue)) {
30.        // 逐个取出节点
31.        q = QUEUE_HEAD(&queue);
32.        // 根据结构体字段获取结构体首地址
33.        h = QUEUE_DATA(q, uv_async_t, queue);

```

```
34.     // 从队列中移除该节点
35.     QUEUE_REMOVE(q);
36.     // 重新插入 async_handles 队列，等待下次事件
37.     QUEUE_INSERT_TAIL(&loop->async_handles, q);
38.     /*
39.         将第一个参数和第二个参数进行比较，如果相等，
40.         则将第三个参数写入第一个参数，返回第二个参数的值，
41.         如果不相等，则返回第一个参数的值。
42.     */
43.     /*
44.         判断触发了哪些 async_pending 在 uv_async_send 里设置成 1,
45.         如果 pending 等于 1，则清 0，返回 1。如果 pending 等于 0，则返回 0
46.     */
47.     if (cmpxchgi(&h->pending, 1, 0) == 0)
48.         continue;
49.
50.     if (h->async_cb == NULL)
51.         continue;
52.     // 执行上层回调
53.     h->async_cb(h);
54. }
55. }
```

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 default_threads[4];
5.     nthreads = ARRAY_SIZE(default_threads);
6.     // 判断用户是否在环境变量中设置了线程数，是的话取用户定义的
```

```

7.     val  =  getenv("UV_THREADPOOL_SIZE");
8.     if  (val  !=  NULL)
9.         nthreads  =  atoi(val);
10.    if  (nthreads  ==  0)
11.        nthreads  =  1;
12.    // #define MAX_THREADPOOL_SIZE 128 最多 128 个线程
13.    if  (nthreads  >  MAX_THREADPOOL_SIZE)
14.        nthreads  =  MAX_THREADPOOL_SIZE;
15.
16.    threads  =  default_threads;
17.    // 超过默认大小, 重新分配内存
18.    if  (nthreads  >  ARRAY_SIZE(default_threads))  {
19.        threads  =  uv_malloc(nthreads  *  sizeof(threads[0]));
20.    }
21.    // 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
22.    if  (uv_cond_init(&cond))
23.        abort();
24.    // 初始化互斥变量, 用于多个子线程互斥访问任务队列
25.    if  (uv_mutex_init(&mutex))
26.        abort();
27.
28.    // 初始化三个队列
29.    QUEUE_INIT(&wq);
30.    QUEUE_INIT(&slow_io_pending_wq);
31.    QUEUE_INIT(&run_slow_work_message);
32.
33.    // 创建多个线程, 工作函数为 worker, sem 为 worker 入参
34.    for  (i  =  0;  i  <  nthreads;  i++)
35.        if  (uv_thread_create(threads  +  i,  worker,  &sem))
36.            abort();
37. }
```

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

```
1. // 给线程池提交一个任务
```

```
2. void uv_work_submit(uv_loop_t* loop,
3.                      struct uv_work* w,
4.                      enum uv_work_kind kind,
5.                      void (*work)(struct uv_work* w),
6.                      void (*done)(struct uv_work* w, int status))
7. {
8.     /* 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
9.      任务的时候才被初始化，init_once -> init_threads */
10.    */
11.    uv_once(&once, init_once);
12.    w->loop = loop;
13.    w->work = work;
14.    w->done = done;
15.    post(&w->wq, kind);
16. }
```

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。

```
1. struct uv_work {
2.     void (*work)(struct uv_work *w);
3.     void (*done)(struct uv_work *w, int status);
4.     struct uv_loop_s* loop;
5.     void* wq[2];
6. };
```

然后调用 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。

```
1. static void post(QUEUE* q, enum uv_work_kind kind) {
2.     // 加锁访问任务队列，因为这个队列是线程池共享的
3.     uv_mutex_lock(&mutex);
4.     // 类型是慢 IO
5.     if (kind == UV_WORK_SLOW_IO) {
6.         /*
7.          插入慢 IO 对应的队列，Libuv 这个版本把任务分为几种类型，
8.          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
9.          run_slow_work_message，然后用 slow_io_pending_wq 维护了一个慢 IO
10.         任务的队列，当处理到 run_slow_work_message 这个节点的时候，
11.         Libuv 会从 slow_io_pending_wq 队列里逐个取出任务节点来执行。
12.     }
```

```

12.    */
13.    QUEUE_INSERT_TAIL(&slow_io_pending_wq, q);
14.    /*
15.        有慢 IO 任务的时候，需要给主队列 wq 插入一个消息节点
16.        run_slow_work_message，说明有慢 IO 任务，所以如果
17.        run_slow_work_message 是空，说明还没有插入主队列。需要进行
18.        q = &run_slow_work_message; 赋值，然后把
19.        run_slow_work_message 插入主队列。如果 run_slow_work_message
20.        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。
21.    */
22.    if (!QUEUE_EMPTY(&run_slow_work_message)) {
23.        uv_mutex_unlock(&mutex);
24.        return;
25.    }
26.    // 说明 run_slow_work_message 还没有插入队列，准备插入队列
27.    q = &run_slow_work_message;
28. }
29. // 把节点插入主队列，可能是慢 IO 消息节点或者一般任务
30. QUEUE_INSERT_TAIL(&wq, q);
31. /*
32.     有空闲线程则唤醒它，如果大家都在忙，
33.     则等到它忙完后就会重新判断是否还有新任务
34. */
35. if (idle_threads > 0)
36.     uv_cond_signal(&cond);
37. // 操作完队列，解锁
38. uv_mutex_unlock(&mutex);
39. }

```

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。

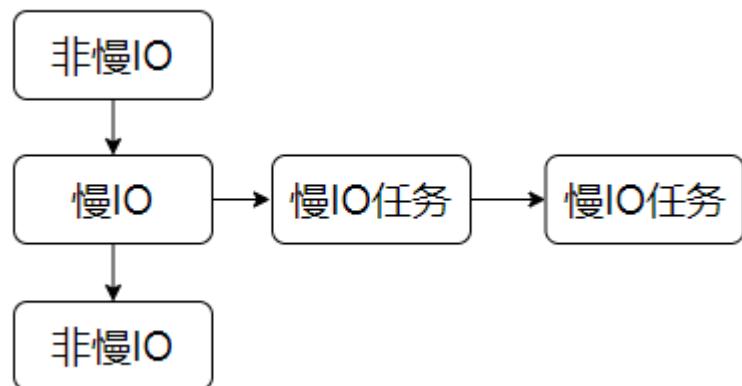


图 4-3

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。

```
1. int uv_queue_work(uv_loop_t* loop,
2.                     uv_work_t* req,
3.                     uv_work_cb work_cb,
4.                     uv_after_work_cb after_work_cb) {
5.
6.     if (work_cb == NULL)
7.         return UV_EINVAL;
8.
9.     uv_req_init(loop, req, UV_WORK);
10.    req->loop = loop;
11.    req->work_cb = work_cb;
12.    req->after_work_cb = after_work_cb;
13.    uv_work_submit(loop,
14.                   &req->work_req,
15.                   UV_WORK_CPU,
16.                   uv_queue_work,
17.                   uv_queue_done);
18.    return 0;
19. }
```

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。

```
1. static void uv_queue_work(struct uv_work* w) {
2.     // 通过结构体某字段拿到结构体地址
3.     uv_work_t* req = container_of(w, uv_work_t, work_req);
4.     req->work_cb(req);
5. }
```

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。

```

1. static void worker(void* arg) {
2.     struct uv_work* w;
3.     QUEUE* q;
4.     int is_slow_work;
5.     // 线程启动成功
6.     uv_sem_post((uv_sem_t*) arg);
7.     arg = NULL;
8.     // 加锁互斥访问任务队列
9.     uv_mutex_lock(&mutex);
10.    for (;;) {
11.        /*
12.         1 队列为空
13.         2 队列不为空，但是队列中只有慢 I/O 任务且正在执行的慢 I/O 任务
14.             个数达到阈值则空闲线程加一，防止慢 I/O 占用过多线程，导致
15.             其它快的任务无法得到执行
16.        */
17.        while (QUEUE_EMPTY(&wq) ||
18.               (QUEUE_HEAD(&wq) == &run_slow_work_message &&
19.                QUEUE_NEXT(&run_slow_work_message) == &wq &&
20.                slow_io_work_running >= slow_work_thread_threshold()))
21.        {
22.            idle_threads += 1;
23.            // 阻塞，等待唤醒
24.            uv_cond_wait(&cond, &mutex);
25.            // 被唤醒，开始干活，空闲线程数减一
26.            idle_threads -= 1;
27.        }
28.        // 取出头结点，头结点可能是退出消息、慢 I/O，一般请求
29.        q = QUEUE_HEAD(&wq);
30.        // 如果头结点是退出消息，则结束线程
31.        if (q == &exit_message) {
32.            /*
33.             唤醒其它因为没有任务正阻塞等待任务的线程，
34.             告诉它们准备退出
35.            */
36.            uv_cond_signal(&cond);
37.            uv_mutex_unlock(&mutex);

```

```
37.         break;
38.     }
39.     // 移除节点
40.     QUEUE_REMOVE(q);
41.     // 重置前后指针
42.     QUEUE_INIT(q);
43.     is_slow_work = 0;
44.     /*
45.      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
46.      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
47.      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
48.      IO 任务，继续判断是否还有非慢 IO 任务可执行。
49.     */
50.     if (q == &run_slow_work_message) {
51.         // 达到阈值，该节点重新入队，因为刚才被删除了
52.         if (slow_io_work_running >= slow_work_thread_threshold())
53.         {
54.             QUEUE_INSERT_TAIL(&wq, q);
55.             continue;
56.         }
57.         /*
58.          没有慢 IO 任务则继续，这时候 run_slow_work_message
59.          已经从队列中被删除，下次有慢 IO 的时候重新入队
60.        */
61.        if (QUEUE_EMPTY(&slow_io_pending_wq))
62.            continue;
63.        // 有慢 IO，开始处理慢 IO 任务
64.        is_slow_work = 1;
65.        /*
66.          正在处理慢 IO 任务的个数累加，用于其它线程判断慢 IO 任务个
67.          数是否达到阈值，slow_io_work_running 是多个线程共享的变量
68.        */
69.        slow_io_work_running++;
70.        // 摘下一个慢 IO 任务
71.        q = QUEUE_HEAD(&slow_io_pending_wq);
72.        // 从慢 IO 队列移除
73.        QUEUE_REMOVE(q);
74.        QUEUE_INIT(q);
75.        /*
76.          取出一个任务后，如果还有慢 IO 任务则把慢 IO 标记节点重新入
77.          队，表示还有慢 IO 任务，因为上面把该标记节点出队了
78.        */
```

```

78.         if (!QUEUE_EMPTY(&slow_io_pending_wq)) {
79.             QUEUE_INSERT_TAIL(&wq, &run_slow_work_message);
80.             // 有空闲线程则唤醒它, 因为还有任务处理
81.             if (idle_threads > 0)
82.                 uv_cond_signal(&cond);
83.         }
84.     }
85.     // 不需要操作队列了, 尽快释放锁
86.     uv_mutex_unlock(&mutex);
87.     // q 是慢 IO 或者一般任务
88.     w = QUEUE_DATA(q, struct uv_work, wq);
89.     // 执行业务的任务函数, 该函数一般会阻塞
90.     w->work(w);
91.     // 准备操作 loop 的任务完成队列, 加锁
92.     uv_mutex_lock(&w->loop->wq_mutex);
93.     // 置空说明执行完了, 见 cancel 逻辑
94.     w->work = NULL;
95.     /*
96.      执行完任务, 插入到 loop 的 wq 队列, 在 uv_work_done 的时候会
97.      执行该队列的节点
98.    */
99.     QUEUE_INSERT_TAIL(&w->loop->wq, &w->wq);
100.    // 通知 loop 的 wq_async 节点
101.    uv_async_send(&w->loop->wq_async);
102.    uv_mutex_unlock(&w->loop->wq_mutex);
103.    // 为下一轮操作任务队列加锁
104.    uv_mutex_lock(&mutex);
105.    /*
106.      执行完慢 IO 任务, 记录正在执行的慢 IO 个数变量减 1,
107.      上面加锁保证了互斥访问这个变量
108.    */
109.    if (is_slow_work) {
110.        slow_io_work_running--;
111.    }
112. }
113. }
```

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。

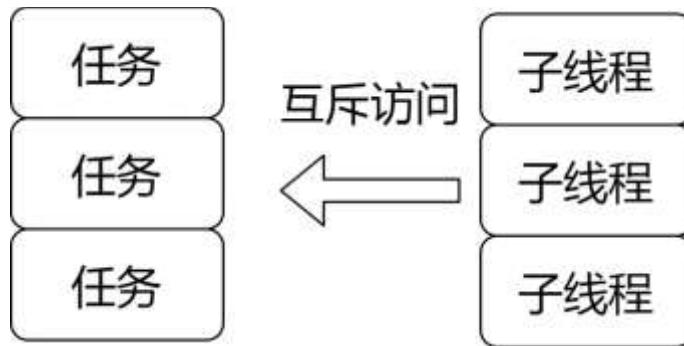


图 4-4

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 I/O 观察者，Libuv 在 Poll I/O 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。

```
1. void uv_work_done(uv_async_t* handle) {  
2.     struct uv_work* w;  
3.     uv_loop_t* loop;  
4.     QUEUE* q;  
5.     QUEUE wq;  
6.     int err;  
7.     // 通过结构体字段获得结构体首地址  
8.     loop = container_of(handle, uv_loop_t, wq_async);  
9.     // 准备处理队列，加锁  
10.    uv_mutex_lock(&loop->wq_mutex);  
11.    /*  
12.        loop->wq 是已完成的任务队列。把 loop->wq 队列的节点全部移到  
13.        wp 变量中，这样一来可以尽快释放锁  
14.    */  
15.    QUEUE_MOVE(&loop->wq, &wq);  
16.    // 不需要使用了，解锁
```

```

17.     uv_mutex_unlock(&loop->wq_mutex);
18.     // wq 队列的节点来自子线程插入
19.     while (!QUEUE_EMPTY(&wq)) {
20.         q = QUEUE_HEAD(&wq);
21.         QUEUE_REMOVE(q);
22.         w = container_of(q, struct uv_work, wq);
23.         // 等于 uv_cancelled 说明这个任务被取消了
24.         err = (w->work == uv_cancelled) ? UV_ECANCELED : 0;
25.         // 执行回调
26.         w->done(w, err);
27.     }
28. }

```

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，然后再到底层。结构图如图 4-5 所示。

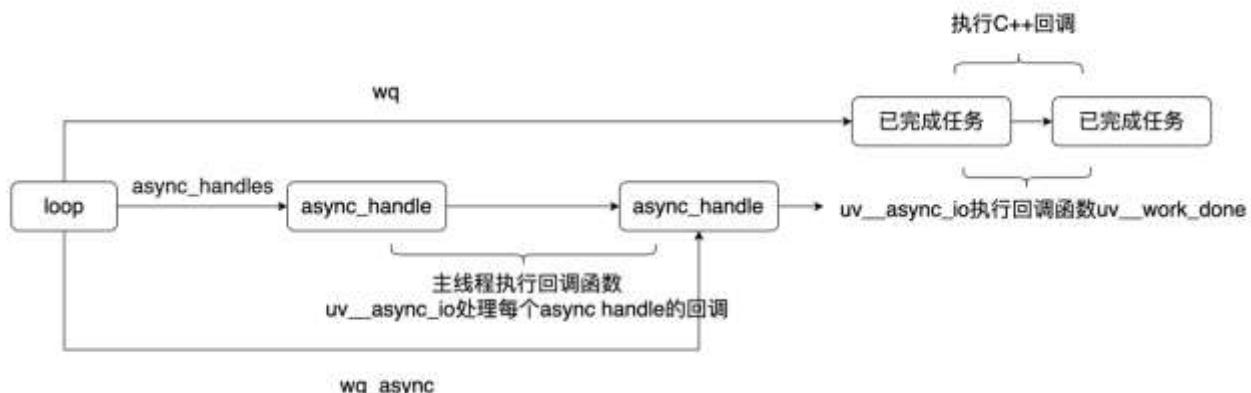


图 4-5

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。

```

1. static int uv_work_cancel(uv_loop_t* loop, uv_req_t* req, struct
   uv_work* w) {
2.     int cancelled;
3.     // 加锁，为了把节点移出队列
4.     uv_mutex_lock(&mutex);
5.     // 加锁，为了判断 w->wq 是否为空

```

```
6.     uv_mutex_lock(&w->loop->wq_mutex);
7.     /*
8.      cancelled 为 true 说明任务还在线程池队列等待处理
9.      1 处理完, w->work == NULL
10.     2 处理中, QUEUE_EMPTY(&w->wq) 为 true, 因
11.        为 worker 在摘下一个任务的时候, 重置 prev 和 next 指针
12.     3 未处理, !QUEUE_EMPTY(&w->wq) 是 true 且 w->work != NULL
13.   */
14.   cancelled = !QUEUE_EMPTY(&w->wq) && w->work != NULL;
15.   // 从线程池任务队列中删除该节点
16.   if (cancelled)
17.     QUEUE_REMOVE(&w->wq);
18.
19.   uv_mutex_unlock(&w->loop->wq_mutex);
20.   uv_mutex_unlock(&mutex);
21.   // 正在执行或者已经执行完了, 则不能取消
22.   if (!cancelled)
23.     return UV_EBUSY;
24.   // 打取消标记, Libuv 执行回调的时候用到
25.   w->work = uv_cancelled;
26.
27.   uv_mutex_lock(&loop->wq_mutex);
28.   /*
29.     插入 loop 的 wq 队列, 对于取消的动作, Libuv 认为是任务执行完了。
30.     所以插入已完成的队列, 执行回调的时候会通知用户该任务的执行结果
31.     是取消, 错误码是 UV_ECANCELED
32.   */
33.   QUEUE_INSERT_TAIL(&loop->wq, &w->wq);
34.   // 通知主线程有任务完成
35.   uv_async_send(&loop->wq_async);
36.   uv_mutex_unlock(&loop->wq_mutex);
37.
38.   return 0;
39. }
```

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`。

第五章 Libuv 流

流的实现在 Libuv 里占了很大的篇幅，是非常核心的逻辑。流的本质是封装了对文件描述符的操作，例如读、写，连接、监听。我们首先看看数据结构，流在 Libuv 里用 `uv_stream_s` 表示，继承于 `uv_handle_s`。

```
1. struct uv_stream_s {
2.     // uv_handle_s 的字段
3.     void* data;
4.     // 所属事件循环
5.     uv_loop_t* loop;
6.     // handle 类型
7.     uv_handle_type type;
8.     // 关闭 handle 时的回调
9.     uv_close_cb close_cb;
10.    // 用于插入事件循环的 handle 队列
11.    void* handle_queue[2];
12.    union {
13.        int fd;
14.        void* reserved[4];
15.    } u;
16.    // 用于插入事件循环的 closing 阶段
17.    uv_handle_t* next_closing;
18.    // 各种标记
19.    unsigned int flags;
20.    // 流拓展的字段
21.    /*
22.     * 户写入流的字节大小，流缓存用户的输入，
23.     * 然后等到可写的时候才执行真正的写
24.     */
25.    size_t write_queue_size;
26.    // 分配内存的函数，内存由用户定义，用来保存读取的数据
27.    uv_alloc_cb alloc_cb;
28.    // 读回调
29.    uv_read_cb read_cb;
30.    // 连接请求对应的结构体
31.    uv_connect_t *connect_req;
32.    /*
33.     * 关闭写端的时候，发送完缓存的数据，
34.     * 执行 shutdown_req 的回调（shutdown_req 在 uv_shutdown 的时候赋值）
35.    */
```

```
35.  */
36. uv_shutdown_t *shutdown_req;
37. /*
38.   流对应的 IO 观察者
39. */
40. uv_io_t io_watcher;
41. // 缓存待写的数据，该字段用于插入队列
42. void* write_queue[2];
43. // 已经完成了数据写入的队列，该字段用于插入队列
44. void* write_completed_queue[2];
45. // 有连接到来并且完成三次握手后，执行的回调
46. uv_connection_cb connection_cb;
47. // 操作流时出错码
48. int delayed_error;
49. // accept 返回的通信 socket 对应的文件描述
50. int accepted_fd;
51. // 同上，用于 IPC 时，缓存多个传递的文件描述符
52. void* queued_fds;
53. }
```

流的实现中，最核心的字段是 IO 观察者，其余的字段是和流的性质相关的。IO 观察者封装了流对应的文件描述符和文件描述符事件触发时的回调。比如读一个流、写一个流、关闭一个流、连接一个流、监听一个流，在 `uv_stream_s` 中都有对应的字段去支持。但是本质上是靠 IO 观察者去驱动的。

- 1 读一个流，就是 IO 观察者中的文件描述符的可读事件触发时，执行用户的读回调。
- 2 写一个流，先把数据写到流中，等到 IO 观察者中的文件描述符可写事件触发时，执行真正的写入，并执行用户的写结束回调。
- 3 关闭一个流，就是 IO 观察者中的文件描述符可写事件触发时，就会执行关闭流的写端。如果流中还有数据没有写完，则先写完（比如发送）后再执行关闭操作，接着执行用户的回调。
- 4 连接流，比如作为客户端去连接服务器。就是 IO 观察者中的文件描述符可读事件触发时（比如建立三次握手成功），执行用户的回调。
- 5 监听流，就是 IO 观察者中的文件描述符可读事件触发时（比如有完成三次握手的连接），执行用户的回调。

下面我们看一下流的具体实现

5.1 初始化流

在使用 `uv_stream_t` 之前需要首先初始化，我们看一下如何初始化一个流。

```

1. void uv_stream_init(uv_loop_t* loop,
2.                      uv_stream_t* stream,
3.                      uv_handle_type type) {
4.     int err;
5.     // 记录 handle 的类型
6.     uv_handle_init(loop, (uv_handle_t*)stream, type);
7.     stream->read_cb = NULL;
8.     stream->alloc_cb = NULL;
9.     stream->close_cb = NULL;
10.    stream->connection_cb = NULL;
11.    stream->connect_req = NULL;
12.    stream->shutdown_req = NULL;
13.    stream->accepted_fd = -1;
14.    stream->queued_fds = NULL;
15.    stream->delayed_error = 0;
16.    QUEUE_INIT(&stream->write_queue);
17.    QUEUE_INIT(&stream->write_completed_queue);
18.    stream->write_queue_size = 0;
19.    /*
20.     初始化 IO 观察者，把文件描述符（这里还没有，所以是-1）和
21.     回调 uv_stream_io 记录在 io_watcher 上，fd 的事件触发时，统一
22.     由 uv_stream_io 函数处理，但也会有特殊情况（下面会讲到）
23.    */
24.    uv_io_init(&stream->io_watcher, uv_stream_io, -1);
25. }

```

初始化一个流的逻辑很简单明了，就是初始化相关的字段，需要注意的是初始化 IO 观察者时，设置的处理函数是 `uv_stream_io`，后面我们会分析这个函数的具体逻辑。

5.2 打开流

```

26. int uv_stream_open(uv_stream_t* stream, int fd, int flags) {
27.     // 还没有设置 fd 或者设置的同一个 fd 则继续，否则返回 UV_EBUSY
28.     if (!(stream->io_watcher.fd == -1 ||
29.           stream->io_watcher.fd == fd))
30.         return UV_EBUSY;
31.     // 设置流的标记
32.     stream->flags |= flags;
33.     // 是 TCP 流则可以设置下面的属性
34.     if (stream->type == UV_TCP) {
35.         // 关闭 nagle 算法

```

```
36.     if ((stream->flags & UV_HANDLE_TCP_NODELAY) &&
37.             uv_tcp_nodelay(fd, 1))
38.         return UV_ERR(errno);
39.     /*
40.         开启 keepalive 机制
41.     */
42.     if ((stream->flags & UV_HANDLE_TCP_KEEPALIVE) &&
43.             uv_tcp_keepalive(fd, 1, 60)) {
44.         return UV_ERR(errno);
45.     }
46. }
47. /*
48. 保存 socket 对应的文件描述符到 I/O 观察者中，Libuv 会在
49. Poll I/O 阶段监听该文件描述符
50. */
51. stream->io_watcher.fd = fd;
52. return 0;
53. }
```

打开一个流，本质上就是给这个流关联一个文件描述符，后续的操作的时候都是基于这个文件描述符的，另外还有一些属性的设置。

5.3 读流

我们在一个流上执行 `uv_read_start` 后，流的数据（如果有的话）就会通过 `read_cb` 回调源源不断地流向调用方。

```
1. int uv_read_start(uv_stream_t* stream,
2.                     uv_alloc_cb alloc_cb,
3.                     uv_read_cb read_cb) {
4.     // 流已经关闭，不能读
5.     if (stream->flags & UV_HANDLE_CLOSING)
6.         return UV_EINVAL;
7.     // 流不可读，说明可能是只写流
8.     if (!(stream->flags & UV_HANDLE_READABLE))
9.         return -ENOTCONN;
10.    // 标记正在读
11.    stream->flags |= UV_HANDLE_READING;
12.    // 记录读回调，有数据的时候会执行这个回调
13.    stream->read_cb = read_cb;
14.    // 分配内存函数，用于存储读取的数据
```

```

15.     stream->alloc_cb = alloc_cb;
16.     // 注册等待读事件
17.     uv__io_start(stream->loop, &stream->io_watcher, POLLIN);
18.     // 激活 handle, 有激活的 handle, 事件循环不会退出
19.     uv__handle_start(stream);
20.     return 0;
21. }
```

执行 `uv_read_start` 本质上是给流对应的文件描述符在 `epoll` 中注册了一个等待可读事件，并记录相应的上下文，比如读回调函数，分配内存的函数。接着打上正在做读取操作的标记。当可读事件触发的时候，读回调就会被执行，除了读取数据，还有一个读操作就是停止读取。对应的函数是 `uv_read_stop`。

```

1. int uv_read_stop(uv_stream_t* stream) {
2.     // 是否正在执行读取操作, 如果不是, 则没有必要停止
3.     if (!(stream->flags & UV_HANDLE_READING))
4.         return 0;
5.     // 清除正在读取的标记
6.     stream->flags &= ~UV_HANDLE_READING;
7.     // 撤销等待读事件
8.     uv__io_stop(stream->loop, &stream->io_watcher, POLLIN);
9.     // 对写事件也不感兴趣, 停掉 handle。允许事件循环退出
10.    if (!uv__io_active(&stream->io_watcher, POLLOUT))
11.        uv__handle_stop(stream);
12.    stream->read_cb = NULL;
13.    stream->alloc_cb = NULL;
14.    return 0;
15. }
```

另外还有一个辅助函数，判断流是否设置了可读属性。

```

1. int uv_is_readable(const uv_stream_t* stream) {
2.     return !(stream->flags & UV_HANDLE_READABLE);
3. }
```

上面的函数只是注册和注销读事件，如果可读事件触发的时候，我们还需要自己去读取数据，我们看一下真正的读逻辑

```

1. static void uv__read(uv_stream_t* stream) {
2.     uv_buf_t buf;
3.     ssize_t nread;
```

```
4.     struct msghdr msg;
5.     char cmsg_space[CMSG_SPACE(UV__CMSG_FD_SIZE)];
6.     int count;
7.     int err;
8.     int is_ipc;
9.     // 清除读取部分标记
10.    stream->flags &= ~UV_STREAM_READ_PARTIAL;
11.    count = 32;
12.    /*
13.        流是 Unix 域类型并且用于 IPC, Unix 域不一定用于 IPC,
14.        用作 IPC 可以支持传递文件描述符
15.    */
16.    is_ipc = stream->type == UV_NAMED_PIPE &&
17.             ((uv_pipe_t*) stream)->ipc;
18.    // 设置了读回调, 正在读, count 大于 0
19.    while (stream->read_cb
20.           && (stream->flags & UV_STREAM_READING)
21.           && (count-- > 0)) {
22.        buf = uv_buf_init(NULL, 0);
23.        // 调用调用方提供的分配内存函数, 分配内存承载数据
24.        stream->alloc_cb((uv_handle_t*) stream, 64 * 1024, &buf);
25.        /*
26.            不是 IPC 则直接读取数据到 buf, 否则用 recvmsg 读取数据
27.            和传递的文件描述符 (如果有的话)
28.        */
29.        if (!is_ipc) {
30.            do {
31.                nread = read(uv_stream_fd(stream),
32.                             buf.base,
33.                             buf.len);
34.            }
35.            while (nread < 0 && errno == EINTR);
36.        } else {
37.            /* ipc uses recvmsg */
38.            msg.msg_flags = 0;
39.            msg.msg_iov = (struct iovec*) &buf;
40.            msg.msg iovlen = 1;
41.            msg.msg_name = NULL;
42.            msg.msg_namelen = 0;
43.            msg.msg_controllen = sizeof(cmsg_space);
44.            msg.msg_control = cmsg_space;
45.            do {
```

```

46.             nread = uv_recvmsg(uv_stream_fd(stream), &msg, 0);
47.         }
48.         while (nread < 0 && errno == EINTR);
49.     }
50.     // 读失败
51.     if (nread < 0) {
52.         // 读繁忙
53.         if (errno == EAGAIN || errno == EWOULDBLOCK) {
54.             // 执行读回调
55.             stream->read_cb(stream, 0, &buf);
56.         } else {
57.             /* Error. User should call uv_close(). */
58.             // 读失败
59.             stream->read_cb(stream, -errno, &buf);
60.         }
61.         return;
62.     } else if (nread == 0) {
63.         // 读到结尾了
64.         uv_stream_eof(stream, &buf);
65.         return;
66.     } else {
67.         // 读成功, 读取数据的长度
68.         ssize_t buflen = buf.len;
69.         /*
70.             是 IPC 则解析读取的数据, 把文件描述符解析出来,
71.             放到 stream 的 accepted_fd 和 queued_fds 字段
72.         */
73.         if (is_ipc) {
74.             err = uv_stream_recv_cmsg(stream, &msg);
75.             if (err != 0) {
76.                 stream->read_cb(stream, err, &buf);
77.                 return;
78.             }
79.         }
80.         // 执行读回调
81.         stream->read_cb(stream, nread, &buf);
82.     }
83. }
84. }
```

uv_read 除了可以读取一般的数据外，还支持读取传递的文件描述符。我们看一下描述符传递的原理。我们知道，父进程 fork 出子进程的时候，子进程是继承父进程的文件描述符列表的。我们看一下进程和文件描述符的关系。

fork 之前如图 5-1 所示。

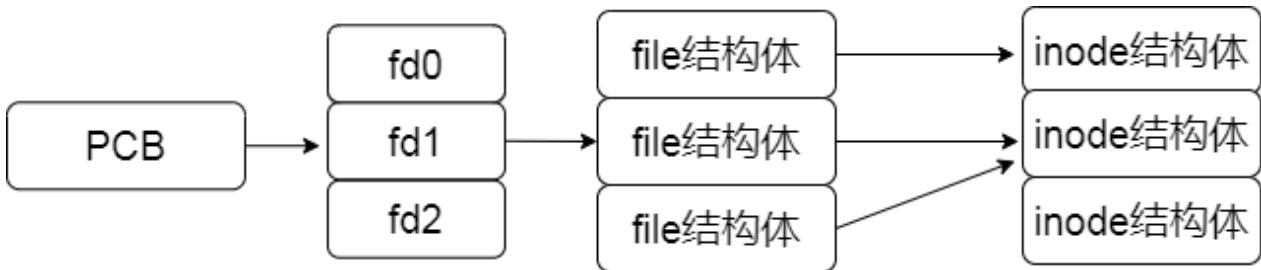


图 5-1

我们再看一下 fork 之后的结构如图 5-2 所示。

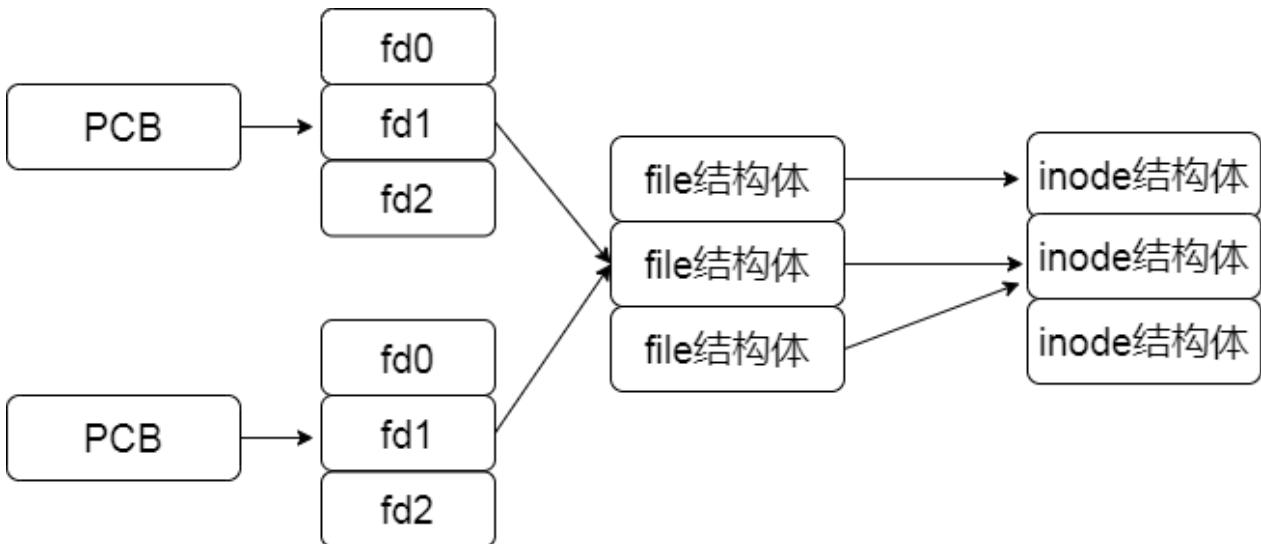


图 5-2

如果父进程或者子进程在 fork 之后创建了新的文件描述符，那父子进程间就不能共享了，假设父进程要把一个文件描述符传给子进程，那怎么办呢？根据进程和文件描述符的关系。传递文件描述符要做的事情，不仅是在子进程中新建一个 fd，还要建立起 fd->file->inode 的关联，不过我们不需要关注这些，因为操作系统都帮我们处理了，我们只需要通过 sendmsg 把想传递的文件描述符发送给 Unix 域的另一端。Unix 域另一端就可以通过 recvmsg 把文件描述符从数据中读取出来。接着使用 uv_stream_recv_cmsg 函数保存数据里解析出来的文件描述符。

```

1. static int uv_stream_recv_cmsg(uv_stream_t* stream,
2.                                 struct msghdr* msg) {
3.     struct cmsghdr* cmsg;
4.     // 遍历 msg
5.     for (cmsg = CMSG_FIRSTHDR(msg);
6.          cmsg != NULL;
7.          cmsg = CMSG_NXTHDR(msg, cmsg)) {
8.         char* start;
9.         char* end;
10.        int err;
11.        void* pv;
12.        int* pi;
13.        unsigned int i;
14.        unsigned int count;
15.
16.        pv = CMSG_DATA(cmsg);
17.        pi = pv;
18.        start = (char*) cmsg;
19.        end = (char*) cmsg + cmsg->cmsg_len;
20.        count = 0;
21.        while (start + CMSG_LEN(count * sizeof(*pi)) < end)
22.            count++;
23.        for (i = 0; i < count; i++) {
24.            /*
25.             accepted_fd 代表当前待处理的文件描述符,
26.             如果已经有值则剩余描述符就通过 uv_stream_queue_fd 排队
27.             如果还没有值则先赋值
28.            */
29.            if (stream->accepted_fd != -1) {
30.                err = uv_stream_queue_fd(stream, pi[i]);
31.            } else {
32.                stream->accepted_fd = pi[i];
33.            }
34.        }
35.    }
36.
37.    return 0;
38. }
```

`uv_stream_recv_cmsg` 会从数据中解析出一个个文件描述符存到 `stream` 中，第一个文件描述符保存在 `accepted_fd`，剩下的使用 `uv_stream_queue_fd` 处理。

```

1. struct uv_stream_queued_fds_s {
2.     unsigned int size;
3.     unsigned int offset;
```

```
4.     int fds[1];
5. };
6.
7. static int uv_stream_queue_fd(uv_stream_t* stream, int fd) {
8.     uv_stream_queued_fds_t* queued_fds;
9.     unsigned int queue_size;
10.    // 原来的内存
11.    queued_fds = stream->queued_fds;
12.    // 没有内存，则分配
13.    if (queued_fds == NULL) {
14.        // 默认 8 个
15.        queue_size = 8;
16.        /*
17.         * 一个元数据内存+多个 fd 的内存
18.         * (前面加*代表解引用后的值的类型所占的内存大小,
19.         * 减一是因为 uv_stream_queued_fds_t
20.         * 结构体本身有一个空间)
21.        */
22.        queued_fds = uv_malloc((queue_size - 1) *
23.                               sizeof(*queued_fds->fds) +
24.                               sizeof(*queued_fds));
25.        if (queued_fds == NULL)
26.            return UV_ENOMEM;
27.        // 容量
28.        queued_fds->size = queue_size;
29.        // 已使用个数
30.        queued_fds->offset = 0;
31.        // 指向可用的内存
32.        stream->queued_fds = queued_fds;
33.        // 之前的内存用完了，扩容
34.    } else if (queued_fds->size == queued_fds->offset) {
35.        // 每次加 8 个
36.        queue_size = queued_fds->size + 8;
37.        queued_fds = uv_realloc(queued_fds,
38.                               (queue_size - 1) * sizeof(*queued_fds->fds) +
39.                               sizeof(*queued_fds));
40.        if (queued_fds == NULL)
41.            return UV_ENOMEM;
42.        // 更新容量大小
43.        queued_fds->size = queue_size;
44.        // 保存新的内存
45.        stream->queued_fds = queued_fds;
46.    }
47.
```

```

48. /* Put fd in a queue */
49. // 保存 fd
50. queued_fds->fds[queued_fds->offset++] = fd;
51.
52. return 0;
53. }

```

内存结构如图 5-3 所示。

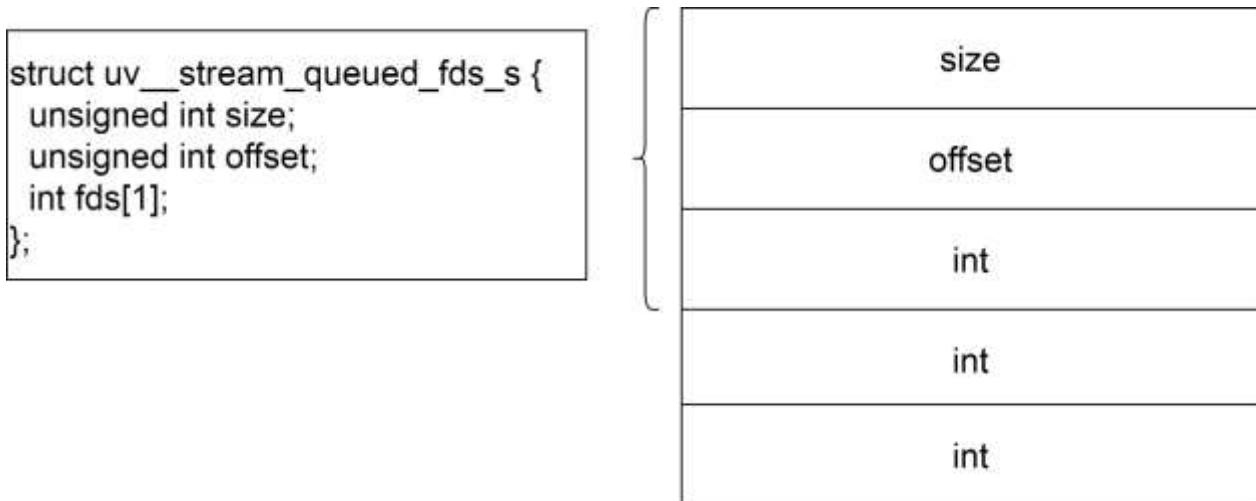


图 5-3

最后我们看一下读结束后的处理,

```

1. static void uv_stream_eof(uv_stream_t* stream,
                           const uv_buf_t* buf) {
2.
3.     // 打上读结束标记
4.     stream->flags |= UV_STREAM_READ_EOF;
5.     // 注销等待可读事件
6.     uv_io_stop(stream->loop, &stream->io_watcher, POLLIN);
7.     // 没有注册等待可写事件则停掉 handle, 否则会影响事件循环的退出
8.     if (!uv_io_active(&stream->io_watcher, POLLOUT))
9.         uv_handle_stop(stream);
10.    uv_stream_osx_interrupt_select(stream);
11.    // 执行读回调
12.    stream->read_cb(stream, UV_EOF, buf);
13.    // 清除正在读标记
14.    stream->flags &= ~UV_STREAM_READING;
15. }

```

我们看到，流结束的时候，首先注销等待可读事件，然后通过回调通知上层。

5.4 写流

我们在流上执行 uv_write 就可以往流中写入数据。

```
1. int uv_write(
2.                 /*
3.                  一个写请求，记录了需要写入的数据和信息。
4.                  数据来自下面的 const uv_buf_t bufs[]
5.                 */
6.                 uv_write_t* req,
7.                 // 往哪个流写
8.                 uv_stream_t* handle,
9.                 // 需要写入的数据
10.                const uv_buf_t bufs[],
11.                // uv_buf_t 个数
12.                unsigned int nbufs,
13.                // 写完后执行的回调
14.                uv_write_cb cb
15. ) {
16.     return uv_write2(req, handle, bufs, nbufs, NULL, cb);
17. }
```

uv_write 是直接调用 uv_write2。第四个参数是 NULL。代表是一般的写数据，不传递文件描述符。

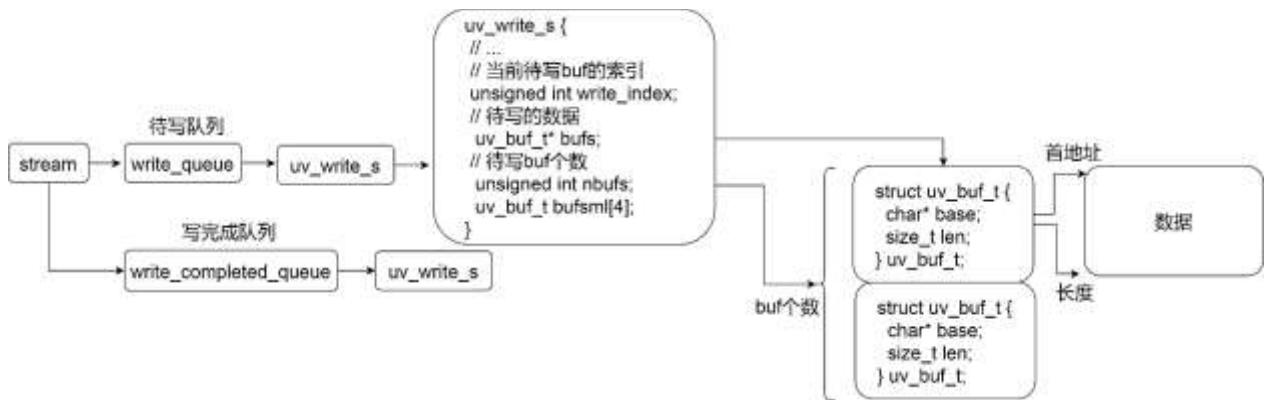
```
1. int uv_write2(uv_write_t* req,
2.                 uv_stream_t* stream,
3.                 const uv_buf_t bufs[],
4.                 unsigned int nbufs,
5.                 uv_stream_t* send_handle,
6.                 uv_write_cb cb) {
7.     int empty_queue;
8.     // 待发送队列是否为空
9.     empty_queue = (stream->write_queue_size == 0);
10.    // 构造一个写请求
11.    uv_req_init(stream->loop, req, UV_WRITE);
12.    // 写请求对应的回调
13.    req->cb = cb;
```

```

14. // 写请求对应的流
15. req->handle = stream;
16. req->error = 0;
17. // 需要发送的文件描述符，也可以是 NULL 说明不需要发送文件描述符
18. req->send_handle = send_handle;
19. QUEUE_INIT(&req->queue);
20. // bufs 指向待写的数据
21. req->bufs = req->bufsml;
22. // 复制调用方的数据过来
23. memcpy(req->bufs, bufs, nbufts * sizeof(bufs[0]));
24. // buf 个数
25. req->nbufts = nbufts;
26. // 当前写成功的 buf 索引，针对 bufs 数组
27. req->write_index = 0;
28. // 待写的数据大小 = 之前的大小 + 本次大小
29. stream->write_queue_size += uv_count_bufs(bufs, nbufts);
30. // 插入待写队列
31. QUEUE_INSERT_TAIL(&stream->write_queue, &req->queue);
32. // 非空说明正在连接，还不能写，比如 TCP 流
33. if (stream->connect_req) {
34.     /* Still connecting, do nothing. */
35. }
36. else if (empty_queue) { // 当前待写队列为空，直接写
37.     uv_write(stream);
38. }
39. else {
40.     // 还有数据没有写完，注册等待写事件
41.     uv_io_start(stream->loop, &stream->io_watcher, POLLOUT);
42.     uv_stream_osx_interrupt_select(stream);
43. }
44. return 0;
45. }

```

uv_write2 的主要逻辑就是封装一个写请求，插入到流的待写队列。然后根据当前流的情况。看是直接写入还是等待会再写入。关系图大致如图 5-4 所示。



uv_write2 只是对写请求进行一些预处理，真正执行写的函数是 uv_write

```

1. static void uv_write(uv_stream_t* stream) {
2.     struct iovec* iov;
3.     QUEUE* q;
4.     uv_write_t* req;
5.     int iovmax;
6.     int iovcnt;
7.     ssize_t n;
8.     int err;
9.
10. start:
11.     // 没有数据需要写
12.     if (QUEUE_EMPTY(&stream->write_queue))
13.         return;
14.     q = QUEUE_HEAD(&stream->write_queue);
15.     req = QUEUE_DATA(q, uv_write_t, queue);
16.     // 从哪里开始写
17.     iov = (struct iovec*) &(req->bufs[req->write_index]);
18.     // 还有多少没写
19.     iovcnt = req->nbufs - req->write_index;
20.     // 最多可以写多少
21.     iovmax = uv_getiovmax();
22.     // 取最小值
23.     if (iovcnt > iovmax)
24.         iovcnt = iovmax;
25.     // 需要传递文件描述符
26.     if (req->send_handle) {
27.         int fd_to_send;
28.         struct msghdr msg;

```

```

29.         struct cmsghdr *cmsg;
30.         union {
31.             char data[64];
32.             struct cmsghdr alias;
33.         } scratch;
34.
35.         if (uv__is_closing(req->send_handle)) {
36.             err = -EBADF;
37.             goto error;
38.         }
39.         // 待发送的文件描述符
40.         fd_to_send = uv__handle_fd((uv_handle_t*) req->send_handle);
41.         memset(&scratch, 0, sizeof(scratch));
42.
43.         msg.msg_name = NULL;
44.         msg.msg_namelen = 0;
45.         msg.msg iov = iov;
46.         msg.msg iovlen = iovcnt;
47.         msg.msg_flags = 0;
48.
49.         msg.msg_control = &scratch.alias;
50.         msg.msg_controllen = CMSG_SPACE(sizeof(fd_to_send));
51.
52.         cmsg = CMSG_FIRSTHDR(&msg);
53.         cmsg->cmsg_level = SOL_SOCKET;
54.         cmsg->cmsg_type = SCM_RIGHTS;
55.         cmsg->cmsg_len = CMSG_LEN(sizeof(fd_to_send));
56.
57.     {
58.         void* pv = CMSG_DATA(cmsg);
59.         int* pi = pv;
60.         *pi = fd_to_send;
61.     }
62.
63.     do {
64.         // 使用 sendmsg 函数发送文件描述符
65.         n = sendmsg(uv__stream_fd(stream), &msg, 0);
66.     }
67.     while (n == -1 && errno == EINTR);
68. } else {
69.     do {
70.         // 写一个或者写批量写

```

```
71.         if (iovcnt == 1) {
72.             n = write(uv_stream_fd(stream),
73.                       iov[0].iov_base,
74.                       iov[0].iov_len);
75.         } else {
76.             n = writev(uv_stream_fd(stream), iov, iovcnt);
77.         }
78.     }
79.     while (n == -1 && errno == EINTR);
80. }
81. // 写失败
82. if (n < 0) {
83. /*
84. 不是写繁忙，则报错，
85. 否则如果设置了同步写标记，则继续尝试写
86. */
87. if (errno != EAGAIN &&
88.     errno != EWOULDBLOCK &&
89.     errno != ENOBUFS) {
90.     err = -errno;
91.     goto error;
92. } else if (stream->flags & UV_STREAM_BLOCKING) {
93.     /* If this is a blocking stream, try again. */
94.     goto start;
95. }
96. } else {
97. // 写成功
98.     while (n >= 0) {
99.         // 当前 buf 首地址
100.        uv_buf_t* buf = &(req->bufs[req->write_index]);
101.        // 当前 buf 的数据长度
102.        size_t len = buf->len;
103.        // 小于说明当前 buf 还没有写完（还没有被消费完）
104.        if ((size_t)n < len) {
105.            // 更新待写的首地址
106.            buf->base += n;
107.            // 更新待写的数据长度
108.            buf->len -= n;
109.        }
110.        // 更新待写队列的长度，这个队列是待写数据的
111.        // 总长度，等于多个 buf 的和
112.    }*/

```

```

113.         stream->write_queue_size -= n;
114.         n = 0;
115.         /*
116.          还没写完, 设置了同步写, 则继续尝试写,
117.          否则退出, 注册待写事件
118.         */
119.         if (stream->flags & UV_STREAM_BLOCKING) {
120.             goto start;
121.         } else {
122.             break;
123.         }
124.     } else {
125.         /*
126.          当前 buf 的数据都写完了, 则更新待写数据的首
127.          地址, 即下一个 buf, 因为当前 buf 写完了
128.         */
129.         req->write_index++;
130.         // 更新 n, 用于下一个循环的计算
131.         n -= len;
132.         // 更新待写队列的长度
133.         stream->write_queue_size -= len;
134.         /*
135.          等于最后一个 buf 了, 说明待写队列的数据
136.          都写完了
137.         */
138.         if (req->write_index == req->nbufs) {
139.             /*
140.              释放 buf 对应的内存, 并把请求插入写完成
141.              队列, 然后准备触发写完成回调
142.             */
143.             uv__write_req_finish(req);
144.             return;
145.         }
146.     }
147. }
148. }
149. /*
150.  写成功, 但是还没有写完, 注册待写事件,
151.  等待可写的时候继续写
152. */
153. uv__io_start(stream->loop, &stream->io_watcher, POLLOUT);
154. uv__stream_osx_interrupt_select(stream);

```

```
155.  
156.     return;  
157. // 写出错  
158. error:  
159.     // 记录错误  
160.     req->error = err;  
161.     /*  
162.         释放内存，丢弃数据，插入写完成队列，  
163.         把 IO 观察者插入 pending 队列，等待 pending 阶段执行回调  
164.     */  
165.     uv_write_req_finish(req);  
166.     // 注销待写事件  
167.     uv_io_stop(stream->loop, &stream->io_watcher, POLLOUT);  
168.     // 如果也没有注册等待可读事件，则把 handle 关闭  
169.     if (!uv_io_active(&stream->io_watcher, POLLIN))  
170.         uv_handle_stop(stream);  
171.     uv_stream_osx_interrupt_select(stream);  
172. }
```

我们看下一个写请求结束后（成功或者失败），Libuv 如何处理的。逻辑在 `uv_write_req_finish` 函数。

```
1. static void uv_write_req_finish(uv_write_t* req) {  
2.     uv_stream_t* stream = req->handle;  
3.     // 从待写队列中移除  
4.     QUEUE_REMOVE(&req->queue);  
5.     // 写成功，并且分配了额外的堆内存，则需要释放，见 uv_write  
6.     if (req->error == 0) {  
7.         if (req->bufs != req->bufsml)  
8.             uv_free(req->bufs);  
9.         req->bufs = NULL;  
10.    }  
11.    // 插入写完成队列  
12.    QUEUE_INSERT_TAIL(&stream->write_completed_queue, &req->queue);  
13.    /*  
14.        把 IO 观察者插入 pending 队列，Libuv 在处理 pending 阶段时，  
15.        会触发 IO 观察者的写事件  
16.    */  
17.    uv_io_feed(stream->loop, &stream->io_watcher);  
18. }
```

uv_write_req_finish 的逻辑比较简单

1 把节点从待写队列中移除

2 req->bufs != req->bufsm1 不相等说明分配了堆内存，需要自己释放

3 并把请求插入写完成队列，把 IO 观察者插入 pending 队列，等待 pending 阶段执行回调，在 pending 节点会执行 IO 观察者的回调（uv_stream_io）。我们看一下 uv_stream_io 如何处理的，下面是具体的处理逻辑。

```

1. // 可写事件触发
2. if (events & (POLLOUT | POLLERR | POLLHUP)) {
3.     // 继续执行写
4.     uv_write(stream);
5.     // 处理写成功回调
6.     uv_write_callbacks(stream);
7.     // 待写队列空，注销等待可写事件，即不需要写了
8.     if (QUEUE_EMPTY(&stream->write_queue))
9.         uv_drain(stream);
10. }
```

我们只关注 uv_write_callbacks。

```

1. static void uv_write_callbacks(uv_stream_t* stream) {
2.     uv_write_t* req;
3.     QUEUE* q;
4.     // 写完成队列非空
5.     while (!QUEUE_EMPTY(&stream->write_completed_queue)) {
6.         q = QUEUE_HEAD(&stream->write_completed_queue);
7.         req = QUEUE_DATA(q, uv_write_t, queue);
8.         QUEUE_REMOVE(q);
9.         uv_req_unregister(stream->loop, req);
10.        // bufs 的内存还没有被释放
11.        if (req->bufs != NULL) {
12.            // 更新待写队列的大小，即减去 req 对应的所有数据的大小
13.            stream->write_queue_size -= uv_write_req_size(req);
14.            /*
15.             bufs 默认指向 bufsm1，超过默认大小时，
16.             bufs 指向新申请的堆内存，所以需要释放
17.             */
18.            if (req->bufs != req->bufsm1)
19.                uv_free(req->bufs);
```

```
20.         req->bufs = NULL;
21.     }
22.     // 执行回调
23.     if (req->cb)
24.         req->cb(req, req->error);
25.     }
26. }
```

uv_write_callbacks 负责更新流的待写队列大小、释放额外申请的堆内存、执行每个写请求的回调。

5.5 关闭流的写端

```
1. // 关闭流的写端
2. int uv_shutdown(uv_shutdown_t* req,
3.                 uv_stream_t* stream,
4.                 uv_shutdown_cb cb) {
5.     // 初始化一个关闭请求，关联的 handle 是 stream
6.     uv_req_init(stream->loop, req, UV_SHUTDOWN);
7.     req->handle = stream;
8.     // 关闭后执行的回调
9.     req->cb = cb;
10.    stream->shutdown_req = req;
11.    // 设置正在关闭的标记
12.    stream->flags |= UV_HANDLE_SHUTTING;
13.    // 注册等待可写事件
14.    uv_io_start(stream->loop, &stream->io_watcher, POLLOUT);
15.    return 0;
16. }
```

关闭流的写端就是相当于给流发送一个关闭请求，把请求挂载到流中，然后注册等待可写事件，在可写事件触发的时候就会执行关闭操作。在分析写流的章节中我们提到，可写事件触发的时候，会执行 uv_drain 注销等待可写事件，除此之外，uv_drain 还做了一个事情，就是关闭流的写端。我们看看具体的逻辑。

```
1. static void uv_drain(uv_stream_t* stream) {
2.     uv_shutdown_t* req;
3.     int err;
4.     // 撤销等待可写事件，因为没有数据需要写入了
5.     uv_io_stop(stream->loop, &stream->io_watcher, POLLOUT);
6.     uv_stream_osx_interrupt_select(stream);
7. }
```

```

8. // 设置了关闭写端，但是还没有关闭，则执行关闭写端
9. if ((stream->flags & UV_HANDLE_SHUTTING) &&
10.    !(stream->flags & UV_HANDLE_CLOSING) &&
11.    !(stream->flags & UV_HANDLE_SHUT)) {
12.    req = stream->shutdown_req;
13.    stream->shutdown_req = NULL;
14.    // 清除标记
15.    stream->flags &= ~UV_HANDLE_SHUTTING;
16.    uv_req_unregister(stream->loop, req);
17.
18.    err = 0;
19.    // 关闭写端
20.    if (shutdown(uv_stream_fd(stream), SHUT_WR))
21.        err = UV_ERR(errno);
22.    // 标记已关闭写端
23.    if (err == 0)
24.        stream->flags |= UV_HANDLE_SHUT;
25.    // 执行回调
26.    if (req->cb != NULL)
27.        req->cb(req, err);
28. }
29. }
```

通过调用 `shutdown` 关闭流的写端，比如 TCP 流发送完数据后可以关闭写端。但是仍然可以读。

5.6 关闭流

```

1. void uv_stream_close(uv_stream_t* handle) {
2.     unsigned int i;
3.     uv_stream_queued_fds_t* queued_fds;
4.     // 从事件循环中删除 I/O 观察者，移出 pending 队列
5.     uv_io_close(handle->loop, &handle->io_watcher);
6.     // 停止读
7.     uv_read_stop(handle);
8.     // 停掉 handle
9.     uv_handle_stop(handle);
10.    // 不可读、写
11.    handle->flags &= ~(UV_HANDLE_READABLE | UV_HANDLE_WRITABLE);
12.    // 关闭非标准流的文件描述符
13.    if (handle->io_watcher.fd != -1) {
14.        /*
15.         Don't close stdio file descriptors.
```

```
16.         Nothing good comes from it.
17.         */
18.         if (handle->io_watcher.fd > STDERR_FILENO)
19.             uv_close(handle->io_watcher.fd);
20.         handle->io_watcher.fd = -1;
21.     }
22.     // 关闭通信 socket 对应的文件描述符
23.     if (handle->accepted_fd != -1) {
24.         uv_close(handle->accepted_fd);
25.         handle->accepted_fd = -1;
26.     }
27.     // 同上，这是在排队等待处理的文件描述符
28.     if (handle->queued_fds != NULL) {
29.         queued_fds = handle->queued_fds;
30.         for (i = 0; i < queued_fds->offset; i++)
31.             uv_close(queued_fds->fds[i]);
32.         uv_free(handle->queued_fds);
33.         handle->queued_fds = NULL;
34.     }
35. }
```

关闭流就是把流注册在 epoll 的事件注销，关闭所持有的文件描述符。

5.7 连接流

连接流是针对 TCP 和 Unix 域的，所以我们首先介绍一下一些网络编程相关的内容，首先我们先要有一个 socket。我们看 Libuv 中如何新建一个 socket。

```
1. int uv_socket(int domain, int type, int protocol) {
2.     int sockfd;
3.     int err;
4.     // 新建一个 socket，并设置非阻塞和 LOEXEC 标记
5.     sockfd = socket(domain, type | SOCK_NONBLOCK | SOCK_CLOEXEC, protocol);
6.     // 不触发 SIGPIPE 信号，比如对端已经关闭，本端又执行写
7. #if defined(SO_NOSIGPIPE)
8.     {
9.         int on = 1;
10.        setsockopt(sockfd, SOL_SOCKET, SO_NOSIGPIPE, &on, sizeof(on));
11.    }
12. #endif
13.
14.    return sockfd;
15. }
```

在 Libuv 中，socket 的模式都是非阻塞的，uv_socket 是 Libuv 中申请 socket 的函数，不过 Libuv 不直接调用该函数，而是封装了一下。

```

1. /*
2. 1 获取一个新的 socket fd
3. 2 把 fd 保存到 handle 里，并根据 flag 进行相关设置
4. 3 绑定到本机随意的地址（如果设置了该标记的话）
5. */
6. static int new_socket(uv_tcp_t* handle,
7.                      int domain,
8.                      unsigned long flags) {
9.     struct sockaddr_storage saddr;
10.    socklen_t slen;
11.    int sockfd;
12.    // 获取一个 socket
13.    sockfd = uv_socket(domain, SOCK_STREAM, 0);
14.
15.    // 设置选项和保存 socket 的文件描述符到 IO 观察者中
16.    uv_stream_open((uv_stream_t*) handle, sockfd, flags);
17.    // 设置了需要绑定标记 UV_HANDLE_BOUND
18.    if (flags & UV_HANDLE_BOUND) {
19.        slen = sizeof(saddr);
20.        memset(&saddr, 0, sizeof(saddr));
21.        // 获取 fd 对应的 socket 信息，比如 IP，端口，可能没有
22.        getsockname(uv_stream_fd(handle),
23.                    (struct sockaddr*) &saddr,
24.                    &slen);
25.
26.        // 绑定到 socket 中，如果没有则绑定到系统随机选择的地址
27.        bind(uv_stream_fd(handle), (struct sockaddr*) &saddr, slen);
28.    }
29.
30.    return 0;
31. }
```

上面的代码就是在 Libuv 申请一个 socket 的逻辑，另外它还支持新建的 socket，可以绑定到一个用户设置的，或者操作系统随机选择的地址。不过 Libuv 并不直接使用这个函数。而是又封装了一层。

```

1. // 如果流还没有对应的 fd，则申请一个新的，如果有则修改流的配置
2. static int maybe_new_socket(uv_tcp_t* handle,
3.                             int domain,
4.                             unsigned long flags) {
5.     struct sockaddr_storage saddr;
6.     socklen_t slen;
```

```
7.
8.    // 已经有 fd 了
9.    if (uv_stream_fd(handle) != -1) {
10.        // 该流需要绑定到一个地址
11.        if (flags & UV_HANDLE_BOUND) {
12.            /*
13.                流是否已经绑定到一个地址了。handle 的 flag 是在
14.                new_socket 里设置的，如果有这个标记说明已经执行过绑定了，
15.                直接更新 flags 就行。
16.            */
17.            if (handle->flags & UV_HANDLE_BOUND) {
18.                handle->flags |= flags;
19.                return 0;
20.            }
21.            // 有 fd，但是可能还没绑定到一个地址
22.            slen = sizeof(saddr);
23.            memset(&saddr, 0, sizeof(saddr));
24.            // 获取 socket 绑定到的地址
25.            if (getsockname(uv_stream_fd(handle),
26.                            (struct sockaddr*) &saddr,
27.                            &slen))
28.                return UV_ERR(errno);
29.            // 绑定过了 socket 地址，则更新 flags 就行
30.            if ((saddr.ss_family == AF_INET6 &&
31.                 ((struct sockaddr_in6*) &saddr)->sin6_port != 0) ||
32.                 (saddr.ss_family == AF_INET &&
33.                 ((struct sockaddr_in*) &saddr)->sin_port != 0)) {
34.                handle->flags |= flags;
35.                return 0;
36.            }
37.            // 没绑定则绑定到随机地址，bind 中实现
38.            if (bind(uv_stream_fd(handle),
39.                     (struct sockaddr*) &saddr,
40.                     slen))
41.                return UV_ERR(errno);
42.        }
43.
44.        handle->flags |= flags;
45.        return 0;
46.    }
47.    // 申请一个新的 fd 关联到流
48.    return new_socket(handle, domain, flags);
```

| 49. }

maybe_new_socket 函数的逻辑分支很多，主要如下

1 如果流还没有关联到 fd，则申请一个新的 fd 关联到流上

2 如果流已经关联了一个 fd。

如果流设置了绑定地址的标记，但是已经通过 Libuv 绑定了一个地址（Libuv 会设置 UV_HANDLE_BOUND 标记，用户也可能是直接调 bind 函数绑定了）。则不需要再次绑定，更新 flags 就行。

如果流设置了绑定地址的标记，但是还没有通过 Libuv 绑定一个地址，这时候通过 getsocketname 判断用户是否自己通过 bind 函数绑定了一个地址，是的话则不需要再次执行绑定操作。否则随机绑定到一个地址。

以上两个函数的逻辑主要是申请一个 socket 和给 socket 绑定一个地址。下面我们开看一下连接流的实现。

```

1. int uv_tcp_connect(uv_connect_t* req,
2.                      uv_tcp_t* handle,
3.                      const struct sockaddr* addr,
4.                      unsigned int addrlen,
5.                      uv_connect_cb cb) {
6.     int err;
7.     int r;
8.
9.     // 已经发起了 connect 了
10.    if (handle->connect_req != NULL)
11.        return UV_EALREADY;
12.    // 申请一个 socket 和绑定一个地址，如果还没有的话
13.    err = maybe_new_socket(handle, addr->sa_family,
14.                           UV_HANDLE_READABLE | UV_HANDLE_WRITABLE);
15.    if (err)
16.        return err;
17.    handle->delayed_error = 0;
18.
19.    do {
20.        // 清除全局错误变量的值
21.        errno = 0;
22.        // 非阻塞发起三次握手
23.        r = connect(uv_stream_fd(handle), addr, addrlen);

```

```
24.     } while (r == -1 && errno == EINTR);  
25.  
26.     if (r == -1 && errno != 0) {  
27.         // 三次握手还没有完成  
28.         if (errno == EINPROGRESS)  
29.             ; /* not an error */  
30.         else if (errno == ECONNREFUSED)  
31.             // 对方拒绝建立连接，延迟报错  
32.             handle->delayed_error = UV_ERR(errno);  
33.         else  
34.             // 直接报错  
35.             return UV_ERR(errno);  
36.     }  
37.     // 初始化一个连接型 request，并设置某些字段  
38.     uv_req_init(handle->loop, req, UV_CONNECT);  
39.     req->cb = cb;  
40.     req->handle = (uv_stream_t*) handle;  
41.     QUEUE_INIT(&req->queue);  
42.     // 连接请求  
43.     handle->connect_req = req;  
44.     // 注册到 Libuv 观察者队列  
45.     uv_io_start(handle->loop, &handle->io_watcher, POLLOUT);  
46.     // 连接出错，插入 pending 队尾  
47.     if (handle->delayed_error)  
48.         uv_io_feed(handle->loop, &handle->io_watcher);  
49.  
50.     return 0;  
51. }
```

连接流的逻辑，大致如下

- 1 申请一个 socket，绑定一个地址。
- 2 根据给定的服务器地址，发起三次握手，非阻塞的，会直接返回继续执行，不会等到三次握手完成。
- 3 往流上挂载一个 connect 型的请求。
- 4 设置 IO 观察者感兴趣的事件为可写。然后把 IO 观察者插入事件循环的 IO 观察者队列。等待可写的时候时候（完成三次握手），就会执行 cb 回调。

可写事件触发时，会执行 uv_stream_io，我们看一下具体的逻辑。

```
1. if (stream->connect_req) {  
2.     uv_stream_connect(stream);  
3.     return;
```

4. }

我们继续看 `uv_stream_connect`。

```

1. static void uv_stream_connect(uv_stream_t* stream) {
2.     int error;
3.     uv_connect_t* req = stream->connect_req;
4.     socklen_t errorsize = sizeof(int);
5.     // 连接出错
6.     if (stream->delayed_error) {
7.         error = stream->delayed_error;
8.         stream->delayed_error = 0;
9.     } else {
10.         // 还是需要判断一下是不是出错了
11.         getsockopt(uv_stream_fd(stream),
12.                     SOL_SOCKET,
13.                     SO_ERROR,
14.                     &error,
15.                     &errorsize);
16.         error = UV_ERR(error);
17.     }
18.     // 还没连接成功，先返回，等待下次可写事件的触发
19.     if (error == UV_ERR(EINPROGRESS))
20.         return;
21.     // 清空
22.     stream->connect_req = NULL;
23.     uv_req_unregister(stream->loop, req);
24.     /*
25.      连接出错则注销之前注册的等待可写队列,
26.      连接成功如果待写队列为空，也注销事件，有数据需要写的时候再注册
27.     */
28.     if (error < 0 || QUEUE_EMPTY(&stream->write_queue)) {
29.         uv_io_stop(stream->loop, &stream->io_watcher, POLLOUT);
30.     }
31.     // 执行回调，通知上层连接结果
32.     if (req->cb)
33.         req->cb(req, error);
34.
35.     if (uv_stream_fd(stream) == -1)
36.         return;
37.     // 连接失败，清空待写的数据和执行写请求的回调（如果有的话）
38.     if (error < 0) {
39.         uv_stream_flush_write_queue(stream, UV_EANCELED);
40.         uv_write_callbacks(stream);
41.     }

```

|42. }

连接流的逻辑是
1 发起非阻塞式连接
2 注册等待可写事件
3 可写事件触发时，把连接结果告诉调用方
4 连接成功则发送写队列的数据，连接失败则清除写队列的数据并执行每个写请求的回调（有的话）。

5.8 监听流

监听流是针对 TCP 或 Unix 域的，主要是把一个 socket 变成 listen 状态。并且设置一些属性。

```
1. int uv_tcp_listen(uv_tcp_t* tcp, int backlog, uv_connection_cb cb)
   {
2.     static int single_accept = -1;
3.     unsigned long flags;
4.     int err;
5.
6.     if (tcp->delayed_error)
7.         return tcp->delayed_error;
8.     // 是否设置了不连续 accept。默认是连续 accept。
9.     if (single_accept == -1) {
10.         const char* val = getenv("UV_TCP_SINGLE_ACCEPT");
11.         single_accept = (val != NULL && atoi(val) != 0);
12.     }
13.    // 设置不连续 accept
14.    if (single_accept)
15.        tcp->flags |= UV_HANDLE_TCP_SINGLE_ACCEPT;
16.
17.    flags = 0;
18.    /*
19.        可能还没有用于 listen 的 fd, socket 地址等。
20.        这里申请一个 socket 和绑定到一个地址
21.        (如果调 listen 之前没有调 bind 则绑定到随机地址)
22.    */
23.    err = maybe_new_socket(tcp, AF_INET, flags);
24.    if (err)
25.        return err;
26.    // 设置 fd 为 listen 状态
27.    if (listen(tcp->io_watcher.fd, backlog))
```

```

28.     return UV_ERR(errno);
29. // 建立连接后的业务回调
30. tcp->connection_cb = cb;
31. tcp->flags |= UV_HANDLE_BOUND;
32. // 设置 io 观察者的回调, 由 epoll 监听到连接到来时执行
33. tcp->io_watcher.cb = uv_server_io;
34. /*
35.   插入观察者队列, 这时候还没有增加到 epoll,
36.   Poll IO 阶段再遍历观察者队列进行处理 (epoll_ctl)
37. */
38. uv_io_start(tcp->loop, &tcp->io_watcher, POLLIN);
39.
40. return 0;
41. }

```

监听流的逻辑看起来很多, 但是主要的逻辑是把流对的 fd 改成 listen 状态, 这样流就可以接收连接请求了。接着设置连接到来时执行的回调。最后注册 I/O 观察者到事件循环。等待连接到来。就会执行 uv_server_io。uv_server_io 再执行 connection_cb。监听流和其它流有一个区别是, 当 I/O 观察者的事件触发时, 监听流执行的回调是 uv_server_io 函数。而其它流是在 uv_stream_io 里统一处理。我们看一下连接到来或者 Unix 域上有数据到来时的处理逻辑。

```

1. void uv_server_io(uv_loop_t* loop, uv_io_t* w, unsigned int events) {
2.     uv_stream_t* stream;
3.     int err;
4.     stream = container_of(w, uv_stream_t, io_watcher);
5.     // 注册等待可读事件
6.     uv_io_start(stream->loop, &stream->io_watcher, POLLIN);
7.     while (uv_stream_fd(stream) != -1) {
8.         /*
9.           通过 accept 拿到和客户端通信的 fd, 我们看到这个
10.          fd 和服务器的 fd 是不一样的
11.        */
12.        err = uv_accept(uv_stream_fd(stream));
13.        // 错误处理
14.        if (err < 0) {
15.            /*
16.              uv_stream_fd(stream) 对应的 fd 是非阻塞的,
17.              返回这个错说明没有连接可用 accept 了, 直接返回
18.            */
19.            if (err == -EAGAIN || err == -EWOULDBLOCK)

```

```

20.         return; /* Not an error. */
21.         if (err == -ECONNABORTED)
22.             continue;
23.         // 进程的打开的文件描述符个数达到阈值，看是否有备用的
24.         if (err == -EMFILE || err == -ENFILE) {
25.             err = uv_emfile_trick(loop, uv_stream_fd(stream));
26.             if (err == -EAGAIN || err == -EWOULDBLOCK)
27.                 break;
28.         }
29.         // 发生错误，执行回调
30.         stream->connection_cb(stream, err);
31.         continue;
32.     }
33.     // 记录拿到的通信 socket 对应的 fd
34.     stream->accepted_fd = err;
35.     // 执行上传回调
36.     stream->connection_cb(stream, 0);
37.     /*
38.      stream->accepted_fd 为-1 说明在回调 connection_cb 里已经消费
39.      了 accepted_fd，否则先注销服务器在 epoll 中的 fd 的读事件，等
40.      待消费后再注册，即不再处理请求了
41.     */
42.     if (stream->accepted_fd != -1) {
43.         /*
44.          The user hasn't yet accepted called uv_accept()
45.        */
46.         uv_io_stop(loop, &stream->io_watcher, POLLIN);
47.         return;
48.     }
49.     /*
50.      是 TCP 类型的流并且设置每次只 accept 一个连接，则定时阻塞，
51.      被唤醒后再 accept，否则一直 accept（如果用户在 connect 回
52.      调里消费了 accept_fd 的话），定时阻塞用于多进程竞争处理连接
53.    */
54.     if (stream->type == UV_TCP &&
55.         (stream->flags & UV_TCP_SINGLE_ACCEPT)) {
56.         struct timespec timeout = { 0, 1 };
57.         nanosleep(&timeout, NULL);
58.     }
59. }
60. }
```

我们看到连接到来时，Libuv 会从已完成连接的队列中摘下一个节点，然后执行 connection_cb 回调。在 connection_cb 回调里，需要 uv_accept 消费 accpet_fd。

```

1. int uv_accept(uv_stream_t* server, uv_stream_t* client) {
2.     int err;
3.     switch (client->type) {
4.         case UV_NAMED_PIPE:
5.         case UV_TCP:
6.             // 把文件描述符保存到 client
7.             err = uv_stream_open(client,
8.                                 server->accepted_fd,
9.                                 UV_STREAM_READABLE
10.                                | UV_STREAM_WRITABLE);
11.            if (err) {
12.                uv_close(server->accepted_fd);
13.                goto done;
14.            }
15.            break;
16.
17.        case UV_UDP:
18.            err = uv_udp_open((uv_udp_t*) client,
19.                             server->accepted_fd);
20.            if (err) {
21.                uv_close(server->accepted_fd);
22.                goto done;
23.            }
24.            break;
25.        default:
26.            return -EINVAL;
27.    }
28.    client->flags |= UV_HANDLE_BOUND;
29.
30. done:
31.     // 非空则继续放一个到 accpet_fd 中等待 accept, 用于文件描述符传递
32.     if (server->queued_fds != NULL) {
33.         uv_stream_queued_fds_t* queued_fds;
34.         queued_fds = server->queued_fds;
35.         // 把第一个赋值到 accept_fd
36.         server->accepted_fd = queued_fds->fds[0];
37.         /*
38.          offset 减去一个单位，如果没有了，则释放内存，
39.          否则需要把后面的往前挪，offset 执行最后一个

```

```
40.     */
41.     if (--queued_fds->offset == 0) {
42.         uv_free(queued_fds);
43.         server->queued_fds = NULL;
44.     } else {
45.         memmove(queued_fds->fds,
46.                  queued_fds->fds + 1,
47.                  queued_fds->offset * sizeof(*queued_fds->fds));
48.     }
49. } else {
50.     // 没有排队的 fd 了，则注册等待可读事件，等待 accept 新的 fd
51.     server->accepted_fd = -1;
52.     if (err == 0)
53.         uv_io_start(server->loop, &server->io_watcher, POLLIN);
54. }
55. return err;
56. }
```

client 是用于和客户端进行通信的流，accept 就是把 accept_fd 保存到 client 中，client 就可以通过 fd 和对端进行通信了。消费完 accept_fd 后，如果还有待处理的 fd 的话，需要补充一个到 accept_fd（针对 Unix 域），其它的继续排队等待处理，如果没有待处理的 fd 则注册等待可读事件，继续处理新的连接。

5.9 销毁流

当我们不再需要一个流的时候，我们会首先调用 uv_close 关闭这个流，关闭流只是注销了事件和释放了文件描述符，调用 uv_close 之后，流对应的结构体就会被加入到 closing 队列，在 closing 阶段的时候，才会执行销毁流的操作，比如丢弃还没有写完成的数据，执行对应流的回调，我们看看销毁流的函数 uv_stream_destroy。

```
1. void uv_stream_destroy(uv_stream_t* stream) {
2.     // 正在连接，则执行回调
3.     if (stream->connect_req) {
4.         uv_req_unregister(stream->loop, stream->connect_req);
5.         stream->connect_req->cb(stream->connect_req, -ECANCELED);
6.         stream->connect_req = NULL;
7.     }
8.     // 丢弃待写的数据，如果有的话
9.     uv_stream_flush_write_queue(stream, -ECANCELED);
10.    // 处理写完成队列，这里是处理被丢弃的数据
```

```

11.     uv__write_callbacks(stream);
12.     // 正在关闭流，直接回调
13.     if (stream->shutdown_req) {
14.         uv__req_unregister(stream->loop, stream->shutdown_req);
15.         stream->shutdown_req->cb(stream->shutdown_req, -ECANCELED);
16.         stream->shutdown_req = NULL;
17.     }
18. }
```

我们看到，销毁流的时候，如果流中还有待写的数据，则会丢弃。我们看一下 `uv_stream_flush_write_queue` 和 `uv_write_callbacks`。

```

1. void uv_stream_flush_write_queue(uv_stream_t* stream, int error) {
2.     uv_write_t* req;
3.     QUEUE* q;
4.     while (!QUEUE_EMPTY(&stream->write_queue)) {
5.         q = QUEUE_HEAD(&stream->write_queue);
6.         QUEUE_REMOVE(q);
7.         req = QUEUE_DATA(q, uv_write_t, queue);
8.         // 把错误写到每个请求中
9.         req->error = error;
10.        QUEUE_INSERT_TAIL(&stream->write_completed_queue, &req->queue);
11.    }
12. }
```

`uv_stream_flush_write_queue` 丢弃待写队列中的请求，并直接插入写完成队列中。`uv_write_callbacks` 是写完或者写出错时执行的函数，它逐个处理写完成队列中的节点，每个节点是一个写请求，执行它的回调，如何分配了堆内存，则释放内存。在写流章节已经分析，不再具体展开。

5.10 事件触发的处理

在流的实现中，读写等操作都只是注册事件到 epoll，事件触发的时候，会执行统一的回调函数 `uv_stream_io`。下面列一下该函数的代码，具体实现在其它章节已经分析。

```

1. static void uv_stream_io(uv_loop_t* loop,
2.                           uv_io_t* w,
3.                           unsigned int events) {
4.     uv_stream_t* stream;
5.     stream = container_of(w, uv_stream_t, io_watcher);
6.     // 是连接流，则执行连接处理函数
7.     if (stream->connect_req) {
```

```
8.         uv_stream_connect(stream);
9.         return;
10.    }
11.    /*
12.     Ignore POLLHUP here. Even if it's set,
13.     there may still be data to read.
14.    */
15.    // 可读事件触发，则执行读处理
16.    if (events & (POLLIN | POLLERR | POLLHUP))
17.        uv_read(stream);
18.    // 读回调关闭了流
19.    if (uv_stream_fd(stream) == -1)
20.        return; /* read_cb closed stream. */
21.    /*
22.     POLLHUP 说明对端关闭了，即不会发生数据过来了。
23.     如果流的模式是持续读，
24.         1 如果只读取了部分（设置 UV_STREAM_READ_PARTIAL），
25.             并且没有读到结尾(没有设置 UV_STREAM_READ_EOF)，
26.             则直接作读结束处理，
27.         2 如果只读取了部分，上面的读回调执行了读结束操作，
28.             则这里就不需要处理了
29.         3 如果没有设置只读了部分，还没有执行读结束操作，
30.             则不能作读结束操作，因为对端虽然关闭了，但是之
31.             前的传过来的数据可能还没有被消费完
32.         4 如果没有设置只读了部分，执行了读结束操作，那这
33.             里也不需要处理
34.    */
35.    if ((events & POLLHUP) &&
36.        (stream->flags & UV_STREAM_READING) &&
37.        (stream->flags & UV_STREAM_READ_PARTIAL) &&
38.        !(stream->flags & UV_STREAM_READ_EOF)) {
39.        uv_buf_t buf = { NULL, 0 };
40.        uv_stream_eof(stream, &buf);
41.    }
42.
43.    if (uv_stream_fd(stream) == -1)
44.        return; /* read_cb closed stream. */
45.    // 可写事件触发
46.    if (events & (POLLOUT | POLLERR | POLLHUP)) {
47.        // 写数据
48.        uv_write(stream);
49.        // 写完后做后置处理，释放内存，执行回调等
```

```
50.         uv_write_callbacks(stream);  
51.         // 待写队列为空，则注销等待写事件  
52.         if (QUEUE_EMPTY(&stream->write_queue))  
53.             uv_drain(stream);  
54.     }  
55. }
```

第六章 C++层

本章介绍 Node.js 中 C++ 层的一些核心模块的原理和实现，这些模块是 Node.js 中很多模块都会使用的。理解这些模块的原理，才能更好地理解在 Node.js 中，JS 是如何通过 C++ 层调用 Libuv，又是如何从 Libuv 返回的。

6.1 BaseObject

BaseObject 是 C++ 层大多数类的基类。

```
1. class BaseObject : public MemoryRetainer {
2. public:
3. // ...
4. private:
5. v8::Local<v8::Object> WrappedObject() const override;
6. // 指向封装的对象
7. v8::Global<v8::Object> persistent_handle_;
8. Environment* env_;
9. };
```

BaseObject 的实现很复杂，这里只介绍常用的一些实现。

6.1.1 构造函数

```
1. // 把对象存储到 persistent_handle_ 中, 必要的时候通过 object() 取出来
2. BaseObject::BaseObject(Environment* env,
3.                         v8::Local<v8::Object> object)
4. : persistent_handle_(env->isolate(), object),
5.   env_(env) {
6.   // 把 this 存到 object 中
7.   object->SetAlignedPointerInInternalField(0, static_cast<void*>(
8.     this));
9. }
```

构造函数用于保存对象间的关系（JS 使用的对象和与其关系的 C++ 层对象，下图中的对象即我们平时在 JS 层使用 C++ 模块创建的对象，比如 new TCP()）。后面我们可以看到用处，关系如图 6-1 所示。

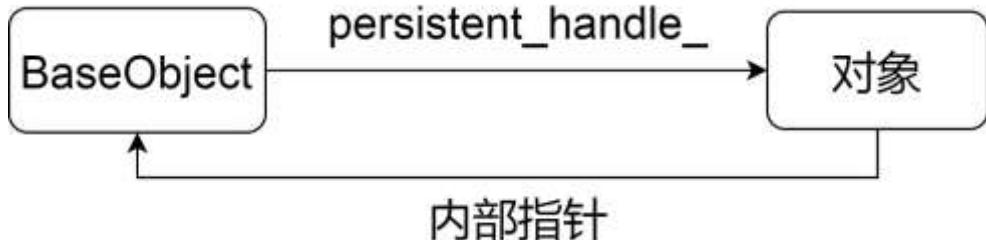


图 6-1

6.1.2 获取封装的对象

```
1. v8::Local<v8::Object> BaseObject::object() const {  
2.     return PersistentToLocal::Default(env()->isolate(),  
3.                                         persistent_handle_);  
4. }
```

6.1.3 从对象中获取保存的 BaseObject 对象

```
1. // 通过 obj 取出里面保存的 BaseObject 对象  
2. BaseObject* BaseObject::FromJSObject(v8::Local<v8::Object> obj) {  
3.     return static_cast<BaseObject*>(obj->GetAlignedPointerFromInternalField(0));  
4. }  
5.  
6. template <typename T>  
7. T* BaseObject::FromJSObject(v8::Local<v8::Object> object) {  
8.     return static_cast<T*>(FromJSObject(object));  
9. }
```

6.1.4 解包

```
1. // 从 obj 中取出对应的 BaseObject 对象  
2. template <typename T>  
3. inline T* Unwrap(v8::Local<v8::Object> obj) {  
4.     return BaseObject::FromJSObject<T>(obj);  
5. }  
6.  
7. // 从 obj 中获取对应的 BaseObject 对象，如果为空则返回第三个参数的值（默认值）
```

```
8. #define ASSIGN_OR_RETURN_UNWRAP(ptr, obj, ...) \
9.   do {           \
10.     *ptr = static_cast<typename std::remove_reference<decltype(* \
    ptr)>::type>( \
11.       BaseObject::FromJSONObject(obj));    \
12.     if (*ptr == nullptr)    \
13.       return __VA_ARGS__; \
14.   } while (0)
```

6.2 AsyncWrap

AsyncWrap 实现 async hook 的模块，不过这里我们只关注它回调 JS 的功能。

```
1. inline v8::MaybeLocal<v8::Value> AsyncWrap::MakeCallback(
2.     const v8::Local<v8::Name> symbol,
3.     int argc,
4.     v8::Local<v8::Value>* argv) {
5.     v8::Local<v8::Value> cb_v;
6.     // 根据字符串表示的属性值，从对象中取出该属性对应的值。是个函数
7.     if (!object()->Get(env()->context(), symbol).ToLocal(&cb_v))
8.         return v8::MaybeLocal<v8::Value>();
9.     // 是个函数
10.    if (!cb_v->IsFunction()) {
11.        return v8::MaybeLocal<v8::Value>();
12.    }
13.    // 回调，见 async_wrap.cc
14.    return MakeCallback(cb_v.As<v8::Function>(), argc, argv);
15. }
```

以上只是入口函数，我们看看真正的实现。

```
1. MaybeLocal<Value> AsyncWrap::MakeCallback(const Local<Function> c  
   b,  
2.                                              int argc,  
3.                                              Local<Value>* argv) {  
4.  
5.  MaybeLocal<Value> ret = InternalMakeCallback(env(), object(), c  
   b, argc, argv, context);  
6.  return ret;  
7. }
```

接着看一下 `InternalMakeCallback`

```
3.                               const Local<Function> call
4.                               back,
5.                               int argc,
6.                               Local<Value> argv[],
7.                               async_context asyncContext
8. ) {
9. // ...省略其他代码
10. // 执行回调
11. callback->Call(env->context(), recv, argc, argv);}
```

6.3 HandleWrap

HandleWrap 是对 Libuv uv_handle_t 的封装, 也是很多 C++类的基类。

```
1. class HandleWrap : public AsyncWrap {
2. public:
3.     // 操作和判断 handle 状态函数, 见 Libuv
4.     static void Close(const v8::FunctionCallbackInfo<v8::Value>& args);
5.     static void Ref(const v8::FunctionCallbackInfo<v8::Value>& args);
6.     static void Unref(const v8::FunctionCallbackInfo<v8::Value>& args);
7.     static void HasRef(const v8::FunctionCallbackInfo<v8::Value>& args);
8.     static inline bool IsAlive(const HandleWrap* wrap) {
9.         return wrap != nullptr && wrap->state_ != kClosed;
10.    }
11.
12.    static inline bool HasRef(const HandleWrap* wrap) {
13.        return IsAlive(wrap) && uv_has_ref(wrap->GetHandle());
14.    }
15.    // 获取封装的 handle
16.    inline uv_handle_t* GetHandle() const { return handle_; }
17.    // 关闭 handle, 关闭成功后执行回调
18.    virtual void Close(
19.        v8::Local<v8::Value> close_callback =
20.        v8::Local<v8::Value>());
21.
22.    static v8::Local<v8::FunctionTemplate> GetConstructorTemplate(
23.        Environment* env);
24.
25. protected:
26.     HandleWrap(Environment* env,
27.                v8::Local<v8::Object> object,
```

```

28.         uv_handle_t* handle,
29.         AsyncWrap::ProviderType provider);
30.     virtual void OnClose() {}
31.     // handle 状态
32.     inline bool IsHandleClosing() const {
33.         return state_ == kClosing || state_ == kClosed;
34.     }
35.
36. private:
37.     friend class Environment;
38.     friend void GetActiveHandles(const v8::FunctionCallbackInfo<v8
39.         ::Value>&);
40.     static void OnClose(uv_handle_t* handle);
41.     // handle 队列
42.     ListNode<HandleWrap> handle_wrap_queue_;
43.     // handle 的状态
44.     enum { kInitialized, kClosing, kClosed } state_;
45.     // 所有 handle 的基类
46.     uv_handle_t* const handle_;
47. };

```

6.3.1 新建 handle 和初始化

```

1. Local<FunctionTemplate> HandleWrap::GetConstructorTemplate(Environment* env) {
2.     Local<FunctionTemplate> tmpl = env->handle_wrap_ctor_template()
3. ;
4.     if (tmpl.IsEmpty()) {
5.         tmpl = env->NewFunctionTemplate(nullptr);
6.         tmpl->SetClassName(FIXED_ONE_BYTE_STRING(env->isolate(),
7.             "HandleWrap"));
8.         tmpl->Inherit(AsyncWrap::GetConstructorTemplate(env));
9.         env->SetProtoMethod(tmpl, "close", HandleWrap::Close);
10.        env->SetProtoMethodNoSideEffect(tmpl,
11.            "hasRef",
12.            HandleWrap::HasRef);
13.        env->SetProtoMethod(tmpl, "ref", HandleWrap::Ref);
14.        env->SetProtoMethod(tmpl, "unref", HandleWrap::Unref);
15.        env->set_handle_wrap_ctor_template(tmpl);
16.    }
17.    return tmpl;
18. /*
19.     object 为 C++ 层为 JS 层提供的对象

```

```
20. handle 为子类具体的 handle 类型，不同模块不一样
21. */
22. HandleWrap::HandleWrap(Environment* env,
23.                         Local<Object> object,
24.                         uv_handle_t* handle,
25.                         AsyncWrap::ProviderType provider)
26.     : AsyncWrap(env, object, provider),
27.       state_(kInitialized),
28.       handle_(handle) {
29.   // 保存 Libuv handle 和 C++对象的关系
30.   handle_->data = this;
31.   HandleScope scope(env->isolate());
32.   CHECK(env->has_run_bootstrapping_code());
33.   // 插入 handle 队列
34.   env->handle_wrap_queue()->PushBack(this);
35. }
```

HandleWrap 继承 BaseObject 类，初始化后关系图如图 6-2 所示。

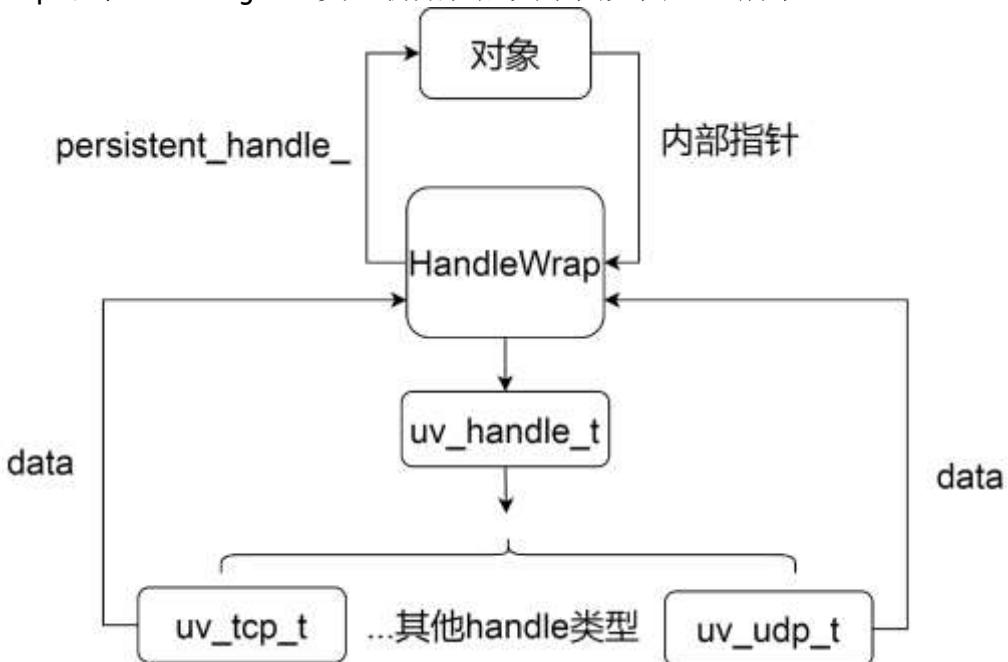


图 6-2

6.3.2 判断和操作 handle 状态

```
1. // 修改 handle 为活跃状态
2. void HandleWrap::Ref(const FunctionCallbackInfo<Value>& args) {
3.     HandleWrap* wrap;
4.     ASSIGN_OR_RETURN_UNWRAP(&wrap, args.Holder());
```

```

5.
6.   if (IsAlive(wrap))
7.     uv_unref(wrap->GetHandle());
8. }
9.
10. // 修改 handle 为不活跃状态
11. void HandleWrap::Unref(const FunctionCallbackInfo<Value>& args)
12. {
13.   HandleWrap* wrap;
14.   ASSIGN_OR_RETURN_UNWRAP(&wrap, args.Holder());
15.   if (IsAlive(wrap))
16.     uv_unref(wrap->GetHandle());
17. }
18.
19. // 判断 handle 是否处于活跃状态
20. void HandleWrap::HasRef(const FunctionCallbackInfo<Value>& args)
21. {
22.   HandleWrap* wrap;
23.   ASSIGN_OR_RETURN_UNWRAP(&wrap, args.Holder());
24.   args.GetReturnValue().Set(HasRef(wrap));
25. }
```

6.3.3 关闭 handle

```

1. // 关闭 handle (JS 层调用) , 成功后执行回调
2. void HandleWrap::Close(const FunctionCallbackInfo<Value>& args) {

3.   HandleWrap* wrap;
4.   ASSIGN_OR_RETURN_UNWRAP(&wrap, args.Holder());
5.   // 传入回调
6.   wrap->Close(args[0]);
7. }

8. // 真正关闭 handle 的函数
9. void HandleWrap::Close(Local<Value> close_callback) {
10.   // 正在关闭或已经关闭
11.   if (state_ != kInitialized)
12.     return;
13.   // 调用 Libuv 函数
14.   uv_close(handle_, OnClose);
15.   // 关闭中
16.   state_ = kClosing;
17.   // 传了回调则保存起来
18.   if (!close_callback.IsEmpty() &&
19.       close_callback->IsFunction() &&
```

```
20.     !persistent().IsEmpty()) {
21.     object()->Set(env()->context(),
22.                     env()->handle_onclose_symbol(),
23.                     close_callback).Check();
24.   }
25. }
26.
27. // 关闭 handle 成功后回调
28. void HandleWrap::OnClose(uv_handle_t* handle) {
29.   BaseObjectPtr<HandleWrap> wrap {
30.     static_cast<HandleWrap*>(handle->data)
31.   };
32.   wrap->Detach();
33.
34.   Environment* env = wrap->env();
35.   HandleScope scope(env->isolate());
36.   Context::Scope context_scope(env->context());
37.   wrap->state_ = kClosed;
38.
39.   wrap->OnClose();
40.   wrap->handle_wrap_queue_.Remove();
41.   // 有 onclose 回调则执行
42.   if (!wrap->persistent().IsEmpty() &&
43.       wrap->object()->Has(env->context(),
44.                               env->handle_onclose_symbol())
45.       .FromMaybe(false)) {
46.     wrap->MakeCallback(env->handle_onclose_symbol(),
47.                         0,
48.                         nullptr);
49.   }
50. }
```

6.4 ReqWrap

ReqWrap 表示通过 Libuv 对 handle 的一次请求。

6.4.1 ReqWrapBase

```
1. class ReqWrapBase {
2. public:
3.   explicit inline ReqWrapBase(Environment* env);
4.   virtual ~ReqWrapBase() = default;
5.   virtual void Cancel() = 0;
6.   virtual AsyncWrap* GetAsyncWrap() = 0;
7.
```

```

8. private:
9. // 一个带前后指针的节点
10. ListNode<ReqWrapBase> req_wrap_queue_;
11. };

```

ReqWrapBase 主要是定义接口的协议。我们看一下 ReqWrapBase 的实现

```

1. ReqWrapBase::ReqWrapBase(Environment* env) {
2.     env->req_wrap_queue()->PushBack(this);
3. }

```

ReqWrapBase 初始化的时候，会把自己加到 env 对象的 req 队列中。

6.4.2 ReqWrap

```

1. template <typename T>
2. class ReqWrap : public AsyncWrap, public ReqWrapBase {
3. public:
4.     inline ReqWrap(Environment* env,
5.                  v8::Local<v8::Object> object,
6.                  AsyncWrap::ProviderType provider);
7.     inline ~ReqWrap() override;
8.     inline void Dispatched();
9.     inline void Reset();
10.    T* req() { return &req_; }
11.    inline void Cancel() final;
12.    inline AsyncWrap* GetAsyncWrap() override;
13.    static ReqWrap* from_req(T* req);
14.    template <typename LibuvFunction, typename... Args>
15.    // 调用 Libuv
16.    inline int Dispatch(LibuvFunction fn, Args... args);
17.
18. public:
19.     typedef void (*callback_t)();
20.     callback_t original_callback_ = nullptr;
21.
22. protected:
23.     T req_;
24. };
25.
26. }

```

我们看一下实现

```

1. template <typename T>

```

```
2. ReqWrap<T>::ReqWrap(Environment* env,
3.                         v8::Local<v8::Object> object,
4.                         AsyncWrap::ProviderType provider)
5.     : AsyncWrap(env, object, provider),
6.     ReqWrapBase(env) {
7.     // 初始化状态
8.     Reset();
9. }
10.
11. // 保存 libuv 数据结构和 ReqWrap 实例的关系
12. template <typename T>
13. void ReqWrap<T>::Dispatched() {
14.     req_.data = this;
15. }
16.
17. // 重置字段
18. template <typename T>
19. void ReqWrap<T>::Reset() {
20.     original_callback_ = nullptr;
21.     req_.data = nullptr;
22. }
23.
24. // 通过 req 成员找所属对象的地址
25. template <typename T>
26. ReqWrap<T>* ReqWrap<T>::from_req(T* req) {
27.     return ContainerOf(&ReqWrap<T>::req_, req);
28. }
29.
30. // 取消线程池中的请求
31. template <typename T>
32. void ReqWrap<T>::Cancel() {
33.     if (req_.data == this)
34.         uv_cancel(reinterpret_cast<uv_req_t*>(&req_));
35. }
36.
37. template <typename T>
38. AsyncWrap* ReqWrap<T>::GetAsyncWrap() {
39.     return this;
40. }
41. // 调用 Libuv 函数
42. template <typename T>
43. template <typename LibuvFunction, typename... Args>
44. int ReqWrap<T>::Dispatch(LibuvFunction fn, Args... args) {
45.     Dispatched();
46.     int err = CallLibuvFunction<T, LibuvFunction>::Call(
```

```

47.     // Libuv 函数
48.     fn,
49.     env()->event_loop(),
50.     req(),
51.     MakeLibuvRequestCallback<T, Args>::For(this, args)...);
52.     if (err >= 0)
53.         env()->IncreaseWaitingRequestCounter();
54.     return err;
55. }
```

我们看到 ReqWrap 抽象了请求 Libuv 的过程，具体设计的数据结构由子类实现。我们看一下某个子类的实现。

```

1. // 请求 Libuv 时，数据结构是 uv_connect_t，表示一次连接请求
2. class ConnectWrap : public ReqWrap<uv_connect_t> {
3. public:
4.     ConnectWrap(Environment* env,
5.                  v8::Local<v8::Object> req_wrap_obj,
6.                  AsyncWrap::ProviderType provider);
7. };
```

6.5 JS 如何使用 C++

JS 调用 C++ 模块是 V8 提供的能力，Node.js 是使用了这个能力。这样我们只需要面对 JS，剩下的事情交给 Node.js 就行。本文首先讲一下利用 V8 如何实现 JS 调用 C++，然后再讲一下 Node.js 是怎么做的。

1 JS 调用 C++

首先介绍一下 V8 中两个非常核心的类 FunctionTemplate 和 ObjectTemplate。顾名思义，这两个类是定义模板的，好比建房子时的设计图一样，通过设计图，我们就可以造出对应的房子。V8 也是，定义某种模板，就可以通过这个模板创建出对应的实例。下面介绍一下这些概念（为了方便，下面都是伪代码）。

1.1 定义一个函数模板

```

1. Local<FunctionTemplate> functionTemplate = v8::FunctionTemplate::New(isolate(), New);
2. // 定义函数的名字
3. functionTemplate->SetClassName('TCP')
```

首先定义一个 FunctionTemplate 对象。我们看到 FunctionTemplate 的第二个入参是一个函数，当我们执行由 FunctionTemplate 创建的函数时，v8 就会执行 New 函数。当然我们也可以不传。

1.2 定义函数模板的 prototype 内容

prototype 就是 JS 里的 function.prototype。如果你理解 JS 里的知识，就很容易理解 C++ 的代码。

```
1. v8::Local<v8::FunctionTemplate> t = v8::FunctionTemplate::New(isolate(), callback);
2. t->SetClassName('test');
3. // 在 prototype 上定义一个属性
4. t->PrototypeTemplate()->Set('hello', 'world');
```

1.3 定义函数模板对应实例模板的内容

实例模板就是一个 ObjectTemplate 对象。它定义了，当以 new 的方式执行由函数模板创建出来的函数时，返回值所具有的属性。

```
1. function A() {
2.     this.a = 1;
3.     this.b = 2;
4. }
5. new A();
```

实例模板类似上面代码中 A 函数里面的代码。我们看看在 V8 里怎么定义。

```
1. t->InstanceTemplate()->Set(key, val);
2. t->InstanceTemplate()->SetInternalFieldCount(1);
```

InstanceTemplate 返回的是一个 ObjectTemplate 对象。SetInternalFieldCount 这个函数比较特殊，也是比较重要的一个地方，我们知道对象就是一块内存，对象有它自己的内存布局，我们知道在 C++ 里，我们定义一个类，也就定义了对象的布局。比如我们有以下定义。

```
1. class demo
2. {
3.     private:
4.     int a;
5.     int b;
6. };
```

在内存中布局如图 6-3 所示。

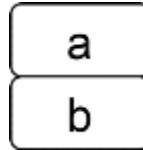


图 6-3

上面这种方式有个问题，就是类定义之后，内存布局就固定了。而 V8 是自己去控制对象的内存布局的。当我们在 V8 中定义一个类的时候，是没有任何属性的。我们看一下 V8 中 HeapObject 类的定义。

```
1. class HeapObject: public Object {
2.     static const int kMapOffset = Object::kSize; // Object::kSize 是 0
3.     static const int kSize = kMapOffset + kPointerSize;
4. };
```

这时候的内存布局如下。



然后我们再看一下 HeapObject 子类 HeapNumber 的定义。

```

1. class HeapNumber: public HeapObject {
2.     // kSize 之前的空间存储 map 对象的指针
3.     static const int kValueOffset = HeapObject::kSize;
4.     // kValueOffset - kSize 之间存储数字的值
5.     static const int kSize = kValueOffset + kDoubleSize;
6. };

```

内存布局如图 6-4 所示。

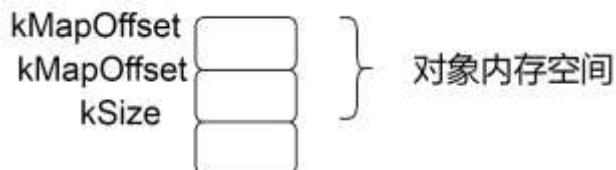


图 6-4

我们发现这些类只有几个类变量，类变量是不保存在对象内存空间的。这些类变量就是定义了对象每个域所占内存空间的信息，当我们定义一个 HeapObject 对象的时候，V8 首先申请一块内存，然后把这块内存首地址强行转成对应对象的指针。然后通过类变量对属性的内存进行存取。我们看看在 V8 里如何申请一个 HeapNumber 对象

```

1. Object* Heap::AllocateHeapNumber(double value, PretenureFlag pretenure) {
2.     // 在哪个空间分配内存，比如新生代，老生代
3.     AllocationSpace space = (pretenure == TENURED) ? CODE_SPACE : NEW_SPACE;
4.     // 在 space 上分配一个 HeapNumber 对象大小的内存
5.     Object* result = AllocateRaw(HeapNumber::kSize, space);
6.     /*
7.         转成 HeapObject，设置 map 属性，map 属性是表示对象类型、大小等信息的
8.     */
9.     HeapObject::cast(result)->set_map(heap_number_map());
10.    // 转成 HeapNumber 对象
11.    HeapNumber::cast(result)->set_value(value);
12.    return result;
13. }

```

回到对象模板的问题。我们看一下对象模板的定义。

```

1. class TemplateInfo: public Struct {
2.     static const int kTagOffset          = HeapObject::kSize;
3.     static const int kPropertyListOffset = kTagOffset + kPointerSize;
4.     static const int kHeaderSize        = kPropertyListOffset + kPointerSize;
5. };

```

```
6.  
7. class ObjectTemplateInfo: public TemplateInfo {  
8.     static const int kConstructorOffset = TemplateInfo::kHeaderSize;  
9.     static const int kInternalFieldCountOffset = kConstructorOffset + kPointerSize;  
10.    static const int kSize = kInternalFieldCountOffset + kHeaderSize;  
11.};
```

内存布局如图 6-5 所示。

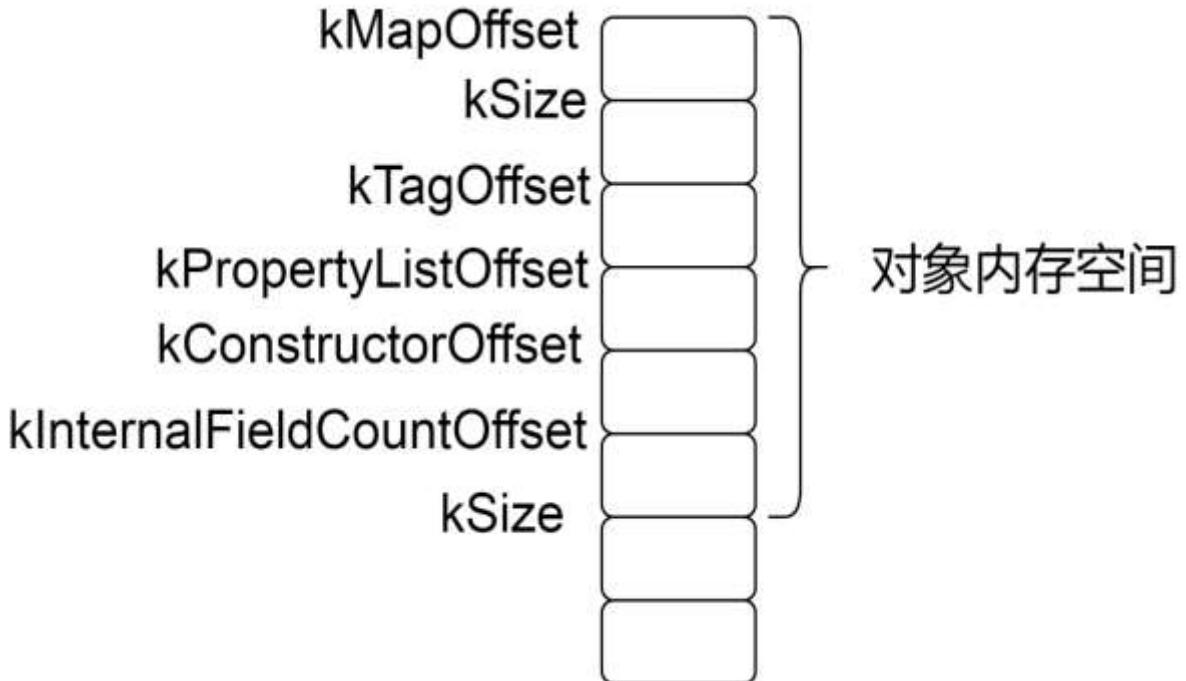


图 6-5

回到对象模板的问题，我们看看 Set(key, val) 做了什么。

```
1. void Template::Set(v8::Handle<String> name, v8::Handle<Data> value,  
2.                      v8::PropertyAttribute attribute) {  
3.     // ...  
4.     i::Handle<i::Object> list(Utils::OpenHandle(this)->property_list());  
5.     NeanderArray array(list);  
6.     array.add(Utils::OpenHandle(*name));  
7.     array.add(Utils::OpenHandle(*value));  
8.     array.add(Utils::OpenHandle(*v8::Integer::New(attribute)));  
9. }
```

上面的代码大致就是给一个 list 后面追加一些内容。我们看看这个 list 是怎么来的，即 property_list 函数的实现。

```
1. // 读取对象中某个属性的值  
2. #define READ_FIELD(p, offset) (*reinterpret_cast<Object**>(FIELD_ADDR(p, offset)))  
3.  
4. static Object* cast(Object* value) {  
5.     return value;  
6. }
```

```

7.
8. Object* TemplateInfo::property_list() {
9.     return Object::cast(READ_FIELD(this, kPropertyListOffset));
10. }

```

从上面代码中我们知道，内部布局如图 6-6 所示。

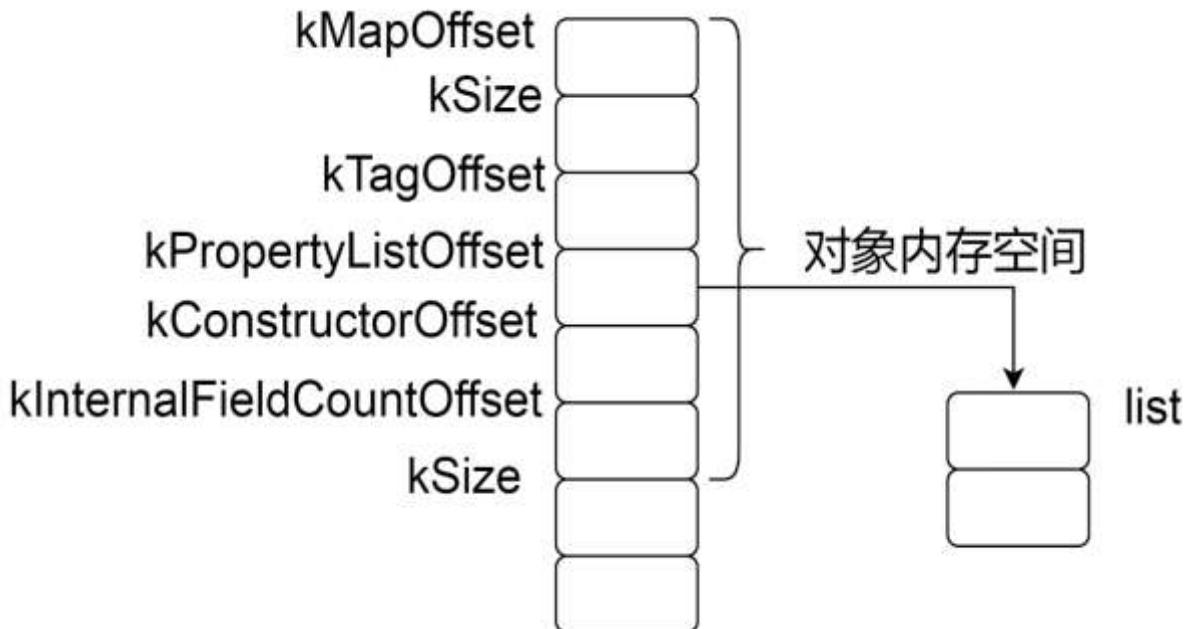


图 6-6

根据内存布局，我们知道 `property_list` 的值是 `list` 指向的值。所以 `Set(key, val)` 操作的内存并不是对象本身的内存，对象利用一个指针指向一块内存保存 `Set(key, val)` 的值。`SetInternalFieldCount` 函数就不一样了，它会影响（扩张）对象本身的内存。我们来看一下它的实现。

```

1. void ObjectTemplate::SetInternalFieldCount(int value) {
2.     // 修改的是 kInternalFieldCountOffset 对应的内存的值
3.     Utils::OpenHandle(this)->set_internal_field_count(i::Smi::FromInt(value));
4. }

```

我们看到 `SetInternalFieldCount` 函数的实现很简单，就是在对象本身的内存中保存一个数字。接下来我们看看这个字段的使用。后面会详细介绍它的用处。

```

1. Handle<JSFunction> Factory::CreateApiFunction(
2.     Handle<FunctionTemplateInfo> obj,
3.     bool is_global) {
4.
5.     int internal_field_count = 0;
6.     if (!obj->instance_template()->IsUndefined()) {
7.         // 获取函数模板的实例模板
8.         Handle<ObjectTemplateInfo> instance_template = Handle<ObjectTemplateInfo>(ObjectTem-
plateInfo::cast(obj->instance_template()));

```

```
9.     // 获取实例模板的 internal_field_count 字段的值（通过 SetInternalFieldCount 设置的那个  
    值）  
10.    internal_field_count = Smi::cast(instance_template->internal_field_count())->value(  
    );  
11. }  
12. // 计算新建对象需要的空间，如果  
13. int instance_size = kPointerSize * internal_field_count;  
14. if (is_global) {  
15.     instance_size += JSGlobalObject::kSize;  
16. } else {  
17.     instance_size += JSObject::kHeaderSize;  
18. }  
19.  
20. InstanceType type = is_global ? JS_GLOBAL_OBJECT_TYPE : JS_OBJECT_TYPE;  
21. // 新建一个函数对象  
22. Handle<JSFunction> result =  
23.     Factory::NewFunction(Factory::empty_symbol(), type, instance_size,  
24.                           code, true);  
25. }
```

我们看到 internal_field_count 的值的意义是，会扩张对象的内存，比如一个对象本身只有 n 字节，如果定义 internal_field_count 的值是 1，对象的内存就会变成 $n + \text{internal_field_count} * \text{一个指针的字节数}$ 。内存布局如图 6-7 所示。

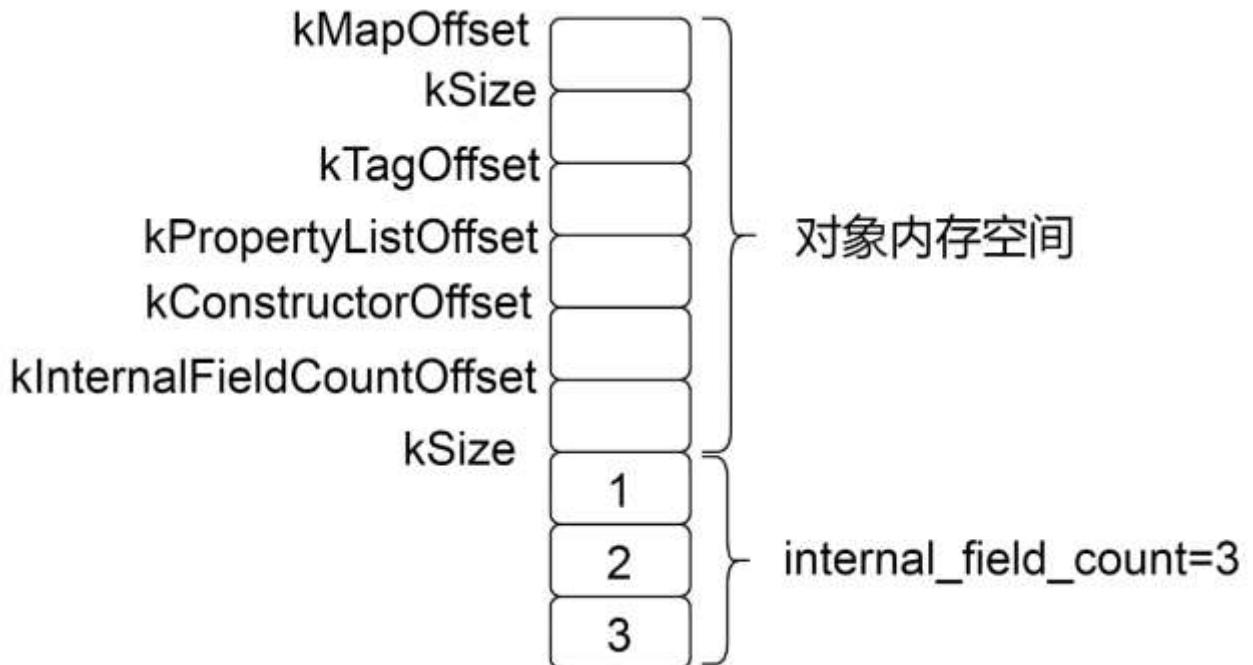


图 6-7

1.4 通过函数模板创建一个函数

```
1. Local<FunctionTemplate> functionTemplate = v8::FunctionTemplate::New(isolate(), New);  
2. global->Set('demo', functionTemplate ->GetFunction());
```

这样我们就可以在 JS 里直接调用 demo 这个变量，然后对应的函数就会被执行。这就是 JS 调用 C++ 的原理。

2 Node.js 是如何处理 JS 调用 C++ 问题的
我们以 TCP 模块为例。

```
1. constant { TCP } = process.binding('tcp_wrap');
2. new TCP(...);
```

Node.js 通过定义一个全局变量 process 统一处理 C++ 模块的调用，具体参考模块加载章节的内容。在 Node.js 中，C++ 模块（类）一般只会定义对应的 Libuv 结构体和一系列类函数，然后创建一个函数模版，并传入一个回调，接着把这些类函数挂载到函数模板中，最后通过函数模板返回一个函数 F 给 JS 层使用，翻译成 JS 大致如下

```
1. // Libuv
2. function uv_tcp_connect(uv_tcp_t, addr, cb) { cb(); }
3.
4. // C++
5. class TCPWrap {
6.
7.   uv_tcp_t = {};
8.
9.   static Connect(cb) {
10.
11.   const tcpWrap = this[0];
12.
13.   uv_tcp_connect(
14.
15.     tcpWrap.uv_tcp_t,
16.
17.     {ip: '127.0.0.1', port: 80},
18.
19.     () => { cb(); }
20.
21.   );
22.
23. }
24.
25. }
26.
27. function FunctionTemplate(cb) {
28.   function Tmp() {
29.     Object.assign(this, map);
30.     cb(this);
31.   }
32.   const map = {};
33.   return {
34.     PrototypeTemplate: function() {
35.       return {
36.         set: function(k, v) {
37.           Tmp.prototype[k] = v;
38.         }
39.       }
40.     },
41.     InstanceTemplate: function() {
42.       return {
```

```
43.         set: function(k, v) {
44.             map[k] = v;
45.         }
46.     },
47.     GetFunction() {
48.         return Tmp;
49.     }
50. }
51. }
52.
53. }
54.
55. const TCPFunctionTemplate = FunctionTemplate((target) => { target[0] = new TCPWrap(); })
56.
57. TCPFunctionTemplate.PrototypeTemplate().set('connect', TCPWrap.Connect);
58. TCPFunctionTemplate.InstanceTemplate().set('name', 'hi');
59. const TCP = TCPFunctionTemplate.GetFunction();
60.
61. // js
62. const tcp = new TCP();
63. tcp.connect(() => { console.log('连接成功'); });
64. tcp.name;
```

我们从 C++ 的层面分析执行 new TCP() 的逻辑，然后再分析 connect 的逻辑，这两个逻辑涉及的机制是其它 C++ 模块也会使用到的。因为 TCP 对应的函数是 Initialize 函数里的 t->GetFunction() 对应的值。所以 new TCP() 的时候，V8 首先会创建一个 C++ 对象，然后执行 New 函数。

```
1. void TCPWrap::New(const FunctionCallbackInfo<Value>& args) {
2.     Environment* env = Environment::GetCurrent(args);
3.
4.     int type_value = args[0].As<Int32>()->Value();
5.     TCPWrap::SocketType type = static_cast<TCPWrap::SocketType>(type_value);
6.
7.     ProviderType provider;
8.     switch (type) {
9.         case SOCKET:
10.             provider = PROVIDER_TCPWRAP;
11.             break;
12.         case SERVER:
13.             provider = PROVIDER_TCPSERVERWRAP;
14.             break;
15.         default:
16.             UNREACHABLE();
17.     }
18. /*
19.     args.This() 为 v8 提供的一个 C++ 对象（由 Initialize 函数定义的模块创建的）
20.     调用该 C++ 对象的 SetAlignedPointerInInternalField(0, this) 关联 this (new TCPWrap()) ,
21.     见 HandleWrap
22. */
23.
24.     new TCPWrap(env, args.This(), provider);
25. }
```

我们沿着 TCPWrap 的继承关系，一直到 HandleWrap

```
1. HandleWrap::HandleWrap(Environment* env,
```

```

2.             Local<Object> object,
3.             uv_handle_t* handle,
4.             AsyncWrap::ProviderType provider)
5.         : AsyncWrap(env, object, provider),
6.         state_(kInitialized),
7.         handle_(handle) {
8.     // 保存 Libuv handle 和 C++对象的关系
9.     handle_->data = this;
10.    HandleScope scope(env->isolate());
11.    // 插入 handle 队列
12.    env->handle_wrap_queue()->PushBack(this);
13. }

```

HandleWrap 首先保存了 Libuv 结构体和 C++对象的关系。然后我们继续沿着 AsyncWrap 分析，AsyncWrap 继承 BaseObject，我们直接看 BaseObject。

```

1. // 把对象存储到 persistent_handle_ 中，必要的时候通过 object()取出来
2. BaseObject::BaseObject(Environment* env, v8::Local<v8::Object> object)
3.   : persistent_handle_(env->isolate(), object), env_(env) {
4.   // 把 this 存到 object 中
5.   object->SetAlignedPointerInInternalField(0, static_cast<void*>(this));
6.   env->AddCleanupHook(DeleteMe, static_cast<void*>(this));
7.   env->modify_base_object_count(1);
8. }

```

我们看 SetAlignedPointerInInternalField。

```

1. void v8::Object::SetAlignedPointerInInternalField(int index, void* value) {
2.   i::Handle<i::JSReceiver> obj = Utils::OpenHandle(this);
3.   i::Handle<i::JSObject>::cast(obj)->SetEmbedderField(
4.     index, EncodeAlignedAsSmi(value, location));
5. }
6.
7. void JSObject::SetEmbedderField(int index, Smi* value) {
8.   // GetHeaderSize 为对象固定布局的大小，kPointerSize * index 为拓展的内存大小，根据索引找到
      对应位置
9.   int offset = GetHeaderSize() + (kPointerSize * index);
10.  // 写对应位置的内存，即保存对应的内容到内存
11.  WRITE_FIELD(this, offset, value);
12. }

```

SetAlignedPointerInInternalField 函数展开后，做的事情就是把一个值保存到 V8 C++对象的内存里。那保存的这个值是啥呢？BaseObject 的入参 object 是由函数模板创建的对象，this 是一个 TCPWrap 对象。所以 SetAlignedPointerInInternalField 函数做的事情就是把一个 TCPWrap 对象保存到一个函数模板创建的对象里，如图 6-8 所示。

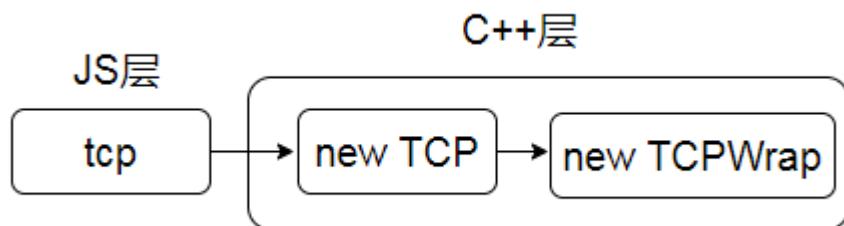


图 6-8

这有啥用呢？我们继续分析。这时候 new TCP 就执行完毕了。我们看看这时候执行 `tcp.connect()` 函数的逻辑。

```
1. template <typename T>
2. void TCPWrap::Connect(const FunctionCallbackInfo<Value>& args,
3.   std::function<int(const char* ip_address, T* addr)> uv_ip_addr) {
4.   Environment* env = Environment::GetCurrent(args);
5.
6.   TCPWrap* wrap;
7.   ASSIGN_OR_RETURN_UNWRAP(&wrap,
8.     args.Holder(),
9.     args.GetReturnValue().Set(UV_EBADF));
10.  // 省略部分不相关代码
11.
12.  args.GetReturnValue().Set(err);
13. }
```

我们只需看一下 `ASSIGN_OR_RETURN_UNWRAP` 宏的逻辑。其中 `args.Holder()` 表示 `Connect` 函数的属主，根据前面的分析我们知道属主是 `Initialize` 函数定义的函数模板创建出来的对象。这个对象保存了一个 `TCPWrap` 对象。`ASSIGN_OR_RETURN_UNWRAP` 主要的逻辑是把在 C++ 对象中保存的那个 `TCPWrap` 对象取出来。然后就可以使用 `TCPWrap` 对象的 `handle` 去请求 `Libuv` 了。

6.7 C++ 层调用 Libuv

刚才我们分析了 JS 调用 C++ 层时是如何串起来的，接着我们看一下 C++ 调用 `Libuv` 和 `Libuv` 回调 C++ 层又是如何串起来的。我们通过 `TCP` 模块的 `connect` 函数继续分析该过程。

```
1. template <typename T>
2. void TCPWrap::Connect(const FunctionCallbackInfo<Value>& args,
3.   std::function<int(const char* ip_address, T* addr)> uv_ip_addr) {
4.   Environment* env = Environment::GetCurrent(args);
5.
6.   TCPWrap* wrap;
7.   ASSIGN_OR_RETURN_UNWRAP(&wrap,
8.     args.Holder(),
9.     args.GetReturnValue().Set(UV_EBADF));
10.
11.  // 第一个参数是 TCPConnectWrap 对象，见 net 模块
12.  Local<Object> req_wrap_obj = args[0].As<Object>();
13.  // 第二个是 ip 地址
14.  node::Utf8Value ip_address(env->isolate(), args[1]);
15.
16.  T addr;
```

```

17. // 把端口, IP 设置到 addr 上, 端口信息在 uv_ip_addr 上下文里了
18. int err = uv_ip_addr(*ip_address, &addr);
19.
20. if (err == 0) {
21.     ConnectWrap* req_wrap =
22.         new ConnectWrap(env,
23.                         req_wrap_obj,
24.                         AsyncWrap::PROVIDER_TCPCONNECTWRAP);
25.     err = req_wrap->Dispatch(uv_tcp_connect,
26.                                 &wrap->handle_,
27.                                 reinterpret_cast<const sockaddr*>(&
28.                                     addr),
29.                                     AfterConnect);
30.     delete req_wrap;
31. }
32.
33. args.GetReturnValue().Set(err);
34. }
```

我们首先看一下 ConnectWrap。我们知道 ConnectWrap 是 ReqWrap 的子类。
`req_wrap_obj` 是 JS 层使用的对象。New ConnectWrap 后结构如图 6-9 所示。

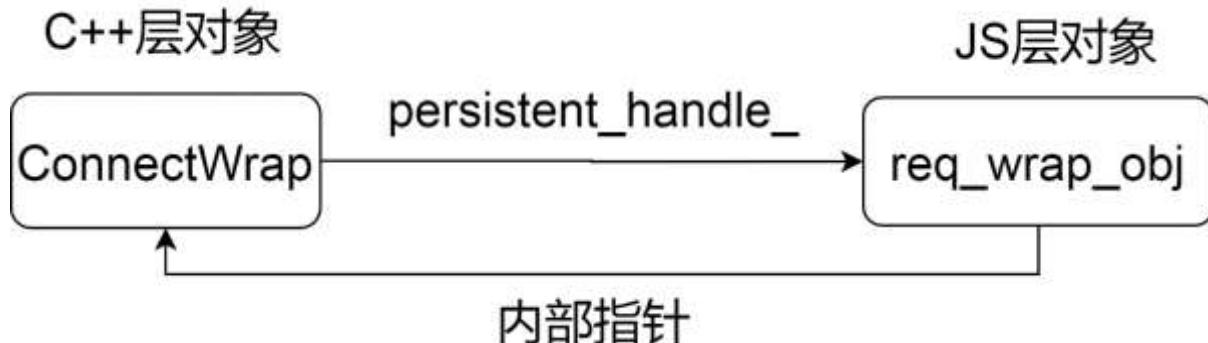


图 6-9

接着我们看一下 Dispatch。

```

1. // 调用 Libuv 函数
2. template <typename T>
3. template <typename LibuvFunction, typename... Args>
4. int ReqWrap<T>::Dispatch(LibuvFunction fn, Args... args) {
5.     // 保存 Libuv 结构体和 C++层对象 ConnectWrap 的关系
6.     req_.data = this;
7.     int err = CallLibuvFunction<T, LibuvFunction>::Call(
8.         fn,
```

```
9.     env()->event_loop(),
10.    req(),
11.    MakeLibuvRequestCallback<T, Args>::For(this, args)...);
12.    if (err >= 0)
13.        env()->IncreaseWaitingRequestCounter();
14.    return err;
15. }
```

调用 Libuv 之前的结构如图 6-10 所示。

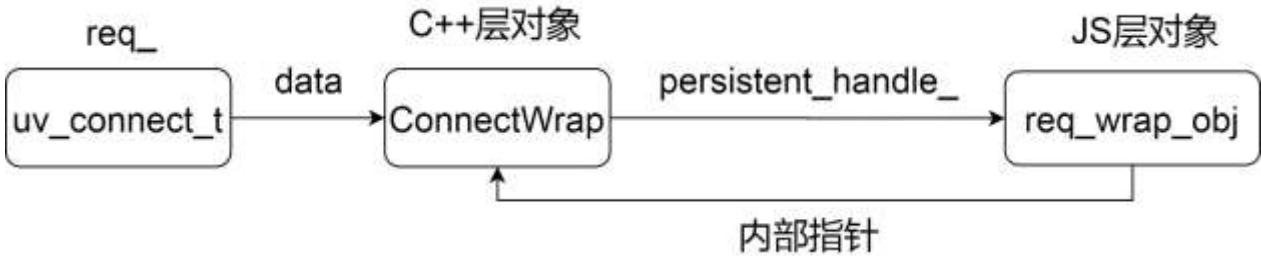


图 6-10

接下来我们分析调用 Libuv 的具体过程。我们看到 Dispatch 函数是一个函数模板。首先看一下 CallLibuvFunction 的实现。

```
1. template <typename ReqT, typename T>
2. struct CallLibuvFunction;
3.
4. // Detect `int uv_foo(uv_loop_t* loop, uv_req_t* request, ...);` .
5.
5. template <typename ReqT, typename... Args>
6. struct CallLibuvFunction<ReqT, int(*)(uv_loop_t*, ReqT*, Args...)> {
7.     using T = int(*)(uv_loop_t*, ReqT*, Args...);
8.     template <typename... PassedArgs>
9.     static int Call(T fn, uv_loop_t* loop, ReqT* req, PassedArgs...
10.      args) {
11.         return fn(loop, req, args...);
12.     }
13. };
14. // Detect `int uv_foo(uv_req_t* request, ...);` .
15. template <typename ReqT, typename... Args>
16. struct CallLibuvFunction<ReqT, int(*)(ReqT*, Args...)> {
17.     using T = int(*)(ReqT*, Args...);
18.     template <typename... PassedArgs>
19.     static int Call(T fn, uv_loop_t* loop, ReqT* req, PassedArgs...
20.      . args) {
21.         return fn(req, args...);
```

```

21. }
22. };
23.
24. // Detect `void uv_foo(uv_req_t* request, ...);`.
25. template <typename ReqT, typename... Args>
26. struct CallLibuvFunction<ReqT, void(*)(ReqT*, Args...)> {
27.     using T = void(*)(ReqT*, Args...);
28.     template <typename... PassedArgs>
29.     static int Call(T fn, uv_loop_t* loop, ReqT* req, PassedArgs...
30.         . args) {
31.         fn(req, args...);
32.     }
33. };

```

CallLibuvFunction 的实现看起来非常复杂，那是因为用了大量的模板参数，CallLibuvFunction 本质上是一个 struct，在 C++ 里和类作用类似，里面只有一个类函数 Call，Node.js 为了适配 Libuv 层各种类型函数的调用，所以实现了三种类型的 CallLibuvFunction，并且使用了大量的模板参数。我们只需要分析一种就可以了。我们根据 TCP 的 connect 函数开始分析。我们首先具体下 Dispatch 函数的模板参数。

```

1. template <typename T>
2. template <typename LibuvFunction, typename... Args>

```

T 对应 ReqWrap 的类型，LibuvFunction 对应 Libuv 的函数类型，这里是 int uv_tcp_connect(uv_connect_t* req, ...)，所以是对应 LibuvFunction 的第二种情况，Args 是执行 Dispatch 时除了第一个实参外的剩余参数。下面我们具体化 Dispatch。

```

1. int ReqWrap<uv_connect_t>::Dispatch(int(*)(uv_connect_t*, Args...
2.     ), Args... args) {
3.     req_.data = this;
4.     int err = CallLibuvFunction<uv_connect_t, int(*)(uv_connect_t*, 
5.         Args...)>::Call(
6.         fn,
7.         env()->event_loop(),
8.         req(),
9.         MakeLibuvRequestCallback<T, Args>::For(this, args)...);
10. }

```

接着我们看一下 MakeLibuvRequestCallback 的实现。

```

1. // 透传参数给 Libuv

```

```
2. template <typename ReqT, typename T>
3. struct MakeLibuvRequestCallback {
4.     static T For(ReqWrap<ReqT>* req_wrap, T v) {
5.         static_assert(!is_callable<T>::value,
6.                         "MakeLibuvRequestCallback missed a callback");
7.
8.         return v;
9.     };
10.
11. template <typename ReqT, typename... Args>
12. struct MakeLibuvRequestCallback<ReqT, void(*)(ReqT*, Args...)> {
13.
14.     using F = void(*)(ReqT* req, Args... args);
15.     // Libuv 回调
16.     static void Wrapper(ReqT* req, Args... args) {
17.         // 通过 Libuv 结构体拿到对应的 C++ 对象
18.         ReqWrap<ReqT>* req_wrap = ReqWrap<ReqT>::from_req(req);
19.         req_wrap->env()->DecreaseWaitingRequestCounter();
20.         // 拿到原始的回调执行
21.         F original_callback = reinterpret_cast<F>(req_wrap->original_
22.             _callback_);
23.         original_callback(req, args...);
24.     }
25.     static F For(ReqWrap<ReqT>* req_wrap, F v) {
26.         // 保存原来的函数
27.         CHECK_NULL(req_wrap->original_callback_);
28.         req_wrap->original_callback_ =
29.             reinterpret_cast<typename ReqWrap<ReqT>::callback_t>(v);
30.
31.         // 返回包裹函数
32.         return Wrapper;
33.     }
34. };
```

MakeLibuvRequestCallback 的实现有两种情况，模板参数的第一个一般是 ReqWrap 子类，第二个一般是 handle，初始化 ReqWrap 类的时候，env 中会记录 ReqWrap 实例的个数，从而知道有多少个请求正在被 Libuv 处理，模板参数的第二个如果是函数则说明没有使用 ReqWrap 请求 Libuv，则使用第二种实现，劫持回调从而记录正在被 Libuv 处理的请求数（如 GetAddrInfo 的实现）。所以我们这里是适配第一种实现。透传 C++ 层参数给 Libuv。我们再来看一下

Dispatch

```

1. int ReqWrap<uv_connect_t>::Dispatch(int(*)(uv_connect_t*, Args...
   ), Args... args) {
2.     req_.data = this;
3.     int err = CallLibuvFunction<uv_connect_t, int(*)(uv_connect
   _t*, Args...)>::Call(
4.         fn,
5.         env()->event_loop(),
6.         req(),
7.         args...);
8.
9.     return err;
10. }
```

再进一步展开。

```

1. static int Call(int(*fn)(uv_connect_t*, Args...), uv_loop_t* loop
   , uv_connect_t* req, PassedArgs... args) {
2.     return fn(req, args...);
3. }
```

最后展开

```

1. static int Call(int(*fn)(uv_connect_t*, Args...), uv_loop_t* loop
   , uv_connect_t* req, PassedArgs... args) {
2.     return fn(req, args...);
3. }
4.
5. Call(
6.     uv_tcp_connect,
7.     env()->event_loop(),
8.     req(),
9.     &wrap->handle_,
10.    AfterConnec
11. )
12.
13. uv_tcp_connect(
14.     env()->event_loop(),
15.     req(),
16.     &wrap->handle_,
17.     AfterConnect
18. );
```

接着我们看看 uv_tcp_connect 做了什么。

```

1. int uv_tcp_connect(uv_connect_t* req,
2.                     uv_tcp_t* handle,
3.                     const struct sockaddr* addr,
```

```
4.             uv_connect_cb cb) {
5.     // ...
6.     return uv_tcp_connect(req, handle, addr, addrlen, cb);
7. }
8.
9. int uv_tcp_connect(uv_connect_t* req,
10.                     uv_tcp_t* handle,
11.                     const struct sockaddr* addr,
12.                     unsigned int addrlen,
13.                     uv_connect_cb cb) {
14.     int err;
15.     int r;
16.
17.     // 关联起来
18.     req->handle = (uv_stream_t*) handle;
19.     // ...
20. }
```

Libuv 中把 req 和 handle 做了关联，如图 6-11 所示。

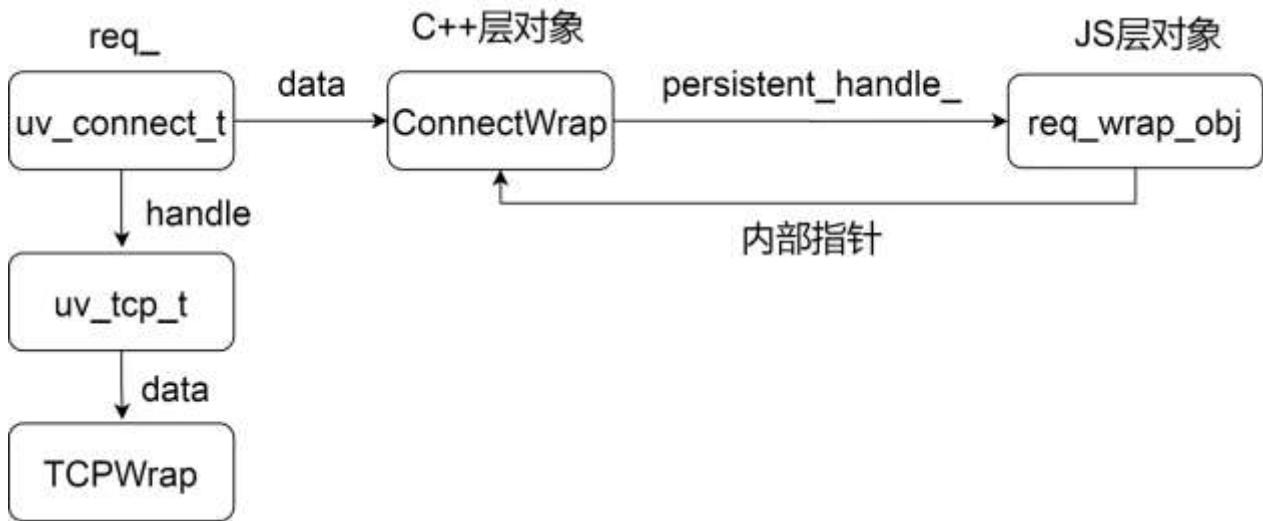


图 6-11

分析完 C++ 调用 Libuv 后，我们看看 Libuv 回调 C++ 和 C++ 回调 JS 的过程。当 Libuv 处理完请求后会执行 `AfterConnect`。

```
1. template <typename WrapType, typename UVType>
2. void ConnectionWrap<WrapType, UVType>::AfterConnect(uv_connect_t*
   req,
3.                                                       int status) {
4.
5.     // 从 Libuv 结构体拿到 C++ 的请求对象
6.     std::unique_ptr<ConnectWrap> req_wrap
7.     (static_cast<ConnectWrap*>(req->data));
8.     // 从 C++ 层请求对象拿到对应的 handle 结构体 (Libuv 里关联起来的)，再
      通过 handle 拿到对应的 C++ 层 handle 对象 (HandleWrap 关联的)
9.     WrapType* wrap = static_cast<WrapType*>(req->handle->data);
```

```

9. Environment* env = wrap->env();
10. ...
11. Local<Value> argv[5] = {
12.     Integer::New(env->isolate(), status),
13.     wrap->object(),
14.     req_wrap->object(),
15.     Boolean::New(env->isolate(), readable),
16.     Boolean::New(env->isolate(), writable)
17. };
18. // 回调 JS 层 oncomplete
19. req_wrap->MakeCallback(env->oncomplete_string(),
20.                         arraysize(argv),
21.                         argv);
22. }

```

6.8 流封装

Node.js 在 C++ 层对流进行了非常多的封装，很多模块都依赖 C++ 层流的机制，流机制的设计中，主要有三个概念

1 资源，这是流机制的核心（StreamResource），

2 对流进行操作（StreamReq）

3 流事件的监听者，当对流进行操作或流本身有事件触发时，会把事件和相关的上下文传递给监听者，监听者处理完后，再通知流（StreamListener）。

通过继承的模式，基类定义接口，子类实现接口的方式。对流的操作进行了抽象和封装。

三者的类关系如图 6-12 所示。

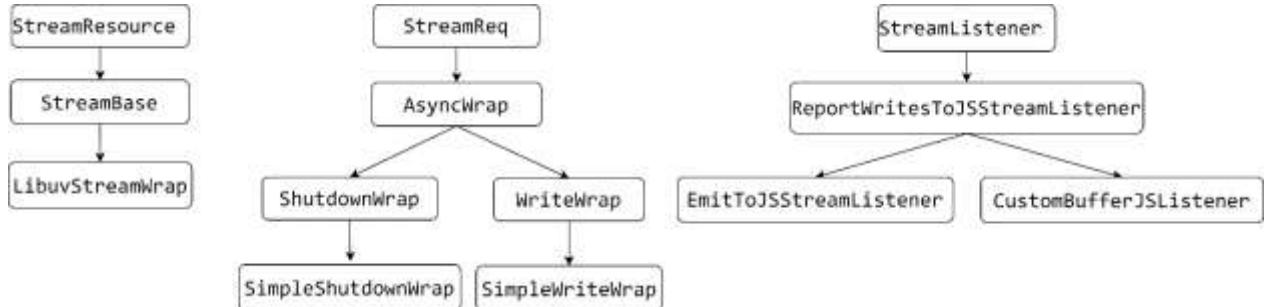


图 6-12

我们看一下读一个流的数据的过程，如图 6-13 所示。

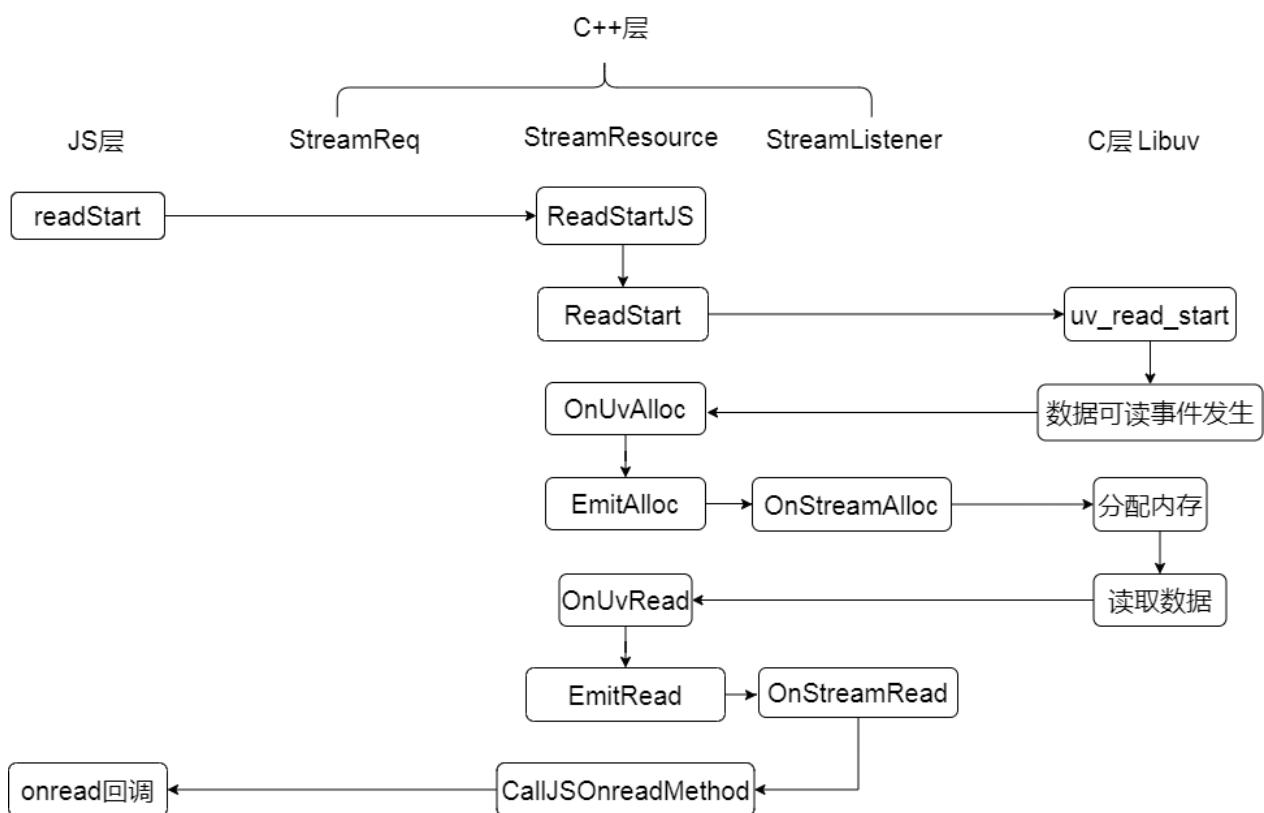


图 6-13

再看一下写的过程，如图 6-14 所示。

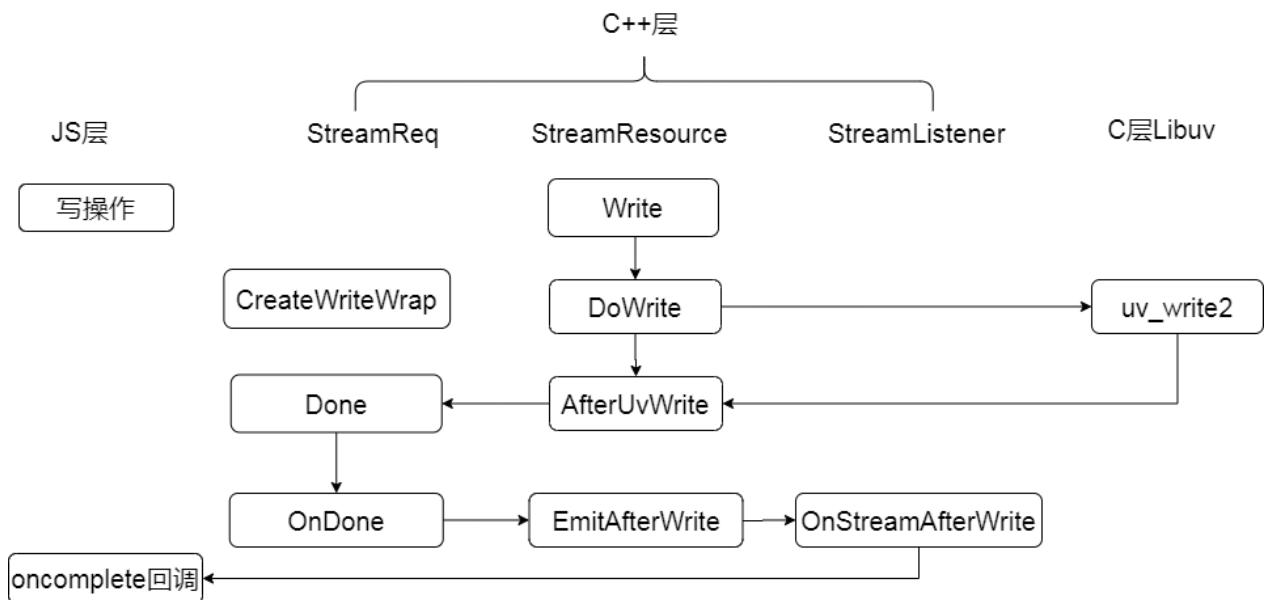


图 6-14

6.8.1 StreamResource

`StreamResource` 定义操作流的通用逻辑和操作结束后触发的回调。但是 `StreamResource` 不定义流的类型，流的类型由子类定义，我们可以在 `StreamResource` 上注册 `listener`，表示对流感兴趣，当流上有数据可读或者事件发生时，就会通知 `listener`。

```

1. class StreamResource {
2.   public:
3.     virtual ~StreamResource();
4.     // 注册/注销等待流可读事件
5.     virtual int ReadStart() = 0;
6.     virtual int ReadStop() = 0;
7.     // 关闭流
8.     virtual int DoShutdown(ShutdownWrap* req_wrap) = 0;
9.     // 写入流
10.    virtual int DoTryWrite(uv_buf_t** bufs, size_t* count);
11.    virtual int DoWrite(WriteWrap* w,
12.                          uv_buf_t* bufs,
13.                          size_t count,
14.                          uv_stream_t* send_handle) = 0;
15.    // ...忽略一些
16.    // 给流增加或删除监听者
17.    void PushStreamListener(StreamListener* listener);
18.    void RemoveStreamListener(StreamListener* listener);
19.
20.   protected:
21.     uv_buf_t EmitAlloc(size_t suggested_size);
22.     void EmitRead(ssize_t nread,
23.                  const uv_buf_t& buf = uv_buf_init(nullptr, 0));
24.     // 流的监听者，即数据消费者
25.     StreamListener* listener_ = nullptr;
26.     uint64_t bytes_read_ = 0;
27.     uint64_t bytes_written_ = 0;
28.     friend class StreamListener;
29. };

```

`StreamResource` 是一个基类，其中有一个成员是 `StreamListener` 类的实例，我们后面分析。我们看一下 `StreamResource` 的实现。

1 增加一个 `listener`

```

1. // 增加一个 listener
2. inline void StreamResource::PushStreamListener(StreamListener* li
   stener) {

```

```
3. // 头插法
4. listener->previous_listener_ = listener_;
5. listener->stream_ = this;
6. listener_ = listener;
7. }
```

我们可以在一个流上注册多个 listener，流的 listener_ 字段维护了流上所有的 listener 队列。关系图如图 6-15 所示。

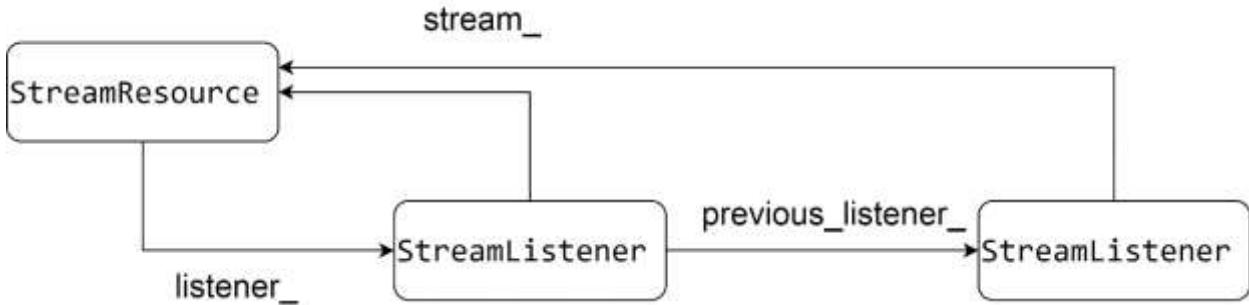


图 6-15

2 删除 listener

```
1. inline void StreamResource::RemoveStreamListener(StreamListener* listener) {
2.     StreamListener* previous;
3.     StreamListener* current;
4.
5.     // 遍历单链表
6.     for (current = listener_, previous = nullptr;
7.          /* No loop condition because we want a crash if listener is not found */
8.          ; previous = current, current = current->previous_listener_
_) {
9.         if (current == listener) {
10.            // 非空说明需要删除的不是第一个节点
11.            if (previous != nullptr)
12.                previous->previous_listener_ = current->previous_listener_
r_;
13.            else
14.                // 删除的是第一个节点，更新头指针就行
15.                listener_ = listener->previous_listener_;
16.                break;
17.        }
18.    }
19.    // 重置被删除 listener 的字段
20.    listener->stream_ = nullptr;
21.    listener->previous_listener_ = nullptr;
```

| 22. }

3 申请存储数据

```
1. // 申请一块内存
2. inline uv_buf_t StreamResource::EmitAlloc(size_t suggested_size)
{
3.   DebugSealHandleScope handle_scope(v8::Isolate::GetCurrent());
4.   return listener_->OnStreamAlloc(suggested_size);
5. }
```

StreamResource 只是定义了操作流的通用逻辑，数据存储和消费由 listener 定义。

4 数据可读

```
1. inline void StreamResource::EmitRead(ssize_t nread, const uv_buf_t& buf) {
2.   if (nread > 0)
3.     // 记录从流中读取的数据的字节大小
4.     bytes_read_ += static_cast<uint64_t>(nread);
5.   listener_->OnStreamRead(nread, buf);
6. }
```

5 写回调

```
1. inline void StreamResource::EmitAfterWrite(WriteWrap* w, int stat
   us) {
2.   DebugSealHandleScope handle_scope(v8::Isolate::GetCurrent());
3.   listener_->OnStreamAfterWrite(w, status);
4. }
```

6 关闭流回调

```
1. inline void StreamResource::EmitAfterShutdown(ShutdownWrap* w, in
   t status) {
2.   DebugSealHandleScope handle_scope(v8::Isolate::GetCurrent());
3.   listener_->OnStreamAfterShutdown(w, status);
4. }
```

7 流销毁回调

```
1. inline StreamResource::~StreamResource() {
2.   while (listener_ != nullptr) {
3.     StreamListener* listener = listener_;
4.     listener->OnStreamDestroy();
5.     if (listener == listener_)
```

```
6.         RemoveStreamListener(listener_);
7.     }
8. }
```

流销毁后需要通知 listener，并且解除关系。

6.8.2 StreamBase

StreamBase 是 StreamResource 的子类，拓展了 StreamResource 的功能。

```
1. class StreamBase : public StreamResource {
2. public:
3.     static constexpr int kStreamBaseField = 1;
4.     static constexpr int kOnReadFunctionField = 2;
5.     static constexpr int kStreamBaseFieldCount = 3;
6.     // 定义一些统一的逻辑
7.     static void AddMethods(Environment* env,
8.                           v8::Local<v8::FunctionTemplate> target);
9.
10.    virtual bool IsAlive() = 0;
11.    virtual bool IsClosing() = 0;
12.    virtual bool IsIPCPipe();
13.    virtual int GetFD();
14.
15.    // 执行 JS 回调
16.    v8::MaybeLocal<v8::Value> CallJSOnreadMethod(
17.        ssize_t nread,
18.        v8::Local<v8::ArrayBuffer> ab,
19.        size_t offset = 0,
20.        StreamBaseJSChecks checks = DONT_SKIP_NREAD_CHECKS);
21.
22.    Environment* stream_env() const;
23.    // 关闭流
24.    int Shutdown(v8::Local<v8::Object> req_wrap_obj = v8::Local<v8
25.                 ::Object>());
26.    // 写入流
27.    StreamWriteResult Write(
28.        uv_buf_t* bufs,
29.        size_t count,
30.        uv_stream_t* send_handle = nullptr,
31.        v8::Local<v8::Object> req_wrap_obj = v8::Local<v8::Object>
32.        ());
33.    // 创建一个关闭请求
34.    virtual ShutdownWrap* CreateShutdownWrap(v8::Local<v8::Object>
35.                                              object);
```

```

33. // 创建一个写请求
34. virtual WriteWrap* CreateWriteWrap(v8::Local<v8::Object> objec
   t);
35.
36. virtual AsyncWrap* GetAsyncWrap() = 0;
37. virtual v8::Local<v8::Object> GetObject();
38. static StreamBase* FromObject(v8::Local<v8::Object> obj);
39.
40. protected:
41. explicit StreamBase(Environment* env);
42.
43. // JS Methods
44. int ReadStartJS(const FunctionCallbackInfo<v8::Value>& arg
   s);
45. // 省略系列方法
46. void AttachToObject(v8::Local<v8::Object> obj);
47.
48. template <int (StreamBase::*Method)()
49.           const FunctionCallbackInfo<v8::Value>& args>
50. static void JSMethod(const FunctionCallbackInfo<v8::Value>
   & args);
51.
52. private:
53. Environment* env_;
54. EmitToJSSStreamListener default_listener_;
55.
56. void SetWriteResult(const StreamWriteResult& res);
57. static void AddMethod(Environment* env,
   v8::Local<v8::Signature> sig,
   enum v8::PropertyAttribute attributes,
   v8::Local<v8::FunctionTemplate> t,
   JSMethodFunction* stream_method,
   v8::Local<v8::String> str);
58. };

```

1 初始化

```

1. inline StreamBase::StreamBase(Environment* env) : env_(env) {
2.   PushStreamListener(&default_listener_);
3. }

```

StreamBase 初始化的时候会默认设置一个 listener。

2 关闭流

```

1. // 关闭一个流, req_wrap_obj 是 JS 层传进来的对象

```

```
2. inline int StreamBase::Shutdown(v8::Local<v8::Object> req_wrap_obj) {
3.     Environment* env = stream_env();
4.     HandleScope handle_scope(env->isolate());
5.     AsyncHooks::DefaultTriggerAsyncIdScope trigger_scope(GetAsyncWrap());
6.     // 创建一个用于请求 Libuv 的数据结构
7.     ShutdownWrap* req_wrap = CreateShutdownWrap(req_wrap_obj);
8.     // 子类实现，不同流关闭的逻辑不一样
9.     int err = DoShutdown(req_wrap);
10.    // 执行出错则销毁 JS 层对象
11.    if (err != 0 && req_wrap != nullptr) {
12.        req_wrap->Dispose();
13.    }
14.
15.    const char* msg = Error();
16.    if (msg != nullptr) {
17.        req_wrap_obj->Set(
18.            env->context(),
19.            env->error_string(),
20.            OneByteString(env->isolate(), msg)).Check();
21.        ClearError();
22.    }
23.
24.    return err;
25. }
```

3 写

```
1. // 写 Buffer，支持发送文件描述符
2. int StreamBase::WriteBuffer(const FunctionCallbackInfo<Value>& args) {
3.     Environment* env = Environment::GetCurrent(args);
4.
5.     Local<Object> req_wrap_obj = args[0].As<Object>();
6.     uv_buf_t buf;
7.     // 数据内容和长度
8.     buf.base = Buffer::Data(args[1]);
9.     buf.len = Buffer::Length(args[1]);
10.
11.    uv_stream_t* send_handle = nullptr;
12.    // 是对象并且流支持发送文件描述符
13.    if (args[2]->IsObject() && IsIPCPipe()) {
14.        Local<Object> send_handle_obj = args[2].As<Object>();
15.    }
```

```

16.     HandleWrap* wrap;
17.     // 从返回 js 的对象中获取 internalField 中指向的 C++ 层对象
18.     ASSIGN_OR_RETURN_UNWRAP(&wrap, send_handle_obj, UV_EINVAL);

19.     // 拿到 Libuv 层的 handle
20.     send_handle = reinterpret_cast<uv_stream_t*>(wrap->GetHandle
());
21.     // Reference LibuvStreamWrap instance to prevent it from bei
ng garbage
22.     // collected before `AfterWrite` is called.
23.     // 设置到 JS 层请求对象中
24.     req_wrap_obj->Set(env->context(),
25.                         env->handle_string(),
26.                         send_handle_obj).Check();
27. }
28.
29. StreamWriteResult res = Write(&buf, 1, send_handle, req_wrap_o
bj);
30. SetWriteResult(res);
31.
32. return res.err;
33. }
```

```

1. inline StreamWriteResult StreamBase::Write(
2.     uv_buf_t* bufs,
3.     size_t count,
4.     uv_stream_t* send_handle,
5.     v8::Local<v8::Object> req_wrap_obj) {
6.     Environment* env = stream_env();
7.     int err;
8.
9.     size_t total_bytes = 0;
10.    // 计算需要写入的数据大小
11.    for (size_t i = 0; i < count; ++i)
12.        total_bytes += bufs[i].len;
13.    // 同上
14.    bytes_written_ += total_bytes;
15.    // 是否需要发送文件描述符，不需要则直接写
16.    if (send_handle == nullptr) {
17.        err = DoTryWrite(&bufs, &count);
18.        if (err != 0 || count == 0) {
19.            return StreamWriteResult { false, err, nullptr, total_byte
s };

```

• 158 •

```
20.    }
21. }
22.
23. HandleScope handle_scope(env->isolate());
24.
25. AsyncHooks::DefaultTriggerAsyncIdScope trigger_scope(GetAsyncWrap());
26. // 创建一个用于请求 Libuv 的写请求对象
27. WriteWrap* req_wrap = CreateWriteWrap(req_wrap_obj);
28. // 执行写, 子类实现, 不同流写操作不一样
29. err = DoWrite(req_wrap, bufs, count, send_handle);
30.
31. const char* msg = Error();
32. if (msg != nullptr) {
33.     req_wrap_obj->Set(env->context(),
34.                         env->error_string(),
35.                         OneByteString(env->isolate(), msg)).Check(
36. );
37.     ClearError();
38.
39.     return StreamWriteResult { async, err, req_wrap, total_bytes }
40. }
```

4 读

```

16. Environment* env = env_;
17. env->stream_base_state()[kReadBytesOrError] = nread;
18. env->stream_base_state()[kArrayBufferOffset] = offset;
19.
20. Local<Value> argv[] = {
21.     ab.IsEmpty() ? Undefined(env->isolate()).As<Value>() : ab.As
   <Value>()
22. };
23. // GetAsyncWrap 在 StreamBase 子类实现，拿到 StreamBase 类对象
24. AsyncWrap* wrap = GetAsyncWrap();
25. // 获取回调执行
26. Local<Value> onread = wrap->object()->GetInternalField(kOnRead
   FunctionField);
27. return wrap->MakeCallback(onread.As<Function>(), arraysize(arg
   v), argv);
28. }

```

4 流通用方法

```

1. void StreamBase::AddMethod(Environment* env,
2.                             Local<Signature> signature,
3.                             enum PropertyAttribute attributes,
4.                             Local<FunctionTemplate> t,
5.                             JSMETHODFunction* stream_method,
6.                             Local<String> string) {
7.     // 新建一个函数模板
8.     Local<FunctionTemplate> templ =
9.         env->NewFunctionTemplate(stream_method,
10.                               signature,
11.                               v8::ConstructorBehavior::kThrow,
12.                               v8::SideEffectType::kHasNoSideEff
   ect);
13.     // 设置原型属性
14.     t->PrototypeTemplate()->SetAccessorProperty(
15.         string, templ, Local<FunctionTemplate>(), attributes);
16. }
17.
18. void StreamBase::AddMethods(Environment* env, Local<FunctionTemp
   late> t) {
19.     HandleScope scope(env->isolate());
20.
21.     enum PropertyAttribute attributes =
22.         static_cast<PropertyAttribute>(ReadOnly | DontDelete | Don
   tEnum);
23.     Local<Signature> sig = Signature::New(env->isolate(), t);

```

```
24. // 设置原型属性
25. AddMethod(env, sig, attributes, t, GetFD, env->fd_string());
26. // 忽略部分
27. env->SetProtoMethod(t, "readStart", JSMethod<&StreamBase::ReadStartJS>);
28. env->SetProtoMethod(t, "readStop", JSMethod<&StreamBase::ReadStopJS>);
29. env->SetProtoMethod(t, "shutdown", JSMethod<&StreamBase::Shutdown>);
30. env->SetProtoMethod(t, "writev", JSMethod<&StreamBase::Writev>);
31. env->SetProtoMethod(t, "writeBuffer", JSMethod<&StreamBase::WriteBuffer>);
32. env->SetProtoMethod(
33.     t, "writeAsciiString", JSMethod<&StreamBase::WriteString<ASCII>>);
34. env->SetProtoMethod(
35.     t, "writeUtf8String", JSMethod<&StreamBase::WriteString<UTF8>>);
36. t->PrototypeTemplate()->Set(FIXED_ONE_BYTE_STRING(env->isolate()),
37.                                 "isStreamBase",
38.                                 True(env->isolate())));
39. // 设置访问器
40. t->PrototypeTemplate()->SetAccessor(
41.     // 键名
42.     FIXED_ONE_BYTE_STRING(env->isolate(), "onread"),
43.     // getter
44.     BaseObject::InternalFieldGet<kOnReadFunctionField>,
45.     // setter, Value::IsFunction 是 set 之前的校验函数, 见
        InternalFieldSet(模板函数) 定义
46.     BaseObject::InternalFieldSet<kOnReadFunctionField, &Value::IsFunction>);
47. }
```

5 其它函数

```
1. // 默认 false, 子类重写
2. bool StreamBase::IsIPCPipe() {
3.     return false;
4. }
5.
6. // 子类重写
7. int StreamBase::GetFD() {
```

```

8.     return -1;
9. }
10.
11. Local<Object> StreamBase::GetObject() {
12.     return GetAsyncWrap()->object();
13. }
14.
15. // 工具函数和实例 this 无关，和入参有关
16. void StreamBase::GetFD(const FunctionCallbackInfo<Value>& args)
   {
17.     // Mimic implementation of StreamBase::GetFD() and UDPWrap::Ge
       tFD().
18.     // 从 JS 层对象获取它关联的 C++ 对象，不一定是 this
19.     StreamBase* wrap = StreamBase::FromObject(args.This()).As<Objec
       t>();
20.     if (wrap == nullptr) return args.GetReturnValue().Set(UV_EINVA
       L);
21.
22.     if (!wrap->IsAlive()) return args.GetReturnValue().Set(UV_EINV
       AL);
23.
24.     args.GetReturnValue().Set(wrap->GetFD());
25. }
26.
27. void StreamBase::GetBytesRead(const FunctionCallbackInfo<Value>&
   args) {
28.     StreamBase* wrap = StreamBase::FromObject(args.This()).As<Objec
       t>();
29.     if (wrap == nullptr) return args.GetReturnValue().Set(0);
30.
31.     // uint64_t -> double. 53bits is enough for all real cases.
32.     args.GetReturnValue().Set(static_cast<double>(wrap->bytes_read
      _));
33. }
```

6.8.3 LibuvStreamWrap

LibuvStreamWrap 是 StreamBase 的子类。实现了父类的接口，也拓展了流的能力。

```

1. class LibuvStreamWrap : public HandleWrap, public StreamBase {
2. public:
3.     static void Initialize(v8::Local<v8::Object> target,
4.                           v8::Local<v8::Value> unused,
5.                           v8::Local<v8::Context> context,
6.                           void* priv);
7.
```

```
8. int GetFD() override;
9. bool IsAlive() override;
10. bool IsClosing() override;
11. bool IsIPCPipe() override;
12.
13. // JavaScript functions
14. int ReadStart() override;
15. int ReadStop() override;
16.
17. // Resource implementation
18. int DoShutdown(ShutdownWrap* req_wrap) override;
19. int DoTryWrite(uv_buf_t** bufs, size_t* count) override;
20. int DoWrite(WriteWrap* w,
21.              uv_buf_t* bufs,
22.              size_t count,
23.              uv_stream_t* send_handle) override;
24.
25. inline uv_stream_t* stream() const {
26.     return stream_;
27. }
28. // 是否是 Unix 域或命名管道
29. inline bool is_named_pipe() const {
30.     return stream()->type == UV_NAMED_PIPE;
31. }
32. // 是否是 Unix 域并且支持传递文件描述符
33. inline bool is_named_pipe_ipc() const {
34.     return is_named_pipe() &&
35.            reinterpret_cast<const uv_pipe_t*>(stream())->ipc != 0
36. ;
37. }
38. inline bool is_tcp() const {
39.     return stream()->type == UV_TCP;
40. }
41. // 创建请求 Libuv 的对象
42. ShutdownWrap* CreateShutdownWrap(v8::Local<v8::Object> object)
    override;
43. WriteWrap* CreateWriteWrap(v8::Local<v8::Object> object) override;
44. // 从 JS 层对象获取对应的 C++ 对象
45. static LibuvStreamWrap* From(Environment* env, v8::Local<v8::Object> object);
46.
47. protected:
48. LibuvStreamWrap(Environment* env,
49.                  v8::Local<v8::Object> object,
```

```

50.             uv_stream_t* stream,
51.             AsyncWrap::ProviderType provider);
52.
53.     AsyncWrap* GetAsyncWrap() override;
54.
55.     static v8::Local<v8::FunctionTemplate> GetConstructorTemplate(
56.         Environment* env);
57.
58. private:
59.     static void GetWriteQueueSize(
60.         const v8::FunctionCallbackInfo<v8::Value>& info);
61.     static void SetBlocking(const v8::FunctionCallbackInfo<v8::Value>& args);
62.
63. // Callbacks for libuv
64.     void OnUvAlloc(size_t suggested_size, uv_buf_t* buf);
65.     void OnUvRead(ssize_t nread, const uv_buf_t* buf);
66.
67.     static void AfterUvWrite(uv_write_t* req, int status);
68.     static void AfterUvShutdown(uv_shutdown_t* req, int status);
69.
70.     uv_stream_t* const stream_;
71. };

```

1 初始化

```

1. LibuvStreamWrap::LibuvStreamWrap(Environment* env,
2.                                     Local<Object> object,
3.                                     uv_stream_t* stream,
4.                                     AsyncWrap::ProviderType provider
5. )
6.     : HandleWrap(env,
7.                 object,
8.                 reinterpret_cast<uv_handle_t*>(stream),
9.                 provider),
10.    StreamBase(env),
11.    stream_(stream) {
12.     StreamBase::AttachToObject(object);

```

LibuvStreamWrap 初始化的时候，会把 JS 层使用的对象的内部指针指向自己，见 HandleWrap。

2 写操作

```

1. // 工具函数，获取待写数据字节的大小
2. void LibuvStreamWrap::GetWriteQueueSize(

```

```
3.     const FunctionCallbackInfo<Value>& info) {
4.     LibuvStreamWrap* wrap;
5.     ASSIGN_OR_RETURN_UNWRAP(&wrap, info.This());
6.     uint32_t write_queue_size = wrap->stream()->write_queue_size;
7.     info.GetReturnValue().Set(write_queue_size);
8. }
9.
10. // 设置非阻塞
11. void LibuvStreamWrap::SetBlocking(const FunctionCallbackInfo<Value>& args) {
12.     LibuvStreamWrap* wrap;
13.     ASSIGN_OR_RETURN_UNWRAP(&wrap, args.Holder());
14.     bool enable = args[0]->IsTrue();
15.     args.GetReturnValue().Set(uv_stream_set_blocking(wrap->stream(),
16. ), enable));
16. }
17. // 定义一个关闭的请求
18. typedef SimpleShutdownWrap<ReqWrap<uv_shutdown_t>> LibuvShutdownWrap;
19. // 定义一个写请求
20. typedef SimpleWriteWrap<ReqWrap<uv_write_t>> LibuvWriteWrap;
21.
22. ShutdownWrap* LibuvStreamWrap::CreateShutdownWrap(Local<Object>
object) {
23.     return new LibuvShutdownWrap(this, object);
24. }
25.
26. WriteWrap* LibuvStreamWrap::CreateWriteWrap(Local<Object> object
) {
27.     return new LibuvWriteWrap(this, object);
28. }
29.
30. // 发起关闭请求, 由父类调用, req_wrap 是 C++层创建的对象
31. int LibuvStreamWrap::DoShutdown(ShutdownWrap* req_wrap_) {
32.     LibuvShutdownWrap* req_wrap = static_cast<LibuvShutdownWrap*>(
req_wrap_);
33.     return req_wrap->Dispatch(uv_shutdown, stream(), AfterUvShutdo
wn);
34. }
35.
36. // 关闭请求结束后执行请求的通用回调 Done
37. void LibuvStreamWrap::AfterUvShutdown(uv_shutdown_t* req, int st
atus) {
38.     LibuvShutdownWrap* req_wrap = static_cast<LibuvShutdownWrap*>(
39.         LibuvShutdownWrap::from_req(req));
40.     HandleScope scope(req_wrap->env()->isolate());
```

```
41.     Context::Scope context_scope(req_wrap->env()->context());
42.     req_wrap->Done(status);
43. }
44.
45. int LibuvStreamWrap::DoTryWrite(uv_buf_t** bufs, size_t* count)
46. {
47.     int err;
48.     size_t written;
49.     uv_buf_t* vbufs = *bufs;
50.     size_t vcount = *count;
51.     err = uv_try_write(stream(), vbufs, vcount);
52.     if (err == UV_ENOSYS || err == UV_EAGAIN)
53.         return 0;
54.     if (err < 0)
55.         return err;
56.     // 写成功的字节数, 更新数据
57.     written = err;
58.     for (; vcount > 0; vbufs++, vcount--) {
59.         // Slice
60.         if (vbufs[0].len > written) {
61.             vbufs[0].base += written;
62.             vbufs[0].len -= written;
63.             written = 0;
64.             break;
65.         }
66.         // Discard
67.     } else {
68.         written -= vbufs[0].len;
69.     }
70. }
71.
72. *bufs = vbufs;
73. *count = vcount;
74.
75. return 0;
76. }
77.
78.
79. int LibuvStreamWrap::DoWrite(WriteWrap* req_wrap,
80.                             uv_buf_t* bufs,
81.                             size_t count,
82.                             uv_stream_t* send_handle) {
83.     LibuvWriteWrap* w = static_cast<LibuvWriteWrap*>(req_wrap);
84.     return w->Dispatch(uv_write2,
85.                         stream(),
```

```
86.             bufs,
87.             count,
88.             send_handle,
89.             AfterUvWrite);
90. }
91.
92.
93.
94. void LibuvStreamWrap::AfterUvWrite(uv_write_t* req, int status)
95. {
96.     LibuvWriteWrap* req_wrap = static_cast<LibuvWriteWrap*>(
97.         LibuvWriteWrap::from_req(req));
98.     HandleScope scope(req_wrap->env()->isolate());
99.     Context::Scope context_scope(req_wrap->env()->context());
100.    req_wrap->Done(status);
101. }
```

3 读操作

```
1. // 调用 Libuv 实现启动读逻辑
2. int LibuvStreamWrap::ReadStart() {
3.     return uv_read_start(stream(), [](uv_handle_t* handle,
4.                                     size_t suggested_size,
5.                                     uv_buf_t* buf) {
6.         static_cast<LibuvStreamWrap*>(handle->data)->OnUvAlloc(suggested_size, buf);
7.     }, [](uv_stream_t* stream, ssize_t nread, const uv_buf_t* buf) {
8.         static_cast<LibuvStreamWrap*>(stream->data)->OnUvRead(nread, buf);
9.     });
10. }
11.
12. // 实现停止读逻辑
13. int LibuvStreamWrap::ReadStop() {
14.     return uv_read_stop(stream());
15. }
16.
17. // 需要分配内存时的回调，由 Libuv 回调，具体分配内存逻辑由 listener 实现
18. void LibuvStreamWrap::OnUvAlloc(size_t suggested_size, uv_buf_t* buf) {
19.     HandleScope scope(env()->isolate());
20.     Context::Scope context_scope(env()->context());
21.
22.     *buf = EmitAlloc(suggested_size);
```

```

23. }
24. // 处理传递的文件描述符
25. template <class WrapType>
26. static MaybeLocal<Object> AcceptHandle(Environment* env,
27.                                         LibuvStreamWrap* parent)
28. {
29.     EscapableHandleScope scope(env->isolate());
30.     Local<Object> wrap_obj;
31.     // 根据类型创建一个表示客户端的对象，然后把文件描述符保存其中
32.     if (!WrapType::Instantiate(env, parent, WrapType::SOCKET).ToLocal(&wrap_obj))
33.         return Local<Object>();
34.     // 解出 C++ 层对象
35.     HandleWrap* wrap = Unwrap<HandleWrap>(wrap_obj);
36.     CHECK_NOT_NULL(wrap);
37.     // 拿到 C++ 对象中封装的 handle
38.     uv_stream_t* stream = reinterpret_cast<uv_stream_t*>(wrap->GetHandle());
39.     // 从服务器流中摘下一个 fd 保存到 stream
40.     if (uv_accept(parent->stream(), stream))
41.         ABORT();
42.     return scope.Escape(wrap_obj);
43. }
44.
45. // 实现 OnUvRead，流中有数据或读到结尾时由 Libuv 回调
46. void LibuvStreamWrap::OnUvRead(ssize_t nread, const uv_buf_t* buf) {
47.     HandleScope scope(env()->isolate());
48.     Context::Scope context_scope(env()->context());
49.     uv_handle_type type = UV_UNKNOWN_HANDLE;
50.     // 是否支持传递文件描述符并且有待处理的文件描述符，则判断文件描述符
51.     // 类型
52.     if (is_named_pipe_ipc() &&
53.         uv_pipe_pending_count(reinterpret_cast<uv_pipe_t*>(stream(
54.             ))) > 0) {
55.         type = uv_pipe_pending_type(reinterpret_cast<uv_pipe_t*>(stream(
56.             )));
57.     }
58.     MaybeLocal<Object> pending_obj;
59.     // 根据类型创建一个新的 C++ 对象表示客户端，并且从服务器中摘下一个
      fd 保存到客户端

```

```
60.     if (type == UV_TCP) {
61.         pending_obj = AcceptHandle<TCPWrap>(env(), this);
62.     } else if (type == UV_NAMED_PIPE) {
63.         pending_obj = AcceptHandle<PipeWrap>(env(), this);
64.     } else if (type == UV_UDP) {
65.         pending_obj = AcceptHandle<UDPWrap>(env(), this);
66.     } else {
67.         CHECK_EQ(type, UV_UNKNOWN_HANDLE);
68.     }
69.     // 有需要处理的文件描述符则设置到 JS 层对象中，JS 层使用
70.     if (!pending_obj.IsEmpty()) {
71.         object()
72.             ->Set(env()->context(),
73.                   env()->pending_handle_string(),
74.                   pending_objToLocalChecked())
75.             .Check();
76.     }
77. }
78. // 触发读事件，listener 实现
79. EmitRead(nread, *buf);
80. }
```

读操作不仅支持读取一般的数据，还可以读取文件描述符，C++层会新建一个流对象表示该文件描述符。在 JS 层可以使用。

6.8.4 ConnectionWrap

ConnectionWrap 是 LibuvStreamWrap 子类，拓展了连接的接口。适用于带有连接属性的流，比如 Unix 域和 TCP。

```
1. // WrapType 是 C++层的类，UVType 是 Libuv 的类型
2. template <typename WrapType, typename UVType>
3. class ConnectionWrap : public LibuvStreamWrap {
4. public:
5.     static void OnConnection(uv_stream_t* handle, int status);
6.     static void AfterConnect(uv_connect_t* req, int status);
7.
8. protected:
9.     ConnectionWrap(Environment* env,
10.                  v8::Local<v8::Object> object,
11.                  ProviderType provider);
12.
13.     UVType handle_;
14. };
```



```
5. // 拿到 Libuv 结构体对应的 C++ 层对象
6. WrapType* wrap_data = static_cast<WrapType*>(handle->data);
7. Environment* env = wrap_data->env();
8. HandleScope handle_scope(env->isolate());
9. Context::Scope context_scope(env->context());
10.
11. // 和客户端通信的对象
12. Local<Value> client_handle;
13.
14. if (status == 0) {
15.     // Instantiate the client javascript object and handle.
16.     // 新建一个 JS 层使用对象
17.     Local<Object> client_obj;
18.     if (!WrapType::Instantiate(env, wrap_data, WrapType::SOCKET)
19.         .ToLocal(&client_obj))
20.         return;
21.
22.     // Unwrap the client javascript object.
23.     WrapType* wrap;
24.     // 把 JS 层使用的对象 client_obj 所对应的 C++ 层对象存到 wrap 中
25.     ASSIGN_OR_RETURN_UNWRAP(&wrap, client_obj);
26.     // 拿到对应的 handle
27.     uv_stream_t* client = reinterpret_cast<uv_stream_t*>(&wrap->
28.     handle_);
29.     // 从 handleaccept 到的 fd 中拿一个保存到 client, client 就可以和客
户端通信了
30.     if (uv_accept(handle, client))
31.         return;
32.     client_handle = client_obj;
33. } else {
34.     client_handle = Undefined(env->isolate());
35. }
36. // 回调 JS, client_handle 相当于在 JS 层执行 new TCP
37. Local<Value> argv[] = {
38.     Integer::New(env->isolate(), status),
39.     client_handle
40. };
41. wrap_data->MakeCallback(env->onconnection_string(),
42.                           arraysize(argv),
43.                           argv);
44. }
```

我们看一下 TCP 的 Instantiate。

```

1. MaybeLocal<Object> TCPWrap::Instantiate(Environment* env,
2.                                         AsyncWrap* parent,
3.                                         TCPWrap::SocketType type)
4. {
5.     EscapableHandleScope handle_scope(env->isolate());
6.     AsyncHooks::DefaultTriggerAsyncIdScope trigger_scope(parent);
7. 
8.     Local<Function> constructor = env->tcp_constructor_template()
9.                             ->GetFunction(env->context())
10.                            .ToLocalChecked();
11.    Local<Value> type_value = Int32::New(env->isolate(), type);
12. 
13.    return handle_scope.EscapeMaybe(
14.        constructor->NewInstance(env->context(), 1, &type_value));
15. }

```

6.8.5 StreamReq

StreamReq 表示操作流的一次请求。主要保存了请求上下文和操作结束后的通用逻辑。

```

1. // 请求 Libuv 的基类
2. class StreamReq {
3. public:
4.     // JS 层传进来的对象的 internalField[1] 保存了 StreamReq 类对象
5.     static constexpr int kStreamReqField = 1;
6.     // stream 为所操作的流, req_wrap_obj 为 JS 层传进来的对象
7.     explicit StreamReq(StreamBase* stream,
8.                         v8::Local<v8::Object> req_wrap_obj) : stream_
9.     _{stream} {
10.         // JS 层对象指向当前 StreamReq 对象
11.         AttachToObject(req_wrap_obj);
12.     }
13.     // 子类定义
14.     virtual AsyncWrap* GetAsyncWrap() = 0;
15.     // 获取相关联的原始 js 对象
16.     v8::Local<v8::Object> object();
17.     // 请求结束后的回调, 会执行子类的 onDone, onDone 由子类实现
18.     void Done(int status, const char* error_str = nullptr);
19.     // JS 层对象不再执行 StreamReq 实例
20.     void Dispose();
21.     // 获取所操作的流
22.     inline StreamBase* stream() const { return stream_; }
23.     // 从 JS 层对象获取 StreamReq 对象

```

```
23. static StreamReq* FromObject(v8::Local<v8::Object> req_wrap_obj);
24. // 请求 JS 层对象的 internalField 所有指向
25. static inline void ResetObject(v8::Local<v8::Object> req_wrap_obj);
26.
27. protected:
28. // 请求结束后回调
29. virtual void OnDone(int status) = 0;
30. void AttachToObject(v8::Local<v8::Object> req_wrap_obj);
31.
32. private:
33. StreamBase* const stream_;
34. };
```

StreamReq 有一个成员为 stream_，表示 StreamReq 请求中操作的流。下面我们看一下实现。

1 JS 层请求上下文和 StreamReq 的关系管理。

```
1. inline void StreamReq::AttachToObject(v8::Local<v8::Object> req_wrap_obj) {
2.     req_wrap_obj->SetAlignedPointerInInternalField(kStreamReqField,
3.                                                     this);
4.
5. inline StreamReq* StreamReq::FromObject(v8::Local<v8::Object> req_wrap_obj) {
6.     return static_cast<StreamReq*>(
7.         req_wrap_obj->GetAlignedPointerFromInternalField(kStreamReqField));
8.
9.
10. inline void StreamReq::Dispose() {
11.     object()->SetAlignedPointerInInternalField(kStreamReqField, nullptr);
12.     delete this;
13.
14.
15. inline void StreamReq::ResetObject(v8::Local<v8::Object> obj) {
16.     obj->SetAlignedPointerInInternalField(0, nullptr); // BaseObject field.
17.     obj->SetAlignedPointerInInternalField(StreamReq::kStreamReqField, nullptr);
18. }
```

2 获取原始 JS 层请求对象

```

1. // 获取和该请求相关联的原始 js 对象
2. inline v8::Local<v8::Object> StreamReq::object() {
3.     return GetAsyncWrap()->object();
4. }
```

3 请求结束回调

```

1. inline void StreamReq::Done(int status, const char* error_str) {

2.     AsyncWrap* async_wrap = GetAsyncWrap();
3.     Environment* env = async_wrap->env();
4.     if (error_str != nullptr) {
5.         async_wrap->object()->Set(env->context(),
6.                                         env->error_string(),
7.                                         OneByteString(env->isolate(),
8.                                                       error_str))
9.                                         .Check();
10.    }
11.    // 执行子类的 OnDone
12.    OnDone(status);
13. }
```

流操作请求结束后会统一执行 Done， Done 会执行子类实现的 OnDone 函数。

6.8.6 ShutdownWrap

ShutdownWrap 是 StreamReq 的子类，表示一次关闭流请求。

```

1. class ShutdownWrap : public StreamReq {
2. public:
3.     ShutdownWrap(StreamBase* stream,
4.                  v8::Local<v8::Object> req_wrap_obj)
5.     : StreamReq(stream, req_wrap_obj) { }
6.
7.     void OnDone(int status) override;
8. }
```

ShutdownWrap 实现了 OnDone 接口，在关闭流结束后被基类执行。

```

1. /*
2.  关闭结束时回调，由请求类 (ShutdownWrap) 调用 Libuv，
3.  所以 Libuv 操作完成后，首先执行请求类的回调，请求类通知流，流触发
4.  对应的事件，进一步通知 listener
```

```
5. */
6. inline void ShutdownWrap::OnDone(int status) {
7.     stream()->EmitAfterShutdown(this, status);
8.     Dispose();
9. }
```

6.8.7 SimpleShutdownWrap

SimpleShutdownWrap 是 ShutdownWrap 的子类。实现了 GetAsyncWrap 接口。OtherBase 可以是 ReqWrap 或者 AsyncWrap。

```
1. template <typename OtherBase>
2. class SimpleShutdownWrap : public ShutdownWrap, public OtherBase
3. {
4.     SimpleShutdownWrap(StreamBase* stream,
5.                         v8::Local<v8::Object> req_wrap_obj);
6.
7.     AsyncWrap* GetAsyncWrap() override { return this; }
8. };
```

6.8.8 WriteWrap

WriteWrap 是 StreamReq 的子类，表示一次往流写入数据的请求。

```
1. class WriteWrap : public StreamReq {
2.     public:
3.     void SetAllocatedStorage(AllocatedBuffer&& storage);
4.
5.     WriteWrap(StreamBase* stream,
6.               v8::Local<v8::Object> req_wrap_obj)
7.     : StreamReq(stream, req_wrap_obj) { }
8.
9.     void OnDone(int status) override;
10.
11.     private:
12.     AllocatedBuffer storage_;
13. };
```

WriteWrap 实现了 OnDone 接口，在写结束时被基类执行。

```
1. inline void WriteWrap::OnDone(int status) {
2.     stream()->EmitAfterWrite(this, status);
```

```

3.   Dispose();
4. }
```

请求结束后调用流的接口通知流写结束了，流会通知 listener，listener 会调用流的接口通知 JS 层。

6.8.9 SimpleWriteWrap

SimpleWriteWrap 是 WriteWrap 的子类。实现了 GetAsyncWrap 接口。和 SimpleShutdownWrap 类型。

```

1. template <typename OtherBase>
2. class SimpleWriteWrap : public WriteWrap, public OtherBase {
3. public:
4.   SimpleWriteWrap(StreamBase* stream,
5.                   v8::Local<v8::Object> req_wrap_obj);
6.
7.   AsyncWrap* GetAsyncWrap() override { return this; }
8. };
```

6.8.10 StreamListener

```

1. class StreamListener {
2. public:
3.   virtual ~StreamListener();
4.   // 分配存储数据的内存
5.   virtual uv_buf_t OnStreamAlloc(size_t suggested_size) = 0;
6.   // 有数据可读时回调，消费数据的函数
7.   virtual void OnStreamRead(ssize_t nread, const uv_buf_t& buf) = 0;
8.   // 流销毁时回调
9.   virtual void OnStreamDestroy() {}
10.  // 监听者所属流
11.  inline StreamResource* stream() { return stream_; }
12.
13. protected:
14.   // 流是监听者是一条链表，该函数把结构传递给下一个节点
15.   void PassReadErrorToPreviousListener(ssize_t nread);
16.   // 监听者所属流
17.   StreamResource* stream_ = nullptr;
18.   // 下一个节点，形成链表
19.   StreamListener* previous_listener_ = nullptr;
20.   friend class StreamResource;
21. };
```

StreamListener 是类似一个订阅者，它会对流的状态感兴趣，比如数据可读、可写、流关闭等。一个流可以注册多个 listener，多个 listener 形成一个链表。

```
1. // 从 listen 所属的流的 listener 队列中删除自己
2. inline StreamListener::~StreamListener() {
3.     if (stream_ != nullptr)
4.         stream_->RemoveStreamListener(this);
5. }
6. // 读出错，把信息传递给前一个 listener
7. inline void StreamListener::PassReadErrorToPreviousListener(ssize
   _t nread) {
8.     CHECK_NOT_NULL(previous_listener_);
9.     previous_listener_->OnStreamRead(nread, uv_buf_init(nullptr, 0)
   );
10. }
11. // 实现流关闭时的处理逻辑
12. inline void StreamListener::OnStreamAfterShutdown(ShutdownWrap*
   w, int status) {
13.     previous_listener_->OnStreamAfterShutdown(w, status);
14. }
15. // 实现写结束时的处理逻辑
16. inline void StreamListener::OnStreamAfterWrite(WriteWrap* w, int
   status) {
17.     previous_listener_->OnStreamAfterWrite(w, status);
18. }
```

StreamListener 的逻辑不多，具体的实现在子类。

6.8.11 ReportWritesToJSStreamListener

ReportWritesToJSStreamListener 是 StreamListener 的子类。覆盖了部分接口和拓展了一些功能。

```
1. class ReportWritesToJSStreamListener : public StreamListener {
2.     public:
3.         // 实现父类的这两个接口
4.         void OnStreamAfterWrite(WriteWrap* w, int status) override;
5.         void OnStreamAfterShutdown(ShutdownWrap* w, int status) override;
6.
7.     private:
8.         void OnStreamAfterReqFinished(StreamReq* req_wrap, int status);
9. }
```

1 OnStreamAfterReqFinished

OnStreamAfterReqFinished 是请求操作流结束后的统一的回调。

```

1. void ReportWritesToJSStreamListener::OnStreamAfterWrite(
2.     WriteWrap* req_wrap, int status) {
3.     OnStreamAfterReqFinished(req_wrap, status);
4. }
5.
6. void ReportWritesToJSStreamListener::OnStreamAfterShutdown(
7.     ShutdownWrap* req_wrap, int status) {
8.     OnStreamAfterReqFinished(req_wrap, status);
9. }
```

我们看一下具体实现

```

1. void ReportWritesToJSStreamListener::OnStreamAfterReqFinished(
2.     StreamReq* req_wrap, int status) {
3.     // 请求所操作的流
4.     StreamBase* stream = static_cast<StreamBase*>(stream_);
5.     Environment* env = stream->stream_env();
6.     AsyncWrap* async_wrap = req_wrap->GetAsyncWrap();
7.     HandleScope handle_scope(env->isolate());
8.     Context::Scope context_scope(env->context());
9.     // 获取原始的 JS 层对象
10.    Local<Object> req_wrap_obj = async_wrap->object();
11.
12.    Local<Value> argv[] = {
13.        Integer::New(env->isolate(), status),
14.        stream->GetObject(),
15.        Undefined(env->isolate())
16.    };
17.
18.    const char* msg = stream->Error();
19.    if (msg != nullptr) {
20.        argv[2] = OneByteString(env->isolate(), msg);
21.        stream->ClearError();
22.    }
23.    // 回调 JS 层
24.    if (req_wrap_obj->Has(env->context(), env->oncomplete_string())
25.        .FromJust())
26.        async_wrap->MakeCallback(env->oncomplete_string(), arraysize
27.            (argv), argv);
28. }
```

OnStreamAfterReqFinished 会回调 JS 层。

6.8.12 EmitToJSStreamListener

EmitToJSStreamListener 是 ReportWritesToJSStreamListener 的子类

```
1. class EmitToJSStreamListener : public ReportWritesToJSStreamListener {
2.     public:
3.     uv_buf_t OnStreamAlloc(size_t suggested_size) override;
4.     void OnStreamRead(ssize_t nread, const uv_buf_t& buf) override;
5. }
```

我们看一下实现

```
1. // 分配一块内存
2. uv_buf_t EmitToJSStreamListener::OnStreamAlloc(size_t suggested_size) {
3.     Environment* env = static_cast<StreamBase*>(stream_)->stream_env();
4.     return env->AllocateManaged(suggested_size).release();
5. }
6. // 读取数据结束后回调
7. void EmitToJSStreamListener::OnStreamRead(ssize_t nread, const uv_buf_t& buf_) {
8.     StreamBase* stream = static_cast<StreamBase*>(stream_);
9.     Environment* env = stream->stream_env();
10.    HandleScope handle_scope(env->isolate());
11.    Context::Scope context_scope(env->context());
12.    AllocatedBuffer buf(env, buf_);
13.    // 读取失败
14.    if (nread <= 0) {
15.        if (nread < 0)
16.            stream->CallJSOnreadMethod(nread, Local<ArrayBuffer>());
17.        return;
18.    }
19.
20.    buf.Resize(nread);
21.    // 读取成功回调 JS 层
22.    stream->CallJSOnreadMethod(nread, buf.To ArrayBuffer());
23. }
```

我们看到 listener 处理完数据后又会回调流的接口，具体的逻辑由子类实现。我们来看一个子类的实现（流默认的 listener）。

```

1. class EmitToJSStreamListener : public ReportWritesToJSStreamListe
   ner {
2. public:
3.     uv_buf_t OnStreamAlloc(size_t suggested_size) override;
4.     void OnStreamRead(ssize_t nread, const uv_buf_t& buf) override;
5. };

```

EmitToJSStreamListener 会实现 OnStreamRead 等方法，接着我们看一下创建一个 C++ 层的 TCP 对象是怎样的。下面是 TCPWrap 的继承关系。

```

1. class TCPWrap : public ConnectionWrap<TCPWrap, uv_tcp_t>{}
2. // ConnectionWrap 拓展了建立 TCP 连接时的逻辑
3. class ConnectionWrap : public LibuvStreamWrap{}
4. class LibuvStreamWrap : public HandleWrap, public StreamBase{}
5. class StreamBase : public StreamResource {}

```

我们看到 TCP 流是继承于 StreamResource 的。新建一个 TCP 的 C++ 的对象时（tcp_wrap.cc），会不断往上调用父类的构造函数，其中在 StreamBase 中有一个关键的操作。

```

1. inline StreamBase::StreamBase(Environment* env) : env_(env) {
2.     PushStreamListener(&default_listener_);
3. }
4.
5. EmitToJSStreamListener default_listener_;

```

StreamBase 会默认给流注册一个 listener。我们看下 EmitToJSStreamListener 具体的定义。

```

1. class ReportWritesToJSStreamListener : public StreamListener {
2. public:
3.     void OnStreamAfterWrite(WriteWrap* w, int status) override;
4.     void OnStreamAfterShutdown(ShutdownWrap* w, int status) overrid
   e;
5.
6. private:
7.     void OnStreamAfterReqFinished(StreamReq* req_wrap, int status);

8. };
9.
10. class EmitToJSStreamListener : public ReportWritesToJSStreamList
    ener {
11. public:
12.     uv_buf_t OnStreamAlloc(size_t suggested_size) override;

```

```
13. void OnStreamRead(ssize_t nread, const uv_buf_t& buf) override
14. ;
```

EmitToJSStreamListener 继承 StreamListener，定义了分配内存和读取接收数据的函数。接着我们看一下 PushStreamListener 做了什么事情。

```
1. inline void StreamResource::PushStreamListener(StreamListener* listener) {
2.     // 头插法
3.     listener->previous_listener_ = listener_;
4.     listener->stream_ = this;
5.     listener_ = listener;
6. }
```

PushStreamListener 就是构造出一个 `listener` 链表结构。然后我们看一下对于流来说，读取数据的整个链路。首先是 JS 层调用 `readStart`

```
1. function tryReadStart(socket) {
2.     socket._handle.reading = true;
3.     const err = socket._handle.readStart();
4.     if (err)
5.         socket.destroy(errnoException(err, 'read'));
6. }
7.
8. // 注册等待读事件
9. Socket.prototype._read = function(n) {
10.     tryReadStart(this);
11. };
```

我们看看 `readStart`

```
1. int LibuvStreamWrap::ReadStart() {
2.     return uv_read_start(stream(), [](uv_handle_t* handle,
3.                                     size_t suggested_size,
4.                                     uv_buf_t* buf) {
5.         static_cast<LibuvStreamWrap*>(handle->data)->OnUvAlloc(suggested_size, buf);
6.     }, [](uv_stream_t* stream, ssize_t nread, const uv_buf_t* buf) {
7.     static_cast<LibuvStreamWrap*>(stream->data)->OnUvRead(nread, buf);
8. });
9. }
```

ReadStart 调用 Libuv 的 uv_read_start 注册等待可读事件，并且注册了两个回调函数 OnUvAlloc 和 OnUvRead。

```

1. void LibuvStreamWrap::OnUvRead(ssize_t nread, const uv_buf_t* buf
) {
2.     EmitRead(nread, *buf);
3. }
4.
5. inline void StreamResource::EmitRead(ssize_t nread, const uv_buf_
t& buf) {
6.     // bytes_read_ 表示已读的字节数
7.     if (nread > 0)
8.         bytes_read_ += static_cast<uint64_t>(nread);
9.     listener_->OnStreamRead(nread, buf);
10. }
```

通过层层调用最后会调用 listener_ 的 OnStreamRead。我们看看 TCP 的 OnStreamRead

```

1. void EmitToJSStreamListener::OnStreamRead(ssize_t nread, const uv
    _buf_t& buf_) {
2.     StreamBase* stream = static_cast<StreamBase*>(stream_);
3.     Environment* env = stream->stream_env();
4.     HandleScope handle_scope(env->isolate());
5.     Context::Scope context_scope(env->context());
6.     AllocatedBuffer buf(env, buf_);
7.     stream->CallJSOnreadMethod(nread, buf.ToArrayBuffer());
8. }
```

继续回调 CallJSOnreadMethod

```

1. MaybeLocal<Value> StreamBase::CallJSOnreadMethod(ssize_t nread,
2.                                                 Local<ArrayBuffe
    r> ab,
3.                                                 size_t offset,
4.                                                 StreamBaseJSChc
    ks checks) {
5.     Environment* env = env_;
6.     // ...
7.     AsyncWrap* wrap = GetAsyncWrap();
8.     CHECK_NOT_NULL(wrap);
9.     Local<Value> onread = wrap->object()->GetInternalField(kOnReadF
    unctionField);
10.    CHECK(onread->IsFunction());
11.    return wrap->MakeCallback(onread.As<Function>(), arraysize(arg
    v), argv);
12. }
```

CallJSOnreadMethod 会回调 JS 层的 onread 回调函数。onread 会把数据 push 到流中，然后触发 data 事件。

第七章 信号处理

7.1 信号的概念和实现原理

信号是进程间通信的一种简单的方式，我们首先了解一下信号的概念和在操作系统中的实现原理。在操作系统内核的实现中，每个进程对应一个 `task_struct` 结构体（PCB），PCB 中有一个字段记录了进程收到的信号（每一个比特代表一种信号）和信号对应的处理函数。这个和订阅者/发布者模式非常相似，我们看一下 PCB 中信号对应的数据结构。

```
1. struct task_struct {  
2.     // 收到的信号  
3.     long signal;  
4.     // 处理信号过程中屏蔽的信息  
5.     long blocked;  
6.     // 信号对应的处理函数  
7.     struct sigaction sigaction[32];  
8.     ...  
9. };  
10.  
11. struct sigaction {  
12.     // 信号处理函数  
13.     void (*sa_handler)(int);  
14.     // 处理信号时屏蔽哪些信息，和 PCB 的 block 字段对应  
15.     sigset_t sa_mask;  
16.     // 一些标记，比如处理函数只执行一次，类似 events 模块的 once  
17.     int sa_flags;  
18.     // 清除调用栈信息，glibc 使用  
19.     void (*sa_restorer)(void);  
20. };
```

Linux 下支持多种信号，进程收到信号时，操作系统提供了默认处理，我们也可以显式注册处理信号的函数，但是有些信号会导致进程退出，这是我们无法控制的。我们来看一下在 Linux 下信号使用的例子。

```
1. #include <stdio.h>  
2. #include <unistd.h>  
3. #include <stdlib.h>
```

```
4. #include <signal.h>
5.
6. void handler(int);
7.
8. int main()
9. {
10.     signal(SIGINT, handler);
11.     while(1);
12.     return(0);
13. }
14.
15. void sighandler(int signum)
16. {
17.     printf("收到信号%d", signum);
18. }
```

我们注册了一个信号对应的处理函数，然后进入 while 循环保证进程不会退出，这时候，如果我们给这个进程发送一个 SIGINT 信号（ctrl+c 或者 kill -2 pid）。则进程会执行对应的回调，然后输出：收到信号 2。了解了信号的基本原理后，我们看一下 Libuv 中关于信号的设计和实现。

7.2 Libuv 信号处理的设计思想

由于操作系统实现的限制，我们无法给一个信号注册多个处理函数，对于同一个信号，如果我们调用操作系统接口多次，后面的就会覆盖前面设置的值。想要实现一个信号被多个函数处理，我们只能在操作系统之上再封装一层，**Libuv** 正是这样做的。**Libuv** 中关于信号处理的封装和订阅者/发布者模式很相似。用户调用 **Libuv** 的接口注册信号处理函数，**Libuv** 再向操作系统注册对应的处理函数，等待操作系统收到信号时，会触发 **Libuv** 的回调，**Libuv** 的回调会通过管道通知事件循环收到的信号和对应的上下文，接着事件循环在 Poll IO 阶段就会处理收到所有信号以及对应的处理函数。整体架构如图 7-1 所示

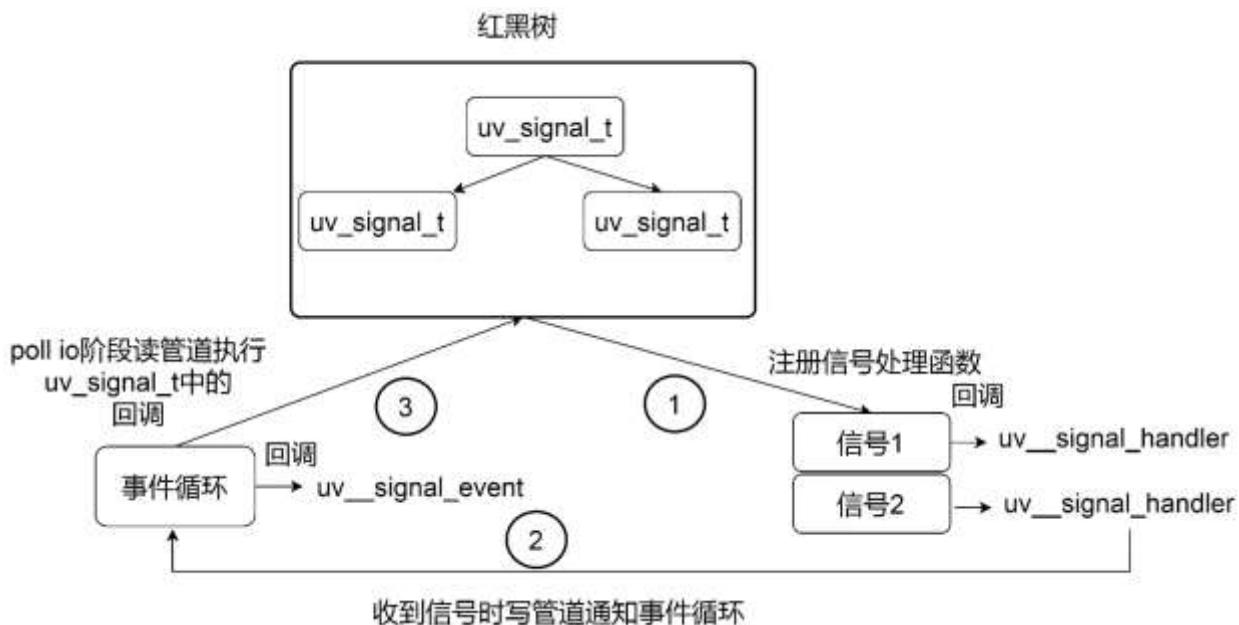


图 7-1

下面我们具体分析 Libuv 中信号处理的实现。

7.3 通信机制的实现

当进程收到信号的时候，信号处理函数需要通知 Libuv 事件循环，从而在事件循环中执行对应的回调，实现函数是 `uv_signal_loop_once_init`，我们看一下 `uv_signal_loop_once_init` 的逻辑。

```
1. static int uv_signal_loop_once_init(uv_loop_t* loop) {
2.     /*
3.         申请一个管道用于和事件循环通信，通知事件循环是否收到信号，
4.         并设置非阻塞标记
5.     */
6.     uv_make_pipe(loop->signal_pipefd, UV_F_NONBLOCK);
7.     /*
8.         设置信号 I/O 观察者的处理函数和文件描述符，
9.         Libuv 在 Poll IO 时，发现管道读端 loop->signal_pipefd[0] 可读，
10.        则执行 uv_signal_event
11.    */
12.    uv_io_init(&loop->signal_io_watcher,
13.               uv_signal_event,
14.               loop->signal_pipefd[0]);
15.    */
```

```
16.     插入 Libuv 的 IO 观察者队列，并注册感兴趣的事件为可读
17.     */
18.     uv__io_start(loop, &loop->signal_io_watcher, POLLIN);
19.
20.     return 0;
21. }
```

`uv_signal_loop_once_init` 首先申请一个管道，用于通知事件循环是否收到信号。然后往 Libuv 的 IO 观察者队列注册一个观察者，Libuv 在 Poll IO 阶段会把观察者加到 `epoll1` 中。IO 观察者里保存了管道读端的文件描述符 `loop->signal_pipefd[0]` 和回调函数 `uv_signal_event`。`uv_signal_event` 是收到任意信号时的回调，它会继续根据收到的信号进行逻辑分发。执行完的架构如图 7-2 所示。

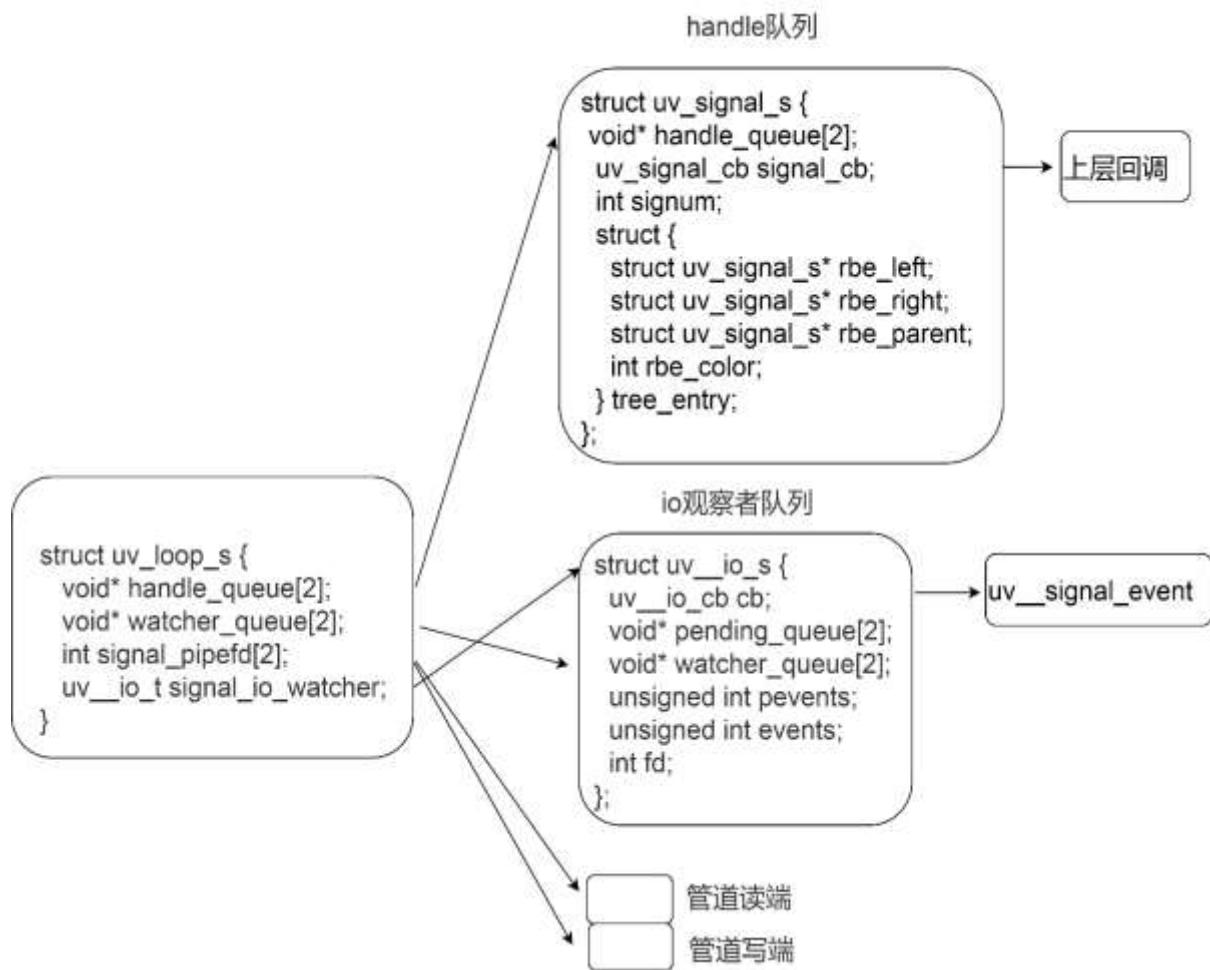


图 7-2

7.4 信号结构体的初始化

Libuv 中信号使用 uv_signal_t 表示。

```
1. int uv_signal_init(uv_loop_t* loop, uv_signal_t* handle) {
2.     // 申请和 Libuv 的通信管道并且注册 IO 观察者
3.     uv__signal_loop_once_init(loop);
4.     uv_handle_init(loop, (uv_handle_t*) handle, UV_SIGNAL);
5.     handle->signum = 0;
6.     handle->caught_signals = 0;
7.     handle->dispatched_signals = 0;
8.
9.     return 0;
10. }
```

上面的代码的逻辑比较简单，只是初始化 uv_signal_t 结构体的一些字段。

7.5 信号处理的注册

我们可以通过 uv_signal_start 注册一个信号处理函数。我们看看这个函数的逻辑

```
1. static int uv_signal_start(uv_signal_t* handle,
2.                             uv_signal_cb signal_cb,
3.                             int signum,
4.                             int oneshot) {
5.     sigset_t saved_sigmask;
6.     int err;
7.     uv_signal_t* first_handle;
8.     // 注册过了，重新设置处理函数就行
9.     if (signum == handle->signum) {
10.         handle->signal_cb = signal_cb;
11.         return 0;
12.     }
13.     // 这个 handle 之前已经设置了其它信号和处理函数，则先解除
14.     if (handle->signum != 0) {
15.         uv_signal_stop(handle);
16.     }
17.     // 屏蔽所有信号
18.     uv_signal_block_and_lock(&saved_sigmask);
19. /*
```

```

20.     查找注册了该信号的第一个 handle,
21.     优先返回设置了 UV_SIGNAL_ONE_SHOT flag 的,
22.     见 compare 函数
23. */
24. first_handle = uv_signal_first_handle(signum);
25. /*
26.     1 之前没有注册过该信号的处理函数则直接设置
27.     2 之前设置过, 但是是 one shot, 但是现在需要
28.         设置的规则不是 one shot, 需要修改。否则第
29.         二次会不会触发。因为一个信号只能对应一
30.         个信号处理函数, 所以, 以规则宽的为准, 在回调
31.         里再根据 flags 判断是不是真的需要执行
32.     3 如果注册过信号和处理函数, 则直接插入红黑树就行。
33. */
34. if (
35.     first_handle == NULL ||
36.     (!oneshot && (first_handle->flags & UV_SIGNAL_ONE_SHOT))
37. ) {
38.     // 注册信号和处理函数
39.     err = uv_register_handler(signum, oneshot);
40.     if (err) {
41.         uv_unlock_and_unblock(&saved_sigmask);
42.         return err;
43.     }
44. }
45. // 记录感兴趣的信号
46. handle->signum = signum;
47. // 只处理该信号一次
48. if (oneshot)
49.     handle->flags |= UV_SIGNAL_ONE_SHOT;
50. // 插入红黑树
51. RB_INSERT(uv_signal_tree_s, &uv_signal_tree, handle);
52. uv_unlock_and_unblock(&saved_sigmask);
53. // 信号触发时的业务层回调
54. handle->signal_cb = signal_cb;
55. uv_handle_start(handle);
56.
57. return 0;
58. }

```

上面的代码比较多, 大致的逻辑如下

1 判断是否需要向操作系统注册一个信号的处理函数。主要是调用操作系统的函数来处理的，代码如下

```
1. // 给当前进程注册信号处理函数，会覆盖之前设置的 signum 的处理函数
2. static int uv_signal_register_handler(int signum, int oneshot) {
3.     struct sigaction sa;
4.
5.     memset(&sa, 0, sizeof(sa));
6.     // 全置一，说明收到 signum 信号的时候，暂时屏蔽其它信号
7.     if (sigfillset(&sa.sa_mask))
8.         abort();
9.     // 所有信号都由该函数处理
10.    sa.sa_handler = uv_signal_handler;
11.    sa.sa_flags = SA_RESTART;
12.    // 设置了 oneshot，说明信号处理函数只执行一次，然后被恢复为系统的默认处理函数
13.    if (oneshot)
14.        sa.sa_flags |= SA_RESETHAND;
15.
16.    // 注册
17.    if (sigaction(signum, &sa, NULL))
18.        return UV_ERR(errno);
19.
20.    return 0;
21. }
```

我们看到所有信号的处理函数都是 `uv_signal_handler`，我们一会会分析 `uv_signal_handler` 的实现。

2 进程注册的信号和回调是在一棵红黑树管理的，每次注册的时候会往红黑树插入一个节点。Libuv 用黑红树维护信号的上下文，插入的规则是根据信号的大小和 flags 等信息。RB_INSERT 实现了往红黑树插入一个节点，红黑树中的节点是父节点的值比左孩子大，比右孩子小的。执行完 RB_INSERT 后的架构如图 7-3 所示。

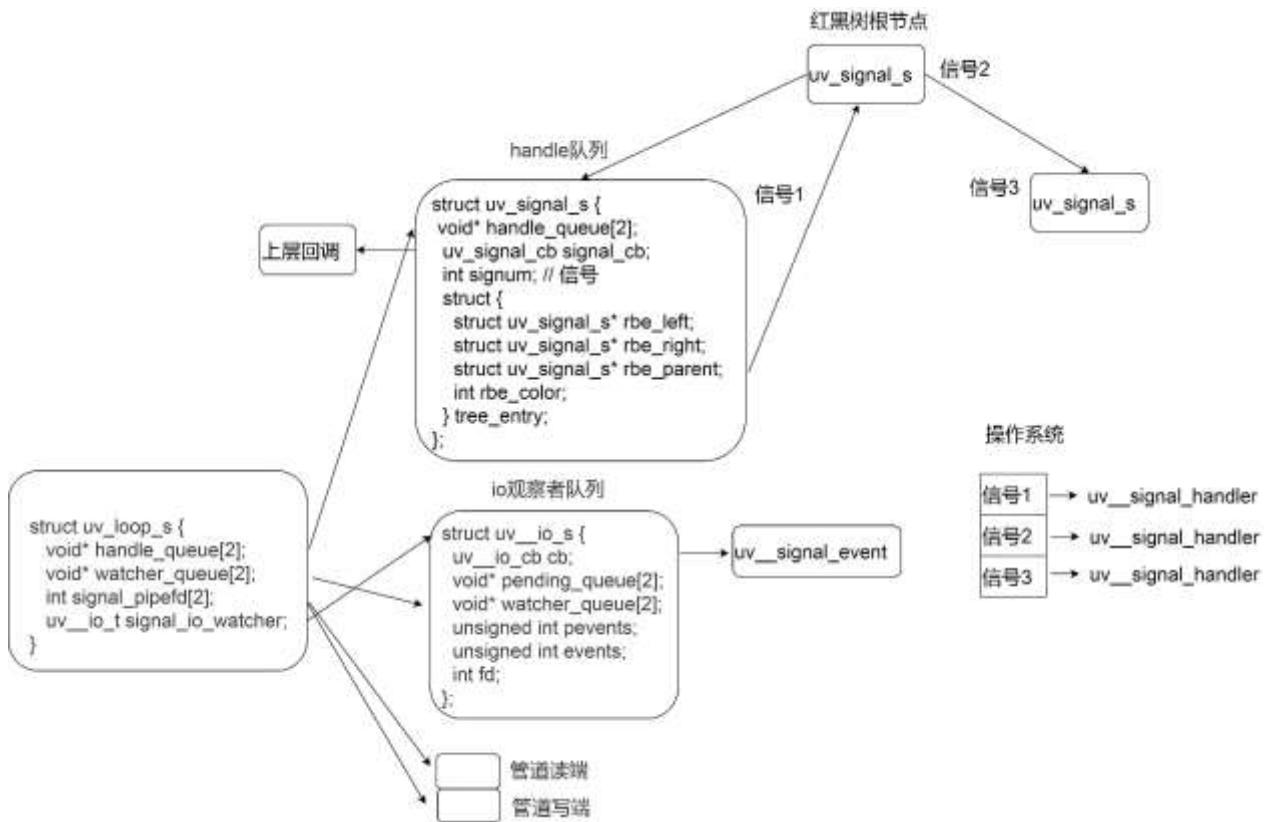


图 7-3

我们看到，当我们每次插入不同的信号的时候，Libuv 会在操作系统和红黑树中修改对应的数据结构。那么如果我们插入重复的信号呢？刚才我们已经分析过，插入重复的信号时，如果在操作系统注册过，并且当前插入的信号 flags 是 one shot，而之前是非 one shot 时，Libuv 会调用操作系统的接口去修改配置。那么对于红黑树来说，插入重复信号会如何处理呢？从刚才 RB_INSERT 的代码中我们看到每次插入红黑树时，红黑树会先判断是否存在相同值的节点，如果是的话直接返回，不进行插入。这么看起来我们无法给一个信号注册多个处理函数，但其实是可以的，重点在比较大小的函数。我们看看该函数的实现。

```

1. static int uv_signal_compare(uv_signal_t* w1, uv_signal_t* w2) {
2.     int f1;
3.     int f2;
4.
5.     // 返回信号值大的
6.     if (w1->signum < w2->signum) return -1;
7.     if (w1->signum > w2->signum) return 1;
8.
9.     // 设置了 UV_SIGNAL_ONE_SHOT 的大
10.    f1 = w1->flags & UV_SIGNAL_ONE_SHOT;

```

```
11.     f2 = w2->flags & UV_SIGNAL_ONE_SHOT;
12.     if (f1 < f2) return -1;
13.     if (f1 > f2) return 1;
14.
15. // 地址大的值就大
16. if (w1->loop < w2->loop) return -1;
17. if (w1->loop > w2->loop) return 1;
18.
19. if (w1 < w2) return -1;
20. if (w1 > w2) return 1;
21.
22. return 0;
23. }
```

我们看到 Libuv 比较的不仅是信号的大小，在信号一样的情况下，Libuv 还会比较其它的因素，除非两个 uv_signal_t 指针指向的是同一个 uv_signal_t 结构体，否则它们是不会被认为重复的，所以红黑树中会存着信号一样的节点。假设我们按照 1 (flags 为 one shot) , 2 (flags 为非 one shot) , 3 (flags 为 one shot) 的顺序插入红黑树，并且节点 3 比节点 1 的地址大。所形成的结构如图 7-4 所示。

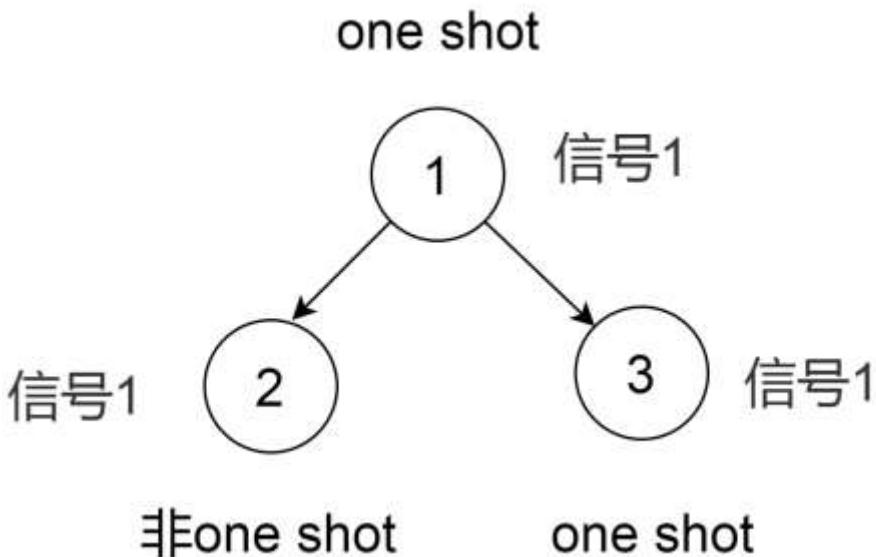


图 7-4

7.6 信号的处理

我们上一节已经分析过，不管注册什么信号，它的处理函数都是这个 `uv_signal_handler` 函数。我们自己的业务回调函数，是保存在 `handle` 里的。而 Libuv

维护了一棵红黑树，记录了每个 handle 注册的信号和回调函数，那么当任意信号到来的时候。uv_signal_handler 就会被调用。下面我们看看 uv_signal_handler 函数。

```

1. /*
2.  信号处理函数, signum 为收到的信号,
3.  每个子进程收到信号的时候都由该函数处理,
4.  然后通过管道通知 Libuv
5. */
6. static void uv_signal_handler(int signum) {
7.     uv_signal_msg_t msg;
8.     uv_signal_t* handle;
9.     int saved_errno;
10.    // 保持上一个系统调用的错误码
11.    saved_errno = errno;
12.    memset(&msg, 0, sizeof msg);
13.
14.    if (uv_signal_lock()) {
15.        errno = saved_errno;
16.        return;
17.    }
18.    // 找到该信号对应的所有 handle
19.    for (handle = uv_signal_first_handle(signum);
20.         handle != NULL && handle->signum == signum;
21.         handle = RB_NEXT(uv_signal_tree_s,
22.                           &uv_signal_tree,
23.                           handle))
24.    {
25.        int r;
26.        // 记录上下文
27.        msg.signum = signum;
28.        msg.handle = handle;
29.        do {
30.            // 通知 Libuv, 哪些 handle 需要处理该信号,
31.            // 在 Poll IO 阶段处理
32.            r = write(handle->loop->signal_pipefd[1],
33.                      &msg,
34.                      sizeof msg);
35.        } while (r == -1 && errno == EINTR);
36.        // 该 handle 收到信号的次数
37.        if (r != -1)
38.            handle->caught_signals++;
39.    }

```

```
40.  
41.     uv_signal_unlock();  
42.     errno = saved_errno;  
43. }
```

uv_signal_handler 函数会调用 uv_signal_first_handle 遍历红黑树，找到注册了该信号的所有 handle，我们看一下 uv_signal_first_handle 的实现。

```
1. static uv_signal_t* uv_signal_first_handle(int signum) {  
2.     uv_signal_t lookup;  
3.     uv_signal_t* handle;  
4.  
5.     lookup.signum = signum;  
6.     lookup.flags = 0;  
7.     lookup.loop = NULL;  
8.  
9.     handle = RB_NFIND(uv_signal_tree_s,  
10.                          &uv_signal_tree,  
11.                          &lookup);  
12.  
13.     if (handle != NULL && handle->signum == signum)  
14.         return handle;  
15.     return NULL;  
16. }
```

uv_signal_first_handle 函数通过 RB_NFIND 实现红黑树的查找，RB_NFIND 是一个宏。

```
1. #define RB_NFIND(name, x, y)    name##_RB_NFIND(x, y)
```

我们看看 name##_RB_NFIND 即 uv_signal_tree_s_RB_NFIND 的实现

```
1. static struct uv_signal_t * uv_signal_tree_s_RB_NFIND(struct uv_  
_signal_tree_s *head, struct uv_signal_t *elm)  
2. {  
3.     struct uv_signal_t *tmp = RB_ROOT(head);  
4.     struct uv_signal_t *res = NULL;  
5.     int comp;  
6.     while (tmp) {  
7.         comp = cmp(elm, tmp);  
8.         /*  
9.             elm 小于当前节点则往左子树找，大于则往右子树找，  
10.            等于则返回  
11.         */
```

```

12.     if (comp < 0) {
13.         // 记录父节点
14.         res = tmp;
15.         tmp = RB_LEFT(tmp, field);
16.     }
17.     else if (comp > 0)
18.         tmp = RB_RIGHT(tmp, field);
19.     else
20.         return (tmp);
21. }
22. return (res);
23. }
```

`uv_signal_tree_s_RB_NFIND` 的逻辑就是根据红黑树的特点进行搜索，这里的重点是 `cmp` 函数。刚才我们已经分析过 `cmp` 的逻辑。这里会首先查找没有设置 `one shot` 标记的 `handle`（因为它的值小），然后再查找设置了 `one shot` 的 `handle`，一旦遇到设置了 `one shot` 的 `handle`，则说明后面被匹配的 `handle` 也是设置了 `one shot` 标记的。每次找到一个 `handle`，就会封装一个 `msg` 写入管道（即和 Libuv 通信的管道）。信号的处理就完成了。接下来在 Libuv 的 Poll IO 阶段才做真正的处理。我们知道在 Poll IO 阶段。`epoll` 会检测到管道 `loop->signal_pipefd[0]` 可读，然后会执行 `uv_signal_event` 函数。我们看看这个函数的代码。

```

1. // 如果收到信号, Libuv Poll IO 阶段, 会执行该函数
2. static void uv_signal_event(uv_loop_t* loop, uv_io_t* w,
3. unsigned int events) {
4.     uv_signal_msg_t* msg;
5.     uv_signal_t* handle;
6.     char buf[sizeof(uv_signal_msg_t) * 32];
7.     size_t bytes, end, i;
8.     int r;
9.
10.    bytes = 0;
11.    end = 0;
12.    // 计算出数据的大小
13.    do {
14.        // 读出所有的 uv_signal_msg_t
15.        r = read(loop->signal_pipefd[0],
16.                  buf + bytes,
17.                  sizeof(buf) - bytes);
18.        if (r == -1 && errno == EINTR)
19.            continue;
20.        if (r == -1 &&
21.            (errno == EAGAIN ||
```

```
22.         errno == EWOULDBLOCK)) {
23.         if (bytes > 0)
24.             continue;
25.         return;
26.     }
27.     if (r == -1)
28.         abort();
29.     bytes += r;
30.     /*
31.      根据收到的字节数算出有多少个 uv_signal_msg_t 结构体,
32.      从而算出结束位置
33.     */
34.     end=(bytes/sizeof(uv_signal_msg_t))*sizeof(uv_signal_msg_t);
35.     // 循环处理每一个 msg
36.     for (i = 0; i < end; i += sizeof(uv_signal_msg_t)) {
37.         msg = (uv_signal_msg_t*) (buf + i);
38.         // 取出上下文
39.         handle = msg->handle;
40.         // 收到的信号和 handle 感兴趣的信号一致, 执行回调
41.         if (msg->signum == handle->signum) {
42.             handle->signal_cb(handle, handle->signum);
43.         }
44.         // 处理信号个数, 和收到的个数对应
45.         handle->dispatched_signals++;
46.         // 只执行一次, 恢复系统默认的处理函数
47.         if (handle->flags & UV_SIGNAL_ONE_SHOT)
48.             uv_signal_stop(handle);
49.         /*
50.          处理完所有收到的信号才能关闭 uv_signal_t,
51.          见 uv_close 或 uv_signal_close
52.        */
53.         if ((handle->flags & UV_HANDLE_CLOSING) &&
54.             (handle->caught_signals==handle->dispatched_signals))
55.         {
56.             uv_make_close_pending((uv_handle_t*) handle);
57.         }
58.     }
59.     bytes -= end;
60.     if (bytes) {
61.         memmove(buf, buf + end, bytes);
62.         continue;
63.     }
```

```

64.     } while (end == sizeof buf);
65. }
```

uv_signal_event 函数的逻辑如下

- 1 读出管道里的数据，计算出 msg 的个数。
- 2 遍历收到的数据，解析出一个个 msg。
- 3 从 msg 中取出上下文 (handle 和信号)，执行上层回调。
- 4 如果 handle 设置了 one shot 则需要执行 uv_signal_stop (我们接下来分析)。
- 5 如果 handle 设置了 closing 标记，则判断所有收到的信号是否已经处理完。即收到的个数和处理的个数是否一致。需要处理完所有收到的信号才能关闭 uv_signal_t。

7.7 取消/关闭信号处理

当一个信号对应的 handle 设置了 one shot 标记，在收到信号并且执行完回调后，Libuv 会调用 uv_signal_stop 关闭该 handle 并且从红黑树中移除该 handle。另外我们也可以显式地调用 uv_close (会调用 uv_signal_stop) 关闭或取消信号的处理。下面我们看看 uv_signal_stop 的实现。

```

1. static void uv_signal_stop(uv_signal_t* handle) {
2.     uv_signal_t* removed_handle;
3.     sigset_t saved_sigmask;
4.     uv_signal_t* first_handle;
5.     int rem_oneshot;
6.     int first_oneshot;
7.     int ret;
8.
9.     /* If the watcher wasn't started, this is a no-op. */
10.    // 没有注册过信号，则不需要处理
11.    if (handle->signum == 0)
12.        return;
13.    // 屏蔽所有信号
14.    uv_signal_block_and_lock(&saved_sigmask);
15.    // 移出红黑树
16.    removed_handle = RB_REMOVE(uv_signal_tree_s, &uv_signal_tree
17.        , handle);
18.    first_handle = uv_signal_first_handle(handle->signum);
19.    // 为空说明没有 handle 会处理该信号了，解除该信号的设置
20.    if (first_handle == NULL) {
21.        uv_signal_unregister_handler(handle->signum);
22.    } else {
23.        // 被处理的 handle 是否设置了 one shot
```

```
24.     rem_oneshot = handle->flags & UV_SIGNAL_ONE_SHOT;
25.     /*
26.         剩下的第一个 handle 是否设置了 one shot,
27.         如果是则说明该信号对应的所有剩下的 handle 都是 one shot
28.     */
29.     first_oneshot = first_handle->flags & UV_SIGNAL_ONE_SHOT;
30.     /*
31.         被移除的 handle 没有设置 oneshot 但是当前的第一个 handle 设置了
32.         one shot, 则需要修改该信号处理函数为 one shot, 防止收到多次信
33.         号, 执行多次回调
34.     */
35.     if (first_oneshot && !rem_oneshot) {
36.         ret = uv_signal_register_handler(handle->signum, 1);
37.         assert(ret == 0);
38.     }
39. }
40.
41. uv_signal_unlock_and_unblock(&saved_sigmask);
42.
43. handle->signum = 0;
44. uv_handle_stop(handle);
45. }
```

7.8 信号在 Node.js 中的使用

分析完 Libuv 的实现后，我们看看 Node.js 上层是如何使用信号的，首先我们看一下 C++ 层关于信号模块的实现。

```
1. static void Initialize(Local<Object> target,
2.                         Local<Value> unused,
3.                         Local<Context> context,
4.                         void* priv) {
5.     Environment* env = Environment::GetCurrent(context);
6.     Local<FunctionTemplate> constructor = env->NewFunctionTemplate(
e(New);
7.     constructor->InstanceTemplate()->SetInternalFieldCount(1);
8.     // 导出的类名
9.     Local<String> signalString =
10.      FIXED_ONE_BYTE_STRING(env->isolate(), "Signal");
11.    constructor->SetClassName(signalString);
12.    constructor->Inherit(HandleWrap::GetConstructorTemplate(env)
 );
13.    // 给 Signal 创建的对象注入两个函数
14.    env->SetProtoMethod(constructor, "start", Start);
```

```

15.     env->SetProtoMethod(constructor, "stop", Stop);
16.
17.     target->Set(env->context(), signalString,
18.                   constructor->GetFunction(env->context()).ToLocal
19.             Checked()).Check();
19. }
```

当我们在 JS 中 new Signal 的时候，首先会创建一个 C++对象，然后作为入参执行 New 函数。

```

1. static void New(const FunctionCallbackInfo<Value>& args) {
2.     CHECK(args.IsConstructCall());
3.     Environment* env = Environment::GetCurrent(args);
4.     new SignalWrap(env, args.This());
5. }
```

当我们在 JS 层操作 Signal 实例的时候，就会执行 C++层对应的方法。主要的方法是注册和删除信号。

```

1. static void Start(const FunctionCallbackInfo<Value>& args) {
2.     SignalWrap* wrap;
3.     ASSIGN_OR_RETURN_UNWRAP(&wrap, args.Holder());
4.     Environment* env = wrap->env();
5.     int signum;
6.     if (!args[0]->Int32Value(env->context()).To(&signum)) return;

7.     int err = uv_signal_start(
8.         &wrap->handle_,
9.         // 信号产生时执行的回调
10.        [] (uv_signal_t* handle, int signum) {
11.            SignalWrap* wrap = ContainerOf(&SignalWrap::handle_,
12.                                            handle);
13.            Environment* env = wrap->env();
14.            HandleScope handle_scope(env->isolate());
15.            Context::Scope context_scope(env->context());
16.            Local<Value> arg = Integer::New(env->isolate(),
17.                                            signum);
18.            // 触发 JS 层 onsignal 函数
19.            wrap->MakeCallback(env->onsignal_string(), 1, &arg);
20.        },
21.        signum);
22.
23.     if (err == 0) {
24.         CHECK(!wrap->active_);
25.         wrap->active_ = true;
26.         Mutex::ScopedLock lock(handled_signals_mutex);
```

```
27.     handled_signals[signum]++;
28. }
29.
30.     args.GetReturnValue().Set(err);
31. }
32.
33. static void Stop(const FunctionCallbackInfo<Value>& args) {
34.     SignalWrap* wrap;
35.     ASSIGN_OR_RETURN_UNWRAP(&wrap, args.Holder());
36.
37.     if (wrap->active_) {
38.         wrap->active_ = false;
39.         DecreaseSignalHandlerCount(wrap->handle_.signum);
40.     }
41.
42.     int err = uv_signal_stop(&wrap->handle_);
43.     args.GetReturnValue().Set(err);
44. }
```

接着我们看在 JS 层如何使用。Node.js 在初始化的时候，在 `is_main_thread.js` 中执行了。

```
1. process.on('newListener', startListeningIfSignal);
2. process.on('removeListener', stopListeningIfSignal)
```

`newListener` 和 `removeListener` 事件在注册和删除事件的时候都会被触发。我们看一下这两个函数的实现

```
1. /*
2. {
3.     SIGINT: 2,
4.     ...
5. }
6. */
7. const { signals } = internalBinding('constants').os;
8.
9. let Signal;
10. const signalWraps = new Map();
11.
12. function isSignal(event) {
13.     return typeof event === 'string' && signals[event] !== undefined;
14. }
15.
16. function startListeningIfSignal(type) {
17.     if (isSignal(type) && !signalWraps.has(type)) {
```

```

18. if (Signal === undefined)
19.     Signal = internalBinding('signal_wrap').Signal;
20. const wrap = new Signal();
21. // 不影响事件循环的退出
22. wrap.unref();
23. // 挂载信号处理函数
24. wrap.onsignal = process.emit.bind(process, type, type);
25. // 通过字符拿到数字
26. const signum = signals[type];
27. // 注册信号
28. const err = wrap.start(signum);
29. if (err) {
30.     wrap.close();
31.     throw errnoException(err, 'uv_signal_start');
32. }
33. // 该信号已经注册，不需要往底层再注册了
34. signalWraps.set(type, wrap);
35. }
36. }

```

startListeningIfSignal 函数的逻辑分为以下几个

- 1 判断该信号是否注册过了，如果注册过了则不再注册。Libuv 本身支持在同一个信号上注册多个处理函数，Node.js 的 JS 层也做了这个处理。
 - 2 调用 unref，信号的注册不应该影响事件循环的退出
 - 3 挂载事件处理函数，当信号触发的时候，执行对应的处理函数（一个或多个）。
 - 4 往底层注册信号并设置该信号已经注册的标记
- 我们再来看一下 stopListeningIfSignal。

```

1. function stopListeningIfSignal(type) {
2.     const wrap = signalWraps.get(type);
3.     if (wrap !== undefined && process.listenerCount(type) === 0) {
4.         wrap.close();
5.         signalWraps.delete(type);
6.     }
7. }

```

只有当信号被注册过并且事件处理函数个数为 0，才做真正的删除。

第八章 DNS

Node.js 的 DNS 模块使用了 cares 库和 Libuv 的线程池实现。cares 是一个异步 DNS 解析库，它自己实现了 DNS 协议的封包和解析，配合 Libuv 事件驱动机制，在 Node.js 中实现

异步的 DNS 解析。另外通过 IP 查询域名或者域名查询 IP 是直接调用操作系统提供的接口实现的，因为这两个函数是阻塞式的 API，所以 Node.js 是通过 Libuv 的线程池实现异步查询。除了提供直接的 DNS 查询外，Node.js 还提供了设置 DNS 服务器、新建一个 DNS 解析实例（Resolver）等功能。这些功能是使用 cares 实现的。下面我们开始分析 DNS 模块的原理和实现。

8.1 通过域名找 IP

我们看一下在 Node.js 中如何查询一个域名对应的 IP 的信息

```
1. dns.lookup('www.a.com', function(err, address, family) {  
2.     console.log(address);  
3. });
```

DNS 功能的 JS 层实现在 dns.js 中

```
1. const req = new GetAddrInfoReqWrap();  
2. req.callback = callback;  
3. req.family = family;  
4. req.hostname = hostname;  
5. req.oncomplete = all ? onlookupall : onlookup;  
6.  
7. const err = cares.getaddrinfo(  
8.     req, toASCII(hostname), family, hints, verbatim  
9. );
```

Node.js 设置了一些参数后，调用 cares_wrap.cc 的 getaddrinfo 方法，在 care_wrap.cc 的初始化函数中我们看到，getaddrinfo 函数对应的函数是 GetAddrInfo。

```
1. void Initialize(Local<Object> target,  
2.                  Local<Value> unused,  
3.                  Local<Context> context) {  
4.     Environment* env = Environment::GetCurrent(context);  
5.     env->SetMethod(target, "getaddrinfo", GetAddrInfo);  
6.     ...  
7. }
```

GetAddrInfo 的主要逻辑如下

```
1. auto req_wrap = new GetAddrInfoReqWrap(env, req_wrap_obj, args[4]->IsTrue());  
2.
```

```

3. struct addrinfo hints;
4. memset(&hints, 0, sizeof(struct addrinfo));
5. hints.ai_family = family;
6. hints.ai_socktype = SOCK_STREAM;
7. hints.ai_flags = flags;
8.
9. int err = uv_getaddrinfo(env->event_loop(),
10.                           req_wrap->req(),
11.                           AfterGetAddrInfo,
12.                           *hostname,
13.                           nullptr,
14.                           &hints);

```

GetAddrInfo 是对 uv_getaddrinfo 的封装，回调函数是 AfterGetAddrInfo

```

1. int uv_getaddrinfo(uv_loop_t* loop,
2.                     // 上层传进来的 req
3.                     uv_getaddrinfo_t* req,
4.                     // 解析完后的上层回调
5.                     uv_getaddrinfo_cb cb,
6.                     // 需要解析的名字
7.                     const char* hostname,
8.                     /*
9.                      查询的过滤条件：服务名。比如
10.                     http smtp。也可以是一个端口。
11.                     见下面注释
12.                     */
13.                     const char* service,
14.                     // 其它查询过滤条件
15.                     const struct addrinfo* hints)
{
16.
17.     size_t hostname_len;
18.     size_t service_len;
19.     size_t hints_len;
20.     size_t len;
21.     char* buf;
22.
23.     hostname_len = hostname ? strlen(hostname) + 1 : 0;
24.     service_len = service ? strlen(service) + 1 : 0;
25.     hints_len = hints ? sizeof(*hints) : 0;
26.     buf = uv_malloc(hostname_len + service_len + hints_len);

```

```
27.     uv_req_init(loop, req, UV_GETADDRINFO);
28.     req->loop = loop;
29.     // 设置请求的回调
30.     req->cb = cb;
31.     req->addrinfo = NULL;
32.     req->hints = NULL;
33.     req->service = NULL;
34.     req->hostname = NULL;
35.     req->retcode = 0;
36.     len = 0;
37.
38.     if (hints) {
39.         req->hints = memcpy(buf + len, hints, sizeof(*hints));
40.         len += sizeof(*hints);
41.     }
42.
43.     if (service) {
44.         req->service = memcpy(buf + len, service, service_len);
45.         len += service_len;
46.     }
47.
48.     if (hostname)
49.         req->hostname = memcpy(buf + len, hostname, hostname_len);
50.     // 传了 cb 则是异步
51.     if (cb) {
52.         uv_work_submit(loop,
53.                         &req->work_req,
54.                         UV_WORK_SLOW_IO,
55.                         uv_getaddrinfo_work,
56.                         uv_getaddrinfo_done);
57.         return 0;
58.     } else {
59.         // 阻塞式查询, 然后执行回调
60.         uv_getaddrinfo_work(&req->work_req);
61.         uv_getaddrinfo_done(&req->work_req, 0);
62.         return req->retcode;
63.     }
64. }
```

我们看到这个函数首先是对一个 request 进行初始化，然后根据是否传了回调，决定走异步还是同步的模式。同步的方式比较简单，就是直接阻塞 Libuv 事件循环，直到解析完

成。如果是异步，则给线程池提交一个慢 I/O 的任务。其中工作函数是 `uv_getaddrinfo_work`。回调是 `uv_getaddrinfo_done`。我们看一下这两个函数。

```

1. // 解析的工作函数
2. static void uv_getaddrinfo_work(struct uv_work* w) {
3.     uv_getaddrinfo_t* req;
4.     int err;
5.     // 根据结构体的字段获取结构体首地址
6.     req = container_of(w, uv_getaddrinfo_t, work_req);
7.     // 阻塞在这
8.     err = getaddrinfo(req->hostname,
9.                        req->service,
10.                       req->hints,
11.                       &req->addrinfo);
12.    req->retcode = uv_getaddrinfo_translate_error(err);
13. }
```

`uv_getaddrinfo_work` 函数主要是调用了系统提供的 `getaddrinfo` 去做解析。该函数会导致进程阻塞。结果返回后，执行 `uv_getaddrinfo_done`。

```

1. static void uv_getaddrinfo_done(struct uv_work* w, int status)
{
2.     uv_getaddrinfo_t* req;
3.
4.     req = container_of(w, uv_getaddrinfo_t, work_req);
5.     uv_req_unregister(req->loop, req);
6.     // 释放初始化时申请的内存
7.     if (req->hints)
8.         uv_free(req->hints);
9.     else if (req->service)
10.        uv_free(req->service);
11.     else if (req->hostname)
12.        uv_free(req->hostname);
13.     else
14.         assert(0);
15.
16.     req->hints = NULL;
17.     req->service = NULL;
18.     req->hostname = NULL;
19.     // 解析请求被用户取消了
20.     if (status == UV_ECANCELED) {
21.         assert(req->retcode == 0);
```

```
22.         req->retcode = UV_EAI_CANCELED;
23.     }
24. // 执行上层回调
25. if (req->cb)
26.     req->cb(req, req->retcode, req->addrinfo);
27.
28. }
```

uv__getaddrinfo_done 会执行 C++ 层的回调，从而执行 JS 层的回调。

8.2 cares

除了通过 IP 查询域名和域名查询 IP 外，其余的 DNS 功能都由 cares 实现，我们看一下 cares 的基本用法。

8.2.1 cares 使用和原理

```
1. // channel 是 cares 的核心结构体
2. ares_channel channel;
3. struct ares_options options;
4. // 初始化 channel
5. status = ares_init_options(&channel, &options, optmask);
6. // 把 argv 的数据存到 addr
7. ares_inet_pton(AF_INET, *argv, &addr4);
8. // 把 addr 数据存到 channel 并发起 DNS 查询
9. ares_gethostbyaddr(channel,
10.                     &addr4,
11.                     sizeof(addr4),
12.                     AF_INET,
13.                     callback, *argv);
14. for (;;)
15. {
16.     int res;
17.     FD_ZERO(&read_fds);
18.     FD_ZERO(&write_fds);
19.     // 把 channel 对应的 fd 存到 read_fd 和 write_fds
20.     nfds = ares_fds(channel, &read_fds, &write_fds);
21.     if (nfds == 0)
22.         break;
23.     // 设置超时时间
24.     tvp = ares_timeout(channel, NULL, &tv);
25.     // 阻塞在 select，等待 DNS 回包
```

```
26.         res = select(nfds, &read_fds, &write_fds, NULL, tvp);  
27.         if (-1 == res)  
28.             break;  
29.         // 处理 DNS 相应  
30.         ares_process(channel, &read_fds, &write_fds);  
31.     }
```

上面是一个典型的事情驱动模型，首先初始化一些信息，然后发起一个非阻塞的请求，接着阻塞在多路复用 API，该 API 返回后，执行触发了事件的回调。

8.2.2 cares_wrap.cc 的通用逻辑

在 Node.js 中，Node.js 和 cares 的整体交互如图 8-1 所示。

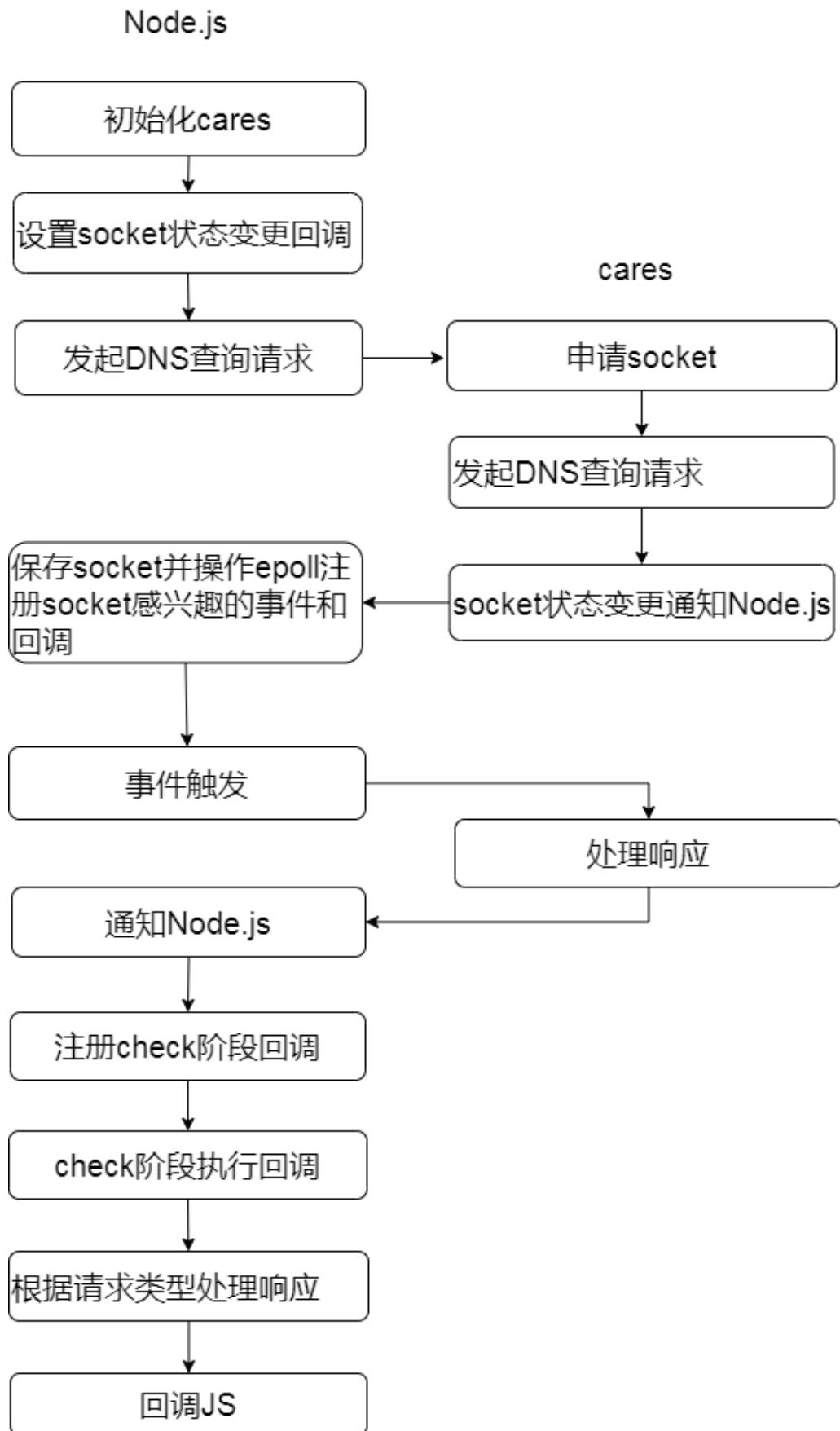


图 8-1

我们通过 cares_wrap.cc 分析其中的原理。我们从 DNS 模块提供的 resolveCname 函数开始。resolveCname 函数由以下代码导出 (dns.js)。

`bindDefaultResolver(module.exports, getDefaultResolver())`
我们看一下这两个函数 (dns/utils.js)。

```

1. class Resolver {
2.   constructor() {
3.     this._handle = new ChannelWrap();
4.   }
5.   // ...
6. }
7.
8. let defaultResolver = new Resolver();
9.
10. function getDefaultResolver() {
11.   return defaultResolver;
12. }
13.
14. function resolver(bindingName) {
15.   function query(name, /* options, */ callback) {
16.     let options;
17.     const req = new QueryReqWrap();
18.     req.bindingName = bindingName;
19.     req.callback = callback;
20.     req.hostname = name;
21.     req.oncomplete = onresolve;
22.     req.ttl = !(options && options.ttl);
23.     const err = this._handle[bindingName](req, toASCII(name));
24.     if (err) throw dnsException(err, bindingName, name);
25.     return req;
26.   }
27.   Object.defineProperty(query, 'name', { value: bindingName });
28.   return query;
29. }
30. // 给原型链注入一个新的属性, defaultResolver 中也生效
31. Resolver.prototype.resolveCname = resolveMap.CNAME = resolver('queryCname');

```

getDefaultResolver 导出的是一个 Resolver 对象，里面有 resolveCname 等一系列方法。接着看一下 bindDefaultResolver，我们一会再看 ChannelWrap。

```

1. const resolverKeys = [
2.   'resolveCname',

```

```
3. // ...
4. ]
5. function bindDefaultResolver(target, source) {
6.   resolverKeys.forEach((key) => {
7.     target[key] = source[key].bind(defaultResolver);
8.   });
9. }
```

看起来很绕，其实就把 Resolve 对象的方法导出到 DNS 模块。这样用户就可以使用了。我们看到 resolveCname 是由 resolver 函数生成的，resolver 函数对 cares 系列函数进行了封装，最终调用的是 this._handle.queryCname 函数。我们来看一下这个 handle（ChannelWrap 类对象）的实现（cares_wrap.cc）。我们先看一下 cares_wrap.cc 模块导出的 API。

```
1. Local<FunctionTemplate> channel_wrap = env->NewFunctionTemplate(C
   hanneWrap::New);
2. channel_wrap->InstanceTemplate()->SetInternalFieldCount(1);
3. channel_wrap->Inherit(AsyncWrap::GetConstructorTemplate(env));
4. // Query 是 C++ 函数模板
5. env->SetProtoMethod(channel_wrap,
6.                       "queryCname",
7.                       Query<QueryCnameWrap>);
8. // ...
9. Local<String> channelWrapString = FIXED_ONE_BYTE_STRING(env->isol
   ate(), "ChannelWrap");
10. channel_wrap->SetClassName(channelWrapString);
11. target->Set(env->context(),
12. channelWrapString, channel_wrap->GetFunction(context).ToLocalChec
   ked()).Check();
```

handle 对应的就是以上代码导出的对象。当我们在 JS 层执行 new ChannelWrap 的时候。最终会调用 C++ 层创建一个对象，并且执行 ChannelWrap::New。

```
1. void ChannelWrap::New(const FunctionCallbackInfo<Value>& args) {
2.   Environment* env = Environment::GetCurrent(args);
3.   new ChannelWrap(env, args.This());
4. }
```

我们看一下类 ChannelWrap 的定义。

```
1. class ChannelWrap : public AsyncWrap {
2. public:
3.   // ...
4.
5. private:
6.   // 超时管理
7.   uv_timer_t* timer_handle_;
8.   // cares 数据类型
```

```

9.     ares_channel channel_;
10.    // 标记查询结果
11.    bool query_last_ok_;
12.    // 使用的 DNS 服务器
13.    bool is_servers_default_;
14.    // 是否已经初始化 cares 库
15.    bool library_inited_;
16.    // 正在发起的查询个数
17.    int active_query_count_;
18.    // 发起查询的任务队列
19.    node_ares_task_list task_list_;
20. };

```

接着我们看看 ChannelWrap 构造函数的代码。

```

1. ChannelWrap::ChannelWrap(...) {
2.     Setup();
3. }

```

ChannelWrap 里直接调用了 Setup

```

1. void ChannelWrap::Setup() {
2.     struct ares_options options;
3.     memset(&options, 0, sizeof(options));
4.     options.flags = ARES_FLAG_NOCHECKRESP;
5.     /*
6.         caresd socket 状态（读写）发生变更时，执行的函数，
7.         第一个入参是 sock_state_cb_data
8.     */
9.     options.sock_state_cb = ares_sockstate_cb;
10.    options.sock_state_cb_data = this;
11.
12.    // 还没初始化则初始化
13.    if (!library_inited_) {
14.        Mutex::ScopedLock lock(ares_library_mutex);
15.        // 初始化 cares 库
16.        ares_library_init(ARES_LIB_INIT_ALL);
17.    }
18.    // 设置使用 cares 的配置
19.    ares_init_options(&channel_,
20.                      &options,
21.                      ARES_OPT_FLAGS | ARES_OPT_SOCK_STATE_CB);
22.    library_inited_ = true;
23. }

```

我们看到，Node.js 在这里初始化 cares 相关的逻辑。其中最重要的就是设置了 cares socket 状态变更时执行的回调 `ares_sockstate_cb`（比如 `socket` 需要读取数据或者写入数据）。前面的 `cares` 使用例子中讲到了 `cares` 和事件驱动模块的配合使用，那么

`cares` 和 `Libuv` 是如何配合的呢? `cares` 提供了一种机制, 就是 `socket` 状态变更时通知事件驱动模块。DNS 解析本质上也是网络 IO, 所以发起一个 DNS 查询也就是对应一个 `socket`。DNS 查询是由 `cares` 发起的, 这就意味着 `socket` 是在 `cares` 中维护的, 那 `Libuv` 怎么知道呢? 正是 `cares` 提供的通知机制, 使得 `Libuv` 知道发起 DNS 查询对应的 `socket`, 从而注册到 `Libuv` 中, 等到事件触发后, 再通知 `cares`。下面我们看一下具体的实现。我们从发起一个 `cname` 查询开始分析。首先回顾一下 `cares_wrap` 模块导出的 `cname` 查询函数,

`env->SetProtoMethod(channel_wrap, "queryCname", Query<QueryCnameWrap>);`
Query 是 C++ 模板函数, `QueryCnameWrap` 是 C++ 类

```
1. template <class Wrap>
2. static void Query(const FunctionCallbackInfo<Value>& args) {
3.     Environment* env = Environment::GetCurrent(args);
4.     ChannelWrap* channel;
5.     // Holder 中保存了 ChannelWrap 对象, 解包出来
6.     ASSIGN_OR_RETURN_UNWRAP(&channel, args.Holder());
7.     Local<Object> req_wrap_obj = args[0].As<Object>();
8.     Local<String> string = args[1].As<String>();
9.     /*
10.      根据参数新建一个对象, 这里是 QueryCnameWrap,
11.      并且保存对应的 ChannelWrap 对象和操作相关的对象
12.    */
13.     Wrap* wrap = new Wrap(channel, req_wrap_obj);
14.
15.     node::Utf8Value name(env->isolate(), string);
16.     // 发起请求数加一
17.     channel->ModifyActivityQueryCount(1);
18.     // 调用 Send 函数发起查询
19.     int err = wrap->Send(*name);
20.     if (err) {
21.         channel->ModifyActivityQueryCount(-1);
22.         delete wrap;
23.     }
24.
25.     args.GetReturnValue().Set(err);
26. }
```

Query 只实现了一些通用的逻辑, 然后调用 `Send` 函数, 具体的 `Send` 函数逻辑由各个具体的类实现。

8.2.3 具体实现

我们看一下 QueryCnameWrap 类。

```

1. class QueryCnameWrap: public QueryWrap {
2. public:
3.     QueryCnameWrap(ChannelWrap* channel,
4.                      Local<Object> req_wrap_obj)
5.         : QueryWrap(channel, req_wrap_obj, "resolveCname") {
6.     }
7.
8.     int Send(const char* name) override {
9.         AresQuery(name, ns_c_in, ns_t cname);
10.        return 0;
11.    }
12.
13. protected:
14.     void Parse(unsigned char* buf, int len) override {
15.         HandleScope handle_scope(env()->isolate());
16.         Context::Scope context_scope(env()->context());
17.
18.         Local<Array> ret = Array::New(env()->isolate());
19.         int type = ns_t cname;
20.         int status = ParseGeneralReply(env(), buf, len, &type, ret);
21.
22.         if (status != ARES_SUCCESS) {
23.             ParseError(status);
24.             return;
25.         }
26.         this->CallOnComplete(ret);
27.     }
28. };

```

我们看到 QueryCnameWrap 类的实现非常简单，主要定义 Send 和 Parse 的实现，最终还是会调用基类对应的函数。我们看一下基类 QueryWrap 中 AresQuery 的实现。

```

1. void AresQuery(const char* name,
2.                  int dnsclass,
3.                  int type) {
4.     ares_query(channel_>cares_channel(),
5.                name,
6.                dnsclass,
7.                type,
8.                Callback,

```

```
9.                     static_cast<void*>(this));
10.    }
```

AresQuery 函数提供统一发送查询操作。查询完成后执行 Callback 回调。接下来就涉及到 cares 和 Node.js 的具体交互了。Node.js 把一个任务交给 cares 后，cares 会新建一个 socket，接着 cares 会通过 Node.js 设置的回调 ares_sockstate_cb 通知 Node.js。我们看一下 ares_query 的关键逻辑。

```
1. void ares_query(ares_channel channel, const char *name, int dnsclass,
2.                   int type, ares_callback callback, void *arg)
3. {
4.     struct qquery *qquery;
5.     unsigned char *qbuf;
6.     int qlen, rd, status;
7.
8.     qquery = ares_malloc(sizeof(struct qquery));
9.     // 保存 Node.js 的回调，查询完成时回调
10.    qquery->callback = callback;
11.    qquery->arg = arg;
12.    ares_send(channel, qbuf, qlen, qcallback, qquery);
13. }
14.
15. static void qcallback(void *arg, int status, int timeouts, unsigned char *abuf, int alen)
16. {
17.     struct qquery *qquery = (struct qquery *) arg;
18.     unsigned int ancount;
19.     int rcode;
20.
21.     if (status != ARES_SUCCESS)
22.         qquery->callback(qquery->arg, status, timeouts, abuf, alen);
23.     else
24.     {
25.         // ...
26.         // 执行 Node.js 回调
27.         qquery->callback(qquery->arg,
28.                           status,
29.                           timeouts,
30.                           abuf,
31.                           alen);
32.     }
33.     ares_free(qquery);
34. }
35.
```

`ares_query` 保存了 `Node.js` 的回调，并且设置回调 `qcallback`，查询成功后会回调 `qcallback`，`qcallback` 再回调 `Node.js`。接着执行 `ares_send`，`ares_send` 会调用 `ares_send_query`。

```

1. void ares_send_query(ares_channel channel,
2.                      struct query *query,
3.                      struct timeval *now)
4. {
5.     struct server_state *server = &channel->servers[query->server
6. ];
7.     if (server->udp_socket == ARES_SOCKET_BAD)
8.     {
9.         // 申请一个 socket
10.        if (open_udp_socket(channel, server) == -1)
11.        {
12.            skip_server(channel, query, query->server);
13.            next_server(channel, query, now);
14.            return;
15.        }
16.        // 发送 DNS 查询
17.        if (socket_write(channel, server->udp_socket, query->qbuf,
18. query->qlen) == -1)
19.        {
20.            skip_server(channel, query, query->server);
21.            next_server(channel, query, now);
22.            return;
23.    }

```

`ares_send_query` 首先申请一个 `socket`，然后发送数据。因为 `UDP` 不是面向连接的，可以直接发送。我们看一下 `open_udp_socket`。

```

1. static int open_udp_socket(ares_channel channel, struct server_st
   ate *server)
2. {
3.     ares_socket_t s;
4.     ares_socklen_t salen;
5.     union {
6.         struct sockaddr_in sa4;
7.         struct sockaddr_in6 sa6;
8.     } saddr;
9.     struct sockaddr *sa;
10.
11.    // 申请一个 socket

```

```
12.     s = open_socket(channel, server->addr.family, SOCK_DGRAM, 0);
13.     // 绑定服务器地址
14.     connect_socket(channel, s, sa, salen)
15.
16.     // 通知 Node.js, 1,0 表示对 socket 的读事件感兴趣, 因为发送了请求, 等待响应
17.     SOCK_STATE_CALLBACK(channel, s, 1, 0);
18.     // 保存 socket
19.     server->udp_socket = s;
20.     return 0;
21. }
22.
23. #define SOCK_STATE_CALLBACK(c, s, r, w)
    \
24.     do {
        \
25.         if ((c)->sock_state_cb)
            \
26.             (c)->sock_state_cb((c)->sock_state_cb_data, (s), (r), (w))
        ;
        \
27.     } WHILE_FALSE
28.
```

ares_send_query 函数做了三件事

1 申请了 socket,

2 通知 Node.js

3 发送了 DNS 查询请求

这时候流程走到了 Node.js, 我们看一下 cares 回调 Node.js 的时候, Node.js 怎么处理的

```
1. struct node_ares_task : public MemoryRetainer {
2.     ChannelWrap* channel;
3.     // 关联的 socket
4.     ares_socket_t sock;
5.     // IO 观察者和回调
6.     uv_poll_t poll_watcher;
7. };
8.
9. void ares_sockstate_cb(void* data,
10.                         ares_socket_t sock,
11.                         int read,
12.                         int write) {
13.     ChannelWrap* channel = static_cast<ChannelWrap*>(data);
14.     node_ares_task* task;
15.     // 任务
```

```

16. node_ares_task lookup_task;
17. lookup_task.sock = sock;
18. // 该任务是否已经存在
19. auto it = channel->task_list()->find(&lookup_task);
20.
21. task = (it == channel->task_list()->end()) ? nullptr : *it;
22.
23. if (read || write) {
24.     if (!task) {
25.         // 开启定时器，超时后通知 cares
26.         channel->StartTimer();
27.         // 创建一个任务
28.         task = ares_task_create(channel, sock);
29.         // 保存到任务列表
30.         channel->task_list()->insert(task);
31.     }
32.     // 注册 IO 观察者到 epoll，感兴趣的事件根据 cares 传的进行设置，有事
件触发后执行回调 ares_poll_cb
33.     uv_poll_start(&task->poll_watcher,
34.                     (read ? UV_READABLE : 0) | (write ? UV_WRITABLE
35.                      : 0),
36.                     ares_poll_cb);
37. } else {
38.     // socket 关闭了，删除任务
39.     channel->task_list()->erase(it);
40.     // 关闭该任务对应观察者 io，然后删除删除该任务
41.     channel->env()->CloseHandle(&task->poll_watcher, ares_poll_c
lose_cb);
42.     // 没有任务了，关闭定时器
43.     if (channel->task_list()->empty()) {
44.         channel->CloseTimer();
45.     }
46. }
47. }

```

每一个 DNS 查询的任务，在 Node.js 中用 `node_ares_task` 管理。它封装了请求对应的 `channel`、查询请求对应的 `socket` 和 `uv_poll_t`。我们看一下 `ares_task_create`

```

1. node_ares_task* ares_task_create(ChannelWrap* channel, ares_sooke
t_t sock) {
2.     auto task = new node_ares_task();
3.
4.     task->channel = channel;
5.     task->sock = sock;

```

```
6. // 初始化 uv_poll_t, 保存文件描述符 sock 到 uv_poll_t
7. if (uv_poll_init_socket(channel->env()->event_loop(), &task->poll_watcher, sock) < 0) {
8.     delete task;
9.     return nullptr;
10. }
11.
12. return task;
13. }
```

首先创建一个 node_ares_task 对象。然后初始化 uv_poll_t 并且把文件描述符保存到 uv_poll_t。uv_poll_t 是对文件描述符、回调、IO 观察者的封装。文件描述符的事件触发时，会执行 IO 观察者的回调，从而执行 uv_poll_t 保存的回调。我们继续回到 ares_sockstate_cb，当 cares 通知 Node.js socket 状态变更的时候，Node.js 就会修改 epoll 节点的配置（感兴趣的事件）。当事件触发的时候，会执行 ares_poll_cb。我们看一下该函数。

```
1. void ares_poll_cb(uv_poll_t* watcher, int status, int events) {
2.     node_ares_task* task = ContainerOf(&node_ares_task::poll_watcher, watcher);
3.     ChannelWrap* channel = task->channel;
4.
5.     // 有事件触发，重置超时时间
6.     uv_timer_again(channel->timer_handle());
7.
8.     // 通知 cares 处理响应
9.     ares_process_fd(channel->cares_channel(),
10.                      events & UV_READABLE ? task->sock : ARES_SOCKET_BAD,
11.                      events & UV_WRITABLE ? task->sock : ARES_SOCKET_BAD);
12. }
```

当 socket 上感兴趣的事件触发时，Node.js 调 ares_process_fd 处理。真正的处理函数是 processfds。

```
1. static void processfds(ares_channel channel,
2.                         fd_set *read_fds, ares_socket_t read_fd,
3.                         fd_set *write_fds, ares_socket_t write_fd)
4. {
5.     struct timeval now = ares_tvnow();
6.
7.     write_tcp_data(channel, write_fds, write_fd, &now);
8.     read_tcp_data(channel, read_fds, read_fd, &now);
```

```

9.   read_udp_packets(channel, read_fds, read_fd, &now);
10.  process_timeouts(channel, &now);
11.  process_broken_connections(channel, &now);
12. }
```

`processfds` 是统一的处理函数，在各自函数内会做相应的判断和处理。我们这里是收到了 UDP 响应。则会执行 `read_udp_packets`

```

1. static void read_udp_packets(ares_channel channel, fd_set *read_fds,
2.                               ares_socket_t read_fd, struct timeval
3.                               l *now){
4. // 读取响应
5. count = socket_recvfrom(channel, server->udp_socket, (void *)buf,
6.                         sizeof(buf), 0, &from.sa, &fromlen);
5. // 处理响应，最终调用 query->callback 回调 Node.js
6. process_answer(channel, buf, (int)count, i, 0, now);
7. }
```

`Cares` 读取响应然后解析响应，最后回调 `Node.js`。`Node.js` 设置的回调函数是 `Callback`

```

1. static void Callback(void* arg, int status, int timeouts,
2.                      unsigned char* answer_buf, int answer_len)
3. {
4.     QueryWrap* wrap = FromCallbackPointer(arg);
5.     unsigned char* buf_copy = nullptr;
6.     if (status == ARES_SUCCESS) {
7.         buf_copy = node::Malloc<unsigned char>(answer_len);
8.         memcpy(buf_copy, answer_buf, answer_len);
9.     }
10.    wrap->response_data_ = std::make_unique<responseData>();
11.    responseData* data = wrap->response_data_.get();
12.    data->status = status;
13.    data->is_host = false;
14.    data->buf = MallocedBuffer<unsigned char>(buf_copy, answer_len);
15.    // 执行 QueueResponseCallback
16.    wrap->QueueResponseCallback(status);
17. }
18.
19. void QueueResponseCallback(int status) {
20.     BaseObjectPtr<QueryWrap> strong_ref{this};
21.     // 产生一个 native immediate 任务，在 check 阶段执行
```

```
22.     env()->SetImmediate([this, strong_ref](Environment*) {  
23.         // check 阶段执行  
24.         AfterResponse();  
25.         // Delete once strong_ref goes out of scope.  
26.         Detach();  
27.     });  
28.  
29.     channel_->set_query_last_ok(status != ARES_ECONNREFUSED);  
30.     channel_->ModifyActivityQueryCount(-1);  
31. }  
32.  
33. void AfterResponse() {  
34.     const int status = response_data_->status;  
35.     // 调用对应的子类的 Parse  
36.     if (status != ARES_SUCCESS) {  
37.         ParseError(status);  
38.     } else if (!response_data_->is_host) {  
39.         Parse(response_data_->buf.data, response_data_->buf.size);  
40.     } else {  
41.         Parse(response_data_->host.get());  
42.     }  
43. }
```

任务完成后，Node.js 会在 check 阶段（Node.js v10 是使用 `async handle` 通知 Libuv）加入一个节点，然后 check 阶段的时候执行对应子类的 `Parse` 函数，这里以 `QueryCnameWrap` 的 `Parse` 为例。

```
1. void Parse(unsigned char* buf, int len) override {  
2.     HandleScope handle_scope(env()->isolate());  
3.     Context::Scope context_scope(env()->context());  
4.  
5.     Local<Array> ret = Array::New(env()->isolate());  
6.     int type = ns_t_cname;  
7.     int status = ParseGeneralReply(env(), buf, len, &type, ret);  
8.  
9.     if (status != ARES_SUCCESS) {  
10.         ParseError(status);  
11.         return;  
12.     }  
13.     this->CallOnComplete(ret);  
14. }
```

收到 DNS 回复后，调用 `ParseGeneralReply` 解析回包，然后执行 JS 层 DNS 模块的回调。从而执行用户的回调。

```

1. void CallOnComplete(Local<Value> answer,
2.                      Local<Value> extra = Local<Va
3.                      lue>()) {
4.     HandleScope handle_scope(env()->isolate());
5.     Context::Scope context_scope(env()->context());
6.     Local<Value> argv[] = {
7.         Integer::New(env()->isolate(), 0),
8.         answer,
9.         extra
10.    };
11.   const int argc = arraysize(argv) - extra.IsEmpty();
12.   MakeCallback(env()->oncomplete_string(), argc, argv);
13. }

```

第九章 Unix 域

Unix 域一种进程间通信的方式，Unix 域不仅支持没有继承关系的进程间进行通信，而且支持进程间传递文件描述符。Unix 域是 Node.js 中核心的功能，它是进程间通信的底层基础，child_process 和 cluster 模块都依赖 Unix 域的能力。从实现和使用上来看，Unix 域类似 TCP，但是因为它是基于同主机进程的，不像 TCP 需要面临复杂的网络的问题，所以实现也没有 TCP 那么复杂。Unix 域和传统的 socket 通信一样，遵循网络编程的那一套流程，由于在同主机内，就不必要使用 IP 和端口的方式。Node.js 中，Unix 域采用的是一个文件作为标记。大致原理如下。

- 1 服务器首先拿到一个 socket。
 - 2 服务器 bind 一个文件，类似 bind 一个 IP 和端口一样，对于操作系统来说，就是新建一个文件（不一定是在硬盘中创建，可以设置抽象路径名），然后把文件路径信息存在 socket 中。
 - 3 调用 listen 修改 socket 状态为监听状态。
 - 4 客户端通过同样的文件路径调用 connect 去连接服务器。这时候用于表示客户端的结构体插入服务器的连接队列，等待处理。
 - 5 服务器调用 accept 摘取队列的节点，然后新建一个通信 socket 和客户端进行通信。
- Unix 域通信本质还是基于内存之间的通信，客户端和服务器都维护一块内存，这块内存分为读缓冲区和写缓冲区。从而实现全双工通信，而 Unix 域的文件路径，只不过是为了让客户端进程可以找到服务端进程，后续就可以互相往对方维护的内存里写数据，从而实现进程间通信。

9.1 Unix 域在 Libuv 中的使用

接下来我们看一下在 Libuv 中关于 Unix 域的实现和使用。

9.1.1 初始化

Unix 域使用 uv_pipe_t 结构体表示，使用之前首先需要初始化 uv_pipe_t。下面看一下它的实现逻辑。

```
1. int uv_pipe_init(uv_loop_t* loop, uv_pipe_t* handle, int ipc) {
2.     uv_stream_init(loop, (uv_stream_t*)handle, UV_NAMED_PIPE);
3.     handle->shutdown_req = NULL;
4.     handle->connect_req = NULL;
5.     handle->pipe_fname = NULL;
6.     handle->ipc = ipc;
7.     return 0;
8. }
```

uv_pipe_init 逻辑很简单，就是初始化 uv_pipe_t 结构体的一些字段。uv_pipe_t 继承于 stream，uv_stream_init 就是初始化 stream（父类）的字段。uv_pipe_t 中有一个字段 ipc，该字段标记了是否允许在该 Unix 域通信中传递文件描述符。

9.1.2 绑定 Unix 域路径

开头说过，Unix 域的实现类似 TCP 的实现。遵循网络 socket 编程那一套流程。服务端使用 bind，listen 等函数启动服务。

```
1. // name 是 unix 路径名称
2. int uv_pipe_bind(uv_pipe_t* handle, const char* name) {
3.     struct sockaddr_un saddr;
4.     const char* pipe_fname;
5.     int sockfd;
6.     int err;
7.     pipe_fname = NULL;
8.     pipe_fname = uv_strdup(name);
9.     name = NULL;
10.    // 流式 Unix 域套接字
11.    sockfd = uv_socket(AF_UNIX, SOCK_STREAM, 0);
12.    memset(&saddr, 0, sizeof(saddr));
13.    strncpy(saddr.sun_path, pipe_fname, sizeof(saddr.sun_path) - 1);
14.    saddr.sun_path[sizeof(saddr.sun_path) - 1] = '\0';
15.    saddr.sun_family = AF_UNIX;
16.    // 绑定到路径，TCP 是绑定到 IP 和端口
17.    if (bind(sockfd, (struct sockaddr*)&saddr, sizeof(saddr))) {
18.        // ...
19.    }
20. }
```

```

21. // 设置绑定成功标记
22. handle->flags |= UV_HANDLE_BOUND;
23. // Unix 域的路径
24. handle->pipe_fname = pipe_fname;
25. // 保存 socket 对应的 fd
26. handle->io_watcher.fd = sockfd;
27. return 0;
28. }

```

uv_pipe_bind 函数首先申请一个 socket，然后调用操作系统的 bind 函数把 Unix 域路径保存到 socket 中。最后标记已经绑定标记，并且保存 Unix 域的路径和 socket 对应的 fd 到 handle 中，后续需要使用。我们看到 Node.js 中 Unix 域的类型是 SOCK_STREAM。Unix 域支持两种数据模式。

1 流式（SOCK_STREAM），类似 TCP，数据为字节流，需要应用层处理粘包问题。

2 数据报模式（SOCK_DGRAM），类似 UDP，不需要处理粘包问题。

通过 Unix 域虽然可以实现进程间的通信，但是我们拿到的数据可能是“乱的”，这是为什么呢？一般情况下，客户端给服务器发送 1 个字节，然后服务器处理，如果是基于这种场景，那么数据就不会是乱的。因为每次就是一个需要处理的数据单位。但是如果客户端给服务器发送 1 个字节，服务器还没来得及处理，客户端又发送了一个字节，那么这时候服务器再处理的时候，就会有问题。因为两个字节混一起了。就好比在一个 TCP 连接上先后发送两个 HTTP 请求一样，如果服务器没有办法判断两个请求的数据边界，那么处理就会有问题。所以这时候，我们需要定义一个应用层协议，并且实现封包解包的逻辑，才能真正完成进程间通信。

9.1.3 启动服务

绑定了路径后，就可以调用 listen 函数使得 socket 处于监听状态。

```

1. int uv_pipe_listen(uv_pipe_t* handle, int backlog, uv_connection_cb
    cb) {
2.     // uv_stream_fd(handle) 得到 bind 函数中获取的 socket
3.     if (listen(uv_stream_fd(handle), backlog))
4.         return UV_ERR(errno);
5.     // 保存回调，有进程调用 connect 的时候时触发，由 uv_server_io 函数触
    发
6.     handle->connection_cb = cb;
7.     // IO 观察者的回调
8.     handle->io_watcher.cb = uv_server_io;
9.     // 注册 IO 观察者到 Libuv，等待连接，即读事件到来
10.    uv_io_start(handle->loop, &handle->io_watcher, POLLIN);
11.    return 0;
12. }

```

uv_pipe_listen 执行操作系统的 listen 函数使得 socket 成为监听型的套接字。然后把 socket 对应的文件描述符和回调封装成 IO 观察者。注册到 Libuv 中。等到有读事件到来（有连接到来）。就会执行 uv_server_io 函数，摘下对应的客户端节点。最后执行 connection_cb 回调。

9.1.4 发起连接

这时候，我们已经成功启动了一个 Unix 域服务。接下来就是看客户端的逻辑。

```
1. void uv_pipe_connect(uv_connect_t* req,
2.                      uv_pipe_t* handle,
3.                      const char* name,
4.                      uv_connect_cb cb) {
5.     struct sockaddr_un saddr;
6.     int new_sock;
7.     int err;
8.     int r;
9.     // 判断是否已经有 socket 了，没有的话需要申请一个，见下面
10.    new_sock = (uv_stream_fd(handle) == -1);
11.    // 客户端还没有对应的 socket fd
12.    if (new_sock) {
13.        handle->io_watcher.fd= uv_socket(AF_UNIX,
14.                                         SOCK_STREAM,
15.                                         0);
16.    }
17.    // 需要连接的服务器信息。主要是 Unix 域路径信息
18.    memset(&saddr, 0, sizeof saddr);
19.    strncpy(saddr.sun_path, name, sizeof(saddr.sun_path) - 1);
20.    saddr.sun_path[sizeof(saddr.sun_path) - 1] = '\0';
21.    saddr.sun_family = AF_UNIX;
22.    // 非阻塞式连接服务器，Unix 域路径是 name
23.    do {
24.        r = connect(uv_stream_fd(handle),
25.                    (struct sockaddr*)&saddr, sizeof saddr);
26.    }
27.    while (r == -1 && errno == EINTR);
28.    // 忽略错误处理逻辑
29.    err = 0;
30.    // 设置 socket 的可读写属性
31.    if (new_sock) {
32.        err = uv_stream_open((uv_stream_t*)handle,
33.                             uv_stream_fd(handle),
```

```

34.                                     UV_HANDLE_READABLE | UV_HANDLE_WRITAB
35.     LE);
36. // 把 IO 观察者注册到 Libuv, 等到连接成功或者可以发送请求
37. if (err == 0)
38.     uv_io_start(handle->loop,
39.                  &handle->io_watcher,
40.                  POLLIN | POLLOUT);
41.
42.out:
43. // 记录错误码, 如果有的话
44. handle->delayed_error = err;
45. // 保存调用者信息
46. handle->connect_req = req;
47. uv_req_init(handle->loop, req, UV_CONNECT);
48. req->handle = (uv_stream_t*)handle;
49. req->cb = cb;
50. QUEUE_INIT(&req->queue);
51. /*
52.     如果连接出错, 在 pending 阶段会执行 uv_stream_io,
53.     从而执行 req 对应的回调。错误码是 delayed_error
54. */
55. if (err)
56.     uv_io_feed(handle->loop, &handle->io_watcher);
57. }

```

`uv_pipe_connect` 函数首先以非阻塞的方式调用操作系统的 `connect` 函数，调用 `connect` 后操作系统把客户端对应的 socket 直接插入服务器 socket 的待处理 socket 队列中，等待服务器处理。这时候 socket 是处于连接中的状态，当服务器调用 `accept` 函数处理连接时，会修改连接状态为已连接（这和 TCP 不一样，TCP 是完成三次握手后就会修改为连接状态，而不是 `accept` 的时候），并且会触发客户端 socket 的可写事件。事件驱动模块就会执行相应的回调（`uv_stream_io`），从而执行 C++ 和 JS 的回调。

9.1.5 关闭 Unix 域

我们可以通过 `uv_close` 关闭一个 Unix 域 handle。`uv_close` 中会调用 `uv_pipe_close`。

```

1. void uv_pipe_close(uv_pipe_t* handle) {
2. // 如果是 Unix 域服务器则需要删除 Unix 域路径并删除指向的堆内存
3. if (handle->pipe_fname) {
4.     unlink(handle->pipe_fname);
5.     uv_free((void*)handle->pipe_fname);
6.     handle->pipe_fname = NULL;
7. }

```

```
8. // 关闭流相关的内容
9. uv_stream_close((uv_stream_t*)handle);
10. }
```

关闭 Unix 域 handle 时，Libuv 会自动删除 Unix 域路径对应的文件。但是如果进程异常退出时，该文件可能不会被删除，这样会导致下次监听的时候报错 listen EADDRINUSE，所以安全起见，我们可以在进程退出或者监听之前判断该文件是否存在，存在的话则删除。另外还有一个问题是，如果两个不相关的进程使用了同一个文件则会导致误删，所以 Unix 域对应的文件，我们需要小心处理，最好能保证唯一性。

Unix 域大致的流程和网络编程一样。分为服务端和客户端两面。Libuv 在操作系统提供的 API 的基础上。和 Libuv 的异步非阻塞结合。在 Libuv 中为进程间提供了一种通信方式。下面看一下在 Node.js 中是如何使用 Libuv 提供的功能的。

9.2 Unix 域在 Node.js 中的使用

9.2.1 Unix 域服务器

在 Node.js 中，我们可以通过以下代码创建一个 Unix 域服务器

```
1. const server = net.createServer((client) => {
2.   // 处理 client
3. });
4. server.listen('/tmp/test.sock', () => {
5.   console.log(`bind unix domain success`);
6. });
```

我们从 listen 函数开始分析这个过程。

```
1. Server.prototype.listen = function(...args) {
2.   const normalized = normalizeArgs(args);
3.   let options = normalized[0];
4.   const cb = normalized[1];
5.   // 调用底层的 listen 函数成功后执行的回调
6.   if (cb !== null) {
7.     this.once('listening', cb);
8.   }
9.   if (options.path && isPipeName(options.path)) {
10.    const pipeName = this._pipeName = options.path;
11.    backlog = options.backlog || backlogFromArgs;
12.    listenIncluster(this, pipeName, -1, -1, backlog, undefined,
13.                  options.exclusive);
14.    /*
15.      Unix 域使用文件实现的，客户端需要访问该文件的权限才能通信，
```

```

16.      这里做权限控制
17.      */
18.      let mode = 0;
19.      if (options.readableAll === true)
20.          mode |= PipeConstants.UV_READABLE;
21.      if (options.writableAll === true)
22.          mode |= PipeConstants.UV_WRITABLE;
23.      if (mode !== 0) {
24.          // 修改文件的访问属性
25.          const err = this._handle.fchmod(mode);
26.          if (err) {
27.              this._handle.close();
28.              this._handle = null;
29.              throw errnoException(err, 'uv_pipe_chmod');
30.          }
31.      }
32.      return this;
33.  }
34. }

```

这段代码中最主要的是 `listenIncluster` 函数。我们看一下该函数的逻辑。

```

1. function listenIncluster(server, address, port, addressType,
2.                           backlog, fd, exclusive, flags) {
3.     exclusive = !!exclusive;
4.     if (cluster === undefined) cluster = require('cluster');
5.     if (cluster.isMaster || exclusive) {
6.         server._listen2(address, port, addressType, backlog, fd, flag
s);
7.         return;
8.     }
9. }

```

直接调用 `_listen2` (`isMaster` 只有在 `cluster.fork` 创建的进程中才是 `false`, 其余情况都是 `true`, 包括 `child_process` 模块创建的子进程)。我们继续看 `listen` 函数。

```

1. Server.prototype._listen2 = setupListenHandle;
2.
3. function setupListenHandle(address,
4.                             port,
5.                             addressType,
6.                             backlog,
7.                             fd,
8.                             flags) {
9.     this._handle = createServerHandle(address,
10.                                       port,

```

```
11.                               addressType,
12.                               fd,
13.                               flags);
14. // 有完成连接完成时触发
15. this._handle.onconnection = onconnection;
16. const err = this._handle.listen(backlog || 511);
17. if (err) {
18.   // 触发 error 事件
19. }
20. // 下一个 tick 触发 listen 回调
21. defaultTriggerAsyncIdScope(this[async_id_symbol],
22.                             process.nextTick,
23.                             emitListeningNT,
24.                             this);
25. }
```

首先调用 `createServerHandle` 创建一个 `handle`, 然后执行 `listen` 函数。我们首先看一下 `createServerHandle`。

```
26. function createServerHandle(address,
27.                               port,
28.                               addressType,
29.                               fd,
30.                               flags) {
31.   let handle = new Pipe(PipeConstants.SERVER);
32.   handle.bind(address, port);
33.   return handle;
34. }
```

创建了一个 `Pipe` 对象, 然后调用它的 `bind` 和 `listen` 函数, 我们看 `new Pipe` 的逻辑, 从 `pipe_wrap.cc` 的导出逻辑, 我们知道, 这时候会新建一个 C++ 对象, 然后执行 `New` 函数, 并且把新建的 C++ 对象等信息作为入参。

```
1. void PipeWrap::New(const FunctionCallbackInfo<Value>& args) {
2.   Environment* env = Environment::GetCurrent(args);
3.   // 类型
4.   int type_value = args[0].As<Int32>().Value();
5.   PipeWrap::SocketType type = static_cast<PipeWrap::SocketType>(type_value);
6.   // 是否是用于 IPC
7.   bool ipc;
8.   ProviderType provider;
9.   switch (type) {
10.     case SOCKET:
```

```

11.         provider = PROVIDER_PIPEWRAP;
12.         ipc = false;
13.         break;
14.     case SERVER:
15.         provider = PROVIDER_PIPESERVERWRAP;
16.         ipc = false;
17.         break;
18.     case IPC:
19.         provider = PROVIDER_PIPEWRAP;
20.         ipc = true;
21.         break;
22.     default:
23.         UNREACHABLE();
24.     }
25.
26.     new PipeWrap(env, args.This(), provider, ipc);
27. }
```

New 函数处理了参数，然后执行了 new PipeWrap 创建一个对象。

```

1. PipeWrap::PipeWrap(Environment* env,
2.                         Local<Object> object,
3.                         ProviderType provider,
4.                         bool ipc)
5. : ConnectionWrap(env, object, provider) {
6.     int r = uv_pipe_init(env->event_loop(), &handle_, ipc);
7. }
```

new Pipe 执行完后，就会通过该 C++ 对象调用 Libuv 的 bind 和 listen 完成服务器的启动，就不再展开分析。

9.2.2 Unix 域客户端

接着我们看一下 Unix 域作为客户端使用时的过程。

```

1. Socket.prototype.connect = function(...args) {
2.     const path = options.path;
3.     // Unix 域路径
4.     var pipe = !!path;
5.     if (!this._handle) {
6.         // 创建一个 C++ 层 handle，即 pipe_wrap.cc 导出的 Pipe 类
7.         this._handle = pipe ?
```

```
8.         new Pipe(PipeConstants.SOCKET) :
9.         new TCP(TCPConstants.SOCKET) ;
10.        // 挂载 onread 方法到 this 中
11.        initSocketHandle(this) ;
12.    }
13.
14.    if (cb != null) {
15.        this.once('connect', cb) ;
16.    }
17.    // 执行 internalConnect
18.    defaultTriggerAsyncIdScope(
19.        this[async_id_symbol], internalConnect, this, path
20.    );
21.    return this;
22.};
```

首先新建一个 handle，值是 new Pipe。接着执行了 internalConnect，internalConnect 函数的主要逻辑如下

```
1. const req = new PipeConnectWrap() ;
2. // address 为 Unix 域路径
3. req.address = address;
4. req.oncomplete = afterConnect;
5. // 调用 C++ 层 connect
6. err = self._handle.connect(req, address, afterConnect);
```

我们看 C++ 层的 connect 函数，

```
1. void PipeWrap::Connect(const FunctionCallbackInfo<Value>& args) {
2.     Environment* env = Environment::GetCurrent(args);
3.
4.     PipeWrap* wrap;
5.     ASSIGN_OR_RETURN_UNWRAP(&wrap, args.Holder());
6.     // PipeConnectWrap 对象
7.     Local<Object> req_wrap_obj = args[0].As<Object>();
8.     // Unix 域路径
9.     node::Utf8Value name(env->isolate(), args[1]);
10.    /*
11.        新建一个 ConnectWrap 对象，ConnectWrap 是对 handle 进行一次连接请求
12.        的封装，内部维护一个 uv_connect_t 结构体，req_wrap_obj 的一个字段
13.        指向 ConnectWrap 对象，用于保存对应的请求上下文
14.    }
```

```

14.  */
15. ConnectWrap* req_wrap =
16.     new ConnectWrap(env,
17.                     req_wrap_obj,
18.                     AsyncWrap::PROVIDER_PIPECONNECTWRAP);
19. // 调用 Libuv 的 connect 函数
20. uv_pipe_connect(req_wrap->req(),
21.                   &wrap->handle_,
22.                   *name,
23.                   AfterConnect);
24. // req_wrap->req_.data = req_wrap; 关联起来
25. req_wrap->Dispatched();
26. // uv_pipe_connect() doesn't return errors.
27. args.GetReturnValue().Set(0);
28. }

```

uv_pipe_connect 函数，第一个参数是 uv_connect_t 结构体（request），第二个是一个 uv_pipe_t 结构体（handle），handle 是对 Unix 域客户端的封装，request 是请求的封装，它表示基于 handle 发起一次连接请求。连接成功后会执行 AfterConnect。由前面分析我们知道，当连接成功时，首先会执行回调 Libuv 的 uv_stream_io，然后执行 C++ 层的 AfterConnect。

```

1. // 主动发起连接，成功/失败后的回调
2. template <typename WrapType, typename UVType> = PipeWrap, uv_pipe_t
3. void ConnectionWrap<WrapType, UVType>::AfterConnect(uv_connect_t* req
4. , int status) {
5. // 在 Connect 函数里关联起来的
6. ConnectWrap* req_wrap = static_cast<ConnectWrap*>(req->data);
7. // 在 uv_pipe_connect 中完成关联的
8. WrapType* wrap = static_cast<WrapType*>(req->handle->data);
9. Environment* env = wrap->env();
10.
11. HandleScope handle_scope(env->isolate());
12. Context::Scope context_scope(env->context());
13.
14. bool readable, writable;
15. // 是否连接成功
16. if (status) {
17.     readable = writable = 0;
18. } else {
19.     readable = uv_is_readable(req->handle) != 0;
20.     writable = uv_is_writable(req->handle) != 0;

```

```
21.     }
22.
23.     Local<Value> argv[5] = {
24.         Integer::New(env->isolate(), status),
25.         wrap->object(),
26.         req_wrap->object(),
27.         Boolean::New(env->isolate(), readable),
28.         Boolean::New(env->isolate(), writable)
29.     };
30.     // 执行 JS 层的 oncomplete 回调
31.     req_wrap->MakeCallback(env->oncomplete_string(),
32.                             arraysize(argv),
33.                             argv);
34.
35.     delete req_wrap;
36. }
```

我们再回到 JS 层的 afterConnect

```
1. function afterConnect(status, handle, req, readable, writable) {
2.     var self = handle.owner;
3.     handle = self._handle;
4.     if (status === 0) {
5.         self.readable = readable;
6.         self.writable = writable;
7.         self._unrefTimer();
8.         // 触发 connect 事件
9.         self.emit('connect');
10.        // 可读并且没有处于暂停模式，则注册等待可读事件
11.        if (readable && !self.isPaused())
12.            self.read(0);
13.    }
14. }
```

至此，作为客户端对服务器的连接就完成了。后续就可以进行通信。

第十章 定时器

Node.js V14 对定时器模块进行了重构，之前版本的实现是用一个 map，以超时时间为键，每个键对应一个队列。即有同样超时时间的节点在同一个队列。每个队列对应一个底

层的一个节点（二叉堆里的节点），Node.js 在事件循环的 timer 阶段会从二叉堆里找出超时的节点，然后执行回调，回调里会遍历队列，判断哪个节点超时了。14 重构后，只使用了一个二叉堆的节点。我们看一下它的实现，首先看下定时器模块的整体关系图，如图 10-1 所示。



图 10-1

下面我们先看一下定时器模块的几个重要的数据结构。

10.1 Libuv 的实现

Libuv 中使用二叉堆实现了定时器。最快到期的节点是根节点。

10.1.1 Libuv 中维护定时器的数据结构

```
1. // 取出 loop 中的计时器堆指针
2. static struct heap *timer_heap(const uv_loop_t* loop) {
3.     return (struct heap*) &loop->timer_heap;
4. }
```

10.1.2 比较函数

因为 Libuv 使用二叉堆实现定时器，这就涉及到节点插入堆的时候的规则。

```
5. static int timer_less_than(const struct heap_node* ha,
6.                             const struct heap_node* hb) {
7.     const uv_timer_t* a;
8.     const uv_timer_t* b;
9.     // 通过结构体成员找到结构体首地址
10.    a = container_of(ha, uv_timer_t, heap_node);
11.    b = container_of(hb, uv_timer_t, heap_node);
12.    // 比较两个结构体中的超时时间
13.    if (a->timeout < b->timeout)
14.        return 1;
```

```
15.     if (b->timeout < a->timeout)
16.         return 0;
17. // 超时时间一样的话，看谁先创建
18.     if (a->start_id < b->start_id)
19.         return 1;
20.     if (b->start_id < a->start_id)
21.         return 0;
22.
23.     return 0;
24. }
```

10.1.3 初始化定时器结构体

如果需要使用定时器，首先要对定时器的结构体进行初始化。

```
1. // 初始化 uv_timer_t 结构体
2. int uv_timer_init(uv_loop_t* loop, uv_timer_t* handle) {
3.     uv__handle_init(loop, (uv_handle_t*)handle, UV_TIMER);
4.     handle->timer_cb = NULL;
5.     handle->repeat = 0;
6.     return 0;
7. }
```

10.1.4 插入一个定时器

```
1. // 启动一个计时器
2. int uv_timer_start(uv_timer_t* handle,
3.                     uv_timer_cb cb,
4.                     uint64_t timeout,
5.                     uint64_t repeat) {
6.     uint64_t clamped_timeout;
7.
8.     if (cb == NULL)
9.         return UV_EINVAL;
10.    // 重新执行 start 的时候先把之前的停掉
11.    if (uv__is_active(handle))
12.        uv_timer_stop(handle);
13.    // 超时时间，为绝对值
14.    clamped_timeout = handle->loop->time + timeout;
15.    if (clamped_timeout < timeout)
16.        clamped_timeout = (uint64_t) -1;
17.    // 初始化回调，超时时间，是否重复计时，赋予一个独立无二的 id
```

```

18.     handle->timer_cb = cb;
19.     handle->timeout = clamped_timeout;
20.     handle->repeat = repeat;
21.     // 用于超时时间一样的时候，比较定时器在二叉堆的位置，见 cmp 函数
22.     handle->start_id = handle->loop->timer_counter++;
23.     // 插入最小堆
24.     heap_insert(timer_heap(handle->loop),
25.                   (struct heap_node*) &handle->heap_node,
26.                   timer_less_than);
27.     // 激活该 handle
28.     uv__handle_start(handle);
29.
30.     return 0;
31. }
```

10.1.5 停止一个定时器

```

1. // 停止一个计时器
2. int uv_timer_stop(uv_timer_t* handle) {
3.     if (!uv__is_active(handle))
4.         return 0;
5.     // 从最小堆中移除该计时器节点
6.     heap_remove(timer_heap(handle->loop),
7.                 (struct heap_node*) &handle->heap_node,
8.                 timer_less_than);
9.     // 清除激活状态和 handle 的 active 数减一
10.    uv__handle_stop(handle);
11.
12.    return 0;
13. }
```

10.1.6 重新设置定时器

重新设置定时器类似插入一个定时器，它首先需要把之前的定时器从二叉堆中移除，然后重新插入二叉堆。

```

1. // 重新启动一个计时器，需要设置 repeat 标记
2. int uv_timer_again(uv_timer_t* handle) {
3.     if (handle->timer_cb == NULL)
4.         return UV_EINVAL;
5.     // 如果设置了 repeat 标记说明计时器是需要重复触发的
6.     if (handle->repeat) {
```

```
7.      // 先把旧的节点从最小堆中移除，然后再重新开启一个计时器
8.      uv_timer_stop(handle);
9.      uv_timer_start(handle,
10.          handle->timer_cb,
11.          handle->repeat,
12.          handle->repeat);
13. }
14.
15. return 0;
16. }
```

10.1.7 计算二叉堆中超时时间最小值

超时时间最小值，主要用于判断 Poll IO 节点是阻塞的最长时间。

```
1. // 计算最小堆中最小节点的超时时间，即最小的超时时间
2. int uv_next_timeout(const uv_loop_t* loop) {
3.     const struct heap_node* heap_node;
4.     const uv_timer_t* handle;
5.     uint64_t diff;
6.     // 取出堆的根节点，即超时时间最小的
7.     heap_node = heap_min(timer_heap(loop));
8.     if (heap_node == NULL)
9.         return -1; /* block indefinitely */
10.
11.    handle = container_of(heap_node, uv_timer_t, heap_node);
12.    // 如果最小的超时时间小于当前时间，则返回 0，说明已经超时
13.    if (handle->timeout <= loop->time)
14.        return 0;
15.    // 否则计算还有多久超时，返回给 epoll，epoll 的 timeout 不能大于
16.    // diff
16.    diff = handle->timeout - loop->time;
17.    if (diff > INT_MAX)
18.        diff = INT_MAX;
19.
20.    return diff;
21. }
```

10.1.8 处理定时器

处理超时定时器就是遍历二叉堆，判断哪个节点超时了。

```
1. // 找出已经超时的节点，并且执行里面的回调
```

```

2. void uv_run_timers(uv_loop_t* loop) {
3.     struct heap_node* heap_node;
4.     uv_timer_t* handle;
5.
6.     for (;;) {
7.         heap_node = heap_min(timer_heap(loop));
8.         if (heap_node == NULL)
9.             break;
10.
11.        handle = container_of(heap_node, uv_timer_t, heap_node);
12.        // 如果当前节点的时间大于当前时间则返回，说明后面的节点也没有超
时
13.        if (handle->timeout > loop->time)
14.            break;
15.        // 移除该计时器节点，重新插入最小堆，如果设置了 repeat 的话
16.        uv_timer_stop(handle);
17.        uv_timer_again(handle);
18.        // 执行超时回调
19.        handle->timer_cb(handle);
20.    }
21.}

```

10.2 核心数据结构

10.2.1 TimersList

相对超时时间一样的定时器会被放到同一个队列，比如当前执行 `setTimeout(()=>{}, 10000)` 和 5 秒后执行 `setTimeout(()=>{}, 10000)`，这两个任务就会在同一个 List 中，这个队列由 TimersList 来管理。对应图 1 中的 List 那个队列。

```

1. function TimersList(expiry, msecs) {
2.     // 用于链表
3.     this._idleNext = this;
4.     this._idlePrev = this;
5.     this.expiry = expiry;
6.     this.id = timerListId++;
7.     this.msecs = msecs;
8.     // 在优先队列里的位置
9.     this.priorityQueuePosition = null;
10.}

```

expiry 记录的是链表中最快超时的节点的绝对时间。每次执行定时器阶段时会动态更新，msecs 是超时时间的相对值（相对插入时的当前时间）。用于计算该链表中的节点是否超时。后续我们会看到具体的用处。

10.2.2 优先队列

```
1. const timerListQueue = new PriorityQueue(compareTimersLists, setPosition)
```

Node.js 用优先队列对所有 TimersList 链表进行管理，优先队列本质是一个二叉堆（小根堆），每个 TimersList 链表在二叉堆里对应一个节点。根据 TimersList 的结构，我们知道每个链表都保存链表中最快到期的节点的过期时间。二叉堆以该时间为依据，即最快到期的 list 对应二叉堆中的根节点。根节点的到期时间就是整个 Node.js 定时器最快到期的时间，Node.js 把 Libuv 中定时器节点的超时时间设置为该值，在事件循环的定时器阶段就会处理定时的节点，并且不断遍历优先队列，判断当前节点是否超时，如果超时了，就需要处理，如果没有超时，说明整个二叉堆的节点都没有超时。然后重新设置 Libuv 定时器节点新的到期时间。

另外，Node.js 中用一个 map 保存了超时时间到 TimersList 链表的映射关系。这样就可以根据相对超时时间快速找到对应的列表，利用空间换时间。了解完定时器整体的组织和核心数据结构，我们可以开始进入真正的源码分析了。

10.3 设置定时器处理函数

Node.js 在初始化的时候设置了处理定时器的函数。

```
setupTimers(processImmediate, processTimers);  
setupTimers 对应的 C++ 函数是
```

```
1. void SetupTimers(const FunctionCallbackInfo<Value>& args) {  
2.     auto env = Environment::GetCurrent(args);  
3.     env->set_immediate_callback_function(args[0].As<Function>());  
4.     env->set_timers_callback_function(args[1].As<Function>());  
5. }
```

SetupTimers 在 env 中保存了两个函数，processImmediate 是处理 setImmediate 的，processTimers 是处理定时器的。当有节点超时时，Node.js 会执行该函数处理超时的节点，后续会看到该函数的具体处理逻辑。下面我们看一下如何设置一个定时器。

10.4 设置定时器

```
1. function setTimeout(callback, after, arg1, arg2, arg3) {  
2.     // 忽略处理参数 args 逻辑  
3.     // 新建一个 Timeout 对象  
4.     const timeout = new Timeout(callback,
```

```

5.                     after,
6.                     args,
7.                     false,
8.                     true);
9.     insert(timeout, timeout._idleTimeout);
10.    return timeout;
11. }

```

setTimeout 主要包含两个操作，new Timeout 和 insert。我们逐个分析一下。

1 setTimeout

```

1. function Timeout(callback, after, args, isRepeat, isRefed) {
2.     after *= 1; // Coalesce to number or NaN
3.     // 关于 setTimeout 的超时时间为 0 的问题在这里可以揭开迷雾
4.     if (!(after >= 1 && after <= TIMEOUT_MAX)) {
5.         after = 1;
6.     }
7.     // 超时时间相对值
8.     this._idleTimeout = after;
9.     // 前后指针，用于链表
10.    this._idlePrev = this;
11.    this._idleNext = this;
12.    // 定时器的开始时间
13.    this._idleStart = null;
14.    // 超时回调
15.    this._onTimeout = callback;
16.    // 执行回调时传入的参数
17.    this._timerArgs = args;
18.    // 是否定期触发超时，用于 setInterval
19.    this._repeat = isRepeat ? after : null;
20.    this._destroyed = false;
21.    // this._idleStart = now();
22.    // 激活底层的定时器节点（二叉堆的节点），说明有定时节点需要处理
23.    if (isRefed)
24.        incRefCount();
25.    // 记录状态
26.    this[kRefed] = isRefed;
27. }

```

Timeout 主要是新建一个对象记录一些定时器的相对超时时间（用于支持 setInterval，重新插入队列时找到所属队列）、开始时间（用于计算定时器是否超时）等上下文信息。这里有一个关键的逻辑是 isRefed 的值。Node.js 支持 ref 和 unref 状态的定时器

(`setTimeout` 和 `setUnrefTimeout`)，`unref` 状态的定时器，不会影响事件循环的退出。即当只有 `unref` 状态的定时器时，事件循环会结束。当 `isRefed` 为 `true` 时会执行 `incRefCount()`；

```
1. function incRefCount() {
2.   if (refCount++ === 0)
3.     toggleTimerRef(true);
4. }
5.
6. void ToggleTimerRef(const FunctionCallbackInfo<Value>& args) {
7.   Environment::GetCurrent(args)->ToggleTimerRef(args[0]->IsTrue());
8. }
9.
10. void Environment::ToggleTimerRef(bool ref) {
11.   if (started_cleanup_) return;
12.   // 打上 ref 标记,
13.   if (ref) {
14.     uv_ref(reinterpret_cast<uv_handle_t*>(timer_handle()));
15.   } else {
16.     uv_unref(reinterpret_cast<uv_handle_t*>(timer_handle()));
17.   }
18. }
```

我们看到最终会调用 Libuv 的 `uv_ref` 或 `uv_unref` 修改定时器相关 `handle` 的状态，因为 `Node.js` 只会在 `Libuv` 中注册一个定时器 `handle` 并且是常驻的，如果 `JS` 层当前没有设置定时器，则需要修改定时器 `handle` 的状态为 `unref`，否则会影响事件循环的退出。`refCount` 值便是记录 `JS` 层 `ref` 状态的定时器个数的。所以当我们第一次执行 `setTimeout` 的时候，`Node.js` 会激活 `Libuv` 的定时器节点。接着我们看一下 `insert`。

```
1. let nextExpiry = Infinity;
2. function insert(item, msecs, start = getLibuvNow()) {
3.   msecs = MathTrunc(msecs);
4.   // 记录定时器的开始时间, 见 Timeout 函数的定义
5.   item._idleStart = start;
6.   // 该相对超时时间是否已经存在对应的链表
7.   let list = timerListMap[msecs];
8.   // 还没有
9.   if (list === undefined) {
10.     // 算出绝对超时时间, 第一个节点是该链表中最早到期的节点
11.     const expiry = start + msecs;
12.     // 新建一个链表
13.     timerListMap[msecs] = list = new TimersList(expiry, msecs)
14.     ;
15.     // 插入优先队列
16.     timerListQueue.insert(list);
17.     /*
18.       nextExpiry 记录所有超时节点中最快到期的节点,
19.       如果有更快到期的, 则修改底层定时器节点的过期时间
20.     */
21. }
```

```

20.     if (nextExpiry > expiry) {
21.         // 修改底层超时节点的超时时间
22.         scheduleTimer(msecs);
23.         nextExpiry = expiry;
24.     }
25. }
26. // 把当前节点加到链表里
27. L.append(list, item);
28.

```

Insert 的主要逻辑如下

- 1 如果该超时时间还没有对应的链表，则新建一个链表，每个链表都会记录该链表中最快到期的节点的值，即第一个插入的值。然后把链表插入优先队列，优先队列会根据该链表的最快过期时间的值，把链表对应的节点调整到相应的位置
- 2 如果当前设置的定时器，比之前所有的定时器都快到期，则需要修改底层的定时器节点，使得更快触发超时。
- 3 把当前的定时器节点插入对应的链表尾部。即该链表中最久超时的节点。

假设我们在 0s 的时候插入一个节点，下面是插入第一个节点时的结构图如图 10-2 所示。

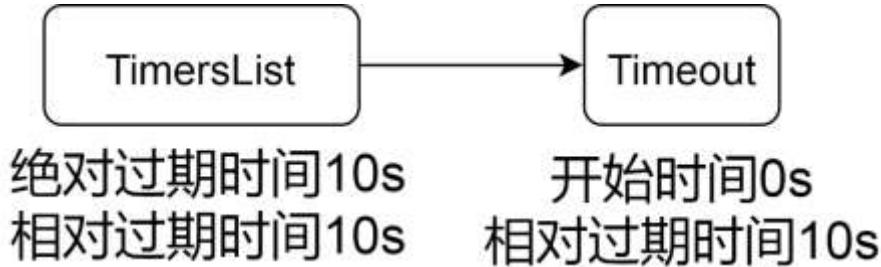


图 10-2

下面我们看一下多个节点的情况。假设 0s 的时候插入两个节点 10s 过期和 11s 过期。如图 10-3 所示。

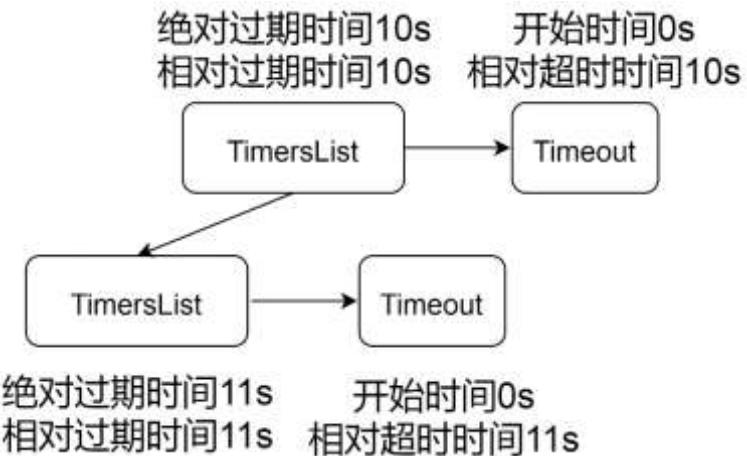


图 10-3

然后在 1s 的时候，插入一个新的 11s 过期的节点，9s 的时候插入一个新的 10s 过期节点。我们看一下这时候的关系图如图 10-4 所示。

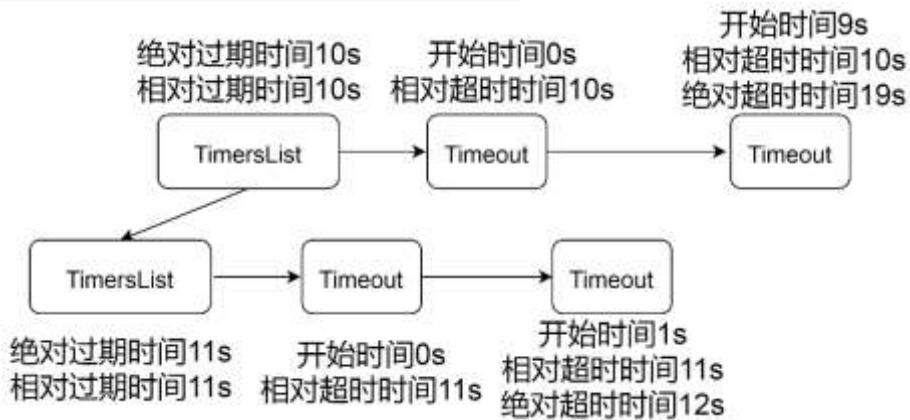


图 10-4

我们看到优先队列中，每一个节点是一个链表，父节点对应的链表的第一个元素是比子节点链表的第一个元素先超时的，但是链表中后续节点的超时就不一定。比如子节点 1s 开始的节点就比父节点 9s 开始的节点先超时。因为同一队列，只是相对超时时间一样，而还有一个重要的因素是开始的时间。虽然某节点的相对超时时间长，但是如果它比另一个节点开始的早，那么就有可能比它先超时。后续我们会看到具体是怎么实现的。

10.5 处理定时器

前面我们讲到了设置定时器处理函数和设置一个定时器，但是在哪触发这个处理定时器的函数呢？答案在 `scheduleTimer` 函数。Node.js 的实现中，所有 JS 层设置的定时器对应 Libuv 的一个定时器节点，Node.js 维护了 JS 层所有定时器的超时最小值。在第一个设置定时器或者设置一个新的定时器时，如果新设置的定时器比当前的最小值小，则会通过 `scheduleTimer` 修改超时时间。超时的时候，就会执行回调。`scheduleTimer` 函数是对 C++ 函数的封装。

```
1. void ScheduleTimer(const FunctionCallbackInfo<Value>& args) {
2.     auto env = Environment::GetCurrent(args);
3.     env->ScheduleTimer(args[0]->IntegerValue(env->context()).FromJust())
4. ;
5.
6. void Environment::ScheduleTimer(int64_t duration_ms) {
7.     if (started_cleanup_) return;
8.     uv_timer_start(timer_handle(), RunTimers, duration_ms, 0);
```

9. }

uv_timer_start 就是开启底层计时，即往 Libuv 的二叉堆插入一个节点（如果该 handle 已经存在二叉堆，则先删除）。超时时间是 duration_ms，就是最快到期的时间，超时回调是 RunTimers，在 timer 阶段会判断是否过期。是的话执行 RunTimers 函数。我们先看一下 RunTimers 函数的主要代码。

```
1. Local<Function> cb = env->timers_callback_function();
2. ret = cb->Call(env->context(), process, 1, &arg);
```

RunTimers 会执行 timers_callback_function。timers_callback_function 是在 Node.js 初始化的时候设置的 processTimers 函数。现在我们知道 Node.js 是如何设置超时的处理函数，也知道了什么时候会执行该回调。那我们就来看一下回调时具体处理逻辑。

```
1. void Environment::RunTimers(uv_timer_t* handle) {
2.     Local<Function> cb = env->timers_callback_function();
3.     MaybeLocal<Value> ret;
4.     Local<Value> arg = env->GetNow();
5.
6.     do {
7.         // 执行 js 回调 processTimers 函数
8.         ret = cb->Call(env->context(), process, 1, &arg);
9.     } while (ret.IsEmpty() && env->can_call_into_js());
10.
11.    // 如果还有未超时的节点，则 ret 为第一个未超时的节点的超时时间
12.    int64_t expiry_ms = retToLocalChecked()->IntegerValue(env->content()).FromJust();
13.    uv_handle_t* h = reinterpret_cast<uv_handle_t*>(handle);
14.
15.    /*
16.        1 等于 0 说明所有节点都执行完了，但是定时器节点还是在 Libuv 中，
17.        不过改成非激活状态，即不会影响 Libuv 退出，因为当前没有需要处
理的节点了 (handle) ,
18.        2 不等于 0 说明没有还要节点需要处理，这种情况又分为两种
19.            1 还有激活状态的定时器，即不允许事件循环退出
20.            2 定时器都是非激活状态的，允许事件循环退出
21.            具体见 Timeout 的 unref 和 ref 方法
22.    */
23.    if (expiry_ms != 0) {
24.        // 算出下次超时的相对值
25.        int64_t duration_ms =
26.            llabs(expiry_ms) - (uv_now(env->event_loop()) - env->
timer_base());
```

```
27.      // 重新把 handle 插入 Libuv 的二叉堆
28.      env->ScheduleTimer(duration_ms > 0 ? duration_ms : 1);
29.      /*
30.          见 internal/timer.js 的 processTimers
31.          1 大于 0 说明还有节点没超时，并且不允许事件循环退出，
32.              需要保持定时器的激活状态（如果之前是激活状态则不影响），
33.          2 小于 0 说明定时器不影响 Libuv 的事件循环的结束，改成非激活状
34.              态
35.          */
36.          if (expiry_ms > 0)
37.              uv_ref(h);
38.          else
39.              uv_unref(h);
40.          } else {
41.              uv_unref(h);
42. }
```

该函数主要是执行回调，然后如果还有没超时的节点，重新设置 Libuv 定时器的时间。看看 JS 层面。

```
1.  function processTimers(now)  {
2.      nextExpiry = Infinity;
3.      let list;
4.      let ranAtLeastOneList = false;
5.      // 取出优先队列的根节点，即最快到期的节点
6.      while (list = timerListQueue.peek())  {
7.          // 还没过期，则取得下次到期的时间，重新设置超时时间
8.          if (list.expiry > now)  {
9.              nextExpiry = list.expiry;
10.             // 返回下一次过期的时间，负的说明允许事件循环退出
11.             return refCount > 0 ? nextExpiry : -
12.                 nextExpiry;
13.         }
14.         // 处理超时节点
15.         listOnTimeout(list, now);
16.     }
17.     // 所有节点都处理完了
18.     return 0;
19. }
20.
21. function listOnTimeout(list, now)  {
```

```

22.     const msecs = list.msecs;
23.     let ranAtLeastOneTimer = false;
24.     let timer;
25.     // 遍历具有统一相对过期时间的队列
26.     while (timer = L.peek(list)) {
27.         // 算出已经过去的时间
28.         const diff = now - timer._idleStart;
29.         // 过期的时间比超时时间小，还没过期
30.         if (diff < msecs) {
31.             /*
32.                 整个链表节点的最快过期时间等于当前
33.                 还没过期节点的值，链表是有序的
34.             */
35.             list.expiry = MathMax(timer._idleStart + msecs,
36.                                   now + 1);
37.             // 更新 id，用于决定在优先队列里的位置
38.             list.id = timerListId++;
39.             /*
40.                 调整过期时间后，当前链表对应的节点不一定是优先队列
41.                 里的根节点了，可能有它更快到期，即当前链表对应的节
42.                 点可能需要往下沉
43.             */
44.             timerListQueue.percolateDown(1);
45.             return;
46.         }
47.
48.         // 准备执行用户设置的回调，删除这个节点
49.         L.remove(timer);
50.
51.         let start;
52.         if (timer._repeat)
53.             start = getLibuvNow();
54.         try {
55.             const args = timer._timerArgs;
56.             // 执行用户设置的回调
57.             if (args === undefined)
58.                 timer._onTimeout();
59.             else
60.                 timer._onTimeout(...args);
61.         } finally {
62.             /*
63.                 设置了重复执行回调，即来自 setInterval。

```

```
64.             则需要重新加入链表。
65.             */
66.             if (timer._repeat &&
67.                 timer._idleTimeout != -1) {
68.                     // 更新超时时间，一样的时间间隔
69.                     timer._idleTimeout = timer._repeat;
70.                     // 重新插入链表
71.                     insert(timer, timer._idleTimeout, start);
72.                 } else if (!timer._idleNext &&
73.                            !timer._idlePrev &&
74.                            !timer._destroyed) {
75.                     timer._destroyed = true;
76.                     // 是 ref 类型，则减去一个，防止阻止事件循环退出
77.                     if (timer[kRefed])
78.                         refCount--;
79.                 }
80.             // 为空则删除
81.             if (list === timerListMap[msecs]) {
82.                 delete timerListMap[msecs];
83.                 // 从优先队列中删除该节点，并调整队列结构
84.                 timerListQueue.shift();
85.             }
86.         }
```

上面的代码主要是遍历优先队列

1 如果当前节点超时，则遍历它对应的链表。遍历链表的时候如果遇到超时的节点则执行。如果遇到没有超时的节点，则说明后面的节点也不会超时了，因为链表是有序的，接着重新计算出最快超时时间，修改链表的 expiry 字段。调整在优先队列的位置。因为修改后的 expiry 可能会导致位置发生变化。如果链表的节点全部都超时了，则从优先队列中删除链表对应的节点。重新调整优先队列的节点。

2 如果当前节点没有超时则说明后面的节点也不会超时了。因为当前节点是优先队列中最快到期（最小的）的节点。接着设置 Libuv 的定时器时间为当前节点的时间。等待下一次超时处理。

10.6 ref 和 unref

setTimeout 返回的是一个 Timeout 对象，该提供了 ref 和 unref 接口，刚才提到了关于定时器影响事件循环退出的内容，我们看一下这个原理。刚才说到 Node.js 定时器模块在 Libuv 中只对应一个定时器节点。在 Node.js 初始化的时候，初始化了该节点。

```
1. void Environment::InitializeLibuv(bool start_profiler_idle_notifi
   er) {
```

```

2. // 初始化定时器
3. CHECK_EQ(0, uv_timer_init(event_loop(), timer_handle()));
4. // 置 unref 状态
5. uv_unref(reinterpret_cast<uv_handle_t*>(timer_handle()));
6. }
```

我们看到底层定时器节点默认是 unref 状态的，所以不会影响事件循环的退出。因为初始化时 JS 层没有定时节点。可以通过 Node.js 提供的接口修改该状态。Node.js 支持 ref 状态的 Timeout (setTimeout) 和 unref 状态的 Timeout (setUnrefTimeout)。

```

1. function Timeout(callback, after, args, isRepeat, isRefed) {
2.   if (isRefed)
3.     incRefCount();
4.   this[kRefed] = isRefed;
5. }
```

最后一个参数就是控制 ref 还是 unref 的。我们继续看一下如果 isRefed 为 true 的时候的逻辑

```

1. function incRefCount() {
2.   if (refCount++ === 0)
3.     toggleTimerRef(true);
4. }
```

refCount 初始化的时候是 1，所以在新加第一个 Timeout 的时候，if 成立。我们接着看 toggleTimerRef，该函数对应的代码如下

```

1. void Environment::ToggleTimerRef(bool ref) {
2.   // 打上 ref 标记,
3.   if (ref) {
4.     uv_ref(reinterpret_cast<uv_handle_t*>(timer_handle()));
5.   } else {
6.     uv_unref(reinterpret_cast<uv_handle_t*>(timer_handle()));
7.   }
8. }
```

该函数正是给定时器对应的 handle 设置状态的。setTimeout 的时候，isRefed 的值是 true 的，Node.js 还提供了另外一个函数 setUnrefTimeout。

```

1. function setUnrefTimeout(callback, after) {
2.   const timer = new Timeout(callback, after, undefined, false, false);
3.   insert(timer, timer._idleTimeout);
4.   return timer;
5. }
```

该函数和 setTimeout 最主要的区别是 new Timeout 的时候，最后一个参数是 false (isRefed 变量的值)，所以 setUnrefTimeout 设置的定时器是不会影响 Libuv 事件循环退出的。另外除了 Node.js 直接提供的 api 后。我们还可以通过 Timeout 对象提供的 ref 和 unref 手动控制这个状态。

现在通过一个例子具体来看一下。

```
1. const timeout = setTimeout(() => {
2.   console.log(1)
3. }, 10000);
4. timeout.unref();
5. // timeout.ref(); 加这一句会输出 1
```

上面的代码中，1 是不会输出，除非把注释去掉。Unref 和 ref 是相反的参数，即把定时器模块对应的 Libuv handle 改成 unref 状态。

第十一章 setImmediate 和 nextTick

setImmediate 对应 Libuv 的 check 阶段。所提交的任务会在 Libuv 事件循环的 check 阶段被执行，check 阶段的任务会在每一轮事件循环中被执行，但是 setImmediate 提交的任务只会执行一次，下面我们会看到 Node.js 是怎么处理的，我们看一下具体的实现。

11.1 setImmediate

11.1.1 设置处理 immediate 任务的函数

在 Node.js 初始化的时候，设置了处理 immediate 任务的函数

```
1. // runNextTicks 用于处理 nextTick 产生的任务，这里不关注
2. const { processImmediate, processTimers } = getTimerCallbacks(runNextTicks);
3. setupTimers(processImmediate, processTimers);
```

我们先看看一下 setupTimers (timer.cc) 的逻辑。

```
1. void SetupTimers(const FunctionCallbackInfo<Value>& args) {
2.   auto env = Environment::GetCurrent(args);
3.   env->set_immediate_callback_function(args[0].As<Function>());
4.   env->set_timers_callback_function(args[1].As<Function>());
5. }
```

SetupTimers 在 env 中保存了两个函数 processImmediate, processTimers, processImmediate 是处理 immediate 任务的, processTimers 是处理定时器任务的, 在定时器章节我们已经分析过。

11.1.2 注册 check 阶段的回调

在 Node.js 初始化的时候, 同时初始化了 immediate 任务相关的数据结构和逻辑。

```

1. void Environment::InitializeLibuv(bool start_profiler_idle_notifier) {
2.     // 初始化 immediate 相关的 handle
3.     uv_check_init(event_loop(), immediate_check_handle());
4.     // 修改状态为 unref, 避免没有任务的时候, 影响事件循环的退出
5.     uv_unref(reinterpret_cast<uv_handle_t*>(immediate_check_handle(
    )));
6.     // 激活 handle, 设置回调
7.     uv_check_start(immediate_check_handle(), CheckImmediate);
8.     // 在 idle 阶段也插入一个相关的节点
9.     uv_idle_init(event_loop(), immediate_idle_handle());
10. }
```

Node.js 默认会往 check 阶段插入一个节点, 并设置回调为 `CheckImmediate`, 但是初始化状态是 unref 的, 所以如果没有 immediate 任务的话, 不会影响事件循环的退出。我们看一下 `CheckImmediate` 函数

```

1. void Environment::CheckImmediate(uv_check_t* handle) {
2.     // 省略部分代码
3.     // 没有 Immediate 节点需要处理
4.     if (env->immediate_info()->count() == 0 || 
5.         !env->can_call_into_js())
6.         return;
7.     do {
8.         // 执行 JS 层回调 immediate_callback_function
9.         MakeCallback(env->isolate(),
10.                      env->process_object(),
11.                      env->immediate_callback_function(),
12.                      0,
13.                      nullptr,
14.                      {0, 0}).ToLocalChecked();
15.     } while (env->immediate_info()->has_outstanding() &&
16.              env->can_call_into_js());
17. /*
18.     所有 immediate 节点都处理完了, 置 idle 阶段对应节点为非激活状态,
```

```
19.     允许 Poll IO 阶段阻塞和事件循环退出
20.     */
21.     if (env->immediate_info()->ref_count() == 0)
22.         env->ToggleImmediateRef(false);
23. }
```

我们看到每一轮事件循环时，CheckImmediate 都会被执行，但是如果不需要处理的任务则直接返回。如果有任务，CheckImmediate 函数执行 immediate_callback_function 函数，这正是 Node.js 初始化的时候设置的函数 processImmediate。看完初始化和处理 immediate 任务的逻辑后，我们看一下如何产生一个 immediate 任务。

11.1.3 setImmediate 生成任务

我们可以通过 setImmediate 生成一个任务。

```
1. function setImmediate(callback, arg1, arg2, arg3) {
2.     let i, args;
3.     switch (arguments.length) {
4.         case 1:
5.             break;
6.         case 2:
7.             args = [arg1];
8.             break;
9.         case 3:
10.            args = [arg1, arg2];
11.            break;
12.        default:
13.            args = [arg1, arg2, arg3];
14.            for (i = 4; i < arguments.length; i++) {
15.                args[i - 1] = arguments[i];
16.            }
17.            break;
18.    }
19.
20.    return new Immediate(callback, args);
21. }
```

setImmediate 的代码比较简单，新建一个 Immediate。我们看一下 Immediate 的类。

```
1. const Immediate = class Immediate {
2.     constructor(callback, args) {
3.         this._idleNext = null;
```

```

4.         this._idlePrev = null;
5.         this._onImmediate = callback;
6.         this._argv = args;
7.         this._destroyed = false;
8.         this[kRefed] = false;
9.         this.ref();
10.        // Immediate 链表的节点个数，包括 ref 和 unref 状态
11.        immediateInfo[kCount]++;
12.        // 加入链表中
13.        immediateQueue.append(this);
14.    }
15.    // 打上 ref 标记，往 Libuv 的 idle 链表插入一个激活状态的节点，如果没有的话
16.    ref() {
17.        if (this[kRefed] === false) {
18.            this[kRefed] = true;
19.            if (immediateInfo[kRefCount]++ === 0)
20.                toggleImmediateRef(true);
21.        }
22.        return this;
23.    }
24.    // 和上面相反
25.    unref() {
26.        if (this[kRefed] === true) {
27.            this[kRefed] = false;
28.            if (--immediateInfo[kRefCount] === 0)
29.                toggleImmediateRef(false);
30.        }
31.        return this;
32.    }
33.
34.    hasRef() {
35.        return !!this[kRefed];
36.    }
37.};

```

Immediate 类主要做了两个事情。

1 生成一个节点插入到链表。

```

1. const immediateQueue = new ImmediateList();
2.

```

```
3. // 双向非循环的链表
4. function ImmediateList() {
5.     this.head = null;
6.     this.tail = null;
7. }
8. ImmediateList.prototype.append = function(item) {
9.     // 尾指针非空, 说明链表非空, 直接追加在尾节点后面
10.    if (this.tail !== null) {
11.        this.tail._idleNext = item;
12.        item._idlePrev = this.tail;
13.    } else {
14.        // 尾指针是空说明链表是空的, 头尾指针都指向 item
15.        this.head = item;
16.    }
17.    this.tail = item;
18. };
19.
20. ImmediateList.prototype.remove = function(item) {
21.     // 如果 item 在中间则自己全身而退, 前后两个节点连上
22.     if (item._idleNext !== null) {
23.         item._idleNext._idlePrev = item._idlePrev;
24.     }
25.
26.     if (item._idlePrev !== null) {
27.         item._idlePrev._idleNext = item._idleNext;
28.     }
29.     // 是头指针, 则需要更新头指针指向 item 的下一个, 因为 item 被删除了,
30.     // 尾指针同理
31.     if (item === this.head)
32.         this.head = item._idleNext;
33.     if (item === this.tail)
34.         this.tail = item._idlePrev;
35.     // 重置前后指针
36.     item._idleNext = null;
37.     item._idlePrev = null;
37.};
```

2 如果还没有往 Libuv 的 idle 链表里插入一个激活节点的话, 则插入一个。从之前的分析, 我们知道, Node.js 在 check 阶段插入了一个 unref 节点, 在每次 check 阶段都会执行该节点的回调, 那么这个 idle 节点有什么用呢? 答案在 uv_backend_timeout 函数中, uv_backend_timeout 定义了 Poll IO 阻塞的时长, 如果有 ref 状态的 idle 节点则 Poll IO 阶段不会阻塞 (但是不会判断是否有 check 节点)。所以当有 immediate 任务

时，Node.js 会把这个 idle 插入 idle 阶段中，表示有任务处理，不能阻塞 Poll IO 阶段。没有 immediate 任务时，则移除 idle 节点。总的来说，idle 节点的意义是标记是否有 immediate 任务需要处理，有的话就不能阻塞 Poll IO 阶段，并且不能退出事件循环。

```

1. void ToggleImmediateRef(const FunctionCallbackInfo<Value>& args) {
2.     Environment::GetCurrent(args)->ToggleImmediateRef(args[0]->IsTrue())
3. }
4.
5. void Environment::ToggleImmediateRef(bool ref) {
6.     if (started_cleanup_) return;
7.     // 改变 handle 的状态（激活或不激活），防止在 Poll IO 阶段阻塞
8.     if (ref) {
9.         uv_idle_start(immediate_idle_handle(), [] (uv_idle_t*) { });
10.    } else {
11.        // 不阻塞 Poll IO，允许事件循环退出
12.        uv_idle_stop(immediate_idle_handle());
13.    }
14. }
```

这是 setImmediate 函数的整个过程。和定时器一样，我们可以调用 immediate 任务的 ref 和 unref 函数，控制它对事件循环的影响。

11.1.4 处理 `setImmediate` 产生的任务

最后我们看一下在 check 阶段时，是如何处理 immediate 任务的。由前面分析我们知道 processImmediate 函数是处理 immediate 任务的函数，来自 getTimerCallbacks (internal/timer.js)。

```

1. function processImmediate() {
2.     /*
3.      上次执行 processImmediate 的时候如果由未捕获的异常，
4.      则 outstandingQueue 保存了未执行的节点，下次执行 processImmediate
5.      的时候，
6.      优先执行 outstandingQueue 队列的节点
7.     */
8.     const queue = outstandingQueue.head !== null ?
9.         outstandingQueue : immediateQueue;
10.    let immediate = queue.head;
11.    /*
12.      在执行 immediateQueue 队列的话，先置空队列，避免执行回调
13.      的时候一直往队列加节点，死循环。所以新加的接口会插入新的队
列，
```

```
13.     不会在本次被执行。并打一个标记，全部 immediateQueue 节点都被执
14.     行则清空，否则会再执行 processImmediate 一次，见
    Environment::CheckImmediate
15.     */
16.     if (queue != outstandingQueue) {
17.         queue.head = queue.tail = null;
18.         immediateInfo[kHasOutstanding] = 1;
19.     }
20.
21.     let prevImmediate;
22.     let ranAtLeastOneImmediate = false;
23.     while (immediate != null) {
24.         // 执行微任务
25.         if (ranAtLeastOneImmediate)
26.             runNextTicks();
27.         else
28.             ranAtLeastOneImmediate = true;
29.
30.         // 微任务把该节点删除了，则不需要指向它的回调了，继续下一
    个
31.         if (immediate._destroyed) {
32.             outstandingQueue.head = immediate = prevImmediate._idl
    eNext;
33.             continue;
34.         }
35.
36.         immediate._destroyed = true;
37.         // 执行完要修改个数
38.         immediateInfo[kCount]--;
39.         if (immediate[kRefed])
40.             immediateInfo[kRefCount]--;
41.         immediate[kRefed] = null;
42.         // 见上面 if (immediate._destroyed) 的注释
43.         prevImmediate = immediate;
44.         // 执行回调，指向下一个节点
45.         try {
46.             const argv = immediate._argv;
47.             if (!argv)
48.                 immediate._onImmediate();
49.             else
50.                 immediate._onImmediate(...argv);
51.         } finally {
```

```

52.         immediate._onImmediate = null;
53.         outstandingQueue.head = immediate = immediate._idleNext
  t;
54.     }
55. }
56. // 当前执行的是 outstandingQueue 的话则把它清空
57. if (queue === outstandingQueue)
58.     outstandingQueue.head = null;
59. // 全部节点执行完
60. immediateInfo[kHasOutstanding] = 0;
61. }

```

processImmediate 的逻辑就是逐个执行 immediate 任务队列的节点。Immediate 分两个队列，正常情况下，插入的 immediate 节点插入到 immediateQueue 队列。如果执行的时候有异常，则未处理完的节点就会被插入到 outstandingQueue 队列，等下一次执行。另外我们看到 runNextTicks。runNextTicks 在每执行完 immediate 节点后，都先处理 tick 任务然后再处理下一个 immediate 节点。

11.1.5 Node.js 的 setTimeout(fn,0) 和 setImmediate 谁先执行的问题

我们首先看一下下面这段代码

```

1. setTimeout(()=>{ console.log('setTimeout'); },0)
2. setImmediate(()=>{ console.log('setImmediate'); })

```

我们执行上面这段代码，会发现输出是不确定的。下面来看一下为什么。Node.js 的事件循环分为几个阶段(phase)。setTimeout 是属于定时器阶段，setImmediate 是属于 check 阶段。顺序上定时器阶段是比 check 更早被执行的。其中 setTimeout 的实现代码里有一个很重要的细节。

```

1. after *= 1; // coalesce to number or NaN
2. if (!(after >= 1 && after <= TIMEOUT_MAX)) {
3.     if (after > TIMEOUT_MAX) {
4.         process.emitWarning(`错误提示`);
5.     }
6.     after = 1; // schedule on next tick, follows browser b
  ehavior
7. }

```

我们发现虽然我们传的超时时间是 0，但是 0 不是合法值，Node.js 会把超时时间变成 1。这就是导致上面的代码输出不确定的原因。我们分析一下这段代码的执行过程。Node.js 启动的时候，会编译执行上面的代码，开始一个定时器，挂载一个 setImmediate

节点在队列。然后进入 Libuv 的事件循环，然后执行定时器阶段，Libuv 判断从开启定时器到现在是否已经过去了 1 毫秒，是的话，执行定时器回调，否则执行下一个节点，执行完其它阶段后，会执行 check 阶段。这时候就会执行 setImmediate 的回调。所以，一开始的那段代码的输出结果是取决于启动定时器的时间到 Libuv 执行定时器阶段是否过去了 1 毫秒。

11.2 nextTick

nextTick 用于异步执行一个回调函数，和 setTimeout、setImmediate 类似，不同的地方在于他们的执行时机，setTimeout 和 setImmediate 的任务属于事件循环的一部分，但是 nextTick 的任务不属于事件循环的一部分，具体的执行时机我们会在本节分析。

11.2.1 初始化 nextTick

nextTick 函数是在 Node.js 启动过程中，在执行 bootstrap/node.js 时挂载到 process 对象中。

```
1. const { nextTick, runNextTicks } = setupTaskQueue();
2. process.nextTick = nextTick;
```

真正的定义在 task_queues.js。

```
1. setupTaskQueue() {
2.     setTickCallback(processTicksAndRejections);
3.     return {
4.         nextTick,
5.     };
6. },
```

nextTick 接下来会讲，setTickCallback 是注册处理 tick 任务的函数，

```
1. static void SetTickCallback(const FunctionCallbackInfo<Value>& args)
{
2.     Environment* env = Environment::GetCurrent(args);
3.     CHECK(args[0]->IsFunction());
4.     env->set_tick_callback_function(args[0].As<Function>());
5. }
```

只是简单地保存处理 tick 任务的函数。后续会用到

11.2.2 nextTick 生产任务

```

1. function nextTick(callback) {
2.     let args;
3.     switch (arguments.length) {
4.         case 1: break;
5.         case 2: args = [arguments[1]]; break;
6.         case 3: args = [arguments[1], arguments[2]]; break;
7.         case 4: args = [arguments[1], arguments[2], arguments[3]];
8.     }
9.     default:
10.         args = new Array(arguments.length - 1);
11.         for (let i = 1; i < arguments.length; i++)
12.             args[i - 1] = arguments[i];
13.     // 第一个任务，开启 tick 处理逻辑
14.     if (queue.isEmpty())
15.         setHasTickScheduled(true);
16.     const asyncId = newAsyncId();
17.     const triggerAsyncId = getDefaultTriggerAsyncId();
18.     const tickObject = {
19.         [async_id_symbol]: asyncId,
20.         [trigger_async_id_symbol]: triggerAsyncId,
21.         callback,
22.         args
23.     };
24.     // 插入队列
25.     queue.push(tickObject);
26. }
```

这就是我们执行 nextTick 时的逻辑。每次调用 nextTick 都会往队列中追加一个节点。

11.2.3 处理 tick 任务

我们再看一下处理的 tick 任务的逻辑。Node.js 在初始化时，通过执行 `setTickCallback(processTicksAndRejections)` 注册了处理 tick 任务的函数。Node.js 在初始化时把处理 tick 任务的函数保存到 `env` 中。另外，Node.js 使用 `TickInfo` 类管理 tick 的逻辑。

```

1. class TickInfo : public MemoryRetainer {
2. public:
3.     inline AliasedUint8Array& fields();
```

```
4.     inline bool has_tick_scheduled() const;
5.     inline bool has_rejection_to_warn() const;
6. private:
7.     inline explicit TickInfo(v8::Isolate* isolate);
8.     enum Fields { kHasTickScheduled = 0, kHasRejectionToWarn, kFieldsCount };
9.
10.    AliasedUint8Array fields_;
11.};
```

TickInfo 主要是有两个标记位，kHasTickScheduled 标记是否有 tick 任务需要处理。然后通过 InternalCallbackScope 类的对象方法 Close 函数执行 tick_callback_function。当 Node.js 底层需要执行一个 js 回调时，会调用 AsyncWrap 的 MakeCallback。MakeCallback 里面调用了 InternalMakeCallback。

```
1. MaybeLocal<Value> InternalMakeCallback(Environment* env,
   Local<Object> recv,
2. const Local<Function> callback, int argc, Local<Value> argv[],
3. async_context asyncContext) {
4.     InternalCallbackScope scope(env, recv, asyncContext);
5.     // 执行用户层 js 回调
6.     scope.Close();
7.
8.     return ret;
9. }
```

我们看 InternalCallbackScope 的 Close

```
1. void InternalCallbackScope::Close() {
2.     // 省略部分代码
3.     TickInfo* tick_info = env_->tick_info();
4.     // 没有 tick 任务则不需要往下走，在插入 tick 任务的时候会设置这个为
   // true，没有任务时变成 false
5.     if (!tick_info->has_tick_scheduled() && !tick_info->has_rejection_
   _to_warn()) {
6.         return;
7.     }
8.
9.     HandleScope handle_scope(env_->isolate());
10.    Local<Object> process = env_->process_object();
11.
12.    if (!env_->can_call_into_js()) return;
```

```

13. // 处理 tick 的函数
14. Local<Function> tick_callback = env_->tick_callback_function();

15. // 处理 tick 任务
16. if (tick_callback->Call(env_->context(), process, 0, nullptr).IsEmpty())
17.     failed_ = true;
18. }
19.

```

我们看到每次执行 js 层的回调的时候，就会处理 tick 任务。Close 函数可以主动调用，或者在 InternalCallbackScope 对象析构的时候被调用。除了执行 js 回调时是主动调用 Close 外，一般处理 tick 任务的时间点就是在 InternalCallbackScope 对象被析构的时候。所以在定义了 InternalCallbackScope 对象的时候，一般就会在对象析构的时候，进行 tick 任务的处理。另外一种就是在执行的 js 回调里，调用 runNextTicks 处理 tick 任务。比如执行 immediate 任务的过程中。

```

1. function runNextTicks() {
2.   if (!hasTickScheduled() && !hasRejectionToWarn())
3.     runMicrotasks();
4.   if (!hasTickScheduled() && !hasRejectionToWarn())
5.     return;
6.   processTicksAndRejections();
7. }

```

我们看 processTicksAndRejections 是如何处理 tick 任务的。

```

1. function processTicksAndRejections() {
2.   let tock;
3.   do {
4.     while (tock = queue.shift()) {
5.       const asyncId = tock[async_id_symbol];
6.       emitBefore(asyncId, tock[trigger_async_id_symbol]);
7.
8.     try {
9.       const callback = tock.callback;
10.      if (tock.args === undefined) {
11.        callback();
12.      } else {
13.        const args = tock.args;
14.        switch (args.length) {
15.          case 1: callback(args[0]); break;
16.          case 2: callback(args[0], args[1]); break;

```

```
17.         case 3: callback(args[0], args[1], args[2]); break;
18.         case 4: callback(args[0], args[1], args[2], args[3])
19.     ; break;
20.     default: callback(...args);
21. }
22. } finally {
23.     if (destroyHooksExist())
24.         emitDestroy(asyncId);
25. }
26. emitAfter(asyncId);
27. }
28. runMicrotasks();
29. } while (!queue.isEmpty() || processPromiseRejections());
30. setHasTickScheduled(false);
31. setHasRejectionToWarn(false);
32. }
33. }
```

从 `processTicksAndRejections` 代码中，我们可以看到，Node.js 是实时从任务队列里取节点执行的，所以如果我们在 `nextTick` 的回调里一直调用 `nextTick` 的话，就会导致死循环。

```
1. function test() {
2.     process.nextTick(() => {
3.         console.log(1);
4.         test()
5.     });
6. }
7. test();
8.
9. setTimeout(() => {
10.    console.log(2)
11. }, 10)
```

上面的代码中，会一直输出 1，不会输出 2。而在 Node.js 源码的很多地方都处理了这个问题，首先把要执行的任务队列移到一个变量 `q2` 中，清空之前的队列 `q1`。接着遍历 `q2` 指向的队列，如果执行回调的时候又新增了节点，只会加入到 `q1` 中。`q2` 不会导致死循环。

11.2.4 `nextTick` 的使用

我们知道 `nextTick` 可用于延迟执行一些逻辑，我们看一下哪些场景下可以使用 `nextTick`。

```

1. const { EventEmitter } = require('events');
2. class DemoEvents extends EventEmitter {
3.   constructor() {
4.     super();
5.     this.emit('start');
6.   }
7. }
8.
9. const demoEvents = new DemoEvents();
10. demoEvents.on('start', () => {
11.   console.log('start');
12. });

```

以上代码在构造函数中会触发 start 事件，但是事件的注册却在构造函数之后执行，而在构造函数之前我们还没有拿到 DemoEvents 对象，无法完成事件的注册。这时候，我们就可以使用 `nextTick`。

```

1. const { EventEmitter } = require('events');
2. class DemoEvents extends EventEmitter {
3.   constructor() {
4.     super();
5.     process.nextTick(() => {
6.       this.emit('start');
7.     })
8.   }
9. }
10.
11. const demoEvents = new DemoEvents();
12. demoEvents.on('start', () => {
13.   console.log('start');
14. });

```

第十二章 文件

文件操作是我们使用 Node.js 时经常会用到的功能。Node.js 中，文件模块的 API 几乎都提供了同步和异步的版本。同步的 API 直接在主线程中调用操作系统提供的接口，它会导致主线程阻塞。异步 API 则是在 Libuv 提供的线程池中执行阻塞式 API 实现的。这样就不会导致主线程阻塞。文件 I/O 不同于网络 I/O，文件 I/O 由于兼容性问题，无法像网络 I/O 一样利用操作系统提供的能力直接实现异步。在 Libuv 中，文件操作是以线程池实现的，操作文件的时候，会阻塞在某个线程。所以这种异步只是对用户而言。文件模块虽然提供的接口非常多，源码也几千行，但是很多逻辑都是类似的，所以我们只讲解不同的地方。介绍文件模块之前先介绍一下 Linux 操作系统中的文件。

Linux 系统中万物皆文件，从应用层来看，我们拿到都是一个文件描述符，我们操作的也是这个文件描述符。使用起来非常简单，那是因为操作系统帮我们做了很多事情。简单来说，文件描述符只是一个索引。它的底层可以对应各种各样的资源，包括普通文件，网络，内存等。当我们操作一个资源之前，我们首先会调用操作系统的接口拿到一个文件描述符，操作系统也记录了这个文件描述符底层对应的资源、属性、操作函数等。当我们后续操作这个文件描述符的时候，操作系统就会执行对应的操作。比如我们在 write 的时候，传的文件描述符是普通文件和网络 socket，底层所做的操作是不一样的。但是我们一般不需要关注这些。我们只需要从抽象的角度去使用它。本章介绍 Node.js 中关于文件模块的原理和实现。

12.4 同步 API

在 Node.js 中，同步 API 的本质是直接在主线程里调用操作系统提供的系统调用。下面以 `readFileSync` 为例，看一下整体的流程，如图 12-1 所示。

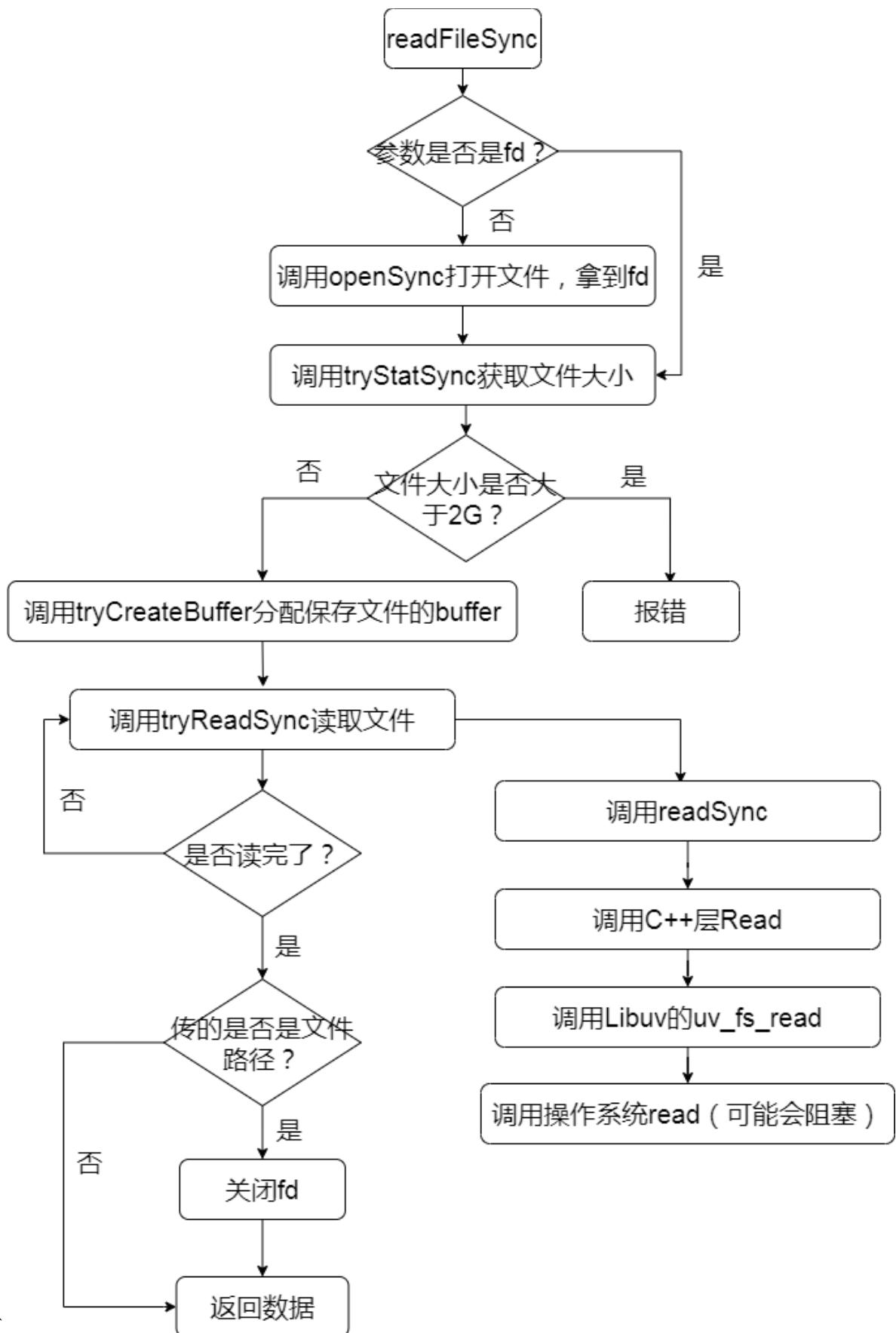


图 12-1

下面我们看一下具体的代码

```
1. function readFileSync(path, options) {
2.   options = getOptions(options, { flag: 'r' });
3.   // 传的是 fd 还是文件路径
4.   const isUserFd = isFd(path);
5.   // 传的是路径，则先同步打开文件
6.   const fd = isUserFd ? path : fs.openSync(path, options.flag,
7.     0o666);
8.   // 查看文件的 stat 信息，拿到文件的大小
9.   const stats = tryStatSync(fd, isUserFd);
10.  // 是否是一般文件
11.  const size = isFileType(stats, S_IFREG) ? stats[8] : 0;
12.  let pos = 0;
13.  let buffer;
14.  let buffers;
15.  if (size === 0) {
16.    buffers = [];
17.  } else {
18.    // 一般文件且有大小，则分配一个大小为 size 的 buffer，size 需要小
于 2G
19.    buffer = tryCreateBuffer(size, fd, isUserFd);
20.  }
21.
22.  let bytesRead;
23.  // 不断地同步读文件内容
24.  if (size !== 0) {
25.    do {
26.      bytesRead = tryReadSync(fd, isUserFd, buffer, pos, siz
e - pos);
27.      pos += bytesRead;
28.    } while (bytesRead !== 0 && pos < size);
29.  } else {
30.    do {
31.      /*
32.       * 文件大小为 0，或者不是一般文件，也尝试去读，
33.       * 但是因为不知道大小，所以只能分配一个一定大小的 buffer,
34.       * 每次读取一定大小的内容
35.    */
36.  }
```

```

36.         buffer = Buffer.allocUnsafe(8192);
37.         bytesRead = tryReadSync(fd, isUserFd, buffer, 0, 8192)
38.         ;
39.         // 把读取到的内容放到 buffers 里
40.         if (bytesRead !== 0) {
41.             buffers.push(buffer.slice(0, bytesRead));
42.         }
43.         // 记录读取到的数据长度
44.         pos += bytesRead;
45.     } while (bytesRead !== 0);
46.     // 用户传的是文件路径, Node.js 自己打开了文件, 所以需要自己关闭
47.     if (!isUserFd)
48.         fs.closeSync(fd);
49.     // 文件大小是 0 或者非一般文件的话, 如果读到了内容
50.     if (size === 0) {
51.         // 把读取到的所有内容放到 buffer 中
52.         buffer = Buffer.concat(buffers, pos);
53.     } else if (pos < size) {
54.         buffer = buffer.slice(0, pos);
55.     }
56.     // 编码
57.     if (options.encoding) buffer = buffer.toString(options.encoding)
58.     ;
59. }

```

tryReadSync 调用的是 fs.readSync，然后到 binding.read(node_file.cc 中定义的 Read 函数)。Read 函数主要逻辑如下

```

1. FSReqWrapSync req_wrap_sync;
2. const int bytesRead = SyncCall(env,
3.                                 args[6],
4.                                 &req_wrap_sync,
5.                                 "read",
6.                                 uv_fs_read,
7.                                 fd,
8.                                 &uvbuf,
9.                                 1,
10.                                pos);

```

我们看一下 SyncCall 的实现

```
1. int SyncCall(Environment* env,
2.                 v8::Local<v8::Value> ctx,
3.                 FSReqWrapSync* req_wrap,
4.                 const char* syscall,
5.                 Func fn,
6.                 Args... args) {
7.     /*
8.      req_wrap->req 是一个 uv_fs_t 结构体，属于 request 类，
9.      管理一次文件操作的请求
10.     */
11.    int err = fn(env->event_loop(),
12.                  &(req_wrap->req),
13.                  args...,
14.                  nullptr);
15.    // 忽略出错处理
16.    return err;
17. }
```

我们看到最终调用的是 Libuv 的 `uv_fs_read`，并使用 `uv_fs_t` 管理本次请求。因为是阻塞式调用，所以 Libuv 会直接调用操作系统的系统调用 `read` 函数。这是 Node.js 中同步 API 的过程。

12.5 异步 API

文件系统的 API 中，异步的实现是依赖于 Libuv 的线程池的。Node.js 把任务放到线程池，然后返回主线程继续处理其它事情，等到条件满足时，就会执行回调。我们以 `readFile` 为例讲解这个过程。异步读取文件的流程图，如图 12-2 所示。

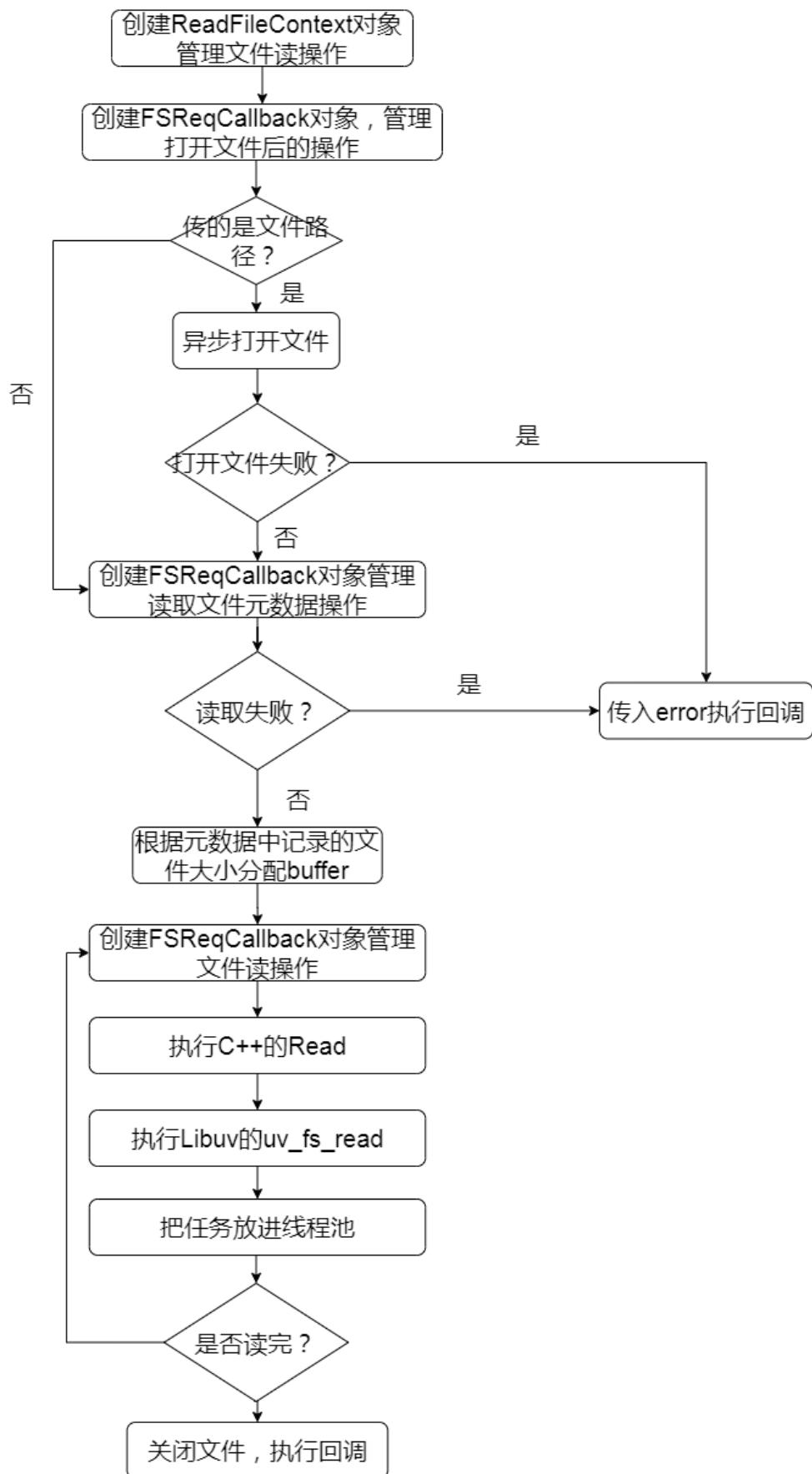


图 12-2

下面我们看具体的实现

```
1. function readFile(path, options, callback) {
2.   callback = maybeCallback(callback || options);
3.   options = getOptions(options, { flag: 'r' });
4.   // 管理文件读的对象
5.   if (!ReadFileContext)
6.     ReadFileContext = require('internal/fs/read_file_context');
7.   const context = new ReadFileContext(callback, options.encoding)
8.   // 传的是文件路径还是 fd
9.   context.isUserFd = isFd(path); // File descriptor ownership

10.  // C++层的对象，封装了uv_fs_t结构体，管理一次文件读请求
11.  const req = new FSReqCallback();
12.  req.context = context;
13.  // 设置回调，打开文件后，执行
14.  req.oncomplete = readfileAfterOpen;
15.  // 传的是fd，则不需要打开文件，下一个tick直接执行回调读取文件
16.  if (context.isUserFd) {
17.    process.nextTick(function tick() {
18.      req.oncomplete(null, path);
19.    });
20.    return;
21.  }
22.
23.  path = getValidatedPath(path);
24.  const flagsNumber = stringToFlags(options.flags);
25.  // 调用C++层open打开文件
26.  binding.open(pathModule.toNamespacedPath(path),
27.                flagsNumber,
28.                0o666,
29.                req);
30. }
```

ReadFileContext 对象用于管理文件读操作整个过程，FSReqCallback 是对 uv_fs_t 的封装，每次读操作对于 Libuv 来说就是一次请求，该请求的上下文就是使用 uv_fs_t 表示。请求完成后，会执行 FSReqCallback 对象的 oncomplete 函数。所以我们继续看 readfileAfterOpen。

```
1. function readfileAfterOpen(err, fd) {
```

```

2.   const context = this.context;
3.   // 打开出错则直接执行用户回调，传入 err
4.   if (err) {
5.     context.callback(err);
6.     return;
7.   }
8.   // 保存打开文件的 fd
9.   context.fd = fd;
10.  // 新建一个 FSReqCallback 对象管理下一个异步请求和回调
11.  const req = new FSReqCallback();
12.  req.oncomplete = readFileAfterStat;
13.  req.context = context;
14.  // 获取文件的元数据，拿到文件大小
15.  binding.fstat(fd, false, req);
16. }

```

拿到文件的元数据后，执行 `readFileAfterStat`，这段逻辑和同步的类似，根据元数据中记录的文件大小，分配一个 buffer 用于后续读取文件内容。然后执行读操作。

```

1. read() {
2.   let buffer;
3.   let offset;
4.   let length;
5.
6.   // 省略部分 buffer 处理的逻辑
7.   const req = new FSReqCallback();
8.   req.oncomplete = readFileAfterRead;
9.   req.context = this;
10.
11.   read(this.fd, buffer, offset, length, -1, req);
12. }

```

再次新建一个 `FSReqCallback` 对象管理异步读取操作和回调。我们看一下 C++ 层 `read` 函数的实现。

```

1. // 拿到 C++ 层的 FSReqCallback 对象
2. FSReqBase* req_wrap_async = GetReqWrap(env, args[5]);
3. // 异步调用 uv_fs_read
4. AsyncCall(env, req_wrap_async, args, "read", UTF8, AfterInteger, uv_
    fs_read, fd, &uvbuf, 1, pos);

```

`AsyncCall` 最后调用 `Libuv` 的 `uv_fs_read` 函数。我们看一下这个函数的关键逻辑。

```
1. do  {                                \
2.     if  (cb  !=  NULL)   {           \
3.         uv__req_register(loop,  req);  \
4.         uv__work_submit(loop,          \
5.                         &req->work_req,  \
6.                         UV__WORK_FAST_IO,  \
7.                         uv__fs_work,  \
8.                         uv__fs_done);  \
9.     return  0;                      \
10. }                                \
11. else  {                                \
12.     uv__fs_work(&req->work_req);  \
13.     return  req->result;            \
14. }                                \
15. }                                \
16. while  (0)
```

uv__work_submit 是给线程池提交一个任务，当子线程执行这个任务时，就会执行 uv__fs_work，uv__fs_work 会调用操作系统的系统调用 read，可能会导致阻塞。等到读取成功后执行 uv__fs_done。uv__fs_done 会执行 C++ 层的回调，从而执行 JS 层的回调。JS 层的回调是 readFileAfterRead，这里就不具体展开，readFileAfterRead 的逻辑是判断是否读取完毕，是的话执行用户回调，否则继续发起读取操作。

12.6 文件监听

文件监听是非常常用的功能，比如我们修改了文件后 webpack 重新打包代码或者 Node.js 服务重启，都用到了文件监听的功能，Node.js 提供了两套文件监听的机制。

12.6.1 基于轮询的文件监听机制

基于轮询机制的文件监听 API 是 watchFile。流程如图 12-3 所示。

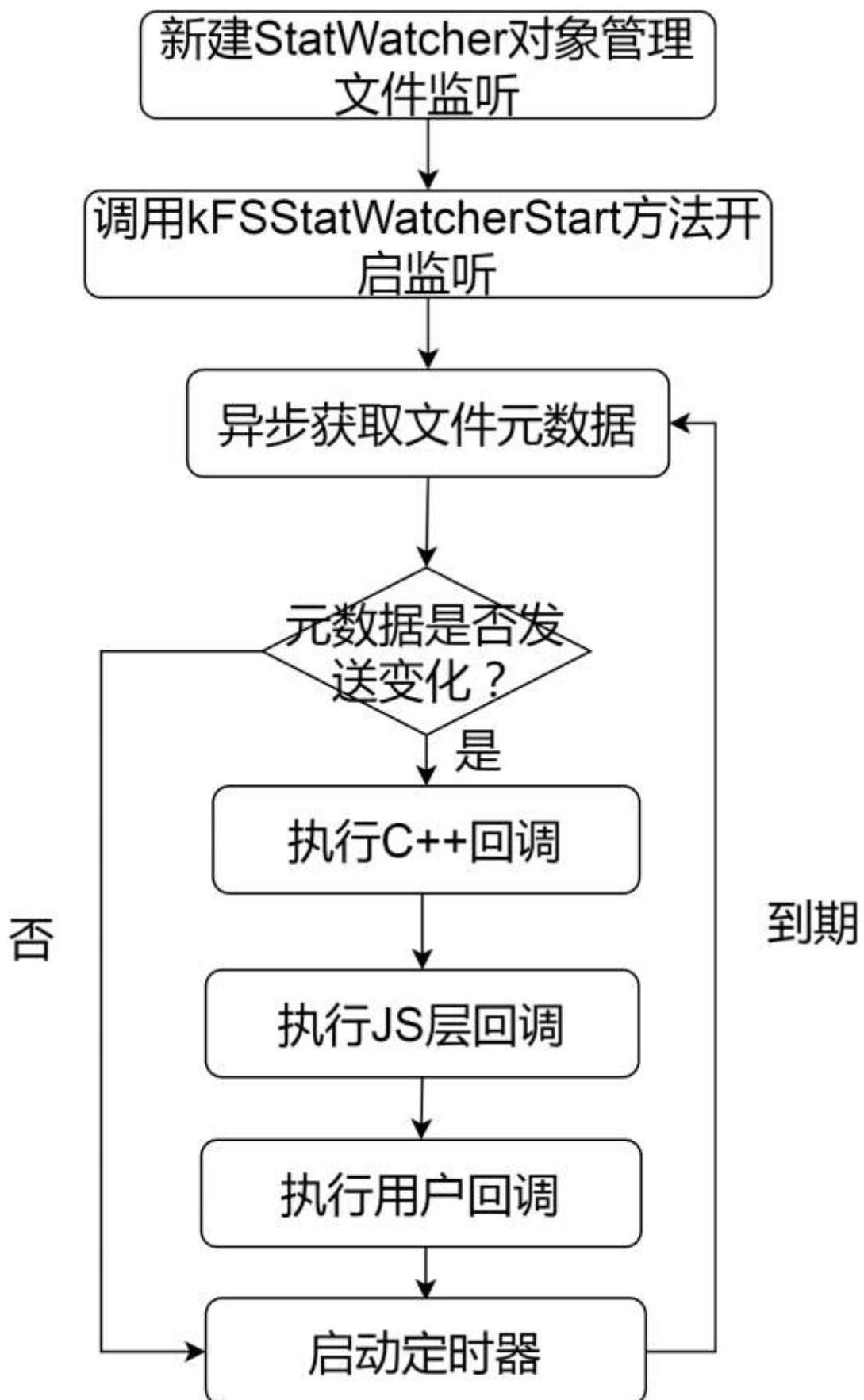


图 12-3

我们看一下具体实现。

```
1. function watchFile(filename, options, listener) {
2.   filename = getValidatedPath(filename);
3.   filename = pathModule.resolve(filename);
4.   let stat;
5.   // 省略部分参数处理逻辑
6.   options = {
7.     interval: 5007,
8.     // 一直轮询
9.     persistent: true,
10.    ...options
11.  };
12.
13.  // 缓存处理, filename 是否已经开启过监听
14.  stat = statWatchers.get(filename);
15.
16.  if (stat === undefined) {
17.    if (!watchers)
18.      watchers = require('internal/fs/watchers');
19.    stat = new watchers.StatWatcher(options.bigint);
20.    // 开启监听
21.    stat[watchers.kFSStatWatcherStart](filename,
22.                                         options.persistent,
23.                                         options.interval);
24.    // 更新缓存
25.    statWatchers.set(filename, stat);
26.  }
27.
28.  stat.addListener('change', listener);
29.  return stat;
30.}
```

StatWatcher 是管理文件监听的类，我们看一下 `watchers.kFSStatWatcherStart` 方法的实现。

```
1. StatWatcher.prototype[kFSStatWatcherStart] = function(filename, persist
ent, interval) {
2.   this._handle = new _StatWatcher(this[kUseBigint]);
3.   this._handle.onchange = onchange;
```

```

4.     filename = getValidatedPath(filename, 'filename');
5.     const err = this._handle.start(toNamespacedPath(filename),
6.                                     interval);
7. }
```

新建一个_StatWatcher 对象，_StatWatcher 是 C++ 模块提供的功能
`(node_stat_watcher.cc)`，然后执行它的 start 方法。Start 方法执行 Libuv 的
`uv_fs_poll_start` 开始监听文件。

```

1. int uv_fs_poll_start(uv_fs_poll_t* handle, uv_fs_poll_cb cb,
2. const char* path, unsigned int interval) {
3.     // 管理文件监听的数据结构
4.     struct poll_ctx* ctx;
5.     uv_loop_t* loop;
6.     size_t len;
7.     int err;
8.
9.     loop = handle->loop;
10.    len = strlen(path);
11.    // calloc 会把内存初始化为 0
12.    ctx = uv_calloc(1, sizeof(*ctx) + len);
13.    ctx->loop = loop;
14.    // C++ 层回调
15.    ctx->poll_cb = cb;
16.    // 多久轮询一次
17.    ctx->interval = interval ? interval : 1;
18.    ctx->start_time = uv_now(loop);
19.    // 关联的 handle
20.    ctx->parent_handle = handle;
21.    // 监听的文件路径
22.    memcpy(ctx->path, path, len + 1);
23.    // 初始化定时器结构体
24.    err = uv_timer_init(loop, &ctx->timer_handle);
25.    // 异步查询文件元数据
26.    err = uv_fs_stat(loop, &ctx->fs_req, ctx->path, poll_cb);
27.
28.    if (handle->poll_ctx != NULL)
29.        ctx->previous = handle->poll_ctx;
30.    // 关联负责管理轮询的对象
31.    handle->poll_ctx = ctx;
32.    uv_handle_start(handle);
33.    return 0;
```

| 34. }

Start 函数初始化一个 poll_ctx 结构体，用于管理文件监听，然后发起异步请求文件元数据的请求，获取元数据后，执行 poll_cb 回调。

```
1. static void poll_cb(uv_fs_t* req) {
2.     uv_stat_t* statbuf;
3.     struct poll_ctx* ctx;
4.     uint64_t interval;
5.     // 通过结构体字段获取结构体首地址
6.     ctx = container_of(req, struct poll_ctx, fs_req);
7.     statbuf = &req->statbuf;
8.     /*
9.      第一次不执行回调，因为没有可对比的元数据，第二次及后续的操作才可能
10.     执行回调，busy_polling 初始化的时候为 0，第一次执行的时候置
11.     busy_polling=1
12.     */
13.     if (ctx->busy_polling != 0)
14.         // 出错或者 stat 发生了变化则执行回调
15.         if (ctx->busy_polling < 0 ||
16.             !statbuf_eq(&ctx->statbuf, statbuf))
17.             ctx->poll_cb(ctx->parent_handle,
18.                         0,
19.                         &ctx->statbuf,
20.                         statbuf);
21.     // 保存当前获取到的 stat 信息，置 1
22.     ctx->statbuf = *statbuf;
23.     ctx->busy_polling = 1;
24.
25. out:
26.     uv_fs_req_cleanup(req);
27.
28.     if (ctx->parent_handle == NULL) {
29.         uv_close((uv_handle_t*)&ctx->timer_handle, timer_close_cb);
30.         return;
31.     }
32.     /*
33.      假设在开始时间点为 1，interval 为 10 的情况下执行了 stat，stat
34.      完成执行并执行 poll_cb 回调的时间点是 3，那么定时器的超时时间
35.      则为 10-3=7，即 7 个单位后就要触发超时，而不是 10，是因为 stat
36.      阻塞消耗了 3 个单位的时间，所以下次执行超时回调函数时说明从
37.      start 时间点开始算，已经经历了 x 单位各 interval，然后超时回调里
```

```

38.      又执行了 stat 函数，再到执行 stat 回调，这个时间点即 now=start+x
39.      单位个 interval+stat 消耗的时间。得出 now-start 为 interval 的
40.      x 倍+stat 消耗，即对 interval 取余可得到 stat 消耗，所以当前轮，
41.      定时器的超时时间为 interval - ((now-start) % interval)
42.  */
43.  interval = ctx->interval;
44.  interval = (uv_now(ctx->loop) - ctx->start_time) % interval;
45.
46.  if (uv_timer_start(&ctx->timer_handle, timer_cb, interval, 0))
47.      abort();
48. }

```

基于轮询的监听文件机制本质上是不断轮询文件的元数据，然后和上一次的元数据进行对比，如果有不一致的就认为文件变化了，因为第一次获取元数据时，还没有可以对比的数据，所以不认为是文件变化，这时候开启一个定时器。隔一段时间再去获取文件的元数据，如此反复，直到用户调 stop 函数停止这个行为。下面是 Libuv 关于文件变化的定义。

```

1. static int statbuf_eq(const uv_stat_t* a, const uv_stat_t* b) {
2.     return a->st_ctim.tv_nsec == b->st_ctim.tv_nsec
3.         && a->st_mtim.tv_nsec == b->st_mtim.tv_nsec
4.         && a->st_birthtim.tv_nsec == b->st_birthtim.tv_nsec
5.         && a->st_ctim.tv_sec == b->st_ctim.tv_sec
6.         && a->st_mtim.tv_sec == b->st_mtim.tv_sec
7.         && a->st_birthtim.tv_sec == b->st_birthtim.tv_sec
8.         && a->st_size == b->st_size
9.         && a->st_mode == b->st_mode
10.        && a->st_uid == b->st_uid
11.        && a->st_gid == b->st_gid
12.        && a->st_ino == b->st_ino
13.        && a->st_dev == b->st_dev
14.        && a->st_flags == b->st_flags
15.        && a->st_gen == b->st_gen;
16. }

```

12.6.2 基于 inotify 的文件监听机制

我们看到基于轮询的监听其实效率是很低的，因为需要我们不断去轮询文件的元数据，如果文件大部分时间里都没有变化，那就会白白浪费 CPU。如果文件改变了会主动通知我们那就好了，这就是基于 inotify 机制的文件监听。Node.js 提供的接口是 watch。watch 的实现和 watchFile 的比较类似。

```
1. function watch(filename, options, listener) {
2.     // Don't make changes directly on options object
3.     options = copyObject(options);
4.     // 是否持续监听
5.     if (options.persistent === undefined)
6.         options.persistent = true;
7.     // 如果是目录，是否监听所有子目录和文件的变化
8.     if (options.recursive === undefined)
9.         options.recursive = false;
10.    // 有些平台不支持
11.    if (options.recursive && !(isOSX || isWindows))
12.        throw new ERR FEATURE_UNAVAILABLE_ON_PLATFORM('watch recursive');
13.    if (!watchers)
14.        watchers = require('internal/fs/watchers');
15.    // 新建一个 FSWatcher 对象管理文件监听，然后开启监听
16.    const watcher = new watchers.FSWatcher();
17.    watcher[watchers.kFSWatchStart](filename,
18.                                    options.persistent,
19.                                    options.recursive,
20.                                    options.encoding);
21.
22.    if (listener) {
23.        watcher.addListener('change', listener);
24.    }
25.
26.    return watcher;
27.}
```

FSWatcher 函数是对 C++ 层 FSEvent 模块的封装。我们来看一下 start 函数的逻辑，start 函数透过 C++ 层调用了 Libuv 的 uv_fs_event_start 函数。在讲解 uv_fs_event_start 函数前，我们先了解一下 inotify 的原理和它在 Libuv 中的实现。inotify 是 Linux 系统提供用于监听文件系统的机制。inotify 机制的逻辑大致是

- 1 init_inotify 创建一个 inotify 的实例，返回一个文件描述符。类似 epoll。
- 2 inotify_add_watch 往 inotify 实例注册一个需监听的文件（inotify_rm_watch 是移除）。
- 3 read(inotify 实例对应的文件描述符，&buf, sizeof(buf))，如果没有事件触发，则阻塞（除非设置了非阻塞）。否则返回待读取的数据长度。buf 就是保存了触发事件的信息。

Libuv 在 inotify 机制的基础上做了一层封装。我们看一下 inotify 在 Libuv 的架构图如图 12-4 所示。

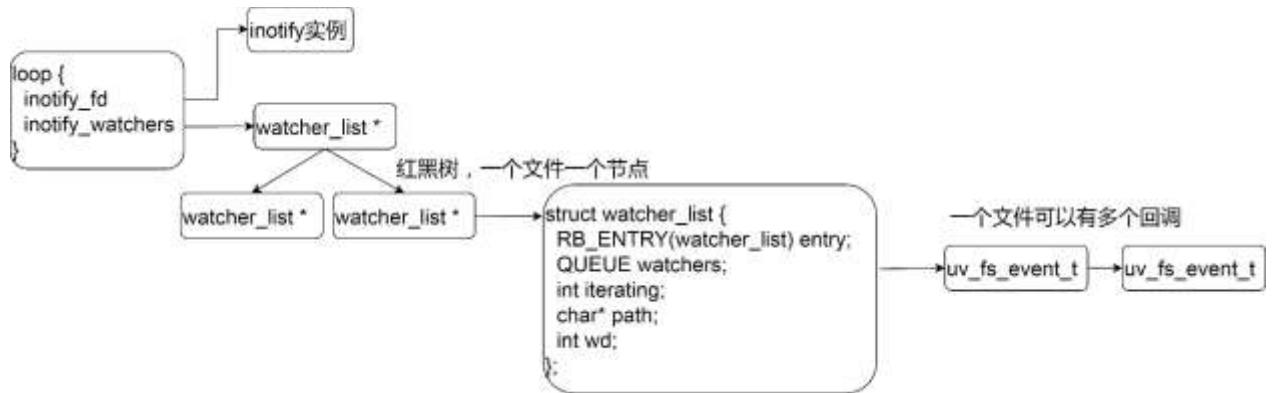


图 12-4

我们再来看一下 Libuv 中的实现。我们从一个使用例子开始。

```

1. int main(int argc, char **argv) {
2.     // 实现循环核心结构体 loop
3.     loop = uv_default_loop();
4.     uv_fs_event_t *fs_event_req = malloc(sizeof(uv_fs_event_t));
5.     // 初始化 fs_event_req 结构体的类型为 UV_FS_EVENT
6.     uv_fs_event_init(loop, fs_event_req);
7.     /*
8.         argv[argc] 是文件路径,
9.         uv_fs_event_start 向底层注册监听文件 argv[argc],
10.        cb 是事件触发时的回调
11.    */
12.    uv_fs_event_start(fs_event_req,
13.                      cb,
14.                      argv[argc],
15.                      UV_FS_EVENT_RECURSIVE);
16.    // 开启事件循环
17.    return uv_run(loop, UV_RUN_DEFAULT);
18. }
```

Libuv 在第一次监听文件的时候(调用 `uv_fs_event_start` 的时候), 会创建一个 `inotify` 实例。

```

1. static int init_inotify(uv_loop_t* loop) {
2.     int err;
3.     // 初始化过了则直接返回
4.     if (loop->inotify_fd != -1)
5.         return 0;
6.     /*
```

```
7.      调用操作系统的 inotify_init 函数申请一个 inotify 实例,
8.      并设置 UV_IN_NONBLOCK, UV_IN_CLOEXEC 标记
9.  */
10.     err = new_inotify_fd();
11.     if (err < 0)
12.         return err;
13.     // 记录 inotify 实例对应的文件描述符, 一个事件循环一个 inotify 实
   例
14.     loop->inotify_fd = err;
15.     /*
16.         inotify_read_watcher 是一个 IO 观察者,
17.         uv_io_init 设置 IO 观察者的文件描述符 (待观察的文件) 和回调
18.     */
19.     uv_io_init(&loop->inotify_read_watcher,
20.                uv_inotify_read,
21.                loop->inotify_fd);
22.     // 往 Libuv 中注册该 IO 观察者, 感兴趣的事件为可读
23.     uv_start(loop, &loop->inotify_read_watcher, POLLIN);
24.
25.     return 0;
26. }
```

Libuv 把 inotify 实例对应的 fd 通过 uv_io_start 注册到 epoll 中，当有文件变化的时候，就会执行回调 uv_inotify_read。分析完 Libuv 申请 inotify 实例的逻辑，我们回到 main 函数看看 uv_fs_event_start 函数。用户使用 uv_fs_event_start 函数来往 Libuv 注册一个待监听的文件。我们看看实现。

```
1. int uv_fs_event_start(uv_fs_event_t* handle,
2.                         uv_fs_event_cb cb,
3.                         const char* path,
4.                         unsigned int flags) {
5.     struct watcher_list* w;
6.     int events;
7.     int err;
8.     int wd;
9.
10.    if (uv_is_active(handle))
11.        return UV_EINVAL;
12.    // 申请一个 inotify 实例
13.    err = init_inotify(handle->loop);
14.    if (err)
15.        return err;
```

```

16. // 监听的事件
17. events = UV_IN_ATTRIB
18.           | UV_IN_CREATE
19.           | UV_IN_MODIFY
20.           | UV_IN_DELETE
21.           | UV_IN_DELETE_SELF
22.           | UV_IN_MOVE_SELF
23.           | UV_IN_MOVED_FROM
24.           | UV_IN_MOVED_TO;
25. // 调用操作系统的函数注册一个待监听的文件，返回一个对应于该文件的
   id
26. wd = uv_inotify_add_watch(handle->loop->inotify_fd, path, events);
27. if (wd == -1)
28.     return UV_ERR(errno);
29. // 判断该文件是不是已经注册过了
30. w = find_watcher(handle->loop, wd);
31. // 已经注册过则跳过插入的逻辑
32. if (w)
33.     goto no_insert;
34. // 还没有注册过则插入 Libuv 维护的红黑树
35. w = uv_malloc(sizeof(*w) + strlen(path) + 1);
36. if (w == NULL)
37.     return UV_ENOMEM;
38.
39. w->wd = wd;
40. w->path = strcpy((char*)(w + 1), path);
41. QUEUE_INIT(&w->watchers);
42. w->iterating = 0;
43. // 插入 Libuv 维护的红黑树, inotify_watchers 是根节点
44. RB_INSERT(watcher_root, CAST(&handle->loop->inotify_watchers), w);

45.
46. no_insert:
47. // 激活该 handle
48. uv_handle_start(handle);
49. // 同一个文件可能注册了很多个回调，w 对应一个文件，注册在用一个文件
   的回调排成队
50. QUEUE_INSERT_TAIL(&w->watchers, &handle->watchers);
51. // 保存信息和回调
52. handle->path = w->path;
53. handle->cb = cb;

```

```
54.     handle->wd = wd;
55.
56.     return 0;
57. }
```

下面我们逐步分析上面的函数逻辑。

1 如果是首次调用该函数则新建一个 inotify 实例。并且往 Libuv 插入一个观察者 io，Libuv 会在 Poll IO 阶段注册到 epoll 中。

2 往操作系统注册一个待监听的文件。返回一个 id。

3 Libuv 判断该 id 是不是在自己维护的红黑树中。不在红黑树中，则插入红黑树。返回一个红黑树中对应的节点。把本次请求的信息封装到 handle 中（回调时需要）。然后把 handle 插入刚才返回的节点的队列中。

这时候注册过程就完成了。Libuv 在 Poll IO 阶段如果检测到有文件发生变化，则会执行回调 uv_inotify_read。

```
1. static void uv_inotify_read(uv_loop_t* loop,
2.                               uv_io_t* dum
3.                               my,
4.                               unsigned int
5.                               events) {
6.     const struct uv_inotify_event* e;
7.     struct watcher_list* w;
8.     uv_fs_event_t* h;
9.     QUEUE queue;
10.    QUEUE* q;
11.    const char* path;
12.    ssize_t size;
13.    const char* p;
14.    /* needs to be large enough for sizeof(inotify_event) + str
15.       len(path) */
16.    char buf[4096];
17.    // 一次可能没有读完
18.    while (1) {
19.        do
20.            // 读取触发的事件信息, size 是数据大小, buffer 保存数据
21.            size = read(loop->inotify_fd, buf, sizeof(buf));
22.        while (size == -1 && errno == EINTR);
23.        // 没有数据可取了
24.        if (size == -1) {
25.            assert(errno == EAGAIN || errno == EWOULDBLOCK);
26.            break;
27.        }
28.    }
29. }
```

```

25.      // 处理 buffer 的信息
26.      for (p = buf; p < buf + size; p += sizeof(*e) + e->
   len) {
27.          // buffer 里是多个 uv_inotify_event 结构体，里面保存了事件
   信息和文件对应的 id (wd 字段)
28.          e = (const struct uv_inotify_event*)p;
29.
30.          events = 0;
31.          if (e->mask & (UV_IN_ATTRIB|UV_IN MODIFY))
32.              events |= UV_CHANGE;
33.          if (e->mask & ~(UV_IN_ATTRIB|UV_IN MODIFY))
34.              events |= UV_RENAME;
35.          // 通过文件对应的 id (wd 字段) 从红黑树中找到对应的节点
36.          w = find_watcher(loop, e->wd);
37.
38.          path = e->len ? (const char*)(e + 1) : uv_base_name_r(w->path);
39.          w->iterating = 1;
40.          // 把红黑树中，wd 对应节点的 handle 队列移到 queue 变量，准备
   处理
41.          QUEUE_MOVE(&w->watchers, &queue);
42.          while (!QUEUE_EMPTY(&queue)) {
43.              // 头结点
44.              q = QUEUE_HEAD(&queue);
45.              // 通过结构体偏移拿到首地址
46.              h = QUEUE_DATA(q, uv_fs_event_t, watchers);
47.              // 从处理队列中移除
48.              QUEUE_REMOVE(q);
49.              // 放回原队列
50.              QUEUE_INSERT_TAIL(&w->watchers, q);
51.              // 执行回调
52.              h->cb(h, path, events, 0);
53.          }
54.      }
55.  }
56. }
```

uv_inotify_read 函数的逻辑就是从操作系统中把数据读取出来，这些数据中保存了哪些文件触发了用户感兴趣的事件。然后遍历每个触发了事件的文件。从红黑树中找到该文件对应的红黑树节点。再取出红黑树节点中维护的一个 handle 队列，最后执行 handle 队列中每个节点的回调。

12.7 Promise 化 API

Node.js 的 API 都是遵循 callback 模式的，比如我们要读取一个文件的内容。我们通常会这样写

```
1. const fs = require('fs');
2. fs.readFile('filename', 'utf-8', (err, data) => {
3.   console.log(data)
4. })
```

为了支持 Promise 模式，我们通常这样写

```
1. const fs = require('fs');
2. function readFile(filename) {
3.   return new Promise((resolve, reject) => {
4.     fs.readFile(filename, 'utf-8', (err, data) => {
5.       err ? reject(err) : resolve(data);
6.     });
7.   });
8. }
```

但是在 Node.js V14 中，文件模块支持了 Promise 化的 api。我们可以直接使用 await 进行文件操作。我们看一下使用例子。

```
1. const { open, readFile } = require('fs').promises;
2. async function runDemo() {
3.   try {
4.     console.log(await readFile('11111.md', { encoding: 'utf-
5.     8' }));
6.   } catch (e){
7.   }
8. }
9. runDemo();
```

从例子中我们看到，和之前的 API 调用方式类似，不同的地方在于我们不用再写回调了，而是通过 await 的方式接收结果。这只是新版 API 的特性之一。在新版 API 之前，文件模块大部分 API 都是类似工具函数，比如 readFile, writeFile，新版 API 中支持面向对象的调用方式。

```
1. const { open, readFile } = require('fs').promises;
2. async function runDemo() {
3.   let filehandle;
```

```

4.  try {
5.      filehandle = await open('filename', 'r');
6.      // console.log(await readFile(filehandle, { encoding: 'utf-
8' }));
7.      console.log(await filehandle.readFile({ encoding: 'utf-
8' }));
8.  } finally {
9.      if (filehandle) {
10.         await filehandle.close();
11.     }
12.   }
13. }
14. runDemo();

```

面向对象的模式中，我们首先需要通过 open 函数拿到一个 FileHandle 对象（对文件描述符的封装），然后就可以在该对象上调各种文件操作的函数。在使用面向对象模式的 API 时有一个需要注意的地方是 Node.js 不会为我们关闭文件描述符，即使文件操作出错，所以我们需要自己手动关闭文件描述符，否则会造成文件描述符泄漏，而在非面向对象模式中，在文件操作完毕后，不管成功还是失败，Node.js 都会为我们关闭文件描述符。下面我们看一下具体的实现。首先介绍一个 FileHandle 类。该类是对文件描述符的封装，提供了面向对象的 API。

```

1. class FileHandle {
2.   constructor(filehandle) {
3.     // filehandle 为 C++ 对象
4.     this[kHandle] = filehandle;
5.     this[kFd] = filehandle.fd;
6.   }
7.
8.   get fd() {
9.     return this[kFd];
10.  }
11.
12.  readFile(options) {
13.    return readFile(this, options);
14.  }
15.
16.  close = () => {
17.    this[kFd] = -1;
18.    return this[kHandle].close();
19.  }
20.  // 省略部分操作文件的 api
21. }

```

`FileHandle` 的逻辑比较简单，首先封装了一系列文件操作的 API，然后实现了 `close` 函数用于关闭底层的文件描述符。

1 操作文件系统 API

这里我们以 `readFile` 为例进行分析

```
1. async function readFile(path, options) {  
2.   options = getOptions(options, { flag: 'r' });  
3.   const flag = options.flag || 'r';  
4.   // 以面向对象的方式使用，这时候需要自己关闭文件描述符  
5.   if (path instanceof FileHandle)  
6.     return readFileHandle(path, options);  
7.   // 直接调用，首先需要先打开文件描述符，读取完毕后 Node.js 会主动关闭文  
件描述符  
8.   const fd = await open(path, flag, 0o666);  
9.   return readFileHandle(fd, options).finally(fd.close);  
10. }
```

从 `readFile` 代码中我们看到不同调用方式下，`Node.js` 的处理是不一样的，当 `FileHandle` 是我们维护时，关闭操作也是我们负责执行，当 `FileHandle` 是 `Node.js` 维护时，`Node.js` 在文件操作完毕后，不管成功还是失败都会主动关闭文件描述符。接着我们看到 `readFileHandle` 的实现。

```
1. async function readFileHandle(filehandle, options) {  
2.   // 获取文件元信息  
3.   const statFields = await binding.fstat(filehandle.fd, false, kU  
sePromises);  
4.  
5.   let size;  
6.   // 是不是普通文件，根据文件类型获取对应大小  
7.   if ((statFields[1/* mode */] & S_IFMT) === S_IFREG) {  
8.     size = statFields[8/* size */];  
9.   } else {  
10.     size = 0;  
11.   }  
12.   // 太大了  
13.   if (size > kIoMaxLength)  
14.     throw new ERR_FS_FILE_TOO_LARGE(size);  
15.  
16.   const chunks = [];  
17.   // 计算每次读取的大小  
18.   const chunkSize = size === 0 ?  
19.     kReadFileMaxChunkSize :
```

```

20.     MathMin(size, kReadFileMaxChunkSize);
21.     let endOfFile = false;
22.     do {
23.         // 分配内存承载数据
24.         const buf = Buffer.alloc(chunkSize);
25.         // 读取的数据和大小
26.         const { bytesRead, buffer } =
27.             await read(filehandle, buf, 0, chunkSize, -1);
28.         // 是否读完了
29.         endOfFile = bytesRead === 0;
30.         // 读取了有效数据则把有效数据部分存起来
31.         if (bytesRead > 0)
32.             chunks.push(buffer.slice(0, bytesRead));
33.     } while (!endOfFile);
34.
35.     const result = Buffer.concat(chunks);
36.     if (options.encoding) {
37.         return result.toString(options.encoding);
38.     } else {
39.         return result;
40.     }
41. }
```

接着我们看 `read` 函数的实现

```

1. async function read(handle, buffer, offset, length, position) {
2.     // ...
3.     const bytesRead = (await binding.read(handle.fd, buffer, offset
4.         , length, position, kUsePromises)) || 0;
5.     return { bytesRead, buffer };
6. }
```

`Read` 最终执行了 `node_file.cc` 的 `Read`。我们看一下 `Read` 函数的关键代码。

```

1. static void Read(const FunctionCallbackInfo<Value>& args) {
2.     Environment* env = Environment::GetCurrent(args);
3.     // ...
4.     FSReqBase* req_wrap_async = GetReqWrap(env, args[5]);
5.     // 异步执行，有两种情况
6.     if (req_wrap_async != nullptr) {
7.         AsyncCall(env, req_wrap_async, args, "read", UTF8, AfterInteg
8.             er,
9.                 uv_fs_read, fd, &uvbuf, 1, pos);
10.    } else {
11.        // 同步执行，比如 fs.readFileSync
```

```
11.     CHECK_EQ(argc, 7);
12.     FSReqWrapSync req_wrap_sync;
13.     FS_SYNC_TRACE_BEGIN(read);
14.     const int bytesRead = SyncCall(env, args[6], &req_wrap_sync,
15.                                     "read",
16.                                     uv_fs_read, fd, &uvbuf, 1, po
17.                                     s);
18.     FS_SYNC_TRACE_END(read, "bytesRead", bytesRead);
19.     args.GetReturnValue().Set(bytesRead);
20. }
```

`Read` 函数分为三种情况，同步和异步，其中异步又分为两种，`callback` 模式和 `Promise` 模式。我们看一下异步模式的实现。我们首先看一下这句代码。

```
1. FSReqBase* req_wrap_async = GetReqWrap(env, args[5]);
```

`GetReqWrap` 根据第六个参数获取对应的值。

```
1. FSReqBase* GetReqWrap(Environment* env, v8::Local<v8::Value> valu
   e,
                           bool use_bigint) {
2.     // 是对象说明是继承 FSReqBase 的对象, 比如 FSReqCallback (异步模
   式)
3.     if (value->IsObject()) {
4.         return Unwrap<FSReqBase>(value.As<v8::Object>());
5.     } else if (value->StrictEquals(env->fs_use_promises_symbol()))
6.     {
7.         // Promise 模式 (异步模式)
8.         if (use_bigint) {
9.             return FSReqPromise<AliasedBigUint64Array>::New(env, use_bi
   gint);
10.        } else {
11.            return FSReqPromise<AliasedFloat64Array>::New(env, use_big
   int);
12.        }
13.    }
14.    // 同步模式
15.    return nullptr;
16. }
```

这里我们只关注 `Promise` 模式。所以 `GetReqWrap` 返回的是一个 `FSReqPromise` 对象，我们回到 `Read` 函数。看到以下代码

```

1. FSReqBase* req_wrap_async = GetReqWrap(env, args[5]);
2. AsyncCall(env, req_wrap_async, args, "read", UTF8, AfterInteger,
3.           uv_fs_read, fd, &uvbuf, 1, pos);

```

继续看 AsyncCall 函数 (node_file-inl.h)

```

1. template <typename Func, typename... Args>
2. FSReqBase* AsyncCall(Environment* env,
3.                       FSReqBase* req_wrap,
4.                       const v8::FunctionCallbackInfo<v8::Value>& a
    rgs,
5.                       const char* syscall, enum encoding enc,
6.                       uv_fs_cb after, Func fn, Args... fn_args) {
7.
8.     return AsyncDestCall(env, req_wrap, args,
9.                           syscall, nullptr, 0, enc,
10.                          after, fn, fn_args...);
10. }

```

AsyncCall 是对 AsyncDestCall 的封装

```

1. template <typename Func, typename... Args>
2. FSReqBase* AsyncDestCall(Environment* env, FSReqBase* req_wrap,
3.                           const v8::FunctionCallbackInfo<v8::Value
    >& args,
4.                           const char* syscall, const char* dest,
5.                           size_t len, enum encoding enc, uv_fs_cb
    after,
6.                           Func fn, Args... fn_args) {
7.     CHECK_NOT_NULL(req_wrap);
8.     req_wrap->Init(syscall, dest, len, enc);
9.     // 调用 libuv 函数
10.    int err = req_wrap->Dispatch(fn, fn_args..., after);
11.    // 失败则直接执行回调，否则返回一个 Promise，见 SetReturnValue 函
    数
12.    if (err < 0) {
13.        uv_fs_t* uv_req = req_wrap->req();
14.        uv_req->result = err;
15.        uv_req->path = nullptr;
16.        after(uv_req); // after may delete req_wrap if there is an
    error
17.        req_wrap = nullptr;
18.    } else {
19.        req_wrap->SetReturnValue(args);

```

```
20. }
21.
22. return req_wrap;
23. }
```

AsyncDestCall 函数主要做了两个操作，首先通过 Dispatch 调用底层 Libuv 的函数，比如这里是 uv_fs_read。如果出错执行回调返回错误，否则执行 req_wrap->SetReturnValue(args)。我们知道 req_wrap 是在 GetReqWrap 函数中由 FSReqPromise<AliasedBigUint64Array>::New(env, use_bigint) 创建。

```
1. template <typename AliasedBufferT>
2. FSReqPromise<AliasedBufferT>*
3. FSReqPromise<AliasedBufferT>::New(Environment* env, bool use_bigint) {
4.     v8::Local<v8::Object> obj;
5.     // 创建一个 C++ 对象存到 obj 中
6.     if (!env->fsreqpromise_constructor_template()
7.         ->NewInstance(env->context()))
8.         .ToLocal(&obj)) {
9.     return nullptr;
10. }
11. // 设置一个 promise 属性，值是一个 Promise::Resolver
12. v8::Local<v8::Promise::Resolver> resolver;
13. if (!v8::Promise::Resolver::New(env->context()).ToLocal(&resolver) ||
14.     obj->Set(env->context(), env->promise_string(), resolver).
15.     IsNothing())) {
16.     return nullptr;
17. }
18. // 返回另一个 C++ 对象，里面保存了 obj，obj 也保存了指向 FSReqPromise
19. // 对象的指针
20. return new FSReqPromise(env, obj, use_bigint);
21. }
```

所以 req_wrap 是一个 FSReqPromise 对象。我们看一下 FSReqPromise 对象的 SetReturnValue 方法。

```
1. template <typename AliasedBufferT>
2. void FSReqPromise<AliasedBufferT>::SetReturnValue(
3.     const v8::FunctionCallbackInfo<v8::Value>& args) {
4.     // 拿到 Promise::Resolver 对象
5.     v8::Local<v8::Value> val =
6.         object()->Get(env()->context(),
7.                         env()->promise_string()).ToLocalChecked();
```

```

8.   v8::Local<v8::Promise::Resolver> resolver = val.As<v8::Promise::Resolver>();
9.   // 拿到一个 Promise 作为返回值, 即 JS 层拿到的值
10.  args.GetReturnValue().Set(resolver->GetPromise());
11. }
```

至此我们看到了新版 API 实现的核心逻辑, 正是这个 `Promise` 返回值。通过层层返回后, 在 JS 层就拿到这个 `Promise`, 然后处于 `pending` 状态等待决议。我们继续看一下 `Promise` 决议的逻辑。在分析 `Read` 函数中我们看到执行 `Libuv` 的 `uv_fs_read` 函数时, 设置的回调是 `AfterInteger`。那么当读取文件成功后就会执行该函数。所以我们看看该函数的逻辑。

```

1. void AfterInteger(uv_fs_t* req) {
2.   // 通过属性拿到对象的地址
3.   FSReqBase* req_wrap = FSReqBase::from_req(req);
4.   FSReqAfterScope after(req_wrap, req);
5.
6.   if (after.Proceed())
7.     req_wrap->Resolve(Integer::New(req_wrap->env()->isolate(), re
      q->result));
8. }
```

接着我们看一下 `Resolve`

```

1. template <typename AliasedBufferT>
2. void FSReqPromise<AliasedBufferT>::Resolve(v8::Local<v8::Value> v
  alue) {
3.   finished_ = true;
4.   v8::HandleScope scope(env()->isolate());
5.   InternalCallbackScope callback_scope(this);
6.   // 拿到保存的 Promise 对象, 修改状态为 resolve, 并设置结果
7.   v8::Local<v8::Value> val =
8.     object()->Get(env()->context(),
9.                   env()->promise_string()).ToLocalChecked();
10.  v8::Local<v8::Promise::Resolver> resolver = val.As<v8::Promise::Resolver>();
11.  USE(resolver->Resolve(env()->context(), value).FromJust());
12. }
```

`Resolve` 函数修改 `Promise` 的状态和设置返回值, 从而 JS 层拿到这个决议的值。回到 `fs` 层

```

1. const bytesRead = (await binding.read(handle.fd,
2.                      buffer,
```

```
3.                     offset,
4.                     length,
5.                     position, kUsePromises))|0;
```

我们就拿到了返回值。

12.8 流式 API

前面分析了 Node.js 中文件模块的多种文件操作的方式，不管是同步、异步还是 Promise 化的 API，它们都有一个问题就是对于用户来说，文件操作都是一次性完成的，比如我们调用 `readFile` 读取一个文件时，Node.js 会通过一次或多次调用操作系统的接口把所有的文件内容读到内存中，同样我们调用 `writeFile` 写一个文件时，Node.js 会通过一次或多次调用操作系统接口把用户的数据写入硬盘，这对内存来说是非常有压力的。假设我们有这样的一个场景，我们需要读取一个文件的内容，然后返回给前端，如果我们直接读取整个文件内容，然后再执行写操作这无疑是非常消耗内存，也是非常低效的。

```
1. const http = require('http');
2. const fs = require('fs');
3. const server = http.createServer((req, res) => {
4.   fs.readFile('11111.md', (err, data) => {
5.     res.end(data);
6.   })
7. }).listen(11111);
```

这时候我们需要使用流式的 API。

```
1. const http = require('http');
2. const fs = require('fs');
3. const server = http.createServer((req, res) => {
4.   fs.createReadStream('11111.md').pipe(res);
5. }).listen(11111);
```

流式 API 的好处在于文件的内容并不是一次性读取到内存的，而是部分读取，消费完后再继续读取。Node.js 内部帮我们做了流量的控制，如图 12-5 所示。

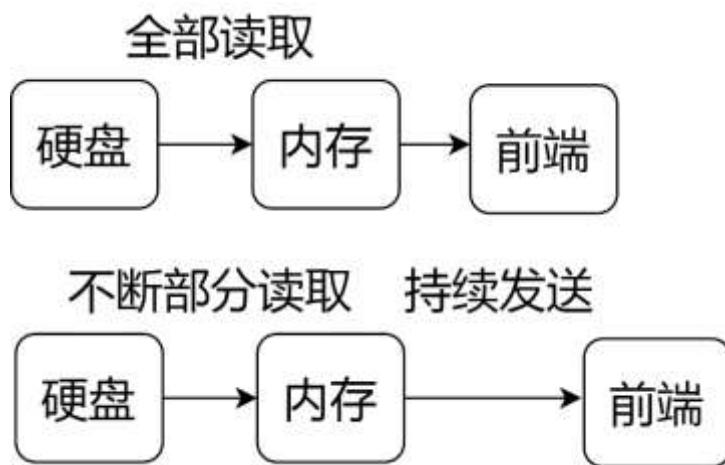


图 12-5

下面我们看一下 Node.js 流式 API 的具体实现。

12.8.1 可读文件流

可读文件流是对文件进行流式读取的抽象。我们可以通过 `fs.createReadStream` 创建一个文件可读流。文件可读流继承于可读流，所以我们可以以可读流的方式使用它。

```

1. const fs = require('fs');
2. const { Writable } = require('stream');
3. class DemoWritable extends Writable {
4.   _write(data, encoding, cb) {
5.     console.log(data);
6.     cb(null);
7.   }
8. }
9. fs.createReadStream('11111.md').pipe(new DemoWritable);
  
```

或者

```

1. const fs = require('fs');
2. const readStream = fs.createReadStream('11111.md');
3. readStream.on('data', (data) => {
4.   console.log(data)
5. });
  
```

我们看一下 `createReadStream` 的实现。

```

1. fs.createReadStream = function(path, options) {
2.   return new ReadStream(path, options);
  
```

| 3. };

CreateReadStream 是对 ReadStream 的封装。

```
1. function ReadStream(path, options) {
2.   if (!(this instanceof ReadStream))
3.     return new ReadStream(path, options);
4.
5.   options = copyObject(getOptions(options, {}));
6.   // 可读流的阈值
7.   if (options.highWaterMark === undefined)
8.     options.highWaterMark = 64 * 1024;
9.
10.  Readable.call(this, options);
11.
12.  handleError((this.path = getPathFromURL(path)));
13.  // 支持传文件路径或文件描述符
14.  this.fd = options.fd === undefined ? null : options.fd;
15.  this.flags = options.flags === undefined ? 'r' : options.flags
16. ;
17.  this.mode = options.mode === undefined ? 0o666 : options.mode;
18.
19.  // 读取的开始和结束位置
20.  this.start = typeof this.fd !== 'number' && options.start ===
undefined ?
21.    0 : options.start;
22.  this.end = options.end;
23.  // 流出错或结束时是否自动销毁流
24.  this.autoClose = options.autoClose === undefined ? true : opti
ons.autoClose;
25.  this.pos = undefined;
26.  // 已读的字节数
27.  this.bytesRead = 0;
28.  // 流是否已经关闭
29.  this.closed = false;
30.  // 参数校验
31.  if (this.start !== undefined) {
32.    if (typeof this.start !== 'number') {
33.      throw new errors.TypeError('ERR_INVALID_ARG_TYPE',
34.                                'start',
35.                                'number',
36.                                this.start);
37.    }
38.    // 默认读取全部内容
39.    if (this.end === undefined) {
```

```

38.     this.end = Infinity;
39. } else if (typeof this.end !== 'number') {
40.     throw new errors.TypeError('ERR_INVALID_ARG_TYPE',
41.                               'end',
42.                               'number',
43.                               this.end);
44. }
45.
46. // 从文件的哪个位置开始读, start 是开始位置, pos 是当前位置, 初始化等于开始位置
47. this.pos = this.start;
48. }
49. // 如果是根据一个文件名创建一个流, 则首先打开这个文件
50. if (typeof this.fd !== 'number')
51.     this.open();
52.
53. this.on('end', function() {
54.     // 流结束时自动销毁流
55.     if (this.autoClose) {
56.         this.destroy();
57.     }
58. });
59. }

```

ReadStream 初始化完后做了两个操作，首先调用 open 打开文件（如果需要的话），接着监听流结束事件，用户可以设置 autoClose 选项控制当流结束或者出错时是否销毁流，对于文件流来说，销毁流意味着关闭本地文件描述符。我们接着看一下 open 的实现

```

1. // 打开文件
2. ReadStream.prototype.open = function() {
3.     var self = this;
4.     fs.open(this.path, this.flags, this.mode, function(er, fd) {
5.         if (er) {
6.             // 发生错误, 是否需要自动销毁流
7.             if (self.autoClose) {
8.                 self.destroy();
9.             }
10.            // 通知用户
11.            self.emit('error', er);
12.            return;
13.        }
14.
15.        self.fd = fd;
16.        // 触发 open, 一般用于 Node.js 内部逻辑
17.        self.emit('open', fd);

```

```
18.    // start the flow of data.  
19.    // 打开成功后开始流式读取文件内容  
20.    self.read();  
21.  });  
22. };
```

open 函数首先打开文件，打开成功后开启流式读取。从而文件内容就会源源不断地流向目的流。我们继续看一下读取操作的实现。

```
1. // 实现可读流的钩子函数  
2. ReadStream.prototype._read = function(n) {  
3.    // 如果没有调用 open 而是直接调用该方法则先执行 open  
4.    if (typeof this.fd !== 'number') {  
5.        return this.once('open', function() {  
6.            this._read(n);  
7.        });  
8.    }  
9.    // 流已经销毁则不处理  
10.   if (this.destroyed)  
11.     return;  
12.   // 判断池子空间是否足够，不够则申请新的  
13.   if (!pool || pool.length - pool.used < kMinPoolSpace) {  
14.       // discard the old pool.  
15.       allocNewPool(this.readableHighWaterMark);  
16.   }  
17.  
18.   // 计算可读的最大数量  
19.   var thisPool = pool;  
20.   /*  
21.      可读取的最大值，取可用内存大小和 Node.js 打算读取的大小  
22.      中的小值，n 不是用户想读取的大小，而是可读流内部的逻辑  
23.      见_stream_readable.js 的 this._read(state.highWaterMark)  
24.   */  
25.   var toRead = Math.min(pool.length - pool.used, n);  
26.   var start = pool.used;  
27.   // 已经读取了部分了，则计算剩下读取的大小，和计算读取的 toRead 比较取  
28.   if (this.pos !== undefined)  
29.       toRead = Math.min(this.end - this.pos + 1, toRead);  
30.  
31.   // 读结束  
32.   if (toRead <= 0)  
33.       return this.push(null);  
34.
```

```

35. // pool.used 是即将读取的数据存储在 pool 中的开始位置, this.pos 是从
   文件的那个位置开始读取
36. fs.read(this.fd, pool, pool.used, toRead, this.pos, (er, bytes
   Read) => {
37.   if (er) {
38.     if (this.autoClose) {
39.       this.destroy();
40.     }
41.     this.emit('error', er);
42.   } else {
43.     var b = null;
44.     if (bytesRead > 0) {
45.       // 已读的字节数累加
46.       this.bytesRead += bytesRead;
47.       // 获取有效数据
48.       b = thisPool.slice(start, start + bytesRead);
49.     }
50.     // push 到底层流的 bufferList 中, 底层的 push 会触发 data 事件
51.     this.push(b);
52.   }
53. });
54.
55. // 重新设置已读指针的位置
56. if (this.pos !== undefined)
57.   this.pos += toRead;
58. pool.used += toRead;
59. };

```

代码看起来很多，主要的逻辑是调用异步 read 函数读取文件的内容，然后放到可读流中，可读流会触发 data 事件通知用户有数据到来，然后继续执行 read 函数，从而不断驱动着数据的读取（可读流会根据当前情况判断是否继续执行 read 函数，以达到流量控制的目的）。最后我们看一下关闭和销毁一个文件流的实现。

```

1. ReadStream.prototype.close = function(cb) {
2.   this.destroy(null, cb);
3. };

```

当我们设置 autoClose 为 false 的时候，我们就需要自己手动调用 close 函数关闭可读文件流。关闭文件流很简单，就是正常地销毁流。我们看看销毁流的时候，Node.js 做了什么。

```

1. // 关闭底层文件
2. ReadStream.prototype._destroy = function(err, cb) {
3.   const isOpen = typeof this.fd !== 'number';
4.   if (isOpen) {

```

```
5.     this.once('open', closeFsStream.bind(null, this, cb, err));
6.     return;
7.   }
8.
9.   closeFsStream(this, cb);
10.  this.fd = null;
11. };
12.
13. function closeFsStream(stream, cb, err) {
14.   fs.close(stream.fd, (er) => {
15.     er = er || err;
16.     cb(er);
17.     stream.closed = true;
18.     if (!er)
19.       stream.emit('close');
20.   });
21. }
```

销毁文件流就是关闭底层的文件描述符。另外如果是因为发生错误导致销毁或者关闭文件描述符错误则不会触发 close 事件。

12.8.2 可写文件流

可写文件流是对文件进行流式写入的抽象。我们可以通过 `fs.createWriteStream` 创建一个文件可写流。文件可些流继承于可写流，所以我们可以以可写流的方式使用它。

```
1. const fs = require('fs');
2. const writeStream = fs.createWriteStream('123.md');
3. writeStream.end('world');
```

或者

```
1. const fs = require('fs');
2. const { Readable } = require('stream');
3.
4. class DemoReadStream extends Readable {
5.   constructor() {
6.     super();
7.     this.i = 0;
8.   }
9.   _read(n) {
10.     this.i++;
11.     if (this.i > 10) {
12.       this.push(null);
13.     } else {
```

```

14.         this.push('1'.repeat(n));
15.     }
16.
17. }
18. }
19. new DemoReadStream().pipe(fs.createWriteStream('123.md'));

```

我们看一下 `createWriteStream` 的实现。

```

1. fs.createWriteStream = function(path, options) {
2.   return new WriteStream(path, options);
3. };

```

`createWriteStream` 是对 `WriteStream` 的封装，我们看一下 `WriteStream` 的实现

```

1. function WriteStream(path, options) {
2.   if (!(this instanceof WriteStream))
3.     return new WriteStream(path, options);
4.   options = copyObject(getOptions(options, {}));
5.
6.   Writable.call(this, options);
7.
8.   handleError((this.path = getPathFromURL(path)));
9.   this.fd = options.fd === undefined ? null : options.fd;
10.  this.flags = options.flags === undefined ? 'w' : options.flags
11. ;
12.  this.mode = options.mode === undefined ? 0o666 : options.mode;
13.
14.  // 写入的开始位置
15.  this.start = options.start;
16.  // 流结束和触发错误的时候是否销毁流
17.  this.autoClose = options.autoClose === undefined ? true : !options.autoClose;
18.  // 当前写入位置
19.  this.pos = undefined;
20.  // 写成功的字节数
21.  this.bytesWritten = 0;
22.  this.closed = false;
23.
24.  if (this.start !== undefined) {
25.    if (typeof this.start !== 'number') {
26.      throw new errors.TypeError('ERR_INVALID_ARG_TYPE',
27.                                'start',
28.                                'number',
29.                                this.start);

```

```
28.      }
29.      if (this.start < 0) {
30.          const errVal = `start: ${this.start}`;
31.          throw new errors.RangeError('ERR_OUT_OF_RANGE',
32.                                         'start',
33.                                         '>= 0',
34.                                         errVal);
35.      }
36.      // 记录写入的开始位置
37.      this.pos = this.start;
38.  }
39.
40.  if (options.encoding)
41.    this.setDefaultEncoding(options.encoding);
42.  // 没有传文件描述符则打开一个新的文件
43.  if (typeof this.fd !== 'number')
44.    this.open();
45.
46.  // 监听可写流的 finish 事件，判断是否需要执行销毁操作
47.  this.once('finish', function() {
48.    if (this.autoClose) {
49.      this.destroy();
50.    }
51.  });
52. }
```

`WriteStream` 初始化了一系列字段后，如果传的是文件路径则打开文件，如果传的文件描述符则不需要再次打开文件。后续对文件可写流的操作就是对文件描述符的操作。我们首先看一下写入文件的逻辑。我们知道可写流只是实现了一些抽象的逻辑，具体的写逻辑是具体的流通过`_write`或者`_writev`实现的，我们看一下`_write`的实现。

```
1. WriteStream.prototype._write = function(data, encoding, cb) {
2.   if (!(data instanceof Buffer)) {
3.     const err = new errors.TypeError('ERR_INVALID_ARG_TYPE',
4.                                     'data',
5.                                     'Buffer',
6.                                     data);
7.     return this.emit('error', err);
8.   }
9.   // 还没打开文件，则等待打开成功后再执行写操作
10.  if (typeof this.fd !== 'number') {
11.    return this.once('open', function() {
12.      this._write(data, encoding, cb);
13.    });
14.  }
```

```

15. // 执行写操作, 0 代表从 data 的哪个位置开始写, 这里是全部写入, 所以是
16. fs.write(this.fd, data, 0, data.length, this.pos, (er, bytes)
=> {
17.   if (er) {
18.     if (this.autoClose) {
19.       this.destroy();
20.     }
21.     return cb(er);
22.   }
23.   // 写入成功的字节长度
24.   this.bytesWritten += bytes;
25.   cb();
26. });
27. // 下一个写入的位置
28. if (this.pos !== undefined)
29.   this.pos += data.length;
30. };

```

_write 就是根据用户传入数据的大小，不断调用 fs.write 往底层写入数据，直到写完成或者出错。接着我们看一下批量写的逻辑。

```

1. // 实现可写流批量写钩子
2. WriteStream.prototype._writev = function(data, cb) {
3.   if (typeof this.fd !== 'number') {
4.     return this.once('open', function() {
5.       this._writev(data, cb);
6.     });
7.   }
8.
9.   const self = this;
10.  const len = data.length;
11.  const chunks = new Array(len);
12.  var size = 0;
13.  // 计算待写入的总大小，并且把数据保存到 chunk 数组中，准备写入
14.  for (var i = 0; i < len; i++) {
15.    var chunk = data[i].chunk;
16.
17.    chunks[i] = chunk;
18.    size += chunk.length;
19.  }
20.  // 执行批量写
21.  writev(this.fd, chunks, this.pos, function(er, bytes) {
22.    if (er) {
23.      self.destroy();

```

```
24.     return cb(er);
25. }
26. // 写成功的字节数，可能小于希望写入的字节数
27. self.bytesWritten += bytes;
28. cb();
29. });
30. /*
31.   更新下一个写入位置，如果写部分成功，计算下一个写入位置时
32.   也会包括没写成功的字节数，所以是假设 size 而不是 bytes
33. */
34. if (this.pos !== undefined)
35.   this.pos += size;
36. };
```

批量写入的逻辑和_write类似，只不过它调用的是不同的接口往底层写。接下来我们看关闭文件可写流的实现。

```
1. WriteStream.prototype.close = function(cb) {
2.   // 关闭文件成功后执行的回调
3.   if (cb) {
4.     if (this.closed) {
5.       process.nextTick(cb);
6.       return;
7.     } else {
8.       this.on('close', cb);
9.     }
10.   }
11.
12. /*
13.   如果 autoClose 是 false，说明流结束触发 finish 事件时，不会销毁流，
14.   见 WriteStream 初始化代码 以这里需要监听 finish 事件，保证可写流结
束时可以关闭文件描述符
15. */
16. if (!this.autoClose) {
17.   this.on('finish', this.destroy.bind(this));
18. }
19.
20. // 结束流，会触发 finish 事件
21. this.end();
22. };
```

可写文件流和可读文件流不一样。默认情况下，可读流在读完文件内容后 Node.js 会自动销毁流（关闭文件描述符），而写入文件，在某些情况下 Node.js 是无法知道什么时候流结束的，这需要我们显式地通知 Node.js。在下面的例子中，我们是不需要显式通知 Node.js 的

```
1. fs.createReadStream('11111.md').pipe(fs.createWriteStream('123.md'));
```

因为可读文件流在文件读完后会调用可写文件的 end 方法，从而关闭可读流和可写流对应的文件描述符。而在以下代码中情况就变得复杂。

```
1. const stream = fs.createWriteStream('123.md');
2. stream.write('hello');
3. // stream.close 或 stream.end();
```

在默认情况，我们可以调用 end 或者 close 去通知 Node.js 流结束。但是如果我们设置了 autoClose 为 false，那么我们只能调用 close 而不能调用 end。否则会造成文件描述符泄漏。因为 end 只是关闭了流。但是没有触发销毁流的逻辑。而 close 会触发销毁流的逻辑。我们看一下具体的代码。

```
1. const fs = require('fs');
2. const stream = fs.createWriteStream('123.md');
3. stream.write('hello');
4. // 防止进程退出
5. setInterval(() => {});
```

以上代码会导致文件描述符泄漏，我们在 Linux 下执行以下代码，通过 ps aux 找到进程 id，然后执行 lsof -p pid 就可以看到进程打开的所有文件描述符。输出如 12-6 所示。

```
17w      REG    8,1          5  686565 /home/cyb/123.md
```

图 12-6

文件描述符 17 指向了 123.md 文件。所以文件描述符没有被关闭，引起文件描述符泄漏。我们修改一下代码。

```
1. const fs = require('fs');
2. const stream = fs.createWriteStream('123.md');
3. stream.end('hello');
4. setInterval(() => {});
```

下面是以上代码的输出，我们看到没有 123.md 对应的文件描述符，如图 12-7 所示。

```
15w      FIFO  0,12          0t0  548114 pipe
16u  a_inode  0,13            0    10832 [eventfd]
```

图 12-7

我们继续修改代码

```
1. const fs = require('fs');
2. const stream = fs.createWriteStream('123.md', {autoClose: false})
   ;
```

```
3. stream.end('hello');
4. setInterval(() => {});
```

以上代码的输出如图 12-8 所示。

```
17w      REG  8,1          5  686565 /home/cyb/123.md
```

图 12-8

我们看到使用 end 也无法关闭文件描述符。继续修改。

```
1. const fs = require('fs');
2. const stream = fs.createWriteStream('123.md', {autoClose: false})
3. stream.close();
4. setInterval(() => {});
```

以上代码的输出如图 12-9 所示。

```
15w      FIFO  0,12          0t0  548892 pipe
16u  a_inode  0,13          0    10832 [eventfd]
#
```

图 12-9

我们看到成功关闭了文件描述符。

第十三章 进程

进程是操作系统里非常重要的概念，也是不容易理解的概念，但是看起来很复杂的进程，其实在操作系统的代码里，也只是一些数据结构和算法，只不过它比一般的数据结构和算法更复杂。进程在操作系统里，是用一个 task_struct 结构体表示的。因为操作系统是大部分是用 C 语言实现的，没有对象这个概念。如果我们用 JS 来理解的话，每个进程就是一个对象，每次新建一个进程，就是新建一个对象。task_struct 结构体里保存了一个进程所需要的一些信息，包括执行状态、执行上下文、打开的文件、根目录、工作目录、收到的信号、信号处理函数、代码段、数据段的信息、进程 id、执行时间、退出码等等。本章将会介绍 Node.js 进程模块的原理和实现。

13.2 Node.js 主进程

当我们执行 node index.js 的时候，操作系统就会创建一个 Node.js 进程，我们的代码就是在该进程中执行。从代码角度来说，我们在 Node.js 中感知进程的方式是通过 process 对象。本节我们分析一下这个对象。

13.2.1 创建 process 对象

Node.js 启动的时候会执行以下代码创建 process 对象 (env.cc)。

```

1. Local<Object> process_object = node::CreateProcessObject(this).FromMaybe(Local<Object>());
2. set_process_object(process_object);

```

process 对象通过 CreateProcessObject 创建，然后保存到 env 对象中。我们看一下 CreateProcessObject。

```

1. MaybeLocal<Object> CreateProcessObject(Environment* env)  {
2.     Isolate* isolate = env->isolate();
3.     EscapableHandleScope scope(isolate);
4.     Local<Context> context = env->context();
5.
6.     Local<FunctionTemplate> process_template = FunctionTemplate::New(
    isolate);
7.     process_template->SetClassName(env->process_string());
8.     Local<Function> process_ctor;
9.     Local<Object> process;
10.    // 新建 process 对象
11.    if (!process_template->GetFunction(context).ToLocal(&process_ctor)
12.        || !process_ctor->NewInstance(context).ToLocal(&process))  {
13.        return MaybeLocal<Object>();
14.    }
15.    // 设置一系列属性，这是我们平时通过 process 对象访问的属性
16.    // Node.js 的版本
17.    READONLY_PROPERTY(process, "version",
18.                      FIXED_ONE_BYTE_STRING(isolate(),
19.                                              NODE_VERSION));
20.
21.    return scope.Escape(process);
22. }

```

这是使用 V8 创建一个对象的典型例子，并且设置了一些属性。Node.js 启动过程中，很多地方都会给 process 挂载属性。下面我们看我们常用的 process.env 是怎么挂载的。

13.2.2 挂载 env 属性

```

1. Local<String> env_string = FIXED_ONE_BYTE_STRING(isolate_, "env");
2. Local<Object> env_var_proxy;
3. // 设置 process 的 env 属性
4. if (!CreateEnvVarProxy(context(),
5.                         isolate_,
6.                         as_callback_data()))

```

```
7.     .ToLocal(&env_var_proxy) ||  
8.     process_object()->Set(context(),  
9.                             env_string,  
10.                            env_var_proxy).IsNothing()) {  
11.     return MaybeLocal<Value>();  
12. }
```

上面的代码通过 CreateEnvVarProxy 创建了一个对象，然后保存到 env_var_proxy 中，最后给 process 挂载了 env 属性。它的值是 CreateEnvVarProxy 创建的对象。

```
1. MaybeLocal<Object> CreateEnvVarProxy(Local<Context> context,  
2.                                         Isolate* isolate,  
3.                                         Local<Object> data) {  
4.     EscapableHandleScope scope(isolate);  
5.     Local<ObjectTemplate> env_proxy_template = ObjectTemplate::New(isolate);  
6.     env_proxy_template->SetHandler(NamedPropertyHandlerConfiguration(  
7.                                         EnvGetter,  
8.                                         EnvSetter,  
9.                                         EnvQuery,  
10.                                        EnvDeleter,  
11.                                        EnvEnumerator,  
12.                                        data,  
13.                                         PropertyHandlerFlags::kHasNoSideEffect));  
14.     return scope.EscapeMaybe(env_proxy_template->NewInstance(context));  
15. }
```

CreateEnvVarProxy 首先申请一个对象模板，然后设置通过该对象模板创建的对象的访问描述符。我们看一下 getter 描述符（EnvGetter）的实现，getter 描述符和我们在 JS 里使用的类似。

```
1. static void EnvGetter(Local<Name> property,  
2.                         const PropertyCallbackInfo<Value>& info) {  
3.     Environment* env = Environment::GetCurrent(info);  
4.     MaybeLocal<String> value_string = env->env_vars()->Get(env->isolate(), property.As<String>());  
5.     if (!value_string.IsEmpty()) {  
6.         info.GetReturnValue().Set(value_stringToLocalChecked());  
7.     }  
8. }
```

我们看到 getter 是从 env->env_vars() 中获取数据，那么 env->env_vars() 又是什么呢？env_vars 是一个 kv 存储系统，其实就是一个 map。它只在 Node.js 初始化的时候设置（创建 env 对象时）。

```
set_env_vars(per_process::system_environment);
```

那么 per_process::system_environment 又是什么呢？我们继续往下看，

```
std::shared_ptr<KVStore> system_environment = std::make_shared<RealEnvStore>();
```

我们看到 system_environment 是一个 RealEnvStore 对象。我们看一下 RealEnvStore 类的实现。

```
1. class RealEnvStore final : public KVStore {
2. public:
3.     MaybeLocal<String> Get(Isolate* isolate, Local<String> key) const override;
4.     void Set(Isolate* isolate, Local<String> key, Local<String> value) override;
5.     int32_t Query(Isolate* isolate, Local<String> key) const override;
6.     void Delete(Isolate* isolate, Local<String> key) override;
7.     Local<Array> Enumerate(Isolate* isolate) const override;
8. };
```

比较简单，就是增删改查，我们看一下查询 Get 的实现。

```
1. MaybeLocal<String> RealEnvStore::Get(Isolate* isolate,
2.
3.     Local<String> property) const {
4.     Mutex::ScopedLock lock(per_process::env_var_mutex);
5.
6.     node::Utf8Value key(isolate, property);
7.     size_t init_sz = 256;
8.     MaybeStackBuffer<char, 256> val;
9.     int ret = uv_os_getenv(*key, *val, &init_sz);
10.    if (ret >= 0) { // Env key value fetch success.
11.        MaybeLocal<String> value_string =
12.            String::NewFromUtf8(isolate,
13.                *val,
14.                NewStringType::kNormal,
```

```
14.                                     init_sz);  
15.         return value_string;  
16.     }  
17.  
18.     return MaybeLocal<String>();  
19. }
```

我们看到是通过 `uv_os_getenv` 获取的数据。`uv_os_getenv` 是对 `getenv` 函数的封装，进程的内存布局中，有一部分是用于存储环境变量的，`getenv` 就是从那一块内存中把数据读取出来。我们执行 `execve` 的时候可以设置环境变量。具体的我们在子进程章节会看到。至此，我们知道 `process` 的 `env` 属性对应的值就是进程环境变量的内容。

13.2.3 挂载其它属性

在 `Node.js` 的启动过程中会不断地挂载属性到 `process`。主要在 `bootstrap/node.js` 中。不一一列举。

```
1. const rawMethods = internalBinding('process_methods');  
2. process.dlopen = rawMethods.dlopen;  
3. process.uptime = rawMethods.uptime;  
4. process.nextTick = nextTick;
```

下面是 `process_methods` 模块导出的属性，主列出常用的。

```
1. env->SetMethod(target, "memoryUsage", MemoryUsage);  
2. env->SetMethod(target, "cpuUsage", CPUUsage);  
3. env->SetMethod(target, "hrtime", Hrtime);  
4. env->SetMethod(target, "dlopen", binding::DLOpen);  
5. env->SetMethodNoSideEffect(target, "uptime", Uptime);
```

我们看到在 JS 层访问 `process` 属性的时候，访问的是对应的 C++ 层的这些方法，大部分也只是对 `Libuv` 的封装。另外在 `Node.js` 初始化的过程中会执行 `PatchProcessObject`。`PatchProcessObject` 函数会挂载一些额外的属性给 `process`。

```
1. // process.argv  
2. process->Set(context,  
3.                 FIXED_ONE_BYTE_STRING(isolate, "argv"),  
4.                 ToV8Value(context, env->argv()).ToLocalChecked()).Check();  
5.  
6. READONLY_PROPERTY(process,  
7.                     "pid",  
8.                     Integer::New(isolate, uv_os_getpid()));  
9.
```

```

10. CHECK(process->SetAccessor(context,
11.                               FIXED_ONE_BYTE_STRING(isolate, "ppid"),
12.                               GetParentProcessId).FromJust())

```

在 Node.js 初始化的过程中，在多个地方都会给 process 对象挂载属性，这里只列出了一部分，有兴趣的同学可以从 bootstrap/node.js 的代码开始看都挂载了什么属性。因为 Node.js 支持多线程，所以针对线程的情况，有一些特殊的处理。

```

1. const perThreadSetup = require('internal/process/per_thread');
2. // rawMethods 来自 process_methods 模块导出的属性
3. const wrapped = perThreadSetup.wrapProcessMethods(rawMethods);
4. process.hrtime = wrapped.hrtime;
5. process.cpuUsage = wrapped.cpuUsage;
6. process.memoryUsage = wrapped.memoryUsage;
7. process.kill = wrapped.kill;
8. process.exit = wrapped.exit;

```

大部分函数都是对 process_methods 模块 (node_process_methods.cc) 的封装。但是有一个属性我们需要关注一下，就是 exit，因为在线程中调用 process.exit 的时候，只会退出单个线程，而不是整个进程。

```

1. function exit(code) {
2.   if (code || code === 0)
3.     process.exitCode = code;
4.
5.   if (!process._exiting) {
6.     process._exiting = true;
7.     process.emit('exit', process.exitCode || 0);
8.   }
9.   process.reallyExit(process.exitCode || 0);
10. }

```

我们继续看 reallyExit

```

1. static void ReallyExit(const FunctionCallbackInfo<Value>& args) {
2.   Environment* env = Environment::GetCurrent(args);
3.   RunAtExit(env);
4.   int code = args[0]->Int32Value(env->context()).FromMaybe(0);
5.   env->Exit(code);
6. }

```

调用了 env 的 Exit。

```
1. void Environment::Exit(int exit_code) {
2.     if (is_main_thread()) {
3.         stop_sub_worker_contexts();
4.         DisposePlatform();
5.         exit(exit_code);
6.     } else {
7.         worker_context_>Exit(exit_code);
8.     }
9. }
```

这里我们看到了重点，根据当前是主线程还是子线程会做不同的处理。一个线程会对应一个 env，env 对象中的 worker_context_ 保存就是线程对象（Worker）。我们先看子线程的逻辑。

```
1. void Worker::Exit(int code) {
2.     Mutex::ScopedLock lock(mutex_);
3.     if (env_ != nullptr) {
4.         exit_code_ = code;
5.         Stop(env_);
6.     } else {
7.         stopped_ = true;
8.     }
9. }
10.
11. int Stop(Environment* env) {
12.     env->ExitEnv();
13.     return 0;
14. }
15.
16. void Environment::ExitEnv() {
17.     set_can_call_into_js(false);
18.     set_stopping(true);
19.     isolate_->TerminateExecution();
20.     // 退出 Libuv 事件循环
21.     SetImmediateThreadsafe([](Environment* env) { uv_stop(env->event_
    loop()); });
22. }
```

我们看到子线程最后调用 uv_stop 提出了 Libuv 事件循环，然后退出。我们再来看主线程的退出逻辑。

```

1. if (is_main_thread()) {
2.     stop_sub_worker_contexts();
3.     DisposePlatform();
4.     exit(exit_code);
5. }
```

我们看到最后主进程中调用 exit 退出进程。但是退出前还有一些处理工作，我们看 stop_sub_worker_contexts

```

1. void Environment::stop_sub_worker_contexts() {
2.     while (!sub_worker_contexts_.empty()) {
3.         Worker* w = *sub_worker_contexts_.begin();
4.         remove_sub_worker_context(w);
5.         w->Exit(1);
6.         w->JoinThread();
7.     }
8. }
```

sub_worker_contexts 保存的是 Worker 对象列表，每次创建一个线程的时候，就会往里追加一个元素。这里遍历这个列表，然后调用 Exit 函数，这个刚才我们已经分析过，就是退出 Libuv 事件循环。主线程接着调 JoinThread，JoinThread 主要是为了阻塞等待子线程退出，因为子线程在退出的时候，可能会被操作系统挂起（执行时间片到了），这时候主线程被调度执行，但是这时候主线程还不能退出，所以这里使用 join 阻塞等待子线程退出。Node.js 的 JoinThread 除了对线程 join 函数的封装。还做了一些额外的事情，比如触发 exit 事件。

3.3 创建子进程

因为 Node.js 是单进程的，但有很多事情可能不适合在主进程里处理的，所以 Node.js 提供了子进程模块，我们可以创建子进程做一些额外任务的处理，另外，子进程的好处是，一旦子进程出问题挂掉不会影响主进程。我们首先看一下在用 C 语言如何创建一个进程。

```

1. #include<unistd.h>
2. #include<stdlib.h>
3.
4. int main(int argc, char *argv[]) {
5.     pid_t pid = fork();
6.     if (pid < 0) {
7.         // 错误
8.     } else if (pid == 0) {
9.         // 子进程，可以使用 exec*系列函数执行新的程序
10.    } else {
```

```
11.          // 父进程  
12.      }  
13. }
```

fork 函数的特点，我们听得最多的可能是执行一次返回两次，我们可能会疑惑，执行一个函数怎么可能返回了两次呢？之前我们讲过，进程是 task_struct 表示的一个实例，调用 fork 的时候，操作系统会新建一个新的 task_struct 实例出来（变成两个进程），fork 返回两次的意思其实是在在两个进程分别返回一次，执行的都是 fork 后面的一行代码。而操作系统根据当前进程是主进程还是子进程，设置了 fork 函数的返回值。所以不同的进程，fork 返回值不一样，也就是我们代码中 if else 条件。但是 fork 只是复制主进程的内容，如果我们想执行另外一个程序，怎么办呢？这时候就需要用到 exec* 系列函数，该系列函数会覆盖旧进程 (task_struct) 的部分内容，重新加载新的程序内容。这也是 Node.js 中创建子进程的底层原理。Node.js 虽然提供了很多种创建进程的方式，但是本质上是同步和异步两种方式。

13.3.1 异步创建进程

我们首先看一下异步方式创建进程时的关系图如图 13-1 所示。

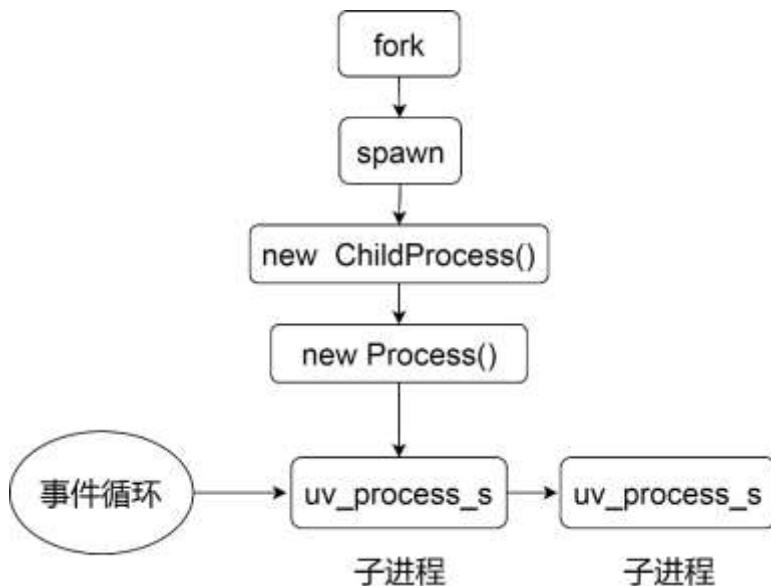


图 13-1

我们从 fork 这个函数开始，看一下整个流程。

```
1. function fork(modulePath /* , args, options */) {  
2.     // 一系列参数处理  
3.     return spawn(options.execPath, args, options);  
4. }
```

我们接着看 spawn

```

1. var spawn = exports.spawn = function(*file, args, options*) {
2.     var opts = normalizeSpawnArguments.apply(null, arguments);
3.     var options = opts.options;
4.     var child = new ChildProcess();
5.     child.spawn({
6.         file: opts.file,
7.         args: opts.args,
8.         cwd: options.cwd,
9.         windowsHide: !!options.windowsHide,
10.        windowsVerbatimArguments: !!options.windowsVerbatimArguments,
11.       detached: !!options.detached,
12.       envPairs: opts.envPairs,
13.       stdio: options.stdio,
14.       uid: options.uid,
15.       gid: options.gid
16.     });
17.
18.     return child;
19. };

```

我们看到 spawn 函数只是对 ChildProcess 的封装。然后调用它的 spawn 函数。我们看看 ChildProcess。

```

1. function ChildProcess() {
2.     // C++层定义
3.     this._handle = new Process();
4. }
5.
6. ChildProcess.prototype.spawn = function(options) {
7.     // 创建进程
8.     const err = this._handle.spawn(options);
9. }
10.

```

ChildProcess 是对 C++ 层的封装，不过 Process 在 C++ 层也没有太多逻辑，进行参数的处理然后调用 Libuv 的 uv_spawn。我们通过 uv_spawn 来到了 C 语言层。我们看看 uv_spawn 的整体流程。

```
1. int uv_spawn(uv_loop_t* loop,
```

```
2.         uv_process_t* process,
3.         const uv_process_options_t* options) {
4.
5.     uv__handle_init(loop, (uv_handle_t*)process, UV_PROCESS);
6.     QUEUE_INIT(&process->queue);
7.     // 处理进程间通信
8.     for (i = 0; i < options->stdio_count; i++) {
9.         err = uv__process_init_stdio(options->stdio + i, pipes[i]);
10.        if (err)
11.            goto error;
12.    }
13.    /*
14.     创建一个管道用于创建进程期间的父进程子通信,
15.     设置 UV_O_CLOEXEC 标记, 子进程执行 execvp
16.     的时候管道的一端会被关闭
17.    */
18.    err = uv__make_pipe(signal_pipe, 0);
19.    // 注册子进程退出信号的处理函数
20.    uv_signal_start(&loop->child_watcher, uv_chld, SIGCHLD);
21.
22.    uv_rwlock_wrlock(&loop->cloexec_lock);
23.    // 创建子进程
24.    pid = fork();
25.    // 子进程
26.    if (pid == 0) {
27.        uv__process_child_init(options,
28.                               stdio_count,
29.                               pipes,
30.                               signal_pipe[1]);
31.        abort();
32.    }
33.    // 父进程
34.    uv_rwlock_wrunlock(&loop->cloexec_lock);
35.    // 关闭管道写端, 等待子进程写
36.    uv_close(signal_pipe[1]);
37.
38.    process->status = 0;
39.    exec_errno = 0;
40.    // 判断子进程是否执行成功
41.    do
42.        r = read(signal_pipe[0], &exec_errno, sizeof(exec_errno));
43.    while (r == -1 && errno == EINTR);
44.    // 忽略处理 r 的逻辑
45.    // 保存通信的文件描述符到对应的数据结构
46.    for (i = 0; i < options->stdio_count; i++) {
```

```

47.     uv__process_open_stream(options->stdio + i, pipes[i]);
48. }
49.
50. // 插入 Libuv 事件循环的结构体
51. if (exec_errno == 0) {
52.     QUEUE_INSERT_TAIL(&loop->process_handles, &process->queue);
53.     uv__handle_start(process);
54. }
55.
56. process->pid = pid;
57. process->exit_cb = options->exit_cb;
58.
59. return exec_errno;
60. }

```

uv_spawn 的逻辑大致分为下面几个

1 处理进程间通信

2 注册子进程退出处理函数

3 创建子进程

4 插入 Libuv 事件循环的 `process_handles` 对象，保存状态码和回调等。

我们分析 2,3，进程间通信我们单独分析。

1 处理子进程退出

主进程在创建子进程之前，会注册 SIGCHLD 信号。对应的处理函数是 `uv_chld`。当进程退出的时候。Node.js 主进程会收到 SIGCHLD 信号。然后执行 `uv_chld`。该函数遍历 Libuv 进程队列中的节点，通过 `waitpid` 判断该节点对应的进程是否已经退出后，从而处理已退出的节点，然后移出 Libuv 队列，最后执行已退出进程的回调。

```

1. static void uv_chld(uv_signal_t* handle, int signum) {
2.     uv_process_t* process;
3.     uv_loop_t* loop;
4.     int exit_status;
5.     int term_signal;
6.     int status;
7.     pid_t pid;
8.     QUEUE pending;
9.     QUEUE* q;

```

```
10.    QUEUE* h;
11.    // 保存进程（已退出的状态）的队列
12.    QUEUE_INIT(&pending);
13.    loop = handle->loop;
14.
15.    h = &loop->process_handles;
16.    q = QUEUE_HEAD(h);
17.    // 收集已退出的进程
18.    while (q != h) {
19.        process = QUEUE_DATA(q, uv_process_t, queue);
20.        q = QUEUE_NEXT(q);
21.
22.        do
23.            /*
24.                WNOHANG 非阻塞等待子进程退出，其实就是看子进程是否退出了，
25.                没有的话就直接返回，而不是阻塞
26.            */
27.            pid = waitpid(process->pid, &status, WNOHANG);
28.            while (pid == -1 && errno == EINTR);
29.
30.            if (pid == 0)
31.                continue;
32.            /*
33.                进程退出了，保存退出状态，移出队列，
34.                插入 pending 队列，等待处理
35.            */
36.            process->status = status;
37.            QUEUE_REMOVE(&process->queue);
38.            QUEUE_INSERT_TAIL(&pending, &process->queue);
39.    }
40.
41.    h = &pending;
42.    q = QUEUE_HEAD(h);
43.    // 是否有退出的进程
44.    while (q != h) {
45.        process = QUEUE_DATA(q, uv_process_t, queue);
46.        q = QUEUE_NEXT(q);
47.        QUEUE_REMOVE(&process->queue);
48.        QUEUE_INIT(&process->queue);
49.        uv_handle_stop(process);
50.
51.        if (process->exit_cb == NULL)
```

```

52.         continue;
53.
54.     exit_status = 0;
55.     // 获取退出信息，执行上传回调
56.     if (WIFEXITED(process->status))
57.         exit_status = WEXITSTATUS(process->status);
58.     // 是否因为信号而退出
59.     term_signal = 0;
60.     if (WIFSIGNALED(process->status))
61.         term_signal = WTERMSIG(process->status);
62.
63.     process->exit_cb(process, exit_status, term_signal);
64. }
65. }

```

当主进程下的子进程退出时，父进程主要负责收集子进程退出状态和原因等信息，然后执行上层回调。

2 创建子进程 (uv_process_child_init)

主进程首先使用 `uv_make_pipe` 申请一个匿名管道用于主进程和子进程通信，匿名管道是进程间通信中比较简单的一种，它只用于有继承关系的进程，因为匿名，非继承关系的进程无法找到这个管道，也就无法完成通信，而有继承关系的进程，是通过 `fork` 出来的，父子进程可以获得得到管道。进一步来说，子进程可以使用继承于父进程的资源，管道通信的原理如图 13-2 所示。

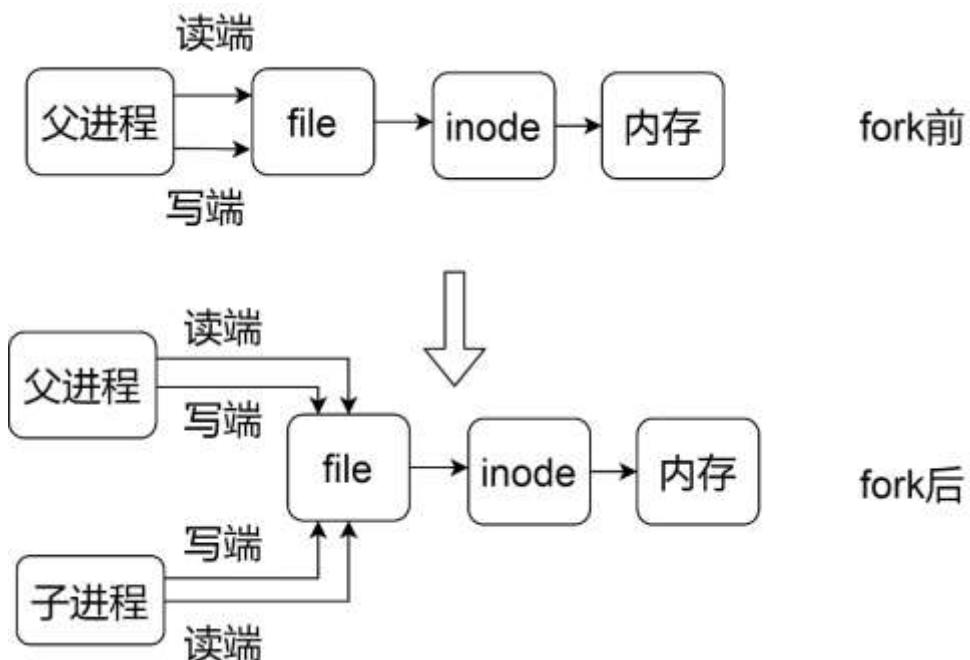


图 13-2

主进程和子进程通过共享 file 和 inode 结构体，实现对同一块内存的读写。主进程 fork 创建子进程后，会通过 read 阻塞等待子进程的消息。我们看一下子进程的逻辑。

```
1. static void uv_process_child_init(const uv_process_options_t* options,
2.                                     int stdio_count,
3.                                     int (*pipes)[2],
4.                                     int error_fd) {
5.     sigset_t set;
6.     int close_fd;
7.     int use_fd;
8.     int err;
9.     int fd;
10.    int n;
11.    // 省略处理文件描述符等参数逻辑
12.    // 处理环境变量
13.    if (options->env != NULL) {
14.        environ = options->env;
15.    }
16.    // 处理信号
17.    for (n = 1; n < 32; n += 1) {
18.        // 这两个信号触发时，默认行为是进程退出且不能阻止的
19.        if (n == SIGKILL || n == SIGSTOP)
20.            continue; /* Can't be changed. */
21.        // 设置为默认处理方式
22.        if (SIG_ERR != signal(n, SIG_DFL))
23.            continue;
24.        // 出错则通知主进程
25.        uv_write_int(error_fd, UV_ERR(errno));
26.        _exit(127);
27.    }
28.    // 加载新的执行文件
29.    execvp(options->file, options->args);
30.    // 加载成功则不会走到这，走到这说明加载执行文件失败
31.    uv_write_int(error_fd, UV_ERR(errno));
32.    _exit(127);
33.}
```

子进程的逻辑主要是处理文件描述符、信号、设置环境变量等。然后加载新的执行文件。因为主进程和子进程通信的管道对应的文件描述符设置了 cloexec 标记。所以当子进程加载新的执行文件时，就会关闭用于和主进程通信的管道文件描述符，从而导致主进程读取管道读端的时候返回 0，这样主进程就知道子进程成功执行了。

13.3.2 同步创建进程

同步方式创建的进程，主进程会等待子进程退出后才能继续执行。接下来看看如何以同步的方式创建进程。JS 层入口函数是 spawnSync。spawnSync 调用 C++ 模块 spawn_sync 的 spawn 函数创建进程，我们看一下对应的 C++ 模块 spawn_sync 导出的属性。

```
1. void SyncProcessRunner::Initialize(Local<Object> target,
2.                                     Local<Value> unused,
3.                                     Local<Context> context,
4.                                     void* priv) {
5.   Environment* env = Environment::GetCurrent(context);
6.   env->SetMethod(target, "spawn", Spawn);
7. }
```

该模块值导出了一个属性 spawn，当我们调用 spawn 的时候，执行的是 C++ 的 Spawn。

```
1. void SyncProcessRunner::Spawn(const FunctionCallbackInfo<Value>&
2.                                args) {
3.   Environment* env = Environment::GetCurrent(args);
4.   env->PrintSyncTrace();
5.   SyncProcessRunner p(env);
6.   Local<Value> result;
7.   if (!p.Run(args[0]).ToLocal(&result)) return;
8.   args.GetReturnValue().Set(result);
9. }
```

Spawn 中主要是新建了一个 SyncProcessRunner 对象并且执行 Run 方法。我们看一下 SyncProcessRunner 的 Run 做了什么。

```
1. MaybeLocal<Object> SyncProcessRunner::Run(Local<Value> options) {
2.   EscapableHandleScope scope(env()->isolate());
3.   Maybe<bool> r = TryInitializeAndRunLoop(options);
4.   Local<Object> result = BuildResultObject();
5.   return scope.Escape(result);
6. }
```

执行了 TryInitializeAndRunLoop。

```
1. Maybe<bool> SyncProcessRunner::TryInitializeAndRunLoop(Local<Value> options) {
2.     int r;
3.
4.     lifecycle_ = kInitialized;
5.     // 新建一个事件循环
6.     uv_loop_ = new uv_loop_t;
7.     if (!ParseOptions(options).To(&r)) return Nothing<bool>();
8.     if (r < 0) {
9.         SetError(r);
10.        return Just(false);
11.    }
12.    // 设置子进程执行的时间
13.    if (timeout_ > 0) {
14.        r = uv_timer_init(uv_loop_, &uv_timer_);
15.        uv_unref(reinterpret_cast<uv_handle_t*>(&uv_timer_));
16.        uv_timer_.data = this;
17.        kill_timer_initialized_ = true;
18.        // 开启一个定时器，超时执行 KillTimerCallback
19.        r = uv_timer_start(&uv_timer_,
20.                           KillTimerCallback,
21.                           timeout_,
22.                           0);
23.    }
24.    // 子进程退出时处理函数
25.    uv_process_options_.exit_cb = ExitCallback;
26.    // 传进去新的 loop 而不是主进程本身的 loop
27.    r = uv_spawn(uv_loop_, &uv_process_, &uv_process_options_);
28.    uv_process_.data = this;
29.
30.    for (const auto& pipe : stdio_pipes_) {
31.        if (pipe != nullptr) {
32.            r = pipe->Start();
33.            if (r < 0) {
34.                SetPipeError(r);
35.                return Just(false);
36.            }
37.        }
38.    }
39.    // 开启一个新的事件循环
40.    r = uv_run(uv_loop_, UV_RUN_DEFAULT);
41.    return Just(true);
```

| 42. }

从上面的代码中，我们可以了解到 Node.js 是如何实现同步创建进程的。同步创建进程时，Node.js 重新开启了一个事件循环，然后新建一个子进程，并且把表示子进程结构体的 handle 插入到新创建的事件循环中，接着 Libuv 一直处于事件循环中，因为一直有一个 uv_process_t (handle)，所以新创建的 uv_run 会一直在执行，所以这时候，Node.js 主进程会“阻塞”在该 uv_run。直到子进程退出，主进程收到信号后，删除新创建的事件循环中的 uv_process_t。然后执行回调 ExitCallback。接着事件循环退出，再次回到 Node.js 原来的事件循环。如图所示 13-3。

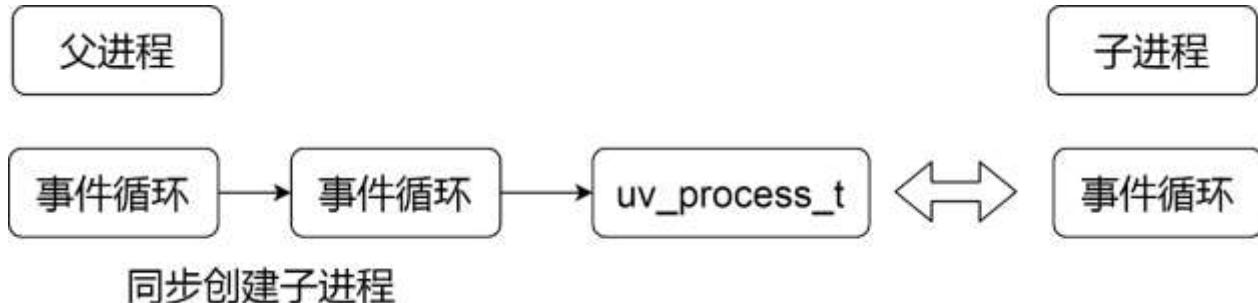


图 13-3

这就是同步的本质和原因。我们分几步分析一下以上代码

13.3.2.1 执行时间

因为同步方式创建子进程会导致 Node.js 主进程阻塞，为了避免子进程有问题，从而影响主进程的执行，Node.js 支持可配置子进程的最大执行时间。我们看到，Node.js 开启了一个定时器，并设置了回调 KillTimerCallback。

```

1. void SyncProcessRunner::KillTimerCallback(uv_timer_t* handle) {
2.     SyncProcessRunner* self = reinterpret_cast<SyncProcessRunner*>(handle->data);
3.     self->OnKillTimerTimeout();
4. }
5.
6. void SyncProcessRunner::OnKillTimerTimeout() {
7.     SetError(UV_ETIMEDOUT);
8.     Kill();
9. }
10.
11. void SyncProcessRunner::Kill() {
12.     if (killed_)
13.         return;
14.     killed_ = true;
15.     if (exit_status_ < 0) {

```

```
16.      // kill_signal_为用户自定义发送的杀死进程的信号
17.      int r = uv_process_kill(&uv_process_, kill_signal_);
18.      // 不支持用户传的信号
19.      if (r < 0 && r != UV_ESRCH) {
20.          SetError(r);
21.          // 回退使用 SIGKILL 信号杀死进程
22.          r = uv_process_kill(&uv_process_, SIGKILL);
23.          CHECK(r >= 0 || r == UV_ESRCH);
24.      }
25.  }
26.
27.  // Close all stdio pipes.
28. CloseStdioPipes();
29.
30.  // 清除定时器
31. CloseKillTimer();
32. }
```

当执行时间到达设置的阈值，Node.js 主进程会给子进程发送一个信号，默认是杀死子进程。

13.3.2.2 子进程退出处理

退出处理主要是记录子进程退出时的错误码和被哪个信号杀死的（如果有的话）。

```
1. void SyncProcessRunner::ExitCallback(uv_process_t* handle,
2.
3.     int64_t exit_status,
4.
5.     int term_signal) {
6.     SyncProcessRunner* self = reinterpret_cast<SyncProcessRunner*>(handle->data);
7.     uv_close(reinterpret_cast<uv_handle_t*>(handle), nullptr);
8.     self->OnExit(exit_status, term_signal);
9. }
10.
11. void SyncProcessRunner::OnExit(int64_t exit_status, int term_signal)
12. {
13.     if (exit_status < 0)
14.         return SetError(static_cast<int>(exit_status));
15.
16.     exit_status_ = exit_status;
```

```

14.     term_signal_ = term_signal;
15. }
```

13.4 进程间通信

进程间通信是多进程系统中非常重要的功能，否则进程就像孤岛一样，不能交流信息。因为进程间的内存是隔离的，如果进程间想通信，就需要一个公共的地方，让多个进程都可以访问，完成信息的传递。在 Linux 中，同主机的进程间通信方式有很多，但是基本都是使用独立于进程的额外内存作为信息承载的地方，然后在通过某种方式让多个进程都可以访问到这块公共内存，比如管道、共享内存、Unix 域、消息队列等等。不过还有另外一种进程间通信的方式，是不属于以上情况的，那就是信号。信号作为一种简单的进程间通信方式，操作系统提供了接口让进程可以直接修改另一个进程的数据（PCB），以此达到通信目的。本节介绍 Node.js 中进程间通信的原理和实现。

13.4.1 创建通信通道

我们从 fork 函数开始分析 Node.js 中进程间通信的逻辑。

```

1. function fork(modulePath) {
2.   // 忽略 options 参数处理
3.   if (typeof options.stdio === 'string') {
4.     options.stdio = stdioStringToArray(options.stdio, 'ipc');
5.   } else if (!Array.isArray(options.stdio)) {
6.     // silent 为 true 则是管道形式和主进程通信，否则是继承
7.     options.stdio = stdioStringToArray(
8.       options.silent ? 'pipe' : 'inherit',
9.       'ipc');
10.  } else if (!options.stdio.includes('ipc')) {
11.    // 必须要 IPC，支持进程间通信
12.    throw new ERR_CHILD_PROCESS_IPC_REQUIRED('options.stdio');
13.  }
14.
15.  return spawn(options.execPath, args, options);
16. }
```

我们看一下 stdioStringToArray 的处理。

```

1. function stdioStringToArray(stdio, channel) {
2.   const options = [];
3.
4.   switch (stdio) {
5.     case 'ignore':
6.     case 'pipe': options.push(stdio, stdio, stdio); break;
```

```
7.     case 'inherit': options.push(0, 1, 2); break;
8.     default:
9.       throw new ERR_INVALID_OPT_VALUE('stdio', stdio);
10.    }
11.
12.    if (channel) options.push(channel);
13.
14.  return options;
15. }
```

stdioStringToArray 会返回一个数组，比如['pipe', 'pipe', 'pipe', 'ipc']或[0, 1, 2, 'ipc']， ipc 代表需要创建一个进程间通信的通道，并且支持文件描述传递。我们接着看 spawn。

```
1. ChildProcess.prototype.spawn = function(options) {
2.   let i = 0;
3.   // 预处理进程间通信的数据结构
4.   stdio = getValidStdio(stdio, false);
5.   const ipc = stdio.ipc;
6.   // IPC 文件描述符
7.   const ipcFd = stdio.ipcFd;
8.   stdio = options.stdio = stdio.stdio;
9.   // 通过环境变量告诉子进程 IPC 文件描述符和数据处理模式
10.  if (ipc !== undefined) {
11.    options.envPairs.push(`NODE_CHANNEL_FD=${ipcFd}`);
12.    options.envPairs.push(`NODE_CHANNEL_SERIALIZATION_MODE=${ser
13.      ialization}`);
14.  }
15.  // 创建子进程
16.  const err = this._handle.spawn(options);
17.  this.pid = this._handle.pid;
18.  // 处理 IPC 通信
19.  if (ipc !== undefined) setupChannel(this, ipc, serialization);
20. }
```

Spawn 中会执行 getValidStdio 预处理进程间通信的数据结构。我们只关注 ipc 的。

```
1. function getValidStdio(stdio, sync) {
2.   let ipc;
3.   let ipcFd;
4.
5.   stdio = stdio.reduce((acc, stdio, i) => {
6.     if (stdio === 'ipc') {
```

```

7.     ipc = new Pipe(PipeConstants.IPC);
8.     ipcFd = i;
9.     acc.push({
10.         type: 'pipe',
11.         handle: ipc,
12.         ipc: true
13.     });
14. } else {
15.     // 其它类型的处理
16. }
17. return acc;
18. }, []);
19.
20. return { stdio, ipc, ipcFd };
21. }

```

我们看到这里会 new Pipe(PipeConstants.IPC); 创建一个 Unix 域用于进程间通信，但是这里只是定义了一个 C++ 对象，还没有可用的文件描述符。我们接着往下看 C++ 层的 spawn 中关于进程间通信的处理。C++ 层首先处理参数，

```

1. static void ParseStdioOptions(Environment* env,
2.                               Local<Object> js_options,
3.                               uv_process_options_t* options) {
4.
5.     Local<Context> context = env->context();
6.     Local<String> stdio_key = env->stdio_string();
7.     // 拿到 JS 层 stdio 的值
8.     Local<Array> stdios =
9.         js_options->Get(context, stdio_key).ToLocalChecked().As<A
10.        rray>();
11.
12.     uint32_t len = stdios->Length();
13.     options->stdio = new uv_stdio_container_t[len];
14.     options->stdio_count = len;
15.     // 遍历 stdio, stdio 是一个对象数组
16.     for (uint32_t i = 0; i < len; i++) {
17.         Local<Object> stdio =
18.             stdios->Get(context, i).ToLocalChecked().As<Object>();
19.
20.         // 拿到 stdio 的类型
21.         Local<Value> type =
22.             stdio->Get(context, env->type_string()).ToLocalChecked
23.             ();
24.         // 创建 IPC 通道
25.         if (type->StrictEquals(env->pipe_string())) {

```

```
22.     options->stdio[i].flags = static_cast<uv_stdio_flags>(
23.         UV_CREATE_PIPE | UV_READABLE_PIPE | UV_WRITABLE_PIPE
24.     );
25.     // 拿到对应的 stream
26.     options->stdio[i].data.stream = StreamForWrap(env, stdio
27. );
28. }
```

这里会把 StreamForWrap 的结果保存到 stream 中，我们看看 StreamForWrap 的逻辑

```
1. static uv_stream_t* StreamForWrap(Environment* env, Local<Object
   > stdio) {
2.     Local<String> handle_key = env->handle_string();
3.     /*
4.         获取对象中的 key 为 handle 的值，即刚才 JS 层的
5.         new Pipe(SOCKET IPC);
6.     */
7.     Local<Object> handle =
8.         stdio->Get(env->context(), handle_key).ToLocalChecked().As
   <Object>();
9.     // 获取 JS 层使用对象所对应的 C++ 对象中的 stream
10.    uv_stream_t* stream = LibuvStreamWrap::From(env, handle)->str
   eam();
11.    CHECK_NOT_NULL(stream);
12.    return stream;
13. }
14.
15. // 从 JS 层使用的 object 中获取关联的 C++ 对象
16. LibuvStreamWrap* LibuvStreamWrap::From(Environment* env, Local<Ob
   ject> object) {
17.     return Unwrap<LibuvStreamWrap>(object);
18. }
```

以上代码获取了 IPC 对应的 stream 结构体。在 Libuv 中会把文件描述符保存到 stream 中。我们接着看 C++ 层调用 Libuv 的 uv_spawn。

```
1. int uv_spawn(uv_loop_t* loop,
2.               uv_process_t* process,
3.               const uv_process_options_t* options) {
4.
5.     int pipes_storage[8][2];
6.     int (*pipes)[2];
7.     int stdio_count;
```

```

8. // 初始化进程间通信的数据结构
9. stdio_count = options->stdio_count;
10. if (stdio_count < 3)
11.     stdio_count = 3;
12.
13. for (i = 0; i < stdio_count; i++) {
14.     pipes[i][0] = -1;
15.     pipes[i][1] = -1;
16. }
17. // 创建进程间通信的文件描述符
18. for (i = 0; i < options->stdio_count; i++) {
19.     err = uv__process_init_stdio(options->stdio + i, pipes[i]);
20.     if (err)
21.         goto error;
22. }
23.
24. // 设置进程间通信文件描述符到对应的数据结构
25. for (i = 0; i < options->stdio_count; i++) {
26.     uv__process_open_stream(options->stdio + i, pipes[i]);
27.
28. }
29.
30. }
```

Libuv 中会创建用于进程间通信的文件描述符，然后设置到对应的数据结构中。

```

1. static int uv__process_open_stream(uv_stdio_container_t* container,
2.                                     int pipefds[2]) {
3.     int flags;
4.     int err;
5.
6.     if (!(container->flags & UV_CREATE_PIPE) || pipefds[0] < 0)
7.         return 0;
8.
9.     err = uv__close(pipefds[1]);
10.    if (err != 0)
11.        abort();
12.
13.    pipefds[1] = -1;
14.    uv__nonblock(pipefds[0], 1);
15.
16.    flags = 0;
17.    if (container->flags & UV_WRITABLE_PIPE)
18.        flags |= UV_HANDLE_READABLE;
19.    if (container->flags & UV_READABLE_PIPE)
```

```
20.     flags |= UV_HANDLE_WRITABLE;
21.
22.     return uv__stream_open(container->data.stream, pipefds[0], flags);
23. }
```

执行完 `uv__process_open_stream`, 用于 IPC 的文件描述符就保存到 `new Pipe(SOCKET.IPC)` 中了。有了 IPC 通道的文件描述符, 进程还需要进一步处理。我们看到 JS 层执行完 `spawn` 后, 主进程通过 `setupChannel` 对进程间通信进行了进一步处理。我们看一下主进程 `setupChannel` 中关于进程间通信的处理。

13.4.2 主进程处理通信通道

1 读端

```
1. function setupChannel(target, channel, serializationMode) {
2.     // channel 是 new Pipe(PipeConstants.IPC);
3.     const control = new Control(channel);
4.     target.channel = control;
5.     // ...
6.     channel.pendingHandle = null;
7.     // 注册处理数据的函数
8.     channel.onread = function(arrayBuffer) {
9.         // 收到的文件描述符
10.        const recvHandle = channel.pendingHandle;
11.        channel.pendingHandle = null;
12.        if (arrayBuffer) {
13.            const nread = streamBaseState[kReadBytesOrError];
14.            const offset = streamBaseState[kArrayBufferOffset];
15.            const pool = new Uint8Array(arrayBuffer, offset, nread);
16.            if (recvHandle)
17.                pendingHandle = recvHandle;
18.            // 解析收到的消息
19.            for (const message of parseChannelMessages(channel, pool)) {
20.                // 是否是内部通信事件
21.                if (isInternal(message)) {
22.                    // 收到 handle
23.                    if (message.cmd === 'NODE_HANDLE') {
24.                        handleMessage(message, pendingHandle, true);
25.                        pendingHandle = null;
26.                    } else {
27.                        handleMessage(message, undefined, true);
28.                    }
29.                } else {
30.                    handleMessage(message, undefined, false);
31.                }
32.            }
33.        }
34.    };
35. }
36.
37. function handleMessage(message, handle, internal) {
38.     const eventName = (internal ? 'internalMessage' : 'message');
39.     process.nextTick(emit, eventName, message, handle);
```

```

40.  }
41. // 开启读
42. channel.readStart();
43. return control;
44. }
```

onread 处理完后会触发 internalMessage 或 message 事件， message 是用户使用的。

2 写端

```

1. target._send = function(message, handle, options, callback) {
2.   let obj;
3.   const req = new WriteWrap();
4.   // 发送给对端
5.   const err = writeChannelMessage(channel, req, message, handle);
6.
7.   return channel.writeQueueSize < (65536 * 2);
8. }
```

我们看看 writeChannelMessage

```

1. writeChannelMessage(channel, req, message, handle) {
2.   const ser = new ChildProcessSerializer();
3.   ser.writeHeader();
4.   ser.writeValue(message);
5.   const serializedMessage = ser.releaseBuffer();
6.   const sizeBuffer = Buffer.allocUnsafe(4);
7.   sizeBuffer.writeUInt32BE(serializedMessage.length);
8.   // channel 是封装了 Unix 域的对象
9.   return channel.writeBuffer(req, Buffer.concat([
10.     sizeBuffer,
11.     serializedMessage
12.   ]), handle);
13. },
```

`channel.writeBuffer` 通过刚才创建的 IPC 通道完成数据的发送，并且支持发送文件描述符。

13.4.3 子进程处理通信通道

接着我们看看子进程的逻辑，Node.js 在创建子进程的时候，主进程会通过环境变量 `NODE_CHANNEL_FD` 告诉子进程 Unix 域通信对应的文件描述符。在执行子进程的时候，会处理这个文件描述符。具体实现在 `setupChildProcessIpcChannel` 函数中。

```
1. function setupChildProcessIpcChannel() {  
2.   // 主进程通过环境变量设置该值  
3.   if (process.env.NODE_CHANNEL_FD) {  
4.     const fd = parseInt(process.env.NODE_CHANNEL_FD, 10);  
5.     delete process.env.NODE_CHANNEL_FD;  
6.     require('child_process')._forkChild(fd, serializationMode);  
7.   }  
8. }
```

接着执行 _forkChild 函数。

```
1. function _forkChild(fd, serializationMode) {  
2.   const p = new Pipe(PipeConstants.IPC);  
3.   p.open(fd);  
4.   const control = setupChannel(process, p, serializationMode);  
5. }
```

该函数创建一个 Pipe 对象，然后把主进程传过来的 fd 保存到该 Pipe 对象。对该 Pipe 对象的读写，就是地对 fd 进行读写。最后执行 setupChannel。setupChannel 主要是完成了 Unix 域通信的封装，包括处理接收的消息、发送消息、处理文件描述符传递等，刚才已经分析过，不再具体分析。最后通过在 process 对象中挂载函数和监听事件，使得子进程具有和主进程通信的能力。所有的通信都是基于主进程通过环境变量 NODE_CHANNEL_FD 传递过来的 fd 进行的。

13.5 文件描述符传递

前面我们已经介绍过传递文件描述符的原理，下面我们看看 Node.js 是如何处理文件描述符传递的。

13.5.1 发送文件描述符

我们看进程间通信的发送函数 send 的实现

```
1. process.send = function(message, handle, options, callback) {  
2.   return this._send(message, handle, options, callback);  
3. };  
4.  
5. target._send = function(message, handle, options, callback) {  
6.   // Support legacy function signature  
7.   if (typeof options === 'boolean') {  
8.     options = { swallowErrors: options };  
9.   }  
10.  
11.   let obj;
```

```

12.
13.    // 发送文件描述符, handle 是文件描述符的封装
14.    if (handle) {
15.        message = {
16.            cmd: 'NODE_HANDLE',
17.            type: null,
18.            msg: message
19.        };
20.        // handle 的类型
21.        if (handle instanceof net.Socket) {
22.            message.type = 'net.Socket';
23.        } else if (handle instanceof net.Server) {
24.            message.type = 'net.Server';
25.        } else if (handle instanceof TCP || handle instanceof Pipe
26.        ) {
27.            message.type = 'net.Native';
28.        } else if (handle instanceof dgram.Socket) {
29.            message.type = 'dgram.Socket';
30.        } else if (handle instanceof UDP) {
31.            message.type = 'dgram.Native';
32.        } else {
33.            throw new ERR_INVALID_HANDLE_TYPE();
34.        }
35.        // 根据类型转换对象
36.        obj = handleConversion[message.type];
37.        // 把 JS 层使用的对象转成 C++层对象
38.        handle=handleConversion[message.type].send.call(target,
39.                                            message,
40.                                            handle,
41.                                            options);
42.    }
43.    // 发送
44.    const req = new WriteWrap();
45.    // 发送给对端
46.    const err = writeChannelMessage(channel, req, message, handle);
47.
48. }

```

Node.js 在发送一个封装了文件描述符的对象之前，首先会把 JS 层使用的对象转成 C++层使用的对象。如 TCP

```
1. send(message, server, options) {
```



```

30.                     send_handle_obj).Check();
31.     }
32.
33.     Write(&buf, 1, send_handle, req_wrap_obj);
34. }
```

`Write` 会调用 `Libuv` 的 `uv_write`, `uv_write` 会把 `Libuv` 层的 `handle` 中的 `fd` 取出来, 使用 `sendmsg` 传递到其它进程。整个发送的过程本质是从 `JS` 层到 `Libuv` 层层层揭开要发送的对象, 最后拿到一个文件描述符, 然后通过操作系统提供的 API 把文件描述符传递给另一个进程, 如图 13-4 所示。

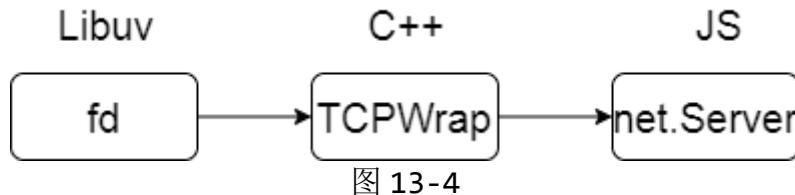


图 13-4

13.5.2 接收文件描述符

分析完发送, 我们再看一下接收的逻辑。前面我们分析过, 当文件描述符收到数据时, 会把文件文件描述符封装成对应的对象。

```

1. void LibuvStreamWrap::OnUvRead(ssize_t nread, const uv_buf_t* buf
) {
2.     HandleScope scope(env()->isolate());
3.     Context::Scope context_scope(env()->context());
4.     uv_handle_type type = UV_UNKNOWN_HANDLE;
5.     // 是否支持传递文件描述符并且有待处理的文件描述符, 则判断文件描述符类
型
6.     if (is_named_pipe_ipc() &&
7.         uv_pipe_pending_count(reinterpret_cast<uv_pipe_t*>(stream()
)) > 0) {
8.         type = uv_pipe_pending_type(reinterpret_cast<uv_pipe_t*>(stre
am()));
9.     }
10.
11.    // 读取成功
12.    if (nread > 0) {
13.        MaybeLocal<Object> pending_obj;
14.        // 根据类型创建一个新的 C++ 对象表示客户端, 并且从服务器中摘下一个
fd 保存到客户端
15.        if (type == UV_TCP) {
16.            pending_obj = AcceptHandle<TCPWrap>(env(), this);
17.        } else if (type == UV_NAMED_PIPE) {
```

```
18.     pending_obj = AcceptHandle<PipeWrap>(env(), this);
19. } else if (type == UV_UDP) {
20.     pending_obj = AcceptHandle<UDPWrap>(env(), this);
21. } else {
22.     CHECK_EQ(type, UV_UNKNOWN_HANDLE);
23. }
24. // 保存到 JS 层使用的对象中，键是 pendingHandle
25. if (!pending_obj.IsEmpty()) {
26.     object()
27.         ->Set(env()->context(),
28.                 env()->pending_handle_string(),
29.                 pending_objToLocalChecked())
30.         .Check();
31. }
32. }
33.
34. EmitRead(nread, *buf);
35. }
```

接着我们看看 JS 层的处理。

```
1. channel.onread = function(arrayBuffer) {
2.     // 收到的文件描述符
3.     const recvHandle = channel.pendingHandle;
4.     channel.pendingHandle = null;
5.     if (arrayBuffer) {
6.         const nread = streamBaseState[kReadBytesOrError];
7.         const offset = streamBaseState[kArrayBufferOffset];
8.         const pool = new Uint8Array(arrayBuffer, offset, nread);
9.         if (recvHandle)
10.             pendingHandle = recvHandle;
11.         // 解析收到的消息
12.         for (const message of parseChannelMessages(channel, pool)) {
13.             if (isInternal(message)) {
14.                 if (message.cmd === 'NODE_HANDLE') {
15.                     handleMessage(message, pendingHandle, true);
16.                     pendingHandle = null;
17.                 } else {
18.                     handleMessage(message, undefined, true);
19.                 }
20.             } else {
21.                 handleMessage(message, undefined, false);
22.             }
23.         }
24.     }
```

| 25. };

这里会触发内部事件 internalMessage

```

1. target.on('internalMessage', function(message, handle) {
2.   // 是否收到了 handle
3.   if (message.cmd !== 'NODE_HANDLE') return;
4.
5.   // 成功收到，发送 ACK
6.   target._send({ cmd: 'NODE_HANDLE_ACK' }, null, true);
7.
8.   const obj = handleConversion[message.type];
9.
10.  /*
11.   C++对象转成 JS 层使用的对象。转完之后再根据里层的字段
12.   message.msg 进一步处理，或者触发 message 事件传给用户
13.  */
14.  obj.got.call(this, message, handle, (handle) => {
15.    handleMessage(message.msg, handle, isInternal(message.msg));
16.  });
16. })
```

我们看到这里会把 C++ 层的对象转成 JS 层使用的对象。如 TCP

```

1. got(message, handle, emit) {
2.   const server = new net.Server();
3.   server.listen(handle, () => {
4.     emit(server);
5.   });
6. }
```

这就是文件描述符传递在 Node.js 中的处理流程，传递文件描述符是一个非常有用的能力，比如一个进程可以把一个 TCP 连接所对应的文件描述符直接发送给另一个进程处理。这也是 cluster 模块的原理。后续我们会看到。在 Node.js 中，整体的处理流程就是，发送的时候把一个 JS 层使用的对象一层层地剥开，变成 C++ 对象，然后再变成 fd，最后通过底层 API 传递给另一个进程。接收的时候就是把一个 fd 一层层地包裹，变成一个 JS 层使用的对象。

第十四章 多线程

线程是操作系统的最小调度单位，它本质上是进程中的一个执行流，我们知道，进程有代码段，线程其实就是进程代码段中的其中一段代码。线程的一种实现是作为进程来实现的

(pthread 线程库)，通过调用 clone，新建一个进程，然后执行父进程代码段里的一个代码片段，其中文件描述符、内存等信息都是共享的。因为内存是共享的，所以线程不能共享栈，否则访问栈的地址的时候，会映射到相同的物理地址，那样就会互相影响，所以每个线程会有自己独立的栈。在调用 clone 函数的时候会设置栈的范围，比如在堆上分配一块内存用于做线程的栈，并且支持设置子线程和主线程共享哪些资源。具体可以参考 clone 系统调用。

由于 Node.js 是单线程的，虽然底层的 Libuv 实现了一个线程池，但是这个线程池只能执行 C、C++ 层定义的任务。如果我们想自定义一些耗时的操作，那就只能在 C++ 层处理，然后暴露接口给 JS 层调用，这个成本是非常高的，在早期的 Node.js 版本里，我们可以用进程去实现这样的需求。但是进程太重了，在新版的 Node.js 中，Node.js 为我们提供了多线程的功能。这一章以 Node.js 多线程模块为背景，分析 Node.js 中多线程的原理，但是不分析 Libuv 的线程实现，它本质是对线程库的简单封装。Node.js 中，线程的实现也非常复杂。虽然底层只是对线程库的封装，但是把它和 Node.js 原本的架构结合起来变得复杂起来。

14.1 使用多线程

对于同步文件操作、DNS 解析等操作，Node.js 使用了内置的线程池支持了异步。但是一些加解密、字符串运算、阻塞型 API 等操作。我们就不能在主线程里处理了，这时候就不得不使用线程，而且多线程还能利用多核的能力。Node.js 的子线程本质上是一个新的事件循环，但是子线程和 Node.js 主线程共享一个 Libuv 线程池，所以如果在子线程里有文件、DNS 等操作就会和主线程竞争 Libuv 线程池。如图 14-1 所示。

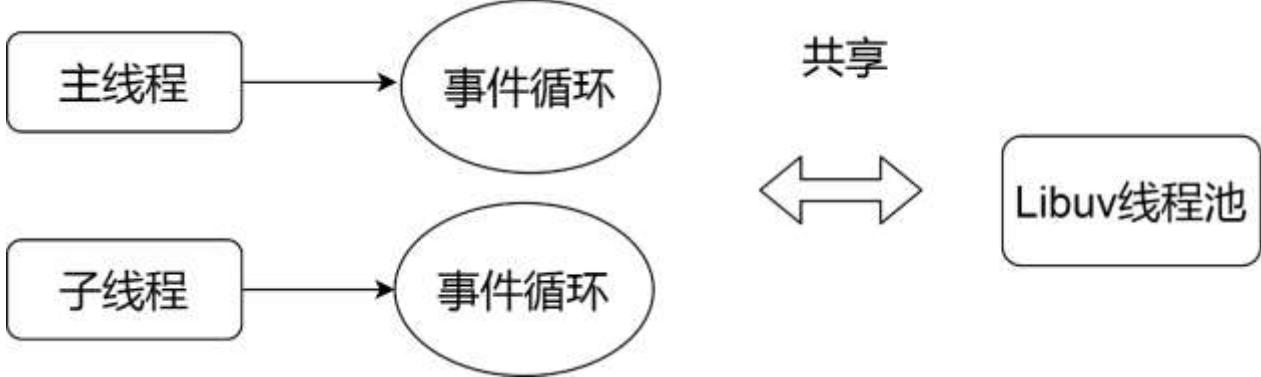


图 14-1

我们看一下在 Node.js 中如何使用线程。

```
1. const { Worker, isMainThread, parentPort } = require('worker_threads');
2. if (isMainThread) {
3.   const worker = new Worker(__filename);
4.   worker.once('message', (message) => {
5.     ...
6.   });
}
```

```

7.     worker.postMessage('Hello, world!');
8. } else {
9.     // 做点耗时的事情
10.    parentPort.once('message', (message) => {
11.        parentPort.postMessage(message);
12.    });
13.

```

上面这段代码会被执行两次，一次是在主线程，一次在子线程。所以首先通过 `isMainThread` 判断当前是主线程还是子线程。主线程的话，就创建一个子线程，然后监听子线程发过来的消息。子线程的话，首先执行业务相关的代码，还可以监听主线程传过来的消息。我们在子线程中可以做一些耗时或者阻塞性的操作，不会影响主线程的执行。我们也可以把这两个逻辑拆分到两个文件。

主线程

```

1. const { Worker, isMainThread, parentPort } = require('worker_threads');
2. const worker = new Worker('子线程文件路径');
3. worker.once('message', (message) => {
4.     ...
5. });
6. worker.postMessage('Hello, world!');

```

子线程

```

1. const { Worker, isMainThread, parentPort } = require('worker_threads');
2. parentPort.once('message', (message) => {
3.     parentPort.postMessage(message);
4. });

```

14.2 线程间通信数据结构

进程间的通信一般需要借助操作系统提供公共的内存来完成。因为进程间的内存是独立的，和进程间通信不一样。多线程的内存是共享的，同个进程的内存，多个线程都可以访问，所以线程间通信可以基于进程内的内存来完成。在 Node.js 中，线程间通信使用的是 `MessageChannel` 实现的，它是全双工的，任意一端都可以随时发送信息。

`MessageChannel` 类似 `socket` 通信，它包括两个端点。定义一个 `MessageChannel` 相当于建立一个 TCP 连接，它首先申请两个端点 (`MessagePort`)，然后把它们关联起来。下面我们看一下线程间通信的实现中，比较重要的几个数据结构。

1 Message 代表一个消息。

2 MessagePortData 是对操作 Message 的封装和对消息的承载。

3 MessagePort 是代表通信的端点。

4 MessageChannel 是代表通信的两端，即两个 MessagePort。

下面我们看一下具体的实现。

14.2.1 Message

Message 类代表的是子线程间通信的一条消息。

```
1. class Message : public MemoryRetainer {
2. public:
3.     explicit Message(MallocedBuffer<char>&& payload = MallocedBuffer<char>());
4.     // 是否是最后一条消息，空消息代表是最后一条消息
5.     bool IsCloseMessage() const;
6.     // 线程间通信的数据需要通过序列化和反序列化处理
7.     v8::MaybeLocal<v8::Value> Deserialize(Environment* env,
8.                                         v8::Local<v8::Context> context);
9.     v8::Maybe<bool> Serialize(Environment* env,
10.                               v8::Local<v8::Context> context,
11.                               v8::Local<v8::Value> input,
12.                               const TransferList& transfer_list,
13.                               v8::Local<v8::Object> source_port =
14.                               v8::Local<v8::Object>());
15.
16.     // 传递 SharedArrayBuffer 型变量
17.     void AddSharedArrayBuffer(std::shared_ptr<v8::BackingStore> backing_store);
18.     // 传递 MessagePort 型变量
19.     void AddMessagePort(std::unique_ptr<MessagePortData>&& data);
20.     // 消息所属端口，端口是消息到达的地方
21.     const std::vector<std::unique_ptr<MessagePortData>>& message_ports() const {
22.         return message_ports_;
23.     }
24.
25. private:
26.     // 保存消息的内容
27.     MallocedBuffer<char> main_message_buf_;
28.     std::vector<std::shared_ptr<v8::BackingStore>> array_buffers_;
29.     std::vector<std::shared_ptr<v8::BackingStore>> shared_array_buffers_;
30.     std::vector<std::unique_ptr<MessagePortData>> message_ports_;
31.     std::vector<v8::CompiledWasmModule> wasm_modules_;
32. };
```

14.2.2 MessagePortData

MessagePortData 是管理消息发送和接收的类。

```
1. class MessagePortData : public MemoryRetainer {
```

```

2. public:
3.   explicit MessagePortData(MessagePort* owner);
4.   ~MessagePortData() override;
5.   // 新增一个消息
6.   void AddToIncomingQueue(Message&& message);
7.   // 关联/解关联回两端的端口
8.   static void Entangle(MessagePortData* a, MessagePortData* b);
9.   void Disentangle();
10.
11. private:
12.   // 用于多线程往对端消息队列插入消息时的互斥变量
13.   mutable Mutex mutex_;
14.   std::list<Message> incoming_messages_;
15.   // 所属端口
16.   MessagePort* owner_ = nullptr;
17.   // 用于多线程访问对端 sibling_ 属性时的互斥变量
18.   std::shared_ptr<Mutex> sibling_mutex_ = std::make_shared<Mutex>();
19.   // 指向通信对端的指针
20.   MessagePortData* sibling_ = nullptr;
21. };

```

我们看一下实现。

```

1. MessagePortData::MessagePortData(MessagePort* owner) : owner_(owner) { }
2.
3. MessagePortData::~MessagePortData() {
4.   // 析构时解除和对端的关系
5.   Disentangle();
6. }
7.
8. // 插入一个 message
9. void MessagePortData::AddToIncomingQueue(Message&& message) {
10.   // 先加锁，保证多线程安全，互斥访问
11.   Mutex::ScopedLock lock(mutex_);
12.   // 插入消息队列
13.   incoming_messages_.emplace_back(std::move(message));
14.   // 通知 owner
15.   if (owner_ != nullptr) {
16.     owner_->TriggerAsync();
17.   }
18. }
19.
20. // 关联回的对端，并保持对端的互斥变量，访问对端时需要使用
21. void MessagePortData::Entangle(MessagePortData* a, MessagePortData* b) {
22.   a->sibling_ = b;
23.   b->sibling_ = a;
24.   a->sibling_mutex_ = b->sibling_mutex_;
25. }
26.
27. // 解除关联
28. void MessagePortData::Disentangle() {
29.   // 加锁操作对端的 sibling 字段
30.   std::shared_ptr<Mutex> sibling_mutex = sibling_mutex_;
31.   Mutex::ScopedLock sibling_lock(*sibling_mutex);
32.   sibling_mutex_ = std::make_shared<Mutex>();
33.   // 对端
34.   MessagePortData* sibling = sibling_;
35.   // 对端非空，则把对端的 sibling 也指向空，自己也指向空
36.   if (sibling_ != nullptr) {

```

```
37.     sibling_> sibling_ = nullptr;
38.     sibling_ = nullptr;
39. }
40.
41. // 插入一个空的消息通知对端和本端
42. AddToIncomingQueue(Message());
43. if (sibling != nullptr) {
44.     sibling->AddToIncomingQueue(Message());
45. }
46. }
```

14.2.3 MessagePort

MessagePort 表示的是通信的一端。

```
1. class MessagePort : public HandleWrap {
2. public:
3.     MessagePort(Environment* env,
4.                  v8::Local<v8::Context> context,
5.                  v8::Local<v8::Object> wrap);
6.     ~MessagePort() override;
7.
8.     static MessagePort* New(Environment* env,
9.                             v8::Local<v8::Context> context,
10.                            std::unique_ptr<MessagePortData> data = nullptr);
11. // 发送消息
12. v8::Maybe<bool> PostMessage(Environment* env,
13.                               v8::Local<v8::Value> message,
14.                               const TransferList& transfer);
15.
16. // 开启/关闭接收消息
17. void Start();
18. void Stop();
19.
20. static void New(const v8::FunctionCallbackInfo<v8::Value>& args);
21. // 提供 JS 层使用的方法
22. static void PostMessage(const v8::FunctionCallbackInfo<v8::Value>& args);
23. static void Start(const v8::FunctionCallbackInfo<v8::Value>& args);
24. static void Stop(const v8::FunctionCallbackInfo<v8::Value>& args);
25. static void Drain(const v8::FunctionCallbackInfo<v8::Value>& args);
26. static void ReceiveMessage(const v8::FunctionCallbackInfo<v8::Value>& args);
27. // 关联对端
28. static void Entangle(MessagePort* a, MessagePort* b);
29. static void Entangle(MessagePort* a, MessagePortData* b);
30.
31. // 解除 MessagePortData 和端口的关系
32. std::unique_ptr<MessagePortData> Detach();
33. // 关闭端口
34. void Close(
35.     v8::Local<v8::Value> close_callback = v8::Local<v8::Value>() override;
36.
37. inline bool IsDetached() const;
38. private:
39.     void OnClose() override;
40.     void OnMessage();
41.     void TriggerAsync();
42.     v8::MaybeLocal<v8::Value> ReceiveMessage(v8::Local<v8::Context> context,
43.                                              bool only_if_receiving);
44. // MessagePortData 用于管理消息的发送和接收
```

```

45. std::unique_ptr<MessagePortData> data_ = nullptr;
46. // 是否开启接收消息标记
47. bool receiving_messages_ = false;
48. // 用于收到消息时通知事件循环，事件循环执行回调处理消息
49. uv_async_t async_;
50. };

```

我们看一下实现，只列出部分函数。

```

1. // 端口是否不接收消息了
2. bool MessagePort::IsDetached() const {
3.     return data_ == nullptr || IsHandleClosing();
4. }
5.
6. // 有消息到达，通知事件循环执行回调
7. void MessagePort::TriggerAsync() {
8.     if (IsHandleClosing()) return;
9.     CHECK_EQ(uv_async_send(&async_), 0);
10. }
11.
12. // 关闭接收消息的端口
13. void MessagePort::Close(v8::Local<v8::Value> close_callback) {
14.     if (data_) {
15.         // 持有锁，防止再接收消息
16.         Mutex::ScopedLock sibling_lock(data_->mutex_);
17.         HandleWrap::Close(close_callback);
18.     } else {
19.         HandleWrap::Close(close_callback);
20.     }
21. }
22.
23. // 新建一个端口，并且可以挂载一个 MessagePortData
24. MessagePort* MessagePort::New(
25.     Environment* env,
26.     Local<Context> context,
27.     std::unique_ptr<MessagePortData> data) {
28.     Context::Scope context_scope(context);
29.     Local<FunctionTemplate> ctor_temp = GetMessagePortConstructorTemplate(env);
30.
31.     Local<Object> instance;
32.     // JS 层使用的对象
33.     if (!ctor_temp->InstanceTemplate()->NewInstance(context).ToLocal(&instance))
34.         return nullptr;
35.     // 新建一个消息端口
36.     MessagePort* port = new MessagePort(env, context, instance);
37.
38.     // 需要挂载 MessagePortData
39.     if (data) {
40.         port->Detach();
41.         port->data_ = std::move(data);
42.         Mutex::ScopedLock lock(port->data_->mutex_);
43.         // 修改 data 的 owner 为当前消息端口
44.         port->data_->owner_ = port;
45.         // data 中可能有消息
46.         port->TriggerAsync();
47.     }
48.     return port;
49. }
50.

```

```
51. // 开始接收消息
52. void MessagePort::Start() {
53.     Debug(this, "Start receiving messages");
54.     receiving_messages_ = true;
55.     Mutex::ScopedLock lock(data_->mutex_);
56.     // 有缓存的消息，通知上层
57.     if (!data_->incoming_messages_.empty())
58.         TriggerAsync();
59. }
60.
61. // 停止接收消息
62. void MessagePort::Stop() {
63.     Debug(this, "Stop receiving messages");
64.     receiving_messages_ = false;
65. }
66. // JS 层调用
67. void MessagePort::Start(const FunctionCallbackInfo<Value>& args) {
68.     MessagePort* port;
69.     ASSIGN_OR_RETURN_UNWRAP(&port, args.This());
70.     if (!port->data_) {
71.         return;
72.     }
73.     port->Start();
74. }
75.
76. void MessagePort::Stop(const FunctionCallbackInfo<Value>& args) {
77.     MessagePort* port;
78.     CHECK(args[0]->IsObject());
79.     ASSIGN_OR_RETURN_UNWRAP(&port, args[0].As<Object>());
80.     if (!port->data_) {
81.         return;
82.     }
83.     port->Stop();
84. }
85.
86. // 读取消息
87. void MessagePort::Drain(const FunctionCallbackInfo<Value>& args) {
88.     MessagePort* port;
89.     ASSIGN_OR_RETURN_UNWRAP(&port, args[0].As<Object>());
90.     port->OnMessage();
91. }
92.
93. // 获取某个端口的消息
94. void MessagePort::ReceiveMessage(const FunctionCallbackInfo<Value>& args) {
95.     CHECK(args[0]->IsObject());
96.     // 第一个参数是端口
97.     MessagePort* port = Unwrap<MessagePort>(args[0].As<Object>());
98.     // 调用对象的 ReceiverMessage 方法
99.     MaybeLocal<Value> payload =
100.         port->ReceiveMessage(port->object()->CreationContext(), false);
101.     if (!payload.IsEmpty())
102.         args.GetReturnValue().Set(payloadToLocalChecked());
103. }
104.
105. // 关联两个端口
106. void MessagePort::Entangle(MessagePort* a, MessagePort* b) {
107.     Entangle(a, b->data_.get());
108. }
109.
110. void MessagePort::Entangle(MessagePort* a, MessagePortData* b) {
111.     MessagePortData::Entangle(a->data_.get(), b);
```

```
112. }
```

14.2.4 MessageChannel

MessageChannel 表示线程间通信的两个端。

```
1. static void MessageChannel(const FunctionCallbackInfo<Value>& args) {
2.     Environment* env = Environment::GetCurrent(args);
3.
4.     Local<Context> context = args.This()->CreationContext();
5.     Context::Scope context_scope(context);
6.
7.     MessagePort* port1 = MessagePort::New(env, context);
8.     MessagePort* port2 = MessagePort::New(env, context);
9.     MessagePort::Entangle(port1, port2);
10.    // port1->object()拿到 JS 层使用的对象，它关联了 MessagePort 对象
11.    args.This()->Set(context, env->port1_string(), port1->object())
12.        .Check();
13.    args.This()->Set(context, env->port2_string(), port2->object())
14.        .Check();
15. }
```

MessageChannel 的逻辑比较简单，新建两个消息端口，并且关联起来，后续就可以基于这两个端口进行通信了。

Message、MessagePortData、MessagePort 和 MessageChannel 的关系图如图 14-2 所示。

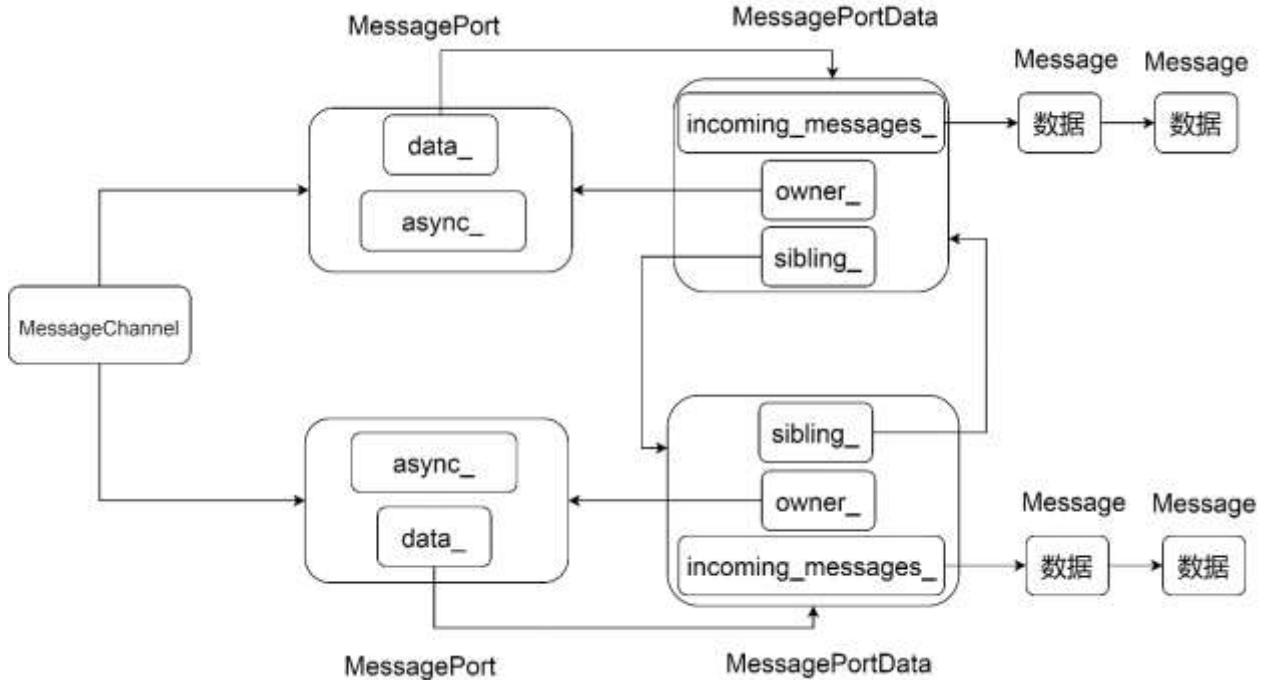


图 14-2

最后我们看一下线程间通信模块导出的一些功能。

```
1. static void InitMessaging(Local<Object> target,
```

```
2.                     Local<Value> unused,
3.                     Local<Context> context,
4.                     void* priv) {
5.     Environment* env = Environment::GetCurrent(context);
6.
7.     {
8.         // 线程间通信的通道
9.         Local<String> message_channel_string = FIXED_ONE_BYTE_STRING(env->isolate(),
10.                                         "MessageChannel");
11.         Local<FunctionTemplate> templ = env->NewFunctionTemplate(MessageChannel);
12.         templ->SetClassName(message_channel_string);
13.         target->Set(context,
14.                         message_channel_string,
15.                         templ->GetFunction(context).ToLocalChecked()).Check();
16.     }
17.     // 新建消息端口的构造函数
18.     target->Set(context,
19.                     env->message_port_constructor_string(),
20.                     GetMessagePortConstructorTemplate(env)
21.                         ->GetFunction(context).ToLocalChecked()).Check();
22.
23.     env->SetMethod(target, "stopMessagePort", MessagePort::Stop);
24.     env->SetMethod(target, "drainMessagePort", MessagePort::Drain);
25.     env->SetMethod(target, "receiveMessageOnPort", MessagePort::ReceiveMessage);
26.     env->SetMethod(target, "moveMessagePortToContext",
27.                     MessagePort::MoveToContext);
28. }
```

14.3 多线程的实现

本节我们从 worker_threads 模块开始分析多线程的实现。这是一个 C++ 模块。我们看一下它导出的功能。require("work_threads") 的时候就是引用了 InitWorker 函数导出的功能。

```
1. void InitWorker(Local<Object> target,
2.                 Local<Value> unused,
3.                 Local<Context> context,
4.                 void* priv) {
5.     Environment* env = Environment::GetCurrent(context);
6.
7.     {
8.         Local<FunctionTemplate> w = env->NewFunctionTemplate(Worker::New);
9.         w->InstanceTemplate()->SetInternalFieldCount(1);
10.        w->Inherit(AsyncWrap::GetConstructorTemplate(env));
11.        // 设置一系列原型方法，就不一一列举
12.        env->SetProtoMethod(w, "setEnvVars", Worker::SetEnvVars);
13.        // 一系列原型方法
14.        /*
15.            导出函数模块对应的函数，即我们代码中
16.            const { Worker } = require("worker_threads"); 中的 Worker
17.        */
18.        Local<String> workerString = FIXED_ONE_BYTE_STRING(env->isolate(), "Worker");
19.        w->SetClassName(workerString);
20.        target->Set(env->context(),
21.                      workerString,
22.                      w->GetFunction(env->context()).ToLocalChecked()).Check();
23. }
```

```

24.      /*
25.       导出 getEnvMessagePort 方法, 获取线程接收消息的端口
26.       const {getEnvMessagePort} = require("worker_threads");
27.      */
28.      env->SetMethod(target, "getEnvMessagePort", GetEnvMessagePort);
29.      /*
30.       线程 id, 这个不是操作系统分配的那个, 而是 Node.js 分配的,
31.       在创建线程的时候设置
32.       const { threadId } = require("worker_threads");
33.      */
34.      target->Set(env->context(),
35.                  env->thread_id_string(),
36.                  Number::New(env->isolate(),
37.                             static_cast<double>(env->thread_id())))
38.                  .Check();
39.      /*
40.       是否是主线程,
41.       const { isMainThread } = require("worker_threads");
42.       这边变量在 Node.js 启动的时候设置为 true, 新开子线程的时候, 没有设
43.       置, 所以是 false
44.      */
45.      target->Set(env->context(),
46.                  FIXED_ONE_BYTE_STRING(env->isolate(), "isMainThread"),
47.                  Boolean::New(env->isolate(), env->is_main_thread()))
48.                  .Check();
49.      /*
50.       如果不是主线程, 导出资源限制的配置,
51.       即在子线程中调用
52.       const { resourceLimits } = require("worker_threads");
53.      */
54.      if (!env->is_main_thread()) {
55.          target->Set(env->context(),
56.                      FIXED_ONE_BYTE_STRING(env->isolate(),
57.                                              "resourceLimits"),
58.                      env->worker_context()->GetResourceLimits(env->isolate())).Check();
59.      }
60.      // 导出几个常量
61.      NODE_DEFINE_CONSTANT(target, kMaxYoungGenerationSizeMb);
62.      NODE_DEFINE_CONSTANT(target, kMaxOldGenerationSizeMb);
63.      NODE_DEFINE_CONSTANT(target, kCodeRangeSizeMb);
64.      NODE_DEFINE_CONSTANT(target, kTotalResourceLimitCount);
65.  }

```

了解 work_threads 模块导出的功能后，我们看在 JS 层执行 new Worker 的时候的逻辑。根据上面代码导出的逻辑，我们知道这时候首先会新建一个 C++ 对象。然后执行 New 回调，并传入新建的 C++ 对象。我们看 New 函数的逻辑。我们省略一系列的参数处理，主要代码如下。

```

1. // args.This() 就是我们刚才传进来的 this
2. Worker* worker = new Worker(env, args.This(),
3.                               url, per_isolate_opts,
4.                               std::move(exec_argv_out));

```

我们再看 Worker 类的声明。

```
1.  class Worker : public AsyncWrap {
2.  public:
3.      // 函数声明
4.
5.  private:
6.
7.      std::shared_ptr<PerIsolateOptions> per_isolate_opts_;
8.      std::vector<std::string> exec_argv_;
9.      std::vector<std::string> argv_;
10.     MultiIsolatePlatform* platform_;
11.     v8::Isolate* isolate_ = nullptr;
12.     bool start_profiler_idle_notifier_;
13.     // 真正的线程 id, 底层返回的
14.     uv_thread_t tid_;
15.
16.     // This mutex protects access to all variables listed below it.
17.     mutable Mutex mutex_;
18.
19.     bool thread_joined_ = true;
20.     const char* custom_error_ = nullptr;
21.     int exit_code_ = 0;
22.     // 线程 id, Node.js 分配, 不是底层返回的
23.     uint64_t thread_id_ = -1;
24.     uintptr_t stack_base_ = 0;
25.
26.     // 线程资源限制配置
27.     double resource_limits_[kTotalResourceLimitCount];
28.     void UpdateResourceConstraints(v8::ResourceConstraints* constraints);
29.
30.     // 栈信息
31.     static constexpr size_t kStackSize = 4 * 1024 * 1024;
32.     static constexpr size_t kStackBufferSize = 192 * 1024;
33.
34.     std::unique_ptr<MessagePortData> child_port_data_;
35.     std::shared_ptr<KVStore> env_vars_;
36.     // 用于线程间通信
37.     MessagePort* child_port_ = nullptr;
38.     MessagePort* parent_port_ = nullptr;
39.     // 线程状态
40.     bool stopped_ = true;
41.     // 是否影响事件循环退出
42.     bool has_ref_ = true;
43.     // 子线程执行时的环境变量, 基类也定义了
44.     Environment* env_ = nullptr;
45. }
```

这里只讲一下 `env_` 的定义, 因为这是一个非常重要的地方。我们看到 `Worker` 类继承 `AsyncWrap`, `AsyncWrap` 继承了 `BaseObject`。`BaseObject` 中也定义了 `env_` 属性。我们看一下在 C++ 中如果子类父类都定义了一个属性时是怎样的。我们来看一个例子

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  class A
5.  {
6.  public:
7.      int value;
8.      A()
9.  {
```

```

10.         value=1;
11.     }
12.     void console()
13.     {
14.         cout<<value<<endl;
15.     }
16.
17. };
18. class B: public A
19. {
20.     public:
21.         int value;
22.     B():A()
23.     {
24.         value=2;
25.     }
26. };
27. int main()
28. {
29.     B b;
30.     // b.value = 3;只会修改子类的，不会修改父类的
31.     b.console();
32.     cout<<b.value<<endl<<"内存大小: "<<sizeof(b)<<endl;
33.     return 0;
34. }
```

以上代码执行时输出

```

1. 1
2. 2
3. 内存大小: 8
```

由输出结果我们可以知道，`b` 内存大小是 8 个字节。即两个 `int`。所以 `b` 的内存布局中两个 `a` 属性都分配了内存。当我们通过 `b.console` 输出 `value` 时，因为 `console` 是在 `A` 上定义的，所以输出 1，但是我们通过 `b.value` 访问时，输出的是 2。因为访问的是 `B` 中定义的 `value`，同理如果我们在 `B` 中定义 `console`，输出也会是 2。`Worker` 中定义的 `env_` 我们后续会看到它的作用。接着我们看一下 `Worker` 类的初始化逻辑。

```

1. Worker::Worker(Environment* env,
2.                 Local<Object> wrap,...)
3.     : AsyncWrap(env, wrap, AsyncWrap::PROVIDER_WORKER),
4.     ...
5.     // 分配线程 id
6.     thread_id_(Environment::AllocateThreadId()),
7.     // 继承主线程的环境变量
8.     env_vars_(env->env_vars()) {
9.
10.    // 新建一个端口和子线程通信
11.    parent_port_ = MessagePort::New(env, env->context());
12.    /*
13.        关联起来，用于通信
14.        const parent_port_ = {data: {sibling: null}};
15.        const child_port_data_ = {sibling: null};
16.        parent_port_.data.sibling = child_port_data_;
17.        child_port_data_.sibling = parent_port_.data;
18.    */
19.    child_port_data_ = std::make_unique<MessagePortData>(nullptr);
20.    MessagePort::Entangle(parent_port_, child_port_data_.get());
21.    // 设置 JS 层 Worker 对象的 messagePort 属性为 parent_port_
22.    object()->Set(env->context(),
```

```
23.         env->message_port_string(),
24.         parent_port_->object()).Check();
25. // 设置 Worker 对象的线程 id, 即 threadId 属性
26. object()->Set(env->context(),
27.                 env->thread_id_string(),
28.                 Number::New(env->isolate(), static_cast<double>(thread_id_)))
29.     .Check();
30. }
```

新建一个 Worker，结构如图 14-3 所示。

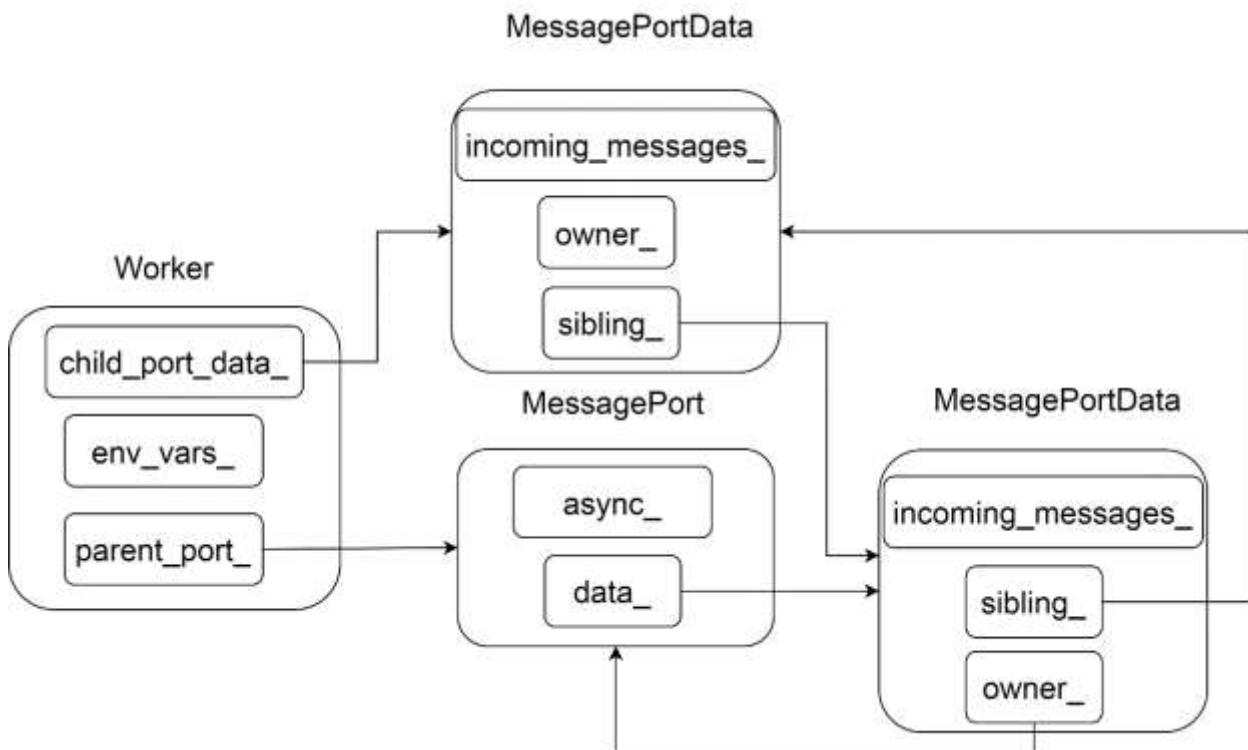


图 14-3

了解了 new Worker 的逻辑后，我们看在 JS 层是如何使用的。我们看 JS 层 Worker 类的构造函数。

```
1. constructor(filename, options = {}) {
2.     super();
3.     // 忽略一系列参数处理, new Worker 就是上面提到的 C++ 层的
4.     this[kHandle] = new Worker(url, options.execArgv, parseResourceLimits(options.resourceLimits));
5.     // messagePort 指向 parent_port
6.     this[kPort] = this[kHandle].messagePort;
7.     this[kPort].on('message', (data) => this[kOnMessage](data));
8.     // 开始接收消息
9.     this[kPort].start();
```

```

10.     // 申请一个通信通道，两个端口
11.     const { port1, port2 } = new MessageChannel();
12.     this[kPublicPort] = port1;
13.     this[kPublicPort].on('message', (message) => this.emit('message', message));
14.     // 向另一端发送消息
15.     this[kPort].postMessage({
16.       argv,
17.       type: messageTypes.LOAD_SCRIPT,
18.       filename,
19.       doEval: !!options.eval,
20.       cwdCounter: cwdCounter || workerIo.sharedCwdCounter,
21.       workerData: options.workerData,
22.       publicPort: port2,
23.       manifestSrc: getOptionValue('--experimental-policy') ?
24.         require('internal/process/policy').src :
25.         null,
26.       hasStdin: !!options.stdin
27.     }, [port2]);
28.     // 开启线程
29.     this[kHandle].startThread();
30.   }

```

上面的代码主要逻辑如下

- 1 保存 messagePort，监听该端口的 message 事件，然后给 messagePort 的对端发送消息，但是这时候还没有接收端口，所以消息会缓存到 MessagePortData，即 child_port_data_ 中。另外我们看到主线程把通信端口 port2 发送给了子线程。
- 2 申请一个通信通道 port1 和 port2，用于主线程和子线程通信。`_parent_port` 和 `child_port` 是给 Node.js 使用的，新申请的端口是给用户使用的。
- 3 创建子线程。

我们看创建线程的时候，做了什么。

```

1. void Worker::StartThread(const FunctionCallbackInfo<Value>& args) {
2.   Worker* w;
3.   ASSIGN_OR_RETURN_UNWRAP(&w, args.This());
4.   Mutex::ScopedLock lock(w->mutex_);
5.
6.   // The object now owns the created thread and should not be garbage collected
7.   // until that finishes.
8.   w->ClearWeak();
9.   // 加入主线程维护的子线程数据结构
10.  w->env()->add_sub_worker_context(w);
11.  w->stopped_ = false;
12.  w->thread_joined_ = false;
13.  // 是否需要阻塞事件循环退出，默认 true
14.  if (w->has_ref_)
15.    w->env()->add_refs(1);

```

```
16.     // 是否需要栈和栈大小
17.     uv_thread_options_t thread_options;
18.     thread_options.flags = UV_THREAD_HAS_STACK_SIZE;
19.     thread_options.stack_size = kStackSize;
20.     // 创建线程
21.     CHECK_EQ(uv_thread_create_ex(&w->tid_, &thread_options, [](void* arg) {
22.
23.         Worker* w = static_cast<Worker*>(arg);
24.         const uintptr_t stack_top = reinterpret_cast<uintptr_t>(&arg);
25.         w->stack_base_ = stack_top - (kStackSize - kStackBufferSize);
26.         // 执行主逻辑
27.         w->Run();
28.
29.         Mutex::ScopedLock lock(w->mutex_);
30.         // 给主线程提交一个任务，通知主线程子线程执行完毕，因为主线程不能直接执行 join 阻塞自己
31.         w->env()->SetImmediateThreadsafe(
32.             [w = std::unique_ptr<Worker>(w)](Environment* env) {
33.                 if (w->has_ref_)
34.                     env->add_refs(-1);
35.                 w->JoinThread();
36.                 // implicitly delete w
37.             });
38.         }, static_cast<void*>(w)), 0);
39.     }
```

StartThread 新建了一个子线程，然后在子线程中执行 Run，我们继续看 Run

```
1.     void Worker::Run() {
2.         // 线程执行所需要的数据结构，比如 loop, isolate，和主线程独立
3.         WorkerThreadData data(this);
4.
5.         {
6.             Locker locker(isolate_);
7.             Isolate::Scope isolate_scope(isolate_);
8.             SealHandleScope outer_seal(isolate_);
9.             // std::unique_ptr<Environment, FreeEnvironment> env_;
10.            DeleteFnPtr<Environment, FreeEnvironment> env_;
11.            // 线程执行完后执行的清除函数
12.            auto cleanup_env = OnScopeLeave([&]() {
13.                // ...
14.            });
15.
16.            {
17.                HandleScope handle_scope(isolate_);
18.                Local<Context> context;
19.                // 新建一个 context，和主线程独立
20.                context = NewContext(isolate_);
21.                Context::Scope context_scope(context);
22.
23.                // 新建一个 env 并初始化，env 中会和新的 context 关联
24.                env_.reset(new Environment(data.isolate_data_.get(),
25.                                           context,
26.                                           std::move(argv_),
27.                                           std::move(exec_argv_),
28.                                           Environment::kNoFlags,
29.                                           thread_id_));
30.                env_->set_env_vars(std::move(env_vars_));
31.                env_->set_abort_on_uncaught_exception(false);
32.                env_->set_worker_context(this);
33.            }
```

```
34.         env_->InitializeLibuv(start_profiler_idle_notifier_);
35.     }
36.     {
37.         Mutex::ScopedLock lock(mutex_);
38.         // 更新子线程所属的 env
39.         this->env_ = env_.get();
40.     }
41.
42.     {
43.         if (!env_->RunBootstrapping().IsEmpty()) {
44.             CreateEnvMessagePort(env_.get());
45.             USE(StartExecution(env_.get(), "internal/main/worker_thread"));
46.         }
47.     }
48.
49.     {
50.         SealHandleScope seal(isolate_);
51.         bool more;
52.         // 开始事件循环
53.         do {
54.             if (is_stopped()) break;
55.             uv_run(&data.loop_, UV_RUN_DEFAULT);
56.             if (is_stopped()) break;
57.
58.             platform_->DrainTasks(isolate_);
59.
60.             more = uv_loop_alive(&data.loop_);
61.             if (more && !is_stopped()) continue;
62.
63.             EmitBeforeExit(env_.get());
64.
65.             more = uv_loop_alive(&data.loop_);
66.         } while (more == true && !is_stopped());
67.     }
68. }
69. }
```

我们分步骤分析上面的代码

1 新建 Isolate、context 和 Environment，子线程在独立的环境执行。然后初始化 Environment。这个在 Node.js 启动过程章节已经分析过，不再分析。

2 更新子线程的 env_。刚才已经分析过，Worker 类中定义了 env_ 属性，所以这里通过 this.env_ 更新时，是不会影响基类（BaseObject）中的值的。因为子线程是在新的环境执行的，所以在新环境中使用该 Worker 实例时，需要使用新的环境变量。而在主线程使用该 Worker 实例时，是通过 BaseObject 的 env() 访问的。从而获取的是主线程的环境。因为 Worker 实例是在主线程和子线程之间共享的，Node.js 在 Worker 类中重新定义了一个 env 属性正是为了解决这个问题。

3 CreateEnvMessagePort

```
4.                                     std::move(child_port_data_));  
5.     if (child_port_ != nullptr)  
6.         env->set_message_port(child_port_->object(isolate_));  
7. }
```

child_port_data_这个变量刚才我们已经看到过，在这里首先申请一个新的端口。并且和 child_port_data_互相关联起来。然后在 env 缓存起来。后续会使用。这时候的关系图如图 14-4 所示。

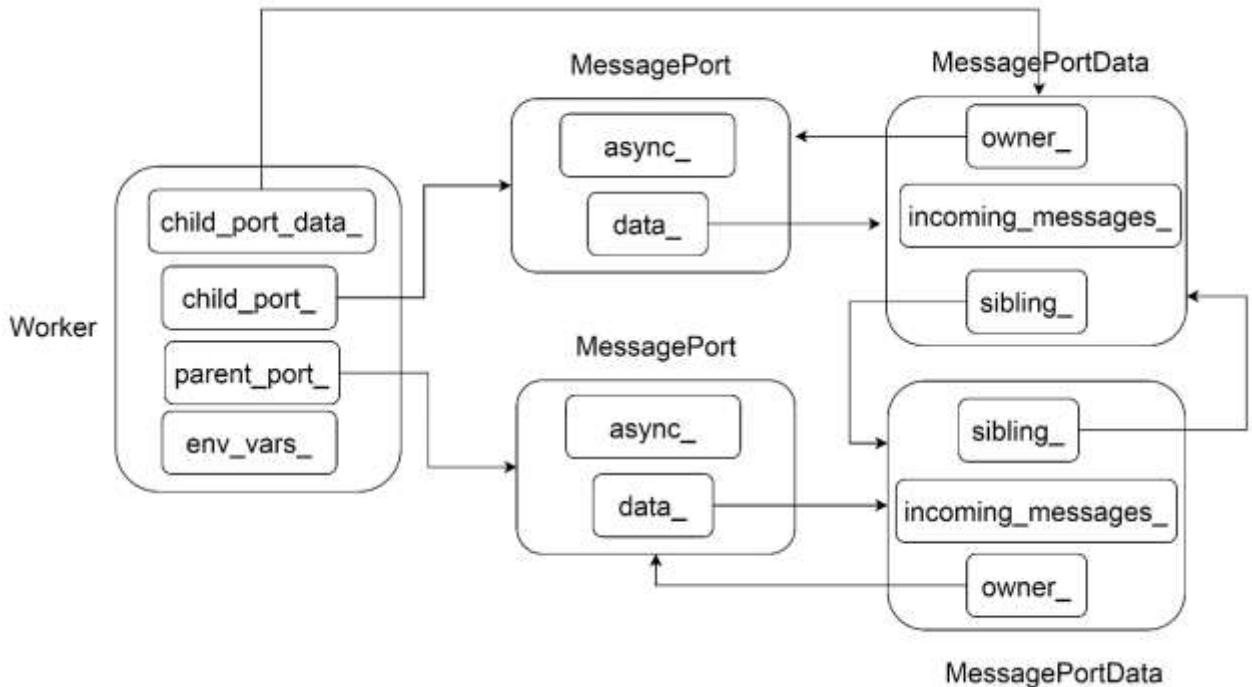


图 14-4

4 执行 internal/main/worker_thread.js

```
1. // 设置 process 对象  
2. patchProcessObject();  
3. // 获取刚才缓存的端口 child_port_  
4. const port = getEnvMessagePort();  
5. port.on('message', (message) => {  
6.     // 加载脚本  
7.     if (message.type === LOAD_SCRIPT) {  
8.         const {  
9.             argv,  
10.             cwdCounter,  
11.             filename,  
12.             doEval,
```

```

13.         workerData,
14.         publicPort,
15.         manifestSrc,
16.         manifestURL,
17.         hasStdin
18.     } = message;
19.
20.     const CJSLoader = require('internal/modules/cjs/loader');
21.     loadPreloadModules();
22.     /*
23.         由主线程申请的 MessageChannel 中某一端的端口,
24.         主线程传递过来的, 保存用于和主线程通信
25.     */
26.     publicWorker.parentPort = publicPort;
27.     // 执行时使用的数据
28.     publicWorker.workerData = workerData;
29.     // 通知主线程, 正在执行脚本
30.     port.postMessage({ type: UP_AND_RUNNING });
31.     // 执行 new Worker(filename) 时传入的文件
32.     CJSLoader.Module.runMain(filename);
33. }
34. // 开始接收消息
35. port.start()

```

我们看到 worker_thread.js 中通过 runMain 完成了子线程的代码执行，然后开始事件循环。

我们看一下当事件循环结束时，Node.js 的逻辑。

```

1. // 给主线程提交一个任务, 通知主线程子线程执行完毕, 因为主线程不能直接执行 join 阻塞自己
2. w->env()->SetImmediateThreadsafe(
3.     [w = std::unique_ptr<Worker>(w)](Environment* env) {
4.         if (w->has_ref_)
5.             env->add_refs(-1);
6.         w->JoinThread();
7.         // implicitly delete w
8.     });
9. }, static_cast<void*>(w)), 0);

```

通过 w->env() 获取的是主线程的执行环境。我们看一下 SetImmediateThreadsafe。

```

1. template <typename Fn>
2. void Environment::SetImmediateThreadsafe(Fn&& cb) {
3.     auto callback = std::make_unique<NativeImmediateCallbackImpl<Fn>>(
4.         std::move(cb), false);
5.     {
6.         Mutex::ScopedLock lock(native_immediates_threadsafe_mutex_);
7.         native_immediates_threadsafe_.Push(std::move(callback));

```

```
8.     }
9.     uv_async_send(&task_queues_async_);
10.    }
```

`SetImmediateThreadsafe` 用于通知执行环境所在的事件循环有异步任务完成。并且是线程安全的。因为可能有多个线程会操作 `native_immediates_threadsafe_`。在主线程事件循环的 `Poll IO` 阶段就会执行 `task_queues_async_` 回调。我们看一下 `task_queues_async_` 对应的回调。

```
1.     uv_async_init(
2.         event_loop(),
3.         &task_queues_async_,
4.         [](uv_async_t* async) {
5.             Environment* env = ContainerOf(
6.                 &Environment::task_queues_async_, async);
7.             env->CleanupFinalizationGroups();
8.             env->RunAndClearNativeImmediates();
9.         });
10.
```

所以在 `Poll IO` 阶段执行的回调是 `RunAndClearNativeImmediates`

```
1.     void Environment::RunAndClearNativeImmediates(bool only_refed) {
2.         TraceEventScope trace_scope(TRACING_CATEGORY_NODE1(environment),
3.                                     "RunAndClearNativeImmediates", this);
4.         size_t ref_count = 0;
5.
6.         if (native_immediates_threadsafe_.size() > 0) {
7.             Mutex::ScopedLock lock(native_immediates_threadsafe_mutex_);
8.             native_immediates_.ConcatMove(std::move(native_immediates_threadsafe_));
9.         }
10.
11.        auto drain_list = [&]() {
12.            TryCatchScope try_catch(this);
13.            DebugSealHandleScope seal_handle_scope(isolate());
14.            while (std::unique_ptr<NativeImmediateCallback> head =
15.                   native_immediates_.Shift()) {
16.                if (head->is_refed())
17.                    ref_count++;
18.
19.                if (head->is_refed() || !only_refed)
20.                    // 执行回调
21.                    head->Call(this);
22.
23.                head.reset();
24.            };
25.        };
26.    }
```

`RunAndClearNativeImmediates` 会执行队列里的回调。对应 `Worker` 的 `JoinThread`

```
1.     void Worker::JoinThread() {
2.         // 阻塞等待子线程结束，执行到这子线程已经结束了
3.         CHECK_EQ(uv_thread_join(&tid_), 0);
4.         thread_joined_ = true;
5.         // 从主线程数据结构中删除该线程对应的实例
6.         env()->remove_sub_worker_context(this);
7.
8.         {
9.             HandleScope handle_scope(env()->isolate());
10.            Context::Scope context_scope(env()->context());
11.        }
12.    }
```

```

12.     // Reset the parent port as we're closing it now anyway.
13.     object()->Set(env()->context(),
14.                     env()->message_port_string(),
15.                     Undefined(env()->isolate())).Check();
16.     // 子线程退出码
17.     Local<Value> args[] = {
18.         Integer::New(env()->isolate(), exit_code_),
19.         custom_error_ != nullptr ?
20.             OneByteString(env()->isolate(), custom_error_).As<Value>() :
21.             Null(env()->isolate()).As<Value>(),
22.     };
23.     // 执行 JS 层回调，触发 exit 事件
24.     MakeCallback(env()->onexit_string(), arraysize(args), args);
25. }
26.

```

最后我们看一下如果结束正在执行的子线程。在 JS 中我能可以通过 terminate 函数终止线程的执行。

```

1.     terminate(callback) {
2.         this[kHandle].stopThread();
3.     }

```

Terminate 是对 C++ 模块 stopThread 的封装。

```

1.     void Worker::StopThread(const FunctionCallbackInfo<Value>& args) {
2.         Worker* w;
3.         ASSIGN_OR_RETURN_UNWRAP(&w, args.This());
4.         w->Exit(1);
5.     }
6.
7.     void Worker::Exit(int code) {
8.         Mutex::ScopedLock lock(mutex_);
9.         // env_ 是子线程执行的 env
10.        if (env_ != nullptr) {
11.            exit_code_ = code;
12.            Stop(env_);
13.        } else {
14.            stopped_ = true;
15.        }
16.    }
17.
18.
19.    int Stop(Environment* env) {
20.        env->ExitEnv();
21.        return 0;
22.    }
23.
24.    void Environment::ExitEnv() {
25.        set_can_call_into_js(false);
26.        set_stopping(true);
27.        isolate_->TerminateExecution();
28.        SetImmediateThreadsafe([](Environment* env) { uv_stop(env->event_loop()); });
29.    }

```

我们看到主线程最终通过 SetImmediateThreadsafe 给子线程所属的 env 提交了一个任务。子线程在 Poll IO 阶段会设置停止事件循环的标记，等到下一次事件循环开始的时候，就会跳出事件循环从而结束子线程的执行。

14.4 线程间通信

本节我们看一下线程间通信的过程。

```
1. const { Worker, isMainThread, parentPort } = require('worker_threads');
2. if (isMainThread) {
3.   const worker = new Worker(__filename);
4.   worker.once('message', (message) => {
5.     ...
6.   });
7.   worker.postMessage('Hello, world!');
8. } else {
9.   // 做点耗时的事情
10.  parentPort.once('message', (message) => {
11.    parentPort.postMessage(message);
12.  });
13. }
```

我们知道 isMainThread 在子线程里是 false，parentPort 就是 messageChannel 中的一端。用于和主线程通信，所以 parentPort.postMessage 给对端发送消息，就是给主线程发送消息，我们再看看 worker.postMessage('Hello, world!')。

```
1. postMessage(...args) {
2.   this[kPublicPort].postMessage(...args);
3. }
```

kPublicPort 指向的就是 messageChannel 的一端。

this[kPublicPort].postMessage(...args) 即给另一端发送消息。我们看一下 postMessage 的实现。

```
1. void MessagePort::PostMessage(const FunctionCallbackInfo<Value>& args) {
2.   Environment* env = Environment::GetCurrent(args);
3.   Local<Object> obj = args.This();
4.   Local<Context> context = obj->CreationContext();
5.
6.   TransferList transfer_list;
7.   if (args[1]->IsObject()) {
8.     // 处理 transfer_list
9.   }
10.  // 拿到 JS 层使用的对象所关联的 MessagePort
```

```

11. MessagePort* port = Unwrap<MessagePort>(args.This());
12.
13. port->PostMessage(env, args[0], transfer_list);
14. }
```

我们接着看 port->PostMessage

```

1. Maybe<bool> MessagePort::PostMessage(Environment* env,
2.                                         Local<Value> message_v,
3.                                         const TransferList& transfer_v) {
4.     Isolate* isolate = env->isolate();
5.     Local<Object> obj = object(isolate);
6.     Local<Context> context = obj->CreationContext();
7.
8.     Message msg;
9.
10.    // 序列化
11.    Maybe<bool> serialization_maybe =
12.        msg.Serialize(env, context, message_v, transfer_v, obj);
13.    // 拿到操作对端 sibling 的锁
14.    Mutex::ScopedLock lock(*data_->sibling_mutex_);
15.
16.    // 把消息插入到对端队列
17.    data_->sibling_->AddToIncomingQueue(std::move(msg));
18.    return Just(true);
19. }
```

PostMessage 通过 AddToIncomingQueue 把消息插入对端的消息队列我们看一下 AddToIncomingQueue

```

1. void MessagePortData::AddToIncomingQueue(Message&& message) {
2.     // 加锁操作消息队列
3.     Mutex::ScopedLock lock(mutex_);
4.     incoming_messages_.emplace_back(std::move(message));
5.     // 通知 owner
6.     if (owner_ != nullptr) {
7.         owner_->TriggerAsync();
8.     }
9. }
```

插入消息队列后，如果有关联的端口，则会通知 Libuv。我们继续看 TriggerAsync。

```

1. void MessagePort::TriggerAsync() {
2.     if (IsHandleClosing()) return;
3.     CHECK_EQ(uv_async_send(&async_), 0);
4. }
```

Libuv 在 Poll IO 阶段就会执行对应的回调。回调是在 new MessagePort 时设置的。

```

1. auto onmessage = [] (uv_async_t* handle) {
2.     MessagePort* channel = ContainerOf(&MessagePort::async_, handle);
3.     channel->OnMessage();
4. };
5. // 初始化 async 结构体，实现异步通信
6. CHECK_EQ(uv_async_init(env->event_loop(),
7.                         &async_,
8.                         onmessage), 0);
```

我们继续看 OnMessage。

```
1. void MessagePort::OnMessage() {
2.     HandleScope handle_scope(env()->isolate());
3.     Local<Context> context = object(env()->isolate())->CreationContext();
4.     // 接收消息条数的阈值
5.     size_t processing_limit;
6.     {
7.         // 加锁操作消息队列
8.         Mutex::ScopedLock(data_->mutex_);
9.         processing_limit = std::max(data_->incoming_messages_.size(),
10.                                     static_cast<size_t>(1000));
11.    }
12.    while (data_) {
13.        // 读取的条数达到阈值，通知 Libuv 下一轮 Poll IO 阶段继续读
14.        if (processing_limit-- == 0) {
15.            // 通知事件循环
16.            TriggerAsync();
17.            return;
18.        }
19.
20.        HandleScope handle_scope(env()->isolate());
21.        Context::Scope context_scope(context);
22.
23.        Local<Value> payload;
24.        // 读取消息
25.        if (!ReceiveMessage(context, true).ToLocal(&payload)) break;
26.        // 没有了
27.        if (payload == env()->no_message_symbol()) break;
28.
29.        Local<Object> event;
30.        Local<Value> cb_args[1];
31.        // 新建一个 MessageEvent 对象，回调 onmessage 事件
32.        if (!env()->message_event_object_template()->NewInstance(context)
33.            .ToLocal(&event) ||
34.            event->Set(context, env()->data_string(), payload).IsNothing() ||
35.            event->Set(context, env()->target_string(), object()).IsNothing() ||
36.            (cb_args[0] = event, false) ||
37.            MakeCallback(env()->onmessage_string(),
38.                         arraysize(cb_args),
39.                         cb_args).IsEmpty()) {
40.            // 如果回调失败，通知 Libuv 下次继续读
41.            if (data_)
42.                TriggerAsync();
43.            return;
44.        }
45.    }
46. }
```

我们看到这里会不断地调用 ReceiveMessage 读取数据，然后回调 JS 层。直到达到阈值或者回调失败。我们看一下 ReceiveMessage 的逻辑。

```
1. MaybeLocal<Value> MessagePort::ReceiveMessage(Local<Context> context,
2.                                                 bool only_if_receiving) {
3.     Message received;
4.     {
5.         // Get the head of the message queue.
6.         // 互斥访问消息队列
7.         Mutex::ScopedLock lock(data_->mutex_);
```

```
8.
9.     bool wants_message = receiving_messages_ || !only_if_receiving;
10.    // 没有消息、不需要接收消息、消息是关闭消息
11.    if (data_->incoming_messages_.empty() ||
12.        (!wants_message &&
13.         !data_->incoming_messages_.front().IsCloseMessage())) {
14.        return env()->no_message_symbol();
15.    }
16.    // 获取队列第一个消息
17.    received = std::move(data_->incoming_messages_.front());
18.    data_->incoming_messages_.pop_front();
19. }
20. // 是关闭消息则关闭端口
21. if (received.IsCloseMessage()) {
22.     Close();
23.     return env()->no_message_symbol();
24. }
25.
26. // 反序列化后返回
27. return received.Deserialize(env(), context);
28. }
```

ReceiveMessage 会消息进行反序列化返回。以上就是线程间通信的整个过程。具体步骤如图 14-5 所示。

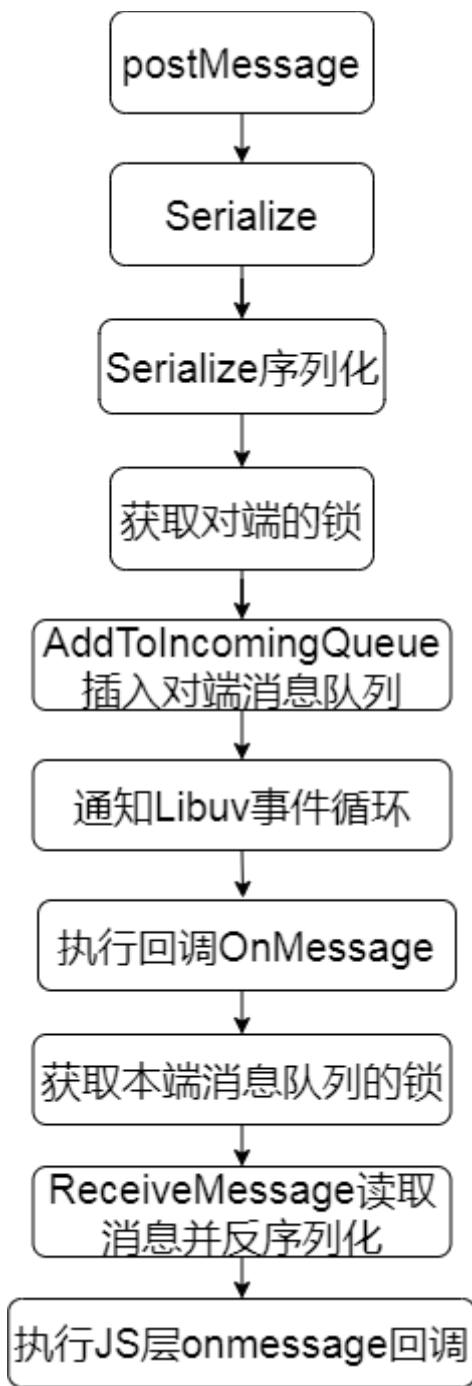


图 14-5

第十五章 cluster

Node.js 是单进程单线程的应用，这种架构带来的缺点是不能很好地利用多核的能力，因为一个线程同时只能在一个核上执行。`child_process` 模块一定程度地解决了这个问题，

child_process 模块使得 Node.js 应用可以在多个核上执行，而 cluster 模块在 child_process 模块的基础上使得多个进程可以监听的同一个端口，实现服务器的多进程架构。本章分析 cluster 模块的使用和原理。

15.1 cluster 使用例子

我们首先看一下 cluster 的一个使用例子。

```

1. const cluster = require('cluster');
2. const http = require('http');
3. const numCPUs = require('os').cpus().length;
4.
5. if (cluster.isMaster) {
6.   for (let i = 0; i < numCPUs; i++) {
7.     cluster.fork();
8.   }
9. } else {
10.   http.createServer((req, res) => {
11.     res.writeHead(200);
12.     res.end('hello world\n');
13.   }).listen(8888);
14. }
```

以上代码在第一次执行的时候，`cluster.isMaster` 为 `true`，说明是主进程，然后通过 `fork` 调用创建一个子进程，在子进程中同样执行以上代码，但是 `cluster.isMaster` 为 `false`，从而执行 `else` 的逻辑，我们看到每个子进程都会监听 8888 这个端口但是又不会引起 EADDRINUSE 错误。下面我们来分析一下具体的实现。

15.2 主进程初始化

我们先看主进程时的逻辑。我们看一下 `require('cluster')` 的时候，Node.js 是怎么处理的。

```

1. const childOrMaster = 'NODE_UNIQUE_ID' in process.env ? 'child' :
   'master';
2. module.exports = require(`internal/cluster/${childOrMaster}`)
```

我们看到 Node.js 会根据当前环境变量的值加载不同的模块，后面我们会看到 `NODE_UNIQUE_ID` 是主进程给子进程设置的，在主进程中，`NODE_UNIQUE_ID` 是不存在的，所以主进程时，会加载 `master` 模块。

```

1. cluster.isWorker = false;
2. cluster.isMaster = true;
```

```
3. // 调度策略
4. cluster.SCHED_NONE = SCHED_NONE;
5. cluster.SCHED_RR = SCHED_RR;
6. // 调度策略的选择
7. let schedulingPolicy = {
8.   'none': SCHED_NONE,
9.   'rr': SCHED_RR
10. }[process.env.NODE_CLUSTER_SCHED_POLICY];
11.
12. if (schedulingPolicy === undefined) {
13.   schedulingPolicy = (process.platform === 'win32') ?
14.     SCHED_NONE : SCHED_RR;
15. }
16.
17. cluster.schedulingPolicy = schedulingPolicy;
18. // 创建子进程
19. cluster.fork = function(env) {
20.   // 参数处理
21.   cluster.setupMaster();
22.   const id = ++ids;
23.   // 调用 child_process 模块的 fork
24.   const workerProcess = createWorkerProcess(id, env);
25.   const worker = new Worker({
26.     id: id,
27.     process: workerProcess
28.   });
29.   // ...
30.   worker.process.on('internalMessage', internal(worker, onmessage));
31.   process.nextTick(emitForkNT, worker);
32.   cluster.workers[worker.id] = worker;
33.   return worker;
34. };
35.
36.
```

`cluster.fork` 是对 `child_process` 模块 `fork` 的封装，每次 `cluster.fork` 的时候，就会新建一个子进程，所以 `cluster` 下面会有多个子进程，`Node.js` 提供的工作模式有轮询和共享两种，下面会具体介绍。`Worker` 是对子进程的封装，通过 `process` 持有子进程的实例，并通过监听 `internalMessage` 和 `message` 事件完成主进程和子进程的通信，`internalMessage` 这是 `Node.js` 定义的内部通信事件，处理函数是 `internal(worker, onmessage)`。我们先看一下 `internal`。

```
1. const callbacks = new Map();
2. let seq = 0;
```

```

3.
4. function internal(worker, cb) {
5.   return function onInternalMessage(message, handle) {
6.     if (message.cmd !== 'NODE_CLUSTER')
7.       return;
8.
9.     let fn = cb;
10.
11.    if (message.ack !== undefined) {
12.      const callback = callbacks.get(message.ack);
13.
14.      if (callback !== undefined) {
15.        fn = callback;
16.        callbacks.delete(message.ack);
17.      }
18.    }
19.
20.    fn.apply(worker, arguments);
21.  };
22. }

```

`internal` 函数对异步消息通信做了一层封装，因为进程间通信是异步的，当我们发送多个消息后，如果收到一个回复，我们无法辨别出该回复是针对哪一个请求的，`Node.js` 通过 `seq` 的方式对每一个请求和响应做了一个编号，从而区分响应对应的请求。接着我们看一下 `message` 的实现。

```

1. function onmessage(message, handle) {
2.   const worker = this;
3.
4.   if (message.act === 'online')
5.     online(worker);
6.   else if (message.act === 'queryServer')
7.     queryServer(worker, message);
8.   else if (message.act === 'listening')
9.     listening(worker, message);
10.  else if (message.act === 'exitedAfterDisconnect')
11.    exitedAfterDisconnect(worker, message);
12.  else if (message.act === 'close')
13.    close(worker, message);
14. }

```

`onmessage` 根据收到消息的不同类型进行相应的处理。后面我们再具体分析。至此，主进程的逻辑就分析完了。

15.3 子进程初始化

我们来看一下子进程的逻辑。当执行子进程时，会加载 `child` 模块。

```
1. const cluster = new EventEmitter();
2. const handles = new Map();
3. const indexes = new Map();
4. const noop = () => {};
5.
6. module.exports = cluster;
7.
8. cluster.isWorker = true;
9. cluster.isMaster = false;
10. cluster.worker = null;
11. cluster.Worker = Worker;
12.
13. cluster._setupWorker = function() {
14.   const worker = new Worker({
15.     id: +process.env.NODE_UNIQUE_ID | 0,
16.     process: process,
17.     state: 'online'
18.   });
19.
20.   cluster.worker = worker;
21.
22.   process.on('internalMessage', internal(worker, onmessage));
23.   // 通知主进程子进程启动成功
24.   send({ act: 'online' });
25.
26.   function onmessage(message, handle) {
27.     if (message.act === 'newconn')
28.       onconnection(message, handle);
29.     else if (message.act === 'disconnect')
30.       _disconnect.call(worker, true);
31.   }
32. };
```

`_setupWorker` 函数在子进程初始化时被执行，和主进程类似，子进程的逻辑也不多，监听 `internalMessage` 事件，并且通知主线程自己启动成功。

15.4 `http.createServer` 的处理

主进程和子进程执行完初始化代码后，子进程开始执行业务代码 `http.createServer`，在 HTTP 模块章节我们已经分析过 `http.createServer` 的过程，这里就不具体分析，我

我们知道 `http.createServer` 最后会调用 `net` 模块的 `listen`, 然后调用 `listenIncluster`。我们从该函数开始分析。

```

1. function listenIncluster(server, address, port, addressType,
2.                           backlog, fd, exclusive, flags) {
3.
4.   const serverQuery = {
5.     address: address,
6.     port: port,
7.     addressType: addressType,
8.     fd: fd,
9.     flags,
10.    };
11.
12.  cluster._getServer(server, serverQuery, listenOnMasterHandle);

13.  function listenOnMasterHandle(err, handle) {
14.    err = checkBindError(err, port, handle);
15.
16.    if (err) {
17.      const ex = exceptionWithHostPort(err,
18.                                       'bind',
19.                                       address,
20.                                       port);
21.      return server.emit('error', ex);
22.    }
23.
24.    server._handle = handle;
25.    server._listen2(address,
26.                    port,
27.                    addressType,
28.                    backlog,
29.                    fd,
30.                    flags);
31.  }
32. }
```

`listenIncluster` 函数会调用子进程 `cluster` 模块的 `_getServer`。

```

1. cluster._getServer = function(obj, options, cb) {
2.   let address = options.address;
3.
4.   // 忽略 index 的处理逻辑
5.
6.   const message = {
7.     act: 'queryServer',
```

```
8.     index,
9.     data: null,
10.    ...options
11.   };
12.
13.   message.address = address;
14.   // 给主进程发送消息
15.   send(message, (reply, handle) => {
16.     // 根据不同模式做处理
17.     if (handle)
18.       shared(reply, handle, indexesKey, cb);
19.     else
20.       rr(reply, indexesKey, cb);
21.   });
22. };
```

`_getServer` 会给主进程发送一个 `queryServer` 的请求。我们看一下 `send` 函数。

```
1. function send(message, cb) {
2.   return sendHelper(process, message, null, cb);
3. }
4.
5. function sendHelper(proc, message, handle, cb) {
6.   if (!proc.connected)
7.     return false;
8.
9.   message = { cmd: 'NODE_CLUSTER', ...message, seq };
10.
11. if (typeof cb === 'function')
12.   callbacks.set(seq, cb);
13.
14. seq += 1;
15. return proc.send(message, handle);
16. }
```

`send` 调用了 `sendHelper`, `sendHelper` 是对异步请求做了一个封装, 我们看一下主进程是如何处理 `queryServer` 请求的。

```
1. function queryServer(worker, message) {
2.   const key = `${message.address}:${message.port}:${message.addre
      ssType}:` + `${message.fd}:${message.index}`;
3.   let handle = handles.get(key);
4.
5.   if (handle === undefined) {
6.     let address = message.address;
7.     let constructor = RoundRobinHandle;
```

```

8.    // 根据策略选取不同的构造函数
9.    if (schedulingPolicy !== SCHED_RR || 
10.        message.addressType === 'udp4' || 
11.        message.addressType === 'udp6') {
12.        constructor = SharedHandle;
13.    }
14.
15.    handle = new constructor(key,
16.                            address,
17.                            message.port,
18.                            message.addressType,
19.                            message.fd,
20.                            message.flags);
21.    handles.set(key, handle);
22. }
23. handle.add(worker, (errno, reply, handle) => {
24.     const { data } = handles.get(key);
25.
26.     send(worker, {
27.         errno,
28.         key,
29.         ack: message.seq,
30.         data,
31.         ...reply
32.     }, handle);
33. });
34. }

```

`queryServer` 首先根据调度策略选择构造函数，然后执行对应的 `add` 方法并且传入一个回调。下面我们看看不同模式下的处理。

15.5 共享模式

下面我们首先看一下共享模式的处理，逻辑如图 19-1 所示。

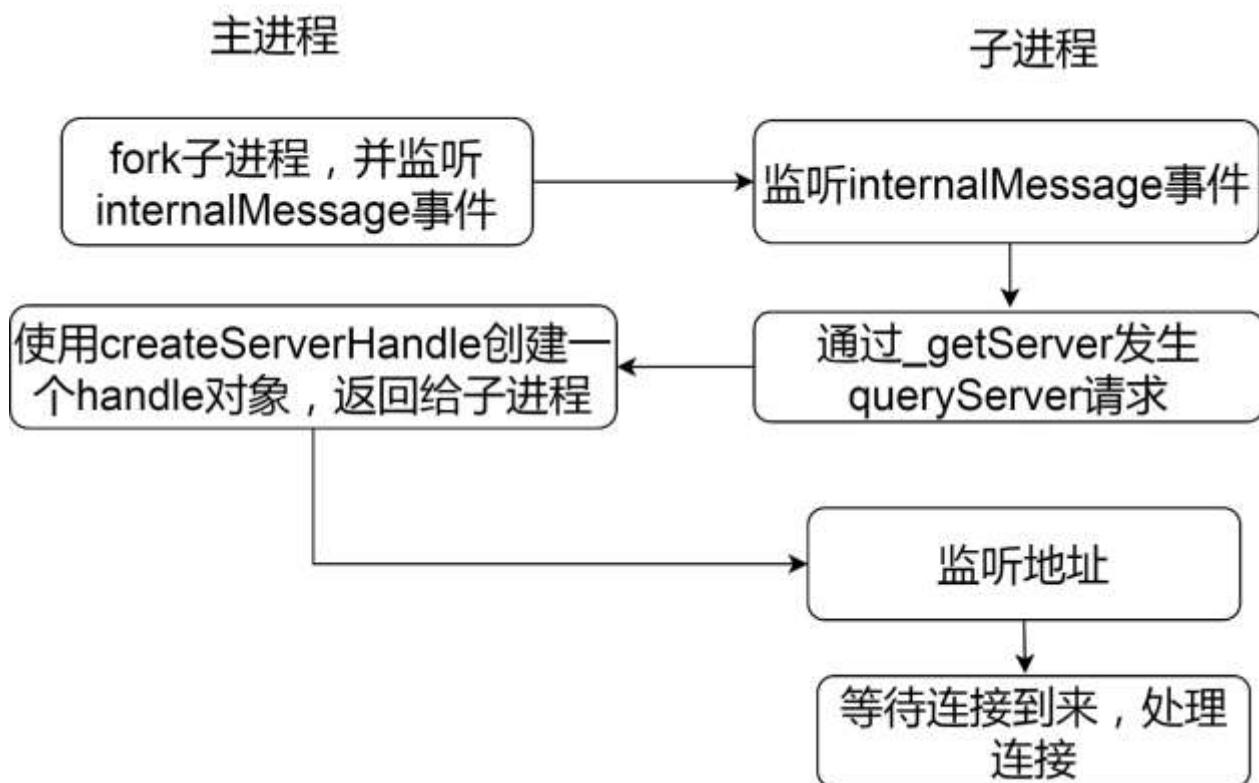


图 19-1

```
1. function SharedHandle(key, address, port, addressType, fd, flags)
  {
2.   this.key = key;
3.   this.workers = [];
4.   this.handle = null;
5.   this.errno = 0;
6.
7.   let rval;
8.   if (addressType === 'udp4' || addressType === 'udp6')
9.     rval = dgram._createSocketHandle(address,
10.                                         port,
11.                                         addressType,
12.                                         fd,
13.                                         flags);
14. else
15.   rval = net._createServerHandle(address,
16.                                         port,
17.                                         addressType,
18.                                         fd,
19.                                         flags);
20.
21. if (typeof rval === 'number')
22.   this(errno) = rval;
```

```

23. else
24.   this.handle = rval;
25. }

```

SharedHandle 是共享模式，即主进程创建好 handle，交给子进程处理。

```

1. SharedHandle.prototype.add = function(worker, send) {
2.   this.workers.push(worker);
3.   send(this.errno, null, this.handle);
4. };

```

SharedHandle 的 add 把 SharedHandle 中创建的 handle 返回给子进程，接着我们看看子进程拿到 handle 后的处理

```

1. function shared(message, handle, indexesKey, cb) {
2.   const key = message.key;
3.
4.   const close = handle.close;
5.
6.   handle.close = function() {
7.     send({ act: 'close', key });
8.     handles.delete(key);
9.     indexes.delete(indexesKey);
10.    return close.apply(handle, arguments);
11.  };
12.  handles.set(key, handle);
13.  // 执行 net 模块的回调
14.  cb(message(errno, handle));
15. }

```

Shared 函数把接收到的 handle 再回传到调用方。即 net 模块。net 模块会执行 listen 开始监听地址，但是有连接到来时，系统只会有一个进程拿到该连接。所以所有子进程存在竞争关系导致负载不均衡，这取决于操作系统的实现。

共享模式实现的核心逻辑主进程在 `_createServerHandle` 创建 handle 时执行 bind 绑定了地址（但没有 listen），然后通过文件描述符传递的方式传给子进程，子进程执行 listen 的时候就不会报端口已经被监听的错误了。因为端口被监听的错误是执行 bind 的时候返回的。

15.6 轮询模式

接着我们看一下 RoundRobinHandle 的处理，逻辑如图 19-2 所示。

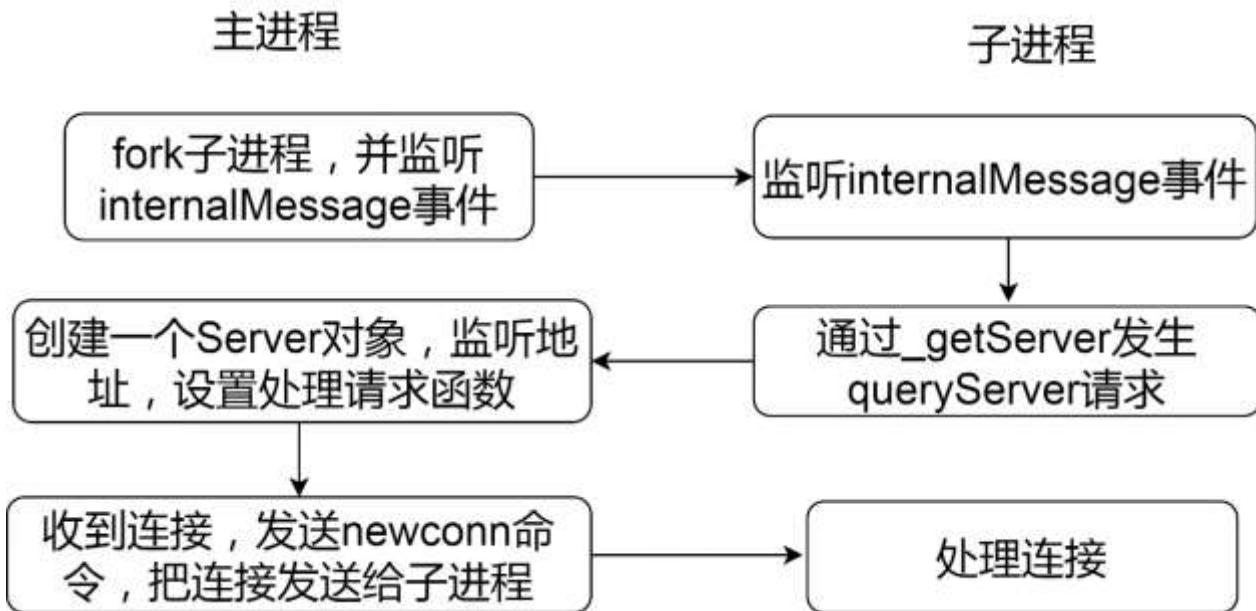


图 19-2

```
1. function RoundRobinHandle(key, address, port, addressType, fd, flags) {
2.   this.key = key;
3.   this.all = new Map();
4.   this.free = [];
5.   this.handles = [];
6.   this.handle = null;
7.   this.server = net.createServer(assert.fail);
8.
9.   if (fd >= 0)
10.    this.server.listen({ fd });
11.   else if (port >= 0) {
12.     this.server.listen({
13.       port,
14.       host: address,
15.       ipv6only: Boolean(flags & constants.UV_TCP_IPV6ONLY),
16.     });
17.   } else
18.     this.server.listen(address); // UNIX socket path.
19.   // 监听成功后，注册 onconnection 回调，有连接到来时执行
20.   this.server.once('listening', () => {
21.     this.handle = this.server._handle;
22.     this.handle.onconnection = (err, handle) => this.distribute(
23.       err, handle);
24.     this.server._handle = null;
25.     this.server = null;
26.   });
27. }
```

26. }

`RoundRobinHandle` 的工作模式是主进程负责监听，收到连接后分发给子进程。我们看一下 `RoundRobinHandle` 的 `add`

```

1. RoundRobinHandle.prototype.add = function(worker, send) {
2.   this.all.set(worker.id, worker);
3.
4.   const done = () => {
5.     if (this.handle.getsockname) {
6.       const out = {};
7.       this.handle.getsockname(out);
8.       send(null, { sockname: out }, null);
9.     } else {
10.       send(null, null, null); // UNIX socket.
11.     }
12.
13.     // In case there are connections pending.
14.     this.handoff(worker);
15.   };
16.   // 说明 listen 成功了
17.   if (this.server === null)
18.     return done();
19.   // 否则等待 listen 成功后执行回调
20.   this.server.once('listening', done);
21.   this.server.once('error', (err) => {
22.     send(err(errno, null);
23.   });
24. };

```

`RoundRobinHandle` 会在 `listen` 成功后执行回调。我们回顾一下执行 `add` 函数时的回调。

```

1. handle.add(worker, (errno, reply, handle) => {
2.   const { data } = handles.get(key);
3.
4.   send(worker, {
5.     errno,
6.     key,
7.     ack: message.seq,
8.     data,
9.     ...reply
10.   }, handle);
11. });

```

回调函数会把 `handle` 等信息返回给子进程。但是在 `RoundRobinHandle` 和 `SharedHandle` 中返回的 `handle` 是不一样的。分别是 `null` 和 `net.createServer` 实例。接着我们回到子进程的上下文。看子进程是如何处理响应的。刚才我们讲过，不同的调度策略，返回的 `handle` 是不一样的，我们看轮询模式下的处理。

```
1. function rr(message, indexesKey, cb) {
2.   let key = message.key;
3.   function listen(backlog) {
4.     return 0;
5.   }
6.
7.   function close() {
8.     // ...
9.   }
10.
11.  const handle = { close, listen, ref: noop, unref: noop };
12.
13.  if (message.sockname) {
14.    handle.getsockname = getsockname; // TCP handles only.
15.  }
16.
17.  handles.set(key, handle);
18.  // 执行 net 模块的回调
19.  cb(0, handle);
20. }
```

`round-robin` 模式下，构造一个假的 `handle` 返回给调用方，因为调用方会调用这些函数。最后回到 `net` 模块。`net` 模块首先保存 `handle`，然后调用 `listen` 函数。当有请求到来时，`round-robin` 模块会执行 `distribute` 分发请求给子进程。

```
1. RoundRobinHandle.prototype.distribute = function(err, handle) {
2.   // 首先保存 handle 到队列
3.   this.handles.push(handle);
4.   // 从空闲队列获取一个子进程
5.   const worker = this.free.shift();
6.   // 分发
7.   if (worker)
8.     this.handoff(worker);
9. };
10.
11. RoundRobinHandle.prototype.handoff = function(worker) {
12.   // 拿到一个 handle
13.   const handle = this.handles.shift();
14.   // 没有 handle，则子进程重新入队
15.   if (handle === undefined) {
```

```

16.     this.free.push(worker); // Add to ready queue again.
17.     return;
18.   }
19.   // 通知子进程有新连接
20.   const message = { act: 'newconn', key: this.key };
21.
22.   sendHelper(worker.process, message, handle, (reply) => {
23.     // 接收成功
24.     if (reply.accepted)
25.       handle.close();
26.     else
27.       // 结束失败，则重新分发
28.       this.distribute(0, handle); // Worker is shutting down. Send to another.
29.
30.     this.handoff(worker);
31.   });
32. };

```

接着我们看一下子进程是怎么处理该请求的。

```

1. function onmessage(message, handle) {
2.   if (message.act === 'newconn')
3.     onconnection(message, handle);
4. }
5.
6. function onconnection(message, handle) {
7.   const key = message.key;
8.   const server = handles.get(key);
9.   const accepted = server !== undefined;
10.  // 回复接收成功
11.  send({ ack: message.seq, accepted });
12.
13.  if (accepted)
14.    // 在 net 模块设置
15.    server.onconnection(0, handle);
16. }

```

我们看到子进程会执行 `server.onconnection`, 这个和我们分析 `net` 模块时触发 `onconnection` 事件是一样的。

15.7 实现自己的 cluster 模块

Node.js 的 cluster 在请求分发时是按照轮询的，无法根据进程当前情况做相应的处理。了解了 cluster 模块的原理后，我们自己来实现一个 cluster 模块。

15.7.1 轮询模式

整体架构如图 19-3 所示。

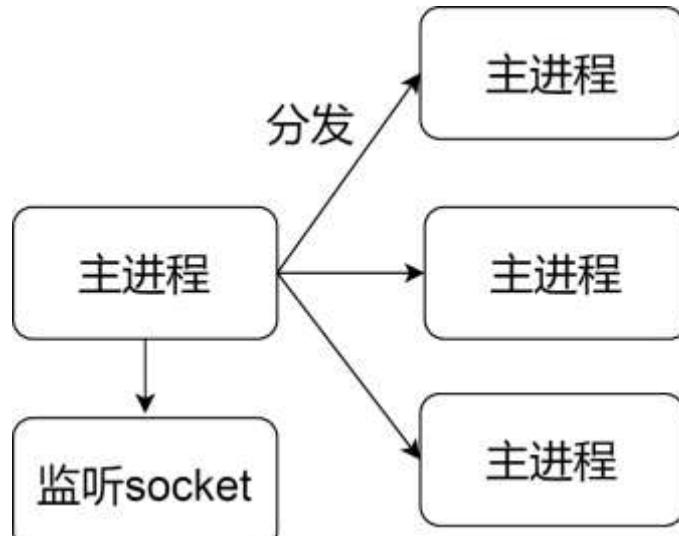


图 19-3

Parent.js

```
1. const childProcess = require('child_process');
2. const net = require('net');
3. const workers = [];
4. const workerNum = 10;
5. let index = 0;
6. for (let i = 0; i < workerNum; i++) {
7.   workers.push(childProcess.fork('child.js', {env: {index: i}}));
8. }
9.
10. const server = net.createServer((client) => {
11.   workers[index].send(null, client);
12.   console.log('dispatch to', index);
13.   index = (index + 1) % workerNum;
14. });
15. server.listen(11111);
```

child.js

```
1. process.on('message', (message, client) => {
```

```

2.     console.log('receive connection from master');
3. });

```

主进程负责监听请求，主进程收到请求后，按照一定的算法把请求通过文件描述符的方式传给 worker 进程，worker 进程就可以处理连接了。在分发算法这里，我们可以根据自己的需求进行自定义，比如根据当前进程的负载，正在处理的连接数。

15.7.2 共享模式

整体架构如图 19-4 所示。

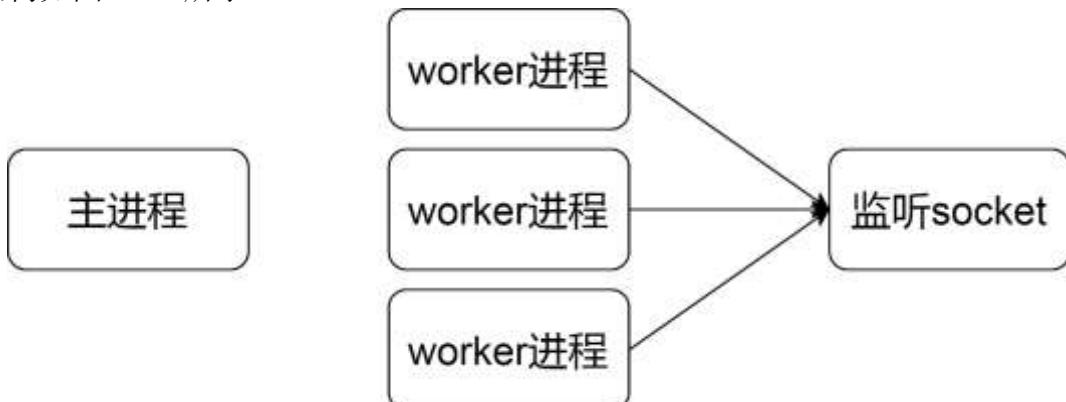


图 19-4

Parent.js

```

1. const childProcess = require('child_process');
2. const net = require('net');
3. const workers = [];
4. const workerNum = 10;
5. const handle = net._createServerHandle('127.0.0.1', 11111, 4);
6.
7. for (let i = 0; i < workerNum; i++) {
8.     const worker = childProcess.fork('child.js', {env: {index: i}})
8.     ;
9.     workers.push(worker);
10.    worker.send(null, handle);
11.    /*
12.        防止文件描述符泄漏，但是重新 fork 子进程的时候就无法
13.        再传递了文件描述符了
14.    */
15.    handle.close();
16. }

```

Child.js

```

1. const net = require('net');

```

```
2. process.on('message', (message, handle) => {
3.   net.createServer(() => {
4.     console.log(process.env.index, 'receive connection');
5.   }).listen({handle});
6.});
```

我们看到主进程负责绑定端口，然后把 `handle` 传给 worker 进程，worker 进程各自执行 `listen` 监听 socket。当有连接到来的时候，操作系统会选择某一个 worker 进程处理该连接。我们看一下共享模式下操作系统中的架构，如图 19-5 所示。

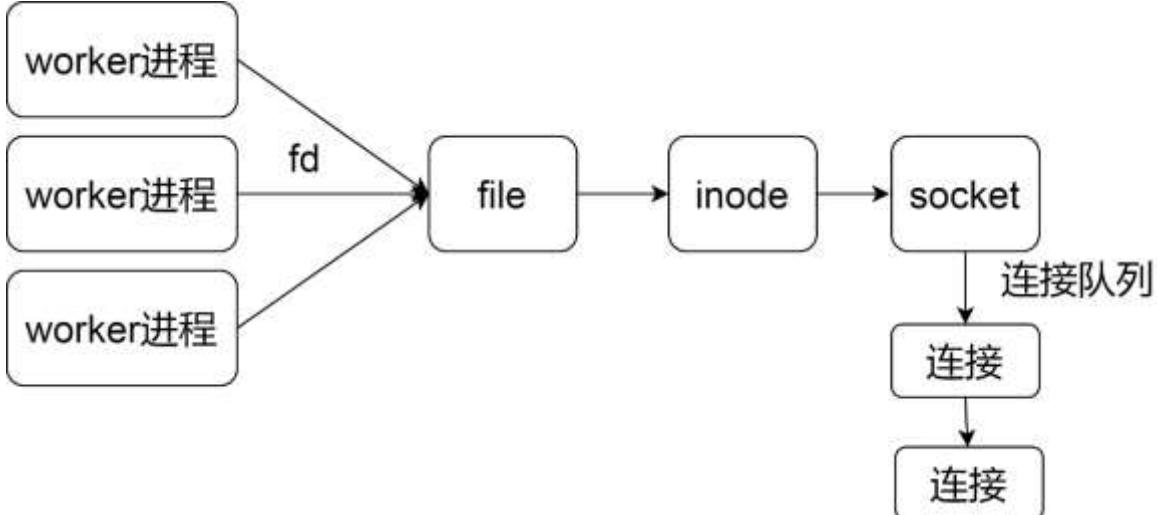


图 19-5

实现共享模式的重点在于理解 `EADDRINUSE` 错误是怎么来的。当主进程执行 `bind` 的时候，结构如图 19-6 所示。

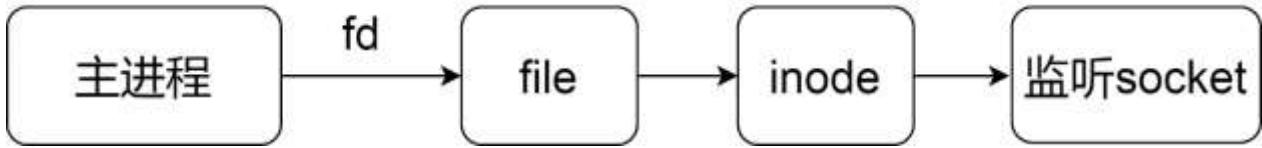


图 19-6

如果其它进程也执行 `bind` 并且端口也一样，则操作系统会告诉我们端口已经被监听了（`EADDRINUSE`）。但是如果我们在子进程中不执行 `bind` 的话，就可以绕过这个限制。那么重点在于，如何在子进程中不执行 `bind`，但是又可以绑定到同样的端口呢？有两种方式。

1 fork

我们知道 `fork` 的时候，子进程会继承主进程的文件描述符，如图 19-7 所示。

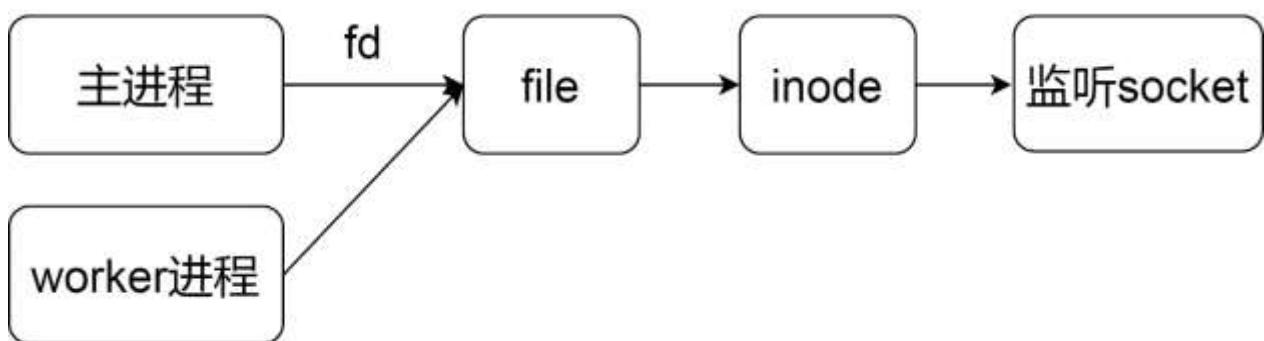


图 19-7

这时候，主进程可以执行 bind 和 listen，然后 fork 子进程，最后 close 掉自己的 fd，让所有的连接都由子进程处理就行。但是在 Node.js 中，我们无法实现，所以这种方式不能满足需求。

2 文件描述符传递

Node.js 的子进程是通过 fork+exec 模式创建的，并且 Node.js 文件描述符设置了 close_on_exec 标记，这就意味着，在 Node.js 中，创建子进程后，文件描述符的结构体如图 19-8 所示（有标准输入、标准输出、标准错误三个 fd）。

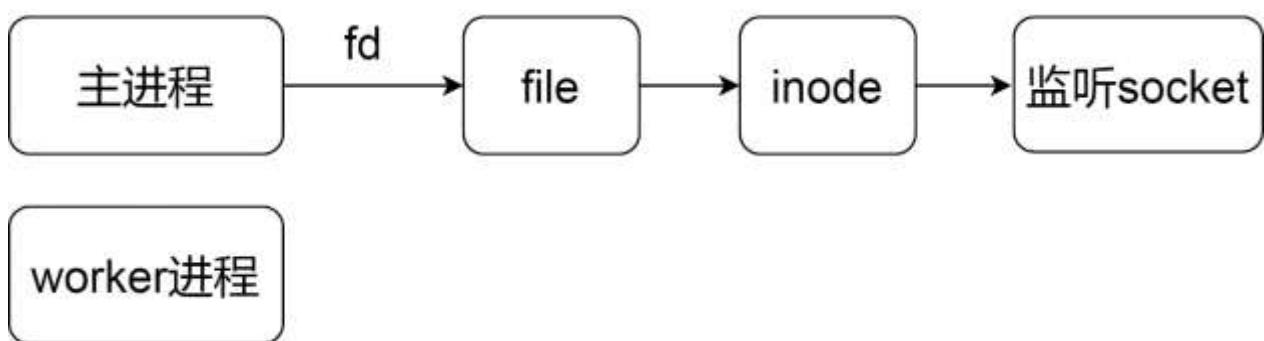


图 19-8

这时候我们可以通过文件描述符传递的方式。把方式 1 中拿不到的 fd 传给子进程。因为在 Node.js 中，虽然我们拿不到 fd，但是我们可以拿得到 fd 对应的 handle，我们通过 IPC 传输 handle 的时候，Node.js 会为我们处理 fd 的问题。最后通过操作系统对传递文件描述符的处理。结构如图 19-9 所示。

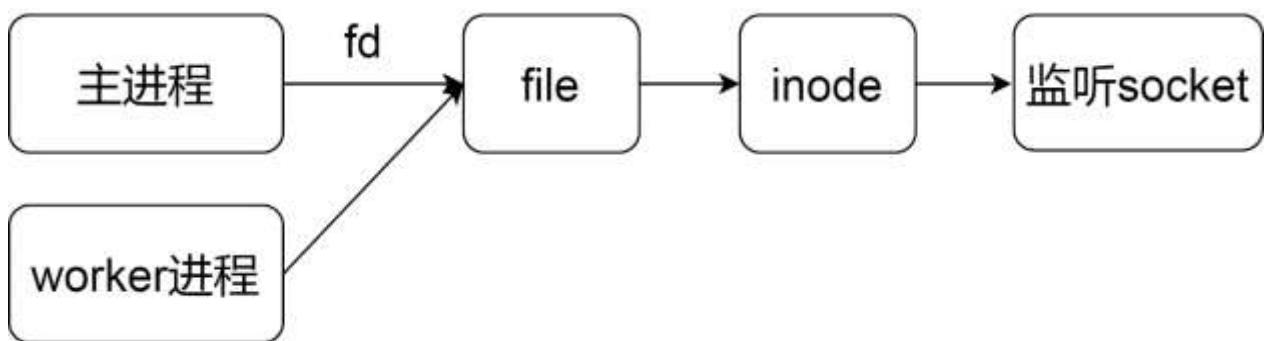


图 19-9

通过这种方式，我们就绕过了 bind 同一个端口的问题。通过以上的例子，我们知道绕过 bind 的问题重点在于让主进程和子进程共享 socket 而不是单独执行 bind。对于传递文件描述符，Node.js 中支持很多种方式。上面的方式是子进程各自执行 listen。还有另一种模式如下

parent.js

```
1. const childProcess = require('child_process');
2. const net = require('net');
3. const workers = [];
4. const workerNum = 10;
5. const server = net.createServer(() => {
6.   console.log('master receive connection');
7. })
8. server.listen(11111);
9. for (let i = 0; i < workerNum; i++) {
10.   const worker = childProcess.fork('child.js', {env: {index: i}});
11.   workers.push(worker);
12.   worker.send(null, server);
13. }
14.
```

child.js

```
1. const net = require('net');
2. process.on('message', (message, server) => {
3.   server.on('connection', () => {
4.     console.log(process.env.index, 'receive connection');
5.   })
6. });
```

上面的方式中，主进程完成了 bind 和 listen。然后把 server 实例传给子进程，子进程就可以监听连接的到来了。这时候主进程和子进程都可以处理连接。

最后写一个客户端测试。

客户端

```
1. const net = require('net');
2. for (let i = 0; i < 50; i++) {
3.   net.connect({port: 11111});
4. }
```

执行 client 我们就可以看到多进程处理连接的情况。

第十六章 UDP

本章介绍 Node.js 中的 UDP 模块，UDP 是传输层非面向连接的不可靠协议，使用 UDP 时，不需要建立连接就可以往对端直接发送数据，减少了三次握手带来的时延，但是 UDP 的不可靠可能会导致数据丢失，所以比较适合要求时延低，少量丢包不影响整体功能的场景，另外 UDP 支持多播、端口复用，可以实现一次给多个主机的多个进程发送数据。下面我们开始分析一下 UDP 的相关内容。

16.1 在 C 语言中使用 UDP

我们首先看一下在 C 语言中如何使用 UDP 功能，这是 Node.js 的底层基础。

16.1.1 服务器流程（伪代码）

```
1. // 申请一个 socket
2. int fd = socket(...);
3. // 绑定一个众所周知的地址，像 TCP 一样
4. bind(fd, ip, port);
5. // 直接阻塞等待消息的到来，UDP 不需要 listen
6. recvmsg();
```

16.1.2 客户端流程

客户端的流程有多种方式，原因在于源 IP、端口和目的 IP、端口可以有多种设置方式。不像服务器一样，服务器端口是需要对外公布的，否则客户端就无法找到目的地进行通信。这就意味着服务器的端口是需要用户显式指定的，而客户端则不然，客户端的 IP 和端口，用户可以自己指定，也可以由操作系统决定，下面我们看看各种使用方式。

16.1.2.1 显式指定源 IP 和端口

```
1. // 申请一个 socket
2. int fd = socket(...);
3. // 绑定一个客户端的地址
4. bind(fd, ip, port);
5. // 给服务器发送数据
6. sendto(fd, 服务器 ip, 服务器端口, data);
```

因为 UDP 不是面向连接的，所以使用 UDP 时，不需要调用 connect 建立连接，只要我们知道服务器的地址，直接给服务器发送数据即可。而面向连接的 TCP，首先需要通过 connect 发起三次握手建立连接，建立连接的本质是在客户端和服务器记录对端的信息，这是后面通信的通行证。

16.1.2.2 由操作系统决定源 ip 和端口

```
1. // 申请一个 socket
2. int fd = socket(...);
3. // 给服务器发送数据
4. sendto(fd, 服务器 ip, 服务器端口, data)
```

我们看到这里没有绑定客户端的源 ip 和端口，而是直接就给服务器发送数据。如果用户不指定 ip 和端口，则操作系统会提供默认的源 ip 和端口。对于 ip，如果是多宿主主机，每次调用 sendto 的时候，操作系统会动态选择源 ip。对于端口，操作系统会在第一次调用 sendto 的时候随机选择一个端口，并且不能修改。另外还有一种使用方式。

```
1. // 申请一个 socket
2. int fd = socket(...);
3. connect(fd, 服务器 ip, 服务器端口);
4. /*
5.   给服务器发送数据,或者 sendto(fd, null,null, data),
6.   调用 sendto 则不需要再指定服务器 ip 和端口
7. */
8. write(fd, data);
```

我们可以先调用 connect 绑定服务器 ip 和端口到 fd，然后直接调用 write 发送数据。虽然使用方式很多，但是归根到底还是对四元组设置的管理。bind 是绑定源 ip 端口到 fd，connect 是绑定服务器 ip 端口到 fd。对于源 ip 和端口，我们可以主动设置，也可以让操作系统随机选择。对于目的 ip 和端口，我们可以在发送数据前设置，也可以在发送数据时设置。这就形成了多种使用方式。

16.1.3 发送数据

我们刚才看到使用 UDP 之前都需要调用 socket 函数申请一个 socket，虽然调用 socket 函数返回的是一个 fd，但是在操作系统中，的确是新建了一个 socket 结构体，fd 只是一个索引，操作这个 fd 的时候，操作系统会根据这个 fd 找到对应的 socket。socket 是一个非常复杂的结构体，我们可以理解为一个对象。这个对象中有两个属性，一个是读缓冲区大小，一个是写缓冲区大小。当我们发送数据的时候，虽然理论上可以发送任意大小的数据，但是因为受限于发送缓冲区的大小，如果需要发送的数据比当前缓冲区大小大则会导致一些问题，我们分情况分析一下。

1 发送的数据大小比当前缓冲区大，如果设置了非阻塞模式，则返回 EAGAIN，如果是阻塞模式，则会引起进程的阻塞。

2 如果发送的数据大小比缓冲区的最大值还大，则会导致报错 EMSGSIZE，这时候我们需要分包发送。我们可能会想到修改缓冲区最大值的大小，但是这个大小也是有限制的。

讲完一些边界情况，我们再来看看正常的流程，我们看看发送一个数据包的流程

1 首先在 socket 的写缓冲区申请一块内存用于数据发送。

2 调用 IP 层发送接口，如果数据包大小超过了 IP 层的限制，则需要分包。

3 继续调用底层的接口把数据发到网络上。

因为 UDP 不是可靠的，所以不需要缓存这个数据包（TCP 协议则需要缓存这个数据包，用于超时重传）。这就是 UDP 发送数据的流程。

16.1.4 接收数据

当收到一个 UDP 数据包的时候，操作系统首先会把这个数据包缓存到 socket 的缓冲区，如果收到的数据包比当前缓冲区大小大，则丢弃数据包，否则把数据包挂载到接收队列，等用户来读取的时候，就逐个摘下接收队列的节点。UDP 和 TCP 不一样，虽然它们都有一个缓存了消息的队列，但是当用户读取数据时，UDP 每次只会返回一个 UDP 数据包，而 TCP 是会根据用户设置的大小返回一个或多个包里的数据。因为 TCP 是面向字节流的，而 UDP 是面向数据包的。

16.2 UDP 模块在 Node.js 中的实现

了解了 UDP 的一些基础和使用后，我们开始分析在 Node.js 中是如何使用 UDP 的，Node.js 又是如何实现 UDP 模块的。

16.2.1 服务器

我们从一个使用例子开始看看 UDP 模块的使用。

```
1. const dgram = require('dgram');
2. // 创建一个 UDP 服务器
3. const server = dgram.createSocket('udp4');
4. // 监听 UDP 数据的到来
5. server.on('message', (msg, rinfo) => {
6.   // 处理数据
7. });
8. // 绑定端口
9. server.bind(41234);
```

我们看到创建一个 UDP 服务器很简单，首先申请一个 socket 对象，在 Node.js 中和操作系统中一样，socket 是对网络通信的一个抽象，我们可以把它理解成对传输层的抽象，它可以代表 TCP 也可以代表 UDP。我们看一下 createSocket 做了什么。

```
1. function createSocket(type, listener) {
2.   return new Socket(type, listener);
3. }
4. function Socket(type, listener) {
5.   EventEmitter.call(this);
```

```
6. let lookup;
7. let recvBufferSize;
8. let sendBufferSize;
9.
10. let options;
11. if (type !== null && typeof type === 'object') {
12.     options = type;
13.     type = options.type;
14.     lookup = options.lookup;
15.     recvBufferSize = options.recvBufferSize;
16.     sendBufferSize = options.sendBufferSize;
17. }
18. const handle = newHandle(type, lookup);
19. this.type = type;
20. if (typeof listener === 'function')
21.     this.on('message', listener);
22. // 保存上下文
23. this[kStateSymbol] = {
24.     handle,
25.     receiving: false,
26.     // 还没有执行 bind
27.     bindState: BIND_STATE_UNBOUND,
28.     connectState: CONNECT_STATE_DISCONNECTED,
29.     queue: undefined,
30.     // 端口复用, 只适用于多播
31.     reuseAddr: options && options.reuseAddr,
32.     ipv6Only: options && options.ipv6Only,
33.     // 发送缓冲区和接收缓冲区大小
34.     recvBufferSize,
35.     sendBufferSize
36. };
37. }
```

我们看到一个 socket 对象是对 handle 的一个封装。我们看看 handle 是什么。

```
1. function newHandle(type, lookup) {
2.     // 用于 dns 解析的函数, 比如我们调 send 的时候, 传的是一个域名
3.     if (lookup === undefined) {
4.         if (dns === undefined) {
5.             dns = require('dns');
6.         }
7.         lookup = dns.lookup;
8.     }
9.
10. if (type === 'udp4') {
11.     const handle = new UDP();
```

```

12.     handle.lookup = lookup4.bind(handle, lookup);
13.     return handle;
14. }
15. // 忽略 ipv6 的处理
16. }
```

handle 又是对 UDP 模块的封装，UDP 是 C++ 模块，在之前章节中我们讲过相关的知识，这里就不详细讲述了，当我们在 JS 层 new UDP 的时候，会新建一个 C++ 对象。

```

1. UDPWrap::UDPWrap(Environment* env, Local<Object> object)
2.   : HandleWrap(env,
3.                 object,
4.                 reinterpret_cast<uv_handle_t*>(&handle_),
5.                 AsyncWrap::PROVIDER_UDPWRAP) {
6.   int r = uv_udp_init(env->event_loop(), &handle_);
7. }
```

执行了 uv_udp_init 初始化 udp 对应的 handle (uv_udp_t)。我们看一下 Libuv 的定义。

```

1. int uv_udp_init_ex(uv_loop_t* loop, uv_udp_t* handle, unsigned int
2.   flags) {
3.   int domain;
4.   int err;
5.   int fd;
6. 
7.   /* Use the lower 8 bits for the domain */
8.   domain = flags & 0xFF;
9.   // 申请一个 socket, 返回一个 fd
10.  fd = uv_socket(domain, SOCK_DGRAM, 0);
11.  uv_handle_init(loop, (uv_handle_t*)handle, UV_UDP);
12.  handle->alloc_cb = NULL;
13.  handle->recv_cb = NULL;
14.  handle->send_queue_size = 0;
15.  handle->send_queue_count = 0;
16.  /*
17.   * 初始化 IO 观察者（还没有注册到事件循环的 Poll IO 阶段）,
18.   * 监听的文件描述符是 fd, 回调是 uv_udp_io
19.   */
20.  uv_io_init(&handle->io_watcher, uv_udp_io, fd);
21.  // 初始化写队列
22.  QUEUE_INIT(&handle->write_queue);
23.  QUEUE_INIT(&handle->write_completed_queue);
24.  return 0;
25. }
```

就是我们在 JS 层执行 dgram.createSocket('udp4') 的时候，在 Node.js 中主要的执行过程。回到最开始的例子，我们看一下执行 bind 的时候的逻辑。

```
1. Socket.prototype.bind = function(port_, address_ /* , callback */)
2. {
3.     let port = port_;
4.     // socket 的上下文
5.     const state = this[kStateSymbol];
6.     // 已经绑定过了则报错
7.     if (state.bindState !== BIND_STATE_UNBOUND)
8.         throw new ERR_SOCKET_ALREADY_BOUND();
9.     // 否则标记已经绑定了
10.    state.bindState = BIND_STATE_BINDING;
11.    // 没传地址则默认绑定所有地址
12.    if (!address) {
13.        if (this.type === 'udp4')
14.            address = '0.0.0.0';
15.        else
16.            address = '::';
17.    }
18.    // dns 解析后在绑定，如果需要的话
19.    state.handle.lookup(address, (err, ip) => {
20.        if (err) {
21.            state.bindState = BIND_STATE_UNBOUND;
22.            this.emit('error', err);
23.            return;
24.        }
25.        const err = state.handle.bind(ip, port || 0, flags);
26.        if (err) {
27.            const ex = exceptionWithHostPort(err, 'bind', ip, port);
28.            state.bindState = BIND_STATE_UNBOUND;
29.            this.emit('error', ex);
30.            // Todo: close?
31.            return;
32.        }
33.        startListening(this);
34.    return this;
35. }
```

bind 函数主要的逻辑是 handle.bind 和 startListening。我们一个个看。我们看一下 C++ 层的 bind。

```
1. void UDPWrap::DoBind(const FunctionCallbackInfo<Value>& args, int
family) {
```

```

2.     UDPWrap* wrap;
3.     ASSIGN_OR_RETURN_UNWRAP(&wrap,
4.                             args.Holder(),
5.                             args.GetReturnValue().Set(UV_EBADF));
6.
7.     // bind(ip, port, flags)
8.     CHECK_EQ(args.Length(), 3);
9.     node::Utf8Value address(args.GetIsolate(), args[0]);
10.    Local<Context> ctx = args.GetIsolate()->GetCurrentContext();
11.    uint32_t port, flags;
12.    struct sockaddr_storage addr_storage;
13.    int err = sockaddr_for_family(family,
14.                                   address.out(),
15.                                   port,
16.                                   &addr_storage);
17.    if (err == 0) {
18.        err = uv_udp_bind(&wrap->handle_,
19.                          reinterpret_cast<const sockaddr*>(&addr_st
20.                           orage),
21.                          flags);
22.
23.        args.GetReturnValue().Set(err);
24.    }

```

也没有太多逻辑，处理参数然后执行 `uv_udp_bind` 设置一些标记、属性和端口复用（端口复用后续会单独分析），然后执行操作系统 `bind` 的函数把本端的 ip 和端口保存到 `socket` 中。我们继续看 `startListening`。

```

1. function startListening(socket) {
2.     const state = socket[kStateSymbol];
3.     // 有数据时的回调，触发 message 事件
4.     state.handle.onmessage = onMessage;
5.     // 重点，开始监听数据
6.     state.handle.recvStart();
7.     state.receiving = true;
8.     state.bindState = BIND_STATE_BOUND;
9.     // 设置操作系统的接收和发送缓冲区大小
10.    if (state.recvBufferSize)
11.        bufferSize(socket, state.recvBufferSize, RECV_BUFFER);
12.
13.    if (state.sendBufferSize)
14.        bufferSize(socket, state.sendBufferSize, SEND_BUFFER);
15.
16.    socket.emit('listening');
17. }

```

重点是 recvStart 函数，我们看 C++ 的实现。

```
1. void UDPWrap::RecvStart(const FunctionCallbackInfo<Value>& args)
{
2.     UDPWrap* wrap;
3.     ASSIGN_OR_RETURN_UNWRAP(&wrap,
4.                             args.Holder(),
5.                             args.GetReturnValue().Set(UV_EBADF));
6.     int err = uv_udp_recv_start(&wrap->handle_, OnAlloc, OnRecv);
7.     // UV_EALREADY means that the socket is already bound but that's okay
8.     if (err == UV_EALREADY)
9.         err = 0;
10.    args.GetReturnValue().Set(err);
11. }
```

OnAlloc, OnRecv 分别是分配内存接收数据的函数和数据到来时执行的回调。继续看 Libuv

```
1. int uv_udp_recv_start(uv_udp_t* handle,
2.                         uv_alloc_cb alloc_cb,
3.                         uv_udp_recv_cb recv_cb) {
4.     int err;
5.
6.
7.     err = uv__udp_maybe_deferred_bind(handle, AF_INET, 0);
8.     if (err)
9.         return err;
10.    // 保存一些上下文
11.    handle->alloc_cb = alloc_cb;
12.    handle->recv_cb = recv_cb;
13.    // 注册 IO 观察者到 loop，如果事件到来，等到 Poll IO 阶段处理
14.    uv__io_start(handle->loop, &handle->io_watcher, POLLIN);
15.    uv__handle_start(handle);
16.
17.    return 0;
18. }
```

uv_udp_recv_start 主要是注册 IO 观察者到 loop，等待事件到来的时候，到这，服务器就启动了。

16.2.2 客户端

接着我们看一下客户端的使用方式和流程

```

1. const dgram = require('dgram');
2. const message = Buffer.from('Some bytes');
3. const client = dgram.createSocket('udp4');
4. client.connect(41234, 'localhost', (err) => {
5.   client.send(message, (err) => {
6.     client.close();
7.   });
8. });

```

我们看到 Node.js 首先调用 connect 绑定服务器的地址，然后调用 send 发送信息，最后调用 close。我们一个个分析。首先看 connect。

```

1. Socket.prototype.connect = function(port, address, callback) {
2.   port = validatePort(port);
3.   // 参数处理
4.   if (typeof address === 'function') {
5.     callback = address;
6.     address = '';
7.   } else if (address === undefined) {
8.     address = '';
9.   }
10.
11. const state = this[kStateSymbol];
12. // 不是初始化状态
13. if (state.connectState !== CONNECT_STATE_DISCONNECTED)
14.   throw new ERR_SOCKET_DGRAM_IS_CONNECTED();
15. // 设置 socket 状态
16. state.connectState = CONNECT_STATE_CONNECTING;
17. // 还没有绑定客户端地址信息，则先绑定随机地址（操作系统决定）
18. if (state.bindState === BIND_STATE_UNBOUND)
19.   this.bind({ port: 0, exclusive: true }, null);
20. // 执行 bind 的时候，state.bindState 不是同步设置的
21. if (state.bindState !== BIND_STATE_BOUND) {
22.   enqueue(this, _connect.bind(this, port, address, callback));
23.   return;
24. }
25.
26. _connect.call(this, port, address, callback);
27. };

```

这里分为两种情况，一种是在 connect 之前已经调用了 bind，第二种是没有调用 bind，如果没有调用 bind，则在 connect 之前先要调用 bind（因为 bind 中不仅仅绑定了 ip 端口，还有端口复用的处理）。这里只分析没有调用 bind 的情况，因为这是最长的路径。bind 刚才我们分析过了，我们从以下代码继续分析

```
1. if (state.bindState !== BIND_STATE_BOUND) {  
2.   enqueue(this, _connect.bind(this, port, address, callback));  
3.   return;  
4. }
```

enqueue 把任务加入任务队列，并且监听了 listening 事件（该事件在 bind 成功后触发）。

```
1. function enqueue(self, toEnqueue) {  
2.   const state = self[kStateSymbol];  
3.   if (state.queue === undefined) {  
4.     state.queue = [];  
5.     self.once('error', onListenError);  
6.     self.once('listening', onListenSuccess);  
7.   }  
8.   state.queue.push(toEnqueue);  
9. }
```

这时候 connect 函数就执行完了，等待 bind 成功后 (nextTick) 会执行 startListening 函数。

```
1. function startListening(socket) {  
2.   const state = socket[kStateSymbol];  
3.   state.handle.onmessage = onMessage;  
4.   // 注册等待可读事件  
5.   state.handle.recvStart();  
6.   state.receiving = true;  
7.   // 标记已 bind 成功  
8.   state.bindState = BIND_STATE_BOUND;  
9.   // 设置读写缓冲区大小  
10.  if (state.recvBufferSize)  
11.    bufferSize(socket, state.recvBufferSize, RECV_BUFFER);  
12.  if (state.sendBufferSize)  
13.    bufferSize(socket, state.sendBufferSize, SEND_BUFFER);  
14.  // 触发 listening 事件  
15.  socket.emit('listening');  
16. }  
17. }
```

我们看到 startListening 触发了 listening 事件，从而执行我们刚才入队的回调 onListenSuccess。

```
1. function onListenSuccess() {  
2.   this.removeListener('error', onListenError);  
3.   clearQueue.call(this);
```

```

4. }
5.
6. function clearQueue() {
7.   const state = this[kStateSymbol];
8.   const queue = state.queue;
9.   state.queue = undefined;
10.
11.   for (const queueEntry of queue)
12.     queueEntry();
13. }
```

回调就是把队列中的回调执行一遍，connect 函数设置的回调是_connect。

```

1. function _connect(port, address, callback) {
2.   const state = this[kStateSymbol];
3.   if (callback)
4.     this.once('connect', callback);
5.
6.   const afterDns = (ex, ip) => {
7.     defaultTriggerAsyncIdScope(
8.       this[async_id_symbol],
9.       doConnect,
10.      ex, this, ip, address, port, callback
11.    );
12.  };
13.
14. state.handle.lookup(address, afterDns);
15. }
```

这里的 address 是服务器地址，_connect 函数主要逻辑是

1 监听 connect 事件

2 对服务器地址进行 dns 解析（只能是本地的配的域名）。解析成功后执行 afterDns，最后执行 doConnect，并传入解析出来的 ip。我们看看 doConnect

```

1. function doConnect(ex, self, ip, address, port, callback) {
2.   const state = self[kStateSymbol];
3.   // dns 解析成功，执行底层的 connect
4.   if (!ex) {
5.     const err = state.handle.connect(ip, port);
6.     if (err) {
7.       ex = exceptionWithHostPort(err, 'connect', address, port);
8.     }
9.   }
10.
11. // connect 成功，触发 connect 事件
```

```
12. state.connectState = CONNECT_STATE_CONNECTED;
13. process.nextTick(() => self.emit('connect'));
14. }
```

connect 函数通过 C++ 层，然后调用 Libuv，到操作系统的 connect。作用是把服务器地址保存到 socket 中。connect 的流程就走完了。接下来我们就可以调用 send 和 recv 发送和接收数据。

16.2.3 发送数据

发送数据接口是 sendto，它是对 send 的封装。

```
1. Socket.prototype.send = function(buffer,
2.                                     offset,
3.                                     length,
4.                                     port,
5.                                     address,
6.                                     callback) {
7.
8.   let list;
9.   const state = this[kStateSymbol];
10.  const connected = state.connectState === CONNECT_STATE_CONNECT
    ED;
11. // 没有调用 connect 绑定过服务端地址，则需要传服务端地址信息
12.  if (!connected) {
13.    if (address || (port && typeof port !== 'function')) {
14.      buffer = sliceBuffer(buffer, offset, length);
15.    } else {
16.      callback = port;
17.      port = offset;
18.      address = length;
19.    }
20.  } else {
21.    if (typeof length === 'number') {
22.      buffer = sliceBuffer(buffer, offset, length);
23.      if (typeof port === 'function') {
24.        callback = port;
25.        port = null;
26.      }
27.    } else {
28.      callback = offset;
29.    }
30.  // 已经绑定了服务端地址，则不能再传了
31.  if (port || address)
32.    throw new ERR_SOCKET_DGRAM_IS_CONNECTED();
```

```

33. }
34. // 如果没有绑定服务器端口，则这里需要传，并且校验
35. if (!connected)
36.     port = validatePort(port);
37. // 忽略一些参数处理逻辑
38. // 没有绑定客户端地址信息，则需要先绑定，值由操作系统决定
39. if (state.bindState === BIND_STATE_UNBOUND)
40.     this.bind({ port: 0, exclusive: true }, null);
41. // bind 还没有完成，则先入队，等待 bind 完成再执行
42. if (state.bindState !== BIND_STATE_BOUND) {
43.     enqueue(this, this.send.bind(this,
44.                                 list,
45.                                 port,
46.                                 address,
47.                                 callback));
48.     return;
49. }
50. // 已经绑定了，设置服务端地址后发送数据
51. const afterDns = (ex, ip) => {
52.     defaultTriggerAsyncIdScope(
53.         this[async_id_symbol],
54.         doSend,
55.         ex, this, ip, list, address, port, callback
56.     );
57. };
58. // 传了地址则可能需要 dns 解析
59. if (!connected) {
60.     state.handle.lookup(address, afterDns);
61. } else {
62.     afterDns(null, null);
63. }
64. }

```

我们继续看 doSend 函数。

```

1. function doSend(ex, self, ip, list, address, port, callback) {
2.     const state = self[kStateSymbol];
3.     // dns 解析出错
4.     if (ex) {
5.         if (typeof callback === 'function') {
6.             process.nextTick(callback, ex);
7.             return;
8.         }
9.         process.nextTick(() => self.emit('error', ex));
10.        return;

```

```
11. }
12. // 定义一个请求对象
13. const req = new SendWrap();
14. req.list = list; // Keep reference alive.
15. req.address = address;
16. req.port = port;
17. /*
18.   设置 Node.js 和用户的回调, oncomplete 由 C++ 层调用,
19.   callback 由 oncomplete 调用
20. */
21. if (callback) {
22.   req.callback = callback;
23.   req.oncomplete = afterSend;
24. }
25.
26. let err;
27. // 根据是否需要设置服务端地址, 调 C++ 层函数
28. if (port)
29.   err = state.handle.send(req, list, list.length, port, ip, !!callback);
30. else
31.   err = state.handle.send(req, list, list.length, !!callback);

32. /*
33.   err 大于等于 1 说明同步发送成功了, 直接执行回调,
34.   否则等待异步回调
35. */
36. if (err >= 1) {
37.   if (callback)
38.     process.nextTick(callback, null, err - 1);
39.   return;
40. }
41. // 发送失败
42. if (err && callback) {
43.   const ex=exceptionWithHostPort(err, 'send', address, port);
44.   process.nextTick(callback, ex);
45. }
46. }
```

我们穿过 C++ 层, 直接看 Libuv 的代码。

```
1. int uv__udp_send(uv_udp_send_t* req,
2.                   uv_udp_t* handle,
3.                   const uv_buf_t bufs[],
4.                   unsigned int nbufs,
```

```

5.          const struct sockaddr* addr,
6.          unsigned int addrlen,
7.          uv_udp_send_cb send_cb) {
8.      int err;
9.      int empty_queue;
10.     assert(nbufs > 0);
11.     // 还没有绑定服务端地址，则绑定
12.     if (addr) {
13.         err = uv_udp_maybe_deferred_bind(handle,
14.                                         addr->sa_family,
15.                                         0);
16.         if (err)
17.             return err;
18.     }
19.     // 当前写队列是否为空
20.     empty_queue = (handle->send_queue_count == 0);
21.     // 初始化一个写请求
22.     uv_req_init(handle->loop, req, UV_UDP_SEND);
23.     if (addr == NULL)
24.         req->addr.ss_family = AF_UNSPEC;
25.     else
26.         memcpy(&req->addr, addr, addrlen);
27.     // 保存上下文
28.     req->send_cb = send_cb;
29.     req->handle = handle;
30.     req->nbufs = nbufs;
31.     // 初始化数据，预分配的内存不够，则分配新的堆内存
32.     req->bufs = req->bufsml;
33.     if (nbufs > ARRAY_SIZE(req->bufsml))
34.         req->bufs = uv_malloc(nbufs * sizeof(bufs[0]));
35.     // 复制过去堆中
36.     memcpy(req->bufs, bufs, nbufs * sizeof(bufs[0]));
37.     // 更新写队列数据
38.     handle->send_queue_size += uv_count_bufs(req->bufs,
39.                                              req->nbufs);
40.     handle->send_queue_count++;
41.     // 插入写队列，等待可写事件的发生
42.     QUEUE_INSERT_TAIL(&handle->write_queue, &req->queue);
43.     uv_handle_start(handle);
44.     // 当前写队列为空，则直接开始写，否则设置等待可写队列
45.     if (empty_queue &&
46.         !(handle->flags & UV_HANDLE_UDP_PROCESSING)) {
47.         // 发送数据
48.         uv_udp_sendmsg(handle);

```

```
50.    // 写队列是否非空，则设置等待可写事件，可写的时候接着写
51.    if (!QUEUE_EMPTY(&handle->write_queue))
52.        uv__io_start(handle->loop, &handle->io_watcher, POLLOUT);
53.    } else {
54.        uv__io_start(handle->loop, &handle->io_watcher, POLLOUT);
55.    }
56.    return 0;
57. }
```

该函数首先记录写请求的上下文，然后把写请求插入写队列中，当待写队列为空，则直接执行 `uv__udp_sendmsg` 进行写操作，否则等待可写事件的到来，当可写事件触发的时候，执行的函数是 `uv__udp_io`。

```
1. static void uv__udp_io(uv_loop_t* loop, uv__io_t* w, unsigned int
   revents) {
2.     uv_udp_t* handle;
3.     if (revents & POLLOUT) {
4.         uv__udp_sendmsg(handle);
5.         uv__udp_run_completed(handle);
6.     }
7. }
```

我们先看 `uv__udp_sendmsg`

```
1. static void uv__udp_sendmsg(uv_udp_t* handle) {
2.     uv_udp_send_t* req;
3.     QUEUE* q;
4.     struct msghdr h;
5.     ssize_t size;
6.     // 逐个节点发送
7.     while (!QUEUE_EMPTY(&handle->write_queue)) {
8.         q = QUEUE_HEAD(&handle->write_queue);
9.         req = QUEUE_DATA(q, uv_udp_send_t, queue);
10.        memset(&h, 0, sizeof h);
11.        // 忽略参数处理
12.        h.msg iov = (struct iovec*) req->bufs;
13.        h.msg iovlen = req->nbufs;
14.
15.        do {
16.            size = sendmsg(handle->io_watcher.fd, &h, 0);
17.        } while (size == -1 && errno == EINTR);
18.
19.        if (size == -1) {
20.            // 繁忙则先不发了，等到可写事件
```

```

21.     if (errno == EAGAIN || errno == EWOULDBLOCK || errno == EN
    OBUFS)
22.     break;
23. }
24. // 记录发送结果
25. req->status = (size == -1 ? UV__ERR(errno) : size);
26. // 发送“完”移出写队列
27. QUEUE_REMOVE(&req->queue);
28. // 加入写完成队列
29. QUEUE_INSERT_TAIL(&handle->write_completed_queue, &req->queu
e);
30. /*
31.     有节点数据写完了，把 IO 观察者插入 pending 队列，
32.     pending 阶段执行回调 uv_udp_io
33. */
34. uv_io_feed(handle->loop, &handle->io_watcher);
35. }
36. }
```

该函数遍历写队列，然后逐个发送节点中的数据，并记录发送结果。

1 如果写繁忙则结束写逻辑，等待下一次写事件触发。

2 如果写成功则把节点插入写完成队列中，并且把 IO 观察者插入 pending 队列。

等待 pending 阶段执行回调时，执行的函数是 uv_udp_io。我们再次回到 uv_udp_io 中

```

1. if (revents & POLLOUT) {
2.     uv_udp_sendmsg(handle);
3.     uv_udp_run_completed(handle);
4. }
```

我们看到这时候会继续执行数据发送的逻辑，然后处理写完成队列。我们看 uv_udp_run_completed。

```

1. static void uv_udp_run_completed(uv_udp_t* handle) {
2.     uv_udp_send_t* req;
3.     QUEUE* q;
4.     handle->flags |= UV_HANDLE_UDP_PROCESSING;
5.     // 逐个节点处理
6.     while (!QUEUE_EMPTY(&handle->write_completed_queue)) {
7.         q = QUEUE_HEAD(&handle->write_completed_queue);
8.         QUEUE_REMOVE(q);
9.         req = QUEUE_DATA(q, uv_udp_send_t, queue);
10.        uv_req_unregister(handle->loop, req);
11.        // 更新待写数据大小
```

```
12.     handle->send_queue_size -
13.         = uv_count_bufs(req->bufs, req->nbufs);
14.         handle->send_queue_count--;
15.         // 如果重新申请了堆内存，则需要释放
16.         if (req->bufs != req->bufsml)
17.             uv_free(req->bufs);
18.         req->bufs = NULL;
19.         if (req->send_cb == NULL)
20.             continue;
21.         // 执行回调
22.         if (req->status >= 0)
23.             req->send_cb(req, 0);
24.         else
25.             req->send_cb(req, req->status);
26.     }
27.     // 写队列为空，则注销等待可写事件
28.     if (QUEUE_EMPTY(&handle->write_queue)) {
29.         uv_io_stop(handle->loop, &handle->io_watcher, POLLOUT);
30.         if (!uv_io_active(&handle->io_watcher, POLLIN))
31.             uv_handle_stop(handle);
32.     }
33. }
```

这就是发送的逻辑，发送完后 Libuv 会调用 C++ 回调，最后回调 JS 层回调。具体到操作系统也是类似的实现，操作系统首先判断数据的大小是否小于写缓冲区，是的话申请一块内存，然后构造 UDP 协议数据包，再逐层往下调，最后发送出来，但是如果数据超过了底层的报文大小限制，则会被分片。

16.2.4 接收数据

UDP 服务器启动的时候，就注册了等待可读事件的发送，如果收到了数据，则在 Poll IO 阶段就会被处理。前面我们讲过，回调函数是 `uv_udp_io`。我们看一下事件触发的时候，该函数怎么处理的。

```
1. static void uv_udp_io(uv_loop_t* loop, uv_io_t* w, unsigned int
   revents) {
2.     uv_udp_t* handle;
3.
4.     handle = container_of(w, uv_udp_t, io_watcher);
5.     // 可读事件触发
6.     if (revents & POLLIN)
7.         uv_udp_recvmsg(handle);
8. }
```

我们看 uv_udp_recvmsg 的逻辑。

```

1. static void uv_udp_recvmsg(uv_udp_t* handle) {
2.     struct sockaddr_storage peer;
3.     struct msghdr h;
4.     ssize_t nread;
5.     uv_buf_t buf;
6.     int flags;
7.     int count;
8.
9.     count = 32;
10.
11.    do {
12.        // 分配内存接收数据, C++层设置的
13.        buf = uv_buf_init(NULL, 0);
14.        handle->alloc_cb((uv_handle_t*) handle, 64 * 1024, &buf);
15.        memset(&h, 0, sizeof(h));
16.        memset(&peer, 0, sizeof(peer));
17.        h.msg_name = &peer;
18.        h.msg_namelen = sizeof(peer);
19.        h.msg_iov = (void*) &buf;
20.        h.msg iovlen = 1;
21.        // 调操作系统的函数读取数据
22.        do {
23.            nread = recvmsg(handle->io_watcher.fd, &h, 0);
24.        }
25.        while (nread == -1 && errno == EINTR);
26.        // 调用 C++层回调
27.        handle->recv_cb(handle,
28.                           nread,
29.                           &buf,
30.                           (const struct sockaddr*) &peer,
31.                           flags);
32.    }
33. }
```

最终通过操作系统调用 recvmsg 读取数据，操作系统收到一个 udp 数据包的时候，会挂载到 socket 的接收队列，如果接收队列满了则会丢弃，当用户调用 recvmsg 函数的时候，操作系统就把接收队列中节点逐个返回给用户。读取完后，Libuv 会回调 C++ 层，然后 C++ 层回调到 JS 层，最后触发 message 事件，这就是对应开始那段代码的 message 事件。

16.2.5 多播

我们知道，TCP 是基于连接和可靠的，多播则会带来过多的连接和流量，所以 TCP 是不支持多播的，而 UDP 则支持多播。多播分为局域网多播和广域网多播，我们知道在局域网内

发生一个数据，是会以广播的形式发送到各个主机的，主机根据目的地址判断是否需要处理该数据包。如果 UDP 是单播的模式，则只会有一个主机会处理该数据包。如果 UDP 是多播的模式，则有多个主机处理该数据包。多播的时候，存在一个多播组的概念，这就是 IGMP 做的事情。它定义了组的概念。只有加入这个组的主机才能处理该组的数据包。假设有以下局域网，如图 16-1 所示。

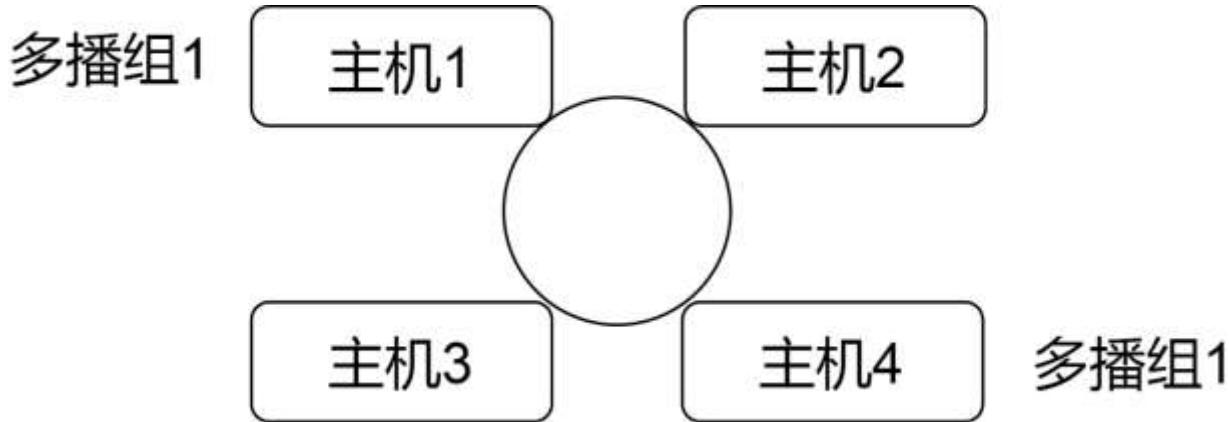


图 16-1

当主机 1 给多播组 1 发送数据的时候，主机 4 可以收到，主机 2，3 则无法收到。
我们再来看看广域网的多播。广域网的多播需要路由器的支持，多个路由器之间会使用多播路由协议交换多播组的信息。假设有以下广域网，如图 16-2 所示。

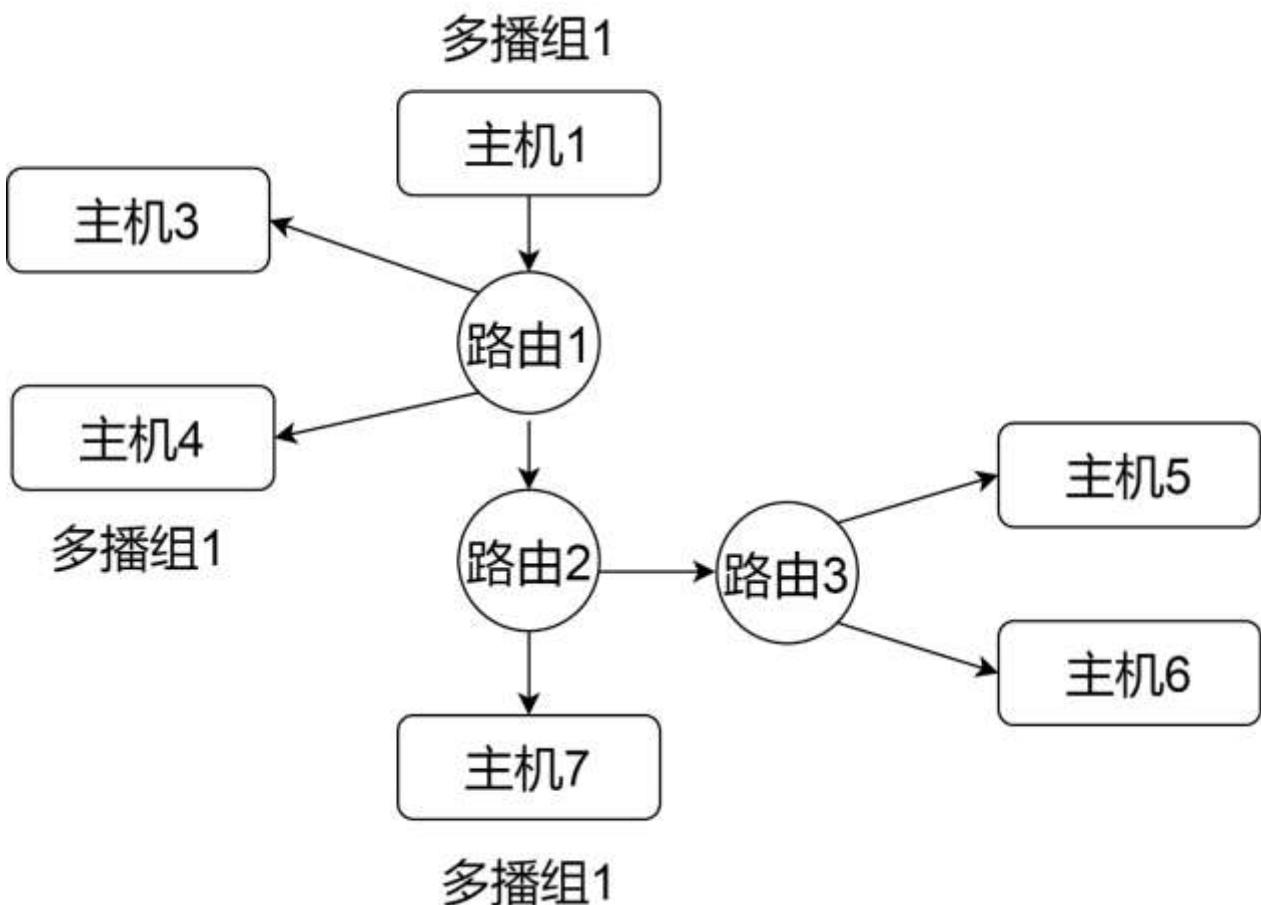


图 16-2

当主机 1 给多播组 1 发送数据的时候，路由器 1 会给路由器 2 发送一份数据（通过多播路由协议交换了信息，路由器 1 知道路由器 2 的主机 7 在多播组 1 中），但是路由器 2 不会给路由器 3 发送数据，因为它知道路由器 3 对应的网络中没有主机在多播组 1。

以上是多播的一些概念。Node.js 中关于多播的实现，基本是对操作系统 API 的封装，所以就不打算讲解，我们直接看操作系统中对于多播的实现。

16.2.5.1 加入一个多播组

可以通过以下接口加入一个多播组。

```

1. setsockopt(fd,
2.         IPPROTO_IP,
3.         IP_ADD_MEMBERSHIP,
4.         &mreq, // 记录出口 ip 和加入多播组的 ip
5.         sizeof(mreq));
  
```

mreq 的结构体定义如下

```
1. struct ip_mreq
2. {
3.     // 加入的多播组 ip
4.     struct in_addr imr_multiaddr;
5.     // 出口 ip
6.     struct in_addr imr_interface;
7. };
```

我们看一下 setsockopt 的实现（只列出相关部分代码）

```
1. case IP_ADD_MEMBERSHIP:
2. {
3.     struct ip_mreq mreq;
4.     static struct options optmem;
5.     unsigned long route_src;
6.     struct rtable *rt;
7.     struct device *dev=NULL;
8.     err=verify_area(VERIFY_READ, optval, sizeof(mreq));
9.     memcpy_fromfs(&mreq,optval,sizeof(mreq));
10.    // 没有设置 device 则根据多播组 ip 选择一个 device
11.    if(mreq.imr_interface.s_addr==INADDR_ANY)
12.    {
13.        if((rt=ip_rt_route(mreq.imr_multiaddr.s_addr,
14.                            &optmem, &route_src))!=NULL)
15.        {
16.            dev=rt->rt_dev;
17.            rt->rt_use--;
18.        }
19.    }
20.    else
21.    {
22.        // 根据设置的 ip 找到对应的 device
23.        for(dev = dev_base; dev; dev = dev->next)
24.        {
25.            // 在工作状态、支持多播, ip 一样
26.            if((dev->flags&IFF_UP)&&
27.                (dev->flags&IFF_MULTICAST)&&
28.                (dev->pa_addr==mreq.imr_interface.s_addr
29.                 ))
30.                break;
31.        }
32.    }
33.    // 加入多播组
34.    return ip_mc_join_group(sk,
35.                           dev,
```

```

36.                                     mreq.imr_multiaddr.s_addr);
37.     }
38.

```

首先拿到加入的多播组 IP 和出口 IP 对应的 device 后，调用 ip_mc_join_group，在 socket 结构体中，有一个字段维护了该 socket 加入的多播组信息，如图 16-3 所示。



图 16-3

我们接着看一下 ip_mc_join_group

```

1. int ip_mc_join_group(struct sock *sk ,
2.                      struct device *dev,
3.                      unsigned long addr)
4. {
5.     int unused= -1;
6.     int i;
7.     // 还没有加入过多播组则分配一个 ip_mc_socklist 结构体
8.     if(sk->ip_mc_list==NULL)
9.     {
10.         if((sk->ip_mc_list=(struct ip_mc_socklist *)kmalloc(sizeof(*sk->ip_mc_list), GFP_KERNEL))==NULL)
11.             return -ENOMEM;
12.         memset(sk->ip_mc_list, '\0', sizeof(*sk->ip_mc_list));
13.     }
14.     // 遍历加入的多播组队列，判断是否已经加入过
15.     for(i=0;i<IP_MAX_MEMBERSHIPS;i++)
16.     {
17.         if(sk->ip_mc_list->multiaddr[i]==addr &&
18.             sk->ip_mc_list->multidev[i]==dev)
19.             return -EADDRINUSE;
20.         if(sk->ip_mc_list->multidev[i]==NULL)
21.             // 记录可用位置的索引
22.             unused=i;
23.     }
24.     // 到这说明没有加入过当前设置的多播组，则记录并且加入
25.     if(unused== -1)

```

```
26.         return -ENOBUFS;
27.     sk->ip_mc_list->multiaddr[unused]=addr;
28.     sk->ip_mc_list->multidev[unused]=dev;
29.     // addr 为多播组 ip
30.     ip_mc_inc_group(dev,addr);
31.     return 0;
32. }
```

ip_mc_join_group 函数的主要逻辑是把 socket 想加入的多播组信息记录到 socket 的 ip_mc_list 字段中（如果还没有加入过该多播组的话）。接着调 ip_mc_inc_group 往下走。device 的 ip_mc_list 字段维护了主机中使用了该 device 的多播组信息，如图 16-4 所示。

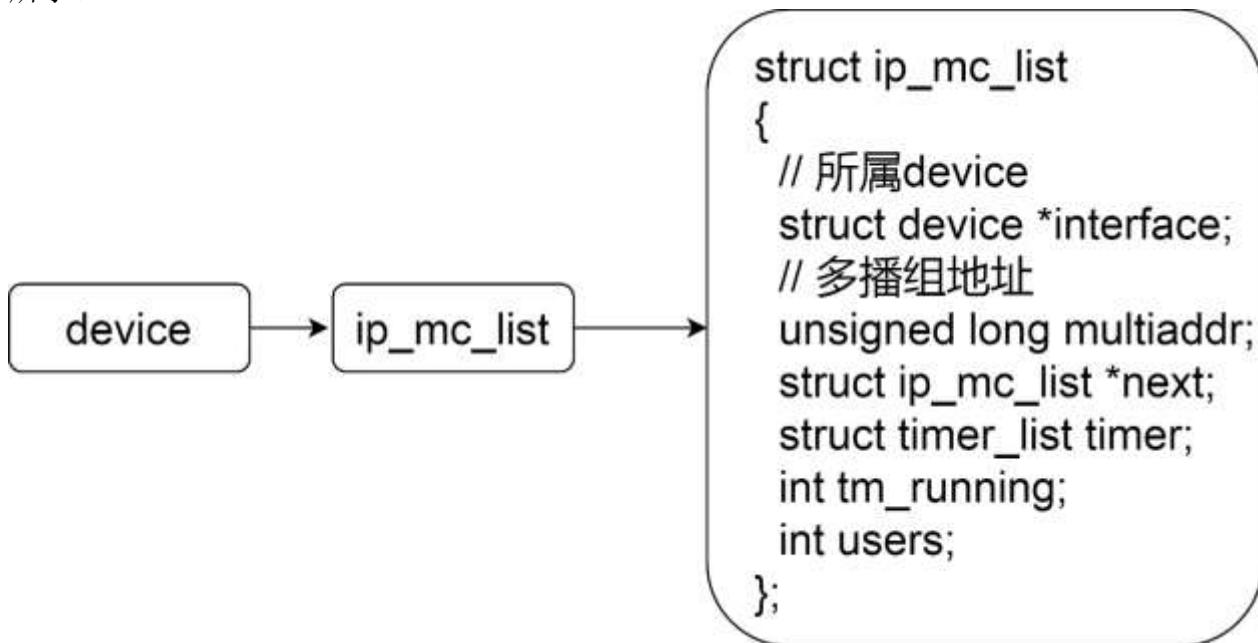


图 16-4

```
1. static void ip_mc_inc_group(struct device *dev,
2.                               unsigned long addr)
3. {
4.     struct ip_mc_list *i;
5.     /*
6.      遍历该设备维护的多播组队列,
7.      判断是否已经有 socket 加入过该多播组, 是则引用数加一
8.     */
9.     for(i=dev->ip_mc_list;i!=NULL;i=i->next)
10.    {
11.        if(i->multiaddr==addr)
12.        {
13.            i->users++;
14.            return;
```

```

15.         }
16.     }
17.     // 到这说明，还没有 socket 加入过当前多播组，则记录并加入
18.     i=(struct ip_mc_list *)kmalloc(sizeof(*i), GFP_KERNEL);
19.     if(!i)
20.         return;
21.     i->users=1;
22.     i->interface=dev;
23.     i->multiaddr=addr;
24.     i->next=dev->ip_mc_list;
25.     // 通过 igmp 通知其它方
26.     igmp_group_added(i);
27.     dev->ip_mc_list=i;
28. }
```

ip_mc_inc_group 函数的主要逻辑是判断 socket 想要加入的多播组是不是已经存在于当前 device 中，如果不是则新增一个节点。继续调用 igmp_group_added

```

1. static void igmp_group_added(struct ip_mc_list *im)
2. {
3.     // 初始化定时器
4.     igmp_init_timer(im);
5.     /*
6.      发送一个 igmp 数据包，同步多播组信息（socket 加入
7.      了一个新的多播组）
8.     */
9.     igmp_send_report(im->interface,
10.                      im->multiaddr,
11.                      IGMP_HOST_MEMBERSHIP_REPORT);
12.     // 转换多播组 ip 到多播 mac 地址，并记录到 device 中
13.     ip_mc_filter_add(im->interface, im->multiaddr);
14. }
```

我们看看 igmp_send_report 和 ip_mc_filter_add 的具体逻辑。

```

1. static void igmp_send_report(struct device *dev,
2.                               unsigned long address,
3.                               int type)
4. {
5.     // 申请一个 skb 表示一个数据包
6.     struct sk_buff *skb=alloc_skb(MAX_IGMP_SIZE, GFP_ATOMIC);
7.     int tmp;
8.     struct igmphdr *igh;
9.     /*
10.      构建 ip 头，ip 协议头的源 ip 是 INADDR_ANY,
```

```
11.     即随机选择一个本机的，目的 ip 为多播组 ip (address)
12. */
13.     tmp=ip_build_header(skb,
14.                           INADDR_ANY,
15.                           address,
16.                           &dev,
17.                           IPPROTO_IGMP,
18.                           NULL,
19.                           skb->mem_len, 0, 1);
20. */
21.     data 表示所有的数据部分，tmp 表示 ip 头大小，所以 igh
22.     就是 ip 协议的数据部分，即 igmp 报文的内容
23. */
24.     igh=(struct igmphdr *) (skb->data+tmp);
25.     skb->len=tmp+sizeof(*igh);
26.     igh->csum=0;
27.     igh->unused=0;
28.     igh->type=type;
29.     igh->group=address;
30.     igh->csum=ip_compute_csum((void *)igh,sizeof(*igh));
31.     // 调用 ip 层发送出去
32.     ip_queue_xmit(NULL,dev,skb,1);
33. }
```

igmp_send_report 其实就是构造一个 IGMP 协议数据包，然后发送出去，告诉路由器某个主机加入了多播组，IGMP 的协议格式如下

```
1. struct igmphdr
2. {
3.     // 类型
4.     unsigned char type;
5.     unsigned char unused;
6.     // 校验和
7.     unsigned short csum;
8.     // igmp 的数据部分，比如加入多播组的时候，group 表示多播组 ip
9.     unsigned long group;
10. }
```

接着我们看 ip_mc_filter_add

```
1. void ip_mc_filter_add(struct device *dev, unsigned long addr)
2. {
3.     char buf[6];
4.     // 把多播组 ip 转成 mac 多播地址
5.     addr=ntohl(addr);
```

```

6.     buf[0]=0x01;
7.     buf[1]=0x00;
8.     buf[2]=0x5e;
9.     buf[5]=addr&0xFF;
10.    addr>>=8;
11.    buf[4]=addr&0xFF;
12.    addr>>=8;
13.    buf[3]=addr&0x7F;
14.    dev_mc_add(dev,buf,ETH_ALEN,0);
15. }

```

我们知道 IP 地址是 32 位，mac 地址是 48 位，但是 IANA 规定，IP V4 组播 MAC 地址的高 24 位是 0x01005E，第 25 位是 0，低 23 位是 ipv4 组播地址的低 23 位。而多播的 IP 地址高四位固定是 1110。另外低 23 位被映射到 MAC 多播地址的 23 位，所以多播 IP 地址中，有 5 位是可以随机组合的。这就意味着，每 32 个多播 IP 地址，映射到一个 MAC 地址。这会带来一些问题，假设主机 x 加入了多播组 a，主机 y 加入了多播组 b，而 a 和 b 对应的 mac 多播地址是一样的。当主机 z 给多播组 a 发送一个数据包的时候，这时候主机 x 和 y 的网卡都会处理该数据包，并上报到上层，但是多播组 a 对应的 MAC 多播地址和多播组 b 是一样的。我们拿到一个多播组 ip 的时候，可以计算出它的多播 MAC 地址，但是反过来就不行，因为一个多播 mac 地址对应了 32 个多播 ip 地址。那主机 x 和 y 怎么判断是不是发给自己的数据包？因为 device 维护了一个本 device 上的多播 IP 列表，操作系统根据收到的数据包中的 IP 目的地址和 device 的多播 IP 列表对比。如果在列表中，则说明是发给自己的。最后我们看看 dev_mc_add。device 中维护了当前的 mac 多播地址列表，它会把这个列表信息同步到网卡中，使得网卡可以处理该列表中多播 mac 地址的数据包，如图 16-5 所示。

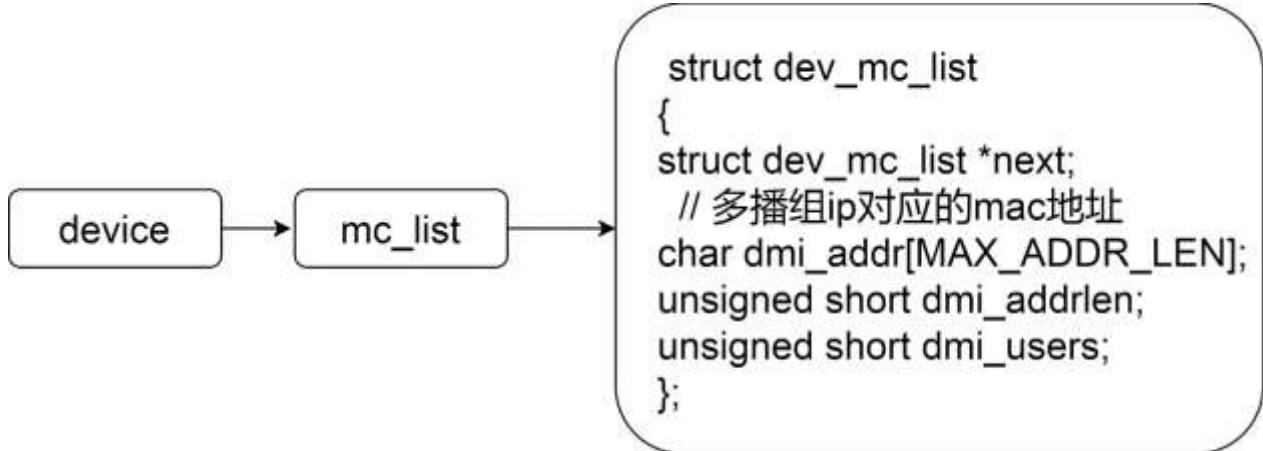


图 16-5

```

1. void dev_mc_add(struct device *dev, void *addr, int alen, int new
   only)
2. {
3.     struct dev_mc_list *dmi;

```

```
4.     // device 维护的多播 mac 地址列表
5.     for(dmi=dev->mc_list;dmi!=NULL;dmi=dmi->next)
6.     {
7.         // 已存在，则引用计数加一
8.         if(memcmp(dmi->dmi_addr,addr,dmi->dmi_addrlen)==0 &&
9.             dmi->dmi_addrlen==alen)
10.        {
11.            if(!newonly)
12.                dmi->dmi_users++;
13.            return;
14.        }
15.    }
16.    // 不存在则新增一个项到 device 列表中
17.    dmi=(struct dev_mc_list *)kmalloc(sizeof(*dmi), GFP_KERNEL);
18.    memcpy(dmi->dmi_addr, addr, alen);
19.    dmi->dmi_addrlen=alen;
20.    dmi->next=dev->mc_list;
21.    dmi->dmi_users=1;
22.    dev->mc_list=dmi;
23.    dev->mc_count++;
24.    // 通知网卡需要处理该多播 mac 地址
25.    dev_mc_upload(dev);
26. }
```

网卡的工作模式有几种，分别是正常模式（只接收发给自己的数据包）、混杂模式（接收所有数据包）、多播模式（接收一般数据包和多播数据包）。网卡默认是只处理发给自己的数据包，所以当我们加入一个多播组的时候，我们需要告诉网卡，当收到该多播组的数据包时，需要处理，而不是忽略。dev_mc_upload 函数就是通知网卡。

```
1. void dev_mc_upload(struct device *dev)
2. {
3.     struct dev_mc_list *dmi;
4.     char *data, *tmp;
5.     // 不工作了
6.     if(!(dev->flags&IFF_UP))
7.         return;
8.     /*
9.         当前是混杂模式，则不需要设置多播了，因为网卡会处理所有
10.        收到的数据，不管是发给自己的
11.     */
12.     if(dev->flags&IFF_PROMISC)
13.     {
14.         dev->set_multicast_list(dev, -1, NULL);
15.         return;
16.     }
```

```

17.  /*
18.      多播地址个数, 为 0, 则设置网卡工作模式为正常模式,
19.      因为不需要处理多播了
20. */
21. if(dev->mc_count==0)
22. {
23.     dev->set_multicast_list(dev,0,NULL);
24.     return;
25. }
26.
27. data=kmalloc(dev->mc_count*dev->addr_len, GFP_KERNEL);
28. // 复制所有的多播 mac 地址信息
29. for(tmp = data, dmi=dev->mc_list;dmi!=NULL;dmi=dmi->next)
30. {
31.     memcpy(tmp,dmi->dmi_addr, dmi->dmi_addrlen);
32.     tmp+=dev->addr_len;
33. }
34. // 告诉网卡
35. dev->set_multicast_list(dev,dev->mc_count,data);
36. kfree(data);
37. }

```

最后我们看一下 set_multicast_list

```

1. static void set_multicast_list(struct device *dev, int num_addrs,
   void *addrs)
2. {
3.     int ioaddr = dev->base_addr;
4.     // 多播模式
5.     if (num_addrs > 0) {
6.         outb(RX_MULT, RX_CMD);
7.         inb(RX_STATUS); /* Clear status. */
8.     } else if (num_addrs < 0) { // 混杂模式
9.         outb(RX_PROM, RX_CMD);
10.        inb(RX_STATUS);
11.    } else { // 正常模式
12.        outb(RX_NORM, RX_CMD);
13.        inb(RX_STATUS);
14.    }
15. }

```

set_multicast_list 就是设置网卡工作模式的函数。至此，我们就成功加入了一个多播组。离开一个多播组也是类似的过程。

16.2.5.2 维护多播组信息

加入多播组后，我们可以主动退出多播组，但是如果主机挂了，就无法主动退出了，所以多播路由也会定期向所有多播组的所有主机发送探测报文，所以主机需要监听来自多播路由的探测报文。

```
1. void ip_mc_allhost(struct device *dev)
2. {
3.     struct ip_mc_list *i;
4.     for(i=dev->ip_mc_list;i!=NULL;i=i->next)
5.         if(i->multiaddr==IGMP_ALL_HOSTS)
6.             return;
7.     i=(struct ip_mc_list *)kmalloc(sizeof(*i), GFP_KERNEL);
8.     if(!i)
9.         return;
10.    i->users=1;
11.    i->interface=dev;
12.    i->multiaddr=IGMP_ALL_HOSTS;
13.    i->next=dev->ip_mc_list;
14.    dev->ip_mc_list=i;
15.    ip_mc_filter_add(i->interface, i->multiaddr);
16. }
```

设备启动的时候，操作系统会设置网卡监听目的 IP 是 224.0.0.1 的报文，使得可以处理目的 IP 是 224.0.0.1 的多播消息。该类型的报文是多播路由用于查询局域网当前多播组情况的，比如查询哪些多播组已经没有成员了，如果没有成员则删除路由信息。我们看看如何处理某设备的 IGMP 报文。

```
1. int igmp_rcv(struct sk_buff *skb, struct device *dev, struct options *opt,
2.                 unsigned long daddr, unsigned short len, unsigned long saddr,
3.                 int redo,
4.                 struct inet_protocol *protocol)
5. {
6.     // IGMP 报头
7.     struct igmphdr *igh=(struct igmphdr *)skb->h.raw;
8.     // 该数据包是发给所有多播主机的，用于查询本多播组中是否还有成员
9.     if(igh->type==IGMP_HOST_MEMBERSHIP_QUERY && daddr==IGMP_ALL_H
10.        OSTS)
11.         igmp_heard_query(dev);
12.     // 该数据包是其它成员对多播路由查询报文的回复，同多播组的主机也会
13.        收到
14.     if(igh->type==IGMP_HOST_MEMBERSHIP_REPORT && daddr==igh->gro
15.        up)
```

```

12.         igmp_heard_report(dev, igh->group);
13.         kfree_skb(skb, FREE_READ);
14.         return 0;
15.     }

```

IGMP V1 只处理两种报文，分别是组成员查询报文（查询组是否有成员），其它成员回复多播路由的报告报文。组成员查询报文由多播路由发出，所有的多播组中的所有主机都可以收到。组成员查询报文的 IP 协议头的目的地址是 224.0.0.1 (IGMP_ALL_HOSTS)，代表所有的组播主机都可以处理该报文。我们看一下这两种报文的具体实现。

```

1. static void igmp_heard_query(struct device *dev)
2. {
3.     struct ip_mc_list *im;
4.     for(im=dev->ip_mc_list;im!=NULL;im=im->next)
5.         // IGMP_ALL_HOSTS 表示所有组播主机
6.         if(!im->tm_running && im->multiaddr!=IGMP_ALL_HOSTS)
7.             igmp_start_timer(im);
8. }

```

该函数用于处理组播路由的查询报文，`dev->ip_mc_list` 是该设备对应的所有多播组信息，这里针对该设备中的每一个多播组，开启对应的定时器，超时后会发送回复报文给多播路由。我们看一下开启定时器的逻辑。

```

1. // 开启一个定时器
2. static void igmp_start_timer(struct ip_mc_list *im)
3. {
4.     int tv;
5.     if(im->tm_running)
6.         return;
7.     tv=random()%10*HZ);           /* Pick a number any number 8 */
8.     im->timer.expires=tv;
9.     im->tm_running=1;
10.    add_timer(&im->timer);
11. }

```

随机选择一个超时时间，然后插入系统维护的定时器队列。为什么使用定时器，而不是立即回复呢？因为多播路由只需要知道某个多播组是否至少还有一个成员，如果说有的话就保存该多播组信息，否则就删除路由项。如果某个多播组在局域网中有多个成员，那么多个成员都会处理该报文，如果都立即响应，则会引起过多没有必要的流量，因为组播路由只需要收到一个响应就行。我们看看超时时的逻辑。

```

1. static void igmp_init_timer(struct ip_mc_list *im)
2. {
3.     im->tm_running=0;

```

```
4.     init_timer(&im->timer);
5.     im->timer.data=(unsigned long)im;
6.     im->timer.function=&igmp_timer_expire;
7. }
8.
9. static void igmp_timer_expire(unsigned long data)
10. {
11.     struct ip_mc_list *im=(struct ip_mc_list *)data;
12.     igmp_stop_timer(im);
13.     igmp_send_report(im->interface, im->multiaddr, IGMP_HOST_MEMBERSHIP_REPORT);
14. }
```

我们看到，超时后会执行 igmp_send_report 发送一个类型是 IGMP_HOST_MEMBERSHIP_REPORT 的 IGMP、目的 IP 是多播组 IP 的报文，说明该多播组还有成员。该报文不仅会发送给多播路由，还会发给同多播组的所有主机。其它主机也是类似的逻辑，即开启一个定时器。所以最快到期的主机会先发送回复报文给多播路由和同多播组的成员，我们看一下其它同多播组的主机收到该类报文时的处理逻辑。

```
1. // 成员报告报文并且多播组是当前设置关联的多播组
2. if(igh->type==IGMP_HOST_MEMBERSHIP_REPORT && daddr==igh->group)
3.     igmp_heard_report(dev,igh->group);
```

当一个多播组的其它成员针对多播路由的查询报文作了响应，因为该响应报文的目的 IP 是多播组 IP，所以该多播组的其它成员也能收到该报文。当某个主机收到该类型的报文的时候，就知道同多播组的其它成员已经回复了多播路由了，我们就不需要回复了。

```
1. /*
2.     收到其它组成员，对于多播路由查询报文的回复，则自己就不用回复了，
3.     因为多播路由知道该组还有成员，不会删除路由信息，减少网络流量
4. */
5. static void igmp_heard_report(struct device *dev, unsigned long address)
6. {
7.     struct ip_mc_list *im;
8.     for(im=dev->ip_mc_list;im!=NULL;im=im->next)
9.         if(im->multiaddr==address)
10.             igmp_stop_timer(im);
11. }
```

我们看到，这里会删除定时器。即不会作为响应了。

2.3 其它 socket 关闭，退出它之前加入过的多播

```

1. void ip_mc_drop_socket(struct sock *sk)
2. {
3.     int i;
4.
5.     if(sk->ip_mc_list==NULL)
6.         return;
7.
8.     for(i=0;i<IP_MAX_MEMBERSHIPS;i++)
9.     {
10.         if(sk->ip_mc_list->multidev[i])
11.         {
12.             ip_mc_dec_group(sk->ip_mc_list->multidev[i], sk->ip_
13.             mc_list->multiaddr[i]);
14.             sk->ip_mc_list->multidev[i]=NULL;
15.         }
16.         kfree_s(sk->ip_mc_list,sizeof(*sk->ip_mc_list));
17.         sk->ip_mc_list=NULL;
18.     }

```

设备停止工作了，删除对应的多播信息

```

1. void ip_mc_drop_device(struct device *dev)
2. {
3.     struct ip_mc_list *i;
4.     struct ip_mc_list *j;
5.     for(i=dev->ip_mc_list;i!=NULL;i=j)
6.     {
7.         j=i->next;
8.         kfree_s(i,sizeof(*i));
9.     }
10.    dev->ip_mc_list=NULL;
11. }

```

以上是 IGMP V1 版本的实现，在后续 V2 V3 版本了又增加了很多功能，比如离开组报文，针对离开报文中的多播组，增加特定组查询报文，用于查询某个组中是否还有成员，另外还有路由选举，当局域网中有多个多播路由，多播路由之间通过协议选举出 IP 最小的路由为查询路由，定时给多播组发送探测报文。然后成为查询器的多播路由，会定期给其它多播路由同步心跳。否则其它多播路由会在定时器超时时认为当前查询路由已经挂了，重新选举。

16.2.5.3 开启多播

UDP 的多播能力是需要用户主动开启的，原因是防止用户发送 UDP 数据包的时候，误传了一个多播地址，但其实用户是想发送一个单播的数据包。我们可以通过 setBroadcast 开启多播能力。我们看 Libuv 的代码。

```
1. int uv_udp_set_broadcast(uv_udp_t* handle, int on) {
2.     if (setsockopt(handle->io_watcher.fd,
3.                     SOL_SOCKET,
4.                     SO_BROADCAST,
5.                     &on,
6.                     sizeof(on))) {
7.         return UV__ERR(errno);
8.     }
9.
10.    return 0;
11. }
```

再看看操作系统的实现。

```
1. int sock_setsockopt(struct sock *sk, int level, int optname,
2.                      char *optval, int optlen){
3.     ...
4.     case SO_BROADCAST:
5.         sk->broadcast=val?1:0;
6. }
```

我们看到实现很简单，就是设置一个标记位。当我们发送消息的时候，如果目的地址是多播地址，但是又没有设置这个标记，则会报错。

```
1. if (!sk->broadcast && ip_chk_addr(sin.sin_addr.s_addr)==IS_BROADCAST)
2.     return -EACCES;
```

上面代码来自调用 udp 的发送函数（例如 sendto）时，进行的校验，如果发送的目的 ip 是多播地址，但是没有设置多播标记，则报错。

16.2.5.4 多播的问题

服务器

```
1. const dgram = require('dgram');
2. const udp = dgram.createSocket('udp4');
```

```

3.
4. udp.bind(1234, () => {
5.   // 局域网多播地址 (224.0.0.0~224.0.0.255, 该范围的多播数据包, 路
 由器不会转发)
6.   udp.addMembership('224.0.0.114');
7. });
8.
9. udp.on('message', (msg, rinfo) => {
10.   console.log(`receive msg: ${msg} from ${rinfo.address}:${rinfo.port}`);
11. });

```

服务器绑定 1234 端口后，加入多播组 224.0.0.114，然后等待多播数据的到来。

客户端

```

1. const dgram = require('dgram');
2. const udp = dgram.createSocket('udp4');
3. udp.bind(1234, () => {
4.   udp.addMembership('224.0.0.114');
5. });
6. udp.send('test', 1234, '224.0.0.114', (err) => {});

```

客户端绑定 1234 端口后，也加入了多播组 224.0.0.114，然后发送数据，但是发现服务端没有收到数据，客户端打印了 receive msg test from 169.254.167.41:1234。这怎么多了一个 IP 出来？原来我主机有两个局域网地址。当我们加入多播组的时候，不仅可以设置加入哪个多播组，还能设置出口的设备和 IP。当我们调用 `udp.addMembership('224.0.0.114')` 的时候，我们只是设置了我们加入的多播组，没有设置出口。这时候操作系统会为我们选择一个。根据输出，我们发现操作系统选择的是 169.254.167.41（子网掩码是 255.255.0.0）。因为这个 IP 和 192 开头的那个不是同一子网，但是我们加入的是局域网的多播 IP，所有服务端无法收到客户端发出的数据包。下面是 Node.js 文档的解释。

Tells the kernel to join a multicast group at the given `multicastAddress` and `multicastInterface` using the `IP_ADD_MEMBERSHIP` socket option. If the `multicastInterface` argument is not specified, the operating system will choose one interface and will add membership to it. To add membership to every available interface, call `addMembership` multiple times, once per interface.

我们看一下操作系统的相关逻辑。

```

1. if(MULTICAST(daddr) && *dev==NULL && skb->sk && *skb->sk->ip_mc_name)
2.   *dev=dev_get(skb->sk->ip_mc_name);

```

上面的代码来自操作系统发送 IP 数据包时的逻辑，如果目的 IP 似乎多播地址并且 ip_mc_name 非空（即我们通过 addMembership 第二个参数设置的值），则出口设备就是我们设置的值。否则操作系统自己选。所以我们需要显示指定这个出口，把代码改成 udp.addMembership('224.0.0.114', '192.168.8.164');重新执行发现客户端和服务器都显示了 receive msg test from 192.168.8.164:1234。为什么客户端自己也会收到呢？原来操作系统发送多播数据的时候，也会给自己发送一份。我们看看相关逻辑

```
1. // 目的地是多播地址，并且不是回环设备
2. if (MULTICAST(iph->daddr) && !(dev->flags&IFF_LOOPBACK))
3. {
4.     // 是否需要给自己一份，默认为 true
5.     if(sk==NULL || sk->ip_mc_loop)
6.     {
7.         // 给所有多播组的所有主机的数据包，则直接给自己一份
8.         if(iph->daddr==IGMP_ALL_HOSTS)
9.             ip_loopback(dev,skb);
10.        else
11.        {
12.            // 判断目的 ip 是否在当前设备的多播 ip 列表中，是的回传一份
13.            struct ip_mc_list *imc=dev->ip_mc_list;
14.            while(imc!=NULL)
15.            {
16.                if(imc->multiaddr==iph->daddr)
17.                {
18.                    ip_loopback(dev,skb);
19.                    break;
20.                }
21.                imc=imc->next;
22.            }
23.        }
24.    }
25. }
```

以上代码来自 IP 层发送数据包时的逻辑。如果我们设置了 sk->ip_mc_loop 字段为 1，并且数据包的目的 IP 在出口设备的多播列表中，则需要给自己回传一份。那么我们如何关闭这个特性呢？调用 udp.setMulticastLoopback(false)就可以了。

16.2.5.5 其它功能

UDP 模块还提供了其它一些功能

1 获取本端地址 address

如果用户没有显示调用 bind 绑定自己设置的 IP 和端口，那么操作系统就会随机选择。通过 address 函数就可以获取操作系统选择的源 IP 和端口。

2 获取对端的地址

通过 remoteAddress 函数可以获取对端地址。该地址由用户调用 connect 或 sendto 函数时设置。

3 获取/设置缓冲区大小 get/setRecvBufferSize, get/setSendBufferSize

4 setMulticastLoopback

发送多播数据包的时候，如果多播 IP 在出口设备的多播列表中，则给回环设备也发一份。

5 setMulticastInterface

设置多播数据的出口设备

6 加入或退出多播组 addMembership/dropMembership

7 addSourceSpecificMembership/dropSourceSpecificMembership

这两个函数是设置本端只接收特性源（主机）的多播数据包。

8 setTTL

单播 ttl（单播的时候，IP 协议头中的 ttl 字段）。

9 setMulticastTTL

多播 ttl（多播的时候，IP 协议的 ttl 字段）。

10 ref/unref

这两个函数设置如果 Node.js 主进程中只有 UDP 对应的 handle 时，是否允许 Node.js 退出。Node.js 事件循环的退出的条件之一是是否还有 ref 状态的 handle。这些都是对操作系统 API 的封装，就不一一分析。

16.2.6 端口复用

我们在网络编程中经常会遇到端口重复绑定的错误，根据到底是我们不能绑定到同一个端口和 IP 两次。但是在 UDP 中，这是允许的，这就是端口复用的功能，在 TCP 中，我们通过端口复用来解决服务器重启时重新绑定到同一个端口的问题，因为我们知道端口有一个 2msl 的等待时间，重启服务器重新绑定到这个端口时，默认会报错，但是如果设置了端口复用（Node.js 自动帮我们设置了），则可以绕过这个限制。UDP 中也支持端口复用的功能，但是功能、用途和 TCP 的不太一样。因为多个进程可以绑定同一个 IP 和端口。但是一般只用于多播的情况下。下面我们来分析一下 udp 端口复用的逻辑。在 Node.js 中，使用 UDP 的时候，可以通过 reuseAddr 选项使得进程可以复用端口，并且每一个想复用端口的 socket 都需要设置 reuseAddr。我们看一下 Node.js 中关于 reuseAddr 的逻辑。

```

1. Socket.prototype.bind = function(port_, address_ /* , callback */) {
2.     let flags = 0;
3.     if (state.reuseAddr)
4.         flags |= UV_UDP_REUSEADDR;
5.     state.handle.bind(ip, port || 0, flags);
6. };

```

我们看到 Node.js 在 bind 的时候会处理 reuseAddr 字段。我们直接看 Libuv 的逻辑。

```
1. int uv_udp_bind(uv_udp_t* handle,
2.                   const struct sockaddr* addr,
3.                   unsigned int addrlen,
4.                   unsigned int flags) {
5.     if (flags & UV_UDP_REUSEADDR) {
6.         err = uv_set_reuse(fd);
7.     }
8.     bind(fd, addr, addrlen))
9.     return 0;
10. }
11.
12. static int uv_set_reuse(int fd) {
13.     int yes;
14.     yes = 1;
15.
16.     if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(yes)))
17.         return UV_ERR(errno);
18.     return 0;
19. }
```

我们看到 Libuv 通过最终通过 setsockopt 设置了端口复用，并且是在 bind 之前。我们不妨再深入一点，看一下 Linux 内核的实现。

```
1. asmlinkage long sys_setsockopt(int fd, int level, int optname, char __user *optval, int
   optlen)
2. {
3.     int err;
4.     struct socket *sock;
5.
6.     if (optlen < 0)
7.         return -EINVAL;
8.
9.     if ((sock = sockfd_lookup(fd, &err))!=NULL)
10.    {
11.        if (level == SOL_SOCKET)
12.            err=sock_setsockopt(sock,level,optname,optval,optlen);
13.        else
14.            err=sock->ops->setsockopt(sock, level, optname, optval, optlen);
15.        sockfd_put(sock);
16.    }
17.    return err;
18. }
```

sys_setsockopt 是 setsockopt 对应的系统调用，我们看到 sys_setsockopt 也只是个入口函数，具体函数是 sock_setsockopt。

```
1. int sock_setsockopt(struct socket *sock, int level, int optname,
2.                      char __user *optval, int optlen)
3. {
4.     struct sock *sk=sock->sk;
5.     int val;
6.     int valbool;
7.     int ret = 0;
8.
9.     if (get_user(val, (int __user *)optval))
10.        return -EFAULT;
```

```

11.
12.     valbool = val?1:0;
13.
14.     lock_sock(sk);
15.
16.     switch(optname)
17.     {
18.         case SO_REUSEADDR:
19.             sk->sk_reuse = valbool;
20.             break;
21.         // ...
22.     release_sock(sk);
23.     return ret;
24. }

```

操作系统的处理很简单，只是做了一个标记。接下来我们看一下 bind 的时候是怎么处理的，因为端口是否重复和能否复用是在 bind 的时候判断的。这也是为什么在 TCP 中，即使两个进程不能绑定到同一个 IP 和端口，但是如果我们在主进程里执行了 bind 之后，再 fork 函数时，是可以实现绑定同一个 IP 端口的。言归正传我们看一下 UDP 中执行 bind 时的逻辑。

```

1. int inet_bind(struct socket *sock, struct sockaddr *uaddr, int addr_len)
2. {
3.     if (sk->sk_prot->get_port(sk, snum)) {
4.         inet->saddr = inet->recv_saddr = 0;
5.         err = -EADDRINUSE;
6.         goto out_release_sock;
7.     }
8.
9. }

```

每个协议都可以实现自己的 get_port 钩子函数。用来判断当前的端口是否允许被绑定。如果不允许则返回 EADDRINUSE，我们看看 UDP 协议的实现。

```

1. static int udp_v4_get_port(struct sock *sk, unsigned short snum)
2. {
3.     struct hlist_node *node;
4.     struct sock *sk2;
5.     struct inet_sock *inet = inet_sk(sk);
6.     // 通过端口找到对应的链表，然后遍历链表
7.     sk_for_each(sk2, node, &udp_hash[snum & (UDP_HTABLE_SIZE - 1)]) {
8.         struct inet_sock *inet2 = inet_sk(sk2);
9.         // 端口已使用，则判断是否可以复用
10.        if (inet2->num == snum &&
11.            sk2 != sk &&
12.            (!inet2->recv_saddr || !inet->recv_saddr ||
13.             inet2->recv_saddr == inet->recv_saddr) &&
14.             // 每个 socket 都需要设置端口复用标记
15.             (!sk2->sk_reuse || !sk->sk_reuse))
16.             // 不可以复用，报错
17.             goto fail;
18.     }
19. }
20. // 可以复用
21. inet->num = snum;
22. if (sk_unhashed(sk)) {

```

```
23.     // 找到端口对应的位置
24.     struct hlist_head *h = &udp_hash[snum & (UDP_HTABLE_SIZE - 1)];
25.     // 插入链表
26.     sk_add_node(sk, h);
27.     sock_prot_inc_use(sk->sk_prot);
28. }
29. return 0;
30.
31. fail:
32.     write_unlock_bh(&udp_hash_lock);
33.     return 1;
34. }
```

分析之前我们先看一下操作系统的一些数据结构，UDP 协议的实现中，会使用如下的数据结构记录每一个 UDP socket，如图 16-6 所示。

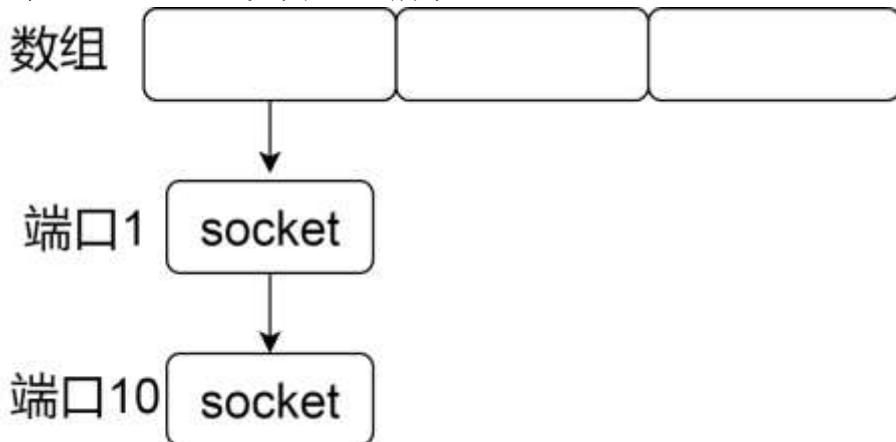


图 16-6

我们看到操作系统使用一个数组作为哈希表，每次操作一个 socket 的时候，首先会根据 socket 的源端口和哈希算法计算得到一个数组索引，然后把 socket 插入索引锁对应的链表中，即哈希冲突的解决方法是链地址法。回到代码的逻辑，当用户想绑定一个端口的时候，操作系统会根据端口拿到对应的 socket 链表，然后逐个判断是否有相等的端口，如果有则判断是否可以复用。例如两个 socket 都设置了复用标记则可以复用。最后把 socket 插入到链表中。

```
1. static inline void hlist_add_head(struct hlist_node *n, struct hlist_head *h)
2. {
3.     // 头结点
4.     struct hlist_node *first = h->first;
5.     n->next = first;
6.     if (first)
7.         first->pnext = &n->next;
8.     h->first = n;
9.     n->pnext = &h->first;
10. }
```

我们看到操作系统是以头插法的方式插入新节点的。接着我们看一下操作系统是如何使用这些数据结构的。

16.2.6.1 多播

我们先看一个例子，我们在同主机上新建两个 JS 文件（客户端），代码如下

```

1. const dgram = require('dgram');
2. const udp = dgram.createSocket({type: 'udp4', reuseAddr: true});
3. udp.bind(1234, '192.168.8.164', () => {
4.   udp.addMembership('224.0.0.114', '192.168.8.164');
5. });
6. udp.on('message', (msg) => {
7.   console.log(msg)
8. });

```

上面代码使得两个进程都监听了同样的 IP 和端口。接下来我们写一个 UDP 服务器。

```

1. const dgram = require('dgram');
2. const udp = dgram.createSocket({type: 'udp4'});
3. const socket = udp.bind(5678);
4. socket.send('hi', 1234, '224.0.0.114', (err) => {
5.   console.log(err)
6. });

```

上面的代码给一个多播组发送了一个数据，执行上面的代码，我们可以看到两个客户端进程都收到了数据。我们看一下收到数据时，操作系统是如何把数据分发给每个监听了同样 IP 和端口的进程的。下面是操作系统收到一个 UDP 数据包时的逻辑。

```

1. int udp_rcv(struct sk_buff *skb)
2. {
3.   struct sock *sk;
4.   struct udphdr *uh;
5.   unsigned short ulen;
6.   struct rtable *rt = (struct rtable*)skb->dst;
7.   // ip 头中记录的源 ip 和目的 ip
8.   u32 saddr = skb->nh.iph->saddr;
9.   u32 daddr = skb->nh.iph->daddr;
10.  int len = skb->len;
11.  // udp 协议头结构体
12.  uh = skb->h.uh;
13.  ulen = ntohs(uh->len);
14.  // 广播或多播包
15.  if(rt->rt_flags & (RTCF_BROADCAST|RTCF_MULTICAST))
16.    return udp_v4_mcast_deliver(skb, uh, saddr, daddr);
17.  // 单播
18.  sk = udp_v4_lookup(saddr, uh->source, daddr, uh->dest, skb->dev->ifindex);
19.  // 找到对应的 socket
20.  if (sk != NULL) {
21.    // 把数据插到 socket 的消息队列
22.    int ret = udp_queue_rcv_skb(sk, skb);
23.    sock_put(sk);
24.    if (ret > 0)
25.      return -ret;
26.    return 0;
27.  }
28.  return(0);
29. }

```

我们看到单播和非单播时处理逻辑是不一样的，我们先看一下非单播的情况

```
1. static int udp_v4_mcast_deliver(struct sk_buff *skb, struct udphdr *uh,
2.                                     u32 saddr, u32 daddr)
3. {
4.     struct sock *sk;
5.     int dif;
6.
7.     read_lock(&udp_hash_lock);
8.     // 通过端口找到对应的链表
9.     sk = sk_head(&udp_hash[ntohs(uh->dest) & (UDP_HTABLE_SIZE - 1)]);
10.    dif = skb->dev->ifindex;
11.    sk = udp_v4_mcast_next(sk, uh->dest, daddr, uh->source, saddr, dif);
12.    if (sk) {
13.        struct sock *sknext = NULL;
14.        // 遍历每一个需要处理该数据包的 socket
15.        do {
16.            struct sk_buff *skb1 = skb;
17.            sknext = udp_v4_mcast_next(sk_next(sk),
18.                                         uh->dest, daddr,
19.                                         uh->source,
20.                                         saddr,
21.                                         dif);
22.            if(sknex)
23.                // 复制一份
24.                skb1 = skb_clone(skb, GFP_ATOMIC);
25.                // 插入每一个 socket 的数据包队列
26.                if(skb1) {
27.                    int ret = udp_queue_rcv_skb(sk, skb1);
28.                    if (ret > 0)
29.                        kfree_skb(skb1);
30.                }
31.                sk = sknext;
32.            } while(sknex);
33.        } else
34.            kfree_skb(skb);
35.        read_unlock(&udp_hash_lock);
36.        return 0;
37.    }
```

在非单播的情况下，操作系统会遍历链表找到每一个可以接收该数据包的 socket，然后把数据包复制一份，挂载到 socket 的接收队列。这就解释了本节开头的例子，即两个客户端进程都会收到 UDP 数据包。

16.2.6.2 单播

接着我们再来看一下单播的情况。首先我们看一个例子。我们同样新建两个 JS 文件用作客户端。

```
1. const dgram = require('dgram');
2. const udp = dgram.createSocket({type: 'udp4', reuseAddr: true});
3. const socket = udp.bind(5678);
4. socket.on('message', (msg) => {
5.     console.log(msg)
6. })
```

然后再新建一个 JS 文件用作服务器。

```

1. const dgram = require('dgram');
2. const udp = dgram.createSocket({type: 'udp4'});
3. const socket = udp.bind(1234);
4. udp.send('hi', 5678)

```

执行以上代码，首先执行客户端，再执行服务器，我们会发现只有一个进程会收到数据。下面我们分析具体的原因，单播时收到会调用 `udp_v4_lookup` 函数找到接收该 UDP 数据包的 socket，然后把数据包挂载到 socket 的接收队列中。我们看看 `udp_v4_lookup`。

```

1. static __inline__ struct sock *udp_v4_lookup(u32 saddr, u16 sport,
2.                                              u32 daddr, u16 dport, int dif)
3. {
4.     struct sock *sk;
5.     sk = udp_v4_lookup_longway(saddr, sport, daddr, dport, dif);
6.     return sk;
7. }
8.
9. static struct sock *udp_v4_lookup_longway(u32 saddr, u16 sport,
10.                                            u32 daddr, u16 dport, int dif)
11. {
12.     struct sock *sk, *result = NULL;
13.     struct hlist_node *node;
14.     unsigned short hnum = ntohs(dport);
15.     int badness = -1;
16.     // 遍历端口对应的链表
17.     sk_for_each(sk, node, &udp_hash[hnum & (UDP_HTABLE_SIZE - 1)]) {
18.         struct inet_sock *inet = inet_sk(sk);
19.
20.         if (inet->num == hnum && !ipv6_only_sock(sk)) {
21.             int score = (sk->sk_family == PF_INET ? 1 : 0);
22.             if (inet->rcv_saddr) {
23.                 if (inet->rcv_saddr != daddr)
24.                     continue;
25.                 score+=2;
26.             }
27.             if (inet->daddr) {
28.                 if (inet->daddr != saddr)
29.                     continue;
30.                 score+=2;
31.             }
32.             if (inet->dport) {
33.                 if (inet->dport != sport)
34.                     continue;
35.                 score+=2;
36.             }
37.             if (sk->sk_bound_dev_if) {
38.                 if (sk->sk_bound_dev_if != dif)
39.                     continue;
40.                 score+=2;
41.             }
42.             // 全匹配，直接返回，否则记录当前最好的匹配结果
43.             if(score == 9) {
44.                 result = sk;
45.                 break;
46.             } else if(score > badness) {
47.                 result = sk;
48.                 badness = score;

```

```
49.         }
50.     }
51. }
52. return result;
53. }
```

我们看到代码很多，但是逻辑并不复杂，操作系统收到根据端口从哈希表中拿到对应的链表，然后遍历该链表找出最匹配的 socket。然后把数据挂载到 socket 上。但是有一个细节需要注意，如果有两个进程都监听了同一个 IP 和端口，那么哪一个进程会收到数据呢？这个取决于操作系统的实现，从 Linux 源码我们看到，插入 socket 的时候是使用头插法，查找的时候是从头开始找最匹配的 socket。即后面插入的 socket 会先被搜索到。但是 Windows 下结构却相反，先监听了该 IP 端口的进程会收到数据。

第十七章 TCP

本章我们主要看一下 Node.js 中对 TCP 的封装，我们首先看一下在网络编程中，是如何编写一个服务器和客户端的（伪代码）。

服务器

```
1. const fd = socket();
2. bind(fd, ip, port);
3. listen(fd);
4. const acceptedFd = accept(fd);
5. handle(acceptedFd);
```

我们看一下这几个函数的作用

1 **socket**: **socket** 函数用于从操作系统申请一个 **socket** 结构体，Linux 中万物皆文件，所以最后操作系统会返回一个 **fd**，**fd** 在操作系统中类似数据库的 **id**，操作系统底层维护了 **fd** 对应的资源，比如网络、文件、管道等，后续就可以通过该 **fd** 去操作对应的资源。

2 **bind**: **bind** 函数用于给 **fd** 对应的 **socket** 设置地址（IP 和端口），后续需要用到。

3 **listen**: **listen** 函数用于修改 **fd** 对应的 **socket** 的状态和监听状态。只有监听状态的 **socket** 可以接受客户端的连接。**socket** 我们可以理解有两种，一种是监听型的，一种是通信型的，监听型的 **socket** 只负责处理三次握手，建立连接，通信型的负责和客户端通信。

4 **accept**: **accept** 函数默认会阻塞进程，直到有连接到来并完成三次握手。

执行完以上代码，就完成了一个服务器的启动。这时候关系图如图 17-1 所示。

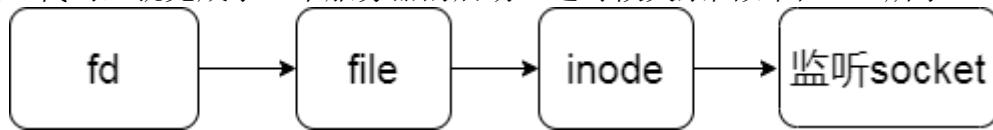


图 17-1

客户端

```
1. const fd = socket();
2. const connectRet = connect(fd, ip, port);
```

```
|3. write(fd, 'hello');
```

客户端比服务器稍微简单一点，我们看看这几个函数的作用。

1 socket: 和服务器一样，客户端也需要申请一个 socket 用于和服务器通信。

2 connect: connect 会开始三次握手过程，默认情况下会阻塞进程，直到连接有结果，连接结果通过返回值告诉调用方，如果三次握手完成，那么我们就可以开始发送数据了。

3 write: write 用于给服务器发送数据，不过并不是直接发送，这些数据只是保存到 socket 的发送缓冲区，底层会根据 TCP 协议决定什么时候发送数据。

我们看一下当客户端发送第一个握手的 syn 包时，socket 处于 syn 发送状态，我们看看这时候的服务器是怎样的，如图 17-2 所示。

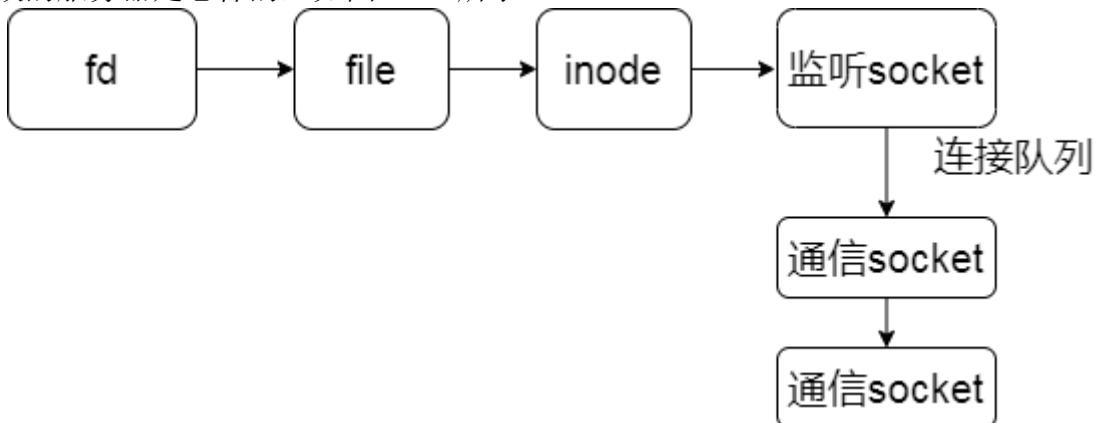


图 17-2

我们看到这时候，服务器对应的 socket 中，会新建一个 socket 用于后续通信（socket 结构体有一个字段指向该队列）。并且标记该 socket 的状态为收到 syn，然后发送 ack，即第二次握手，等到客户端回复第三次握手的数据包时，就完成了连接的建立。不同的操作系统版本实现不一样，有的版本实现中，已完成连接和正在建立连接的 socket 是在一个队列中的，有的版本实现中，已完成连接和正在建立连接的 socket 是分为两个队列维护的。

当客户端和服务器完成了 TCP 连接后，就可以进行数据通信了，这时候服务器的 accept 就会从阻塞中被唤醒，并从连接队列中摘下一个已完成连接的 socket 结点，然后生成一个新的 fd。后续就可以在该 fd 上和对端通信。那么当客户端发送一个 TCP 数据包过来的时候，操作系统是如何处理的呢？

1 操作系统首先根据 TCP 报文的源 IP、源端口、目的 IP、目的端口等信息从 socket 池中找到对应的 socket。

2 操作系统判断读缓冲区是否还有足够的空间，如果空间不够，则丢弃 TCP 报文，否则把报文对应的数据结构挂载到 socket 的数据队列，等待读取。

了解了 TCP 通信的大致过程后，我们看一下 Node.js 中是如何封装底层的能力的。

17.1 TCP 客户端

17.1.1 建立连接

net.connect 是 Node.js 中发起 TCP 连接的 API。本质上是对底层 TCP connect 函数的封装。connect 返回一个表示客户端的 Socket 对象。我们看一下 Node.js 中的具体实现。我们首先看一下 connect 函数的入口定义。

```
1. function connect(...args) {  
2.     // 处理参数  
3.     var normalized = normalizeArgs(args);  
4.     var options = normalized[0];  
5.     // 申请一个 socket 表示一个客户端  
6.     var socket = new Socket(options);  
7.     // 设置超时，超时后会触发 timeout，用户可以自定义处理超时逻辑  
8.     if (options.timeout) {  
9.         socket.setTimeout(options.timeout);  
10.    }  
11.    // 调用 socket 的 connect  
12.    return Socket.prototype.connect.call(socket, normalized);  
13.}
```

从代码中可以看到，connect 函数是对 Socket 对象的封装。Socket 表示一个 TCP 客户端。我们分成三部分分析。

1 new Socket
2 setTimeout
3 Socket 的 connect

1 new Socket

我们看看新建一个 Socket 对象，做了什么事情。

```
1. function Socket(options) {  
2.     // 是否正在建立连接，即三次握手中  
3.     this.connecting = false;  
4.     // 触发 close 事件时，该字段标记是否由于错误导致了 close  
5.     this._hadError = false;  
6.     // 对应的底层 handle，比如 tcp_wrap  
7.     this._handle = null;  
8.     // 定时器 id  
9.     this[kTimeout] = null;  
10.    options = options || {};
```

```

11.    // socket 是双向流
12.    stream.Duplex.call(this, options);
13.    // 还不能读写, 先设置成 false, 连接成功后再重新设置
14.    this.readable = this.writable = false;
15.    // 注册写端关闭的回调
16.    this.on('finish', onSocketFinish);
17.    // 注册读端关闭的回调
18.    this.on('_socketEnd', onSocketEnd);
19.    // 是否允许半开关, 默认不允许
20.    this.allowHalfOpen = options && options.allowHalfOpen || false;
21. }

```

Socket 是对 C++ 模块 `tcp_wrap` 的封装。主要是初始化了一些属性和监听一些事件。

2 setTimeout

```

1. Socket.prototype.setTimeout = function(msecs, callback) {
2.     // 清除之前的, 如果有的话
3.     clearTimeout(this[kTimeout]);
4.     // 0 代表清除
5.     if (msecs === 0) {
6.         if (callback) {
7.             this.removeListener('timeout', callback);
8.         }
9.     } else {
10.         // 开启一个定时器, 超时时间是 msecs, 超时回调是_onTimeout
11.         this[kTimeout] = setUnrefTimeout(this._onTimeout.bind(this),
12.                                         msecs);
13.         /*
14.             监听 timeout 事件, 定时器超时时, 底层会调用 Node.js 的回调,
15.             Node.js 会调用用户的回调 callback
16.         */
17.         if (callback) {
18.             this.once('timeout', callback);
19.         }
20.     }
21.     return this;
22. };

```

`setTimeout` 做的事情就是设置一个超时时间，这个时间用于检测 `socket` 的活跃情况（比如有数据通信），当 `socket` 活跃时，`Node.js` 会重置该定时器，如果 `socket` 一直不活跃则超时会触发 `timeout` 事件，从而执行 `Node.js` 的 `_onTimeout` 回调，在回调里再触发用户传入的回调。我们看一下超时处理函数 `_onTimeout`。

```
1. Socket.prototype._onTimeout = function() {
2.     this.emit('timeout');
3. };
```

直接触发 timeout 函数，回调用户的函数。我们看到 setTimeout 只是设置了一个定时器，然后触发 timeout 事件，Node.js 并没有帮我们做额外的操作，所以我们需要自己处理，比如关闭 socket。

```
1. socket.setTimeout(10000);
2. socket.on('timeout', () => {
3.     socket.close();
4. });
```

另外我们看到这里是使用 setUnrefTimeout 设置的定时器，因为这一类定时器不应该阻止事件循环的退出。

3 connect 函数

在第一步我们已经创建了一个 socket，接着我们调用该 socket 的 connect 函数开始发起连接。

```
1. // 建立连接，即三次握手
2. Socket.prototype.connect = function(...args) {
3.     let normalized;
4.     /* 忽略参数处理 */
5.     var options = normalized[0];
6.     var cb = normalized[1];
7.     // TCP 在 tcp_wrap.cc 中定义
8.     this._handle = new TCP(TCPConstants.SOCKET);
9.     // 有数据可读时的回调
10.    this._handle.onread = onread;
11.    // 连接成功时执行的回调
12.    if (cb !== null) {
13.        this.once('connect', cb);
14.    }
15.    // 正在连接
16.    this.connecting = true;
17.    this.writable = true;
18.    // 重置定时器
19.    this._unrefTimer();
20.    // 可能需要 DNS 解析，解析成功再发起连接
21.    lookupAndConnect(this, options);
22.    return this;
23.};
```

connect 函数主要是三个逻辑

1 首先通过 new TCP() 创建一个底层的 handle，比如我们这里是 TCP（对应 tcp_wrap.cc 的实现）。

2 设置一些回调

3 做 DNS 解析（如果需要的话），然后发起三次握手。

我们看一下 new TCP 意味着什么，我们看 tcp_wrap.cc 的实现

```

1. void TCPWrap::New(const FunctionCallbackInfo<Value>& args) {
2.     // 要以 new TCP 的形式调用
3.     CHECK(args.IsConstructCall());
4.     // 第一个入参是数字
5.     CHECK(args[0]->IsInt32());
6.     Environment* env = Environment::GetCurrent(args);
7.     // 作为客户端还是服务器
8.     int type_value = args[0].As<Int32>()->Value();
9.     TCPWrap::SocketType type = static_cast<TCPWrap::SocketType>(type_
    value);
10.
11.    ProviderType provider;
12.    switch (type) {
13.        // 作为客户端，即发起连接方
14.        case SOCKET:
15.            provider = PROVIDER_TCPWRAP;
16.            break;
17.        // 作为服务器
18.        case SERVER:
19.            provider = PROVIDER_TCPSERVERWRAP;
20.            break;
21.        default:
22.            UNREACHABLE();
23.    }
24.    new TCPWrap(env, args.This(), provider);
25. }
```

new TCP 对应到 C++ 层，就是创建一个 TCPWrap 对象。并初始化对象中的 handle_ 字段

```

1. TCPWrap::TCPWrap(Environment* env,
2.                     Local<Object> object,
3.                     ProviderType provider)
4.         : ConnectionWrap(env, object, provider) {
5.     int r = uv_tcp_init(env->event_loop(), &handle_);
6. }
```

初始化完底层的数据结构后，我们继续看 lookupAndConnect，lookupAndConnect 主要是对参数进行校验，然后进行 DNS 解析（如果传的是域名的话），DNS 解析成功后执行 internalConnect

```
1. function internalConnect(
2.     self,
3.     // 需要连接的远端 IP、端口
4.     address,
5.     port,
6.     addressType,
7.     /*
8.         用于和对端连接的本地 IP、端口（如果不设置，
9.         则操作系统自己决定）
10.    */
11.    localAddress,
12.    localPort)  {
13.    var err;
14.    /*
15.        如果传了本地的地址或端口，则 TCP 连接中的源 IP
16.        和端口就是传的，否则由操作系统自己选
17.    */
18.    if (localAddress || localPort)  {
19.        // IP v4
20.        if (addressType === 4)  {
21.            localAddress = localAddress || '0.0.0.0';
22.            // 绑定地址和端口到 handle
23.            err = self._handle.bind(localAddress, localPort);
24.        } else if (addressType === 6)  {
25.            localAddress = localAddress || '::';
26.            err = self._handle.bind6(localAddress, localPort);
27.        }
28.
29.        // 绑定是否成功
30.        err = checkBindError(err, localPort, self._handle);
31.        if (err)  {
32.            const ex = exceptionWithHostPort(err,
33.                                              'bind',
34.                                              localAddress,
35.                                              localPort);
36.            self.destroy(ex);
37.            return;
38.        }
39.    }
40.}
```

```

39.    }
40.    // 对端的地址信息
41.    if (addressType === 6 || addressType === 4) {
42.        // 新建一个请求对象, C++层定义
43.        const req = new TCPConnectWrap();
44.        // 设置一些列属性
45.        req.oncomplete = afterConnect;
46.        req.address = address;
47.        req.port = port;
48.        req.localAddress = localAddress;
49.        req.localPort = localPort;
50.        // 调用底层对应的函数
51.        if (addressType === 4)
52.            err = self._handle.connect(req, address, port);
53.        else
54.            err = self._handle.connect6(req, address, port);
55.    }
56.    /*
57.     * 非阻塞调用, 可能在还没发起三次握手之前就报错了,
58.     * 而不是三次握手出错, 这里进行出错处理
59.    */
60.    if (err) {
61.        // 获取 socket 对应的底层 IP 端口信息
62.        var sockname = self._getsockname();
63.        var details;
64.
65.        if (sockname) {
66.            details = sockname.address + ':' + sockname.port;
67.        }
68.        // 构造错误信息, 销魂 socket 并触发 error 事件
69.        const ex = exceptionWithHostPort(err,
70.                                         'connect',
71.                                         address,
72.                                         port,
73.                                         details);
74.        self.destroy(ex);
75.    }
76.}

```

这里的代码比较多，除了错误处理外，主要的逻辑是 bind 和 connect。bind 函数的逻辑很简单（即使是底层的 bind），它就是在底层的一个结构体上设置了两个字段的值。所以我们主要来分析 connect。我们把关于 connect 的这段逻辑拎出来。

```
1.     const req = new TCPConnectWrap();
2.     // 设置一些列属性
3.     req.oncomplete = afterConnect;
4.     req.address = address;
5.     req.port = port;
6.     req.localAddress = localAddress;
7.     req.localPort = localPort;
8.     // 调用底层对应的函数
9.     self._handle.connect(req, address, port);
```

TCPConnectWrap 是 C++ 层提供的类，connect 对应 C++ 层的 Conenct，前面的章节我们已经分析过，不再具体分析。连接完成后，回调函数是 uv_stream_io。在 uv_stream_io 里会调用 connect_req 中的回调。假设连接建立，这时候就会执行 C++ 层的 AfterConnect。AfterConnect 会执行 JS 层的 afterConnect。

```
1. // 连接后执行的回调，成功或失败
2. function afterConnect(status, handle, req, readable, writable) {
    // handle 关联的 socket
3. var self = handle.owner;
4. // 连接过程中执行了 socket 被销毁了，则不需要继续处理
5. if (self.destroyed) {
6.     return;
7. }
8.
9. handle = self._handle;
10. self.connecting = false;
11. self._sockname = null;
12. // 连接成功
13. if (status === 0) {
14.     // 设置读写属性
15.     self.readable = readable;
16.     self.writable = writable;
17.     // socket 当前活跃，重置定时器
18.     self._unrefTimer();
19.     // 触发连接成功事件
20.     self.emit('connect');
21.     // socket 可读并且没有设置暂停模式，则开启读
22.     if (readable && !self.isPaused())
23.         self.read(0);
24. } else {
25.     // 连接失败，报错并销毁 socket
26.     self.connecting = false;
27.     var details;
28.     // 提示出错信息
```

```

29.     if (req.localAddress && req.localPort) {
30.         details = req.localAddress + ':' + req.localPort;
31.     }
32.     var ex = exceptionWithHostPort(status,
33.                                     'connect',
34.                                     req.address,
35.                                     req.port,
36.                                     details);
37.     if (details) {
38.         ex.localAddress = req.localAddress;
39.         ex.localPort = req.localPort;
40.     }
41.     // 销毁 socket
42.     self.destroy(ex);
43. }
44. }
```

一般情况下，连接成功后，JS 层调用 self.read(0) 注册等待可读事件。

17.1.2 读操作

我们看一下 socket 的读操作逻辑，在连接成功后，socket 会通过 read 函数在底层注册等待可读事件，等待底层事件驱动模块通知有数据可读。

```

1. Socket.prototype.read = function(n) {
2.     if (n === 0)
3.         return stream.Readable.prototype.read.call(this, n);
4.
5.     this.read = stream.Readable.prototype.read;
6.     this._consuming = true;
7.     return this.read(n);
8. };
```

这里会执行 Readable 模块的 read 函数，从而执行 _read 函数，_read 函数是由子类实现。所以我们看 Socket 的 _read

```

1. Socket.prototype._read = function(n) {
2.     // 还没建立连接，则建立后再执行
3.     if (this.connecting || !this._handle) {
4.         this.once('connect', () => this._read(n));
5.     } else if (!this._handle.reading) {
6.         this._handle.reading = true;
7.         // 执行底层的 readStart 注册等待可读事件
8.         var err = this._handle.readStart();
```

```
9.         if (err)
10.             this.destroy(errnoException(err, 'read'));
11.     }
12. }
```

但是我们发现 `tcp_wrap.cc` 没有 `readStart` 函数。一路往父类找，最终在 `stream_wrap.cc` 找到了该函数。

```
1. // 注册读事件
2. int LibuvStreamWrap::ReadStart() {
3.     return uv_read_start(stream(),
4.         [] (uv_handle_t* handle,
5.             size_t suggested_size,
6.             uv_buf_t* buf) {
7.         // 分配存储数据的内存
8.         static_cast<LibuvStreamWrap*>(handle->data)->OnUvAlloc(suggested_size,
9.             buf);
10.        },
11.        [] (uv_stream_t* stream, ssize_t nread, const uv_buf_t* buf) {
12.            // 读取数据成功的回调
13.            static_cast<LibuvStreamWrap*>(stream->data)->OnUvRead(nread, buf);
14.        });
}
```

`uv_read_start` 函数在流章节已经分析过，作用就是注册等待可读事件，这里就不再深入。`OnUvAlloc` 是分配存储数据的函数，我们可以不关注，我们看一下 `OnUvRead`，当可读事件触发时会执行 `OnUvRead`

```
1. void LibuvStreamWrap::OnUvRead(ssize_t nread, const uv_buf_t* buf)
{
    HandleScope scope(env()->isolate());
    Context::Scope context_scope(env()->context());
    // 触发 onread 事件
    EmitRead(nread, *buf);
}
```

`OnUvRead` 函数触发 `onread` 回调。

```
1. function onread(nread, buffer) {
2.     var handle = this;
3.     // handle 关联的 socket
```

```

4.     var self = handle.owner;
5.     // socket 有数据到来, 处于活跃状态, 重置定时器
6.     self._unrefTimer();
7.     // 成功读取数据
8.     if (nread > 0) {
9.         // push 到流中
10.        var ret = self.push(buffer);
11.        /*
12.            push 返回 false, 说明缓存的数据已经达到阈值,
13.            不能再触发读, 需要注销等待可读事件
14.        */
15.        if (handle.reading && !ret) {
16.            handle.reading = false;
17.            var err = handle.readStop();
18.            if (err)
19.                self.destroy(errnoException(err, 'read'));
20.        }
21.        return;
22.    }
23.
24.    // 没有数据, 忽略
25.    if (nread === 0) {
26.        debug('not any data, keep waiting');
27.        return;
28.    }
29.    // 不等于结束, 则读出错, 销毁流
30.    if (nread !== UV_EOF) {
31.        return self.destroy(errnoException(nread, 'read'));
32.    }
33.    // 流结束了, 没有数据读了
34.    self.push(null);
35.    /*
36.        也没有缓存的数据了, 可能需要销毁流, 比如是只读流,
37.        或者可读写流, 写端也没有数据了, 参考 maybeDestroy
38.    */
39.    if (self.readableLength === 0) {
40.        self.readable = false;
41.        maybeDestroy(self);
42.    }
43.    // 触发事件
44.    self.emit('_socketEnd');
45. }

```

socket 可读事件触发时大概有下面几种情况

1 有有效数据可读，push 到流中，触发 ondata 事件通知用户。

2 没有有效数据可读，忽略。

3 读出错，销毁流

4 读结束。

我们分析一下 4。在新建一个 socket 的时候注册了流结束的处理函数 onSocketEnd。

```
1. // 读结束后执行的函数
2. function onSocketEnd() {
3.   // 读结束标记
4.   this._readableState.ended = true;
5.   /*
6.     已经触发过 end 事件，则判断是否需要销毁，可能还有写端
7.   */
8.   if (this._readableState.endEmitted) {
9.     this.readable = false;
10.    maybeDestroy(this);
11.  } else {
12.    // 还没有触发 end 则等待触发 end 事件再执行下一步操作
13.    this.once('end', function end() {
14.      this.readable = false;
15.      maybeDestroy(this);
16.    });
17.    /*
18.      执行 read，如果流中没有缓存的数据则会触发 end 事件，
19.      否则等待消费完后再触发
20.    */
21.    this.read(0);
22.  }
23.  /*
24.    1 读结束后，如果不允许半开关，则关闭写端，如果还有数据还没有发送
25.    完毕，则先发送完再关闭
26.    2 重置写函数，后续执行写的时候报错
27.  */
28.  if (!this.allowHalfOpen) {
29.    this.write = writeAfterFIN;
30.    this.destroySoon();
31.  }
32. }
```

当 socket 的读端结束时，socket 的状态变更分为几种情况

1 如果可读流中还有缓存的数据，则等待读取。

2 如果写端也结束了，则销毁流。

3 如果写端没有结束，则判断 `allowHalfOpen` 是否允许半开关，不允许并且写端数据已经发送完毕则关闭写端。

17.1.3 写操作

接着我们看一下在一个流上写的时候，逻辑是怎样的。Socket 实现了单个写和批量写接口。

```

1. // 批量写
2. Socket.prototype._writev = function(chunks, cb) {
3.   this._writeGeneric(true, chunks, '', cb);
4. };
5.
6. // 单个写
7. Socket.prototype._write = function(data, encoding, cb) {
8.   this._writeGeneric(false, data, encoding, cb);
9. };

_writeGeneric

1. Socket.prototype._writeGeneric = function(writev, data, encoding,
2.   cb) {
3.   /*
4.     正在连接，则先保存待写的数据，因为 stream 模块是串行写的，
5.     所以第一次写没完成，不会执行第二次写操作 (_write) ,
6.     所以这里用一个字段而不是一个数组或队列保存数据和编码，
7.     因为有 pendingData 时 _writeGeneric 不会被执行第二次，这里缓存
8.     pendingData 不是为了后续写入，而是为了统计写入的数据总数
9.   */
10.  if (this.connecting) {
11.    this._pendingData = data;
12.    this._pendingEncoding = encoding;
13.    this.once('connect', function connect() {
14.      this._writeGeneric(writev, data, encoding, cb);
15.    });
16.  }
17.  // 开始写，则清空之前缓存的数据
18.  this._pendingData = null;
19.  this._pendingEncoding = '';
20.  // 写操作，有数据通信，刷新定时器
21.  this._unrefTimer();
22.  // 已经关闭，则销毁 socket
23.  if (!this._handle) {
24.    this.destroy(new errors.Error('ERR_SOCKET_CLOSED')), cb);

```

```
25.     return false;
26. }
27. // 新建一个写请求
28. var req = new WriteWrap();
29. req.handle = this._handle;
30. req.oncomplete = afterWrite;
31. // 是否同步执行写完成回调，取决于底层是同步写入，然后执行回调还是异步写入
32. req.async = false;
33. var err;
34. // 是否批量写
35. if (writev) {
36.     // 所有数据都是 buffer 类型，则直接堆起来，否则需要保存编码类型
37.     var allBuffers = data.allBuffers;
38.     var chunks;
39.     var i;
40.     if (allBuffers) {
41.         chunks = data;
42.         for (i = 0; i < data.length; i++)
43.             data[i] = data[i].chunk;
44.     } else {
45.         // 申请 double 个大小的数组
46.         chunks = new Array(data.length << 1);
47.         for (i = 0; i < data.length; i++) {
48.             var entry = data[i];
49.             chunks[i * 2] = entry.chunk;
50.             chunks[i * 2 + 1] = entry.encoding;
51.         }
52.     }
53.     err = this._handle.writev(req, chunks, allBuffers);
54.
55.     // Retain chunks
56.     if (err === 0) req._chunks = chunks;
57. } else {
58.     var enc;
59.     if (data instanceof Buffer) {
60.         enc = 'buffer';
61.     } else {
62.         enc = encoding;
63.     }
64.     err = createWriteReq(req, this._handle, data, enc);
65. }
66.
67. if (err)
68.     return this.destroy(errnoException(err, 'write', req.error),
cb);
```

```

69. // 请求写入底层的数据字节长度
70. this._bytesDispatched += req.bytes;
71. // 在 stream_base.cc 中
    req_wrap_obj->Set(env->async(), True(env->isolate())); 设置
72. if (!req.async) {
73.     cb();
74.     return;
75. }
76.
77. req.cb = cb;
78. // 最后一次请求写数据的字节长度
79. this[kLastWriteQueueSize] = req.bytes;
80. };

```

上面的代码很多，但是逻辑并不复杂，具体实现在 `stream_base.cc` 和 `stream_wrap.cc`，这里不再展开分析，主要是执行 `writev` 和 `createWriteReq` 函数进行写操作。它们底层调用的都是 `uv_write2`（需要传递文件描述符）或 `uv_write`（不需要传递文件描述符）或者 `uv_try_write` 函数进行写操作。这里只分析一下 `async` 的意义，`async` 默认是 `false`，它表示的意义是执行底层写入时，底层是否同步执行回调，`async` 为 `false` 说明写入完成回调是同步执行的。在 `stream_base.cc` 的写函数中有相关的逻辑。

```

1. err = DoWrite(req_wrap, buf_list, count, nullptr);
2. req_wrap_obj->Set(env->async(), True(env->isolate()));

```

当执行 `DoWrite` 的时候，`req_wrap` 中保存的回调可能会被 `Libuv` 同步执行，从而执行 `JS` 代码，这时候 `async` 是 `false`（默认值），说明回调是被同步执行的，如果 `DoWrite` 没有同步执行回调，则说明是异步执行回调。设置 `async` 为 `true`，再执行 `JS` 代码。

17.1.4 关闭写操作

当我们发送完数据后，我们可以通过调用 `socket` 对象的 `end` 函数关闭流的写端。我们看一下 `end` 的逻辑。

```

1. Socket.prototype.end = function(data, encoding, callback) {
2.   stream.Duplex.prototype.end.call(this,
3.                                     data,
4.                                     encoding,
5.                                     callback);
6.   return this;
7. };

```

`Socket` 的 `end` 是调用的 `Duplex` 的 `end`，而 `Duplex` 的 `end` 是继承于 `Writable` 的 `end`。`Writable` 的 `end` 最终会触发 `finish` 事件，`socket` 在初始化的时候监听了该事件。

```
1. this.on('finish', onSocketFinish);
```

我们看看 onSocketFinish。

```
1. // 执行了 end，并且数据发送完毕，则关闭写端
2. function onSocketFinish() {
3.   // 还没连接成功就执行了 end
4.   if (this.connecting) {
5.     return this.once('connect', onSocketFinish);
6.   }
7.   // 写结束了，如果也不能读或者读结束了，则销毁 socket
8.   if (!this.readable || this._readableState.ended) {
9.     return this.destroy();
10.  }
11. // 不支持 shutdown 则直接销毁
12. if (!this._handle || !this._handle.shutdown)
13.   return this.destroy();
14. // 支持 shutdown 则执行关闭，并设置回调
15. var err = defaultTriggerAsyncIdScope(
16.   this[async_id_symbol], shutdownSocket, this, afterShutdown
17. );
18. // 执行 shutdown 失败则直接销毁
19. if (err)
20.   return this.destroy(errnoException(err, 'shutdown'));
21. }
22.
23. // 发送关闭写端的请求
24. function shutdownSocket(self, callback) {
25.   var req = new ShutdownWrap();
26.   req.oncomplete = callback;
27.   req.handle = self._handle;
28.   return self._handle.shutdown(req);
29. }
```

Shutdown 函数在 `stream_base.cc` 中定义，最终调用 `uv_shutdown` 关闭流的写端，在 Libuv 流章节我们已经分析过。接着我们看一下关闭写端后，回调函数的逻辑。

```
1. // 关闭写端成功后的回调
2. function afterShutdown(status, handle, req) {
3.   // handle 关联的 socket
4.   var self = handle.owner;
5.   // 已经销毁了，则不需要往下走了，否则执行销毁操作
6.   if (self.destroyed)
7.     return;
```

```

8. // 写关闭成功，并且读也结束了，则销毁 socket，否则等待读结束再执行销毁
9. if (self._readableState.ended) {
10.   self.destroy();
11. } else {
12.   self.once('_socketEnd', self.destroy);
13. }
14. }

```

17.1.5 销毁

当一个 socket 不可读也不可写的时候、被关闭、发生错误的时候，就会被销毁。销毁一个流就是销毁流的读端、写端。然后执行流子类的`_destroy`函数。我们看一下 socket 的`_destroy`函数

```

1. // 销毁时执行的钩子函数，exception 代表是否因为错误导致的销毁
2. Socket.prototype._destroy = function(exception, cb) {
3.   this.connecting = false;
4.   this.readable = this.writable = false;
5.   // 清除定时器
6.   for (var s = this; s !== null; s = s._parent) {
7.     clearTimeout(s[kTimeout]);
8.   }
9.
10.  if (this._handle) {
11.    // 是否因为出错导致销毁流
12.    var isException = exception ? true : false;
13.    // 关闭底层 handle
14.    this._handle.close(() => {
15.      // close 事件的入参，表示是否因为错误导致的关闭
16.      this.emit('close', isException);
17.    });
18.    this._handle.onread = noop;
19.    this._handle = null;
20.    this._sockname = null;
21.  }
22.  // 执行回调
23.  cb(exception);
24.  // socket 所属的 server，作为客户端时是 null
25.  if (this._server) {
26.    // server 下的连接数减一
27.    this._server._connections--;
28.    /*
29.      是否需要触发 server 的 close 事件，
30.      当所有的连接（socket）都关闭时才触发 server 的是 close 事件

```

```
31.     */
32.     if (this._server._emitCloseIfDrained) {
33.         this._server._emitCloseIfDrained();
34.     }
35. }
36. };
```

_stream_writable.js 中的 destroy 函数只是修改读写流的状态和标记，子类需要定义 _destroy 函数销毁相关的资源，socket 通过调用 close 关闭底层关联的资源，关闭后触发 socket 的 close 事件（回调函数的第一个参数是 boolean 类型，说明是否因为错误导致 socket 关闭）。最后判断该 socket 是否来自服务器创建的，是的话该服务器的连接数减一，如果服务器执行了 close 并且当前连接数为 0，则关闭服务器。

17.2 TCP 服务器

net 模块提供了 createServer 函数创建一个 TCP 服务器。

```
1. function createServer(options, connectionListener) {
2.     return new Server(options, connectionListener);
3. }
4.
5. function Server(options, connectionListener) {
6.     EventEmitter.call(this);
7.     // 注册连接到来时执行的回调
8.     if (typeof options === 'function') {
9.         connectionListener = options;
10.    options = {};
11.    this.on('connection', connectionListener);
12. } else if (options == null || typeof options === 'object') {
13.    options = options || {};
14.    if (typeof connectionListener === 'function') {
15.        this.on('connection', connectionListener);
16.    }
17. } else {
18.     throw new errors.TypeError('ERR_INVALID_ARG_TYPE',
19.                             'options',
20.                             'Object',
21.                             options);
22. }
23. // 服务器建立的连接数
24. this._connections = 0;
25. this._handle = null;
26. this._unref = false;
27. // 服务器下的所有连接是否允许半连接
28. this.allowHalfOpen = options.allowHalfOpen || false;
29. // 有连接时是否注册读事件
30. this.pauseOnConnect = !options.pauseOnConnect;
31. }
```

createServer 返回的就是一个一般的 JS 对象，接着调用 listen 函数监听端口。看一下 listen 函数的逻辑

```

1. Server.prototype.listen = function(...args) {
2.   /*
3.    处理入参，根据文档我们知道 listen 可以接收好几个参数，
4.    假设我们这里是只传了端口号 9297
5.   */
6.   var normalized = normalizeArgs(args);
7.   // normalized = [{port: 9297}, null];
8.   var options = normalized[0];
9.   var cb = normalized[1];
10.  // 第一次 listen 的时候会创建，如果非空说明已经 listen 过
11.  if (this._handle) {
12.    throw new errors.Error('ERR_SERVER_ALREADY_LISTEN');
13.  }
14.  // listen 成功后执行的回调
15.  var hasCallback = (cb !== null);
16.  if (hasCallback) {
17.    // listen 成功的回调
18.    this.once('listening', cb);
19.  }
20.
21.  options = options._handle || options.handle || options;
22.  // 第一种情况，传进来的是一个 TCP 服务器，而不是需要创建一个服务
器
23.  if (options instanceof TCP) {
24.    this._handle = options;
25.    this[async_id_symbol] = this._handle.getAsyncId();
26.    listenIncluster(this, null, -1, -1, backlogFromArgs);
27.    return this;
28.  }
29.  // 第二种，传进来一个对象，并且带了 fd
30.  if (typeof options.fd === 'number' && options.fd >= 0) {
31.    listenIncluster(this,
32.                  null,
33.                  null,
34.                  null,
35.                  backlogFromArgs,
36.                  options.fd);
37.    return this;
38.  }
39.  // 创建一个 tcp 服务器
40.  var backlog;

```

```
41.    if (typeof options.port === 'number' ||
42.        typeof options.port === 'string') {
43.        backlog = options.backlog || backlogFromArgs;
44.        // 第三种 启动一个 TCP 服务器, 传了 host 则先进行 DNS 解析
45.        if (options.host) {
46.            lookupAndListen(this,
47.                            options.port | 0,
48.                            options.host,
49.                            backlog,
50.                            options.exclusive);
51.        } else {
52.            listenIncluster(this,
53.                            null,
54.                            options.port | 0,
55.                            4,
56.                            backlog,
57.                            undefined,
58.                            options.exclusive);
59.        }
60.        return this;
61.    }
62.};
```

我们看到有三种情况，分别是传了一个服务器、传了一个 fd、传了端口（或者 host），但是我们发现，这几种情况最后都是调用了 listenIncluster（lookupAndListen 是先 DNS 解析后再执行 listenIncluster），只是入参不一样，所以我们直接看 listenIncluster。

```
1. function listenIncluster(server,
2.                             address,
3.                             port,
4.                             addressType,
5.                             backlog,
6.                             fd,
7.                             exclusive) {
8.     exclusive = !!exclusive;
9.     if (cluster === null) cluster = require('cluster');
10.    if (cluster.isMaster || exclusive) {
11.        server._listen2(address, port, addressType, backlog, fd);
12.        return;
13.    }
14.}
```

因为我们是在主进程，所以直接执行_listen2，子进程的在 cluster 模块分析。_listen 对应的函数是 setupListenHandle

```

1. function setupListenHandle(address, port, addressType, backlog, fd)
2. {
3.     // 有 handle 则不需要创建了，否则创建一个底层的 handle
4.     if (this._handle) {
5.     } else {
6.         var rval = null;
7.         // 没有传 fd，则说明是监听端口和 IP
8.         if (!address && typeof fd !== 'number') {
9.             rval = createServerHandle('::', port, 6, fd);
10.            /*
11.                 返回 number 说明 bind IPv6 版本的 handle 失败，
12.                 回退到 v4，否则说明支持 IPv6
13.            */
14.            if (typeof rval === 'number') {
15.                // 赋值为 null，才能走下面的 createServerHandle
16.                rval = null;
17.                address = '0.0.0.0';
18.                addressType = 4;
19.            } else {
20.                address = '::';
21.                addressType = 6;
22.            }
23.        }
24.        // 创建失败则继续创建
25.        if (rval === null)
26.            rval = createServerHandle(address,
27.                                      port,
28.                                      addressType,
29.                                      fd);
30.        // 还报错则说明创建服务器失败，报错
31.        if (typeof rval === 'number') {
32.            var error = exceptionWithHostPort(rval,
33.                                              'listen',
34.                                              address,
35.                                              port);
36.            process.nextTick	emitErrorNT, this, error);
37.        return;
38.    }

```

```
39.         this._handle = rval;
40.     }
41.
42.     // 有完成三次握手的连接时执行的回调
43.     this._handle.onconnection = onconnection;
44.     this._handle.owner = this;
45.     // 执行 C++ 层 listen
46.     var err = this._handle.listen(backlog || 511);
47.     // 出错则报错
48.     if (err) {
49.         var ex = exceptionWithHostPort(err,
50.                                         'listen',
51.                                         address,
52.                                         port);
53.         this._handle.close();
54.         this._handle = null;
55.         nextTick(this[async_id_symbol], emitErrorNT, this, ex);
56.         return;
57.     }
58.     // 触发 listen 回调
59.     nextTick(this[async_id_symbol], emitListeningNT, this);
60. }
```

主要是调用 createServerHandle 创建一个 handle，然后调用 listen 函数监听。我们先看 createServerHandle

```
1. function createServerHandle(address, port, addressType, fd) {
2.     var err = 0;
3.     var handle;
4.
5.     var isTCP = false;
6.     // 传了 fd 则根据 fd 创建一个 handle
7.     if (typeof fd === 'number' && fd >= 0) {
8.         try {
9.             handle = createHandle(fd, true);
10.        } catch (e) {
11.            return UV_EINVAL;
12.        }
13.        // 把 fd 存到 handle 中
14.        handle.open(fd);
15.        handle.readable = true;
16.        handle.writable = true;
```

```
17.         assert(!address && !port);
18.         // 管道
19.     } else if (port === -1 && addressType === -1) {
20.         // 创建一个 Unix 域服务器
21.         handle = new Pipe(PipeConstants.SERVER);
22.     } else {
23.         // 创建一个 TCP 服务器
24.         handle = new TCP(TCPConstants.SERVER);
25.         isTCP = true;
26.     }
27. /*
28.     有地址或者 IP 说明是通过 IP 端口创建的 TCP 服务器,
29.     需要调 bind 绑定地址
30. */
31. if (address || port || isTCP) {
32.     // 没有地址, 则优先绑定 IPv6 版本的本地地址
33.     if (!address) {
34.         // Try binding to IPv6 first
35.         err = handle.bind6('::', port);
36.         // 失败则绑定 v4 的
37.         if (err) {
38.             handle.close();
39.             // Fallback to IPv4
40.             return createServerHandle('0.0.0.0', port);
41.         }
42.     } else if (addressType === 6) { // IPv6 或 v4
43.         err = handle.bind6(address, port);
44.     } else {
45.         err = handle.bind(address, port);
46.     }
47. }
48.
49. if (err) {
50.     handle.close();
51.     return err;
52. }
53.
54. return handle;
55. }
```

createServerHandle 主要是调用 createHandle 创建一个 handle 然后执行 bind 函数。创建 handle 的方式有几种，直接调用 C++ 层的函数或者通过 fd 创建。调用 createHandle 可以通过 fd 创建一个 handle

```
1. // 通过 fd 创建一个 handle，作为客户端或者服务器
2. function createHandle(fd, is_server) {
3.     // 判断 fd 对应的类型
4.     const type = TTYWrap.guessHandleType(fd);
5.     // Unix 域
6.     if (type === 'PIPE') {
7.         return new Pipe(
8.             is_server ? PipeConstants.SERVER : PipeConstants.SOCKET
9.         );
10.    // tcp
11.    if (type === 'TCP') {
12.        return new TCP(
13.            is_server ? TCPConstants.SERVER : TCPConstants.SOCKET
14.        );
15.    }
16.
17.    throw new errors.TypeError('ERR_INVALID_FD_TYPE', type);
18.}
```

接着我们看一下 bind 函数的逻辑，

```
1. int uv_tcp_bind(uv_tcp_t* tcp,
2.                  const struct sockaddr* addr,
3.                  unsigned int addrlen,
4.                  unsigned int flags) {
5.     int err;
6.     int on;
7.     // 如果没有 socket 则创建一个，有判断是否设置了 UV_HANDLE_BOUND，是则执行 bind，否则不执行
8.     // bind
9.     err = maybe_new_socket(tcp, addr->sa_family, 0);
10.    if (err)
11.        return err;
12.    on = 1;
13.    // 设置在断开连接的 2 ms 内可以重用端口，所以 Node.js 服务器可以快速重启
14.    if (setsockopt(tcp->io_watcher.fd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)))
15.        return UV_ERR(errno);
16.    errno = 0;
17.    // 执行 bind
18.    if (bind(tcp->io_watcher.fd, addr, addrlen) && errno != EADDRINUSE) {
19.        if (errno == EAFNOSUPPORT)
20.            return UV_EINVAL;
21.        return UV_ERR(errno);
```

```

22. }
23. // bind 是否出错
24. tcp->delayed_error = UV_ERR(errno);
25. // 打上已经执行了 bind 的标记
26. tcp->flags |= UV_HANDLE_BOUND;
27. if (addr->sa_family == AF_INET6)
28.     tcp->flags |= UV_HANDLE_IPV6;
29.
30. return 0;
31. }
```

执行完 bind 后，会继续执行 listen，我们接着看 listen 函数做了什么。我们直接看 `tcp_wrap.cc` 的 Listen。

```

1. void TCPWrap::Listen(const FunctionCallbackInfo<Value>& args) {
2.     TCPWrap* wrap;
3.     ASSIGN_OR_RETURN_UNWRAP(&wrap,
4.                             args.Holder(),
5.                             args.GetReturnValue().Set(UV_EBADF));
6.     int backlog = args[0]->Int32Value();
7.     int err = uv_listen(reinterpret_cast<uv_stream_t*>(&wrap->handle_),
8.                         backlog,
9.                         OnConnection);
10.    args.GetReturnValue().Set(err);
11. }
```

C++层几乎是透传到 Libuv，Libuv 的内容我们不再具体展开，当有三次握手的连接完成时，会执行 OnConnection

```

1. template <typename WrapType, typename UVType>
2. void ConnectionWrap<WrapType, UVType>::OnConnection(uv_stream_t* handle, int status) {
3.     // TCPWrap
4.     WrapType* wrap_data = static_cast<WrapType*>(handle->data);
5.     Environment* env = wrap_data->env();
6.     HandleScope handle_scope(env->isolate());
7.     Context::Scope context_scope(env->context());
8.     Local<Value> argv[] = {
9.         Integer::New(env->isolate(), status),
10.        Undefined(env->isolate())
11.    };
12.
13.    if (status == 0) {
14.        // 新建一个表示和客户端通信的对象，必填 TCPWrap 对象
```

```
15.     Local<Object> client_obj = WrapType::Instantiate(env, wrap_data,
16.     a, WrapType::SOCKET);
17.     WrapType* wrap;
18.     // 解包出一个 TCPWrap 对象存到 wrap
19.     ASSIGN_OR_RETURN_UNWRAP(&wrap, client_obj);
20.     uv_stream_t* client_handle = reinterpret_cast<uv_stream_t*>(&
21.         wrap->handle_);
22.     // 把通信 fd 存储到 client_handle 中
23.     if (uv_accept(handle, client_handle))
24.         return;
25.     argv[1] = client_obj;
26. }
27. // 回调上层的 onconnection 函数
28. wrap_data->MakeCallback(env->onconnection_string(), arraysize(argv)
29. , argv);
```

当建立了新连接时，操作系统会新建一个 socket 表示，同样，在 Node.js 层，也会新建一个对应的对象表示和客户端的通信，接着我们看 JS 层回调。

```
1. // clientHandle 代表一个和客户端建立 TCP 连接的实体
2. function onconnection(err, clientHandle) {
3.     var handle = this;
4.     var self = handle.owner;
5.     // 错误则触发错误事件
6.     if (err) {
7.         self.emit('error', errnoException(err, 'accept'));
8.         return;
9.     }
10.    // 建立过多，关掉
11.    if (self.maxConnections && self._connections >= self.maxConnections) {
12.        clientHandle.close();
13.        return;
14.    }
15.    // 新建一个 socket 用于通信
16.    var socket = new Socket({
17.        handle: clientHandle,
18.        allowHalfOpen: self.allowHalfOpen,
19.        pauseOnCreate: self.pauseOnConnect
20.    });
21.    socket.readable = socket.writable = true;
```

```

22. // 服务器的连接数加一
23. self._connections++;
24. socket.server = self;
25. socket._server = self;
26. // 触发用户层连接事件
27. self.emit('connection', socket);
28.

```

在 JS 层也会封装一个 Socket 对象用于管理和客户端的通信，接着触发 connection 事件。剩下的事情就是应用层处理了。

17.3 keepalive

本节分析基于 TCP 层的长连接问题，相比应用层 HTTP 协议的长连接，TCP 层提供的功能更多。TCP 层定义了三个配置。

- 1 多久没有收到数据包，则开始发送探测包。
- 2 每隔多久，再次发送探测包。
- 3 发送多少个探测包后，就断开连接。

我们看 Linux 内核代码里提供的配置。

```

1. // 多久没有收到数据就发起探测包
2. #define TCP_KEEPALIVE_TIME (120*60*HZ) /* two hours */
3. // 探测次数
4. #define TCP_KEEPALIVE_PROBES 9 /* Max of 9 keepalive probes*/
5. // 每隔多久探测一次
6. #define TCP_KEEPALIVE_INVL (75*HZ)

```

这是 Linux 提供的默认值。下面再看看阈值

```

1. #define MAX_TCP_KEEPIDLE 32767
2. #define MAX_TCP_KEEPINTVL 32767
3. #define MAX_TCP_KEEPCNT 127

```

这三个配置和上面三个一一对应。是上面三个配置的阈值。我们看一下 Node.js 中 keep-alive 的使用。

```
socket.setKeepAlive([enable][, initialDelay])
```

enable: 是否开启 keep-alive，Linux 下默认是不开启的。

initialDelay: 多久没有收到数据包就开始发送探测包。

接着我们看看这个 API 在 Libuv 中的实现。

```
1. int uv_tcp_keepalive(int fd, int on, unsigned int delay) {
2.     if (setsockopt(fd, SOL_SOCKET, SO_KEEPALIVE, &on, sizeof(on)))
3.         return UV_ERR(errno);
4.     // Linux 定义了这个宏
5. #ifdef TCP_KEEPIDLE
6.     /*
7.         on 是 1 才会设置，所以如果我们先开启 keep-alive，并且设置 delay,
8.         然后关闭 keep-alive 的时候，是不会修改之前修改过的配置的。
9.         因为这个配置在 keep-alive 关闭的时候是没用的
10.    */
11.    if (on && setsockopt(fd, IPPROTO_TCP, TCP_KEEPIDLE, &delay, sizeof(delay)))
12.        return UV_ERR(errno);
13. #endif
14.
15.    return 0;
16. }
```

我们看到 Libuv 调用了同一个系统函数两次。我们分别看一下这个函数的意义。参考 Linux2.6.13.1 的代码。

```
1. // net\socket.c
2. asmlinkage long sys_setsockopt(int fd, int level, int optname, char __user *optval, int
   optlen)
3. {
4.     int err;
5.     struct socket *sock;
6.
7.     if ((sock = sockfd_lookup(fd, &err))!=NULL)
8.     {
9.         ...
10.        if (level == SOL_SOCKET)
11.            err=sock_setsockopt(sock,level,optname,optval,optlen);
12.        else
13.            err=sock->ops->setsockopt(sock, level, optname, optval, optlen);
14.        sockfd_put(sock);
15.    }
16.    return err;
17. }
```

当 level 是 SOL_SOCKET 代表修改的 socket 层面的配置。IPPROTO_TCP 是修改 TCP 层的配置（该版本代码里是 SOL_TCP）。我们先看 SOL_SOCKET 层面的。

```
1. // net\socket.c -> net\core\sock.c -> net\ipv4\tcp_timer.c
2. int sock_setsockopt(struct socket *sock, int level, int optname,
3.                      char __user *optval, int optlen) {
4.     ...
5.     case SO_KEEPALIVE:
6.
7.         if (sk->sk_protocol == IPPROTO_TCP)
8.             tcp_set_keepalive(sk, valbool);
9.         // 设置 SOCK_KEEPOPEN 标记位 1
10.        sock_valbool_flag(sk, SOCK_KEEPOPEN, valbool);
11.        break;
12.     ...
```

 13. }

sock_setsockopt 首先调用了 tcp_set_keepalive 函数，然后给对应 socket 的 SOCK_KEEPOPEN 字段打上标记（0 或者 1 表示开启还是关闭）。接下来我们看 tcp_set_keepalive

```

1. void tcp_set_keepalive(struct sock *sk, int val)
2. {
3.     if ((1 << sk->sk_state) & (TCPF_CLOSE | TCPF_LISTEN))
4.         return;
5.     /*
6.         如果 val 是 1 并且之前是 0 (没开启) 那么就开启计时，超时后发送探测包，
7.         如果之前是 1, val 又是 1，则忽略，所以重复设置是无害的
8.     */
9.     if (val && !sock_flag(sk, SOCK_KEEPOPEN))
10.        tcp_reset_keepalive_timer(sk, keepalive_time_when(tcp_sk(sk)));
11.    else if (!val)
12.        // val 是 0 表示关闭，则清除定时器，就不发送探测包了
13.        tcp_delete_keepalive_timer(sk);
14. }
```

我们看看超时后的逻辑。

```

1. // 多久没有收到数据包则发送第一个探测包
2. static inline int keepalive_time_when(const struct tcp_sock *tp)
3. {
4.     // 用户设置的 (TCP_KEEPIDLE) 和系统默认的
5.     return tp->keepalive_time ?: sysctl_tcp_keepalive_time;
6. }
7. // 隔多久发送一个探测包
8. static inline int keepalive_intvl_when(const struct tcp_sock *tp)
9. {
10.    return tp->keepalive_intvl ?: sysctl_tcp_keepalive_intvl;
11. }
12.
13. static void tcp_keepalive_timer (unsigned long data)
14. {
15. ...
16. // 多久没有收到数据包了
17. elapsed = tcp_time_stamp - tp->rcv_tstamp;
18.     // 是否超过了阈值
19.     if (elapsed >= keepalive_time_when(tp)) {
20.         // 发送的探测包个数达到阈值，发送重置包
21.         if ((!tp->keepalive_probes && tp->probes_out >= sysctl_tcp_keepalive_probes) ||
22.             (tp->keepalive_probes && tp->probes_out >= tp->keepalive_probes)) {
23.             tcp_send_active_reset(sk, GFP_ATOMIC);
24.             tcp_write_err(sk);
25.             goto out;
26.         }
27.         // 发送探测包，并计算下一个探测包的发送时间 (超时时间)
28.         tcp_write_wakeup(sk)
29.             tp->probes_out++;
30.             elapsed = keepalive_intvl_when(tp);
31.     } else {
32.         /*
33.             还没到期则重新计算到期时间，收到数据包的时候应该会重置定时器，
```

```
34.         所以执行该函数说明的确是超时了，按理说不会进入这里。
35.     */
36.     elapsed = keepalive_time_when(tp) - elapsed;
37. }
38.
39. TCP_CHECK_TIMER(sk);
40. sk_stream_mem_reclaim(sk);
41.
42. resched:
43. // 重新设置定时器
44. tcp_reset_keepalive_timer (sk, elapsed);
45. ...
```

所以在 SOL_SOCKET 层面是设置是否开启 keep-alive 机制。如果开启了，就会设置定时器，超时的时候就会发送探测包。但是我们发现，SOL_SOCKET 只是设置了是否开启探测机制，并没有定义上面三个配置的值，所以系统会使用默认值进行心跳机制（如果我们设置了开启 keep-alive 的话）。这就是为什么 Libuv 调了两次 setsockopt 函数。第二次的调用设置了上面三个配置中的第一个（后面两个也可以设置，不过 Libuv 没有提供接口，可以自己调用 setsockopt 设置）。那么我们来看一下 Libuv 的第二次调用 setsockopt 是做了什么。我们直接看 TCP 层的实现。

```
1. // net\ipv4\tcp.c
2. int tcp_setsockopt(struct sock *sk, int level, int optname, char __user *optval, int opt
   len)
3. {
4. ...
5. case TCP_KEEPIDLE:
6.     // 修改多久没有收到数据包则发送探测包的配置
7.     tp->keepalive_time = val * HZ;
8.     // 是否开启了 keep-alive 机制
9.     if (sock_flag(sk, SOCK_KEEPOPEN) &&
10.         !((1 << sk->sk_state) &
11.             (TCPF_CLOSE | TCPF_LISTEN))) {
12.         // 当前时间减去上次收到数据包的时候，即多久没有收到数据包了
13.         __u32 elapsed = tcp_time_stamp - tp->rcv_tstamp;
14.         // 算出还要多久可以发送探测包，还是可以直接发（已经触发了）
15.         if (tp->keepalive_time > elapsed)
16.             elapsed = tp->keepalive_time - elapsed;
17.         else
18.             elapsed = 0;
19.         // 设置定时器
20.         tcp_reset_keepalive_timer(sk, elapsed);
21.     }
22. ...
23. }
```

该函数首先修改配置，然后判断是否开启了 keep-alive 的机制，如果开启了，则重新设置定时器，超时的时候就会发送探测包。但是有一个问题是，心跳机制并不是什么时候都好使，如果两端都没有数据来往时，心跳机制能很好地工作，但是一旦本端有数据发送的时候，它就会抑制心跳机制。我们看一下 Linux 内核 5.7.7 的一段相关代码，如图 17-3 所示。

```

skb = tcp_send_head(sk);
if (skb && before(TCP_SKB_CB(skb)->seq, tcp_wnd_end(tp))) {
    int err;
    unsigned int mss = tcp_current_mss(sk);
    unsigned int seg_size = tcp_wnd_end(tp) - TCP_SKB_CB(skb)->seq;

    if (before(tp->pushed_seq, TCP_SKB_CB(skb)->end_seq))
        tp->pushed_seq = TCP_SKB_CB(skb)->end_seq;

    /* We are probing the opening of a window...
     * If (seg_size < TCP_SKB_CB(skb)->end_seq - TCP_SKB_CB(skb)->seq || 
     *     skb->len > mss) { ...
     *     tcp_set_skb_tso_segs(skb, mss);

TCP_SKB_CB(skb)->tcp_flags |= TCPHDR_PSH;
err = tcp_transmit_skb(sk, skb, 1, GFP_ATOMIC);
if (!err)
    tcp_event_new_data_sent(sk, skb);
return err;
} else {
    if (between(tp->snd_up, tp->snd_una + 1, tp->snd_una + 0xFFFF))
        tcp_xmit_probe_skb(sk, 1, mib);
return tcp_xmit_probe_skb(sk, 0, mib);
}

```

图 17-3

上面这一段是心跳机制中，定时器超时时，执行的一段逻辑，我们只需要关注红色框里的代码。一般来说，心跳定时器超时，操作系统会发送一个新的心跳包，但是如果发送队列里还有数据没有发送，那么操作系统会优先发送。或者发送出去的没有 ack，也会优先触发重传。这时候心跳机制就失效了。对于这个问题，Linux 提供了另一个属性 TCP_USER_TIMEOUT。这个属性的功能是，发送了数据，多久没有收到 ack 后，操作系统就认为这个连接断开了。看一下相关代码，如图 17-4 所示。

```
case TCP_USER_TIMEOUT:  
    /* Cap the max time in ms TCP will retry or probe the window  
     * before giving up and aborting (ETIMEDOUT) a connection.  
     */  
    if (val < 0)  
        err = -EINVAL;  
    else  
        icsk->icsk_user_timeout = val;  
    break;
```

图 17-4

下面是设置阈值的代码，如图 17-5 所示。

```
/* If the TCP_USER_TIMEOUT option is enabled, use that  
 * to determine when to timeout instead.  
 */  
if ((icsk->icsk_user_timeout != 0 &&  
    elapsed >= msecs_to_jiffies(icsk->icsk_user_timeout) &&  
    icsk->icsk_probes_out > 0) ||  
    (icsk->icsk_user_timeout == 0 &&  
    icsk->icsk_probes_out >= keepalive_probes(tp))) {  
    tcp_send_active_reset(sk, GFP_ATOMIC);  
    tcp_write_err(sk);  
    goto out;  
}
```

图 17-5

这是超时时判断是否断开连接的代码。我们看到有两个情况下操作系统会认为连接断开了。

1 设置了 TCP_USER_TIMEOUT 时，如果发送包数量大于 1 并且当前时间距离上次收到包的时间间隔已经达到阈值。

2 没有设置 TCP_USER_TIMEOUT，但是心跳包发送数量达到阈值。

所以我们可以同时设置这两个属性。保证心跳机制可以正常运行，Node.js 的 keepalive 有两个层面的内容，第一个是是否开启，第二个是开启后，使用的配置。Node.js 的 setKeepAlive 就是做了这两件事情。只不过它只支持修改一个配置。Node.js 只支持 TCP_KEEPALIVE_TIME。另外我们可以通过一下代码判断配置的值。

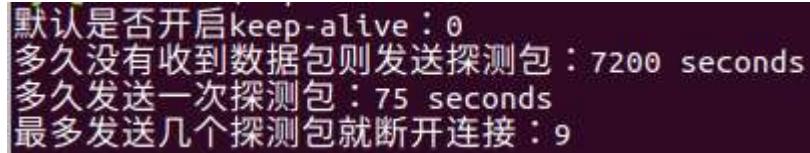
```
1. include <stdio.h>  
2. #include <netinet/tcp.h>  
3.  
4. int main(int argc, const char *argv[]){  
5.     int sockfd;  
6.     int optval;
```

```

8.     socklen_t optlen = sizeof(optval);
9.
10.    sockfd = socket(AF_INET, SOCK_STREAM, 0);
11.    getsockopt(sockfd, SOL_SOCKET, SO_KEEPALIVE, &optval, &optlen);
12.    printf("默认是否开启 keep-alive: %d \n", optval);
13.
14.    getsockopt(sockfd, SOL_TCP, TCP_KEEPIDLE, &optval, &optlen);
15.    printf("多久没有收到数据包则发送探测包: %d seconds \n", optval);
16.
17.    getsockopt(sockfd, SOL_TCP, TCP_KEEPINTVL, &optval, &optlen);
18.    printf("多久发送一次探测包: %d seconds \n", optval);
19.
20.    getsockopt(sockfd, SOL_TCP, TCP_KEEPCNT, &optval, &optlen);
21.    printf("最多发送几个探测包就断开连接: %d \n", optval);
22.
23.    return 0;
24. }

```

输出如图 17-6 所示。



```

默认是否开启keep-alive : 0
多久没有收到数据包则发送探测包 : 7200 seconds
多久发送一次探测包 : 75 seconds
最多发送几个探测包就断开连接 : 9

```

图 17-6

再看一下 wireshark 下的 keepalive 包，如图 17-7 所示。

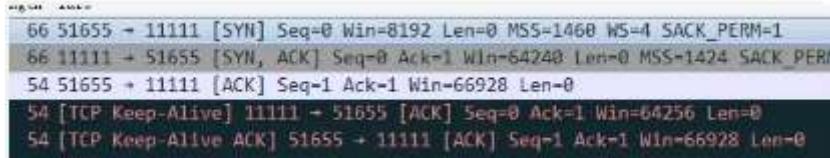


图 17-7

17.4 allowHalfOpen

我们知道 TCP 连接在正常断开的时候，会走四次挥手的流程，在 Node.js 中，当收到对端发送过来的 fin 包时，回复 ack 后，默认会发送 fin 包给对端，以完成四次挥手。但是我们可能会有这样的场景，客户端发送完数据后，发送 fin 包表示自己没有数据可写了，只需要等待服务器返回。这时候如果服务器在收到 fin 包后，也回复 fin，那就会有问题。在 Node.js 中提供了 allowHalfOpen 选项支持半关闭，我们知道 TCP 是全双工的，两端可以同时互相发送数据，allowHalfOpen 相当于把一端关闭了，允许数据单向传输。我们看一下 allowHalfOpen 的实现。allowHalfOpen 是属于 Socket 的选项。我们从 Node.js 收到一个 fin 包开始分析整个流程。首先在新建 Socket 对象的时候，注册对应事件。

```
socket.on('_socketEnd', onSocketEnd);
```

当操作系统收到 fin 包的时候，会触发 socket 的可读事件，执行 Node.js 的读回调。Node.js 执行读取的时候发现，读取已结束，因为对端发送了 fin 包。这时候会触发 _socketEnd 事件。我们看一下相关代码。

```

1. function onSocketEnd() {

```

```
2. // ...
3. if (!this.allowHalfOpen) {
4.     this.write = writeAfterFIN;
5.     this.destroySoon();
6. }
7. }
```

allowHalfOpen 默认是 `false`。`onSocketEnd` 首先设置 `write` 函数为 `writeAfterFIN`，我们看看这时候如果我们写会怎样。我们会收到一个错误。

```
1. function writeAfterFIN(chunk, encoding, cb) {
2.     var er = new Error('This socket has been ended by the other party');
3.     er.code = 'EPIPE';
4.     this.emit('error', er);
5.     if (typeof cb === 'function') {
6.         nextTick(this[async_id_symbol], cb, er);
7.     }
8. }
```

设置完 `write` 后，接着 Node.js 会发送 `fin` 包。

```
1. Socket.prototype.destroySoon = function() {
2.     // 关闭写流
3.     if (this.writable)
4.         this.end();
5.     // 关闭成功后销毁流
6.     if (this._writableState.finished)
7.         this.destroy();
8.     else
9.         this.once('finish', this.destroy);
10.};
```

首先关闭写流，然后执行 `destroy` 函数销毁流。在 `destroy` 中会执行 `_destroy`。`_destroy` 会执行具体的关闭操作，即发送 `fin` 包。

```
1. this._handle.close(() => {
2.     this.emit('close', isException);
3.});
```

我们看到 C++ 层的 `close`。

```
1. void HandleWrap::Close(const FunctionCallbackInfo<Value>& args) {
2.     Environment* env = Environment::GetCurrent(args);
```

```

3.
4.     HandleWrap* wrap;
5.     ASSIGN_OR_RETURN_UNWRAP(&wrap, args.Holder());
6.     // 关闭 handle
7.     uv_close(wrap->handle_, OnClose);
8.     wrap->state_ = kClosing;
9.     // 执行回调，触发 close 事件
10.    if (args[0]->IsFunction()) {
11.        wrap->object()->Set(env->onclose_string(), args[0]);
12.        wrap->state_ = kClosingWithCallback;
13.    }
14. }
```

我们继续往 Libuv 看。

```

1. void uv_close(uv_handle_t* handle, uv_close_cb cb) {
2.     uv_loop_t* loop = handle->loop;
3.
4.     handle->close_cb = cb;
5.     switch (handle->type) {
6.         case UV_TCP:
7.             uv_tcp_close(loop, (uv_tcp_t*)handle);
8.             return;
9.
10.        // ...
11.    }
12. }
```

`uv_tcp_close` 会对 `close` 的封装，我们看 `tcp_close` 的大致实现。

```

1. static void tcp_close(struct sock *sk, int timeout)
2. {
3.
4.     // 监听型的 socket 要关闭建立的连接
5.     if (sk->state == TCP_LISTEN)
6.     {
7.         /* Special case */
8.         tcp_set_state(sk, TCP_CLOSE);
9.         // 关闭已经建立的连接
10.        tcp_close_pending(sk);
11.        release_sock(sk);
12.        return;
13.    }
14.
15.    struct sk_buff *skb;
```

```
16.     // 销毁接收队列中未处理的数据
17.     while((skb=skb_dequeue(&sk->receive_queue))!=NULL)
18.         kfree_skb(skb, FREE_READ);
19.     // 发送 fin 包
20.     tcp_send_fin(sk);
21.     release_sock(sk);
22. }
```

以上是 Node.js 中 socket 收到 fin 包时的默认处理流程，当我们设置 `allowHalfOpen` 为 `true` 的时候，就可以修改这个默认的行为，允许半关闭状态的连接。

17.5 server close

调用 `close` 可以关闭一个服务器，首先我们看一下 Node.js 文档关于 `close` 函数的解释

`Stops the server from accepting new connections and keeps existing connections. This function is asynchronous, the server is finally closed when all connections are ended and the server emits a 'close' event. The optional callback will be called once the 'close' event occurs. Unlike that event, it will be called with an Error as its only argument if the server was not open when it was closed.`

在 Node.js 中，当我们使用 `close` 关闭一个 `server` 时，`server` 会等所有的连接关闭后才会触发 `close` 事件。我们看 `close` 的实现，一探究竟。

```
1. Server.prototype.close = function(cb) {
2.     // 触发回调
3.     if (typeof cb === 'function') {
4.         if (!this._handle) {
5.             this.once('close', function close() {
6.                 cb(new errors.Error('ERR_SERVER_NOT_RUNNING'));
7.             });
8.         } else {
9.             this.once('close', cb);
10.        }
11.    }
12.    // 关闭底层资源
13.    if (this._handle) {
14.        this._handle.close();
15.        this._handle = null;
16.    }
17.    // 判断是否需要立刻触发 close 事件
```

```

18.  this._emitCloseIfDrained();
19.  return this;
20. };

```

close 的代码比较简单，首先监听 close 事件，然后关闭 server 对应的 handle，所以 server 不会再接收新的请求了。最后调用 _emitCloseIfDrained，我们看一下这个函数是干嘛的。

```

1. Server.prototype._emitCloseIfDrained = function() {
2.   // 还有连接或者 handle 非空说明 handle 还没有关闭，则先不触发 close 事
 件
3.   if (this._handle || this._connections) {
4.     return;
5.   }
6.   // 触发 close 事件
7.   const asyncId = this._handle ? this[async_id_symbol] : null;
8.   nextTick(asyncId, emitCloseNT, this);
9. };
10.
11.
12. function emitCloseNT(self) {
13.   self.emit('close');
14. }

```

_emitCloseIfDrained 中有一个拦截的判断，handle 非空或者连接数非 0。由之前的代码我们已经知道 handle 是 null，但是如果这时候连接数非 0，也不会触发 close 事件。那什么时候才会触发 close 事件呢？在 socket 的 _destroy 函数中我们找到修改连接数的逻辑。

```

1. Socket.prototype._destroy = function(exception, cb) {
2.   ...
3.   // socket 所属的 server
4.   if (this._server) {
5.     // server 下的连接数减一
6.     this._server._connections--;
7.     // 是否需要触发 server 的 close 事件，当所有的连接（socket）都关闭时
      才触发 server 的是 close 事件
8.     if (this._server._emitCloseIfDrained) {
9.       this._server._emitCloseIfDrained();
10.    }
11.  }
12. };

```

我们看到每一个连接关闭的时候，都会导致连接数减一，直到为 0 的时候才会触发 close 事件。假设我们启动了一个服务器，接收到了一些客户端的请求，这时候，如果我们想修改一个代码发布，需要重启服务器，怎么办？假设我们有以下代码。

server.js

```
1. const net = require('net');
2. const server = net.createServer().listen(80);
```

client.js

```
1. const net = require('net');
2. net.connect({port:80})
```

如果我们直接杀死进程，那么存量的请求就会无法正常被处理。这会影响我们的服务质量。我们看一下 Node.js 如何在重启时优雅地退出，所谓优雅，即让 Node.js 进程处理完存量请求后再退出。Server 的 close 的实现给了我们一些思路。我们可以监听 server 的 close 事件，等到触发 close 事件后才退出进程。

```
1. const net = require('net');
2. const server = net.createServer().listen(80);
3. server.on('close', () => {
4.   process.exit();
5. });
6. // 防止进程提前挂掉
7. process.on('uncaughtException', () => {
8.
9. });
10. process.on('SIGINT', function() {
11.   server.close();
12. })
```

我们首先监听 SIGINT 信号，当我们使用 SIGINT 信号杀死进程时，首先调用 server.close，等到所有的连接断开，触发 close 时候时，再退出进程。我们首先开启服务器，然后开启两个客户端。接着按下 **ctrl+c**，我们发现这时候服务器不会退出，然后我们关闭两个客户端，这时候 server 就会优雅地退出。

第十八章 HTTP

HTTP 模块实现了 HTTP 服务器和客户端的功能，是 Node.js 的核心模块，也是我们使用得最多的模块。本章我们来分析 HTTP 模块，从中我们可以学习到一个 HTTP 服务器和客户端是怎么实现的，以及 HTTP 协议本身的一些原理和优化。

18.1 HTTP 解析器

HTTP 解析器是 HTTP 模块的核心，不管是作为服务器处理请求还是客户端处理响应都需要使用 HTTP 解析器解析 HTTP 协议。新版 Node.js 使用了新的 HTTP 解析器 `llhttp`。根据官方说明 `llhttp` 比旧版的 `http_parser` 在性能上有了非常大的提高。本节我们分析分析 `llhttp` 的基础原理和使用。HTTP 解析器是一个非常复杂的状态机，在解析数据的过程中，会不断触发钩子函数。下面是 `llhttp` 支持的钩子函数。如果用户定义了对应的钩子，在解析的过程中就会被回调。

```

1. // 开始解析 HTTP 协议
2. int llhttp__on_message_begin(llhttp_t* s, const char* p, const char* endp) {
3.     int err;
4.     CALLBACK_MAYBE(s, on_message_begin, s);
5.     return err;
6. }
7.
8. // 解析出请求 url 时的回调，最后拿到一个 url
9. int llhttp__on_url(llhttp_t* s, const char* p, const char* endp)
10. {
11.     int err;
12.     CALLBACK_MAYBE(s, on_url, s, p, endp - p);
13.     return err;
14.
15. // 解析出 HTTP 响应状态的回调
16. int llhttp__on_status(llhttp_t* s, const char* p, const char* endp) {
17.     int err;
18.     CALLBACK_MAYBE(s, on_status, s, p, endp - p);
19.     return err;
20. }
21.
22. // 解析出头部键时的回调
23. int llhttp__on_header_field(llhttp_t* s, const char* p, const char* endp) {
24.     int err;
25.     CALLBACK_MAYBE(s, on_header_field, s, p, endp - p);
26.     return err;
27. }
```

```
28.  
29. // 解析出头部值时的回调  
30. int llhttp__on_header_value(llhttp_t* s, const char* p, const char* endp) {  
31.     int err;  
32.     CALLBACK_MAYBE(s, on_header_value, s, p, endp - p);  
33.     return err;  
34. }  
35.  
36. // 解析 HTTP 头完成时的回调  
37. int llhttp__on_headers_complete(llhttp_t* s, const char* p, const char* endp) {  
38.     int err;  
39.     CALLBACK_MAYBE(s, on_headers_complete, s);  
40.     return err;  
41. }  
42.  
43. // 解析完 body 的回调  
44. int llhttp__on_message_complete(llhttp_t* s, const char* p, const char* endp) {  
45.     int err;  
46.     CALLBACK_MAYBE(s, on_message_complete, s);  
47.     return err;  
48. }  
49.  
50. // 解析 body 时的回调  
51. int llhttp__on_body(llhttp_t* s, const char* p, const char* endp) {  
52.     int err;  
53.     CALLBACK_MAYBE(s, on_body, s, p, endp - p);  
54.     return err;  
55. }  
56.  
57. // 解析到一个 chunk 结构头时的回调  
58. int llhttp__on_chunk_header(llhttp_t* s, const char* p, const char* endp) {  
59.     int err;  
60.     CALLBACK_MAYBE(s, on_chunk_header, s);  
61.     return err;  
62. }  
63.  
64. // 解析完一个 chunk 时的回调  
65. int llhttp__on_chunk_complete(llhttp_t* s, const char* p, const char* endp) {  
66.     int err;  
67.     CALLBACK_MAYBE(s, on_chunk_complete, s);
```

```

68.     return err;
69. }
```

Node.js 在 node_http_parser.cc 中对 llhttp 进行了封装。该模块导出了一个 HTTPParser。

```

1. Local<FunctionTemplate> t=env->NewFunctionTemplate(Parser::New);
2. t->InstanceTemplate()->SetInternalFieldCount(1);
3. t->SetClassName(FIXED_ONE_BYTE_STRING(env->isolate(),
4.                               "HTTPParser"));
5. target->Set(env->context(),
6. FIXED_ONE_BYTE_STRING(env->isolate(), "HTTPParser"),
7. t->GetFunction(env->context()).ToLocalChecked()).Check();
```

在 Node.js 中我们通过以下方式使用 HTTPParser。

```

1. const parser = new HTTPParser();
2.
3. cleanParser(parser);
4. parser.onIncoming = null;
5. parser[kOnHeaders] = parserOnHeaders;
6. parser[kOnHeadersComplete] = parserOnHeadersComplete;
7. parser[kOnBody] = parserOnBody;
8. parser[kOnMessageComplete] = parserOnMessageComplete;
9. // 初始化 HTTP 解析器处理的报文类型，这里是响应报文
10. parser.initialize(HTTPParser.RESPONSE,
11.      new HTTPClientAsyncResource('HTTPINCOMINGMESSAGE', req),
12.      req.maxHeaderSize || 0,
13.      req.insecureHTTPParser === undefined ?
14.          isLenient() : req.insecureHTTPParser);
15. // 收到数据后传给解析器处理
16. const ret = parser.execute(data);
17. }
```

我们看一下 initialize 和 execute 的代码。Initialize 函数用于初始化 llhttp。

```

1. static void Initialize(const FunctionCallbackInfo<Value>& args) {
2.     Environment* env = Environment::GetCurrent(args);
3.     bool lenient = args[3]->IsTrue();
4.
5.     uint64_t max_http_header_size = 0;
6.     // 头部的最大大小
7.     if (args.Length() > 2) {
8.         max_http_header_size = args[2].As<Number>()->Value();
9.     }
```

```
10. // 没有设置则取 Node.js 的默认值
11. if (max_http_header_size == 0) {
12.     max_http_header_size=env->options()->max_http_header_size;
13. }
14. // 解析的报文类型
15. llhttp_type_t type =
16.     static_cast<llhttp_type_t>(args[0].As<Int32>()->Value());
17.
18. CHECK(type == HTTP_REQUEST || type == HTTP_RESPONSE);
19. Parser* parser;
20. ASSIGN_OR_RETURN_UNWRAP(&parser, args.Holder());
21. parser->Init(type, max_http_header_size, lenient);
22. }
```

Initialize 做了一些预处理后调用 Init。

```
1. void Init(llhttp_type_t type, uint64_t max_http_header_size, bool
   lenient) {
2.     // 初始化 llhttp
3.     llhttp_init(&parser_, type, &settings);
4.     llhttp_set_lenient(&parser_, lenient);
5.     header_nread_ = 0;
6.     url_.Reset();
7.     status_message_.Reset();
8.     num_fields_ = 0;
9.     num_values_ = 0;
10.    have_flushed_ = false;
11.    got_exception_ = false;
12.    max_http_header_size_ = max_http_header_size;
13. }
```

Init 做了一些字段的初始化，最重要的是调用了 llhttp_init 对 llhttp 进行了初始化，另外 kOn 开头的属性是钩子函数，由 node_http_parser.cc 中的回调，而 node_http_parser.cc 也会定义钩子函数，由 llhttp 回调，我们看一下 node_http_parser.cc 钩子函数的定义和实现。

```
1. const llhttp_settings_t Parser::settings = {
2.     Proxy<Call, &Parser::on_message_begin>::Raw,
3.     Proxy<DataCall, &Parser::on_url>::Raw,
4.     Proxy<DataCall, &Parser::on_status>::Raw,
5.     Proxy<DataCall, &Parser::on_header_field>::Raw,
6.     Proxy<DataCall, &Parser::on_header_value>::Raw,
7.     Proxy<Call, &Parser::on_headers_complete>::Raw,
8.     Proxy<DataCall, &Parser::on_body>::Raw,
9.     Proxy<Call, &Parser::on_message_complete>::Raw,
```

```

10. Proxy<Call, &Parser::on_chunk_header>::Raw,
11. Proxy<Call, &Parser::on_chunk_complete>::Raw,
12. };

```

1 开始解析报文的回调

```

1. // 开始解析报文，一个 TCP 连接可能会有多个报文
2. int on_message_begin() {
3.     num_fields_ = num_values_ = 0;
4.     url_.Reset();
5.     status_message_.Reset();
6.     return 0;
7. }

```

2 解析 url 时的回调

```

1. int on_url(const char* at, size_t length) {
2.     int rv = TrackHeader(length);
3.     if (rv != 0) {
4.         return rv;
5.     }
6.
7.     url_.Update(at, length);
8.     return 0;
9. }

```

3 解析 HTTP 响应时的回调

```

1. int on_status(const char* at, size_t length) {
2.     int rv = TrackHeader(length);
3.     if (rv != 0) {
4.         return rv;
5.     }
6.
7.     status_message_.Update(at, length);
8.     return 0;
9. }

```

4 解析到 HTTP 头的键时回调

```

1. int on_header_field(const char* at, size_t length) {
2.     int rv = TrackHeader(length);
3.     if (rv != 0) {
4.         return rv;
5.     }
6.     // 相等说明键对值的解析是一一对应的

```

```
7.     if (num_fields_ == num_values_) {
8.         // start of new field name
9.         // 键的数加一
10.        num_fields_++;
11.        // 超过阈值则先回调 js 消费掉
12.        if (num_fields_ == kMaxHeaderFieldsCount) {
13.            // ran out of space - flush to javascript land
14.            Flush();
15.            // 重新开始
16.            num_fields_ = 1;
17.            num_values_ = 0;
18.        }
19.        // 初始化
20.        fields_[num_fields_ - 1].Reset();
21.    }
22.
23.    // 保存键
24.    fields_[num_fields_ - 1].Update(at, length);
25.
26.    return 0;
27. }
```

当解析的头部个数达到阈值时，Node.js 会先通过 Flush 函数回调 JS 层保存当前的一些数据。

```
1. void Flush() {
2.     HandleScope scope(env()->isolate());
3.
4.     Local<Object> obj = object();
5.     // JS 层的钩子
6.     Local<Value> cb = obj->Get(env()->context(), kOnHeaders).ToLocalChecked();
7.     if (!cb->IsFunction())
8.         return;
9.
10.    Local<Value> argv[2] = {
11.        CreateHeaders(),
12.        url_.ToString(env())
13.    };
14.
15.    MaybeLocal<Value> r = MakeCallback(cb.As<Function>(),
16.                                         arraysize(argv),
17.                                         argv);
18.    url_.Reset();
19.    have_flushed_ = true;
```

```

20.    }
21.
22. Local<Array> CreateHeaders() {
23.     // HTTP 头的个数乘以 2，因为一个头由键和值组成
24.     Local<Value> headers_v[kMaxHeaderFieldsCount * 2];
25.     // 保存键和值到 HTTP 头
26.     for (size_t i = 0; i < num_values_; ++i) {
27.         headers_v[i * 2] = fields_[i].ToString(env());
28.         headers_v[i * 2 + 1] = values_[i].ToString(env());
29.     }
30.
31.     return Array::New(env()->isolate(), headers_v, num_values_ *
32.     2);
32. }
33.

```

Flush 会调用 JS 层的 `kOnHeaders` 钩子函数。

5 解析到 HTTP 头的值时回调

```

1. int on_header_value(const char* at, size_t length) {
2.     int rv = TrackHeader(length);
3.     if (rv != 0) {
4.         return rv;
5.     }
6.     /*
7.      值的个数不等于键的个数说明正解析到键对应的值，即一一对应。
8.      否则说明一个键存在多个值，则不更新值的个数，多个值累加到一个 slot
9.     */
10.    if (num_values_ != num_fields_) {
11.        // start of new header value
12.        num_values_++;
13.        values_[num_values_ - 1].Reset();
14.    }
15.
16.    CHECK_LT(num_values_, arraysize(values_));
17.    CHECK_EQ(num_values_, num_fields_);
18.
19.    values_[num_values_ - 1].Update(at, length);
20.
21.    return 0;
22. }

```

6 解析完 HTTP 头后的回调

```
1. int on_headers_complete() {
2.     header_nread_ = 0;
3.     enum on_headers_complete_arg_index {
4.         A_VERSION_MAJOR = 0,
5.         A_VERSION_MINOR,
6.         A_HEADERS,
7.         A_METHOD,
8.         A_URL,
9.         A_STATUS_CODE,
10.        A_STATUS_MESSAGE,
11.        A_UPGRADE,
12.        A_SHOULD_KEEP_ALIVE,
13.        A_MAX
14.    };
15.
16.    Local<Value> argv[A_MAX];
17.    Local<Object> obj = object();
18.    Local<Value> cb = obj->Get(env()->context(),
19.                                kOnHeadersComplete).ToLocalChecke
d();
20.
21.    Local<Value> undefined = Undefined(env()->isolate());
22.    for (size_t i = 0; i < arraysize(argv); i++)
23.        argv[i] = undefined;
24.    // 之前 flush 过, 则继续 flush 到 JS 层, 否则返回全部头给 js
25.    if (have_flushed_) {
26.        // Slow case, flush remaining headers.
27.        Flush();
28.    } else {
29.        // Fast case, pass headers and URL to JS land.
30.        argv[A_HEADERS] = CreateHeaders();
31.        if (parser_.type == HTTP_REQUEST)
32.            argv[A_URL] = url_.ToString(env());
33.    }
34.
35.    num_fields_ = 0;
36.    num_values_ = 0;
37.
38.    // METHOD
39.    if (parser_.type == HTTP_REQUEST) {
40.        argv[A_METHOD] =
41.            Uint32::NewFromUnsigned(env()->isolate(), parser_.meth
od);
42.    }
43.
44.    // STATUS
```

```

45.     if (parser_.type == HTTP_RESPONSE) {
46.         argv[A_STATUS_CODE] =
47.             Integer::New(env()->isolate(), parser_.status_code);
48.         argv[A_STATUS_MESSAGE] = status_message_.ToString(env());
49.     }
50.
51.     // VERSION
52.     argv[A_VERSION_MAJOR] = Integer::New(env()->isolate(), parse
r_.http_major);
53.     argv[A_VERSION_MINOR] = Integer::New(env()->isolate(), parse
r_.http_minor);
54.
55.     bool should_keep_alive;
56.     // 是否定义了 keepalive 头
57.     should_keep_alive = llhttp_should_keep_alive(&parser_);
58.
59.     argv[A_SHOULD_KEEP_ALIVE] =
60.         Boolean::New(env()->isolate(), should_keep_alive);
61.     // 是否是升级协议
62.     argv[A_UPGRADE] = Boolean::New(env()->isolate(), parser_.upg
rade);
63.
64.     MaybeLocal<Value> head_response;
65.     {
66.         InternalCallbackScope callback_scope(
67.             this, InternalCallbackScope::kSkipTaskQueues);
68.         head_response = cb.As<Function>()->Call(
69.             env()->context(), object(), arraysize(argv), argv);
70.     }
71.
72.     int64_t val;
73.
74.     if (head_response.IsEmpty() || !head_responseToLocalChecked
()
75.                                         ->IntegerValue(env()->co
ntext()))
76.                                         .To(&val)) {
77.         got_exception_ = true;
78.         return -1;
79.     }
80.
81.     return val;
82. }
```

on_headers_complete 会执行 JS 层的 kOnHeadersComplete 钩子。

7 解析 body 时的回调

```
1. int on_body(const char* at, size_t length) {
2.     EscapableHandleScope scope(env()->isolate());
3.
4.     Local<Object> obj = object();
5.     Local<Value> cb = obj->Get(env()->context(), kOnBody).ToLocalChecked();
6.
7.     // We came from consumed stream
8.     if (current_buffer_.IsEmpty()) {
9.         // Make sure Buffer will be in parent HandleScope
10.        current_buffer_ = scope.Escape(Buffer::Copy(
11.            env()->isolate(),
12.            current_buffer_data_,
13.            current_buffer_len_).ToLocalChecked());
14.    }
15.
16.    Local<Value> argv[3] = {
17.        // 当前解析中的数据
18.        current_buffer_,
19.        // body 开始的位置
20.        Integer::NewFromUnsigned(env()->isolate(), at - current_buffer_data_),
21.        // body 当前长度
22.        Integer::NewFromUnsigned(env()->isolate(), length)
23.    };
24.
25.    MaybeLocal<Value> r = MakeCallback(cb.As<Function>(),
26.                                         arraysize(argv),
27.                                         argv);
28.
29.    return 0;
30. }
```

Node.js 中并不是每次解析 HTTP 报文的时候就新建一个 HTTP 解析器，Node.js 使用 FreeList 数据结构对 HTTP 解析器实例进行了管理。

```
1. class FreeList {
2.     constructor(name, max, ctor) {
3.         this.name = name;
4.         // 构造函数
5.         this.ctor = ctor;
6.         // 节点的最大值
```

```

7.     this.max = max;
8.     // 实例列表
9.     this.list = [];
10.    }
11.    // 分配一个实例
12.    alloc() {
13.        // 有空闲的则直接返回, 否则新建一个
14.        return this.list.length > 0 ?
15.            this.list.pop() :
16.            ReflectApply(this.ctor, this, arguments);
17.    }
18.    // 释放实例
19.    free(obj) {
20.        // 小于阈值则放到空闲列表, 否则释放(调用方负责释放)
21.        if (this.list.length < this.max) {
22.            this.list.push(obj);
23.            return true;
24.        }
25.        return false;
26.    }
27. }
```

我们看一下在 Node.js 中对 FreeList 的使用。。

```

1. const parsers = new FreeList('parsers', 1000, function parsersCb(
  ) {
2.   const parser = new HTTPParser();
3.   // 初始化字段
4.   cleanParser(parser);
5.   // 设置钩子
6.   parser.onIncoming = null;
7.   parser[kOnHeaders] = parserOnHeaders;
8.   parser[kOnHeadersComplete] = parserOnHeadersComplete;
9.   parser[kOnBody] = parserOnBody;
10.  parser[kOnMessageComplete] = parserOnMessageComplete;
11.
12.  return parser;
13.});
```

HTTP 解析器的使用

```

1. var HTTPParser = process.binding('http_parser').HTTPParser;
2. var parser = new HTTPParser(HTTPParser.REQUEST);
3.
4. const kOnHeaders = HTTPParser.kOnHeaders;
```

```
5. const kOnHeadersComplete = HTTPParser.kOnHeadersComplete;
6. const kOnBody = HTTPParser.kOnBody;
7. const kOnMessageComplete = HTTPParser.kOnMessageComplete;
8. const kOnExecute = HTTPParser.kOnExecute;
9.
10. parser[kOnHeaders] = function(headers, url) {
11.   console.log('kOnHeaders', headers.length, url);
12. }
13. parser[kOnHeadersComplete] = function(versionMajor, versionMinor
   , headers, method,
14.   url, statusCode, statusMessage, upgrade, shouldKeepAlive) {
15.   console.log('kOnHeadersComplete', headers);
16. }
17.
18. parser[kOnBody] = function(b, start, len) {
19.   console.log('kOnBody', b.slice(start).toString('utf-8'));
20. }
21. parser[kOnMessageComplete] = function() {
22.   console.log('kOnMessageComplete');
23. }
24. parser[kOnExecute] = function() {
25.   console.log('kOnExecute');
26. }
27.
28. parser.execute(Buffer.from(
29.   'GET / HTTP/1.1\r\n' +
30.   'Host: http://localhost\r\n\r\n'
31. ));
```

以上代码的输出

```
1. kOnHeadersComplete [ 'Host', 'http://localhost' ]
2. kOnMessageComplete
```

我们看到只执行了 `kOnHeadersComplete` 和 `kOnMessageComplete`。那其它几个回调什么时候会执行呢？我们接着看。我们把输入改一下。

```
1. parser.execute(Buffer.from(
2.   'GET / HTTP/1.1\r\n' +
3.   'Host: http://localhost\r\n' +
4.   'content-length: 1\r\n\r\n' +
5.   '1'
6. ));
```

上面代码的输出

```

1. kOnHeadersComplete [ 'Host', 'http://localhost', 'content-
   length', '1' ]
2. kOnBody 1
3. kOnMessageComplete

```

我们看到多了一个回调 `kOnBody`，因为我们加了一个 HTTP 头 `content-length` 指示有 `body`，所以 HTTP 解析器解析到 `body` 的时候就会回调 `kOnBody`。那 `kOnHeaders` 什么时候会执行呢？我们继续修改代码。

```

1. parser.execute(Buffer.from(
2.   'GET / HTTP/1.1\r\n' +
3.   'Host: http://localhost\r\n' +
4.   'a: b\r\n'+
5.   '// 很多'a: b\r\n'+
6.   'content-length: 1\r\n\r\n'+
7.   '1'
8. ));

```

以上代码的输出

```

1. kOnHeaders 62 /
2. kOnHeaders 22
3. kOnHeadersComplete undefined
4. kOnBody 1
5. kOnMessageComplete

```

我们看到 `kOnHeaders` 被执行了，并且执行了两次。因为如果 HTTP 头的个数达到阈值，在解析 HTTP 头部的过程中，就先 `flush` 到 JS 层（如果多次达到阈值，则回调多次），并且在解析完所有 HTTP 头后，会在 `kOnHeadersComplete` 回调之前再次回调 `kOnHeaders`（如果还有的话）。最后我们看一下 `kOnExecute` 如何触发。

```

1. var HTTPParser = process.binding('http_parser').HTTPParser;
2. var parser = new HTTPParser(HTTPParser.REQUEST);
3. var net = require('net');
4.
5. const kOnHeaders = HTTPParser.kOnHeaders;
6. const kOnHeadersComplete = HTTPParser.kOnHeadersComplete;
7. const kOnBody = HTTPParser.kOnBody;
8. const kOnMessageComplete = HTTPParser.kOnMessageComplete;
9. const kOnExecute = HTTPParser.kOnExecute;
10.
11. parser[kOnHeaders] = function(headers, url) {
12.   console.log('kOnHeaders', headers.length, url);
13. }

```

```
14. parser[kOnHeadersComplete] = function(versionMajor, versionMinor
   , headers, method,
15.           url, statusCode, statusMessage, upgrade, shouldKeepAlive
   e) {
16.     console.log('kOnHeadersComplete', headers);
17. }
18.
19. parser[kOnBody] = function(b, start, len) {
20.     console.log('kOnBody', b.slice(start).toString('utf-8'));
21. }
22. parser[kOnMessageComplete] = function() {
23.     console.log('kOnMessageComplete');
24. }
25. parser[kOnExecute] = function(a,b) {
26.     console.log('kOnExecute,解析的字节数: ',a);
27. }
28. // 启动一个服务器
29. net.createServer((socket) => {
30.     parser.consume(socket._handle);
31. }).listen(80);
32.
33. // 启动一个客户端
34. setTimeout(() => {
35.     var socket = net.connect({port: 80});
36.     socket.end('GET / HTTP/1.1\r\n' +
37.     'Host: http://localhost\r\n' +
38.     'content-length: 1\r\n\r\n' +
39.     '1');
40. }, 1000);
```

我们需要调用 `parser.consume` 方法并且传入一个 `isStreamBase` 的流 (`stream_base.cc` 定义)，才会触发 `kOnExecute`。因为 `kOnExecute` 是在 `StreamBase` 流可读时触发的。

18.2 HTTP 客户端

我们首先看一下使用 `Node.js` 作为客户端的例子。

```
1. const data = querystring.stringify({
2.   'msg': 'hi'
3. });
4.
5. const options = {
6.   hostname: 'your domain',
7.   path: '/',
8.   method: 'POST',
```

```

9.   headers: {
10.     'Content-Type': 'application/x-www-form-urlencoded',
11.     'Content-Length': Buffer.byteLength(data)
12.   }
13. };
14.
15. const req = http.request(options, (res) => {
16.   res.setEncoding('utf8');
17.   res.on('data', (chunk) => {
18.     console.log(`#${chunk}`);
19.   });
20.   res.on('end', () => {
21.     console.log('end');
22.   });
23. });
24.
25. req.on('error', (e) => {
26.   console.error(`#${e.message}`);
27. });
28. // 发送请求的数据
29. req.write(data);
30. // 设置请求结束
31. req.end();

```

我们看一下 `http.request` 的实现。

```

1. function request(url, options, cb) {
2.   return new ClientRequest(url, options, cb);
3. }

```

HTTP 客户端通过 `_http_client.js` 的 `ClientRequest` 实现，`ClientRequest` 的代码非常多，我们只分析核心的流程。我们看初始化一个请求的逻辑。

```

1. function ClientRequest(input, options, cb) {
2.   // 继承 OutgoingMessage
3.   OutgoingMessage.call(this);
4.   // 是否使用 agent
5.   let agent = options.agent;
6.   // 忽略 agent 的处理，具体参考 _http_agent.js，主要用于复用 TCP 连接
7.   this.agent = agent;
8.   // 建立连接的超时时间
9.   if (options.timeout !== undefined)
10.    this.timeout = getTimerDuration(options.timeout, 'timeout');
11.   // HTTP 头个数的阈值

```

```
12.  const maxHeaderSize = options.maxHeaderSize;
13.  this.maxHeaderSize = maxHeaderSize;
14.  // 监听响应事件
15.  if (cb) {
16.    this.once('response', cb);
17.  }
18.  // 忽略设置 http 协议的请求行或请求头的逻辑
19.  // 建立 TCP 连接后的回调
20.  const oncreate = (err, socket) => {
21.    if (called)
22.      return;
23.    called = true;
24.    if (err) {
25.      process.nextTick(() => this.emit('error', err));
26.      return;
27.    }
28.    // 建立连接成功，执行回调
29.    this.onSocket(socket);
30.    // 连接成功后发送数据
31.    this._deferToConnect(null, null, () => this._flush());
32.  };
33.
34.  // 使用 agent 时，socket 由 agent 提供，否则自己创建 socket
35.  if (this.agent) {
36.    this.agent.addRequest(this, options);
37.  } else {
38.    // 不使用 agent 则每次创建一个 socket，默认使用 net 模块的接口
39.    if (typeof options.createConnection === 'function') {
40.      const newSocket = options.createConnection(options,
41.                                                oncreate);
42.      if (newSocket && !called) {
43.        called = true;
44.        this.onSocket(newSocket);
45.      } else {
46.        return;
47.      }
48.    } else {
49.      this.onSocket(net.createConnection(options));
50.    }
51.  }
52.  // 连接成功后发送待缓存的数据
53.  this._deferToConnect(null, null, () => this._flush());
54. }
```

获取一个 ClientRequest 实例后，不管是通过 agent 还是自己创建一个 TCP 连接，在连接成功后都会执行 onSocket。

```

1. // socket 可用时的回调
2. ClientRequest.prototype.onSocket = function onSocket(socket) {
3.   process.nextTick(onSocketNT, this, socket);
4. };
5.
6. function onSocketNT(req, socket) {
7.   // 申请 socket 过程中，请求已经终止
8.   if (req.aborted) {
9.     // 不使用 agent，直接销毁 socket
10.    if (!req.agent) {
11.      socket.destroy();
12.    } else {
13.      // 使用 agent 触发 free 事件，由 agent 处理 socket
14.      req.emit('close');
15.      socket.emit('free');
16.    }
17.  } else {
18.    // 处理 socket
19.    tickOnSocket(req, socket);
20.  }
21. }
```

我们继续看 tickOnSocket

```

1. // 初始化 HTTP 解析器和注册 data 事件等，等待响应
2. function tickOnSocket(req, socket) {
3.   // 分配一个 HTTP 解析器
4.   const parser = parsers.alloc();
5.   req.socket = socket;
6.   // 初始化，处理响应报文
7.   parser.initialize(HTTPParser.RESPONSE,
8.     new HTTPClientAsyncResource('HTTPINCOMINGMESSAGE', req),
9.     req.maxHeaderSize || 0,
10.    req.insecureHTTPParser === undefined ?
11.      isLenient() : req.insecureHTTPParser);
12.   parser.socket = socket;
13.   parser.outgoing = req;
14.   req.parser = parser;
15.   socket.parser = parser;
16.   // socket 正处理的请求
17.   socket._httpMessage = req;
```

```
18.
19. // Propagate headers limit from request object to parser
20. if (typeof req.maxHeadersCount === 'number') {
21.   parser.maxHeaderPairs = req.maxHeadersCount << 1;
22. }
23. // 解析完 HTTP 头部的回调
24. parser.onIncoming = parserOnIncomingClient;
25. socket.removeListener('error', freeSocketErrorListener);
26. socket.on('error', socketErrorListener);
27. socket.on('data', socketOnData);
28. socket.on('end', socketOnEnd);
29. socket.on('close', socketCloseListener);
30. socket.on('drain', ondrain);
31.
32. if (
33.   req.timeout !== undefined ||
34.   (req.agent && req.agent.options &&
35.     req.agent.options.timeout)
36. ) {
37.   // 处理超时时间
38.   listenSocketTimeout(req);
39. }
40. req.emit('socket', socket);
41. }
```

拿到一个 socket 后，就开始监听 socket 上 http 报文的到来。并且申请一个 HTTP 解析器准备解析 http 报文，我们主要分析超时时间和 data 事件的处理逻辑。

1 超时时间的处理

```
1. function listenSocketTimeout(req) {
2.   // 设置过了则返回
3.   if (req.timeoutCb) {
4.     return;
5.   }
6.   // 超时回调
7.   req.timeoutCb = emitRequestTimeout;
8.   // Delegate socket timeout event.
9.   // 设置 socket 的超时时间，即 socket 上一定时间后没有响应则触发超时
10.  if (req.socket) {
11.    req.socket.once('timeout', emitRequestTimeout);
12.  } else {
13.    req.on('socket', (socket) => {
14.      socket.once('timeout', emitRequestTimeout);
15.    });
16.  }
```

```

17. }
18.
19. function emitRequestTimeout() {
20.   const req = this._httpMessage;
21.   if (req) {
22.     req.emit('timeout');
23.   }
24. }
```

2 处理响应数据

```

1. function socketOnData(d) {
2.   const socket = this;
3.   const req = this._httpMessage;
4.   const parser = this.parser;
5.   // 交给 HTTP 解析器处理
6.   const ret = parser.execute(d);
7.   // ...
8. }
```

当 Node.js 收到响应报文时，会把数据交给 HTTP 解析器处理。http 解析在解析的过程中会不断触发钩子函数。我们看一下 JS 层各个钩子函数的逻辑。

1 解析头部过程中执行的回调

```

1. function parserOnHeaders(headers, url) {
2.   // 保存头和 url
3.   if (this.maxHeaderPairs <= 0 ||
4.       this._headers.length < this.maxHeaderPairs) {
5.     this._headers = this._headers.concat(headers);
6.   }
7.   this._url += url;
8. }
```

2 解析完头部的回调

```

1. function parserOnHeadersComplete(versionMajor,
2.                                     versionMinor,
3.                                     headers,
4.                                     method,
5.                                     url,
6.                                     statusCode,
7.                                     statusMessage,
8.                                     upgrade,
9.                                     shouldKeepAlive) {
10.   const parser = this;
```

```
11.  const { socket } = parser;
12.  // 剩下的 HTTP 头
13.  if (headers === undefined) {
14.    headers = parser._headers;
15.    parser._headers = [];
16.  }
17.
18.  if (url === undefined) {
19.    url = parser._url;
20.    parser._url = '';
21.  }
22.
23.  // Parser is also used by http client
24.  // IncomingMessage
25.  const ParserIncomingMessage=(socket &&
26.                                socket.server &&
27.                                socket.server[kIncomingMessage]
28.                              ) ||
29.                                IncomingMessage;
30.  // 新建一个 IncomingMessage 对象
31.  const incoming = parser.incoming = new ParserIncomingMessage(s
   ocket);
32.  incoming.httpVersionMajor = versionMajor;
33.  incoming.httpVersionMinor = versionMinor;
34.  incoming.httpVersion = `${versionMajor}.${versionMinor}`;
35.  incoming.url = url;
36.  incoming.upgrade = upgrade;
37.
38.  let n = headers.length;
39.  // If parser.maxHeaderPairs <= 0 assume that there's no limit.
40.  if (parser.maxHeaderPairs > 0)
41.    n = MathMin(n, parser.maxHeaderPairs);
42.  // 更新到保存 HTTP 头的对象
43.  incoming._addHeaderLines(headers, n);
44.  // 请求方法或响应行信息
45.  if (typeof method === 'number') {
46.    // server only
47.    incoming.method = methods[method];
48.  } else {
49.    // client only
50.    incoming.statusCode = statusCode;
51.    incoming.statusMessage = statusMessage;
52.  }
53.  // 执行回调
54.  return parser.onIncoming(incoming, shouldKeepAlive);
55. }
```

我们看到解析完头部后会执行另一个回调 `onIncoming`, 并传入 `IncomingMessage` 实例, 这就是我们平时使用的 `res`。在前面分析过, `onIncoming` 设置的值是 `parserOnIncomingClient`。

```

1. function parserOnIncomingClient(res, shouldKeepAlive) {
2.   const socket = this.socket;
3.   // 请求对象
4.   const req = socket._httpMessage;
5.   // 服务器发送了多个响应
6.   if (req.res) {
7.     socket.destroy();
8.     return 0;
9.   }
10.  req.res = res;
11.
12.  if (statusIsInformational(res.statusCode)) {
13.    req.res = null;
14.    // 请求时设置了 expect 头, 则响应码为 100, 可以继续发送数据
15.    if (res.statusCode === 100) {
16.      req.emit('continue');
17.    }
18.    return 1;
19.  }
20.
21.  req.res = res;
22.  res.req = req;
23.
24.  // 等待响应结束, 响应结束后会清除定时器
25.  res.on('end', responseOnEnd);
26.  // 请求终止了或触发 response 事件, 返回 false 说明没有监听 response
  // 事件, 则丢弃数据
27.  if (req.aborted || !req.emit('response', res))
28.    res._dump();
29.
30. }
```

从源码中我们看出在解析完 HTTP 响应头时, 就执行了 `http.request` 设置的回调函数。例如下面代码中的回调。

```

1. http.request('domain', { agent }, (res) => {
2.   // 解析 body
3.   res.on('data', (data) => {
4.     //
5.   });
6.   // 解析 body 结束, 响应结束
```

```
7.     res.on('end', (data) => {
8.         //
9.     });
10.    });
11.   // ...
```

在回调里我们可以把 res 作为一个流使用，在解析完 HTTP 头后，HTTP 解析器会继续解析 HTTP body。我们看一下 HTTP 解析器在解析 body 过程中执行的回调。

```
1. function parserOnBody(b, start, len) {
2.     const stream = this.incoming;
3.     if (len > 0 && !stream._dumped) {
4.         const slice = b.slice(start, start + len);
5.         // 把数据 push 到流中，流会触发 data 事件
6.         const ret = stream.push(slice);
7.         // 数据过载，暂停接收
8.         if (!ret)
9.             readStop(this.socket);
10.    }
11. }
```

最后我们再看一下解析完 body 时 HTTP 解析器执行的回调。

```
1. function parserOnMessageComplete() {
2.     const parser = this;
3.     const stream = parser.incoming;
4.
5.     if (stream !== null) {
6.         // body 解析完了
7.         stream.complete = true;
8.         // 在 body 后可能有 trailer 头，保存下来
9.         const headers = parser._headers;
10.        if (headers.length) {
11.            stream._addHeaderLines(headers, headers.length);
12.            parser._headers = [];
13.            parser._url = '';
14.        }
15.        // 流结束
16.        stream.push(null);
17.    }
18.
19.    // 读取下一个响应，如果有的话
20.    readStart(parser.socket);
21. }
```

我们看到在解析 body 过程中会不断往流中 push 数据，从而不断触发 res 的 data 事件，最后解析 body 结束后，通过 push(null) 通知流结束，从而触发 res.end 事件。我们沿着 onSocket 函数分析完处理响应后我们再来分析请求的过程。执行完 http.request 后我们会得到一个标记请求的实例。然后执行它的 write 方法发送数据。

```

1. OutgoingMessage.prototype.write = function write(chunk, encoding,
   callback) {
2.   const ret = write_(this, chunk, encoding, callback, false);
3.   // 返回 false 说明需要等待 drain 事件
4.   if (!ret)
5.     this[kNeedDrain] = true;
6.   return ret;
7. };
8.
9. function write_(msg, chunk, encoding, callback, fromEnd) {
10.
11.   // 还没有设置 this._header 字段，则把请求行和 HTTP 头拼接到
12.   // this._header 字段
13.   if (!msg._header) {
14.     msg._implicitHeader();
15.   }
16.   let ret;
17.   // chunk 模式则需要额外加一下字段，否则直接发送
18.   if (msg.chunkedEncoding && chunk.length !== 0) {
19.     let len;
20.     if (typeof chunk === 'string')
21.       len = Buffer.byteLength(chunk, encoding);
22.     else
23.       len = chunk.length;
24.     /*
25.       chunk 模式时，http 报文的格式如下
26.       chunk 长度 回车换行
27.       数据 回车换行
28.     */
29.     msg._send(len.toString(16), 'latin1', null);
30.     msg._send(crlf_buf, null, null);
31.     msg._send(chunk, encoding, null);
32.     ret = msg._send(crlf_buf, null, callback);
33.   } else {
34.     ret = msg._send(chunk, encoding, callback);
35.   }
36.
37.   return ret;
38. }
```

我们接着看_send 函数

```
1. OutgoingMessage.prototype._send = function _send(data, encoding,
   callback) {
2.   // 头部还没有发送
3.   if (!this._headerSent) {
4.     // 是字符串则追加到头部, this._header 保存了 HTTP 请求行和 HTTP 头
5.     if (typeof data === 'string' &&
6.         (encoding === 'utf8' ||
7.          encoding === 'latin1' ||
8.          !encoding)) {
9.       data = this._header + data;
10.    } else {
11.      // 否则缓存起来
12.      const header = this._header;
13.      // HTTP 头需要放到最前面
14.      if (this.outputData.length === 0) {
15.        this.outputData = [
16.          data: header,
17.          encoding: 'latin1',
18.          callback: null
19.        ];
20.      } else {
21.        this.outputData.unshift({
22.          data: header,
23.          encoding: 'latin1',
24.          callback: null
25.        });
26.      }
27.      // 更新缓存大小
28.      this.outputSize += header.length;
29.      this._onPendingData(header.length);
30.    }
31.    // 已经在排队等待发送了, 不能修改
32.    this._headerSent = true;
33.  }
34.  return this._writeRaw(data, encoding, callback);
35.};
```

我们继续看_writeRaw

```
1. OutgoingMessage.prototype._writeRaw = function _writeRaw(data, en
   coding, callback) {
2.
3.   // 可写的时候直接发送
```

```

4.  if (conn && conn._httpMessage === this && conn.writable) {
5.    // There might be pending data in the this.output buffer.
6.    // 如果有缓存的数据则先发送缓存的数据
7.    if (this.outputData.length) {
8.      this._flushOutput(conn);
9.    }
10.   // 接着发送当前需要发送的
11.   return conn.write(data, encoding, callback);
12. }
13. // 否先缓存
14. this.outputData.push({ data, encoding, callback });
15. this.outputSize += data.length;
16. this._onPendingData(data.length);
17. return this.outputSize < HIGH_WATER_MARK;
18. }
19.
20. OutgoingMessage.prototype._flushOutput = function _flushOutput(s
ocket) {
21.   // 之前设置了加塞，则操作 socket 先积攒数据
22.   while (this[kCorked]) {
23.     this[kCorked]--;
24.     socket.cork();
25.   }
26.
27.   const outputLength = this.outputData.length;
28.   if (outputLength <= 0)
29.     return undefined;
30.
31.   const outputData = this.outputData;
32.   socket.cork();
33.   // 把缓存的数据写到 socket
34.   let ret;
35.   for (let i = 0; i < outputLength; i++) {
36.     const { data, encoding, callback } = outputData[i];
37.     ret = socket.write(data, encoding, callback);
38.   }
39.   socket.uncork();
40.
41.   this.outputData = [];
42.   this._onPendingData(-this.outputSize);
43.   this.outputSize = 0;
44.
45.   return ret;
46. };

```

写完数据后，我们还需要执行 end 函数标记 HTTP 请求的结束。

```
1. OutgoingMessage.prototype.end = function end(chunk, encoding, callback) {
2.   // 还没结束
3.   // 加塞
4.   if (this.socket) {
5.     this.socket.cork();
6.   }
7.
8.   // 流结束后回调
9.   if (typeof callback === 'function')
10.    this.once('finish', callback);
11.   // 数据写入底层后的回调
12.   const finish = onFinish.bind(undefined, this);
13.   // chunk 模式后面需要发送一个 0\r\n 结束标记, 否则不需要结束标记
14.   if (this._hasBody && this.chunkedEncoding) {
15.     this._send('0\r\n' +
16.               this._trailer + '\r\n', 'latin1', finish);
17.   } else {
18.     this._send('', 'latin1', finish);
19.   }
20.   // uncork 解除塞子, 发送数据
21.   if (this.socket) {
22.     // Fully uncork connection on end().
23.     this.socket._writableState.corked = 1;
24.     this.socket.uncork();
25.   }
26.   this[kCorked] = 0;
27.   // 标记执行了 end
28.   this.finished = true;
29.   // 数据发完了
30.   if (this.outputData.length === 0 &&
31.       this.socket &&
32.       this.socket._httpMessage === this) {
33.     this._finish();
34.   }
35.
36.   return this;
37. };
```

18.3 HTTP 服务器

本节我们来分析使用 Node.js 作为服务器的例子。

```
1. const http = require('http');
```

```

2. http.createServer((req, res) => {
3.   res.write('hello');
4.   res.end();
5. })
6. .listen(3000);

```

接着我们沿着 `createServer` 分析 Node.js 作为服务器的原理。

```

1. function createServer(opts, requestListener) {
2.   return new Server(opts, requestListener);
3. }

```

我们看 `Server` 的实现

```

1. function Server(options, requestListener) {
2.   // 可以自定义表示请求的对象和响应的对象
3.   this[kIncomingMessage] = options.IncomingMessage || IncomingMes
  sage;
4.   this[kServerResponse] = options.ServerResponse || ServerRespons
  e;
5.   // HTTP 头个数的阈值
6.   const maxHeaderSize = options.maxHeaderSize;
7.   this.maxHeaderSize = maxHeaderSize;
8.   // 允许半关闭
9.   net.Server.call(this, { allowHalfOpen: true });
10.  // 有请求时的回调
11.  if (requestListener) {
12.    this.on('request', requestListener);
13.  }
14.  // 服务器 socket 读端关闭时是否允许继续处理队列里的响应 (tcp 上有多个
  请求, 管道化)
15.  this.httpAllowHalfOpen = false;
16.  // 有连接时的回调, 由 net 模块触发
17.  this.on('connection', connectionListener);
18.  // 服务器下所有请求和响应的超时时间
19.  this.timeout = 0;
20.  // 同一个 TCP 连接上, 两个请求之前最多间隔的时间
21.  this.keepAliveTimeout = 5000;
22.  this.maxHeadersCount = null;
23.  // 解析头部的超时时间, 防止 ddos
24.  this.headersTimeout = 60 * 1000; // 60 seconds
25. }

```

接着调用 listen 函数，因为 HTTP Server 继承于 net.Server，net.Server 的 listen 函数前面我们已经分析过，就不再分析。当有请求到来时，会触发 connection 事件。从而执行 connectionListener。

```
1. function connectionListener(socket) {
2.   defaultTriggerAsyncIdScope(
3.     getOrSetAsyncId(socket), connectionListenerInternal, this, so
cket
4.   );
5. }
6.
7. // socket 表示新连接
8. function connectionListenerInternal(server, socket) {
9.   // socket 所属 server
10.  socket.server = server;
11.  // 设置连接的超时时间，超时处理函数为 socketOnTimeout
12.  if (server.timeout && typeof socket.setTimeout === 'function')
      socket.setTimeout(server.timeout);
13.  socket.on('timeout', socketOnTimeout);
14.  // 分配一个 HTTP 解析器
15.  const parser = parsers.alloc();
16.  // 解析请求报文
17.  parser.initialize(
18.    HTTPParser.REQUEST,
19.    new HTTPSAsyncResource('HTTPINCOMINGMESSAGE', socket),
20.    server.maxHeaderSize || 0,
21.    server.insecureHTTPParser === undefined ?
22.      isLenient() : server.insecureHTTPParser,
23.  );
24.  parser.socket = socket;
25.  // 记录开始解析头部的开始时间
26.  parser.parsingHeadersStart = nowDate();
27.  socket.parser = parser;
28.  if (typeof server.maxHeadersCount === 'number') {
29.    parser.maxHeaderPairs = server.maxHeadersCount << 1;
30.  }
31.
32.  const state = {
33.    onData: null,
34.    onEnd: null,
35.    onClose: null,
36.    onDrain: null,
37.    // 同一 TCP 连接上，请求和响应的队列，线头阻塞的原理
38.    outgoing: [],
39.    incoming: [],
```

```

40.    // 待发送的字节数, 如果超过阈值, 则先暂停接收请求的数据
41.    outgoingData: 0,
42.    /*
43.        是否重新设置了 timeout, 用于响应一个请求时,
44.        标记是否重新设置超时时间的标记
45.    */
46.    keepAliveTimeoutSet: false
47. };
48. // 监听 tcp 上的数据, 开始解析 http 报文
49. state.onData = socketOnData.bind(undefined,
50.                                 server,
51.                                 socket,
52.                                 parser,
53.                                 state);
54. state.onEnd = socketOnEnd.bind(undefined,
55.                                 server,
56.                                 socket,
57.                                 parser,
58.                                 state);
59. state.onClose = socketOnClose.bind(undefined, socket, state);

60. state.onDrain = socketOnDrain.bind(undefined, socket, state);

61. socket.on('data', state.onData);
62. socket.on('error', socketOnError);
63. socket.on('end', state.onEnd);
64. socket.on('close', state.onClose);
65. socket.on('drain', state.onDrain);
66. // 解析 HTTP 头部完成后执行的回调
67. parser.onIncoming = parserOnIncoming.bind(undefined,
68.                                              server,
69.                                              socket,
70.                                              state);
71. socket.on('resume', onSocketResume);
72. socket.on('pause', onSocketPause);
73.
74. /*
75.     如果 handle 是继承 StreamBase 的流则执行 consume 消费 http
76.     请求报文, 而不是上面的 onData, tcp 模块的 isStreamBase 为 true
77. */
78. if (socket._handle && socket._handle.isStreamBase &&
79.      !socket._handle._consumed) {
80.     parser._consumed = true;
81.     socket._handle._consumed = true;
82.     parser.consume(socket._handle);

```

```
83. }
84. parser[kOnExecute] =
85.     onParserExecute.bind(undefined,
86.         server,
87.         socket,
88.         parser,
89.         state);
90.
91. socket._paused = false;
92. }
```

执行完 `connectionListener` 后就开始等待 `tcp` 上数据的到来，即 `HTTP` 请求报文。上面代码中 `Node.js` 监听了 `socket` 的 `data` 事件，同时注册了钩子 `kOnExecute`。`data` 事件我们都知道是流上有数据到来时触发的事件。我们看一下 `socketOnData` 做了什么事情。

```
1. function socketOnData(server, socket, parser, state, d) {
2.     // 交给 HTTP 解析器处理，返回已经解析的字节数
3.     const ret = parser.execute(d);
4.     onParserExecuteCommon(server, socket, parser, state, ret, d);
5. }
```

`socketOnData` 的处理逻辑是当 `socket` 上有数据，然后交给 `HTTP` 解析器处理。这看起来没什么问题，那么 `kOnExecute` 是做什么的呢？`kOnExecute` 钩子函数的值是 `onParserExecute`，这个看起来也是解析 `tcp` 上的数据的，看起来和 `onSocketData` 是一样的作用，难道 `tcp` 上的数据有两个消费者？我们看一下 `kOnExecute` 什么时候被回调的。

```
1. void OnStreamRead(ssize_t nread, const uv_buf_t& buf) override {
2.
3.     Local<Value> ret = Execute(buf.base, nread);
4.     Local<Value> cb =
5.         object()->Get(env()->context(), kOnExecute).ToLocalChecke
d();
6.     MakeCallback(cb.As<Function>(), 1, &ret);
7. }
```

`OnStreamRead` 是 `node_http_parser.cc` 实现的函数，所以 `kOnExecute` 在 `node_http_parser.cc` 中的 `OnStreamRead` 中被回调，那么 `OnStreamRead` 又是什么时候被回调的呢？在 C++ 层章节我们分析过，`OnStreamRead` 是 `Node.js` 中 C++ 层流操作的通用函数，当流有数据的时候就会执行该回调。而且 `OnStreamRead` 中也会把数据交给 `HTTP` 解析器解析。这看起来真的有两个消费者？这就很奇怪，为什么一份数据会交给 `HTTP` 解析器处理两次？

```

1. if (socket._handle && socket._handle.isStreamBase && !socket._handle._consumed) {
2.     parser._consumed = true;
3.     socket._handle._consumed = true;
4.     parser.consume(socket._handle);
5. }

```

因为 TCP 流是继承 StreamBase 类的，所以 if 成立。我们看一下 consume 的实现。

```

1. static void Consume(const FunctionCallbackInfo<Value>& args) {
2.     Parser* parser;
3.     ASSIGN_OR_RETURN_UNWRAP(&parser, args.Holder());
4.     CHECK(args[0]->IsObject());
5.     StreamBase* stream = StreamBase::FromObject(args[0].As<Object>());
6.     CHECK_NOT_NULL(stream);
7.     stream->PushStreamListener(parser);
8. }

```

HTTP 解析器把自己注册为 TCP stream 的一个 listener。这会使得 TCP 流上的数据由 node_http_parser.cc 的 OnStreamRead 直接消费，而不是触发 onData 事件。在 OnStreamRead 中会源源不断地把数据交给 HTTP 解析器处理，在解析的过程中，会不断触发对应的钩子函数，直到解析完 HTTP 头部后执行 parserOnIncoming。

```

1. function parserOnIncoming(server, socket, state, req, keepAlive)
{
2.     // 需要重置定时器
3.     resetSocketTimeout(server, socket, state);
4.     // 设置了 keepAlive 则响应后需要重置一些状态
5.     if (server.keepAliveTimeout > 0) {
6.         req.on('end', resetHeadersTimeoutOnReqEnd);
7.     }
8.
9.     // 标记头部解析完毕
10.    socket.parser.parsingHeadersStart = 0;
11.
12.    // 请求入队（待处理的请求队列）
13.    state.incoming.push(req);
14.
15.    if (!socket._paused) {
16.        const ws = socket._writableState;
17.        // 待发送的数据过多，先暂停接收请求数据
18.        if (ws.needDrain ||
19.            state.outgoingData >= socket.writableHighWaterMark) {
20.            socket._paused = true;

```

```
21.     socket.pause();
22. }
23. }
24. // 新建一个表示响应的对象
25. const res = new server[kServerResponse](req);
26. // 设置数据写入待发送队列时触发的回调, 见 OutgoingMessage
27. res._onPendingData = updateOutgoingData.bind(undefined,
28.                                                 socket,
29.                                                 state);
30. // 根据请求的 HTTP 头设置是否支持 keepalive (管道化)
31. res.shouldKeepAlive = keepAlive;
32. /*
33.   socket 当前已经在处理其它请求的响应, 则先排队,
34.   否则挂载响应对象到 socket, 作为当前处理的响应
35. */
36. if (socket._httpMessage) {
37.   state.outgoing.push(res);
38. } else {
39.   res.assignSocket(socket);
40. }
41.
42. // 响应处理完毕后, 需要做一些处理
43. res.on('finish',
44.         resOnFinish.bind(undefined,
45.                           req,
46.                           res,
47.                           socket,
48.                           state,
49.                           server));
50. // 有 expect 请求头, 并且是 http1.1
51. if (req.headers.expect !== undefined &&
52.     (req.httpVersionMajor === 1 &&
53.      req.httpVersionMinor === 1))
54. {
55.   // Expect 头的值是否是 100-continue
56.   if (continueExpression.test(req.headers.expect)) {
57.     res._expect_continue = true;
58.   /*
59.     监听了 checkContinue 事件则触发,
60.     否则直接返回允许继续请求并触发 request 事件
61.   */
62.   if (server.listenerCount('checkContinue') > 0) {
63.     server.emit('checkContinue', req, res);
64.   } else {
65.     res.writeContinue();
```

```

66.         server.emit('request', req, res);
67.     }
68. } else if (server.listenerCount('checkExpectation') > 0) {
69.     /*
70.      值异常，监听了 checkExpectation 事件，
71.      则触发，否则返回 417 拒绝请求
72.     */
73.     server.emit('checkExpectation', req, res);
74. } else {
75.     res.writeHead(417);
76.     res.end();
77. }
78. } else {
79.     // 触发 request 事件说明有请求到来
80.     server.emit('request', req, res);
81. }
82. return 0; // No special treatment.
83. }

```

我们看到这里会触发 request 事件通知用户有新请求到来，用户就可以处理请求了。我们看到 Node.js 解析头部的时候就会执行上层回调，通知有新请求到来，并传入 request 和 response 作为参数，分别对应的是表示请求和响应的对象。另外 Node.js 本身是不会解析 body 部分的，我们可以通过以下方式获取 body 的数据。

```

1. const server = http.createServer((request, response) => {
2.   request.on('data', (chunk) => {
3.     // 处理 body
4.   });
5.   request.on('end', () => {
6.     // body 结束
7.   });
8. })

```

18.3.1 HTTP 管道化的原理和实现

HTTP1.0 的时候，不支持管道化，客户端发送一个请求的时候，首先建立 TCP 连接，然后服务器返回一个响应，最后断开 TCP 连接，这种是最简单的实现方式，但是每次发送请求都需要走三次握手显然会带来一定的时间损耗，所以 HTTP1.1 的时候，支持了管道化。管道化的意思就是可以在一个 TCP 连接上发送多个请求，这样服务器就可以同时处理多个请求，但是由于 HTTP1.1 的限制，多个请求的响应需要按序返回。因为在 HTTP1.1 中，没有标记请求和响应的对应关系。所以 HTTP 客户端会假设第一个返回的响应是对应第一个请求的。如果乱序返回，就会导致问题，如图 18-2 所示。

哪个请求的响应？

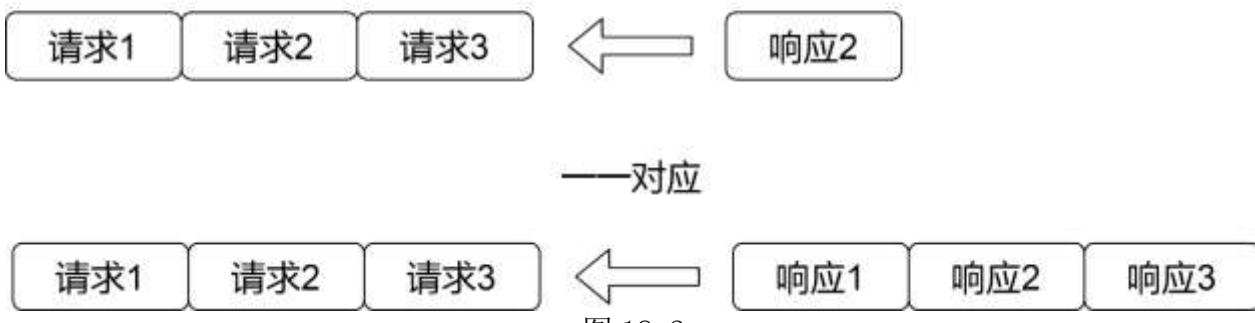


图 18-2

而在 HTTP 2.0 中，每个请求会分配一个 id，响应中也会返回对应的 id，这样就算乱序返回，HTTP 客户端也可以知道响应所对应的请求。在 HTTP 1.1 这种情况下，HTTP 服务器的实现就会变得复杂，服务器可以以串行的方式处理请求，当前面请求的响应返回到客户端后，再继续处理下一个请求，这种实现方式是相对简单的，但是很明显，这种方式相对来说还是比较低效的，另一种实现方式是并行处理请求，串行返回，这样可以让请求得到尽快的处理，比如两个请求都访问数据库，那并行处理两个请求就会比串行快得多，但是这种实现方式相对比较复杂，Node.js 就是属于这种方式，下面我们来看一下 Node.js 中是如何实现的。前面分析过，Node.js 在解析完 HTTP 头部的时候会执行 parserOnIncoming。

```
1. function parserOnIncoming(server, socket, state, req, keepAlive)
  {
2.   // 标记头部解析完毕
3.   socket.parser.parsingHeadersStart = 0;
4.   // 请求入队
5.   state.incoming.push(req);
6.   // 新建一个表示响应的对象，一般是 ServerResponse
7.   const res = new server[kServerResponse](req);
8.   /*
9.     socket 当前已经在处理其它请求的响应，则先排队，
10.    否则挂载响应对象到 socket，作为当前处理的响应
11.   */
12.   if (socket._httpMessage) {
13.     state.outgoing.push(res);
14.   } else {
15.     res.assignSocket(socket); // socket._httpMessage = res;
16.   }
17.   // 响应处理完毕后，需要做一些处理
18.   res.on('finish', resOnFinish.bind(undefined,
19.           req,
20.           res,
21.           socket,
22.           state,
```

```

23.           server));
24.   // 触发 request 事件说明有请求到来
25.   server.emit('request', req, res);
26.   return 0;
27. }

```

当 Node.js 解析 HTTP 请求头完成后，就会创建一个 ServerResponse 对象表示响应。然后判断当前是否有正在处理的响应，如果有则排队等待处理，否则把新建的 ServerResponse 对象作为当前需要处理的响应。最后触发 request 事件通知用户层。用户就可以进行请求的处理了。我们看到 Node.js 维护了两个队列，分别是请求和响应队列，如图 18-3 所示。



图 18-3

当前处理的请求在请求队列的队首，该请求对应的响应会挂载到 socket 的 _httpMessage 属性上。但是我们看到 Node.js 会触发 request 事件通知用户有新请求到来，所有在管道化的情况下，Node.js 会并行处理多个请求（如果是 cpu 密集型的请求则实际上还是会变成串行，这和 Node.js 的单线程相关）。那 Node.js 是如何控制响应的顺序的呢？我们知道每次触发 request 事件的时候，我们都会执行一个函数。比如下面的代码。

```

1. http.createServer((req, res) => {
2.   // 一些网络 IO
3.   res.writeHead(200, { 'Content-Type': 'text/plain' });
4.   res.end('okay');
5. });

```

我们看到每个请求的处理是独立的。假设每个请求都去操作数据库，如果请求 2 比请求 1 先完成数据库的操作，从而请求 2 先执行 res.write 和 res.end。那岂不是请求 2 先返回？我们看一下 ServerResponse 和 OutgoingMessage 的实现，揭开迷雾。ServerResponse 是 OutgoingMessage 的子类。write 函数是在 OutgoingMessage 中实现的，write 的调用链路很长，我们不层层分析，直接看最后的节点。

```

1. function _writeRaw(data, encoding, callback) {
2.   const conn = this.socket;
3.   // socket 对应的响应是自己并且可写
4.   if (conn && conn._httpMessage === this && conn.writable) {
5.     // 如果有缓存的数据则先发送缓存的数据
6.     if (this._outputData.length) {
7.       this._flushOutput(conn);
8.     }

```

```
9.     // 接着发送当前需要发送的
10.    return conn.write(data, encoding, callback);
11. }
12. // socket 当前处理的响应对象不是自己，则先缓存数据。
13. this.outputData.push({ data, encoding, callback });
14. this.outputSize += data.length;
15. this._onPendingData(data.length);
16. return this.outputSize < HIGH_WATER_MARK;
17. }
```

我们看到我们调用 res.write 的时候，Node.js 会首先判断，res 是不是属于当前处理中响应，如果是才会真正发送数据，否则会先把数据缓存起来。分析到这里，相信大家已经差不多明白 Node.js 是如何控制响应按序返回的。最后我们看一下这些缓存的数据什么时候会被发送出去。前面代码已经贴过，当一个响应结束的时候，Node.js 会做一些处理。

```
1. res.on('finish', resOnFinish.bind(undefined,
2.                     req,
3.                     res,
4.                     socket,
5.                     state,
6.                     server));
```

我们看看 resOnFinish

```
1. function resOnFinish(req, res, socket, state, server) {
2.   // 删除响应对应的请求
3.   state.incoming.shift();
4.   clearIncoming(req);
5.   // 解除 socket 上挂载的响应对象
6.   res.detachSocket(socket);
7.   req.emit('close');
8.   process.nextTick(emitCloseNT, res);
9.   // 是不是最后一个响应
10.  if (res._last) {
11.    // 是则销毁 socket
12.    if (typeof socket.destroySoon === 'function') {
13.      socket.destroySoon();
14.    } else {
15.      socket.end();
16.    }
17.  } else if (state.outgoing.length === 0) {
18.    /*
19.     没有待处理的响应了，则重新设置超时时间，
20.     等待请求的到来，一定时间内没有请求则触发 timeout 事件
21.  */
```

```

22.     if (server.keepAliveTimeout &&
23.         typeof socket.setTimeout === 'function') {
24.         socket.setTimeout(server.keepAliveTimeout);
25.         state.keepAliveTimeoutSet = true;
26.     }
27. } else {
28.     // 获取下一个要处理的响应
29.     const m = state.outgoing.shift();
30.     // 挂载到 socket 作为当前处理的响应
31.     if (m) {
32.         m.assignSocket(socket);
33.     }
34. }
35. }

```

我们看到，Node.js 处理完一个响应后，会做一些判断。分别有三种情况，我们分开分析。

1 是否是最后一个响应

什么情况下，会被认为是最后一个响应的？因为响应和请求是一一对应的，最后一个响应就意味着最后一个请求了，那么什么时候被认为是最最后一个请求呢？当非管道化的情况下，一个请求一个响应，然后关闭 TCP 连接，所以非管道化的情况下，tcp 上的第一个也是唯一一个请求就是最后一个请求。在管道化的情况下，理论上就没有所谓的最后一个响应。但是实现上会做一些限制。在管道化的情况下，每一个响应可以通过设置 HTTP 响应头 connection 来定义是否发送该响应后就断开连接，我们看一下 Node.js 的实现。

```

1. // 是否显示删除过 connection 头，是则响应后断开连接，并标记当前响应是最
   后一个
2. if (this._removedConnection) {
3.     this._last = true;
4.     this.shouldKeepAlive = false;
5. } else if (!state.connection) {
6.     /*
7.      没有显示设置了 connection 头，则取默认行为
8.      1 Node.js 的 shouldKeepAlive 默认为 true，也可以根据请求报文里
9.      的 connection 头定义
10.     2 设置 content-length 或使用 chunk 模式才能区分响应报文编边界，
11.        才能支持 keepalive
12.     3 使用了代理，代理是复用 TCP 连接的，支持 keepalive
13.     */
14.     const shouldSendKeepAlive = this.shouldKeepAlive &&
15.         (state.contLen ||
16.          this.useChunkedEncodingByDefault ||
17.          this.agent);
18.     if (shouldSendKeepAlive) {

```

```
19.     header += 'Connection: keep-alive\r\n';
20. } else {
21.     this._last = true;
22.     header += 'Connection: close\r\n';
23. }
24. }
```

另外当读端关闭的时候，也被认为是最后一个请求，毕竟不会再发送请求了。我们看一下读端关闭的逻辑。

```
1. function socketOnEnd(server, socket, parser, state) {
2.     const ret = parser.finish();
3.
4.     if (ret instanceof Error) {
5.         socketOnError.call(socket, ret);
6.         return;
7.     }
8.     // 不允许半开关则终止请求的处理，不响应，关闭写端
9.     if (!server.httpAllowHalfOpen) {
10.         abortIncoming(state.incoming);
11.         if (socket.writable) socket.end();
12.     } else if (state.outgoing.length) {
13.         /*
14.             允许半开关，并且还有响应需要处理，
15.             标记响应队列最后一个节点为最后的响应，
16.             处理完就关闭 socket 写端
17.         */
18.         state.outgoing[state.outgoing.length - 1]._last = true;
19.     } else if (socket._httpMessage) {
20.         /*
21.             没有等待处理的响应了，但是还有正在处理的响应，
22.             则标记为最后一个响应
23.         */
24.         socket._httpMessage._last = true;
25.     } else if (socket.writable) {
26.         // 否则关闭 socket 写端
27.         socket.end();
28.     }
29. }
```

以上就是 Node.js 中判断是否是最后一个响应的情况，如果一个响应被认为是最后一个响应，那么发送响应后就会关闭连接。

2 响应队列为空

我们继续看一下如果不是最后一个响应的时候，Node.js 又是怎么处理的。如果当前的待处理响应队列为空，说明当前处理的响应是目前最后一个需要处理的，但是不是 TCP 连接

上最后一个响应，这时候，Node.js 会设置超时时间，如果超时还没有新的请求，则 Node.js 会关闭连接。

3 响应队列非空

如果当前待处理队列非空，处理完当前请求后会继续处理下一个响应。并从队列中删除该响应。我们看一下 Node.js 是如何处理下一个响应的。

```

1. // 把响应对象挂载到 socket，标记 socket 当前正在处理的响应
2. ServerResponse.prototype.assignSocket = function assignSocket(socket) {
3.   // 挂载到 socket 上，标记是当前处理的响应
4.   socket._httpMessage = this;
5.   socket.on('close', onServerResponseClose);
6.   this.socket = socket;
7.   this.emit('socket', socket);
8.   this._flush();
9. };

```

我们看到 Node.js 是通过 _httpMessage 标记当前处理的响应的，配合响应队列来实现响应的按序返回。标记完后执行 _flush 发送响应的数据（如果这时候请求已经被处理完成）

```

1. OutgoingMessage.prototype._flush = function _flush() {
2.   const socket = this.socket;
3.   if (socket && socket.writable) {
4.     const ret = this._flushOutput(socket);
5.   };
6.
7. OutgoingMessage.prototype._flushOutput = function _flushOutput(socket) {
8.   // 之前设置了加塞，则操作 socket 先积攒数据
9.   while (this[kCorked]) {
10.     this[kCorked]--;
11.     socket.cork();
12.   }
13.
14.   const outputLength = this.outputData.length;
15.   // 没有数据需要发送
16.   if (outputLength <= 0)
17.     return undefined;
18.
19.   const outputData = this.outputData;
20.   // 加塞，让数据一起发送出去
21.   socket.cork();
22.   // 把缓存的数据写到 socket
23.   let ret;

```

```
24. for (let i = 0; i < outputLength; i++) {  
25.   const { data, encoding, callback } = outputData[i];  
26.   ret = socket.write(data, encoding, callback);  
27. }  
28. socket.uncork();  
29.  
30. this.outputData = [];  
31. this._onPendingData(-this.outputSize);  
32. this.outputSize = 0;  
33.  
34. return ret;  
35. }
```

以上就是 Node.js 中对于管道化的实现。

18.3.2 HTTP Connect 方法的原理和实现

分析 HTTP Connect 实现之前我们首先看一下为什么需要 HTTP Connect 方法或者说它出现的背景。Connect 方法主要用于代理服务器的请求转发。我们看一下传统 HTTP 服务器的工作原理，如图 18-4 所示。

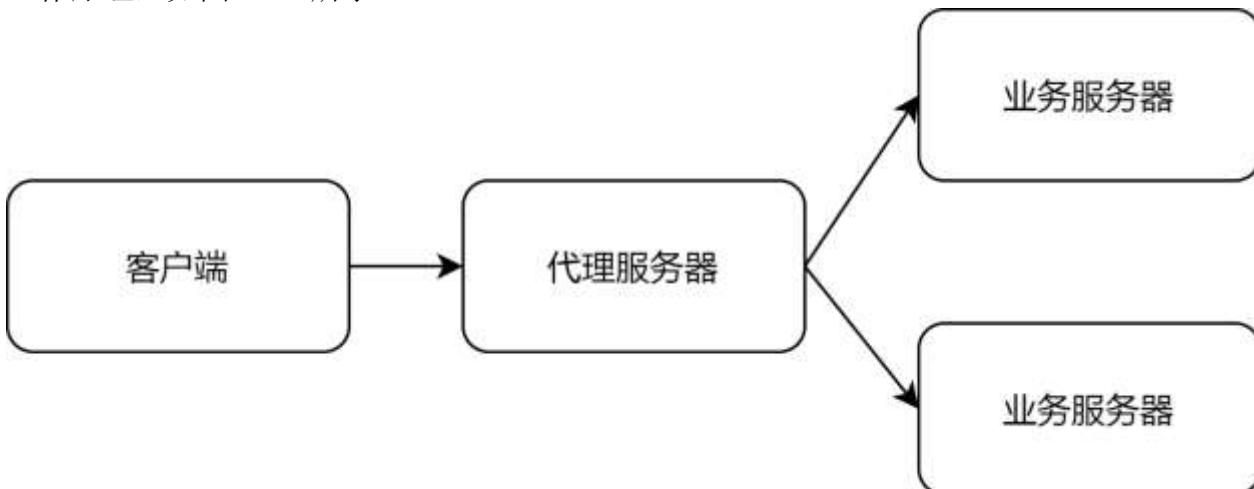


图 18-4

- 1 客户端和代理服务器建立 TCP 连接
- 2 客户端发送 HTTP 请求给代理服务器
- 3 代理服务器解析 HTTP 协议，根据配置拿到业务服务器的地址
- 4 代理服务器和业务服务器建立 TCP 连接，通过 HTTP 协议或者其它协议转发请求
- 5 业务服务器返回数据，代理服务器回复 HTTP 报文给客户端。

接着我们看一下 HTTPS 服务器的原理。

- 1 客户端和服务器建立 TCP 连接
- 2 服务器通过 TLS 报文返回证书信息，并和客户端完成后续的 TLS 通信。
- 3 完成 TLS 通信后，后续发送的 HTTP 报文会经过 TLS 层加密解密后再传输。

那么如果我们想实现一个 HTTPS 的代理服务器怎么做呢？因为客户端只管和直接相连的服务器进行 HTTPS 的通信，如果我们的业务服务器前面还有代理服务器，那么代理服务器就必须要有证书才能和客户端完成 TLS 握手，从而进行 HTTPS 通信。代理服务器和业务服务器使用 HTTP 或者 HTTPS 还是其它协议都可以。这样就意味着我们所有的服务的证书都需要放到代理服务器上，这种场景的限制是，代理服务器和业务服务器都由我们自己管理或者公司统一管理。如果我们想加一个代理对业务服务器不感知那怎么办呢（比如写一个代理服务器用于开发调试）？有一种方式就是为我们的代理服务器申请一个证书，这样客户端和代理服务器就可以完成正常的 HTTPS 通信了。从而也就可以完成代理的功能。另外一种方式就是 HTTP Connect 方法。HTTP Connect 方法的作用是指示服务器帮忙建立一条 TCP 连接到真正的业务服务器，并且透传后续的数据，这样不申请证书也可以完成代理的功能，如图 18-5 所示。

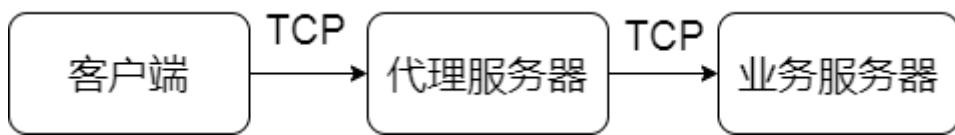


图 18-5

这时候代理服务器只负责透传两端的数据，不像传统的方式一样解析请求然后再转发。这样客户端和业务服务器就可以自己完成 TLS 握手和 HTTPS 通信。代理服务器就像不存在一样。了解了 Connect 的原理后看一下来自 Node.js 官方的一个例子。

```

1. const http = require('http');
2. const net = require('net');
3. const { URL } = require('url');
4. // 创建一个 HTTP 服务器作为代理服务器
5. const proxy = http.createServer((req, res) => {
6.   res.writeHead(200, { 'Content-Type': 'text/plain' });
7.   res.end('okay');
8. });
9. // 监听 connect 事件，有 http connect 请求时触发
10. proxy.on('connect', (req, clientSocket, head) => {
11.   // 获取真正要连接的服务器地址并发起连接
12.   const { port, hostname } = new URL(`http://${req.url}`);
13.   const serverSocket = net.connect(port || 80, hostname, () => {
14.     // 连接成功告诉客户端
15.     clientSocket.write('HTTP/1.1 200 Connection Established\r\n'
+ 'Proxy-agent: Node.js-Proxy\r\n'
+ '\r\n');
16.     // 透传客户端和服务器的数据
17.     serverSocket.write(head);
18.     serverSocket.pipe(clientSocket);
19.     clientSocket.pipe(serverSocket);
20.   });
21. });
22. 
```

```
23. });
24.
25. proxy.listen(1337, '127.0.0.1', () => {
26.
27.   const options = {
28.     port: 1337,
29.     // 连接的代理服务器地址
30.     host: '127.0.0.1',
31.     method: 'CONNECT',
32.     // 我们需要真正想访问的服务器地址
33.     path: 'www.baidu.com',
34.   };
35.   // 发起 http connect 请求
36.   const req = http.request(options);
37.   req.end();
38.   // connect 请求成功后触发
39.   req.on('connect', (res, socket, head) => {
40.     // 发送真正的请求
41.     socket.write('GET / HTTP/1.1\r\n' +
42.                 'Host: www.baidu.com\r\n' +
43.                 'Connection: close\r\n' +
44.                 '\r\n');
45.     socket.on('data', (chunk) => {
46.       console.log(chunk.toString());
47.     });
48.     socket.on('end', () => {
49.       proxy.close();
50.     });
51.   });
52. });
```

官网的这个例子很好地说明了 Connect 的原理，如图 18-6 所示。

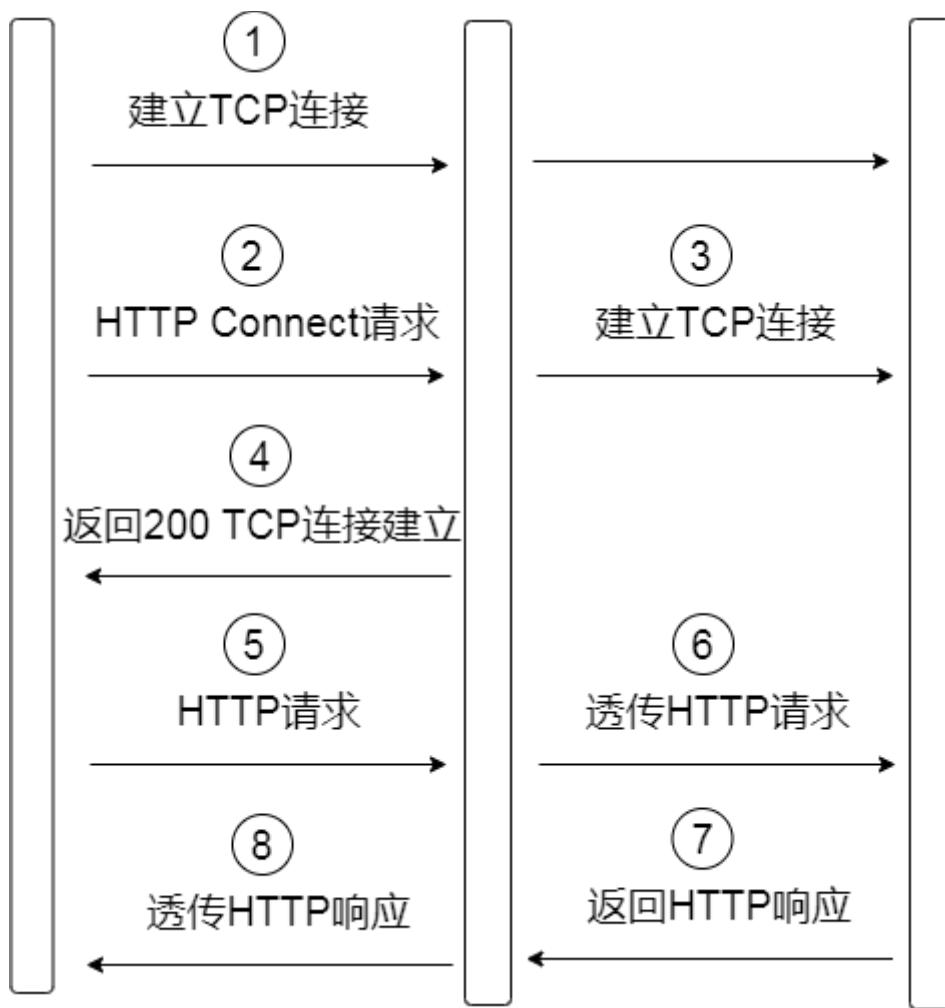


图 18-6

下面我们看一下 Node.js 中 Connect 的实现。我们从 HTTP Connect 请求开始。之前已经分析过，客户端和 Node.js 服务器建立 TCP 连接后，Node.js 收到数据的时候会交给 HTTP 解析器处理，

```

1. // 连接上有数据到来
2. function socketOnData(server, socket, parser, state, d) {
3.   // 交给 HTTP 解析器处理，返回已经解析的字节数
4.   const ret = parser.execute(d);
5.   onParserExecuteCommon(server, socket, parser, state, ret, d);
6. }

```

HTTP 解析数据的过程中会不断回调 Node.js 的回调，然后执行 onParserExecuteCommon。我们这里只关注当 Node.js 解析完所有 HTTP 请求头后执行 parserOnHeadersComplete。

```

1. function parserOnHeadersComplete(versionMajor, versionMinor, headers, method,
2.                                     url, statusCode, statusMessage,
3.                                     upgrade,

```

```
3.                                         shouldKeepAlive) {  
4.     const parser = this;  
5.     const { socket } = parser;  
6.  
7.     // IncomingMessage  
8.     const ParserIncomingMessage = (socket && socket.server &&  
9.                                         socket.server[kIncomingMessage])  
||  
10.                                         IncomingMessage;  
11.     // 新建一个 IncomingMessage 对象  
12.     const incoming = parser.incoming = new ParserIncomingMessage(s  
ocket);  
13.     incoming.httpVersionMajor = versionMajor;  
14.     incoming.httpVersionMinor = versionMinor;  
15.     incoming.httpVersion = `${versionMajor}.${versionMinor}`;  
16.     incoming.url = url;  
17.     // 是否是 connect 请求或者 upgrade 请求  
18.     incoming.upgrade = upgrade;  
19.  
20.     // 执行回调  
21.     return parser.onIncoming(incoming, shouldKeepAlive);  
22. }
```

我们看到解析完 HTTP 头后，Node.js 会创建一个表示请求的对象 IncomingMessage，然后回调 onIncoming。

```
1. function parserOnIncoming(server, socket, state, req, keepAlive)  
{  
2.     // 请求是否是 connect 或者 upgrade  
3.     if (req.upgrade) {  
4.         req.upgrade = req.method === 'CONNECT' ||  
5.                     server.listenerCount('upgrade') > 0;  
6.         if (req.upgrade)  
7.             return 2;  
8.     }  
9.     // ...  
10. }
```

Node.js 解析完头部并且执行了响应的钩子函数后，会执行 onParserExecuteCommon。

```
1. function onParserExecuteCommon(server, socket, parser, state, ret  
, d) {  
2.     if (ret instanceof Error) {  
3.         prepareError(ret, parser, d);  
4.         ret.rawPacket = d || parser.getCurrentBuffer();
```

```

5.     socketOnError.call(socket, ret);
6. } else if (parser.incoming && parser.incoming.upgrade) {
7.     // 处理 Upgrade 或者 CONNECT 请求
8.     const req = parser.incoming;
9.     const eventName = req.method === 'CONNECT' ?
10.         'connect' : 'upgrade';
11.     // 监听了对应的事件则处理, 否则关闭连接
12.     if (eventName === 'upgrade' ||
13.         server.listenerCount(eventName) > 0) {
14.         // 还没有解析的数据
15.         const bodyHead = d.slice(ret, d.length);
16.         socket.readableFlowing = null;
17.         server.emit(eventName, req, socket, bodyHead);
18.     } else {
19.         socket.destroy();
20.     }
21. }
22. }

```

这时候 Node.js 会判断请求是不是 Connect 或者协议升级的 upgrade 请求，是的话继续判断是否有处理该事件的函数，没有则关闭连接，否则触发对应的事件进行处理。所以这时候 Node.js 会触发 Connect 方法。Connect 事件的处理逻辑正如我们开始给出的例子中那样。我们首先和真正的服务器建立 TCP 连接，然后返回响应头给客户端，后续客户就可以和真正的服务器真正进行 TLS 握手和 HTTPS 通信了。这就是 Node.js 中 Connect 的原理和实现。

不过在代码中我们发现一个好玩的地方。那就是在触发 connect 事件的时候，Node.js 给回调函数传入的参数。

```
1. server.emit('connect', req, socket, bodyHead);
```

第一第二个参数没什么特别的，但是第三个参数就有意思了，bodyHead 代表的是 HTTP Connect 请求中除了请求行和 HTTP 头之外的数据。因为 Node.js 解析完 HTTP 头后就不继续处理了。把剩下的数据交给了用户。我们来做一些好玩的事情。

```

1. const http = require('http');
2. const net = require('net');
3. const { URL } = require('url');
4.
5. const proxy = http.createServer((req, res) => {
6.     res.writeHead(200, { 'Content-Type': 'text/plain' });
7.     res.end('okay');
8. });
9. proxy.on('connect', (req, clientSocket, head) => {
10.     const { port, hostname } = new URL(`http://${req.url}`);

```

```
11. const serverSocket = net.connect(port || 80, hostname, () => {
12.   clientSocket.write('HTTP/1.1 200 Connection Established\r\n'
+                         'Proxy-agent: Node.js-Proxy\r\n' +
14.                         '\r\n');
15.   // 把 connect 请求剩下的数据转发给服务器
16.   serverSocket.write(head);
17.   serverSocket.pipe(clientSocket);
18.   clientSocket.pipe(serverSocket);
19. });
20. });
21.
22. proxy.listen(1337, '127.0.0.1', () => {
23.   const net = require('net');
24.   const body = 'GET http://www.baidu.com:80 HTTP/1.1\r\n\r\n';
25.   const length = body.length;
26.   const socket = net.connect({host: '127.0.0.1', port: 1337});
27.   socket.write(`CONNECT www.baidu.com:80 HTTP/1.1\r\n\r\n${body}`);
28.   socket.setEncoding('utf-8');
29.   socket.on('data', (chunk) => {
30.     console.log(chunk)
31.   });
32. });
```

我们新建一个 socket，然后自己构造 HTTP Connect 报文，并且在 HTTP 行后面加一个额外的字符串，这个字符串是两一个 HTTP 请求。当 Node. js 服务器收到 Connect 请求后，我们在 connect 事件的处理函数中，把 Connect 请求多余的那一部分数据传给真正的服务器。这样就节省了发送一个请求的时间。

18.3.3 超时管理

在解析 HTTP 协议或者支持长连接的时候，Node. js 需要设置一些超时的机制，否则会造成攻击或者资源浪费。下面我们看一下 HTTP 服务器中涉及到超时的一些逻辑。

1 解析 HTTP 头部超时

当收到一个 HTTP 请求报文时，会从 HTTP 请求行，HTTP 头，HTTP body 的顺序进行解析，如果用户构造请求，只发送 HTTP 头的一部分。那么 HTTP 解析器就会一直在等待后续数据的到来。这会导致 DDOS 攻击，所以 Node. js 中设置了解析 HTTP 头的超时时间，阈值是 60 秒。如果 60 秒内没有解析完 HTTP 头部，则会触发 timeout 事件。如果用户不处理，则 Node. js 会自动关闭连接。我们看一下 Node. js 的实现。Node. js 在初始化的时候会设置超时时间。

```
1. this.headersTimeout = 60 * 1000; // 60 seconds
```

Node.js 在建立 TCP 连接成功后初始化解析 HTTP 头的开始时间。

```
1. function connectionListenerInternal(server, socket) {
2.   parser.parsingHeadersStart = nowDate();
3. }
```

然后在每次收到数据的时候判断 HTTP 头部是否解析完成，如果没有解析完成并且超时了则会触发 timeout 事件。

```
1. function onParserExecute(server, socket, parser, state, ret) {
2.   socket._unrefTimer();
3.   const start = parser.parsingHeadersStart;
4.   // start 等于 0，说明 HTTP 头已经解析完毕，否则说明正在解析头，然后再判断解析时间是否超时了
5.   if (start !== 0 && nowDate() - start > server.headersTimeout) {
6.     // 触发 timeout，如果没有监听 timeout，则默认会销毁 socket，即关闭连接
7.     const serverTimeout = server.emit('timeout', socket);
8.
9.     if (!serverTimeout)
10.       socket.destroy();
11.     return;
12.   }
13.
14. onParserExecuteCommon(server, socket, parser, state, ret, undefined);
15. }
```

如果在超时之前解析 HTTP 头完成，则把 parsingHeadersStart 置为 0 表示解析完成。

```
1. function parserOnIncoming(server, socket, state, req, keepAlive) {
2.   // 设置了 keepAlive 则响应后需要重置一些状态
3.   if (server.keepAliveTimeout > 0) {
4.     req.on('end', resetHeadersTimeoutOnReqEnd);
5.   }
6.
7.   // 标记头部解析完毕
8.   socket.parser.parsingHeadersStart = 0;
9. }
10.
11. function resetHeadersTimeoutOnReqEnd() {
12.   if (parser) {
13.     parser.parsingHeadersStart = nowDate();
```

```
14. }
15. }
```

另外如果支持长连接，即一个 TCP 连接上可以发送多个请求。则在每个响应结束之后，需要重新初始化解析 HTTP 头的开始时间。当下一个请求数据到来时再次判断解析 HTTP 头部是否超时。这里是响应结束后就开始计算。而不是下一个请求到来时。

2 支持管道化的情况下，多个请求的时间间隔

Node.js 支持在一个 TCP 连接上发送多个 HTTP 请求，所以需要设置一个定时器，如果超时都没有新的请求到来，则触发超时事件。这里涉及定时器的设置和重置。

```
1. // 是不是最后一个响应
2. if (res._last) {
3.     // 是则销毁 socket
4.     if (typeof socket.destroySoon === 'function') {
5.         socket.destroySoon();
6.     } else {
7.         socket.end();
8.     }
9. } else if (state.outgoing.length === 0) {
10.    // 没有待处理的响应了，则重新设置超时时间，等待请求的到来，一定时间内没有请求则触发 timeout 事件
11.    if (server.keepAliveTimeout && typeof socket.setTimeout ===
12.        'function') {
13.        socket.setTimeout(server.keepAliveTimeout);
14.        state.keepAliveTimeoutSet = true;
15.    }
}
```

每次响应结束的时候，Node.js 首先会判断当前响应是不是最后一个，例如读端不可读了，说明不会再有请求到来了，也不会有响应了，那么就不需要保持这个 TCP 连接。如果当前响应不是最后一个，则 Node.js 会根据 `keepAliveTimeout` 的值做下一步判断，如果 `keepAliveTimeout` 非空，则设置定时器，如果 `keepAliveTimeout` 时间内都没有新的请求则触发 `timeout` 事件。那么如果有新请求到来，则需要重置这个定时器。Node.js 在收到新请求的第一个请求包中，重置该定时器。

```
1. function onParserExecuteCommon(server, socket, parser, state, ret
   , d) {
2.     resetSocketTimeout(server, socket, state);
3. }
4.
5. function resetSocketTimeout(server, socket, state) {
6.     if (!state.keepAliveTimeoutSet)
7.         return;
8.
```

```

9.   socket.setTimeout(server.timeout || 0);
10.  state.keepAliveTimeoutSet = false;
11. }

```

`onParserExecuteCommon` 会在每次收到数据时执行，然后 `Node.js` 会重置定时器为 `server.timeout` 的值。

18.4 Agent

本节我们先分析 `Agent` 模块的实现，`Agent` 对 TCP 连接进行了池化管理。简单的情况下，客户端发送一个 HTTP 请求之前，首先建立一个 TCP 连接，收到响应后会立刻关闭 TCP 连接。但是我们知道 TCP 的三次握手是比较耗时的。所以如果我们能复用 TCP 连接，在一个 TCP 连接上发送多个 HTTP 请求和接收多个 HTTP 响应，那么在性能上面就会得到很大的提升。`Agent` 的作用就是复用 TCP 连接。不过 `Agent` 的模式是在一个 TCP 连接上串行地发送请求和接收响应，不支持 HTTP PipeLine 模式。下面我们看一下 `Agent` 模块的具体实现。看它是如何实现 TCP 连接复用的。

```

1. function Agent(options) {
2.   if (!(this instanceof Agent))
3.     return new Agent(options);
4.   EventEmitter.call(this);
5.   this.defaultPort = 80;
6.   this.protocol = 'http:';
7.   this.options = { ...options };
8.   // path 字段表示是本机的进程间通信时使用的路径，比如 Unix 域路径
9.   this.options.path = null;
10.  // socket 个数达到阈值后，等待空闲 socket 的请求
11.  this.requests = {};
12.  // 正在使用的 socket
13.  this.sockets = {};
14.  // 空闲 socket
15.  this.freeSockets = {};
16.  // 空闲 socket 的存活时间
17.  this.keepAliveMsecs = this.options.keepAliveMsecs || 1000;
18.  /*
19.    用完的 socket 是否放到空闲队列，
20.    开启 keepalive 才会放到空闲队列,
21.    不开启 keepalive
22.    还有等待 socket 的请求则复用 socket
23.    没有等待 socket 的请求则直接销毁 socket
24.  */
25.  this.keepAlive = this.options.keepAlive || false;
26.  // 最大的 socket 个数，包括正在使用的和空闲的 socket

```

```
27. this.maxSockets = this.options.maxSockets  
28.           || Agent.defaultMaxSockets;  
29. // 最大的空闲 socket 个数  
30. this.maxFreeSockets = this.options.maxFreeSockets || 256;  
31. }
```

Agent 维护了几个数据结构，分别是等待 socket 的请求、正在使用的 socket、空闲 socket。每一个数据结构是一个对象，对象的 key 是根据 HTTP 请求参数计算的。对象的值是一个队列。具体结构如图 18-7 所示。

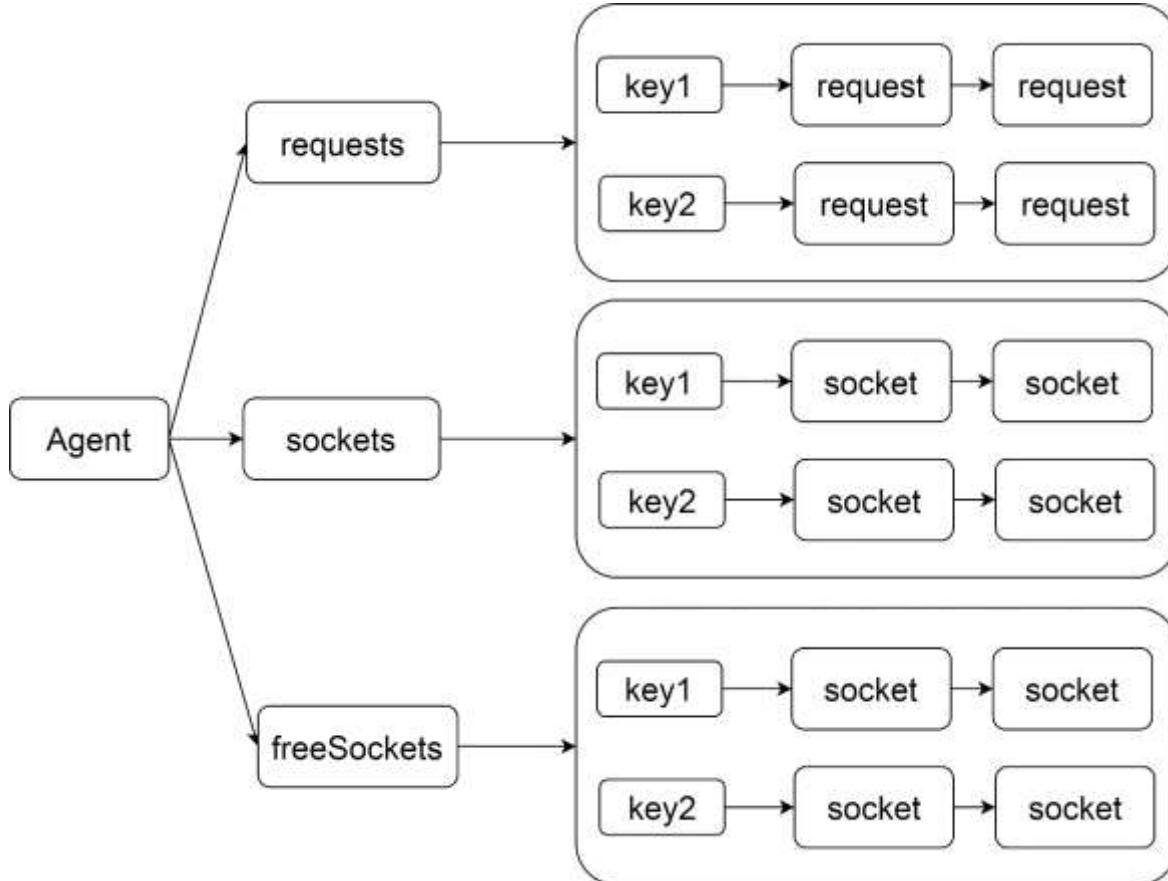


图 18-7

下面我们看一下 Agent 模块的具体实现。

18.4.1 key 的计算

key 的计算是池化管理的核心。正确地设计 key 的计算规则，才能更好地利用池化带来的好处。

```
1. // 一个请求对应的 key  
2. Agent.prototype.getName = function getName(options) {  
3.   let name = options.host || 'localhost';  
4.   name += ':';
```

```

5. if (options.port)
6.   name += options.port;
7. name += ':';
8. if (options.localAddress)
9.   name += options.localAddress;
10. if (options.family === 4 || options.family === 6)
11.   name += `:${options.family}`;
12. if (options.socketPath)
13.   name += `:${options.socketPath}`;
14. return name;
15. };

```

我们看到 key 由 host、port、本地地址、地址簇类型、unix 路径计算而来。所以不同的请求只有这些因子都一样的情况下才能复用连接。另外我们看到 Agent 支持 Unix 域。

18.4.2 创建一个 socket

```

1. function createSocket(req, options, cb) {
2.   options = { ...options, ...this.options };
3.   // 计算 key
4.   const name = this.getName(options);
5.   options._agentKey = name;
6.   options.encoding = null;
7.   let called = false;
8.   // 创建 socket 完毕后执行的回调
9.   const oncreate = (err, s) => {
10.     if (called)
11.       return;
12.     called = true;
13.     if (err)
14.       return cb(err);
15.     if (!this.sockets[name])
16.       this.sockets[name] = [];
17.   }
18.   // 插入正在使用的 socket 队列
19.   this.sockets[name].push(s);
20.   // 监听 socket 的一些事件，用于回收 socket
21.   installListeners(this, s, options);
22.   // 有可用 socket，通知调用方
23.   cb(null, s);
24. };
25. // 创建一个新的 socket，使用 net.createConnection
26. const newSocket = this.createConnection(options, oncreate);
27. if (newSocket)
28.   oncreate(null, newSocket);

```

```
29. }
30.
31. function installListeners(agent, s, options) {
32.   /*
33.     socket 触发空闲事件的处理函数，告诉 agent 该 socket 空闲了，
34.     agent 会回收该 socket 到空闲队列
35.   */
36.   function onFree() {
37.     agent.emit('free', s, options);
38.   }
39.   /*
40.     监听 socket 空闲事件，调用方使用完 socket 后触发，
41.     通知 agent socket 用完了
42.   */
43.   s.on('free', onFree);
44.
45.   function onClose(err) {
46.     agent.removeSocket(s, options);
47.   }
48.   // socket 关闭则 agent 会从 socket 队列中删除它
49.   s.on('close', onClose);
50.
51.   function onRemove() {
52.     agent.removeSocket(s, options);
53.     s.removeListener('close', onClose);
54.     s.removeListener('free', onFree);
55.     s.removeListener('agentRemove', onRemove);
56.   }
57.   // agent 被移除
58.   s.on('agentRemove', onRemove);
59.
60. }
```

创建 socket 的主要逻辑如下

- 1 调用 net 模块创建一个 socket (TCP 或者 Unix 域)，然后插入使用中的 socket 队列，最后通知调用方 socket 创建成功。
- 2 监听 socket 的 close、free 事件和 agentRemove 事件，触发时从队列中删除 socket。

18.4.3 删除 socket

```
1. // 把 socket 从正在使用队列或者空闲队列中移出
2. function removeSocket(s, options) {
3.   const name = this.getName(options);
4.   const sets = [this.sockets];
5.   /*
```

```

6.     socket 不可写了，则有可能是存在空闲的队列中，  

7.     所以需要遍历空闲队列，因为 removeSocket 只会在  

8.     使用完 socket 或者 socket 关闭的时候被调用，前者只有在  

9.     可写状态时会调用，后者是不可写的  

10.    */  

11.    if (!s.writable)  

12.        sets.push(this.freeSockets);  

13.    // 从队列中删除对应的 socket  

14.    for (const sockets of sets) {  

15.        if (sockets[name]) {  

16.            const index = sockets[name].indexOf(s);  

17.            if (index !== -1) {  

18.                sockets[name].splice(index, 1);  

19.                // Don't leak  

20.                if (sockets[name].length === 0)  

21.                    delete sockets[name];  

22.            }  

23.        }  

24.    }  

25.    /*  

26.     如果还有在等待 socket 的请求，则创建 socket 去处理它，  

27.     因为 socket 数已经减一了，说明 socket 个数还没有达到阈值  

28.     但是这里应该先判断是否还有空闲的 socket，有则可以复用，  

29.     没有则创建新的 socket  

30.    */  

31.    if (this.requests[name] && this.requests[name].length) {  

32.        const req = this.requests[name][0];  

33.        const socketCreationHandler = handleSocketCreation(this,  

34.                                                req,  

35.                                                false);  

36.        this.createSocket(req, options, socketCreationHandler);  

37.    }  

38. };

```

前面已经分析过，Agent 维护了两个 socket 队列，删除 socket 就是从这两个队列中找到对应的 socket，然后移除它。移除后需要判断一下是否还有等待 socket 的请求队列，有的话就新建一个 socket 去处理它。因为移除了一个 socket，就说明可以新增一个 socket。

18.4.4 设置 socket keepalive

当 socket 被使用完并且被插入空闲队列后，需要重新设置 socket 的 keepalive 值。等到超时会自动关闭 socket。在一个 socket 上调用一次 `setKeepAlive` 就可以了，这里可能会导致多次调用 `setKeepAlive`，不过也没有影响。

```
1. function keepSocketAlive(socket) {  
2.   socket.setKeepAlive(true, this.keepAliveMsecs);  
3.   socket.unref();  
4.   return true;  
5. };
```

另外需要设置 ref 标记，防止该 socket 阻止事件循环的退出，因为该 socket 是空闲的，不应该影响事件循环的退出。

18.4.5 复用 socket

```
1. function reuseSocket(socket, req) {  
2.   req.reusedSocket = true;  
3.   socket.ref();  
4. };
```

重新使用该 socket，需要修改 ref 标记，阻止事件循环退出，并标记请求使用的是复用 socket。

18.4.6 销毁 Agent

```
1. function destroy() {  
2.   for (const set of [this.freeSockets, this.sockets]) {  
3.     for (const key of ObjectKeys(set)) {  
4.       for (const setName of set[key]) {  
5.         setName.destroy();  
6.       }  
7.     }  
8.   }  
9. };
```

因为 Agent 本质上是一个 socket 池，销毁 Agent 即销毁池里维护的所有 socket。

18.4.7 使用连接池

我们看一下如何使用 Agent。

```
1. function addRequest(req, options, port, localAddress) {  
2.   // 参数处理  
3.   if (typeof options === 'string') {  
4.     options = {  
5.       host: options,  
6.       port,  
7.       localAddress
```

```

8.      };
9.    }
10.
11.   options = { ...options, ...this.options };
12.   if (options.socketPath)
13.     options.path = options.socketPath;
14.
15.   if (!options.servername && options.servername !== '')
16.     options.servername = calculateServerName(options, req);
17.   // 拿到请求对应的 key
18.   const name = this.getName(options);
19.   // 该 key 还没有在使用的 socket 则初始化数据结构
20.   if (!this.sockets[name]) {
21.     this.sockets[name] = [];
22.   }
23.   // 该 key 对应的空闲 socket 列表
24.   const freeLen = this.freeSockets[name] ?
25.                 this.freeSockets[name].length : 0;
26.   // 该 key 对应的所有 socket 个数
27.   const sockLen = freeLen + this.sockets[name].length;
28.   // 该 key 有对应的空闲 socket
29.   if (freeLen) {
30.     // 获取一个该 key 对应的空闲 socket
31.     const socket = this.freeSockets[name].shift();
32.     // 取完了删除，防止内存泄漏
33.     if (!this.freeSockets[name].length)
34.       delete this.freeSockets[name];
35.     // 设置 ref 标记，因为正在使用该 socket
36.     this.reuseSocket(socket, req);
37.     // 设置请求对应的 socket
38.     setRequestSocket(this, req, socket);
39.     // 插入正在使用的 socket 队列
40.     this.sockets[name].push(socket);
41.   } else if (sockLen < this.maxSockets) {
42.     /*
43.       如果该 key 没有对应的空闲 socket 并且使用的
44.       socket 个数还没有得到阈值，则继续创建
45.     */
46.     this.createSocket(req,
47.                       options,
48.                       handleSocketCreation(this, req, true));
49.   } else {
50.     // 等待该 key 下有空闲的 socket
51.     if (!this.requests[name]) {
52.       this.requests[name] = [];

```

```
53.     }
54.     this.requests[name].push(req);
55.   }
56. }
```

当我们需要发送一个 HTTP 请求的时候，我们可以通过 Agent 的 addRequest 方法把请求托管到 Agent 中，当有可用的 socket 时，Agent 会通知我们。addRequest 的代码很长，主要分为三种情况。

1 有空闲 socket，则直接复用，并插入正在使用的 socket 队列中

我们主要看一下 setRequestSocket 函数

```
1. function setRequestSocket(agent, req, socket) {
2.   // 通知请求 socket 创建成功
3.   req.onSocket(socket);
4.   const agentTimeout = agent.options.timeout || 0;
5.   if (req.timeout === undefined || req.timeout === agentTimeout)
6.   {
7.     return;
8.   }
9.   // 开启一个定时器，过期后触发 timeout 事件
10.  socket.setTimeout(req.timeout);
11.  /*
12.    监听响应事件，响应结束后需要重新设置超时时间，
13.    开启下一个请求的超时计算，否则会提前过期
14.  */
15.  req.once('response', (res) => {
16.    res.once('end', () => {
17.      if (socket.timeout !== agentTimeout) {
18.        socket.setTimeout(agentTimeout);
19.      }
20.    });
21.  });
22. }
```

setRequestSocket 函数通过 `req.onSocket(socket)` 通知调用方有可用 socket。然后如果请求设置了超时时间则设置 socket 的超时时间，即请求的超时时间。最后监听响应结束事件，重新设置超时时间。

2 没有空闲 socket，但是使用的 socket 个数还没有达到阈值，则创建新的 socket。

我们主要分析创建 socket 后的回调 handleSocketCreation。

```
1. function handleSocketCreation(agent, request, informRequest) {
2.   return function handleSocketCreation_Inner(err, socket) {
3.     if (err) {
4.       process.nextTick(emitterErrorNT, request, err);
5.     }
6.   };
7. }
```

```

6.    }
7.    /*
8.     是否需要直接通知请求方，这时候 request 不是来自等待
9.     socket 的 requests 队列，而是来自调用方，见 addRequest
10.    */
11.   if (informRequest)
12.     setRequestSocket(agent, request, socket);
13.   else
14.     /*
15.      不直接通知，先告诉 agent 有空闲的 socket，
16.      agent 会判断是否有正在等待 socket 的请求，有则处理
17.      */
18.     socket.emit('free');
19.   };
20. }

```

3 不满足 1, 2，则把请求插入等待 socket 队列。

插入等待 socket 队列后，当有 socket 空闲时会触发 free 事件，我们看一下该事件的处理逻辑。

```

1. // 监听 socket 空闲事件
2. this.on('free', (socket, options) => {
3.   const name = this.getName(options);
4.   // socket 还可写并且还有等待 socket 的请求，则复用 socket
5.   if (socket.writable &&
6.       this.requests[name] && this.requests[name].length) {
7.     // 拿到一个等待 socket 的请求，然后通知它有 socket 可用
8.     const req = this.requests[name].shift();
9.     setRequestSocket(this, req, socket);
10.    // 没有等待 socket 的请求则删除，防止内存泄漏
11.    if (this.requests[name].length === 0) {
12.      // don't leak
13.      delete this.requests[name];
14.    }
15.  } else {
16.    // socket 不可用写或者没有等待 socket 的请求了
17.    const req = socket._httpMessage;
18.    // socket 可写并且请求设置了允许使用复用的 socket
19.    if (req &&
20.        req.shouldKeepAlive &&
21.        socket.writable &&
22.        this.keepAlive) {
23.      let freeSockets = this.freeSockets[name];
24.      // 该 key 下当前的空闲 socket 个数
25.      const freeLen = freeSockets ? freeSockets.length : 0;

```

```
26.     let count = freeLen;
27.     // 正在使用的 socket 个数
28.     if (this.sockets[name])
29.         count += this.sockets[name].length;
30.     /*
31.         该 key 使用的 socket 个数达到阈值或者空闲 socket 达到阈值,
32.         则不复用 socket, 直接销毁 socket
33.     */
34.     if (count > this.maxSockets ||
35.         freeLen >= this.maxFreeSockets) {
35.         socket.destroy();
36.     } else if (this.keepSocketAlive(socket)) {
37.         /*
38.             重新设置 socket 的存活时间, 设置失败说明无法重新设置存活时
39.             间, 则说明可能不支持复用
40.         */
41.         freeSockets = freeSockets || [];
42.         this.freeSockets[name] = freeSockets;
43.         socket[async_id_symbol] = -1;
44.         socket._httpMessage = null;
45.         // 把 socket 从正在使用队列中移除
46.         this.removeSocket(socket, options);
47.         // 插入 socket 空闲队列
48.         freeSockets.push(socket);
49.     } else {
50.         // 不复用则直接销毁
51.         socket.destroy();
52.     }
53. } else {
54.     socket.destroy();
55. }
56. }
57. );
```

当有 socket 空闲时, 分为以下几种情况

- 1 如果有等待 socket 的请求, 则直接复用 socket。
- 2 如果没有等待 socket 的请求, 允许复用并且 socket 个数没有达到阈值则插入空闲队列。
- 3 直接销毁

18.4.8 测试例子

客户端

```
1. const http = require('http');
```

```

2. const keepAliveAgent = new http.Agent({ keepAlive: true, maxSockets: 1 });
3. const options = {port: 10000, method: 'GET', host: '127.0.0.1',}
4. options.agent = keepAliveAgent;
5. http.get(options, () => {});
6. http.get(options, () => {});
7. console.log(options.agent.requests)

```

服务器

```

1. let i =0;
2. const net = require('net');
3. net.createServer((socket) => {
4.   console.log(++i);
5. }).listen(10000);

```

在例子中，首先创建了一个 tcp 服务器。然后在客户端使用 agent。但是 maxSocket 的值为 1，代表最多只能有一个 socket，而这时候客户端发送两个请求，所以有一个请求就会在排队。服务器也只收到了一个连接。

第十九章 模块加载

Node.js 的模块分为用户 JS 模块、Node.js 原生 JS 模块、Node.js 内置 C++ 模块。本章介绍这些模块加载的原理以及 Node.js 中模块加载器的类型和原理。

下面我们以一个例子为开始，分析 Node.js 中模块加载的原理。假设我们有一个文件 demo.js，代码如下

```

1. const myjs= require('myjs');
2. const net = require('net');

```

其中 myjs 的代码如下

```

1. exports.hello = 'world';

```

我们看一下执行 node demo.js 的时候，过程是怎样的。在 Node.js 启动章节我们分析过，Node.js 启动的时候，会执行以下代码。

```
require('internal/modules/cjs/loader').Module.runMain(process.argv[1])
```

其中 runMain 函数在 pre_execution.js 的 initializeCJSLoader 中挂载

```

1. function initializeCJSLoader() {
2.   const CJSLoader = require('internal/modules/cjs/loader');
3.   CJSLoader.Module._initPaths();
4.   CJSLoader.Module.runMain =
5.     require('internal/modules/run_main').executeUserEntryPoint;
6. }

```

我们看到 runMain 是 run_main.js 导出的函数。继续往下看

```
1. const CJSLoader = require('internal/modules/cjs/loader');
2. const { Module } = CJSLoader;
3. function executeUserEntryPoint(main = process.argv[1]) {
4.   const resolvedMain = resolveMainPath(main);
5.   const useESMLoader = shouldUseESMLoader(resolvedMain);
6.   if (useESMLoader) {
7.     runMainESM(resolvedMain || main);
8.   } else {
9.     Module._load(main, null, true);
10. }
11.
12.
13. module.exports = {
14.   executeUserEntryPoint
15. };
```

process.argv[1]就是我们要执行的 JS 文件。最后通过 cjs/loader.js 的 Module._load 加载了我们的 JS。下面我们看一下具体的处理逻辑。

```
1.   Module._load = function(request, parent, isMain) {
2.     const filename = Module._resolveFilename(request, parent, isMain);
3.
4.     const cachedModule = Module._cache[filename];
5.     // 有缓存则直接返回
6.     if (cachedModule !== undefined) {
7.       updateChildren(parent, cachedModule, true);
8.       if (!cachedModule.loaded)
9.         return getExportsForCircularRequire(cachedModule);
10.      return cachedModule.exports;
11.    }
12.    // 是否是可访问的原生 JS 模块，是则返回
13.    const mod = loadNativeModule(filename, request);
14.    if (mod && mod.canBeRequiredByUsers) return mod.exports;
15.    // 非原生 JS 模块，则新建一个 Module 表示加载的模块
16.    const module = new Module(filename, parent);
17.    // 缓存
18.    Module._cache[filename] = module;
19.    // 加载
20.    module.load(filename);
21.    // 调用方拿到的是 module.exports 的值
22.    return module.exports;
23.  };
```

_load 函数主要是三个逻辑

- 1 判断是否有缓存，有则返回。
- 2 没有缓存，则判断是否是原生 JS 模块，是则交给原生模块处理。
- 3 不是原生模块，则新建一个 Module 表示用户的 JS 模块，然后执行 load 函数加载。

这里我们只需要关注 3 的逻辑，在 Node.js 中，用户定义的模块使用 Module 表示。

```

1. function Module(id = '', parent) {
2.   // 模块对应的文件路径
3.   this.id = id;
4.   this.path = path.dirname(id);
5.   // 在模块里使用的 exports 变量
6.   this.exports = {};
7.   this.parent = parent;
8.   // 加入父模块的 children 队列
9.   updateChildren(parent, this, false);
10.  this.filename = null;
11.  // 是否已经加载
12.  this.loaded = false;
13.  this.children = [];
14. }
```

接着看一下 load 函数的逻辑。

```

1.     Module.prototype.load = function(filename) {
2.       this.filename = filename;
3.       // 拓展名
4.       const extension = findLongestRegisteredExtension(filename);
5.       // 根据拓展名使用不同的加载方式
6.       Module._extensions[extension](this, filename);
7.       this.loaded = true;
8.     };
```

Node.js 会根据不同的文件拓展名使用不同的函数处理。

19.1 加载用户模块

在 Node.js 中 _extensions 有三种，分别是 js、json、node。

19.1.1 加载 JSON 模块

加载 JSON 模块是比较简单的

```

1. Module._extensions['.json'] = function(module, filename) {
2.   const content = fs.readFileSync(filename, 'utf8');
3.
4.   try {
5.     module.exports = JSONParse(stripBOM(content));
6.   } catch (err) {
7.     err.message = filename + ': ' + err.message;
8.     throw err;
9.   }
10. };
```

直接读取 JSON 文件的内容，然后解析成对象就行。

19.1.2 加载 JS 模块

```
1. Module._extensions['.js'] = function(module, filename) {
2.   const content = fs.readFileSync(filename, 'utf8');
3.   module._compile(content, filename);
4. }
```

读完文件的内容，然后执行 _compile

```
1. Module.prototype._compile = function(content, filename) {
2.   // 生成一个函数
3.   const compiledWrapper = wrapSafe(filename, content, this);
4.   const dirname = path.dirname(filename);
5.   // require 是对_load 函数的封装
6.   const require = (path) => {
7.     return this.require(path);
8.   };
9.   let result;
10.  // 我们平时使用的 exports 变量
11.  const exports = this.exports;
12.  const thisValue = exports;
13.  // 我们平时使用的 module 变量
14.  const module = this;
15.  // 执行函数
16.  result = compiledWrapper.call(thisValue,
17.                                exports,
18.                                require,
19.                                module,
20.                                filename,
21.                                dirname);
22.  return result;
23. }
```

_compile 里面包括了几个重要的逻辑

1 wrapSafe：包裹我们的代码并生成一个函数

2 require：支持在模块内加载其他模块

3 执行模块代码

我们看一下这三个逻辑。

1 wrapSafe

```
1. function wrapSafe(filename, content, cjsModuleInstance) {
```

```

2.     const wrapper = Module.wrap(content);
3.     return vm.runInThisContext(wrapper, {
4.       filename,
5.       lineOffset: 0,
6.       ...
7.     });
8.   }
9.
10. const wrapper = [
11.   '(function (exports, require, module, __filename, __dirname) { ',
12.   '\n}');
13. ];
14.
15. Module.wrap = function(script) {
16.   return Module.wrapper[0] + script + Module.wrapper[1];
17. };

```

vm.runInThisContext 的第一个参数是”(function() {})”的时候，会返回一个函数。所以执行 Module.wrap 后会返回一个字符串，内容如下

```

2. (function (exports, require, module, __filename, __dirname) {
3.   //
4. });

```

接着我们看一下 require 函数，即我们平时在代码中使用的 require。

2 require

```

1. Module.prototype.require = function(id) {
2.   requireDepth++;
3.   try {
4.     return Module._load(id, this, /* isMain */ false);
5.   } finally {
6.     requireDepth--;
7.   }
8. };

```

require 是对 Module._load 的封装，Module._load 会把模块导出的变量通过 module.exports 属性返回给 require 调用方。因为 Module._load 只会从原生 JS 模块和用户 JS 模块中查找用户需要加载的模块，所以是无法访问 C++ 模块的，访问 C++ 模块可用 process.binding 或 internalBinding。

3 执行代码

我们回到_compile 函数。看一下执行 vm.runInThisContext 返回的函数。

```

compiledWrapper.call(exports,
                      exports,
                      require,
                      module,
                      filename,
                      dirname);

```

相当于执行以下代码

```

1. (function (exports, require, module, __filename, __dirname) {
2.   const myjs= require('myjs');

```

```
3.   const net = require('net');
4. });


```

至此，Node.js 开始执行用户的 JS 代码。刚才我们已经分析过 require 是对 Module._load 的封装，当执行 require 加载用户模块时，又回到了我们正在分析的这个过程。

19.1.3 加载 node 模块

Node 拓展的模块本质上是动态链接库，我们看 require 一个 node 模块的时候的过程。我们从加载 node 模块的源码开始。

```
1. Module._extensions['.node'] = function(module, filename) {
2.   // ...
3.   return process.dlopen(module, path.toNamespacedPath(filename));
4. };


```

直接调了 process.dlopen，该函数在 node.js 里定义。

```
1. const rawMethods = internalBinding('process_methods');
2. process.dlopen = rawMethods.dlopen;


```

找到 process_methods 模块对应的是 node_process_methods.cc。

env->SetMethod(target, "dlopen", binding::DLOpen);

之前说过，Node.js 的拓展模块其实是动态链接库，那么我们先看看一个动态链接库我们是如何使用的。以下是示例代码。

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <dlfcn.h>
4. int main() {
5.   // 打开一个动态链接库，拿到一个 handler
6.   handler = dlopen('xxx.so', RTLD_LAZY);
7.   // 取出动态链接库里的函数 add
8.   add = dlsym(handler, "add");
9.   // 执行
10.  printf("%d", add(1, 1));
11.  dlclose(handler);
12.  return 0;
13. }


```

了解动态链接库的使用，我们继续分析刚才看到的 DLOpen 函数。

```

1. void DL0pen(const FunctionCallbackInfo<Value>& args) {
2.
3.     int32_t flags = DLib::kDefaultFlags;
4.     node::Utf8Value filename(env->isolate(), args[1]); // Cast
5.     env->TryLoadAddon(*filename, flags, [&] (DLib* dlib) {
6.         const bool is_opened = dlib->Open();
7.         node_module* mp = thread_local_modpending;
8.         thread_local_modpending = nullptr;
9.         // 省略部分代码
10.        if (mp->nm_context_register_func != nullptr) {
11.            mp->nm_context_register_func(exports,
12.                                         module,
13.                                         context,
14.                                         mp->nm_priv);
15.        } else if (mp->nm_register_func != nullptr) {
16.            mp->nm_register_func(exports, module, mp->nm_priv);
17.        }
18.        return true;
19.    });
20. }
```

我们看到重点是 TryLoadAddon 函数，该函数的逻辑就是执行它的第三个参数。我们发现第三个参数是一个函数，入参是 DLib 对象。所以我们先看看这个类。

```

1. class DLib {
2. public:
3.     static const int kDefaultFlags = RTLD_LAZY;
4.     DLib(const char* filename, int flags);
5.
6.     bool Open();
7.     void Close();
8.     const std::string filename_;
9.     const int flags_;
10.    std::string errmsg_;
11.    void* handle_;
12.    uv_lib_t lib_;
13. };
```

再看一下实现。

```

1. bool DLib::Open() {
2.     handle_ = dlopen(filename_.c_str(), flags_);
```

```
3.     if (handle_ != nullptr) return true;
4.     errmsg_ = dlerror();
5.     return false;
6. }
```

DLib 就是对动态链接库的一个封装，它封装了动态链接库的文件名和操作。

TryLoadAddon 函数首先根据 require 传入的文件名，构造一个 DLib，然后执行

```
const bool is_opened = dlib->Open();
```

Open 函数打开了一个动态链接库，这时候我们要先了解一下打开一个动态链接库究竟发生了什么。首先我们一般 C++ 插件最后一句代码的定义。

```
NAPI_MODULE(NODE_GYP_MODULE_NAME, init)
```

这是个宏定义。

```
1. #define NAPI_MODULE(modname, regfunc) \
2.   NAPI_MODULE_X(modname, regfunc, NULL, 0)
3. #define NAPI_MODULE_X(modname, regfunc, priv, flags) \
4.   static napi_module _module = \
5.   { \
6.     NAPI_MODULE_VERSION, \
7.     flags, \
8.     __FILE__, \
9.     regfunc, \
10.    #modname, \
11.    priv, \
12.    {0}, \
13.  }; \
14.  static void _register_modname(void) __attribute__((constructor)); \
15.  static void _register_modname(void) { \
16.    napi_module_register(&_module); \
17. }
```

所以一个 node 扩展就是定义了一个 napi_module 模块和一个 register_modname (modname 是我们定义的) 函数。`_attribute((constructor))` 是代表该函数会先执行的意思，具体可以查阅文档。看到这里我们知道，当我们打开一个动态链接库的时候，会执行 register_modname 函数，该函数执行的是

```
napi_module_register(&_module);
```

我们继续展开。

```
1.
2. // Registers a NAPI module.
3. void napi_module_register(napi_module* mod) {
```

```

4.     node::node_module* nm = new node::node_module {
5.         -1,
6.         mod->nm_flags | NM_F_DELETEME,
7.         nullptr,
8.         mod->nm_filename,
9.         nullptr,
10.        napi_module_register_cb,
11.        mod->nm_modname,
12.        mod, // priv
13.        nullptr,
14.    };
15.    node::node_module_register(nm);
16. }

```

Node.js 把 napi 模块转成 node_module。最后调用 node_module_register。

```

1.
2. extern "C" void node_module_register(void* m) {
3.     struct node_module* mp = reinterpret_cast<struct node_module*>(
4.         m);
5.     if (mp->nm_flags & NM_F_INTERNAL) {
6.         mp->nm_link = modlist_internal;
7.         modlist_internal = mp;
8.     } else if (!node_is_initialized) {
9.         mp->nm_flags = NM_F_LINKED;
10.        mp->nm_link = modlist_linked;
11.        modlist_linked = mp;
12.    } else {
13.        thread_local_modpending = mp;
14.    }
15. }

```

napi 模块不是 NM_F_INTERNAL 模块，node_is_initialized 是在 Node.js 初始化时设置的变量，这时候已经是 `true`。所以注册 napi 模块时，会执行 `thread_local_modpending = mp`。`thread_local_modpending` 类似一个全局变量，保存当前加载的模块。分析到这，我们回到 DLOpen 函数。

```

1. node_module* mp = thread_local_modpending;
2. thread_local_modpending = nullptr;

```

这时候我们就知道刚才那个变量 thread_local_modpending 的作用了。

node_module* mp = thread_local_modpending 后我们拿到了我们刚才定义的 napi 模块的信息。接着执行 node_module 的函数 nm_register_func。

```
1. if (mp->nm_context_register_func != nullptr) {
2.     mp->nm_context_register_func(exports,
3.                                     module,
4.                                     context,
5.                                     mp->nm_priv);
6. } else if (mp->nm_register_func != nullptr) {
7.     mp->nm_register_func(exports, module, mp->nm_priv);
8. }
```

从刚才的 node_module 定义中我们看到函数是 napi_module_register_cb。

```
1. static void napi_module_register_cb(v8::Local<v8::Object> exports,
2.                                     v8::Local<v8::Value> module,
3.                                     v8::Local<v8::Context> context,
4.                                     void * priv) {
5.     napi_module_register_by_symbol(exports, module, context,
6.                                     static_cast<napi_module*>(priv)->nm_register_func);
7. }
```

该函数调用 napi_module_register_by_symbol 函数，并传入 napi_module 的 nm_register_func 函数。

```
1. void napi_module_register_by_symbol(v8::Local<v8::Object> exports,
2.                                     v8::Local<v8::Value> module,
3.                                     v8::Local<v8::Context> context,
4.                                     napi_addon_register_func init) {
5.
6.     // Create a new napi_env for this specific module.
7.     napi_env env = v8impl::NewEnv(context);
8.
9.     napi_value _exports;
```

```

10.     env->CallIntoModuleThrow([&](napi_env env) {
11.         _exports = init(env, v8impl::JsValueFromV8LocalValue(exports)
12.     );
13.
14.     if (_exports != nullptr &&
15.         _exports != v8impl::JsValueFromV8LocalValue(exports)) {
16.         napi_value _module = v8impl::JsValueFromV8LocalValue(module);
17.
18.         napi_set_named_property(env, _module, "exports", _exports);
19.     }

```

init 就是我们定义的函数。入参是 env 和 exports，可以对比我们定义的函数的入参。最后我们修改 exports 变量。即设置导出的内容。最后在 JS 里，我们就拿到了 C++ 层定义的内容。

19.2 加载原生 JS 模块

上一节我们了解了 Node.js 执行 node demo.js 的过程，其中我们在 demo.js 中使用 require 加载 net 模块。net 是原生 JS 模块。这时候就会进入原生模块的处理逻辑。原生模块是 Node.js 内部实现的 JS 模块。使用 NativeModule 来表示。

```

1.  class NativeModule {
2.      // 原生 JS 模块的 map
3.      static map = new Map(moduleIds.map((id) => [id, new NativeModule(id)]));
4.
5.      constructor(id) {
6.          this.filename = `${id}.js`;
7.          this.id = id;
8.          this.canBeRequiredByUsers = !id.startsWith('internal/');
9.          this.exports = {};
10.         this.loaded = false;
11.         this.loading = false;
12.         this.module = undefined;
13.         this.exportKeys = undefined;
14.     }
15. }

```

当我们执行 require('net') 时，就会进入_load 函数。_load 函数判断要加载的模块是原生 JS 模块后，会通过 loadNativeModule 函数加载原生 JS 模块。我们看这个函数的定义。

```

1. function loadNativeModule(filename, request) {
2.     const mod = NativeModule.map.get(filename);

```

```
3.     if (mod) {
4.         mod.compileForPublicLoader();
5.         return mod;
6.     }
7. }
```

在 Node.js 启动过程中我们分析过，mod 是一个 NativeModule 对象，接着看 compileForPublicLoader。

```
1. compileForPublicLoader() {
2.     this.compileForInternalLoader();
3.     return this.exports;
4. }
5.
6. compileForInternalLoader() {
7.     if (this.loaded || this.loading) {
8.         return this.exports;
9.     }
10.    // id 就是我们要加载的模块，比如 net
11.    const id = this.id;
12.    this.loading = true;
13.    try {
14.        const fn = compileFunction(id);
15.        fn(this.exports,
16.            // 加载原生 JS 模块的加载器
17.            nativeModuleRequire,
18.            this,
19.            process,
20.            // 加载 C++ 模块的加载器
21.            internalBinding,
22.            primordials);
23.        this.loaded = true;
24.    } finally {
25.        this.loading = false;
26.    }
27.    return this.exports;
28. }
```

我们重点看 compileFunction 这里的逻辑。该函数是 node_native_module_env.cc 模块导出的函数。具体的代码就不贴了，通过层层查找，最后到 node_native_module.cc 的 NativeModuleLoader::CompileAsModule

```

1. MaybeLocal<Function> NativeModuleLoader::CompileAsModule(
2.     Local<Context> context,
3.     const char* id,
4.     NativeModuleLoader::Result* result) {
5.
6.     Isolate* isolate = context->GetIsolate();
7.     // 函数的形参
8.     std::vector<Local<String>> parameters = {
9.         FIXED_ONE_BYTE_STRING(isolate, "exports"),
10.        FIXED_ONE_BYTE_STRING(isolate, "require"),
11.        FIXED_ONE_BYTE_STRING(isolate, "module"),
12.        FIXED_ONE_BYTE_STRING(isolate, "process"),
13.        FIXED_ONE_BYTE_STRING(isolate, "internalBinding"),
14.        FIXED_ONE_BYTE_STRING(isolate, "primordials")};
15.     // 编译出一个函数
16.     return LookupAndCompile(context, id, &parameters, result);
17. }
```

我们继续看 `LookupAndCompile`。

```

1. MaybeLocal<Function> NativeModuleLoader::LookupAndCompile(
2.     Local<Context> context,
3.     const char* id,
4.     std::vector<Local<String>>* parameters,
5.     NativeModuleLoader::Result* result) {
6.
7.     Isolate* isolate = context->GetIsolate();
8.     EscapableHandleScope scope(isolate);
9.
10.    Local<String> source;
11.    // 找到原生 JS 模块内容所在的内存地址
12.    if (!LoadBuiltinModuleSource(isolate, id).ToLocal(&source)) {
13.        return {};
14.    }
15.    // 'net' + '.js'
16.    std::string filename_s = id + std::string(".js");
17.    Local<String> filename =
18.        OneByteString(isolate,
19.                      filename_s.c_str(),
20.                      filename_s.size());
21.    // 省略一些参数处理
22.    // 脚本源码
```

```
23.     ScriptCompiler::Source script_source(source, origin, cached_data)
24.     ;
25.     // 编译出一个函数
26.     MaybeLocal<Function> maybe_fun =
27.         ScriptCompiler::CompileFunctionInContext(context,
28.                                         &script_source,
29.                                         parameters->size(),
30.                                         parameters->data(),
31.                                         0,
32.                                         nullptr,
33.                                         options);
34.     Local<Function> fun = maybe_fun.ToLocalChecked();
35.     return scope.Escape(fun);
36. }
```

LookupAndCompile 函数首先找到加载模块的源码，然后编译出一个函数。我们看一下 LoadBuiltinModuleSource 如何查找模块源码的。

```
1. MaybeLocal<String> NativeModuleLoader::LoadBuiltinModuleSource(Isolate*
   isolate, const char* id) {
2.     const auto source_it = source_.find(id);
3.     return source_it->second.ToStringChecked(isolate);
4. }
```

这里是 id 是 net，通过该 id 从_source 中找到对应的数据，那么_source 是什么呢？因为 Node.js 为了提高效率，把原生 JS 模块的源码字符串直接转成 ASCII 码存到内存里。这样加载这些模块的时候，就不需要硬盘 I/O 了。直接从内存读取就行。我们看一下 _source 的定义（在编译 Node.js 源码或者执行 js2c.py 生成的 node_javascript.cc 中）。

```
1. source_.emplace("net", UnionBytes{net_raw, 46682});
2. source_.emplace("cyb", UnionBytes{cyb_raw, 63});
3. source_.emplace("os", UnionBytes{os_raw, 7548});
```

cyb 是我增加的测试模块。我们可以看一下该模块的内容。

```
1. static const uint8_t cyb_raw[] = {
2.     99, 111, 110, 115, 116, 32, 99, 121, 98, 32, 61, 32, 105, 110, 116, 101, 1
   14, 110, 97, 108, 66, 105, 110, 100, 105, 110, 103, 40, 39, 99,
3.     121, 98, 95, 119, 114, 97, 112, 39, 41, 59, 32, 10, 109, 111, 100, 117,
   108, 101, 46, 101, 120, 112, 111, 114, 116, 115, 32, 61, 32, 99,
4.     121, 98, 59
```

```
5. };
```

我们转成字符串看一下是什么

```
1. Buffer.from([99, 111, 110, 115, 116, 32, 99, 121, 98, 32, 61, 32, 105, 11
   0, 116, 101, 114, 110, 97, 108, 66, 105, 110, 100, 105, 110, 103, 40, 39, 99,
2. 121, 98, 95, 119, 114, 97, 112, 39, 41, 59, 32, 10, 109, 111, 100, 117,
   108, 101, 46, 101, 120, 112, 111, 114, 116, 115, 32, 61, 32, 99,
3. 121, 98, 59].join(',')).split(',')).toString('utf-8')
```

输出

```
1. const cyb = internalBinding('cyb_wrap');
2. module.exports = cyb;
```

所以我们执行 `require('net')` 时，通过 NativeModule 的 `compileForInternalLoader`，最终会在 `_source` 中找到 `net` 模块对应的源码字符串，然后编译成一个函数。

```
1. const fn = compileFunction(id);
2. fn(this.exports,
3.   // 加载原生 JS 模块的加载器
4.   nativeModuleRequire,
5.   this,
6.   process,
7.   // 加载 C++ 模块的加载器
8.   internalBinding,
9.   primordials);
```

由 `fn` 的入参可以知道，我们在 `net`（或其它原生 JS 模块中）只能加载原生 JS 模块和内置的 C++ 模块。当 `fn` 执行完毕后，原生模块加载器就会把 `mod.exports` 的值返回给调用方。

19.3 加载内置 C++ 模块

在原生 JS 模块中我们一般会加载一些内置的 C++ 模块，这是 Node.js 拓展 JS 功能的关键之处。比如我们 `require('net')` 的时候，`net` 模块会加载 `tcp_wrap` 模块。

```
1. const {
2.   TCP,
3.   TCPConnectWrap,
4.   constants: TCPConstants
5. } = internalBinding('tcp_wrap')
```

C++模块加载器也是在 internal/bootstrap/loaders.js 中定义的，分为三种。

1 internalBinding: 不暴露给用户的访问的接口，只能在 Node.js 代码中访问，比如原生 JS 模块（flag 为 NM_F_INTERNAL）。

```
1. let internalBinding;
2. {
3.   const bindingObj = ObjectCreate(null);
4.   internalBinding = function internalBinding(module) {
5.     let mod = bindingObj[module];
6.     if (typeof mod !== 'object') {
7.       mod = bindingObj[module] = getInternalBinding(module);
8.       moduleLoadList.push(` Internal Binding ${module}`);
9.     }
10.    return mod;
11.  };
12. }
```

internalBinding 是在 getInternalBinding 函数基础上加了缓存功能。

getInternalBinding 是 C++ 层定义的函数对 JS 暴露的接口名。它的作用是从 C++ 模块链表中找到对应的模块。

2 process.binding: 暴露给用户调用 C++ 模块的接口，但是只能访问部分 C++ 模块（flag 为 NM_F_BUILTIN 的 C++ 模块）。

```
1. process.binding = function binding(module) {
2.   module = String(module);
3.   if (internalBindingWhitelist.has(module)) {
4.     return internalBinding(module);
5.   }
6.   throw new Error(`No such module: ${module}`);
7. };
```

binding 是在 internalBinding 的基础上加了白名单的逻辑，只对外暴露部分模块。

```
1. const internalBindingWhitelist = new SafeSet([
2.   'async_wrap',
3.   'buffer',
4.   'cares_wrap',
5.   'config',
6.   'constants',
7.   'contextify',
8.   'crypto',
9.   'fs',
10.  'fs_event_wrap',
11.  'http_parser',
12.  'icu',
13.  'inspector',
14.  'js_stream',
15.  'natives',
16.  'os',
17.  'pipe_wrap',
18.  'process_wrap',
19.  'signal_wrap',
20.  'spawn_sync',
21.  'stream_wrap',
22.  'tcp_wrap',
23.  'tls_wrap',
```

```

24.     'tty_wrap',
25.     'udp_wrap',
26.     'url',
27.     'util',
28.     'uv',
29.     'v8',
30.     'zlib'
31.   ]);

```

3 process._linkedBinding: 暴露给用户访问 C++ 模块的接口，用于访问用户自己添加的但是没有加到内置模块的 C++ 模块（flag 为 NM_F_LINKED）。

```

1.  const bindingObj = ObjectCreate(null);
2.  process._linkedBinding = function _linkedBinding(module) {
3.    module = String(module);
4.    let mod = bindingObj[module];
5.    if (typeof mod !== 'object')
6.      mod = bindingObj[module] = getLinkedBinding(module);
7.    return mod;
8.  };

```

_linkedBinding 是在 getLinkedBinding 函数基础上加了缓存功能，getLinkedBinding 是 C++ 层定义的函数对外暴露的名字。getLinkedBinding 从另一个 C++ 模块链表中查找对应的模块。

上一节已经分析过，internalBinding 是加载原生 JS 模块时传入的实参。internalBinding 是对 getInternalBinding 的封装。getInternalBinding 对应的是 binding::GetInternalBinding (node_binding.cc)。

```

1. // 根据模块名查找对应的模块
2. void GetInternalBinding(const FunctionCallbackInfo<Value>& args) {
3.   Environment* env = Environment::GetCurrent(args);
4.   // 模块名
5.   Local<String> module = args[0].As<String>();
6.   node::Utf8Value module_v(env->isolate(), module);
7.   Local<Object> exports;
8.   // 从 C++ 内部模块找
9.   node_module* mod = FindModule(modlist_internal,
10.                                 *module_v,
11.                                 NM_F_INTERNAL);
12.   // 找到则初始化
13.   if (mod != nullptr) {
14.     exports = InitModule(env, mod, module);
15.   } else {
16.     // 省略
17.   }
18.

```

```
19.     args.GetReturnValue().Set(exports);  
20. }
```

modlist_internal 是一条链表，在 Node.js 启动过程的时候，由各个 C++ 模块连成的链表。通过模块名找到对应的 C++ 模块后，执行 InitModule 初始化模块。

```
1. // 初始化一个模块，即执行它里面的注册函数  
2. static Local<Object> InitModule(Environment* env,  
3.                                     node_module* mod,  
4.                                     Local<String> module) {  
5.     Local<Object> exports = Object::New(env->isolate());  
6.     Local<Value> unused = Undefined(env->isolate());  
7.     mod->nm_context_register_func(exports, unused, env->context(), mo  
d->nm_priv);  
8.     return exports;  
9. }
```

执行 C++ 模块的 nm_context_register_func 指向的函数。这个函数就是在 C++ 模块最后一行定义的 Initialize 函数。Initialize 会设置导出的对象。我们从 JS 可以访问 Initialize 导出的对象。V8 中，JS 调用 C++ 函数的规则是函数入参 const FunctionCallbackInfo<Value>& args（拿到 JS 传过来的内容）和设置返回值 args.GetReturnValue().Set（给 JS 返回的内容），GetInternalBinding 函数的逻辑就是执行对应模块的钩子函数，并传一个 exports 变量进去，然后钩子函数会修改 exports 的值，该 exports 的值就是 JS 层能拿到的值。

第二十章 拓展 Node.js

拓展 Node.js 从宏观来说，有几种方式，包括直接修改 Node.js 内核重新编译分发、提供 npm 包。npm 包又可以分为 JS 和 C++ 拓展。本章主要是介绍修改 Node.js 内核和写 C++ 插件。

20.1 修改 Node.js 内核

修改 Node.js 内核的方式也有很多种，我们可以修改 JS 层、C++、C 语言层的代码，也可以新增一些功能或模块。本节分别介绍如何新增一个 Node.js 的 C++ 模块和修改 Node.js 内核。相比修改 Node.js 内核代码，新增一个 Node.js 内置模块需要了解更多的知识。

20.1.1 新增一个内置 C++ 模块

1. 首先在 src 文件夹下新增两个文件。

cyb.h

```

1. #ifndef SRC_CYB_H_
2. #define SRC_CYB_H_
3. #include "v8.h"
4.
5. namespace node {
6. class Environment;
7. class Cyb {
8. public:
9.     static void Initialize(v8::Local<v8::Object> target,
10.                           v8::Local<v8::Value> unused,
11.                           v8::Local<v8::Context> context,
12.                           void* priv);
13. private:
14.     static void Console(const v8::FunctionCallbackInfo<v8::Value>& args);
15. };
16. } // namespace node
17.#endif

```

cyb.cc

```

1. #include "cyb.h"
2. #include "env-inl.h"
3. #include "util-inl.h"
4. #include "node_internals.h"
5.
6. namespace node {
7. using v8::Context;
8. using v8::Function;
9. using v8::FunctionCallbackInfo;
10. using v8::FunctionTemplate;
11. using v8::Local;
12. using v8::Object;
13. using v8::String;
14. using v8::Value;
15.
16. void Cyb::Initialize(Local<Object> target,

```

```
17.                     Local<Value> unused,
18.                     Local<Context> context,
19.                     void* priv) {
20.     Environment* env = Environment::GetCurrent(context);
21.     // 申请一个函数模块，模板函数是 Console
22.     Local<FunctionTemplate> t = env->NewFunctionTemplate(Console);
23.     // 申请一个字符串
24.     Local<String> str = FIXED_ONE_BYTE_STRING(env->isolate(),
25.                                                 "console");
26.     // 设置函数名
27.     t->SetClassName(str);
28.     // 导出函数，target 即 exports
29.     target->Set(env->context(),
30.                   str,
31.                   t->GetFunction(env->context()).ToLocalChecke
32.                   d()).Check();
33. }
34.
35. void Cyb::Console(const FunctionCallbackInfo<Value>& args) {
36.     v8::Isolate* isolate = args.GetIsolate();
37.     v8::Local<String> str = String::NewFromUtf8(isolate,
38.                                                 "hello world");
39.     args.GetReturnValue().Set(str);
40. }
41.
42. } // namespace node
43. // 声明该模块
44. NODE_MODULE_CONTEXT_AWARE_INTERNAL(cyb_wrap, node::Cyb::Initialize)
```

我们新定义一个模块，是不能自动添加到 Node.js 内核的，我们还需要额外的操作。

1 首先我们需要修改 node.gyp 文件。把我们新增的文件加到配置里，否则编译的时候，不会编译这个新增的模块。我们可以在 node.gyp 文件中找到 src/tcp_wrap.cc，然后在它后面加入我们的文件就行。

```
1. src/cyb_wrap.cc
2. src/cyb_wrap.h
```

这时候 Node.js 会编译我们的代码了。但是 Node.js 的内置模块有一定的机制，我们的代码加入了 Node.js 内核，不代表就可以使用了。Node.js 在初始化的时候会调用 RegisterBuiltInModules 函数注册所有的内置 C++ 模块。

```
1. void RegisterBuiltInModules() {
```

```

2. #define V(modname) _register_##modname();
3.     NODE_BUILTIN_MODULES(V)
4. #undef V
5. }
```

我们看到该函数只有一个宏。我们看看这个宏。

```

1. void RegisterBuiltinModules() {
2. #define V(modname) _register_##modname();
3.     NODE_BUILTIN_MODULES(V)
4. #undef V
5. }
6. #define NODE_BUILTIN_MODULES(V) \
7.     NODE_BUILTIN_STANDARD_MODULES(V) \
8.     NODE_BUILTIN_OPENSSL_MODULES(V) \
9.     NODE_BUILTIN_ICU_MODULES(V) \
10.    NODE_BUILTIN_REPORT_MODULES(V) \
11.    NODE_BUILTIN_PROFILER_MODULES(V) \
12.    NODE_BUILTIN_DTRACE_MODULES(V)
```

宏里面又是一堆宏。我们要做的就是修改这个宏。因为我们是自定义的内置模块，所以我们可以在后面增加一个宏。

```

1. #define NODE_BUILTIN_EXTEND_MODULES(V) \
2.     V(cyb_wrap)
```

然后把这个宏追加到那一堆宏后面。

```

1. #define NODE_BUILTIN_MODULES(V) \
2.     NODE_BUILTIN_STANDARD_MODULES(V) \
3.     NODE_BUILTIN_OPENSSL_MODULES(V) \
4.     NODE_BUILTIN_ICU_MODULES(V) \
5.     NODE_BUILTIN_REPORT_MODULES(V) \
6.     NODE_BUILTIN_PROFILER_MODULES(V) \
7.     NODE_BUILTIN_DTRACE_MODULES(V) \
8.     NODE_BUILTIN_EXTEND_MODULES(V)
```

这时候，Node.js 不仅可以编译我们的代码，还会把我们代码中定义的模块注册到内置 C++ 模块里了，接下来就是如何使用 C++ 模块了。

2 在 lib 文件夹新建一个 cyb.js，作为 Node.js 原生模块

```
1. const cyb = internalBinding('cyb_wrap');
```

```
| 2. module.exports = cyb;
```

新增原生模块，我们也要修改 node.gyp 文件，否则代码也不会被编译进 node 内核。我们找到 node.gyp 文件的 lib/net.js，在后面追加 lib/cyb.js。该配置下的文件是给 js2c.py 使用的，如果不修改，我们在 require 的时候，就会找不到该模块。最后我们在 lib/internal/bootstrap/loader 文件里找到 internalBindingWhitelist 变量，在数组最后增加 cyb_wrap，这个配置是给 process.binding 函数使用的，如果不修改这个配置，通过 process.binding 就找不到我们的模块。process.binding 是可以在用户 JS 里使用的。至此，我们完成了所有的修改工作，重新编译 Node.js。然后编写测试程序。

3 新建一个测试文件 testcyb.js

```
| 1. // const cyb = process.binding('cyb_wrap');  
| 2. const cyb = require('cyb');  
| 3. console.log(cyb.console())
```

可以看到，会输出 hello world。

20.1.2 修改 Node.js 内核

本节介绍如何修改 Node.js 内核。修改的部分主要是为了完善 Node.js 的 TCP keepalive 功能。目前 Node.js 的 keepalive 只支持设置开关以及空闲多久后发送探测包。在新版 Linux 内核中，TCP keepalive 包括以下配置。

1 多久没有通信数据包，则开始发送探测包。

2 每隔多久，再次发送探测包。

3 发送多少个探测包后，就认为连接断开。

4 TCP_USER_TIMEOUT，发送了数据，多久没有收到 ack 后，认为连接断开。

Node.js 只支持第一条，所以我们的目的是支持 2, 3, 4。因为这个功能是操作系统提供的，所以首先需要修改 Libuv 的代码。

1 修改 src/unix/tcp.c

在 tcp.c 加入以下代码

```
| 1. int uv_tcp_keepalive_ex(uv_tcp_t* handle,  
| 2.                           int on,  
| 3.                           unsigned int delay,  
| 4.                           unsigned int interval,  
| 5.                           unsigned int count) {  
| 6.     int err;  
| 7.  
| 8.     if (uv__stream_fd(handle) != -1) {  
| 9.         err = uv__tcp_keepalive_ex(uv__stream_fd(handle),  
|10.                               on,  
|11.                               delay,  
|12.                               interval,
```

```
13.                                     count);
14.     if (err)
15.         return err;
16.     }
17.
18.     if (on)
19.         handle->flags |= UV_HANDLE_TCP_KEEPALIVE;
20.     else
21.         handle->flags &= ~UV_HANDLE_TCP_KEEPALIVE;
22.     return 0;
23. }
24.
25. int uv_tcp_timeout(uv_tcp_t* handle, unsigned int timeout) {
26. #ifdef TCP_USER_TIMEOUT
27.     int fd = uv_stream_fd(handle);
28.     if (fd != -1 && setsockopt(fd,
29.                                 IPPROTO_TCP,
30.                                 TCP_USER_TIMEOUT,
31.                                 &timeout,
32.                                 sizeof(timeout))) {
33.         return UV__ERR(errno);
34.     }
35. #endif
36.     return 0;
37. }
38.
39. int uv_tcp_keepalive_ex(int fd,
40.                         int on,
41.                         unsigned int delay,
42.                         unsigned int interval,
43.                         unsigned int count) {
44.     if (setsockopt(fd, SOL_SOCKET, SO_KEEPALIVE, &on, sizeof(on)))
45.         return UV__ERR(errno);
46.
47. #ifdef TCP_KEEPIDLE
48.     if (on && delay && setsockopt(fd,
49.                                 IPPROTO_TCP,
50.                                 TCP_KEEPIDLE,
51.                                 &delay,
52.                                 sizeof(delay)))
53.         return UV__ERR(errno);
54. #endif
55. #ifdef TCP_KEEPINTVL
56.     if (on && interval && setsockopt(fd,
57.                                 IPPROTO_TCP,
```

```
58.                                     TCP_KEEPINTVL,
59.                                     &interval,
60.                                     sizeof(interval)))
61.     return UV__ERR(errno);
62. #endif
63. #ifdef TCP_KEEPCNT
64.     if (on && count && setsockopt(fd,
65.                                         IPPROTO_TCP,
66.                                         TCP_KEEPCNT,
67.                                         &count,
68.                                         sizeof(count)))
69.     return UV__ERR(errno);
70. #endif
71. /* Solaris/SmartOS, if you don't support keep-alive,
72. * then don't advertise it in your system headers...
73. */
74. /* FIXME(bnoordhuis) That's possibly because sizeof(delay) shou
    uld be 1. */
75. #if defined(TCP_KEEPALIVE) && !defined(__sun)
76.     if (on && setsockopt(fd, IPPROTO_TCP, TCP_KEEPALIVE, &delay, s
        izeof(delay)))
77.     return UV__ERR(errno);
78. #endif
79.
80. return 0;
81. }
```

2 修改 [include/uv.h](#)

把在 `tcp.c` 中加入的接口暴露出来。

```
1. UV_EXTERN int uv_tcp_keepalive_ex(uv_tcp_t* handle,
2.                                     int enable,
3.                                     unsigned int delay,
4.                                     unsigned int interval,
5.                                     unsigned int count);
6. UV_EXTERN int uv_tcp_timeout(uv_tcp_t* handle, unsigned int timeo
ut);
```

至此，我们就修改完 Libuv 的代码，也对外暴露了设置的接口，接着我们修改上层的 C++ 和 JS 代码，使得我们可以在 JS 层使用该功能。

3 修改 [src/tcp_wrap.cc](#)

修改 `TCPWrap::Initialize` 函数的代码。

```
1. env->SetProtoMethod(t, "setKeepAliveEx", SetKeepAliveEx);
2. env->SetProtoMethod(t, "setKeepAliveTimeout", SetKeepAliveTimeout
    );
```

首先对 JS 层暴露两个新的 API。我们看看这两个 API 的定义。

```

1. void TCPWrap::SetKeepAliveEx(const FunctionCallbackInfo<Value>& args) {
2.     TCPWrap* wrap;
3.     ASSIGN_OR_RETURN_UNWRAP(&wrap,
4.                             args.Holder(),
5.                             args.GetReturnValue().Set(UV_EBADF));
6.     Environment* env = wrap->env();
7.     int enable;
8.     if (!args[0]->Int32Value(env->context()).To(&enable)) return;
9.     unsigned int delay = static_cast<unsigned int>(args[1].As<Uint32>()->Value());
10.    unsigned int detail = static_cast<unsigned int>(args[2].As<Uint32>()->Value());
11.    unsigned int count = static_cast<unsigned int>(args[3].As<Uint32>()->Value());
12.    int err = uv_tcp_keepalive_ex(&wrap->handle_, enable, delay, detail, count);
13.    args.GetReturnValue().Set(err);
14. }
15.
16. void TCPWrap::SetKeepAliveTimeout(const FunctionCallbackInfo<Value>& args) {
17.     TCPWrap* wrap;
18.     ASSIGN_OR_RETURN_UNWRAP(&wrap,
19.                             args.Holder(),
20.                             args.GetReturnValue().Set(UV_EBADF));
21.     unsigned int time = static_cast<unsigned int>(args[0].As<Uint32>()->Value());
22.     int err = uv_tcp_timeout(&wrap->handle_, time);
23.     args.GetReturnValue().Set(err);
24. }
```

同时还需要在 [src/tcp_wrap.h](#) 中声明这两个函数。

```

1. static void SetKeepAliveEx(const v8::FunctionCallbackInfo<v8::Value>& args);
2. static void SetKeepAliveTimeout(const v8::FunctionCallbackInfo<v8::Value>& args);
```

4 修改 [lib/net.js](#)

```

1. Socket.prototype.setKeepAliveEx = function(setting,
2.                                         secs,
```

```
3.                                         interval,
4.                                         count) {
5.   if (!this._handle) {
6.     this.once('connect', () => this.setKeepAliveEx(setting,
7.                                                       secs,
8.                                                       interval,
9.                                                       count));
10.    return this;
11.  }
12.
13.  if (this._handle.setKeepAliveEx)
14.    this._handle.setKeepAliveEx(setting,
15.                                ~~secs > 0 ? ~~secs : 0,
16.                                ~~interval > 0 ? ~~interval : 0,
17.
18.                                ~~count > 0 ? ~~count : 0);
19.
20.  return this;
21. };
22. Socket.prototype.setKeepAliveTimeout = function(timeout) {
23.   if (!this._handle) {
24.     this.once('connect', () => this.setKeepAliveTimeout(timeout)
25.   );
26.     return this;
27.   }
28.   if (this._handle.setKeepAliveTimeout)
29.     this._handle.setKeepAliveTimeout(~~timeout > 0 ? ~~timeout :
30.                                         0);
31.   return this;
32. };
```

重新编译 Node.js，我们就可以使用这两个新的 API 更灵活地控制 TCP 的 keepalive 了。

```
1. const net = require('net');
2. net.createServer((socket) => {
3.   socket.setKeepAliveEx(true, 1, 2, 3);
4.   // socket.setKeepAliveTimeout(4);
5. }).listen(1101);
```

20.2 使用 N-API 编写 C++插件

本小节介绍使用 N-API 编写 C++插件知识。Node.js C++插件本质是一个动态链接库，写完编译后，生成一个.node 文件。我们在 Node.js 里直接 require 使用，Node.js 会为我们处理一切。

首先建立一个 `test.cc` 文件

```
1. // hello.cc  using N-API
2. #include <node_api.h>
3.
4. namespace demo {
5.
6.     napi_value Method(napi_env env, napi_callback_info args) {
7.         napi_value greeting;
8.         napi_status status;
9.
10.        status = napi_create_string_utf8(env, "world", NAPI_AUTO_LENGTH,
11.                                         &greeting);
12.        if (status != napi_ok) return nullptr;
13.        return greeting;
14.
15.    napi_value init(napi_env env, napi_value exports) {
16.        napi_status status;
17.        napi_value fn;
18.
19.        status = napi_create_function(env, nullptr, 0, Method, nullptr
20.                                      , &fn);
21.        if (status != napi_ok) return nullptr;
22.        status = napi_set_named_property(env, exports, "hello", fn);
23.
24.        if (status != napi_ok) return nullptr;
25.        return exports;
26.
27.    NAPI_MODULE(NODE_GYP_MODULE_NAME, init)
28.
29. } // namespace demo
```

我们不需要具体了解代码的意思，但是从代码中我们大致知道它做了什么事情。剩下的就是阅读 N-API 的 API 文档就可以。接着我们新建一个 binding.gyp 文件。gyp 文件是 node-gyp 的配置文件。node-gyp 可以帮助我们针对不同平台生产不同的编译配置文件。比如 Linux 下的 makefile。

```
1. {
2.   "targets": [
3.     {
4.       "target_name": "test",
5.       "sources": [ "./test.cc" ]
6.     }
7.   ]
8. }
```

语法和 makefile 有点像，就是定义我们编译后的目前文件名，依赖哪些源文件。然后我们安装 node-gyp。

npm install node-gyp -g
Node.js 源码中也有一个 node-gyp，它是帮助 npm 安装拓展模块时，就地编译用的。我们安装的 node-gyp 是帮助我们生成配置文件并编译用的，具体可以参考 Node.js 文档。一切准备就绪。我们开始编译。直接执行

```
node-gyp configure  
node-gyp build
```

在路径 ./build/Release/下生成了 test.node 文件。这就是我们的拓展模块。我们编写测试程序 app.js。

```
1. var addon = require("./build/Release/test");
2. console.log(addon.hello());
```

执行

Node.js app.js
我们看到输出 world。我们已经学会了如何编写一个 Node.js 的拓展模块。剩下的就是阅读 N-API 文档，根据自己的需求编写不同的模块。

第二十一章 JS Stream

流是对生产数据和消费数据过程的抽象，流本身不生产和消费数据，它只是定义了数据处理的流程。可读流是对数据源流向其它地方的过程抽象，属于生产者，可读流是对数据流向某一目的地的过程的抽象。Node.js 中的流分为可读、可写、可读写、转换流。下面我先看一下流的基类。

21.1 流基类和流通用逻辑

```
1. const EE = require('events');
2. const util = require('util');
3. // 流的基类
4. function Stream() {
5.   EE.call(this);
6. }
7. // 继承事件订阅分发的能力
8. util.inherits(Stream, EE);
```

流的基类只提供了一个函数就是 pipe。用于实现管道化。管道化是对数据从一个地方流向另一个地方的抽象。这个方法代码比较多，分开说。

21.1.1 处理数据事件

```
1. // 数据源对象
2. var source = this;
3.
4. // 监听 data 事件，可读流有数据的时候，会触发 data 事件
5. source.on('data', ondata);
6. function ondata(chunk) {
7.   // 源流有数据到达，并且目的流可写
8.   if (dest.writable) {
9.     /*
10.      目的流过载并且源流实现了 pause 方法，
11.      那就暂停可读流的读取操作，等待目的流触发 drain 事件
12.    */
13.  }
14.}
```

```
13.     if (false === dest.write(chunk) && source.pause) {
14.         source.pause();
15.     }
16. }
17. }
18.
19. // 监听 drain 事件，目的流可以消费数据了就会触发该事件
20. dest.on('drain', ondrain);
21. function ondrain() {
22.     // 目的流可继续写了，并且可读流可读，切换成自动读取模式
23.     if (source.readable && source.resume) {
24.         source.resume();
25.     }
26. }
```

这是管道化时流控实现的地方，主要是利用了 write 返回值和 drain 事件。

21.1.2 流关闭/结束处理

```
1. /*
2.  1 dest._isStdio 是 true 表示目的流是标准输出或标准错误（见
3.      process/stdio.js），
4.  2 配置的 end 字段代表可读流触发 end 或 close 事件时，是否自动关闭可写
5.      流，默认是自动关闭。如果配置了 end 是 false，则可读流这两个事件触发
6.      时，我们需要自己关闭可写流。
7.  3 我们看到可读流的 error 事件触发时，可写流是不会被自动关闭的，需要我
8.      们自己监听可读流的 error 事件，然后手动关闭可写流。所以 if 的判断意思
9.      是不是标准输出或标准错误流，并且没有配置 end 是 false 的时候，会自动
10.     关闭可写流。而标准输出和标准错误流是在进程退出的时候才被关闭的。
11. */
12. if (!dest._isStdio && (!options || options.end !== false)) {
13.     // 源流没有数据可读了，执行 end 回调
14.     source.on('end', onend);
15.     // 源流关闭了，执行 close 回调
16.     source.on('close', onclose);
17. }
18.
19. var didOnEnd = false;
20. function onend() {
21.     if (didOnEnd) return;
22.     didOnEnd = true;
23. // 执行目的流的 end，说明写数据完毕
24.     dest.end();
25. }
```

```

26.
27. function onclose() {
28.   if (didOnEnd) return;
29.   didOnEnd = true;
30.   // 销毁目的流
31.   if (typeof dest.destroy === 'function') dest.destroy();
32. }

```

上面是可读源流结束或关闭后，如何处理可写流的逻辑。默认情况下，我们只需要监听可读流的 error 事件，然后执行可写流的关闭操作。

21.1.3 错误处理

```

1. // 可读流或者可写流出错的时候都需要停止数据的处理
2. source.on('error', onerror);
3. dest.on('error', onerror);
4. // 可读流或者可写流触发 error 事件时的处理逻辑
5. function onerror(er) {
6.   // 出错了，清除注册的事件，包括正在执行的 onerror 函数
7.   cleanup();
8.   /*
9.     如果用户没有监听流的 error 事件，则抛出错误，
10.    所以我们业务代码需要监听 error 事件
11. */
12.   if (EE.listenerCount(this, 'error') === 0) {
13.     throw er; // Unhandled stream error in pipe.
14.   }
15. }

```

在 error 事件的处理函数中，通过 cleanup 函数清除了 Node.js 本身注册的 error 事件，所以这时候如果用户没有注册 error 事件，则 error 事件的处理函数个数为 0，所以我们需要注册 error 事件。下面我们再分析 cleanup 函数的逻辑。

21.1.4 清除注册的事件

```

1. // 保证源流关闭、数据读完、目的流关闭时清除注册的事件
2. source.on('end', cleanup);
3. source.on('close', cleanup);
4. dest.on('close', cleanup);
5. // 清除所有可能会绑定的事件，如果没有绑定，执行清除也是无害的
6. function cleanup() {
7.   source.removeListener('data', ondata);
8.   dest.removeListener('drain', ondrain);
9.

```

```
10. source.removeListener('end', onend);
11. source.removeListener('close', onclose);
12.
13. source.removeListener('error', onerror);
14. dest.removeListener('error', onerror);
15.
16. source.removeListener('end', cleanup);
17. source.removeListener('close', cleanup);
18.
19. dest.removeListener('close', cleanup);
20. }
21.
22. // 触发目的流的 pipe 事件
23. dest.emit('pipe', source);
24. // 支持连续的管道化 A.pipe(B).pipe(C)
25. return dest;
```

21.1.5 流的阈值

通过 `getHighWaterMark` (`lib\internal\streams\state.js`) 函数可以计算出流的阈值，阈值用于控制用户读写数据的速度。我们看看这个函数的实现。

```
1. function getHighWaterMark(state, options, duplexKey, isDuplex) {
    // 用户定义的阈值
2.   let hwm = options.highWaterMark;
3.   // 用户定义了，则校验是否合法
4.   if (hwm != null) {
5.     if (typeof hwm !== 'number' || !(hwm >= 0))
6.       throw new errors.TypeError('ERR_INVALID_OPT_VALUE',
7.                                   'highWaterMark',
8.                                   hwm);
9.   return Math.floor(hwm);
10. } else if (isDuplex) {
11.   // 用户没有定义公共的阈值，即读写流公用的阈值
12.   // 用户是否定义了流单独的阈值，比如读流的阈值或者写流的阈值
13.   hwm = options[duplexKey];
14.   // 用户有定义
15.   if (hwm != null) {
16.     if (typeof hwm !== 'number' || !(hwm >= 0))
17.       throw new errors.TypeError('ERR_INVALID_OPT_VALUE',
18.                                   duplexKey,
19.                                   hwm);
20.   return Math.floor(hwm);
21. }
22. }
```

```

23.
24. // 默认值, 对象是 16 个, buffer 是 16KB
25. return state.objectMode ? 16 : 16 * 1024;
26. }

```

`getHighWaterMark` 函数逻辑如下

- 1 用户定义了合法的阈值，则取用户定义的（可读流、可写流、双向流）。
- 2 如果是双向流，并且用户没有可读流可写流共享的定义阈值，根据当前是可读流还是可写流，判断用户是否设置对应流的阈值。有则取用户设置的值作为阈值。
- 3 如果不满足 1,2，则返回默认值。

21.1.6 销毁流

通过调用 `destroy` 函数可以销毁一个流，包括可读流和可写流。并且可以实现 `_destroy` 函数自定义销毁的行为。我们看看可写流的 `destroy` 函数定义。

```

1. function destroy(err, cb) {
2.   // 读流、写流、双向流
3.   const readableDestroyed = this._readableState &&
4.     this._readableState.destroyed;
5.   const writableDestroyed = this._writableState &&
6.     this._writableState.destroyed;
7.   // 流是否已经销毁, 是则直接执行回调
8.   if (readableDestroyed || writableDestroyed) {
9.     // 传了 cb, 则执行, 可选地传入 err, 用户定义的 err
10.    if (cb) {
11.      cb(err);
12.    } else if (err &&
13.               (!this._writableState ||
14.                !this._writableState.errorEmitted)) {
15.      /*
16.        传了 err, 是读流或者没有触发过 error 事件的写流,
17.        则触发 error 事件
18.      */
19.      process.nextTick(emitterNT, this, err);
20.    }
21.    return this;
22.  }
23.  // 还没有销毁则开始销毁流程
24.  if (this._readableState) {
25.    this._readableState.destroyed = true;
26.  }
27.

```

```
28. if (this._writableState) {  
29.   this._writableState.destroyed = true;  
30. }  
31. // 用户可以自定义_destroy 函数  
32. this._destroy(err || null, (err) => {  
33.   // 没有 cb 但是有 error, 则触发 error 事件  
34.   if (!cb && err) {  
35.     process.nextTick(emitterNT, this, err);  
36.     // 可写流则标记已经触发过 error 事件  
37.     if (this._writableState) {  
38.       this._writableState.errorEmitted = true;  
39.     }  
40.   } else if (cb) { // 有 cb 或者没有 err  
41.     cb(err);  
42.   }  
43. });  
44.  
45. return this;  
46. }
```

destroy 函数销毁流的通用逻辑。其中_destroy 函数不同的流不一样，下面分别是可读流和可写流的实现。

1 可读流

```
1. Readable.prototype._destroy = function(err, cb) {  
2.   this.push(null);  
3.   cb(err);  
4. };
```

2 可写流

```
1. Writable.prototype._destroy = function(err, cb) {  
2.   this.end();  
3.   cb(err);  
4. };
```

21.2 可读流

Node.js 中可读流有两种工作模式：流式和暂停式，流式就是有数据的时候就会触发回调，并且把数据传给回调，暂停式就是需要用户自己手动执行读取的操作。我们通过源码去了解一下可读流实现的一些逻辑。因为实现的代码比较多，逻辑也比较绕，本文只分析一些主要的逻辑。我们先看一下 ReadableState，这个对象是表示可读流的一些状态和属性的。

```
1. function ReadableState(options, stream) {
2.     options = options || {};
3.     // 是否是双向流
4.     var isDuplex = stream instanceof Stream.Duplex;
5.     // 数据模式
6.     this.objectMode = !options.objectMode;
7.     // 双向流的时候，设置读端的模式
8.     if (isDuplex)
9.         this.objectMode = this.objectMode ||
10.             !!options.readableObjectMode;
11.    // 读到 highWaterMark 个字节则停止，对象模式的话则是 16 个对象
12.    this.highWaterMark = getHighWaterMark(this,
13.                                         options,
14.                                         'readableHighWaterMark',
15.                                         isDuplex);
16.    // 存储数据的缓冲区
17.    this.buffer = new BufferList();
18.    // 可读数据的长度
19.    this.length = 0;
20.    // 管道的目的源和个数
21.    this.pipes = null;
22.    // 工作模式
23.    this.flowing = null;
24.    // 流是否已经结束
25.    this.ended = false;
26.    // 是否触发过 end 事件了
27.    this.endEmitted = false;
28.    // 是否正在读取数据
29.    this.reading = false;
30.    // 是否同步执行事件
31.    this.sync = true;
32.    // 是否需要触发 readable 事件
33.    this.needReadable = false;
34.    // 是否触发了 readable 事件
35.    this.emittedReadable = false;
36.    // 是否监听了 readable 事件
37.    this.readableListening = false;
38.    // 是否正在执行 resume 的过程
39.    this.resumeScheduled = false;
40.    // 流是否已销毁
41.    this.destroyed = false;
```

```
42.    // 数据编码格式
43.    this.defaultEncoding = options.defaultEncoding || 'utf8';
44.    /*
45.        在管道化中，有多少个写者已经达到阈值，
46.        需要等待触发 drain 事件, awaitDrain 记录达到阈值的写者个数
47.    */
48.    this.awaitDrain = 0;
49.    // 执行 maybeReadMore 函数的时候，设置为 true
50.    this.readingMore = false;
51.    this.decoder = null;
52.    this.encoding = null;
53.    // 编码解码器
54.    if (options.encoding) {
55.        if (!StringDecoder)
56.            StringDecoder = require('string_decoder').StringDecoder;
57.        this.decoder = new StringDecoder(options.encoding);
58.        this.encoding = options.encoding;
59.    }
60.}
```

ReadableState 里包含了一大堆字段，我们可以先不管它，等待用到的时候，再回头看。接着我们开始看可读流的实现。

```
1. function Readable(options) {
2.     if (!(this instanceof Readable))
3.         return new Readable(options);
4.
5.     this._readableState = new ReadableState(options, this);
6.     // 可读
7.     this.readable = true;
8.     // 用户实现的两个函数
9.     if (options) {
10.         if (typeof options.read === 'function')
11.             this._read = options.read;
12.         if (typeof options.destroy === 'function')
13.             this._destroy = options.destroy;
14.     }
15.     // 初始化父类
16.     Stream.call(this);
17.}
```

上面的逻辑不多，需要关注的是 `read` 和 `destroy` 这两个函数，如果我们是直接使用 `Readable` 使用可读流，那在 `options` 里是必须传 `read` 函数的，`destroy` 是可选的。如果我们是以继承的方式使用 `Readable`，那必须实现 `_read` 函数。`Node.js` 只是抽象了流的逻辑，具体的操作（比如可读流就是读取数据）是由用户自己实现的，因为读取操作是业务相关的。下面我们分析一下可读流的操作。

21.2.1 可读流从底层资源获取数据

对用户来说，可读流是用户获取数据的地方，但是对可读流来说，它提供数据给用户的前提是它自己有数据，所以可读流首先需要生产数据。生产数据的逻辑由 `_read` 函数实现。`_read` 函数的逻辑大概是

```
1. const data = getSomeData();
2. readableStream.push(data);
```

通过 `push` 函数，往可读流里写入数据，然后就可以为用户提供数据，我们看看 `push` 的实现，只列出主要逻辑。

```
1. Readable.prototype.push = function(chunk, encoding) {
2.   // 省略了编码处理的代码
3.   return readableAddChunk(this,
4.                           chunk,
5.                           encoding,
6.                           false,
7.                           skipChunkCheck);
8. };
9.
10. function readableAddChunk(stream,
11.                            chunk,
12.                            encoding,
13.                            addToFront,
14.                            skipChunkCheck) {
15.   var state = stream._readableState;
16.   // push null 代表流结束
17.   if (chunk === null) {
18.     state.reading = false;
19.     onEofChunk(stream, state);
20.   } else {
21.     addChunk(stream, state, chunk, false);
22.   }
23.   // 返回是否还可以读取更多数据
24.   return needMoreData(state);
```

```
25. }
26.
27. function addChunk(stream, state, chunk, addToFront) {
28.   // 是流模式并且没有缓存的数据，则直接触发 data 事件
29.   if (state.flowing && state.length === 0 && !state.sync) {
30.     stream.emit('data', chunk);
31.   } else {
32.     // 否则先把数据缓存起来
33.     state.length += state.objectMode ? 1 : chunk.length;
34.     if (addToFront)
35.       state.buffer.unshift(chunk);
36.     else
37.       state.buffer.push(chunk);
38.     // 监听了 readable 事件则触发 readable 事件，通过 read 主动读取
39.     if (state.needReadable)
40.       emitReadable(stream);
41.   }
42.   // 继续读取数据，如果可以的话
43.   maybeReadMore(stream, state);
44. }
```

总的来说，可读流首先要从某个地方获取数据，根据当前的工作模式，直接交付给用户，或者先缓存起来。可以的情况下，继续获取数据。

21.2.2 用户从可读流获取数据

用户可以通过 read 函数或者监听 data 事件来从可读流中获取数据

```
1. Readable.prototype.read = function(n) {
2.   n = parseInt(n, 10);
3.   var state = this._readableState;
4.   // 计算可读的大小
5.   n = howMuchToRead(n, state);
6.   var ret;
7.   // 需要读取的大于 0，则取读取数据到 ret 返回
8.   if (n > 0)
9.     ret = fromList(n, state);
10.  else
11.    ret = null;
12.  // 减去刚读取的长度
13.  state.length -= n;
14.  /*
```

```

15.     如果缓存里没有数据或者读完后小于阈值了,
16.     则可读流可以继续从底层资源里获取数据
17.     */
18.     if (state.length === 0 ||
19.         state.length - n < state.highWaterMark) {
20.         this._read(state.highWaterMark);
21.     }
22.     // 触发 data 事件
23.     if (ret !== null)
24.         this.emit('data', ret);
25.     return ret;
26. };

```

读取数据的操作就是计算缓存里有多少数据可以读，和用户需要的数据大小，取小的，然后返回给用户，并触发 data 事件。如果数据还没有达到阈值，则触发可读流从底层资源中获取数据。从而源源不断地生成数据。

21.3 可写流

可写流是对数据流向的抽象，用户调用可写流的接口，可写流负责控制数据的写入。流程如图 21-1 所示。



图 21-1

下面是可写流的代码逻辑图如图 21-2 所示。

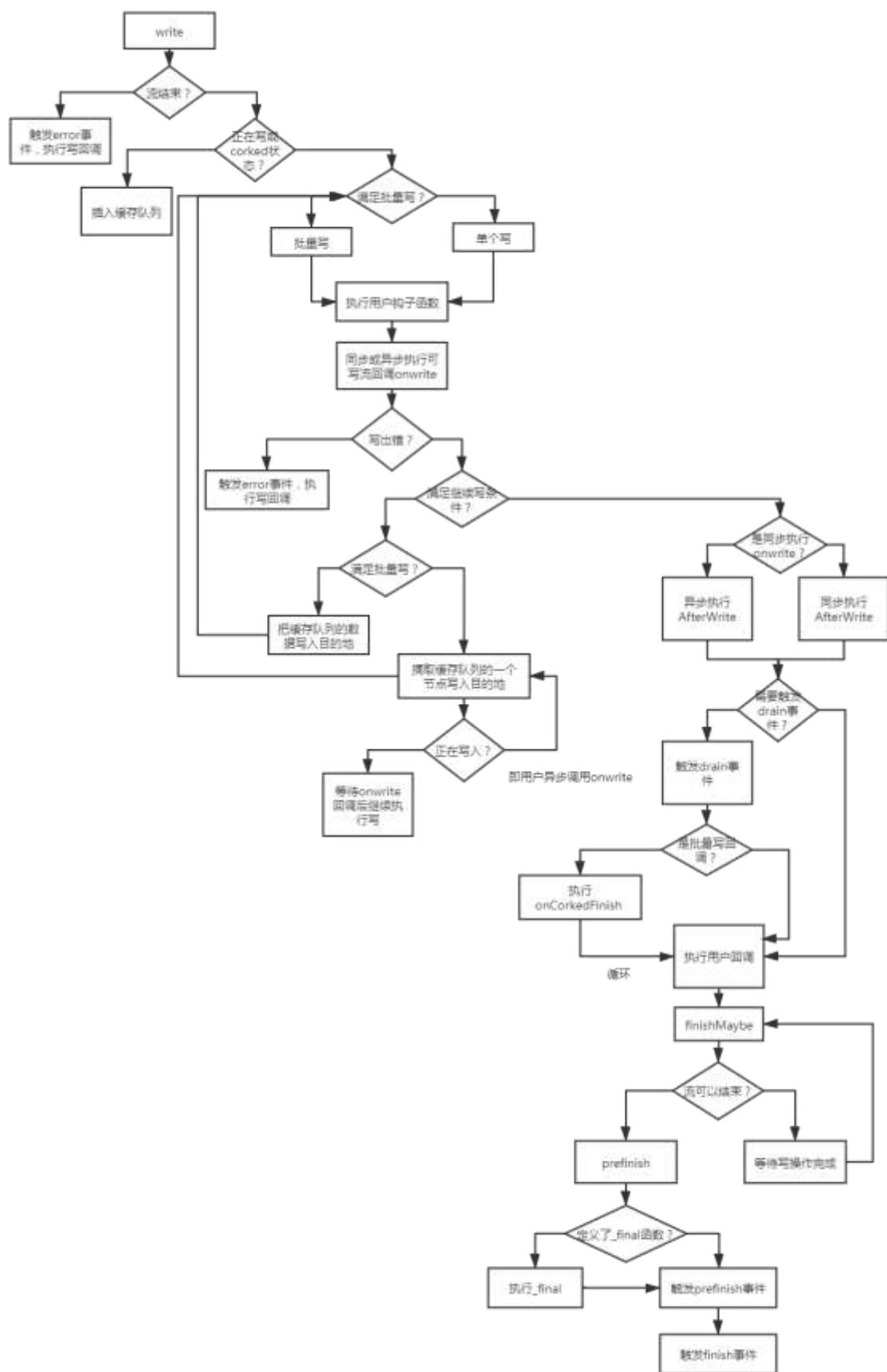


图 21-2

我们看一下可写流的实现。

21.3.1 WritableState

WritableState 是管理可写流配置的类。里面包含了非常的字段，具体含义我们会在后续分析的时候讲解。

```

1. function WritableState(options, stream) {
2.   options = options || {};
3.
4.   // 是不是双向流
5.   var isDuplex = stream instanceof Stream.Duplex;
6.
7.   // 数据模式
8.   this.objectMode = !!options.objectMode;
9.   /*
10.    双向流的流默认共享 objectMode 配置,
11.    用户可以自己配置成非共享, 即读流和写流的数据模式独立
12.   */
13.   if (isDuplex)
14.     this.objectMode = this.objectMode ||
15.                           !!options.writableObjectMode;
16.
17.   /*
18.    阈值, 超过后说明需要暂停调用 write, 0 代表每次调用 write
19.    的时候都返回 false, 用户等待 drain 事件触发后再执行 write
20.   */
21.   this.highWaterMark = getHighWaterMark(this,
22.                                         options, 'writableHighWaterMark', isDuplex);
23.
24.   // 是否调用了_final 函数
25.   this.finalCalled = false;
26.
27.   // 是否需要触发 drain 事件, 重新驱动生产者
28.   this.needDrain = false;
29.
30.   // 正在执行 end 流程
31.   this.ending = false;
32.
33.   // 是否执行过 end 函数
34.   this.ended = false;
35.
36.   // 是否触发了 finish 事件

```

```
37. this.finished = false;
38.
39. // 流是否被销毁了
40. this.destroyed = false;
41.
42. var noDecode = options.decodeStrings === false;
43. // 是否需要 decode 流数据后在执行写（调用用户定义的_write）
44. this.decodeStrings = !noDecode;
45.
46. // 编码类型
47. this.defaultEncoding = options.defaultEncoding || 'utf8';
48.
49. // 待写入的数据长度或对象数
50. this.length = 0;
51.
52. // 正在往底层写
53. this.writing = false;
54.
55. // 加塞，缓存生产者的数据，停止往底层写入
56. this.corked = 0;
57.
58. // 用户定义的_write 或者_writev 是同步还是异步调用可写流的回调函数
  onwrite
59. this.sync = true;
60.
61. // 是否正在处理缓存的数据
62. this.bufferProcessing = false;
63.
64. // 用户实现的钩子_write 函数里需要执行的回调，告诉写流写完成了
65. this.onwrite = onwrite.bind(undefined, stream);
66.
67. // 当前写操作对应的回调
68. this.writecb = null;
69.
70. // 当前写操作的数据长度或对象数
71. this.writelen = 0;
72.
73. // 缓存的数据链表头指针
74. this.bufferedRequest = null;
75.
76. // 指向缓存的数据链表最后一个节点
77. this.lastBufferedRequest = null;
78.
79. // 待执行的回调函数个数
80. this.pendingcb = 0;
```

```

81.
82. // 是否已经触发过 prefinished 事件
83. this.prefinished = false;
84.
85. // 是否已经触发过 error 事件
86. this.errorEmitted = false;
87.
88. // count buffered requests
89. // 缓存的 buffer 数
90. this.bufferedRequestCount = 0;
91.
92. /*
93. 空闲的节点链表，当把缓存数据写入底层时，corkReq 保数据的上下文（如
94. 用户回调），因为这时候，缓存链表已经被清空，
95. this.corkedRequestsFree 始终维护一个空闲节点，最多两个
96. */
97. var corkReq = { next: null, entry: null, finish: undefined };
98. corkReq.finish = onCorkedFinish.bind(undefined, corkReq, this)
99. ;
100. this.corkedRequestsFree = corkReq;
101. }

```

21.3.2 Writable

Writable 是可写流的具体实现，我们可以直接使用 Writable 作为可写流来使用，也可以继承 Writable 实现自己的可写流。

```

1. function Writable(options) {
2.     this._writableState = new WritableState(options, this);
3.     // 可写
4.     this.writable = true;
5.     // 支持用户自定义的钩子
6.     if (options) {
7.         if (typeof options.write === 'function')
8.             this._write = options.write;
9.
10.        if (typeof options.writev === 'function')
11.            this._writev = options.writev;
12.
13.        if (typeof options.destroy === 'function')
14.            this._destroy = options.destroy;
15.

```

```
16.     if (typeof options.final === 'function')
17.         this._final = options.final;
18.     }
19.
20.     Stream.call(this);
21. }
```

可写流继承于流基类，提供几个钩子函数，用户可以自定义钩子函数实现自己的逻辑。如果用户是直接使用 Writable 类作为可写流，则 options.write 函数是必须传的，options.wirte 函数控制数据往哪里写，并且通知可写流是否写完成了。如果用户是以继承 Writable 类的形式使用可写流，则 _write 函数是必须实现的，_write 函数和 options.write 函数的作用是一样的。

21.3.3 数据写入

可写流提供 write 函数给用户实现数据的写入，写入有两种方式。一个是逐个写，一个是批量写，批量写是可选的，取决于用户的实现，如果用户直接使用 Writable 则需要传入 writev，如果是继承方式使用 Writable 则实现 _writev 函数。我们先看一下 write 函数的实现

```
1. Writable.prototype.write = function(chunk, encoding, cb) {
2.     var state = this._writableState;
3.     // 告诉用户是否还可以继续调用 write
4.     var ret = false;
5.     // 数据格式
6.     var isBuf = !state.objectMode && Stream._isUint8Array(chunk);
7.     // 是否需要转成 buffer 格式
8.     if (isBuf && Object.getPrototypeOf(chunk) !== Buffer.prototype)
9.     {
10.         chunk = Stream._uint8ArrayToBuffer(chunk);
11.     }
12.     // 参数处理，传了数据和回调，没有传编码类型
13.     if (typeof encoding === 'function') {
14.         cb = encoding;
15.         encoding = null;
16.     }
17.     // 是 buffer 类型则设置成 buffer，否则如果没传则取默认编码
18.     if (isBuf)
19.         encoding = 'buffer';
20.     else if (!encoding)
21.         encoding = state.defaultEncoding;
22.     if (typeof cb !== 'function')
```

```

23.     cb = nop;
24.     // 正在执行 end, 再执行 write, 报错
25.     if (state.ending)
26.       writeAfterEnd(this, cb);
27.     else if (isBuf || validChunk(this, state, chunk, cb)) {
28.       // 待执行的回调数加一, 即 cb
29.       state.pendingcb++;
30.       // 写入或缓存, 见该函数
31.       ret = writeOrBuffer(this, state, isBuf, chunk, encoding, cb)
32.     ;
33.   }
34.   // 还能不能继续写
35.   return ret;
36. };

```

`write` 函数首先做了一些参数处理和数据转换，然后判断流是否已经结束了，如果流结束再执行写入，则会报错。如果流没有结束则执行写入或者缓存处理。最后通知用户是否还可以继续调用 `write` 写入数据（我们看到如果写入的数据比阈值大，可写流还是会执行写入操作，但是会返回 `false` 告诉用户些不要写入了，如果调用方继续写入的话，也是没会继续写入的，但是可能会导致写入端压力过大）。我们首先看一下 `writeAfterEnd` 的逻辑。然后再看 `writeOrBuffer`。

```

1. function writeAfterEnd(stream, cb) {
2.   var er = new errors.Error('ERR_STREAM_WRITE_AFTER_END');
3.   stream.emit('error', er);
4.   process.nextTick(cb, er);
5. }

```

`writeAfterEnd` 函数的逻辑比较简单，首先触发可写流的 `error` 事件，然后下一个 `tick` 的时候执行用户在调用 `write` 时传入的回调。接着我们看一下 `writeOrBuffer`。

`writeOrBuffer` 函数会对数据进行缓存或者直接写入目的地（目的地可以是文件、socket、内存，取决于用户的实现），取决于当前可写流的状态。

```

1. function writeOrBuffer(stream, state, isBuf, chunk, encoding, cb)
2.   {
3.     // 数据处理
4.     if (!isBuf) {
5.       var newChunk = decodeChunk(state, chunk, encoding);
6.       if (chunk !== newChunk) {
7.         isBuf = true;
8.         encoding = 'buffer';
9.         chunk = newChunk;
10.      }
11.    }

```

```
11. // 对象模式的算一个
12. var len = state.objectMode ? 1 : chunk.length;
13. // 更新待写入数据长度或对象个数
14. state.length += len;
15. // 待写入的长度是否超过了阈值
16. var ret = state.length < state.highWaterMark;
17.
18. /*
19. 超过了阈值，则设置需要等待 drain 事件标记，
20. 这时候用户不应该再执行 write，而是等待 drain 事件触发
21. */
22. if (!ret)
23.     state.needDrain = true;
24. // 如果正在写或者设置了阻塞则先缓存数据，否则直接写入
25. if (state.writing || state.corked) {
26.     // 指向当前的尾节点
27.     var last = state.lastBufferedRequest;
28.     // 插入新的尾结点
29.     state.lastBufferedRequest = {
30.         chunk,
31.         encoding,
32.         isBuf,
33.         callback: cb,
34.         next: null
35.     };
36.     /*
37.         之前还有节点的话，旧的尾节点的 next 指针指向新的尾节点，
38.         形成链表
39.     */
40.     if (last) {
41.         last.next = state.lastBufferedRequest;
42.     } else {
43.         /*
44.             指向 buffer 链表，bufferedRequest 相等于头指针，
45.             插入第一个 buffer 节点的时候执行到这
46.         */
47.         state.bufferedRequest = state.lastBufferedRequest;
48.     }
49.     // 缓存的 buffer 个数加一
50.     state.bufferedRequestCount += 1;
51. } else {
52.     // 直接写入
53.     doWrite(stream, state, false, len, chunk, encoding, cb);
54. }
```

```

55. // 返回是否还可以继续执行 write, 如果没有达到阈值则可以继续写
56. return ret;
57. }

```

writeOrBuffer 函数主要的逻辑如下

1 更新待写入数据的长度，判断是否达到阈值，然后通知用户是否还可以执行 write 继续写入。

2 判断当前是否正在写入或者处于 cork 模式。是的话把数据缓存起来，否则执行写操作。

我们看一下缓存的逻辑和形成的数据结构。

缓存第一个节点时，如图 21-3 所示。

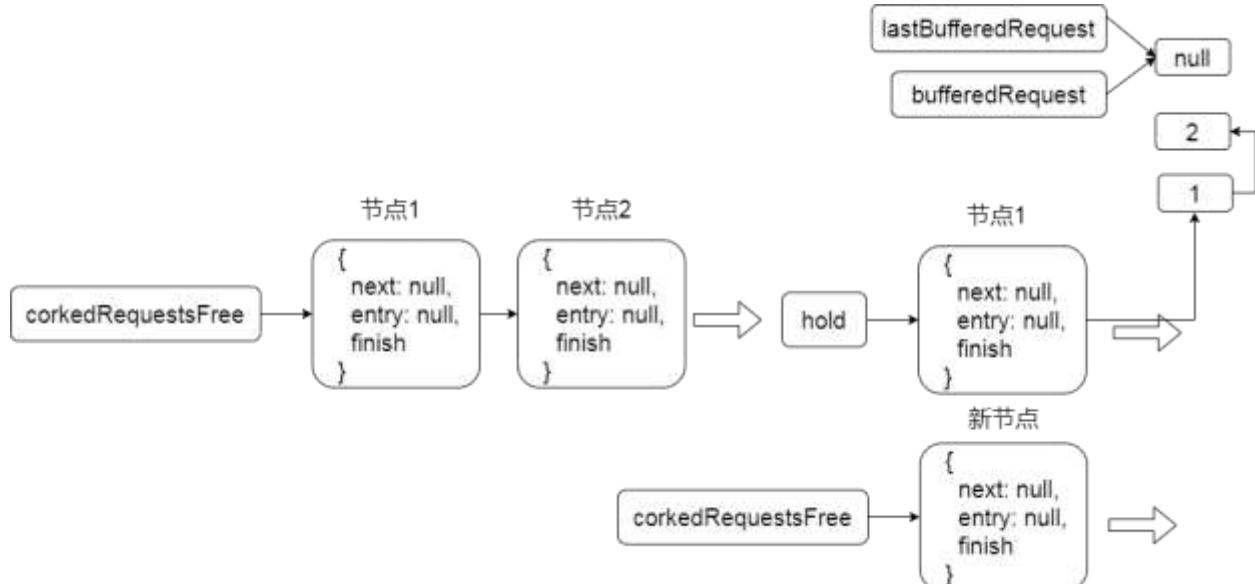


图 21-3

缓存第二个节点时，如图 21-4 所示。

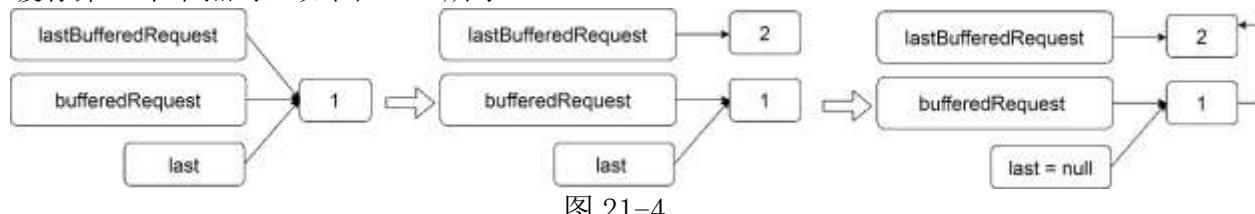


图 21-4

缓存第三个节点时，如图 21-5

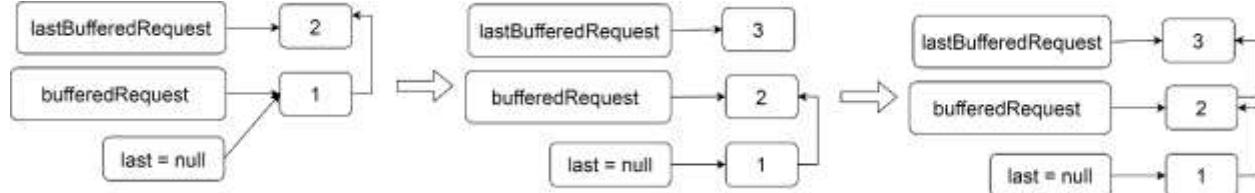


图 21-5

我们看到，函数的数据是以链表的形式管理的，其中 bufferedRequest 是链表头结点，lastBufferedRequest 指向尾节点。假设当前可写流不处于写入或者 cork 状态。我们看一下写入的逻辑。

```
1. function doWrite(stream, state, writev, len, chunk, encoding, cb)
2. {
3.     // 本次写入的数据长度
4.     state.writelen = len;
5.     // 本次写完成后执行的回调
6.     state.writecb = cb;
7.     // 正在写入
8.     state.writing = true;
9.     // 假设用户定义的_writev 或者_write 函数是同步回调 onwrite
10.    state.sync = true;
11.    if (writev)
12.        // chunk 为缓存待写入的 buffer 节点数组
13.        stream._writev(chunk, state.onwrite);
14.    else
15.        // 执行用户定义的写函数, onwrite 是 Node.js 定义的, 在初始化的时候
16.        // 设置了该函数
17.        stream._write(chunk, encoding, state.onwrite);
18.    /*
19.        如果用户是同步回调 onwrite, 则这句代码没有意义,
20.        如果是异步回调 onwrite, 这句代码会在 onwrite 之前执行,
21.        它标记用户是异步回调模式, 在 onwrite 中需要判断回调模式, 即 sync 的
22.        值
23.    */
24.    state.sync = false;
25. }
```

doWrite 函数记录了本次写入的上下文，比如长度，回调，然后设置正在写标记。最后执行写入。如果当前待写入的数据是缓存的数据并且用户实现了_writev 函数，则调用 _writev。否则调用_write。下面我们实现一个可写流的例子，把这里的逻辑串起来。

```
1. const { Writable } = require('stream');
2. class DemoWritable extends Writable {
3.     constructor() {
4.         super();
5.         this.data = null;
6.     }
7.     _write(chunk, encoding, cb) {
8.         // 保存数据
9.         this.data = this.data ? Buffer.concat([this.data, chunk])
10.            : chunk;
11.         // 执行回调告诉可写流写完成了, false 代表写成功, true 代表写失败
12.         cb(null);
13.     }
14. }
```

13. }

`Demowritable` 定义了数据流向的目的地，在用户调用 `write` 的时候，可写流会执行用户定义的 `_write`, `_write` 保存了数据，然后执行回调并传入参数，通知可写流数据写完成了，并通过参数标记写成功还是失败。这时候回到可写流侧。我们看到可写流设置的回调是 `onwrite`, `onwrite` 是在初始化可写流的时候设置的。

1. `this.onwrite = onwrite.bind(undefined, stream);`

我们接着看 `onwrite` 的实现。

```

1. function onwrite(stream, er) {
2.   var state = stream._writableState;
3.   var sync = state.sync;
4.   // 本次写完时执行的回调
5.   var cb = state.writecb;
6.   // 重置内部字段的值
7.   // 写完了，重置回调，还有多少单位的数据没有写入，数据写完，重置本次待
    写入的数据数为 0
8.   state.writing = false;
9.   state.writecb = null;
10.  state.length -= state.writelen;
11.  state.writelen = 0;
12.  // 写出错
13.  if (er)
14.    onwriteError(stream, state, sync, er, cb);
15.  else {
16.    // Check if we're actually ready to finish, but don't emit y
      et
17.    // 是否已经执行了 end，并且数据也写完了（提交写操作和最后真正执行
      中间可能执行了 end）
18.    var finished = needFinish(state);
19.    // 还没结束，并且没有设置阻塞标记，也不在处理 buffer，并且有待处理
      的缓存数据，则进行写入
20.    if (!finished &&
21.        !state.corked &&
22.        !state.bufferProcessing &&
23.        state.bufferedRequest) {
24.      clearBuffer(stream, state);
25.    }
26.    // 用户同步回调 onwrite 则 Node.js 异步执行用户回调
27.    if (sync) {
28.      process.nextTick(afterWrite, stream, state, finished, cb);
29.    } else {

```

```
30.     afterWrite(stream, state, finished, cb);
31.   }
32. }
33. }
```

onwrite 的逻辑如下

1 更新可写流的状态和数据

2 写出错则触发 error 事件和执行用户回调，写成功则判断是否满足继续执行写操作，是的话则继续写，否则执行用户回调。

我们看一下 clearBuffer 函数的逻辑，该逻辑主要是把缓存的数据写到目的地。

```
1. function clearBuffer(stream, state) {
2.   // 正在处理 buffer
3.   state.bufferProcessing = true;
4.   // 指向头结点
5.   var entry = state.bufferedRequest;
6.   // 实现了 _writev 并且有两个以上的数据块，则批量写入，即一次把所有缓存的 buffer 都写入
7.   if (stream._writev && entry && entry.next) {
8.     // Fast case, write everything using _writev()
9.     var l = state.bufferedRequestCount;
10.    var buffer = new Array(l);
11.    var holder = state.corkedRequestsFree;
12.    // 指向待写入数据的链表
13.    holder.entry = entry;
14.
15.    var count = 0;
16.    // 数据是否全部都是 buffer 格式
17.    var allBuffers = true;
18.    // 把缓存的节点放到 buffer 数组中
19.    while (entry) {
20.      buffer[count] = entry;
21.      if (!entry.isBuf)
22.        allBuffers = false;
23.      entry = entry.next;
24.      count += 1;
25.    }
26.    buffer.allBuffers = allBuffers;
27.
28.    doWrite(stream, state, true, state.length, buffer, '', holder.finish);
29.
30.    // 待执行的 cb 加一，即 holder.finish
31.    state.pendingcb++;
32.    // 清空缓存队列
```

```

33.     state.lastBufferedRequest = null;
34.     // 还有下一个节点则更新指针,下次使用
35.     if (holder.next) {
36.         state.corkedRequestsFree = holder.next;
37.         holder.next = null;
38.     } else {
39.         // 没有下一个节点则恢复值, 见初始化时的设置
40.         var corkReq = { next: null, entry: null, finish: undefined
41. };
41.         corkReq.finish = onCorkedFinish.bind(undefined, corkReq, s
tate);
42.         state.corkedRequestsFree = corkReq;
43.     }
44.     state.bufferedRequestCount = 0;
45. } else {
46.     // 慢慢写, 即一个个 buffer 写, 写完后等需要执行用户的 cb, 驱动下一
个写
47.     // Slow case, write chunks one-by-one
48.     while (entry) {
49.         var chunk = entry.chunk;
50.         var encoding = entry.encoding;
51.         var cb = entry.callback;
52.         var len = state.objectMode ? 1 : chunk.length;
53.         // 执行写入
54.         doWrite(stream, state, false, len, chunk, encoding, cb);
55.         entry = entry.next;
56.         // 处理完一个, 减一
57.         state.bufferedRequestCount--;
58.
59.     /*
60.         在 onwrite 里清除这个标记, onwrite 依赖于用户执行, 如果用户没
调,
61.             或者不是同步调, 则退出, 等待执行 onwrite 的时候再继续写
62.     */
63.     if (state.writing) {
64.         break;
65.     }
66. }
67. // 写完了缓存的数据, 则更新指针
68. if (entry === null)
69.     state.lastBufferedRequest = null;
70. }
71. /*
72.     更新缓存数据链表的头结点指向,
73.     1 如果是批量写则 entry 为 null

```

```
74.    2 如果单个写，则可能还有值（如果用户是异步调用 onwrite 的话）
75. */
76. state.bufferedRequest = entry;
77. // 本轮处理完毕（处理完一个或全部）
78. state.bufferProcessing = false;
79. }
```

clearBuffer 的逻辑看起来非常多，但是逻辑并不算很复杂。主要分为两个分支。

1 用户实现了批量写函数，则一次把缓存的时候写入目的地。首先把缓存的数据（链表）全部收集起来，然后执行执行写入，并设置回调是 finish 函数。corkedRequestsFree 字段指向一个节点数最少为一，最多为二的链表，用于保存批量写的数据的上下文。批量写时的数据结构图如图 21-6 和 21-7 所示（两种场景）。

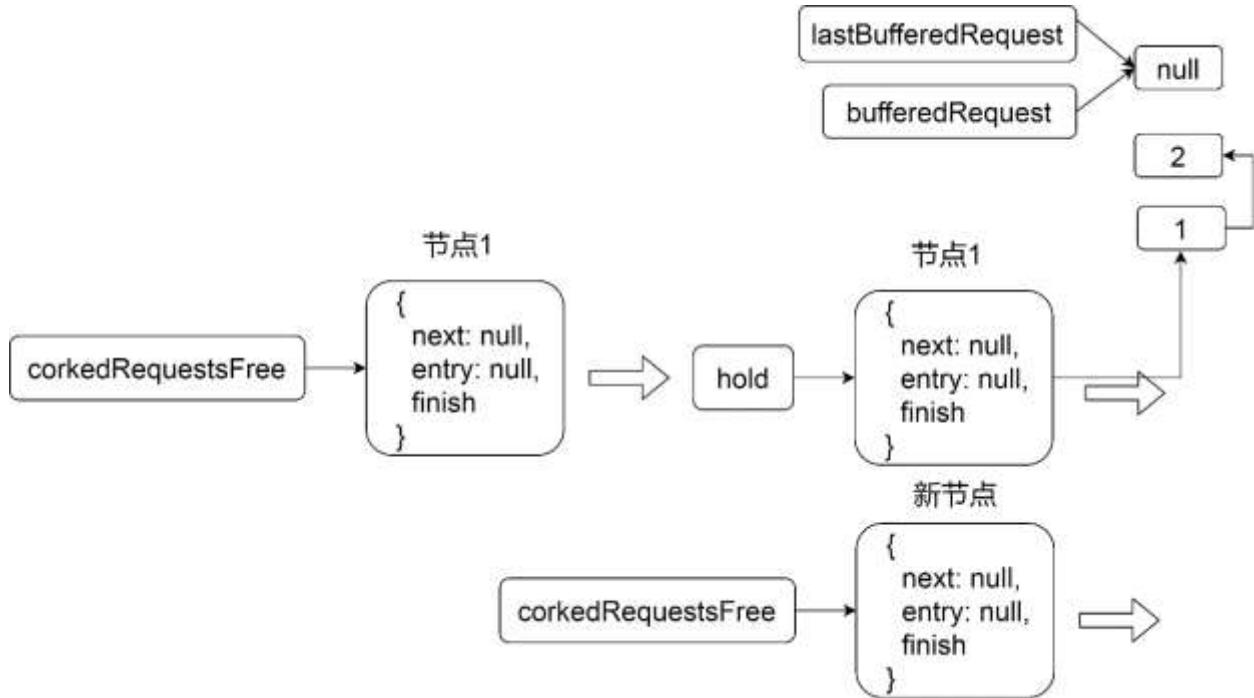


图 21-6

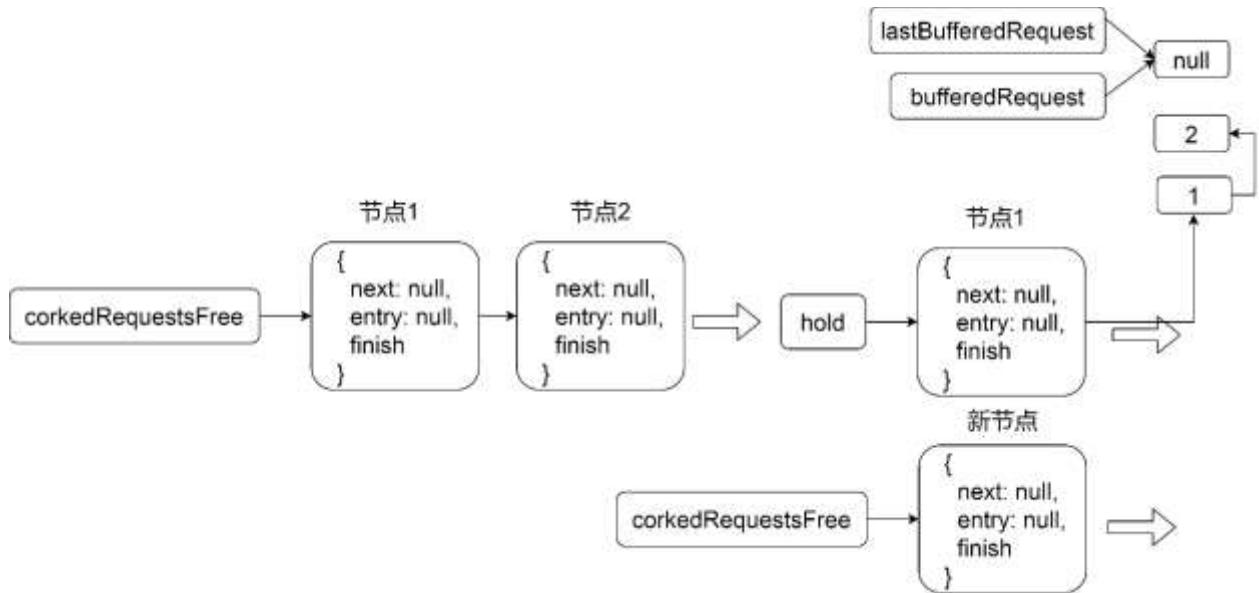


图 21-7

`corkedRequestsFree` 保证最少有一个节点，用于一次批量写，当使用完的时候，会最多保存两个空闲节点。我们看一下批量写成功后，回调函数 `onCorkedFinish` 的逻辑。

```

1. function onCorkedFinish(corkReq, state, err) {
2.   // corkReq.entry 指向当前处理的 buffer 链表头结点
3.   var entry = corkReq.entry;
4.   corkReq.entry = null;
5.   // 遍历执行用户传入的回调回调
6.   while (entry) {
7.     var cb = entry.callback;
8.     state.pendingcb--;
9.     cb(err);
10.    entry = entry.next;
11.  }
12.
13.  // 回收 corkReq, state.corkedRequestsFree 这时候已经等于新的
14.  // corkReq, 指向刚用完的这个 corkReq, 共保存两个
15.  state.corkedRequestsFree.next = corkReq;
16. }
```

`onCorkedFinish` 首先从本次批量写的数据上下文取出回调，然后逐个执行。最后回收节点。`corkedRequestsFree` 总是指向一个空闲节点，所以如果节点超过两个时，每次会把尾节点丢弃，如图 21-8 所示。

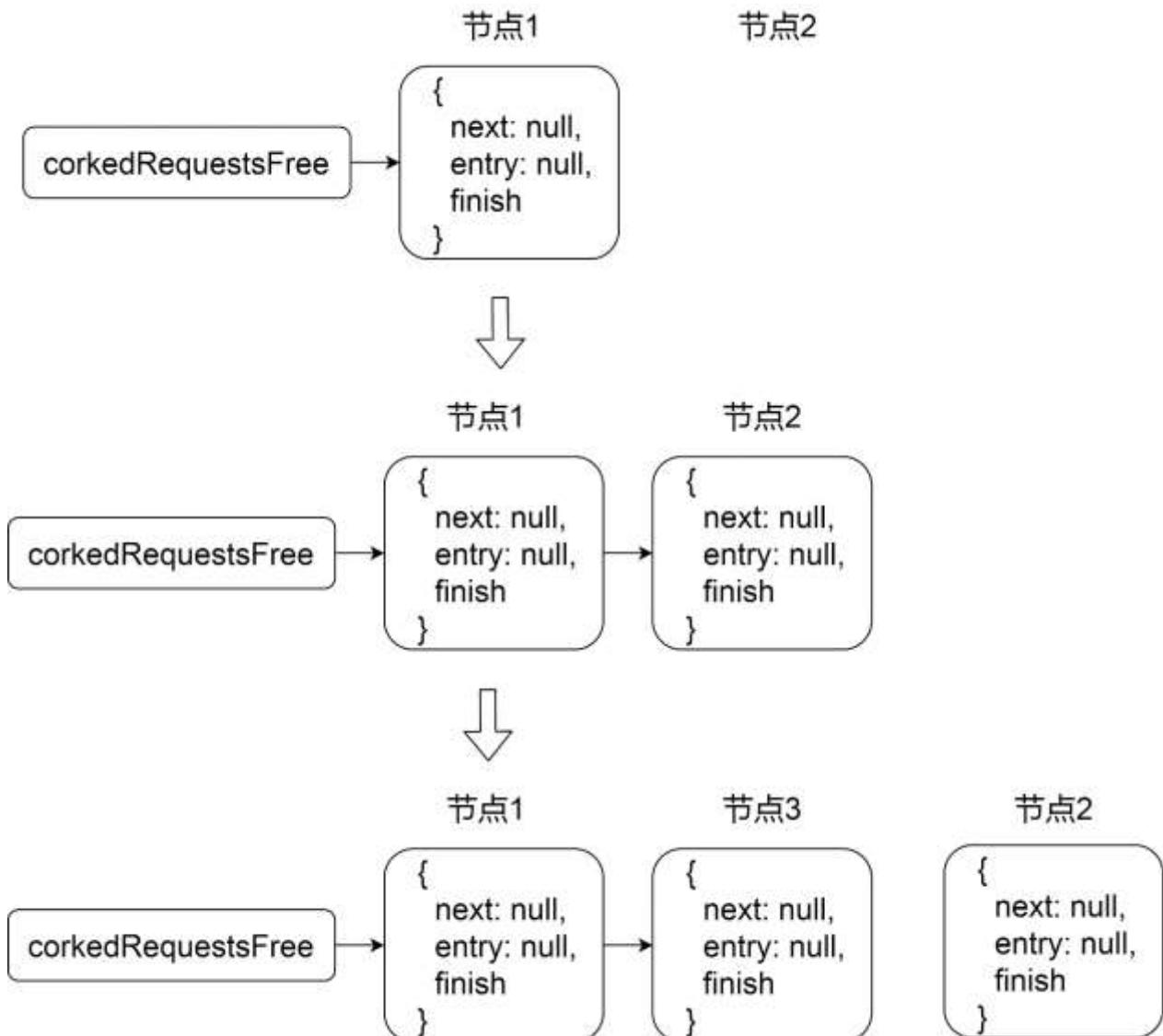


图 21-8

2 接着我们看单个写的场景

单个写的时候，就是通过 doWrite 把数据逐个写到目的地，但是有一个地方需要注意的是，如果用户是异步执行可写流的回调 onwrite（通过 writing 字段，因为 onwrite 会置 writing 为 true，如果执行完 doWrite，writing 为 false 说明是异步回调），则写入一个数据后就不再执行 doWrite 进行写，而是需要等到 onwrite 回调被异步执行时，再执行下一次写，因为可写流是串行地执行写操作。

下面讲一下 sync 字段的作用。sync 字段是用于标记执行用户自定义的 write 函数时，write 函数是同步还是异步执行可写流的回调 onwrite。主要用于控制是同步还是异步执行用户的回调。并且需要保证回调要按照定义的顺序执行。有两个地方涉及了这个逻辑，第一个是 wirte 的时候。我们看一下函数的调用关系，如图 21-9 所示。

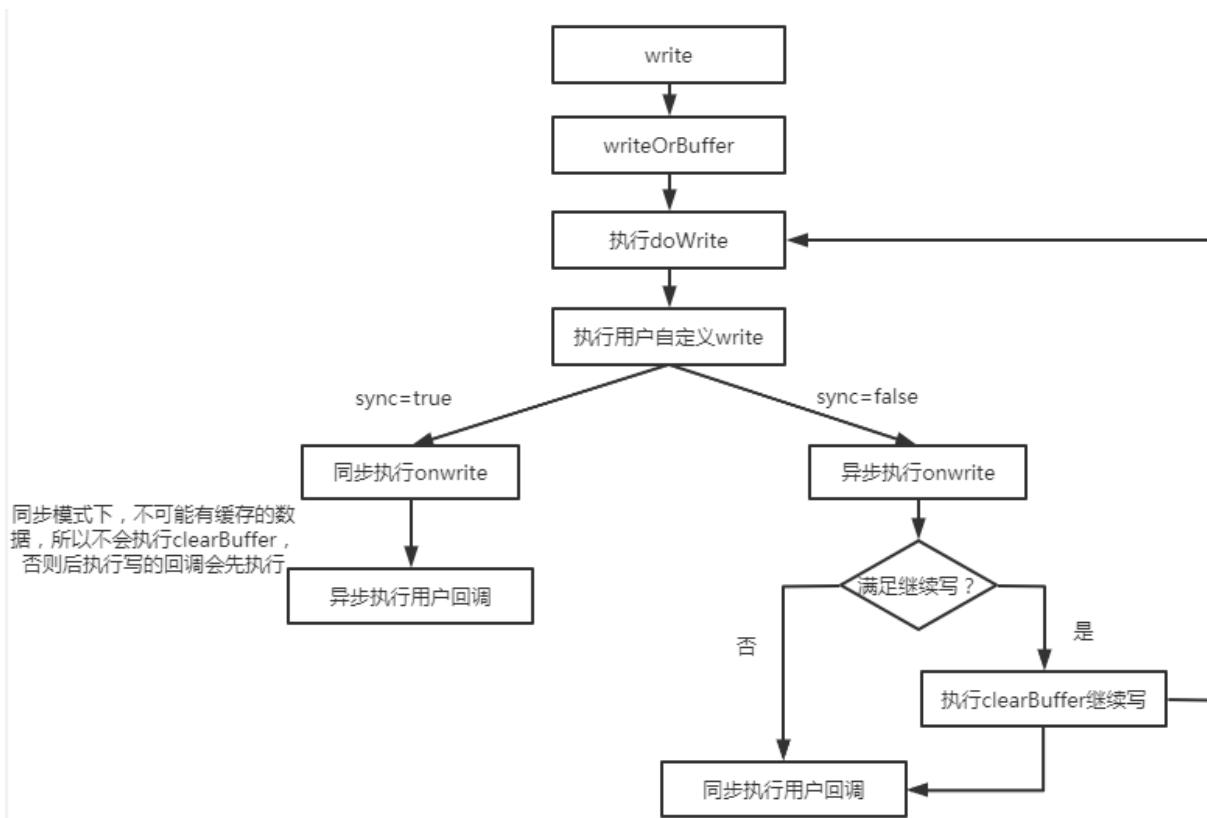


图 21-9

如果用户是同步执行 onwrite，则数据会被实时地消费，不存在缓存数据的情况，这时候 Node.js 异步并且有序地执行用户回调。如果用户连续两次调用了 write 写入数据，并且是以异步执行回调 onwrite，则第一次执行 onwrite 的时候，会存在缓存的数据，这时候还没来得及执行用户回调，就会先发生第二次写入操作，同样，第二次写操作也是异步回调 onwrite，所以接下来就会同步执行的用户回调。这样就保证了用户回调的顺序执行。第二种场景是 uncork 函数。我们看一下函数关系图，如图 21-10 所示。

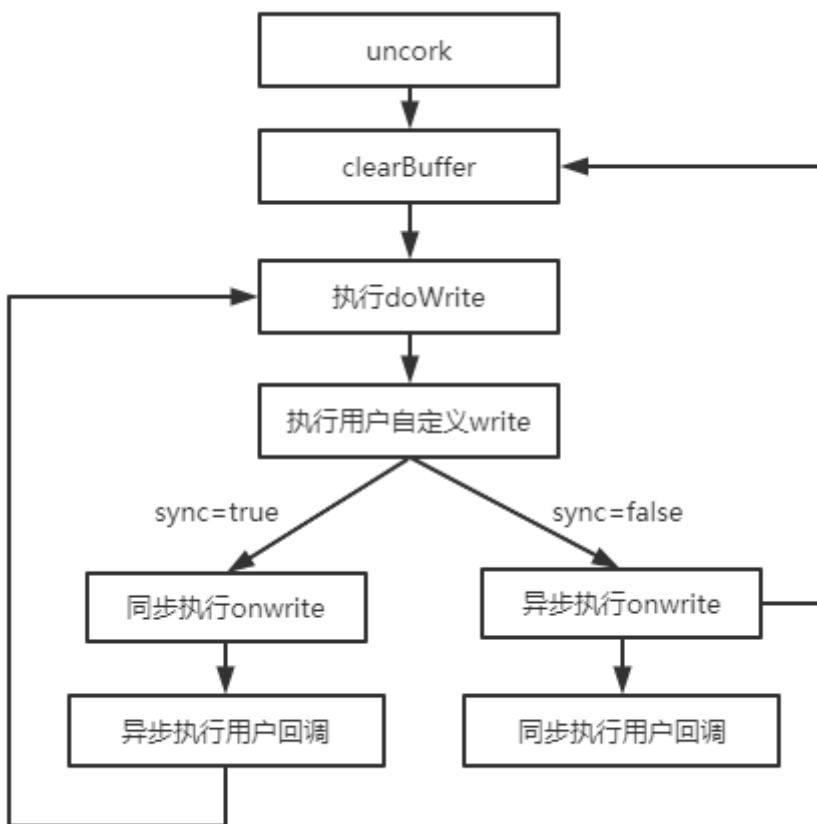


图 21-10

在 uncork 的执行流程中，如果 onwrite 是被同步回调，则在 onwrite 中不会再次调用 clearBuffer，因为这时候的 bufferProcessing 为 true。这时候会先把用户的回调入队，然后再次执行 doWrite 发起下一次写操作。如果 onwrite 是被异步执行，在执行 clearBuffer 中，第一次执行 doWrite 完毕后，clearBuffer 就会退出，并且这时候 bufferProcessing 为 false。等到 onwrite 被回调的时候，再次执行 clearBuffer，同样执行完 doWrite 的时候退出，等待异步回调，这时候用户回调被执行。

我们继续分析 onwrite 的代码，onwrite 最后会调用 afterWrite 函数。

```

1. function afterWrite(stream, state, finished, cb) {
2.   // 还没结束，看是否需要触发 drain 事件
3.   if (!finished)
4.     onwriteDrain(stream, state);
5.   // 准备执行用户回调，待执行的回调减一
6.   state.pendingcb--;
7.   cb();
8.   finishMaybe(stream, state);
9. }
10.
11. function onwriteDrain(stream, state) {
  
```

```

12. // 没有数据需要写了，并且流在阻塞中等待 drain 事件
13. if (state.length === 0 && state.needDrain) {
14.     // 触发 drain 事件然后清空标记
15.     state.needDrain = false;
16.     stream.emit('drain');
17. }
18. }
19.

```

`afterWrite` 主要是判断是否需要触发 `drain` 事件，然后执行用户回调。最后判断流是否已经结束（在异步回调 `onwrite` 的情况下，用户调用回调之前，可能执行了 `end`）。流结束的逻辑我们后面章节单独分析。

21.3.4 cork 和 uncork

cork 和 uncork 类似 `tcp` 中的 `negal` 算法，主要用于累积数据后一次性写入目的地。而不是有一块就实时写入。比如在 `tcp` 中，每次发送一个字节，而协议头远远大于一字节，有效数据占比非常低。使用 `cork` 的时候最好同时提供 `writev` 实现，否则最后 `cork` 就没有意义，因为最终还是需要一块块的数据进行写入。我们看看 `cork` 的代码。

```

1. Writable.prototype.cork = function() {
2.     var state = this._writableState;
3.     state.corked++;
4. };

```

`cork` 的代码非常简单，这里使用一个整数而不是标记位，所以 `cork` 和 `uncork` 需要配对使用。我们看看 `uncork`。

```

1. Writable.prototype.uncork = function() {
2.     var state = this._writableState;
3.
4.     if (state.corked) {
5.         state.corked--;
6.         /*
7.             没有在进行写操作（如果进行写操作则在写操作完成的回调里会执行
8.             clearBuffer），
9.             corked=0,
10.             没有在处理缓存数据（writing 为 false 已经说明），
11.             有缓存的数据待处理
12.         */
13.         if (!state.writing &&
14.             !state.corked &&
15.             !state.bufferProcessing &&
16.             state.bufferedRequest)

```

```
16.     clearBuffer(this, state);
17. }
18. };
```

21.3.5 流结束

流结束首先会把当前缓存的数据写入目的地，并且允许再执行额外的一次写操作，然后把可写流置为不可写和结束状态，并且触发一系列事件。下面是结束一个可写流的函数关系图。如图 21-11 所示。

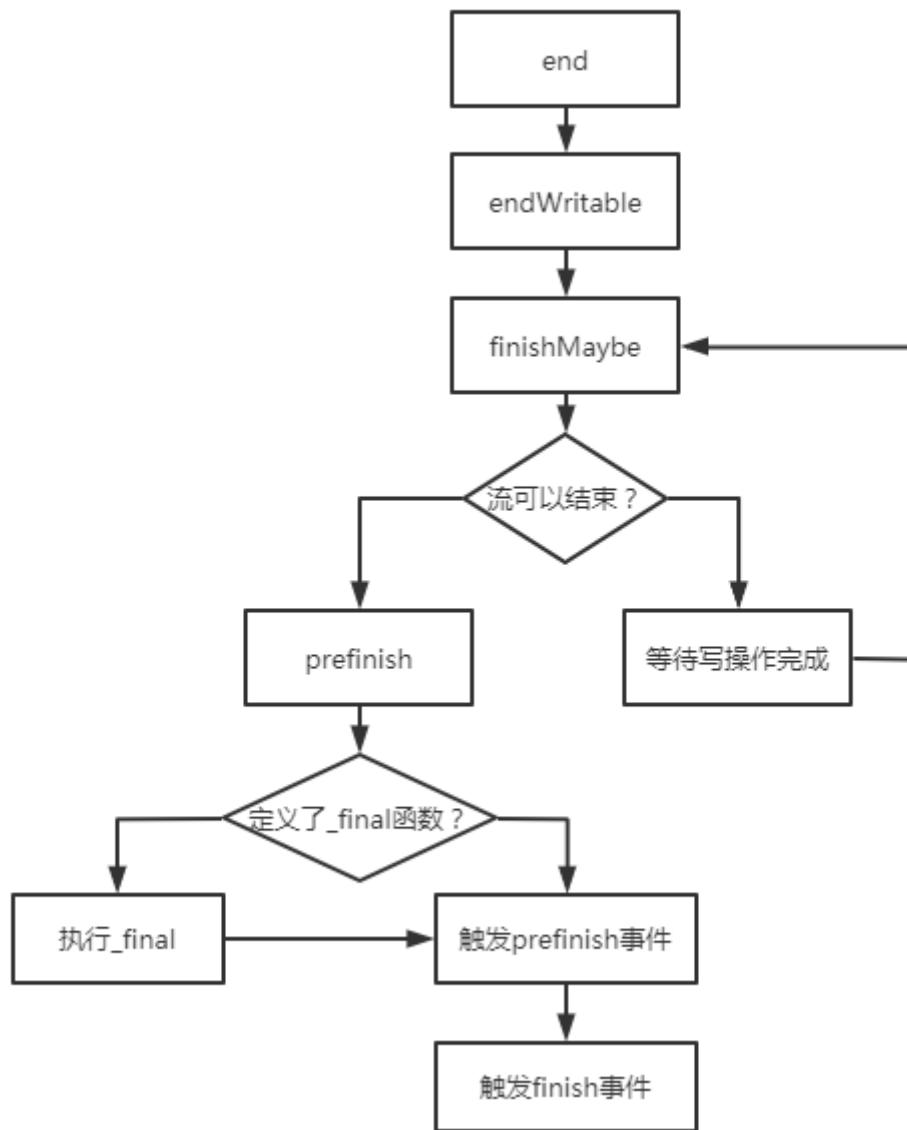


图 21-11

通过 `end` 函数可以结束可写流，我们看看该函数的逻辑。

```
1. Writable.prototype.end = function(chunk, encoding, cb) {
```

```

2.  var state = this._writableState;
3.
4.  if (typeof chunk === 'function') {
5.    cb = chunk;
6.    chunk = null;
7.    encoding = null;
8.  } else if (typeof encoding === 'function') {
9.    cb = encoding;
10.   encoding = null;
11. }
12. // 最后一次写入的机会，可能直接写入，也可以会被缓存（正在写护着处于
13. // corked 状态）
13. if (chunk !== null && chunk !== undefined)
14.   this.write(chunk, encoding);
15.
16. // 如果处于 corked 状态，则上面的写操作会被缓存，uncork 和 write 保存
17. // 可以对剩余数据执行写操作
17. if (state.corked) {
18.   // 置 1，为了 uncork 能正确执行，可以有机会写入缓存的数据
19.   state.corked = 1;
20.   this.uncork();
21. }
22.
23. if (!state.ending)
24.   endWritable(this, state, cb);
25. };

```

我们接着看 endWritable 函数

```

1. function endWritable(stream, state, cb) {
2.   // 正在执行 end 函数
3.   state.ending = true;
4.   // 判断流是否可以结束了
5.   finishMaybe(stream, state);
6.   if (cb) {
7.     // 已经触发了 finish 事件则下一个 tick 直接执行 cb，否则等待 finish
7.     // 事件
8.     if (state.finished)
9.       process.nextTick(cb);
10.    else
11.      stream.once('finish', cb);
12.   }
13. // 流结束，流不可写
14. state.ended = true;
15. stream.writable = false;

```

|16. }

`endWritable` 函数标记流不可写并且处于结束状态。但是只是代表不能再调用 `write` 写数据了，之前缓存的数据需要被写完后才能真正地结束流。我们看 `finishMaybe` 函数的逻辑。该函数用于判断流是否可以结束。

```
1. function needFinish(state) {
2.   /*
3.    执行了 end 函数则设置 ending=true,
4.    当前没有数据需要写入了,
5.    也没有缓存的数据,
6.    还没有触发 finish 事件,
7.    没有正在进行写入
8.   */
9.   return (state.ending &&
10.          state.length === 0 &&
11.          state.bufferedRequest === null &&
12.          !state.finished &&
13.          !state.writing);
14. }
15.
16. // 每次写完成的时候也会调用该函数
17. function finishMaybe(stream, state) {
18.   // 流是否可以结束
19.   var need = needFinish(state);
20.   // 是则先处理 prefinish 事件, 否则先不管, 等待写完成再调用该函数
21.   if (need) {
22.     prefinish(stream, state);
23.     // 如果没有待执行的回调, 则触发 finish 事件
24.     if (state.pendingcb === 0) {
25.       state.finished = true;
26.       stream.emit('finish');
27.     }
28.   }
29.   return need;
30. }
```

当可写流中所有数据和回调都执行了才能结束流，在结束流之前会先处理 `prefinish` 事件。

```
1. function callFinal(stream, state) {
2.   // 执行用户的 final 函数
3.   stream._final((err) => {
4.     // 执行了 callFinal 函数, cb 减一
```

```

5.     state.pendingcb--;
6.     if (err) {
7.       stream.emit('error', err);
8.     }
9.     // 执行 prefinish
10.    state.prefinished = true;
11.    stream.emit('prefinish');
12.    // 是否可以触发 finish 事件
13.    finishMaybe(stream, state);
14.  });
15. }
16. function prefinish(stream, state) {
17.   // 还没触发 prefinish 并且没有执行 finalcall
18.   if (!state.prefinished && !state.finalCalled) {
19.     // 用户传了 final 函数则, 待执行回调数加一, 即 callFinal, 否则直接
        触发 prefinish
20.     if (typeof stream._final === 'function') {
21.       state.pendingcb++;
22.       state.finalCalled = true;
23.       process.nextTick(callFinal, stream, state);
24.     } else {
25.       state.prefinished = true;
26.       stream.emit('prefinish');
27.     }
28.   }
29. }

```

如果用户定义了`_final`函数，则先执行该函数（这时候会阻止`finish`事件的触发），执行完后触发`prefinish`，再触发`finish`。如果没有定义`_final`，则直接触发`prefinish`事件。最后触发`finish`事件。

21.4 双向流

双向流是继承可读、可写的流。

```

1. util.inherits(Duplex, Readable);
2.
3. {
4.   // 把可写流中存在, 并且在可读流和 Duplex 里都不存在的方法加入到
      Duplex
5.   const keys = Object.keys(Writable.prototype);
6.   for (var v = 0; v < keys.length; v++) {
7.     const method = keys[v];
8.     if (!Duplex.prototype[method])

```

```
9.     Duplex.prototype[method] = Writable.prototype[method];
10.    }
11. }
```

```
1. function Duplex(options) {
2.     if (!(this instanceof Duplex))
3.         return new Duplex(options);
4.
5.     Readable.call(this, options);
6.     Writable.call(this, options);
7.     // 双向流默认可读
8.     if (options && options.readable === false)
9.         this.readable = false;
10.    // 双向流默认可写
11.    if (options && options.writable === false)
12.        this.writable = false;
13.    // 默认允许半开关
14.    this.allowHalfOpen = true;
15.    if (options && options.allowHalfOpen === false)
16.        this.allowHalfOpen = false;
17.
18.    this.once('end', onend);
19. }
```

双向流继承了可读流和可写流的能力。双向流实现了以下功能

21.4.1 销毁

如果读写两端都销毁，则双向流销毁。

```
1. Object.defineProperty(Duplex.prototype, 'destroyed', {
2.     enumerable: false,
3.     get() {
4.         if (this._readableState === undefined ||
5.             this._writableState === undefined) {
6.             return false;
7.         }
8.         return this._readableState.destroyed && this._writableState.d
9.     }
}
```

```
10. }
```

我们看如何销毁一个双向流。

```
1. Duplex.prototype._destroy = function(err, cb) {
2.   // 关闭读端
3.   this.push(null);
4.   // 关闭写端
5.   this.end();
6.   // 执行回调
7.   process.nextTick(cb, err);
8. };
```

双向流还有一个特性是是否允许半开关，即可读或可写。onend 是读端关闭时执行的函数。我们看看实现。

```
1. // 关闭写流
2. function onend() {
3.   // 允许半开关或写流已经结束则返回
4.   if (this.allowHalfOpen || this._writableState.ended)
5.     return;
6.   // 下一个 tick 再关闭写流，执行完这段代码，用户还可以写
7.   process.nextTick(onEndNT, this);
8. }
9.
10. function onEndNT(self) {
11.   // 调用写端的 end 函数
12.   self.end();
13. }
```

当双向流允许半开关的情况下，可读流关闭时，可写流可以不关闭。

第二十二章 events 模块

events 模块是 Node.js 中比较简单但是却非常核心的模块，Node.js 中，很多模块都继承于 events 模块，events 模块是发布、订阅模式的实现。我们首先看一个如果使用 events 模块。

```
3. const { EventEmitter } = require('events');
4. class Events extends EventEmitter {}
5. const events = new Events();
```

```
6. events.on('demo', () => {
7.   console.log('emit demo event');
8. });
9. events.emit('demo');
```

接下来我们看一下 events 模块的具体实现。

22.1 初始化

当 new 一个 EventEmitter 或者他的子类时，就会进入 EventEmitter 的逻辑。

```
30. function EventEmitter(opts) {
31.   EventEmitter.init.call(this, opts);
32. }
33.
34. EventEmitter.init = function(opts) {
35.   // 如果是未初始化或者没有自定义_events，则初始化
36.   if (this._events === undefined || 
37.     this._events === ObjectGetPrototypeOf(this)._events) {
38.     this._events = ObjectCreate(null);
39.     this._eventsCount = 0;
40.   }
41.   // 初始化处理函数个数的阈值
42.   this._maxListeners = this._maxListeners || undefined;
43.
44.   // 是否开启捕获 promise reject，默认 false
45.   if (opts && opts.captureRejections) {
46.     this[kCapture] = Boolean(opts.captureRejections);
47.   } else {
48.     this[kCapture] = EventEmitter.prototype[kCapture];
49.   }
50.};
```

EventEmitter 的初始化主要是初始化了一些数据结构和属性。唯一支持的一个参数就是 captureRejections，captureRejections 表示当触发事件，执行处理函数时，EventEmitter 是否捕获处理函数中的异常。后面我们会详细讲解。

22.2 订阅事件

初始化完 EventEmitter 之后，我们就可以开始使用订阅、发布的功能。我们可以通过 addListener、prependListener、on、once 订阅事件。addListener 和 on 是等价的，prependListener 的区别在于处理函数会被插入到队首，而默认是追加到队尾。once 注册的处理函数，最多被执行一次。四个 API 都是通过 _addListener 函数实现的。下面我们看一下具体实现。

```
15. function _addListener(target, type, listener, prepend) {
16.   let m;
17.   let events;
18.   let existing;
19.   events = target._events;
20.   // 还没有初始化_events 则初始化
21.   if (events === undefined) {
22.     events = target._events = ObjectCreate(null);
23.     target._eventsCount = 0;
24.   } else {
```

```

25.  /*
26.   是否定义了 newListener 事件，是的话先触发，如果监听了 newListener 事件，
27.   每次注册其他事件时都会触发 newListener，相当于钩子
28. */
29. if (events.newListener !== undefined) {
30.   target.emit('newListener', type,
31.             listener.listener ? listener.listener : listener);
32.   // 可能会修改_events，这里重新赋值
33.   events = target._events;
34. }
35. // 判断是否存在处理函数
36. existing = events[type];
37. }
38. // 不存在则以函数的形式存储，否则是数组
39. if (existing === undefined) {
40.   events[type] = listener;
41.   ++target._eventsCount;
42. } else {
43.   if (typeof existing === 'function') {
44.     existing = events[type] =
45.       prepend ? [listener, existing] : [existing, listener];
46.   } else if (prepend) {
47.     existing.unshift(listener);
48.   } else {
49.     existing.push(listener);
50.   }
51.
52. // 处理告警，处理函数过多可能是因为之前的没有删除，造成内存泄漏
53. m = _getMaxListeners(target);
54. if (m > 0 && existing.length > m && !existing.warned) {
55.   existing.warned = true;
56.   const w = new Error(`Possible EventEmitter memory leak detected. ` +
57.                         `${existing.length} ${String(type)} listeners ` +
58.                         `added to ${inspect(target, { depth: -1 })}. Use ` +
59.                         `emitter.setMaxListeners() to increase limit`);
60.   w.name = 'MaxListenersExceededWarning';
61.   w.emitter = target;
62.   w.type = type;
63.   w.count = existing.length;
64.   process.emitWarning(w);
65. }
66. }
67.
68. return target;
69. }

```

接下来我们看一下 once 的实现，对比其他几种 api，once 的实现相对比较难，因为我们要控制处理函数最多执行一次，所以我们需要坚持用户定义的函数，保证在事件触发的时候，执行用户定义函数的同时，还需要删除注册的事件。

```

21. EventEmitter.prototype.once = function once(type, listener) {
22.   this.on(type, _onceWrap(this, type, listener));
23.   return this;
24. };
25.
26. function onceWrapper() {
27.   // 还没有触发过
28.   if (!this.fired) {
29.     // 删除他

```

```
30.     this.target.removeListener(this.type, this.wrapFn);
31.     // 触发了
32.     this.fired = true;
33.     // 执行
34.     if (arguments.length === 0)
35.         return this.listener.call(this.target);
36.     return this.listener.apply(this.target, arguments);
37. }
38. }
39. // 支持 once api
40. function _onceWrap(target, type, listener) {
41.     // fired 是否已执行处理函数, wrapFn 包裹 listener 的函数
42.     const state = { fired: false, wrapFn: undefined, target, type, listener };
43.     // 生成一个包裹 listener 的函数
44.     const wrapped = onceWrapper.bind(state);
45.     // 把原函数 listener 也挂到包裹函数中, 用于事件没有触发前, 用户主动删除, 见 removeListener
46.     wrapped.listener = listener;
47.     // 保存包裹函数, 用于执行完后删除, 见 onceWrapper
48.     state.wrapFn = wrapped;
49.     return wrapped;
50. }
```

22.3 触发事件

分析完事件的订阅，接着我们看一下事件的触发。

```
13. EventEmitter.prototype.emit = function emit(type, ...args) {
14.     // 触发的事件是否是 error, error 事件需要特殊处理
15.     let doError = (type === 'error');
16.
17.     const events = this._events;
18.     // 定义了处理函数 (不一定是 type 事件的处理函数)
19.     if (events !== undefined) {
20.         // 如果触发的事件是 error, 并且监听了 kErrorMonitor 事件则触发 kErrorMonitor 事件
21.         if (doError && events[kErrorMonitor] !== undefined)
22.             this.emit(kErrorMonitor, ...args);
23.         // 触发的是 error 事件但是没有定义处理函数
24.         doError = (doError && events.error === undefined);
25.     } else if (!doError) // 没有定义处理函数并且触发的不是 error 事件则不需要处理,
26.         return false;
27.
28.     // If there is no 'error' event listener then throw.
29.     // 触发的是 error 事件, 但是没有定义处理 error 事件的函数, 则报错
30.     if (doError) {
31.         let er;
32.         if (args.length > 0)
33.             er = args[0];
34.         // 第一个入参是 Error 的实例
35.         if (er instanceof Error) {
36.             try {
37.                 const capture = {};
38.                 /*
39.                     给 capture 对象注入 stack 属性, stack 的值是执行 Error.captureStackTrace
40.                     语句的当前栈信息, 但是不包括 emit 的部分
41.                 */
42.                 Error.captureStackTrace(capture, EventEmitter.prototype.emit);
43.                 Object.defineProperty(er, kEnhanceStackBeforeInspector, {
```

```

44.         value: enhanceStackTrace.bind(this, er, capture),
45.         configurable: true
46.     });
47. } catch {}  

48. throw er; // Unhandled 'error' event
49. }
50.
51. let stringifiedEr;
52. const { inspect } = require('internal/util/inspect');
53. try {
54.     stringifiedEr = inspect(er);
55. } catch {
56.     stringifiedEr = er;
57. }
58. const err = new ERR_UNHANDLED_ERROR(stringifiedEr);
59. err.context = er;
60. throw err; // Unhandled 'error' event
61. }
62. // 获取 type 事件对应的处理函数
63. const handler = events[type];
64. // 没有则不处理
65. if (handler === undefined)
66.     return false;
67. // 等于函数说明只有一个
68. if (typeof handler === 'function') {
69.     // 直接执行
70.     const result = ReflectApply(handler, this, args);
71.     // 非空判断是不是 promise 并且是否需要处理, 见 addCatch
72.     if (result !== undefined && result !== null) {
73.         addCatch(this, result, type, args);
74.     }
75. } else {
76.     // 多个处理函数, 同上
77.     const len = handler.length;
78.     const listeners = arrayClone(handler, len);
79.     for (let i = 0; i < len; ++i) {
80.         const result = ReflectApply(listeners[i], this, args);
81.         if (result !== undefined && result !== null) {
82.             addCatch(this, result, type, args);
83.         }
84.     }
85. }
86.
87. return true;
88. }

```

我们看到在 Node.js 中, 对于 error 事件是特殊处理的, 如果用户没有注册 error 事件的处理函数, 可能会导致程序挂掉, 另外我们看到有一个 addCatch 的逻辑, addCatch 是为了支持事件处理函数为异步模式的情况, 比如 async 函数或者返回 Promise 的函数。

```

3. function addCatch(that, promise, type, args) {
4.     // 没有开启捕获则不需要处理
5.     if (!that[kCapture]) {
6.         return;
7.     }
8.     // that throws on second use.
9.     try {
10.         const then = promise.then;
11.

```

```
12. if (typeof then === 'function') {
13.     // 注册 reject 的处理函数
14.     then.call(promise, undefined, function(err) {
15.         process.nextTick	emitUnhandledRejectionOrErr, that, err, type, args);
16.     });
17. }
18. } catch (err) {
19.     that.emit('error', err);
20. }
21. }
22.
23. function emitUnhandledRejectionOrErr(ee, err, type, args) {
24.     // 用户实现了 kRejection 则执行
25.     if (typeof ee[kRejection] === 'function') {
26.         ee[kRejection](err, type, ...args);
27.     } else {
28.         // 保存当前值
29.         const prev = ee[kCapture];
30.         try {
31.             /*
32.                 关闭然后触发 error 事件，意义
33.                 1 防止 error 事件处理函数也抛出 error，导致死循环
34.                 2 如果用户处理了 error，则进程不会退出，所以需要恢复 kCapture 的值
35.                     如果用户没有处理 error，则 nodejs 会触发 uncaughtException，如果用户
36.                     处理了 uncaughtException 则需要灰度 kCapture 的值
37.             */
38.             ee[kCapture] = false;
39.             ee.emit('error', err);
40.         } finally {
41.             ee[kCapture] = prev;
42.         }
43.     }
44. }
```

22.4 取消订阅

我们接着看一下删除事件处理函数的逻辑。

```
3. function removeAllListeners(type) {
4.     const events = this._events;
5.     if (events === undefined)
6.         return this;
7.
8.     // 没有注册 removeListener 事件，则只需要删除数据，否则还需要触发 removeListener 事件
9.     if (events.removeListener === undefined) {
10.         // 等于 0 说明是删除全部
11.         if (arguments.length === 0) {
12.             this._events = ObjectCreate(null);
13.             this._eventsCount = 0;
14.         } else if (events[type] !== undefined) { // 否则是删除某个类型的事件,
15.             // 是唯一一个处理函数，则重置_events，否则删除对应的事件类型
16.             if (--this._eventsCount === 0)
17.                 this._events = ObjectCreate(null);
18.             else
19.                 delete events[type];
20.         }
21.         return this;
22.     }
```

```

23.
24.     // 说明注册了 removeListener 事件, arguments.length === 0 说明删除所有类型的事件
25.     if (arguments.length === 0) {
26.         // 逐个删除, 除了 removeListener 事件, 这里删除了非 removeListener 事件
27.         for (const key of ObjectKeys(events)) {
28.             if (key === 'removeListener') continue;
29.             this.removeAllListeners(key);
30.         }
31.         // 这里删除 removeListener 事件, 见下面的逻辑
32.         this.removeAllListeners('removeListener');
33.         // 重置数据结构
34.         this._events = ObjectCreate(null);
35.         this._eventsCount = 0;
36.         return this;
37.     }
38.     // 删除某类型事件
39.     const listeners = events[type];
40.
41.     if (typeof listeners === 'function') {
42.         this.removeListener(type, listeners);
43.     } else if (listeners !== undefined) {
44.         // LIFO order
45.         for (let i = listeners.length - 1; i >= 0; i--) {
46.             this.removeListener(type, listeners[i]);
47.         }
48.     }
49.
50.     return this;
51. }

```

`removeAllListeners` 函数主要的逻辑有两点，第一个是 `removeListener` 事件需要特殊处理，这类似一个钩子，每次用户删除事件处理函数的时候都会触发该事件。第二个是 `removeListener` 函数。`removeListener` 是真正删除事件处理函数的实现。`removeAllListeners` 是封装了 `removeListener` 的逻辑。

```

13. function removeListener(type, listener) {
14.     let originalListener;
15.     const events = this._events;
16.     // 没有东西可删除
17.     if (events === undefined)
18.         return this;
19.
20.     const list = events[type];
21.     // 同上
22.     if (list === undefined)
23.         return this;
24.     // list 是函数说明只有一个处理函数, 否则是数组, 如果 list.listener === listener 说明是 once
      注册的
25.     if (list === listener || list.listener === listener) {
26.         // type 类型的处理函数就一个, 并且也没有注册其他类型的事件, 则初始化_events
27.         if (--this._eventsCount === 0)
28.             this._events = ObjectCreate(null);
29.         else {
30.             // 就一个执行完删除 type 对应的属性
31.             delete events[type];
32.             // 注册了 removeListener 事件, 则先注册 removeListener 事件
33.             if (events.removeListener)
34.                 this.emit('removeListener', type, list.listener || listener);
35.         }
36.     } else if (typeof list !== 'function') {

```

```
37.     // 多个处理函数
38.     let position = -1;
39.     // 找出需要删除的函数
40.     for (let i = list.length - 1; i >= 0; i--) {
41.         if (list[i] === listener || list[i].listener === listener) {
42.             // 保存原处理函数, 如果有的话
43.             originalListener = list[i].listener;
44.             position = i;
45.             break;
46.         }
47.     }
48.
49.     if (position < 0)
50.         return this;
51.     // 第一个则出队, 否则删除一个
52.     if (position === 0)
53.         list.shift();
54.     else {
55.         if (spliceOne === undefined)
56.             spliceOne = require('internal/util').spliceOne;
57.         spliceOne(list, position);
58.     }
59.     // 如果只剩下下一个, 则值改成函数类型
60.     if (list.length === 1)
61.         events[type] = list[0];
62.     // 触发 removeListener
63.     if (events.removeListener !== undefined)
64.         this.emit('removeListener', type, originalListener || listener);
65.     }
66.
67.     return this;
68. };
```

以上就是 events 模块的核心逻辑，另外还有一些工具函数就不一一分析。

后记

感谢您耐心看到最后，如果有任何建议或意见，欢迎和我交流，如果您支持我的创造，可以进行打赏，最后欢迎 Node.js 爱好者或底层的爱好者一起交流，谢谢大家。

