20. setTimeout 和 setInterval 的陷阱

1. 核心概念 (Core Concept)

`setTimeout` 和 `setInterval` 是 JavaScript 中用于处理延迟或重复执行代码的两个核心函数。它们都利用了浏览器的事件循环机制来异步地调度函数的执行。

- `setTimeout(callback, delay)`: 在指定的延迟（毫秒）后执行一次给定的函数。
- `setInterval(callback, delay)`: 每隔指定的延迟（毫秒）重复执行一次给定的函数。

2. 为什么需要它？ (The "Why")

这两个函数是实现异步或定时任务的基础，它们：

- **避免阻塞主线程:** 将耗时的操作或未来的操作从同步执行流程中分离，防止 UI 冻结。
- **实现定时任务:** 如延时显示内容、动画、轮询等。
- **模拟高级异步模式:** 可以基于它们构建更复杂的异步控制逻辑（尽管现代 JS 已有 Promise, async/await 等）。

3. API 与用法 (API & Usage)

`setTimeout(callback, delay, ...args)`

- `callback`: 要执行的函数。
- `delay`: 延迟的毫秒数。如果省略，默认为 0。请注意，这不是精确的延迟，受事件循环影响。
- `...args` (可选): 传递给 `callback` 函数的额外参数。
- **返回值:** 一个数字 ID，可用于 `clearTimeout()` 取消延时。

```
function greet(name) {
  console.log('Hello, ' + name);
}

const timerId = setTimeout(greet, 2000, 'World'); // 2秒后执行
greet('World')

// 可以取消这个延时执行
// clearTimeout(timerId);
```

`setInterval(callback, delay, ...args)`

- `callback`: 要重复执行的函数。
- `delay`: 重复执行之间的时间间隔（毫秒）。同样受事件循环影响。

- `...args` (可选): 传递给 `callback` 函数的额外参数。
- **返回值:** 一个数字 ID，可用于 `clearInterval()` 取消重复。

```
let count = 0;
const intervalId = setInterval(() => {
  count++;
  console.log(`Interval fired ${count} times`);
  if (count >= 5) {
    clearInterval(intervalId); // 达到条件时停止重复
  }
}, 1000); // 每秒执行一次
```

4. 关键注意事项 (Key Considerations)

4.1. 延迟不是精确的 (Delay is Not Guaranteed)

- `delay` 参数指定的是将回调函数添加到消息队列的最小延迟时间。
- 实际执行时间取决于事件循环中当前是否有其他任务正在执行。如果主线程繁忙，回调函数会等待当前任务完成后才能执行，导致实际延迟大于或等于指定的 `delay`。
- 最小延迟：在大多数浏览器中，连续的 `setTimeout` / `setInterval` 调用（特别是嵌套的或大量并发的）可能有一个最小的延迟限制（通常为 4ms）。这是为了避免资源过度消耗。

4.2. `setInterval` 可能导致任务堆积 (Potential Task Queuing/Stack Up with `setInterval`)

- 如果 `setInterval` 的回调函数执行时间**大于**指定的 `delay` 时间，那么在回调函数尚未执行完毕时，下一个回调任务可能已经根据 `delay` 被添加到消息队列了。
- 这会导致消息队列中累积待执行的同一回调函数实例，当主线程空闲时，这些任务会接连执行，而非间隔 `delay` 时间执行，这可能导致程序行为异常或性能问题。
- **推荐替代方案:** 使用递归的 `setTimeout` 可以避免这个问题。在每次 `setTimeout` 的回调函数执行完毕后，再根据需要调用下一个 `setTimeout`。

```
// 避免 setInterval 堆积的递归 setTimeout 方法
function recursiveTimeout() {
  // 执行任务...
  console.log("Executing task...");

  // 模拟一个耗时操作 (> 100ms)
  let start = Date.now();
  while (Date.now() - start < 200);

  // 任务执行完成后再调度下一个
  setTimeout(recursiveTimeout, 100); // 永远保持100ms的间隔（在任务完成后）
}
```

```
// 注释掉，避免实际运行
// recursiveTimeout();
```

4.3. `this` 的指向问题 (`this` Context Issues)

- 原始的 `setTimeout` 和 `setInterval` 调用中的回调函数，如果在非箭头函数中使用 `function` 关键字定义，其内部的 `this` 默认指向全局对象 (在浏览器中通常是 `window`，在严格模式下是 `undefined`)，而不是调用它们的上下文对象。
  解决方案:
  使用箭头函数作为回调，箭头函数没有自己的 `this`，会捕获其定义时的上下文的 `this`。(推荐)
  使用 `bind()` 方法显式绑定 `this`。
  * 在外部保存 `this` 的引用 (如 `const self = this;`) 并在回调中使用 `self`。

```
class MyClass {
  constructor() {
    this.value = 'initial';
  }

  // 问题: 传统函数, this 指向 window 或 undefined
  methodWithProblem() {
    setTimeout(function() {
      // console.log(this.value); // Error: this 指向不对
    }, 100);
  }

  // 解决方案 1: 箭头函数 (推荐)
  methodWithArrow() {
    setTimeout(() => {
      console.log(this.value); // logs 'initial'
    }, 100);
  }

  // 解决方案 2: bind
  methodWithBind() {
    setTimeout(function() {
      console.log(this.value);
    }.bind(this), 100); // 绑定 this
  }
}

const instance = new MyClass();
// instance.methodWithProblem(); // 会有问题
instance.methodWithArrow();
instance.methodWithBind();
```

4.4. 取消定时器 (`clearTimeout` / `clearInterval`)

- 如果不显式取消，使用 `setInterval` 设置的定时器会一直运行，直到页面关闭。
- 即使使用 `setTimeout` 设置的延迟执行，如果回调函数在执行前不再需要，也应当使用 `clearTimeout` 取消，以避免不必要的资源消耗和潜在的副作用（例如，在组件卸载后仍在尝试更新 DOM）。
- 在使用 React 这样的框架时，在组件卸载 (`componentWillUnmount` 或 `useEffect` 的清理函数) 时清理定时器是防止内存泄漏和错误行为的常见最佳实践。

```
let intervalId = null;

function startFetching() {
  intervalId = setInterval(() => {
    console.log('Fetching data...');
    // ... fetch logic ...
  }, 5000);
}

function stopFetching() {
  if (intervalId !== null) {
    console.log('Stopping fetch interval');
    clearInterval(intervalId);
    intervalId = null; // 清理引用
  }
}

// 开始获取 (例如，在组件挂载时)
// startFetching();

// 停止获取 (例如，在组件卸载时)
// stopFetching();
```

5. 参考资料 (References)

- **MDN Web Docs:**
  - `setTimeout()`
  - `setInterval()`
  - Concurrency model and Event Loop
- **JavaScript Event Loop Explained:** (虽然不是官方文档，但许多权威技术博客深入解释了事件循环对定时器的影响)
  - Philip Roberts: What the heck is the event loop anyway? | JSConf EU
  - Tasks, microtasks, queues and schedules