28. “异步题大汇总”：输出顺序、陷阱解析

1. 核心概念 (Core Concept)

前端异步题的核心考察点在于对 JavaScript **事件循环 (Event Loop)** 机制的理解，特别是任务队列 (Task Queue / MacroTask Queue) 和微任务队列 (MicroTask Queue) 的执行顺序和时机。这些题目通常结合 `setTimeout`、`setInterval`、`Promise`、`async/await`、`process.nextTick` (Node.js)、`queueMicrotask` 等异步 API，要求判断代码的实际执行顺序和输出结果。

2. 为什么需要它？ (The "Why")

- **理解非阻塞性 (Understanding Non-blocking):** 掌握异步执行机制是理解 JavaScript 如何在单线程环境下处理耗时操作（如网络请求、定时器）而不阻塞主线程的关键。
- **精确定时与调度 (Precise Timing & Scheduling):** 正确使用异步 API 并在复杂场景下预测其行为，对于实现精确的定时逻辑、合理的资源加载顺序以及流畅的用户体验至关重要。
- **排查并发问题 (Debugging Concurrency Issues):** 异步代码的执行顺序不确定性是许多 bug 的根源。理解事件循环能帮助开发者更好地分析和解决这些问题。

3. API 与用法 (API & Usage)

核心涉及的异步 API 及其在事件循环中的典型归类：

- **宏任务 (MacroTasks):**
  - `setTimeout(callback, delay)`: 在指定延迟后将回调函数放入任务队列。`delay` 为 0 不意味着立即执行，而是在当前宏任务执行完毕后，下一次事件循环迭代中尽快执行，但仍需等待前面的宏任务。
  - `setInterval(callback, delay)`: 每隔指定延迟将回调函数放入任务队列。每次执行都取决于前一次任务的完成和延迟时间。
  - `setImmediate(callback)` (Node.js): 在当前事件循环迭代结束时执行，通常在 I/O 回调后执行，优先级高于 `setTimeout(..., 0)` 但低于微任务。
  - I/O 操作 (Node.js)
  - UI Rendering
- **微任务 (MicroTasks):**
  - `Promise.prototype.then(onFulfilled, onRejected)`: 当 Promise 状态改变时，将对应的回调函数放入微任务队列。
  - `Promise.prototype.catch(onRejected)`
  - `Promise.prototype.finally(onFinally)`
  - `process.nextTick(callback)` (Node.js): 将回调函数放在当前事件循环迭代的微任务队列最前面，优先于其他微任务执行。

- `queueMicrotask(callback)`：标准 API，用于显式地将回调放入微任务队列。
- `async/await`：`await` 会暂停 `async` 函数的执行，并将剩余部分（`await` 后面的代码）视为一个微任务。Promise 的解决或拒绝会触发 `then` / `catch` 微任务的执行。

事件循环基本流程（简化）：

1. 执行当前宏任务（如执行主脚本）。
2. 检查微任务队列，执行所有微任务直到队列清空。
3. 如有 UI 渲染或其他任务，执行之。
4. 检查任务队列（宏任务队列），取出队列中的第一个宏任务执行。
5. 返回步骤 2，循环往复。

经典代码示例 (来自 MDN 和 Promise/A+ 规范意译)：

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

setTimeout(() => {
  console.log('setTimeout 1');
  Promise.resolve().then(() => {
    console.log('Promise in setTimeout');
  });
}, 0);

Promise.resolve().then(() => {
  console.log('Promise 1');
  setTimeout(() => {
    console.log('setTimeout in Promise');
  }, 0);
}).then(() => {
  console.log('Promise 2');
});

console.log('End');

// 预期输出（浏览器环境，Node.js 11+）：
// Start
// End
// Promise 1
// Promise 2
// setTimeout 1
// Promise in setTimeout
// setTimeout in Promise
```

解释：

1. 执行主脚本，打印 'Start' 和 'End'。这是第一个宏任务。

2. `setTimeout` 的回调被放入任务队列。
3. 第一个 `Promise.then` 的回调被放入微任务队列。
4. 第一个宏任务执行完毕。
5. 检查微任务队列，执行第一个微任务，打印 'Promise 1'。在此微任务中，又创建一个新的 `setTimeout`，其回调被放入任务队列。
6. 微任务队列非空，执行第二个微任务 (由 `.then().then()` 链式调用产生)，打印 'Promise 2'。
7. 微任务队列清空。
8. 进入下一个事件循环迭代，从任务队列中取出第一个宏任务（第一个 `setTimeout` 的回调），执行它，打印 'setTimeout 1'。
9. 在该宏任务中，创建一个 `Promise.resolve().then`，其回调被放入微任务队列。
10. 当前宏任务执行完毕。
11. 检查微任务队列，发现有新的微任务，执行它，打印 'Promise in setTimeout'。
12. 微任务队列清空。
13. 进入下一个事件循环迭代，从任务队列中取出下一个宏任务（在 Promise 回调中创建的 `setTimeout`），执行它，打印 'setTimeout in Promise'。
14. 任务队列和微任务队列都空了。

4. 关键注意事项 (Key Considerations)**

- **微任务优先于宏任务:** 在一个事件循环迭代中，当前宏任务执行完毕后，总是优先清空微任务队列，才会进入下一个宏任务。这是预测输出顺序最重要的规则。
- **`setTimeout(..., 0)` vs 微任务:** `setTimeout(..., 0)` 并不是立即执行，它只是将回调尽可能快地放入任务队列，等待当前及所有后续微任务执行完毕后，才能在下一个宏任务执行。微任务（如 `Promise.then`）在同一轮事件循环中执行，而 `setTimeout` 在下一轮或更远的宏任务中执行。
- **`async/await` 的糖衣:** `async/await` 是 Promise 和 Generator 的语法糖。`await` 关键字会暂停 `async` 函数的执行，并等待 Promise resolved/rejected。在等待期间，控制权交还给事件循环。当 Promise 解决时，`await` 后面的代码会作为微任务被重新调度执行。
- **Node.js 环境差异 (历史版本):** 在 Node.js 10 及更早版本中，`process.nextTick` 的优先级高于 Promise 微任务。Node.js 11+ 的行为与浏览器趋同，Promise 微任务和 `process.nextTick` 都属于同一微任务队列，但 `process.nextTick` 稍优先。`setImmediate` 则属于不同的 check 阶段，通常在 I/O 回调之后但在定时器之前执行。在前端面试通常以浏览器环境为主，但了解 Node.js 的差异也很重要。

5. 参考资料 (References)

- **MDN Web Docs: Concurrency model and Event Loop:** https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop
- **JavaScript Promise A+ specification:** https://promisesaplus.com/ (了解 Promise 的基本行为规范)

- **Node.js - The Event Loop, Timers, and process.nextTick():** https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick (Node.js 环境下的事件循环细节)
- **Tasks, microtasks, queues and schedules:** https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/ (Jake Archibald 的经典文章，详细解释任务和微任务)
