47. JS 内存泄漏场景与排查

1. 核心概念 (Core Concept)

JavaScript 内存泄漏是指由于程序逻辑错误，那些本应被垃圾回收机制回收的对象仍被保留在内存中，无法释放，导致内存占用不断增长直到耗尽可用内存，影响应用性能甚至导致崩溃的现象。

2. 为什么需要它？ (The "Why")

- **性能问题:** 内存泄漏会导致应用程序响应变慢、卡顿，用户体验下降。
- **稳定性问题:** 严重的内存泄漏可能耗尽系统资源，导致浏览器崩溃或服务器宕机。
- **资源浪费:** 占用不必要的内存资源。

3. API 与用法 (API & Usage)

JavaScript 的内存管理是自动进行的，主要依赖垃圾回收机制（Garbage Collection, GC）。GC 的核心任务是找出不再被引用的对象并 K 释放其占用的内存。现代 V8 引擎采用分代垃圾回收（Generation Scavenging）和标记-清除（Mark-Sweep）/标记-整理（Mark-Compact）等多种算法。

尽管 GC 是自动的，但某些编程模式会“欺骗”GC，使其误以为某个对象仍然被需要，从而导致泄漏。以下是常见的内存泄漏场景及概念：

- **全局变量 (Accidental Global Variables):** 未声明的变量会自动添加到全局对象（浏览器中的 `window`，Node.js 中的 `global` 或 `globalThis`）上，即使函数执行完毕也不会被回收。

```
function foo() {
  // bar 是全局变量，本意是局部变量
  bar = 'some value';
}
foo(); // window.bar 现在存在
```

  避免: 始终使用 `let`，`const`，`var` 声明变量。使用严格模式（`'use strict';`）可以避免无意中创建全局变量。
- **被遗忘的定时器或回调 (Leaky Timers and Callbacks):** 使用 `setInterval` 或 `setTimeout` 设置的定时器，或一些事件监听器如果未在不再需要时清除，其内部的函数会持有外部作用域的引用，导致外部作用域内的变量无法被回收。

```
let element = document.getElementById('my-element');
let data = loadData(); // 假设data很大
```

```
setInterval(function() {
  if (element.parentNode === null) {
    // 当element从DOM移除后，定时器回调仍然持有element和data的引用
    console.log('Element removed from DOM, but timer is still running
and holding references.');
    // 正确做法：在此处清除定时器
    // clearInterval(this); // 需要保存定时器ID以便清除
  } else {
    // Do something with element and data
  }
}, 1000);

// --- 正确示例 ---
let timerId = setInterval(function() {
  if (element.parentNode === null) {
     console.log('Element removed from DOM, clearing timer.');
     clearInterval(timerId); // 使用保存的ID清除定时器
     element = null; // 可选：显式解除引用
     data = null;    // 可选：显式解除引用
  } else {
    // ...
  }
}, 1000);
```

避免: 在组件卸载或元素移除时，务必清除定时器（`clearInterval`，`clearTimeout`）和事件监听器（`removeEventListener`）。

- **闭包陷阱 (Closures):** 闭包允许内部函数访问外部函数的变量。如果闭包被长期持有（例如作为事件处理函数或长期存在的对象属性），并且它引用了外部函数作用域中的大量对象，这些对象将无法被回收。

```
function createModal() {
  let hugeData = loadHugeData(); // 假设载入了巨大的数据
  let element = document.getElementById('modal');

  // 这个闭包被作为事件处理函数，会长期持有 hugeData 和 element 的引用
  element.addEventListener('click', function handleClick() {
    console.log('Modal clicked');
    // Do something with hugeData
  });

  // 如果element被从DOM移除，但事件监听未被移除，则闭包和 hugeData 仍在内存中
}
createModal();
```

避免: 谨慎使用闭包，特别是当闭包需要持有大量数据时。在不需要时解除闭包的引用（如移除事件监听器），或设计时避免在长期存在的闭包中捕获大量数据。

- **脱离 DOM 的元素引用 (Out-of-DOM References):** 如果一个 DOM 元素从文档树中移除，但某个 JavaScript 变量仍然持有对该元素的引用，那么该元素及其子节点将不会被垃圾回收。

```
let elements = [];
let element = document.getElementById('myDiv');
elements.push(element); // 将元素添加到数组中

// ... 之后某个时候 ...
element.parentNode.removeChild(element); // 从DOM中移除了

// elements 数组仍然持有对已移除元素的引用，导致其无法被回收
console.log(elements.length); // 1
```

  避免: 在从 DOM 中移除元素后，显式解除对该元素的 JavaScript 变量引用（例如，将变量设为 `null`）是良好的实践， 특히对于长期持有引用的数组或对象。

内存泄漏排查工具: 浏览器开发者工具（如 Chrome DevTools）是排查内存泄漏的主要工具。

- **Performance 面板:** 记录一段时间内的性能，可以观察内存（JS Heap）的增长趋势。持续增长可能表明存在泄漏。
- **Memory 面板:**
  - **Heap snapshot (堆快照):** 捕获当前时刻的 JavaScript 堆内存状态。通过对比在执行特定操作（疑似导致泄漏的操作）前后拍摄的快照，查看哪些对象数量持续增加且未被释放。
  - **Allocation instrumentation on timeline (分配时间线):** 记录对象创建和垃圾回收随着时间的推移情况，帮助识别内存持续增长的模式。

排查步骤示例 (使用 Chrome DevTools Heap Snapshot):

1. 打开开发者工具，选择 Memory 面板。
2. 执行一次可能导致泄漏的操作（例如，打开一个弹窗组件）。
3. 拍摄第一个堆快照 (Take heap snapshot)。
4. 执行一次与操作相反的操作（例如，关闭弹窗组件）。
5. 强制执行垃圾回收 (Collect garbage)。
6. 拍摄第二个堆快照。
7. 在第二个快照中，选择 Comparison 视图，与第一个快照进行对比。
8. 按 Delta 排序，查找数量或大小显著增加的对象，特别是那些“不可达”（Not reachable）的对象，这通常是泄漏的迹象。重点关注自定义对象、事件监听器、DOM 节点等。
9. 点击可疑对象，查看其保留树 (Retainers)，分析是哪个地方持有对该对象的引用，从而找到泄漏源头。

4. 关键注意事项 (Key Considerations)

- **理解垃圾回收机制:** 虽然是自动的，但了解其基本原理（如可达性概念）有助于分析为什么某些对象没有被回收。
- **测试场景复现:** 内存泄漏往往在特定操作序列后出现。排查时需要能够稳定复现泄漏场景。
- **分步分析:** 在对比堆快照时，逐步缩小范围，专注于在疑似泄漏操作前后差异最大的对象类型。
- **警惕第三方库:** 有时泄漏可能发生在使用的第三方库中，需要检查库的使用方式或查找已知问题。

5. 参考资料 (References)

- **MDN Web Docs - Memory Management:** https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_management (概念介绍)
- **MDN Web Docs - Debugging Memory Leaks:** https://developer.mozilla.org/en-US/docs/Web/Performance/MemLeaks (排查方法，推荐工具部分)
- **Chrome Developers - Memory Terminology:** https://developer.chrome.com/docs/devtools/memory-problems/memory-terminology/ (DevTools 内存面板使用指南)
- **V8 Blog - Memory (V8 引擎内存管理，深入理解 GC):** 搜索 V8 官方博客关于内存管理的文章。例如 "Trash talk: the Orinoco garbage collector" 系列文章。
- **Google Developers - Discover memory leaks with Chrome DevTools:** https://developer.chrome.com/docs/devtools/memory-problems/discover-leaks/ (使用 DevTools 排查内存泄漏的详细步骤)