04_coderwhy前端八股文（四） - 事件循环和V8引擎

01-什么是浏览器的事件循环机制，包括它是如何处理异步操作的。

浏览器的事件循环是一个在JavaScript引擎和渲染引擎之间协调工作的机制。因为JavaScript是单线程的，所以所有需要被执行的操作都需要通过一定的机制来协调它们有序的进行。

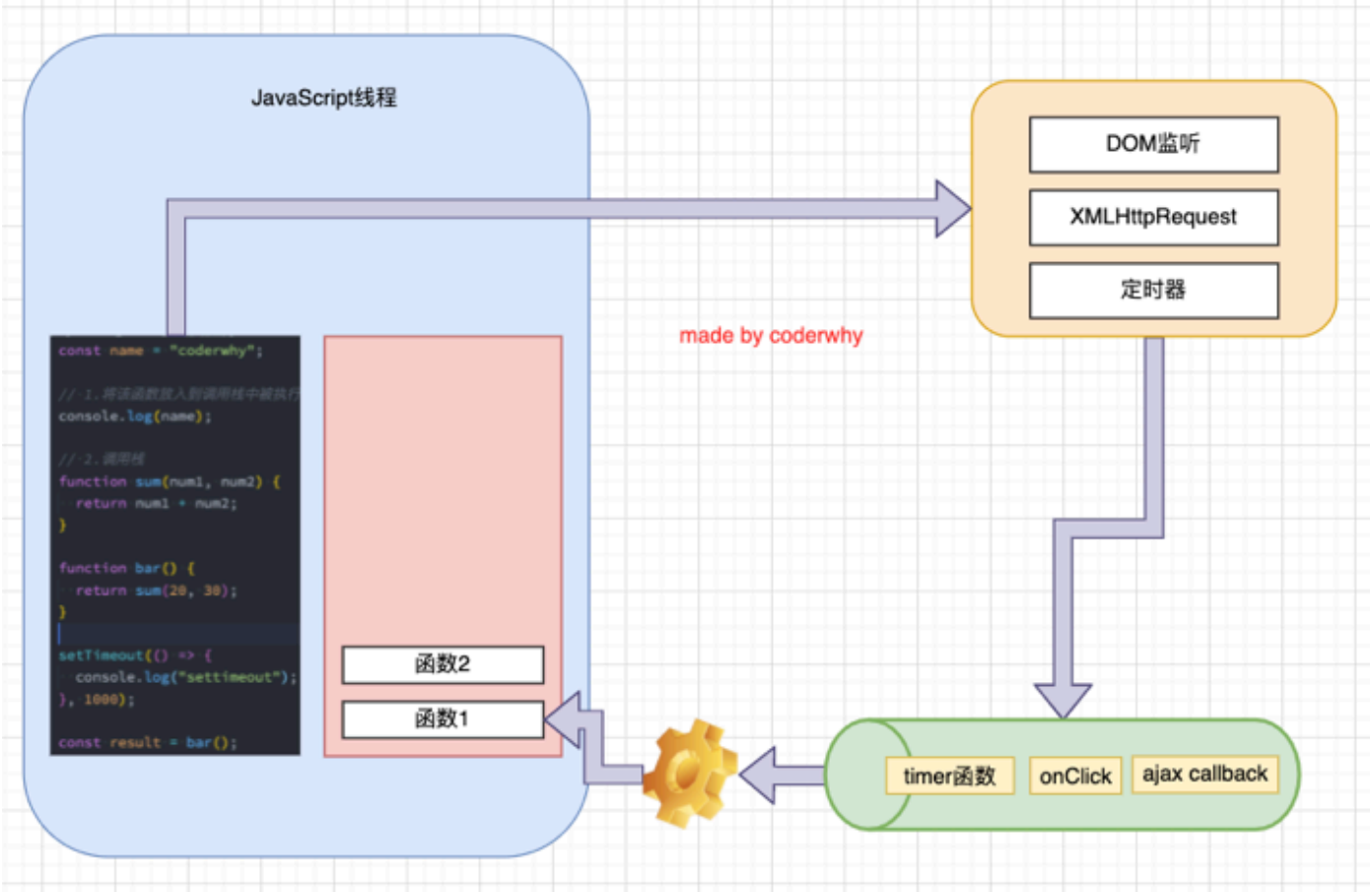
- 它的主要任务是监视调用栈（Call Stack）和任务队列（Task Queue）。
- 当调用栈为空时，事件循环会从任务队列中取出任务执行。

1. **调用栈（Call Stack）**
    - JavaScript是单线程的，调用栈是一个后进先出（LIFO）的数据结构，用于存储在程序执行过程中创建的所有执行上下文（Execution Contexts）。每当函数被调用时，它的执行上下文就会被推入栈中。函数执行完毕后，其上下文会从栈中弹出。
2. **任务队列（Task Queue）**
    - 任务队列是一种先进先出（FIFO）的数据结构，用于存储待处理的事件。这些事件可能包括用户交互事件（如点击、滚动等）、网络请求完成、定时器到期等。

事件循环处理异步操作

浏览器的事件循环通过以下步骤处理异步操作：

1. **执行全局脚本**：加载页面时，浏览器会首先执行全局脚本。
2. **宏任务和微任务**：
    - **宏任务**（MacroTasks）：包括脚本（script）、setTimeout、setInterval、I/O、UI rendering 等。
    - **微任务**（MicroTasks）：包括 Promise.then、MutationObserver、process.nextTick（仅在Node.js中）等。
3. **事件循环的周期**：
    - 执行当前宏任务。
    - 执行完当前宏任务后，检查并执行所有微任务。在微任务执行期间产生的新的微任务也会被连续执行，直到微任务队列清空。
    - 渲染更新界面（如果有必要）。
    - 请求下一个宏任务，重复上述过程。



02-什么是宏任务，什么是微任务？并解释一下它们在事件循环中的角色和区别?

在浏览器的事件循环中，宏任务和微任务是两类不同的任务，它们在异步操作处理上具有不同的优先级和执行时机。

宏任务（MacroTasks）

宏任务是一个比较大的任务单位，可以看作是一个独立的工作单元。

- 当一个宏任务执行完毕后，浏览器可以在两个宏任务之间进行页面渲染或处理其他事务（比如执行微任务）。

常见的宏任务包括:

- 完整的脚本（如一个`<script>`标签）
- `setTimeout`
- `setInterval`
- I/O操作（浏览器中的Ajax、Fetch，Node中的文件系统、网络请求、数据库交互）
- UI交互事件
- `setImmediate`（在Node.js中）
- 等等...

微任务（MicroTasks）

微任务通常是在当前宏任务完成后立即执行的小任务，它们的执行优先级高于宏任务。

- 微任务的执行会在下一个宏任务开始前完成，即在当前宏任务和下一个宏任务之间。

常见的微任务包括：

- `Promise.then`（Promise的回调）
- `Promise.catch`和`Promise.finally`
- `MutationObserver`（监视DOM变更的API）
- `process.nextTick`（仅在Node.js中）
- `queueMicrotask`（显示创建微任务的API）

宏任务和微任务区别

执行顺序：

- 事件循环在执行宏任务队列中的一个宏任务后，会查看微任务队列。如果微任务队列中有任务，事件循环会连续执行所有微任务直到微任务队列为空。
- 宏任务的执行可能触发更多的微任务，而这些微任务会在任何新的宏任务之前执行，确保微任务能够在渲染前或下一个宏任务之前快速响应。

用途不同：

- 由于微任务具有较高的执行优先级，它们适合用于需要尽快执行的小任务，例如处理异步的状态更新。
- 宏任务适合用于分割较大的、需要较长时间执行的任务，以避免阻塞UI更新或其他高优先级的操作。

03-比较Node.js和浏览器中的事件循环机制，指出它们的主要差异。

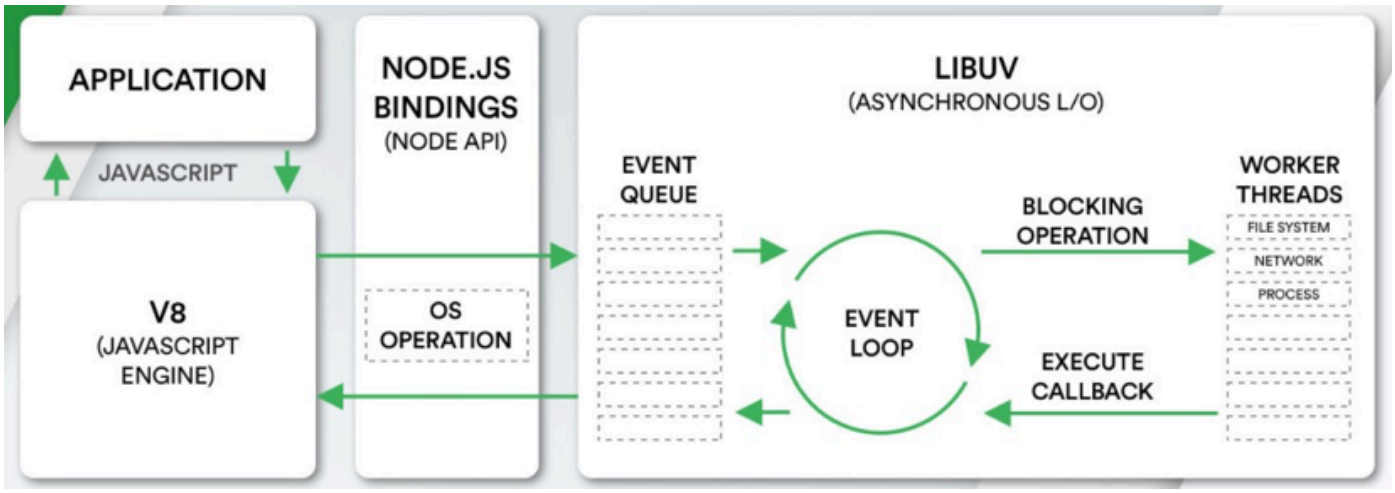
Node事件循环介绍

浏览器中的EventLoop是根据HTML5定义的规范来实现的，不同的浏览器可能会有不同的实现。

而Node中的事件循环是由是由libuv实现的，这是一个处理异步事件的C库。

- libuv是一个多平台的专注于异步IO的库，它最初是为Node开发的，但是现在也被使用到Luvit、Julia、pyuv等其他地方；

这里我们来给出一个Node的架构图：



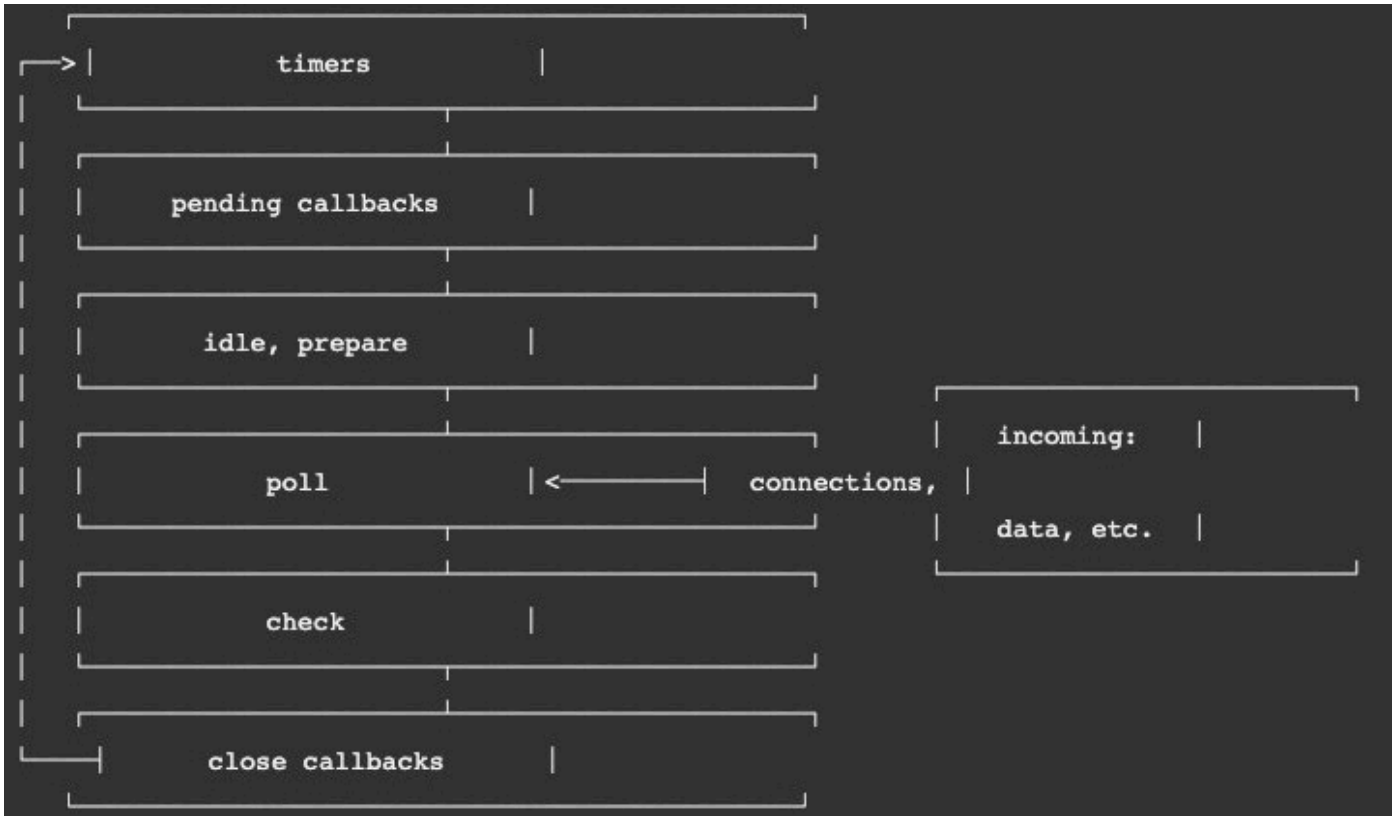
这里我们来给出一个Node的架构图：

- 我们会发现libuv中主要维护了一个EventLoop和worker threads（线程池）；
- EventLoop负责调用系统的一些其他操作：文件的IO、Network、child-processes等

Node事件循环阶段

Node.js的事件循环包含几个主要阶段，每个阶段都有自己的特定类型的任务：

我们来看一下官方给出的图片：



对每个阶段进行详细的解释：

- **timers**：这一阶段执行setTimeout和setInterval的回调函数。

- **Pending Callbacks**：executes I/O callbacks deferred to the next loop iteration（官方的解释）
  - 这意味着在这个阶段，Node.js处理一些上一轮循环中未完成的I/O任务。
  - 具体来说，这些是一些被推迟到下一个事件循环迭代的回调，通常是由于某些操作无法在它们被调度的那一轮事件循环中完成。
  - 比如操作系统在连接TCP时，接收到ECONNREFUSED（连接被拒绝）。
- **idle, prepare**：只用于系统内部调用。
- **poll**：检索新的 I/O 事件；执行与 I/O 相关的回调。
  - **检索新的I/O事件**：这一部分，libuv负责检查是否有I/O操作（如文件读写、网络通信）完成，并准备好了相应的回调函数。
  - **执行I/O相关的回调**：几乎所有类型的I/O回调都会在这里执行，除了那些特别由`timers`和`setImmediate`安排的回调以及某些关闭回调（`close callbacks`）。
- **check**：`setImmediate()` 的回调在这个阶段执行。
- **close callbacks**：如 `socket.on('close', ...)` 这样的回调在这里执行。

Node宏任务微任务

我们会发现从一次事件循环的Tick来说，Node的事件循环更复杂，它也分为微任务和宏任务：

- 宏任务（macrotask）：setTimeout、setInterval、IO事件、setImmediate、close事件；
- 微任务（microtask）：Promise的then回调、process.nextTick、queueMicrotask；

但是，Node中的事件循环中微任务队列划分的会更加精细：

- next tick queue：process.nextTick；
- other queue：Promise的then回调、queueMicrotask；

那么它们的整体执行时机是怎么样的呢?

- **调用栈执行**：Node.js 首先执行全局脚本或模块中的同步代码。这些代码在调用栈中执行，直到栈被清空。
- **处理 `process.nextTick()` 队列**：一旦调用栈为空，Node.js 会首先处理 `process.nextTick()` 队列中的所有回调。这确保了任何在同步执行期间通过 `process.nextTick()` 安排的回调都将在进入任何其他阶段之前执行。
- **处理其他微任务**：处理完 `process.nextTick()` 队列后，Node.js 会处理 Promise 微任务队列。这些微任务包括由 `Promise.then()`、`Promise.catch()` 或 `Promise.finally()` 安排的回调。

开始事件循环的各个阶段：

- **timers阶段**：处理 `setTimeout()` 和 `setInterval()` 回调。
- **I/O 回调阶段**：处理大多数类型的I/O相关回调。
- **poll阶段**：等待新的I/O事件，处理poll队列中的事件。
- **check阶段**：处理 `setImmediate()` 回调。
- **close回调阶段**：处理如 `socket.on('close', ...)` 的回调。

这里有一个特别的Node处理：微任务在事件循环过程中的处理

- 在事件循环的任何阶段之间，以及在上述每个阶段内部的任何单个任务后。

- Node.js 会再次处理 `process.nextTick()` 队列和 Promise 微任务队列。
- 这确保了在事件循环的任何时刻，微任务都可以优先并迅速地被处理。

04-描述process.nextTick在Node.js中事件循环的执行顺序，以及其与微任务的关系。

在 Node.js 中，`process.nextTick()` 是一个在事件循环的各个阶段之间允许开发者插入操作的功能：

- 其特点是具有极高的优先级，可以在当前操作完成后、任何进一步的I/O事件（包括由事件循环管理的其他微任务）处理之前执行。

`process.nextTick()` 的执行顺序：

1. **调用栈清空**：Node.js 首先执行完当前的调用栈中的所有同步代码。
2. **执行 `process.nextTick()` 队列**：一旦调用栈为空，Node.js 会检查 `process.nextTick()` 队列。
    - 如果队列中有任务，Node.js 会执行这些任务，即使在当前事件循环的迭代中有其他微任务或宏任务排队等待。
3. **处理其他微任务**：在 `process.nextTick()` 队列清空之后，Node.js 会处理由 Promises 等产生的微任务队列。
4. **继续事件循环**：处理完所有微任务后，Node.js 会继续进行到事件循环的下一个阶段（例如 timers、I/O callbacks、poll 等）。

与微任务的关系

- **优先级**：`process.nextTick()` 创建的任务和 Promise 是不同的，但它们是一种微任务，并且在所有微任务中具有最高的执行优先级。这意味着 `process.nextTick()` 的回调总是在其他微任务（例如 Promise 回调）之前执行。
- **微任务队列**：在任何事件循环阶段或宏任务之间，以及在宏任务内部可能触发的任何点，Node.js 都可能执行 `process.nextTick()`。执行完这些任务后，才会处理 Promise 微任务队列。

`process.nextTick()` 的命名在 Node.js 社区中曾经引起过一些讨论，因为它可能会导致一些误解。

代码的测试：

```
console.log('Start of script');

setTimeout(() => {
  console.log('First setTimeout');

  queueMicrotask(() => {
    console.log("queueMicrotask execution")
  })

  process.nextTick(() => {
    console.log('nextTick execution');
  });
}, 0);
```

```
14
15 setTimeout(() => {
16   console.log('Second setTimeout');
17 }, 0);
18
19 console.log('End of script');
```