



在 MDN 里面，有这么一段话：

JavaScript 有一个基于事件循环的并发模型，事件循环负责执行代码、收集和处理事件以及执行队列中的子任务。这个模型与其它语言中的模型截然不同，比如 C 和 Java。

大家是不是想问本章讲的就是这个？是也不是。

上面那段话讲的是 JavaScript，不是 Node.js。别急着问这有差吗？还记得开篇中讲的 Node.js 与 JavaScript 的关系吗？JavaScript 的确是有一个基于事件循环的并发模型。但这以前是针对前端在 DOM 中进行操作的。V8 自己有实现一套事件循环，但 **Node.js 中的事件循环则是自己实现的**。

与浏览器不同，作为能跑后端服务的 JavaScript 运行时，必须要有处理系统事件的能力。比如去处理各种文件描述符对应的读写事件。一个最简单的事件循环的伪代码可如下：

```
js
while (还有事件在监听) {
  const events = 从监听中获取所有事件信息;
  for (const event of events) {
    处理(event);
  }
}
```

一个 `setTimeout` 可以是一个 Timer 事件，一个文件读写是一个系统的 I/O 事件。对于浏览器来说，通常是没有后者的。而处理系统 I/O 事件，各系统平台早在多年前就有了成熟的方案了。如 Linux 的 `epoll`、macOS 的 `kqueue`、Windows 的 `IOCP` 等，这些统称为 I/O 多路复用。

我最早接触这类内容是在[云风](#)大大的博客上。他的一段比喻让我至今记忆犹新：

处理大量的连接的读写，`select` 是够低效的。因为 kernel 每次都要对 `select` 传入的一组 Socket 号做轮询，那次在上海，以陈榕的说法讲，这叫



.....使用 `kqueue` 这些，变成了派一些个人去站岗，鬼子来了就可以拿到通知，效率自然高了许多。

Node.js 的事件循环基石就基于此。它基于 Ryan Dahl 自己开发的 `libuv`，完成了自己的事件循环与异步 I/O。

libuv 的诞生

`libuv` 是一个聚焦异步 I/O 的跨平台库。它就是为 Node.js 而生的，后续才衍生出对其他项目的支持，如 [Luvit](#)、[Julia](#)、[uvloop](#) 等（[更多](#)）。



@稀土掘金技术社区

libuv Logo

虽然 `libuv` 为 Node.js 而生，但 Node.js 却不是一开始就拥抱 `libuv`。原初时候，Node.js 用的是 `libev` 和 `libeio` 的结合体。在 Node.js 最早期的一系列版本代码中，我们是可以看到 `libev` 的依赖的（[代码仓库](#)）。`libev` 本身就是一个异步 I/O 的库，但是它只能在遵循 POSIX 规范系统下使用。



于是 Dyan Dahl 另起炉灶，自己搞了一个非常符合 Node.js 自身需求的异步 I/O 库，连里面的 API 都是为 Node.js 的各种能力而设计。**于是 libuv 就诞生了，Node.js 于 0.5.0 的版本中引入了它。**

最初的 libuv 版本中，Linux 与 macOS 等是基于 libev 与 libeio 的进一步封装，多路复用则是基于里面的 epoll 或 kqueue 等。而在 Windows 下，就是之前提到的 IOCP 了。那个时候，在 libuv 的 README 中是这么描述自己的：

这是 Node.js 的新网络层。它的目的是抽象出 Windows 系统的 IOCP 和 UNIX 系统的 libev。我们打算最终在这个库中包含所有的平台差异。

它的 API 也很简单，无非包含文件描述符的监听、读写、连接，以及定时器.....以及因为它是个“网络层”，自然要做各种网络请求，那么对于 DNS 的查询也是必要能力之一，所以 libuv 在最开始的版本中就加入了 c-ares 的依赖，这是一个用于异步 DNS 请求的 C 库。

libuv 在 Node.js 发展到 v0.9.4 版本时，彻底移除了 libev 的依赖。自此以后，它就是清白之身啦。

事件循环 ≠ 异步 I/O

为什么本章的标题不是“事件循环”，也不是“异步 I/O”，而是二者都要？因为这两者并不等价。事件循环是一种并发模型，它的本质是一个死循环，在循环中不断处理到来的事件。异步 I/O 事件只是事件循环中事件的一种，除此之外，还有各种其他事件，前文也说过，定时器事件也是一种事件，但它并不是异步 I/O。所以这二者并不等同。

为了可以简洁明了地讲述，后续都直接以 epoll 举例，大家可类比出 kqueue 或 IOCP。

事件循环就是个死循环，那么把它的逻辑按时序拉平，就是一条直线，或者公交线路。在死循环中，大部分时间都阻塞在等待事件，这个时候并不耗 CPU，而是等待底层 epoll 得到事件。这个等待可以类比成公交车在路线上一直开，等到站。而一

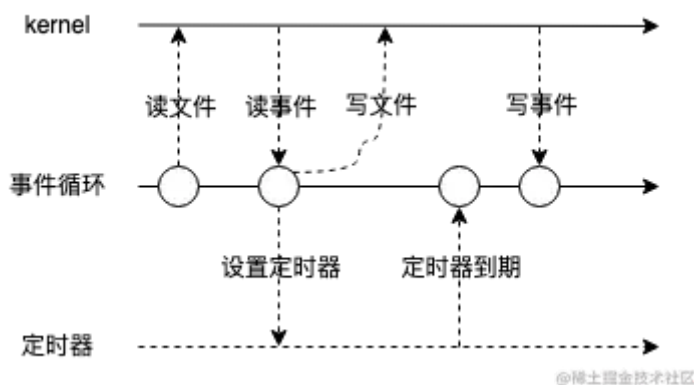


libuv 的事件循环会把对应信息给到等待这个事件的回调函数，通常这个回调函数最终会一路调用至 JavaScript，从而唤起 JavaScript 侧的文件读取的回调函数。这就是用户这段代码的底层表现：

```
fs.readFile(filename, (err, content) => {  
  // 这里就是 callback  
});
```

js

代码里面读取某个文件，然后就调用 libuv 去读取，假设没有更多其他事件，那么 libuv 调用之后，epoll 就会等着读取完毕的事件，底层代码就阻塞在等待事件上，直到读取完成，有事件通知，才会最终调用。这就相当于公交车一路行驶，然后看到到站了，就停下来，执行一波上下客操作。做完后继续上路。后续会有专门的章节去讲 fs 相关的内容，所以此处这段代码就点到为止。



@博士掘金技术社区

上面讲的事件循环就跟上面这张图类似。事件循环那条线可以理解为之前提到的死循环，或者公交路线。我们可以看到，除了始发站之外，剩下的站点都必须由某个事件触发，比如 kernel 传过来的 epoll 事件，或者 libuv 自己内部的定时器事件。也只有在各站点中，事件循环才能去做其他逻辑，比如执行一段代码，或者这段代码里面有去做其他的 I/O 操作、定时器操作等。剩下时间都阻塞在 epoll 等待上面。我们称这一个站点（节点）为一个 tick。

上面的图转化为用户的 Node.js 代码，大概长这样：

```
fs.readFile(filename, (err, content) => {  
  fs.writeFile(filename, content, err => {  
    // 假设这个 `writeFile` 写文件要持续 2 秒钟  
  });  
});
```

js



```
});
```

按站点解释，就是：

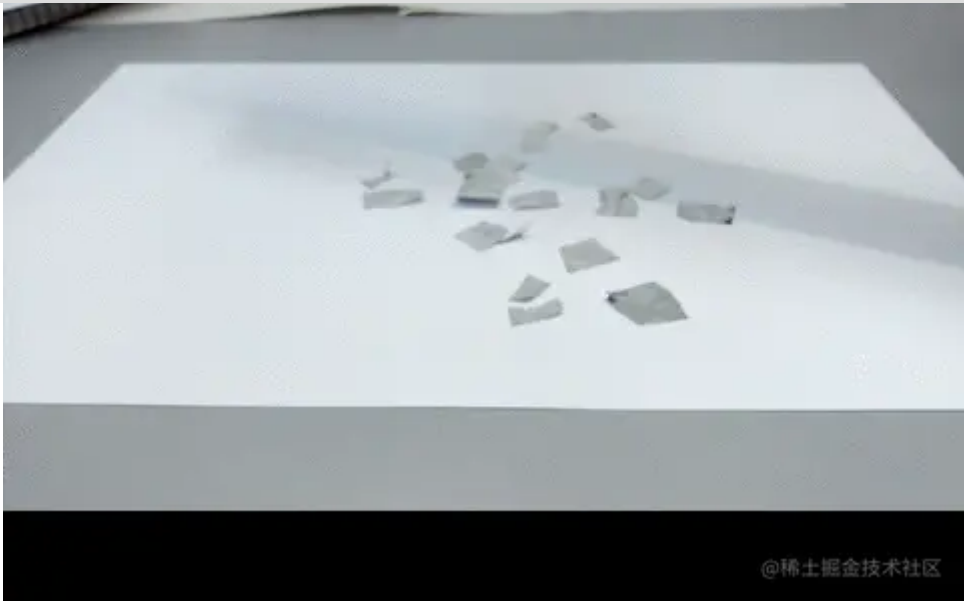
1. 第一站：读取文件（`fs.readFile`）；
2. 事件循环阻塞直到第二站：收到读事件，触发回调。回调内做两件事：
 - 写一个文件（`fs.writeFile`）；
 - 设置一个定时器（`setTimeout`）；
3. 事件循环阻塞直到第三站：收到定时器事件，触发回调，回调内
`console.log`；
4. 事件循环阻塞直到第四站：收到写事件，触发回调，即里面注释的那行。

我在公司内部实现了一个 Web-interoperable Runtime，叫 Hourai.js（蓬莱.js），用于 FaaS。它是一个基于 V8 的 JavaScript 运行时，自然也就有“事件循环”这个机制。内部的这个运行时事件循环用的并不是 libuv，而是一个自研的库，这个库里面有一个概念，从别的线程中做完一个事情，最终把这个事件通知给主事件循环，让主事件循环在下一个 tick 的时候能收到这个通知，并“线程安全”地处理这个事件结果。这个概念在 libuv 里面具象化出来就是 `uv_async_t`，但我在我的事件循环库中给这个概念起了一个更形象的名字：`static_wand`，即静电棒。

系统 I/O 是纯的 epoll 去监听文件描述符，把工作量交给 kernel。而事件循环则不止于此。比如不在 libuv 体系内的异步操作想融入 libuv，就需要靠 `uv_async_t` 了，也就是我提的“静电棒”。这是 libuv 中跨线程通信的机制。

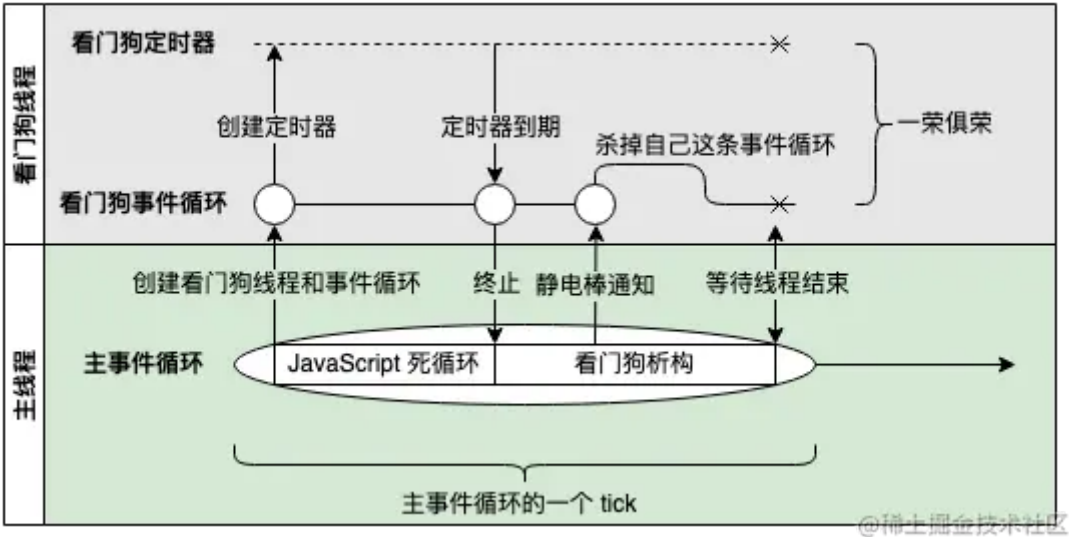
比如我要在一个新线程做一件事情，又想在新线程做完事情之后，能回归到主事件循环做之后的事，这时就需要用到 `uv_async_t` 了。它的作用就是允许你在一条非 libuv 事件循环的线程中通知 libuv 说我某个事情做完了，这样它的公交车站下一站就能以“这条线程做完了”为事件来处理逻辑了。

这个行为就跟“静电棒”一样，把周围任意地方（线程）上的纸屑，最终都按对应顺序吸附到静电棒上，安全地依次进行处理。



静电棒：图源网络

事实上，Node.js 中 `vm` 模块执行的超时就借助了 `uv_async_t` 的能力。因为 `vm` 在执行用户代码的时候，是在主事件循环上执行的，这个时候，只有一个上帝视角的线程才能在超时的时候终止它的执行。这个上帝视角在 Node.js 中被称作 `Watchdog`，看门狗。看门狗在其析构函数中，就是通过 `uv_async_t` 来终止看门狗自身的事件循环。就像这样：



上图中的“静电棒通知”靠的就是 `uv_async_t` 了。能看到，我们是在看门狗线程和主事件循环线程中来回拉扯，那些箭头就是静电的吸力，节点则是纸屑。



当然不好，既然是两条线程，在彼此不知道彼此状态的情况下，去操作另一个线程持有的对象，或者另一个线程正在处理的对象是很危险的，即线程不安全。谁知道那条事件循环是不是正在执行什么关键的操作，比如释放内存、新建内存之类的。直接就去杀掉事件循环，去终止线程，会导致状态不可知。所以要由主事件循环去发一个事件通知说要把自己杀了，看门狗事件循环自然会在死循环阻塞等通知的时候接到这个通知，然后它就可以在自己的安全领域内把自己杀掉了，杀掉后，看门狗的线程自然就跳出了事件循环，线程自然就结束了。而主事件循环内只需静静等待那条线程结束就可安全地做后续事情了。

引申：Node.js 究竟是单线程还是多线程的？

网上经常会有这样的问题出现，Node.js 究竟是单线程还是多线程的？其实不止是 Node.js，对于浏览器上的 JavaScript，大家都会有这样的疑问。如果是单线程的话，异步 I/O 什么的怎么做？Watchdog 怎么做？.....当然，Node.js 后来出的 `worker_thread` 不在我们的讨论范围之内。

我之前在知乎上回答过一个[这类问题](#)：**单线程是对你而言。对底层可不是，只不过其他线程对你不开放的。**

相信大家在看了本节内容后，心中自然就有答案了。

留个代码留一嘴

既然事件循环是个死循环，伪代码也给了。那在 Node.js 里面到底是怎么写的呢？每个版本的 Node.js 多多少少都有些差异，但大差不差。这里我用当下的最新 Stable 版本 Node.js v18.12.1 来贴代码吧。

Node.js v18.12.1 的事件循环相关代码在 `src/api/embed_helpers.cc` 中：

```
do {  
  if (env->is_stopping()) break;  
  uv_run(env->event_loop(), UV_RUN_DEFAULT);  
  if (env->is_stopping()) break;  
  
  platform->DrainTasks(isolate);  
}
```

C++



```
if (EmitProcessBeforeExit(env).IsNothing())  
    break;  
  
{  
    HandleScope handle_scope(isolate);  
    if (env->RunSnapshotSerializeCallback().IsEmpty()) {  
        break;  
    }  
}  
  
// Emit `beforeExit` if the loop became alive either after emitting  
// event, or after running some callbacks.  
more = uv_loop_alive(env->event_loop());  
} while (more == true && !env->is_stopping());
```



@稀土掘金技术社区

留言

输入评论 (Enter换行, Ctrl + Enter发送)

发表评论

Pandy_or2  



前端 @ Heighliner 2小时前

暗黑模式下图片显示有问题

 点赞  回复芝士加  

高级前端开发工程师 6小时前

等事件是不是可以理解是在等事件回调

 点赞  回复慢功夫   

前端工程师 1天前

静电棒的比喻看得有点懵哈哈

 1  回复洪布斯  前端开发 3天前

“引申：Node.js 究竟是单线程还是多线程的？”这段内容的知乎外链 404

 点赞  1

死月 (作者) 2天前

修好了

 点赞  回复Tusi     4天前

循环里等待事件驱动是不是更容易理解

 点赞  1

死月 (作者) 4天前

是等待事件，这整个过程可以被称为事件驱动。事件有定时事件和 io 事件

 1  回复杀生丸    前端开发工程师 4天前

看着还是有点懵，跟eventloop那种有什么区别

 点赞  8

死月 (作者) 4天前

eventloop 翻译过来不就是事件循环吗 🤔

 点赞  回复



“eventloop 翻译过来不就是事件循环吗 🤔”

👍 点赞 💬 回复



死月（作者） 回复 杀生丸 4天前

就是中文名跟英文名的区别？ 🤔

“嗯我是想问跟这个有什么区别”

👍 点赞 💬 回复



杀生丸 🗨️ 回复 死月 4天前

😂 我的意思是这跟浏览器的eventloop事件区别在哪？感觉这个有点没说清楚，也可能是因为我是小白

“就是中文名跟英文名的区别？ 🤔”

👍 点赞 💬 回复



死月（作者） 回复 杀生丸 4天前

没有本质区别，实现不一样而已，浏览器也没有直接与系统级 I/O 通信。浏览器的事件循环也是一个死循环，然后通过各种时间去驱动。

“😂 我的意思是这跟浏览器的eventloop事件区别在哪？感觉这个有点...”

👍 点赞 💬 回复



杀生丸 🗨️ 回复 死月 4天前

嗯嗯eventloop的宏任务微任务倒是理解了，这个死循环怎么理解？

“没有本质区别，实现不一样而已，浏览器也没有直接与系统级 I/O 通信...”

👍 点赞 💬 回复



死月（作者） 回复 杀生丸 4天前

我在另一个评论里回答了，你可以看看，死循环就是在每次循环里等待事件。

“嗯嗯eventloop的宏任务微任务倒是理解了，这个死循环怎么理解？”

👍 1 💬 回复



杀生丸 🗨️ 回复 死月 4天前

嗯嗯好的

“我在另一个评论里回答了，你可以看看，死循环就是在每次循环里等待...”

👍 点赞 💬 回复



js在设计上是单线程的，底层是多线程，但不对你开放。为了能够让你写出并发的逻辑代码，需要一个并发模型，事件循环就是一种并发模型，本质上是死循环，一直在获取任务。

有一点不明白，文中提到“事件循环是一种并发模型”，而在资料中经常看到一句话“js有一个基于事件循环的并发模型”。我理解是并发模型是一种抽象，而事件循环是实现，不知道这样理解对不对？

👍 点赞 💬 3



死月（作者） 4天前

并发模型是一种模型，是一个概念或者理论。事件循环是一种机制。

👍 1 💬 回复



大Y 回复 死月 4天前

百科查阅到：

模型：是通过主观意识借助实体或者虚拟表现，构成客观阐述形态结构的一种表达目的的物件。

机制：是指各要素之间的结构关系和运行方式。

我理解两者不属于一个维度，故不存在完全的互相概括，更形象的关系应该是两者辅助论证。

“并发模型是一种模型，是一个概念或者理论。事件循环是一种机制。”

👍 1 💬 回复



alkiad 4天前

我的理解是，并发是一种实际情况，事件循环是一种并发的处理方式。

本质上并发就是相对于并行而言的：同时有多个事件需要执行，但是主程同一时间只能处理一个事件。

为了完成并发，nodejs使用了事件循环。

👍 1 💬 回复