21.事件循环（Event Loop）完整解析

1. 核心概念 (Core Concept)

事件循环（Event Loop）是 JavaScript 执行异步操作的机制。它协调浏览器（或 Node.js 环境）如何处理任务队列（包含宏任务和微任务），并在主线程空闲时执行这些任务，从而实现非阻塞的 I/O 操作和响应性用户界面。

2. 为什么需要它？ (The "Why")

- **非阻塞 I/O:** JavaScript 是单线程的，没有事件循环，像网络请求或文件读写这样的耗时操作会完全阻塞主线程，导致程序卡死。事件循环允许这些操作在后台进行，完成后再通知主线程处理结果。
- **用户界面响应性:** 在浏览器环境中，DOM 操作、事件监听等都依赖主线程。事件循环确保耗时脚本不会长时间锁定主线程，从而保证用户界面的流畅和响应。
- **异步编程模型支持:** Promise，`async/await`，`setTimeout`，`fetch` 等异步 API 的实现基础就是事件循环。

3. API 与用法 (API & Usage)

事件循环本身是一个核心概念和运行时机制，并非直接对外暴露的 API。但我们可以通过理解以下与事件循环紧密相关的 API 来观察其行为：

- **宏任务 (Macro tasks):**
  - `setTimeout(callback, delay)`: 在指定的延迟后将 callback 函数放入宏任务队列。
  - `setInterval(callback, delay)`: 每隔指定的延迟将 callback 函数放入宏任务队列。
  - `setImmediate(callback)` (Node.js 特有): 尽快将 callback 函数放入 check 阶段任务队列。
  - UI 渲染事件 (如绘制)。
  - I/O 操作完成的回调 (如读取文件完成)。
- **微任务 (Micro tasks):**
  - `Promise.prototype.then(onFulfilled, onRejected)`: 将 onFulfilled/onRejected 回调放入微任务队列。
  - `Promise.prototype.catch(onRejected)`: 同上。
  - `Promise.prototype.finally(onFinally)`: 同上。
  - `queueMicrotask(callback)`: 直接将 callback 函数放入微任务队列。
  - `MutationObserver` 的回调。
  - `process.nextTick(callback)` (Node.js 特有): 将 callback 放入 nextTick 队列，优先级高于微任务。

代码示例 (浏览器环境):

```
console.log('1. Start');

setTimeout(function() {
  console.log('4. setTimeout callback (Macro task)');
}, 0); // 尽管延迟为0，但它仍然是一个宏任务

Promise.resolve('Promise resolved').then(function(value) {
  console.log('3. Promise resolved callback (Micro task):', value);
});

console.log('2. End');

// 预期输出顺序（通常情况下）:
// 1. Start
// 2. End
// 3. Promise resolved callback (Micro task): Promise resolved
// 4. setTimeout callback (Macro task)
```

解释:

1. `console.log('1. Start')` 同步执行。
2. `setTimeout` 的回调被放入宏任务队列。
3. `Promise.resolve().then` 的回调被放入微任务队列。
4. `console.log('2. End')` 同步执行。
5. 主线程同步代码执行完毕，进入事件循环。
6. 事件循环检查微任务队列，发现 `Promise` 的回调，执行 `console.log('3...')`。
7. 微任务队列清空。
8. 事件循环检查宏任务队列，发现 `setTimeout` 的回调，执行 `console.log('4...')`。
9. 进入下一次事件循环迭代。

4. 关键注意事项 (Key Considerations)

- **同步代码优先:** 任何异步任务（宏任务或微任务）都必须等待当前正在执行的同步代码块完全执行完毕后才能有机会执行。
- **微任务优先于宏任务:** 在每一个事件循环迭代中，当同步代码执行完成后，事件循环会先清空所有可用的微任务，然后再执行下一个宏任务队列中的第一个任务。
- **Node.js 与浏览器环境差异:** Node.js 的事件循环比浏览器更复杂，包含了不同的阶段（timers, pending callbacks, idle/prepare, poll, check, close callbacks），以及 `process.nextTick` 队列，它的优先级高于微任务。理解这些差异对于 Node.js 开发至关重要。
- **宏任务之间的间隔:** 宏任务队列中的任务是逐个执行的。执行完一个宏任务后，会检查并清空微任务队列，然后才可能执行下一个宏任务。这确保了 UI 渲染（也是一种宏任务）有机会在连续的耗时宏任务之间发生。

5. 参考资料 (References)

- **MDN Web Docs - Concurrency model and Event Loop:** https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop
- **Philip Roberts: What the heck is the event loop anyway? (视频):** https://www.youtube.com/watch?v=8aGhZQkoFbQ (业界公认的优秀讲解视频)
- **Node.js Docs - The Node.js Event Loop, Timers, and process.nextTick():** https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/
- **HTML Living Standard - Event loops:** https://html.spec.whatwg.org/multipage/webappapis.html#event-loops (官方规范文档)