JavaScript 中的事件循环机制

叶家炜

Apr 04, 2025

JavaScript 的单线程特性决定了它在处理异步任务时必须依赖事件循环机制。这一机制通过协调调用栈、内存堆和任务队列，实现了非阻塞的异步编程模型。例如，当发起一个网络请求时，浏览器不会等待响应返回，而是继续执行后续代码，待数据就绪后再通过回调函数处理结果。这种设计避免了主线程的阻塞，但也带来了执行顺序的复杂性。本文将深入剖析事件循环的核心原理，并探讨其在浏览器与 Node.js 中的差异及实践中的优化技巧。

1　事件循环的核心原理

1.1　运行时环境的三要素

JavaScript 的运行时环境由三部分组成：调用栈（**Call Stack**）、内存堆（**Heap**）和任务队列（**Task Queue**）。调用栈用于追踪函数的执行顺序，每个函数调用会形成一个栈帧；内存堆负责管理对象的动态内存分配；任务队列则存储待处理的异步任务回调。
事件循环的核心逻辑可简化为以下伪代码：

```
while (true) {
  if (调用栈为空) {
    const 任务 = 任务队列.取出下一个任务();
    执行(任务);
  }
}
```

1.2　同步优先与异步分层

同步代码始终优先执行，例如：

```
console.log('A');
setTimeout(() => console.log('B'), 0);
console.log('C');
// 输出顺序：A → C → B
```

`setTimeout` 的回调被推入任务队列，等待调用栈清空后执行。异步任务进一步分为宏任务（如 `setTimeout`）和微任务（如 `Promise`），微任务在每轮事件循环的末尾优先执行。

2 宏任务与微任务的执行规则

2.1 分类与优先级

宏任务包括：

1. setTimeout/setInterval
2. I/O 操作
3. UI 渲染（浏览器）
4. setImmediate（Node.js）

微任务包括：

1. Promise.then/async await
2. MutationObserver
3. process.nextTick（Node.js，优先级高于普通微任务）

2.2 黄金执行顺序

每轮事件循环处理一个宏任务后，会清空所有微任务队列。例如：

```
setTimeout(() => console.log('宏任务1'), 0);
Promise.resolve().then(() => console.log('微任务1'));
setTimeout(() => {
  console.log('宏任务2');
  Promise.resolve().then(() => console.log('微任务2'));
}, 0);
// 输出顺序：微任务1 → 宏任务1 → 微任务2 → 宏任务2
```

第一轮循环执行主线程代码（视为宏任务），触发微任务 微任务 1；随后处理 宏任务 1；下一轮处理 宏任务 2 时，其内部的 微任务 2 会立即执行。

3 浏览器与 Node.js 的差异

3.1 浏览器的事件循环模型

浏览器的事件循环与渲染管线紧密耦合。在一次循环中，可能包含以下步骤：

- 执行一个宏任务
- 清空微任务队列
- 执行 UI 渲染（如果需要）
- 执行 requestAnimationFrame 回调

这使得频繁的微任务可能延迟渲染，例如：

```
function 阻塞渲染() {
  Promise.resolve().then(阻塞渲染);
}
阻塞渲染();
// UI 更新会被无限延迟
```

3.2 Node.js 的六阶段模型

Node.js 基于 libuv 库实现事件循环，分为六个阶段：

- **Timers**：执行 `setTimeout/setInterval` 回调
- **Pending Callbacks**：处理系统错误等挂起回调
- **Idle/Prepare**：内部使用
- **Poll**：检索新的 I/O 事件
- **Check**：执行 `setImmediate` 回调
- **Close**：处理关闭事件（如 `socket.on('close')`）

以下代码演示了 Node.js 中 `setImmediate` 与 `setTimeout` 的优先级：

```
setTimeout(() => console.log('setTimeout'), 0);
setImmediate(() => console.log('setImmediate'));
// 输出顺序可能不确定，取决于事件循环启动时间
```

4 异步编程的最佳实践

4.1 从回调地狱到 async/await

传统回调模式容易引发嵌套问题：

```
fs.readFile('A.txt', (err, dataA) => {
  fs.readFile('B.txt', (err, dataB) => {
    // 回调地狱
  });
});
```

使用 `Promise` 和 `async/await` 可扁平化代码：

```
async function 读取文件() {
  const dataA = await fs.promises.readFile('A.txt');
  const dataB = await fs.promises.readFile('B.txt');
  return [dataA, dataB];
}
```

4.2 性能优化策略

- 拆分长任务：将耗时操作分解为多个微任务

```
function 分片处理() {
  let i = 0;
  function 下一帧() {
    while (i < 1000 && 未超时) {
      // 处理数据
      i++;
    }
    if (i < 1000) {
      setTimeout(下一帧, 0);
    }
  }
  下一帧();
}
```

- 使用 **Web Workers**：将 CPU 密集型任务转移到后台线程

```
const worker = new Worker('task.js');
worker.postMessage(data);
worker.onmessage = (e) => console.log(e.data);
```

5 案例解析

5.1 页面卡顿优化

假设一个页面需要渲染 10,000 条数据，直接操作 DOM 会导致主线程阻塞：

```
// 错误示例
数据列表.forEach(条目 => {
  const div = document.createElement('div');
  div.textContent = 条目;
  document.body.appendChild(div);
});
```

优化方案：使用 `requestIdleCallback` 分批次处理

```
function 分片渲染(数据, 索引 = 0) {
  requestIdleCallback((空闲时间) => {
    while (索引 < 数据.length && 空闲时间.剩余时间() > 0) {
```

```
    创建元素(数据[索引]);
    索引++;
  }
  if (索引 < 数据.length) {
    分片渲染(数据, 索引);
  }
 });
}
```

事件循环机制是 JavaScript 异步编程的基石。理解宏任务与微任务的执行顺序、掌握浏览器与 Node.js 的差异，能够帮助开发者编写高效可靠的代码。随着 WebAssembly 和 Deno 等新技术的发展，异步模型仍在持续演进，但核心原理始终是构建复杂应用的指南针。