26. 实战：用 Promise 重写 setTimeout 任务队列

1. 核心概念 (Core Concept)

这个主题的核心是将基于传统回调函数的 `setTimeout` 异步操作，通过 `Promise` 这一更现代、更易于管理的异步编程模式进行封装和抽象，以实现更优雅的任务队列控制和管理。

2. 为什么需要它？ (The "Why")

1. **解决回调地狱 (Callback Hell):** 使用 `Promise` 可以将嵌套的回调函数展平，使得异步代码更易读、更易于维护。
2. **链式调用与错误处理 (Chaining & Error Handling):** `Promise` 提供的 `.then()` 和 `.catch()` 方法使得连续的异步操作可以链式调用，错误处理也更加集中和规范。
3. **更强大的异步模式 (Enhanced Async Patterns):** 基于 `Promise` 可以轻松实现 `Promise.all` （并行）、`Promise.race` （竞速）等更复杂的异步模式，便于构建更复杂的异步任务流。

3. API 与用法 (API & Usage)

该实战的核心是将 `setTimeout` 异步操作封装到一个返回 `Promise` 的函数中，然后利用 `Promise` 的特性构建任务队列。

核心封装函数示例：

```
/**
 * 封装 setTimeout 到 Promise
 * @param {number} delay - 延迟的毫秒数
 * @returns {Promise<void>}
 */
function delayPromise(delay) {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(); // 延迟时间到，Promise 状态变为 resolved
    }, delay);
  });
}

// 示例：使用 delayPromise 构建一个简单的顺序执行任务队列
async function runTaskQueue() {
  console.log('任务 1 开始...');
  await delayPromise(1000); // 等待 1 秒
  console.log('任务 1 结束.');

  console.log('任务 2 开始...');
```

```
  await delayPromise(500); // 等待 0.5 秒
  console.log('任务 2 结束.');

  console.log('任务 3 开始...');
  await delayPromise(1500); // 等待 1.5 秒
  console.log('任务 3 结束.');
}

// 执行任务队列
runTaskQueue();
```

使用 `Promise.resolve()` 和链式调用构建任务队列 (经典 Promise 风格):

虽然 `async/await` 是 `Promise` 的语法糖，更受欢迎，但理解经典的 `.then()` 链式调用也很重要。

```
function delayPromise(delay, taskMessage) {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log(taskMessage);
      resolve();
    }, delay);
  });
}

// 构建顺序任务队列
Promise.resolve() // 从一个 resolved 的 Promise 开始
  .then(() => delayPromise(1000, '完成任务 A after 1s'))
  .then(() => delayPromise(500, '完成任务 B after 0.5s'))
  .then(() => delayPromise(1500, '完成任务 C after 1.5s'))
  .catch(error => {
     console.error('执行过程中发生错误:', error); // 处理任何可能的错误
  });

console.log('任务队列开始执行 (但输出顺序取决于延迟)');
```

在这个例子中， `Promise.resolve()` 创建一个立即 resolved 的 Promise，后续的 `.then()` 调用会等待前一个 Promise resolved 后再执行，从而实现顺序执行。

4. 关键注意事项 (Key Considerations)

1. **错误处理:** 在封装的 `delayPromise` 中，如果 `setTimeout` 本身没有逻辑错误，Promise 不会 rejected。但在实际应用中，如果任务队列中的某个任务可能出错（如调用了其他失败的异步操作），确保在 `.then()` 链的最后或使用 `async/await` 的 `try...catch` 结构来统一处理错误。
2. **取消任务:** 原生的 `setTimeout` 返回一个 ID，可以通过 `clearTimeout` 取消。但 `Promise` 本身没有直接的取消机制（这是一个设计上的取舍）。要实现可取消的延迟

Promise 或任务队列，需要额外的逻辑，例如在 Promise 外部维护一个状态或使用第三方的可取消 Promise 库。

3. **任务间的状态传递:** 如果任务队列中的任务需要根据前一个任务的结果执行，可以在 `.then()` 的回调函数中接收上一个 Promise resolve 的值，并在自己的回调函数中返回新的值，以便链式传递。使用 `async/await` 则更直观，直接通过变量存储前一个 `await` 的结果。
4. **资源消耗:** 大量创建短延迟的 Promise 可能会带来一定的性能开销（尽管现代 JS 引擎优化得很好），但对于控制异步流程而言，其带来的代码清晰度通常是更重要的。

5. 参考资料 (References)

- **MDN Web Docs - Using Promises:** https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Using_promises (关于 Promise 的基础和用法)
- **MDN Web Docs - setTimeout():** https://developer.mozilla.org/zh-CN/docs/Web/API/setTimeout (关于 setTimeout 的基础)
- **MDN Web Docs - async function:** https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Statements/async_function (关于 async/await 的基础和用法)
- **JavaScript Promise 定时器 - ECMAScript 6 入门 (阮一峰):** https://es6.ruanyifeng.com/#docs/promise#Promise-定时器 (一个经典的 Promise 定时器示例)