参考文章：

Chrome 官方:https://developer.chrome.com/docs/devtools/memory-problems/memory-101/
Tapd:https://www.tapd.cn/65362886/markdown_wikis/show/#1165362886001002684
阮一峰:https://www.ruanyifeng.com/blog/2017/04/memory-leak.html
MDN:https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Memory_Management
阿里巴巴 js 内存分析器:https://github.com/alibaba/JS-Memory-Analysor
Fuite 作者:https://nolanlawson.com/2020/02/19/fixing-memory-leaks-in-web-applications/
Microsoft:https://learn.microsoft.com/en-us/microsoft-edge/devtools-guide-chromium/rendering-tools/js-runtime
https://rocka.me/article/debugging-memory-leak-in-vue2
feedback_cell:https://rohitwhocodes.wordpress.com/2020/08/20/feedback-vectors-in-heap-snapshots/

研发六部——张婵娟分享

应用内存泄露问题查找说明文档.doc

什么是内存泄漏

由于疏忽或错误写法造成程序未能释放已经不再使用的内存

了解内存泄漏之前，建议先了解一下内存相关术语：
https://developer.chrome.com/docs/devtools/memory-problems/memory-101/

javascript 是有自动垃圾回收机制(GC: Garbage Collecation)的,GC 通常情况下主要有两种实现方式:

1. 标记清除法(chrome 是用这个方法实现 GC) (造成泄漏是因为不需要的引用没有被清除)
2. 引用计数法

1、标记清除——JS最常用的垃圾回收机制

当变量进入执行环境时，就标记这个变量为“进入环境”。进入环境的变量所占用的内存就不能释放，当变量离开环境时，则将其标记为“离开环境”。

垃圾回收程序运行的时候，会标记内存中存储的所有变量。然后会将所有在上下文的变量，以及被在上下文中变量引用的变量的标记去掉。

在此之后再被加上标记的变量就是待删除的了，原因时任何在上下文中变量都访问不到它们了；

随后垃圾回收程序做一次内存清理，销毁带标记的所有值并收回他们的内存。

```
var m = 0,n = 19 // 把 m,n,add() 标记为进入环境。
add(m, n) // 把 a, b, c标记为进入环境。
console.log(n) // a,b,c标记为离开环境，等待垃圾回收。
function add(a, b) {a++var c = a + breturn c
}
```

2、引用计数

语言引擎有一张“引用表”，保存了内存里面所有的资源（通常是各种值）的引用次数。如果一个值的引用次数是0，就表示这个值不再用到了，因此可以将这块内存释放。

如果一个值不再需要了，引用数却不为0，垃圾回收机制无法释放这块内存，从而导致内存泄漏。

```
const arr = [1, 2, 3, 4]; //数组是一个值，会占用内存，变量arr是仅有的对这个值的引用，因此引用次数为1。
console.log('hello world'); //这里并没有用到arr，但还是会持续占用内存
```

如果需要这块代码被垃圾回收机制释放，那么只需要设置arr=null，数组的引用次数就为0了，就被垃圾回收了。

怎么知道发生了内存泄漏

从用户感知的角都来说：

1. 随着时间的推移，页面的性能会逐渐变差。
2. 页面的性能一直很差
3. 页面性能延迟或者出现频繁暂停

使用 chrome devtools 来查找

1. 可以看浏览器的任务管理器
2. 可以查看性能监视器的 js 堆内存使用情况(Performance monitor)
3. 可以在 performance 主面板录制一个性能报告图进行分析

如何修复内存泄漏

主要是通过 devtools 的 Memory 面板抓取堆快照和录制时间线结合分析，然后找到泄漏点。一些简单的泄漏或者人为故意写的泄漏的例子在上面的参考文章里面有，它们都比较简单清晰的可以识别，而市场的工程里面往往都不是很清晰，并且找出来的数量可能都是百万级的，通过简单的例子可以让你在市场工程里面遇到一些关键的字眼容易定位到具体的代码里。

通过 Memory 面板抓快照查看

以解决方案 TEClient 工程 master 分支上的最新代码 900007 交易为例：
跑起来工程后选择客户服务——>开始服务——>身份信息录入——>抓一下初始的堆快照
每次抓之前要记得清一下控制台并且点一下垃圾桶执行一次垃圾回收

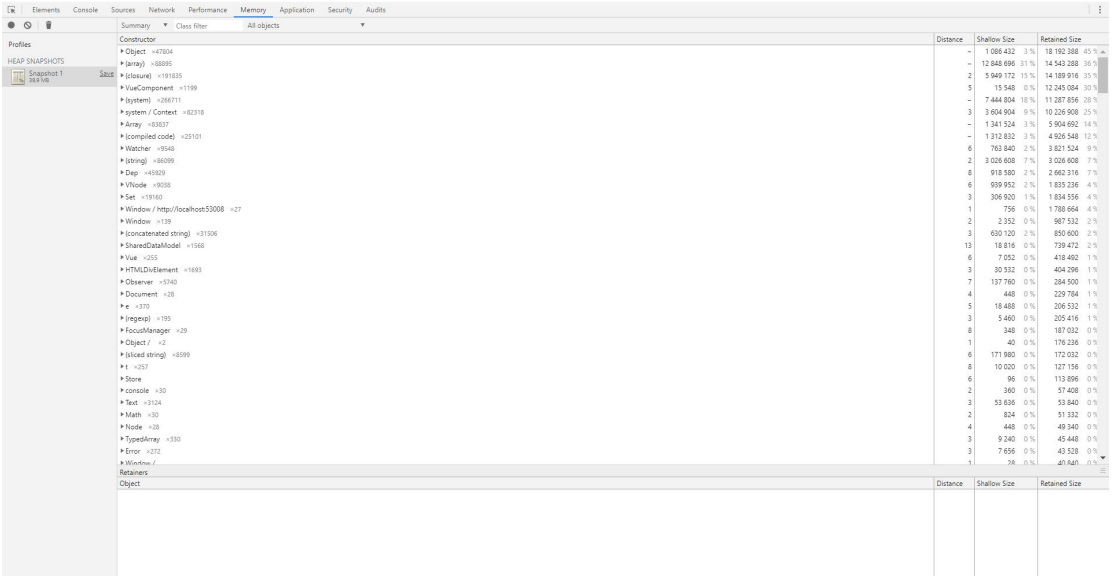

图 1

图 1 就是初始还没开过交易时候第一次抓的堆内存快照

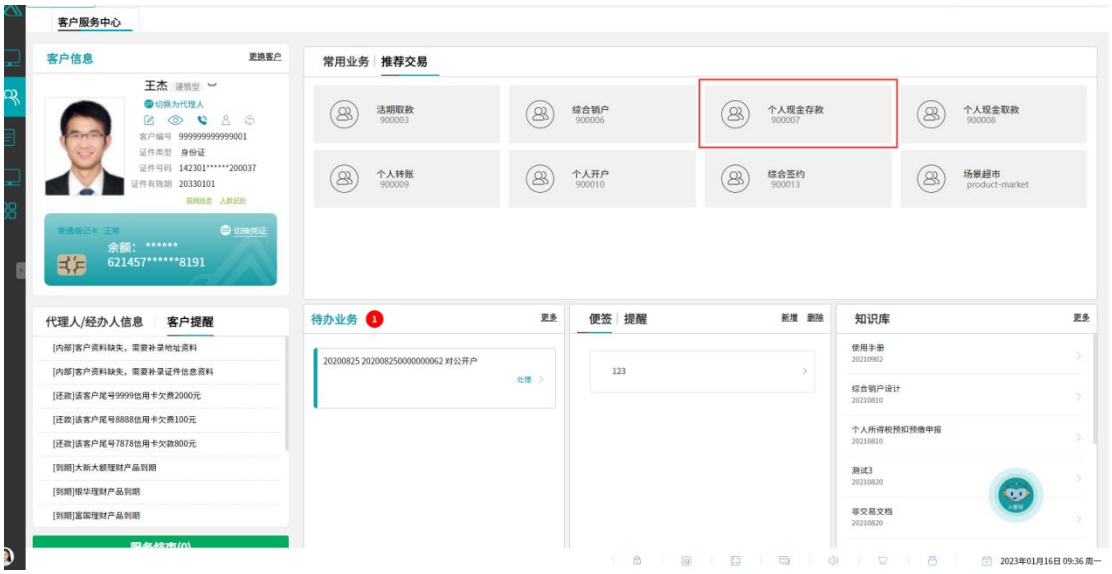

图 2

之后的操作就是点开 900007 交易，然后直接点签页上的 x 关闭交易,然后清理控制台，手动 GC,然后抓快照，重复该操作至少两次

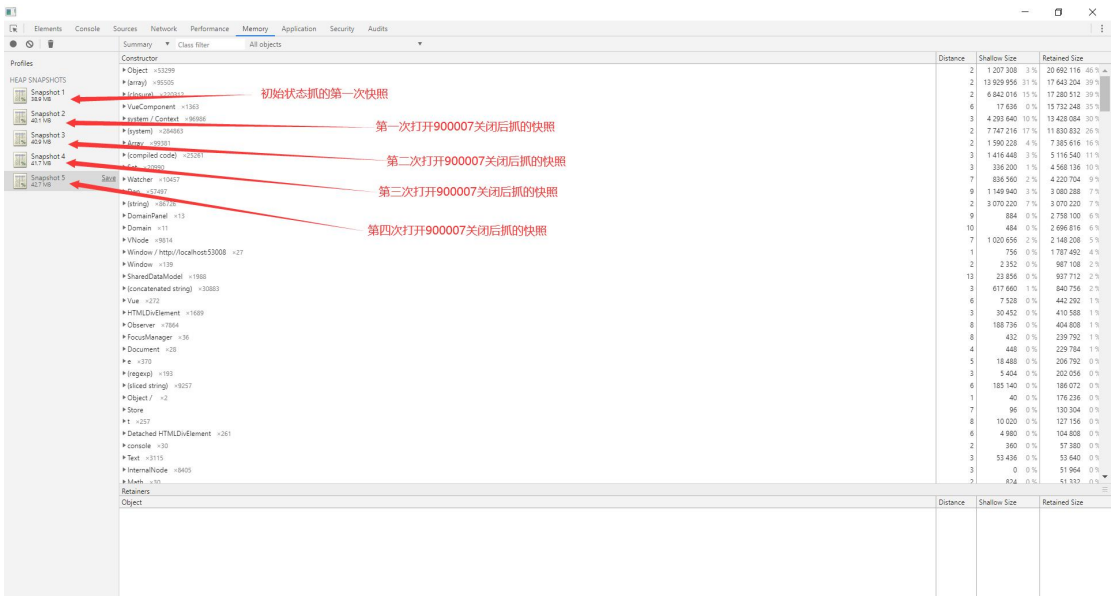

图 3

我是重复了四次操作，算上最开始抓的一共五个快照

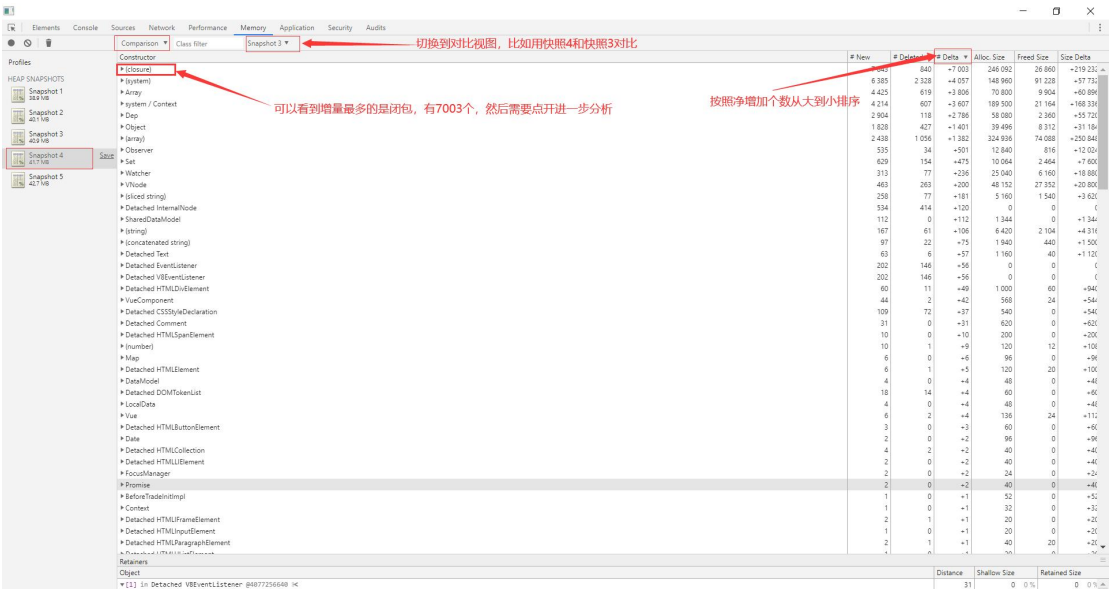

图 4

然后使用对比视图去找两个快照间的差异，这里以快照 4 和快照 3 对比为例，然后再以增量个数从大到小排序，这里每一列的含义分别是：

#New 新增的个数　　#Deleted 移除的个数　　#Delta 增量

Alloc.Size 新分配的内存的大小 Freed Size 释放的内存的大小 Size Delta 增量所占内存大小

我们按增量排序后发现(closure)闭包增加最多，点开后进行进一步分析

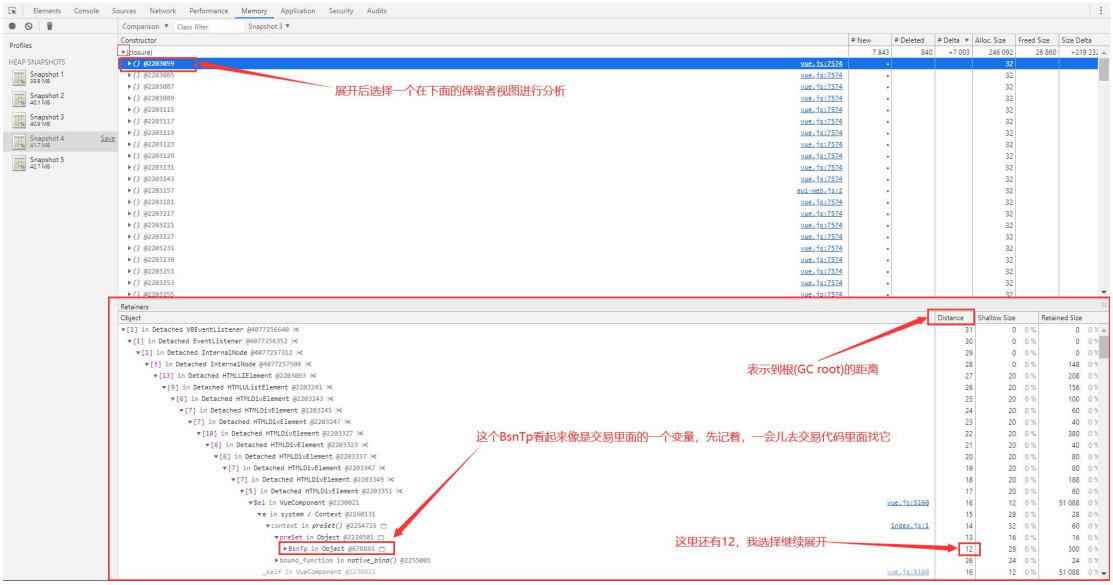

图 5

然后展开(closure)后选择一下进行具体的分析，下面的视图是保留者视图，从上往下是一层一层的引用关系，但这个关系相当于是反过来的，下一行的是上一行的父级，最下面的一般就是根，保留着视图里面后面三列分别是当前对象到根的最短引用距离、浅尺寸、保留尺寸，这几个概念在前面的内存术语会有介绍

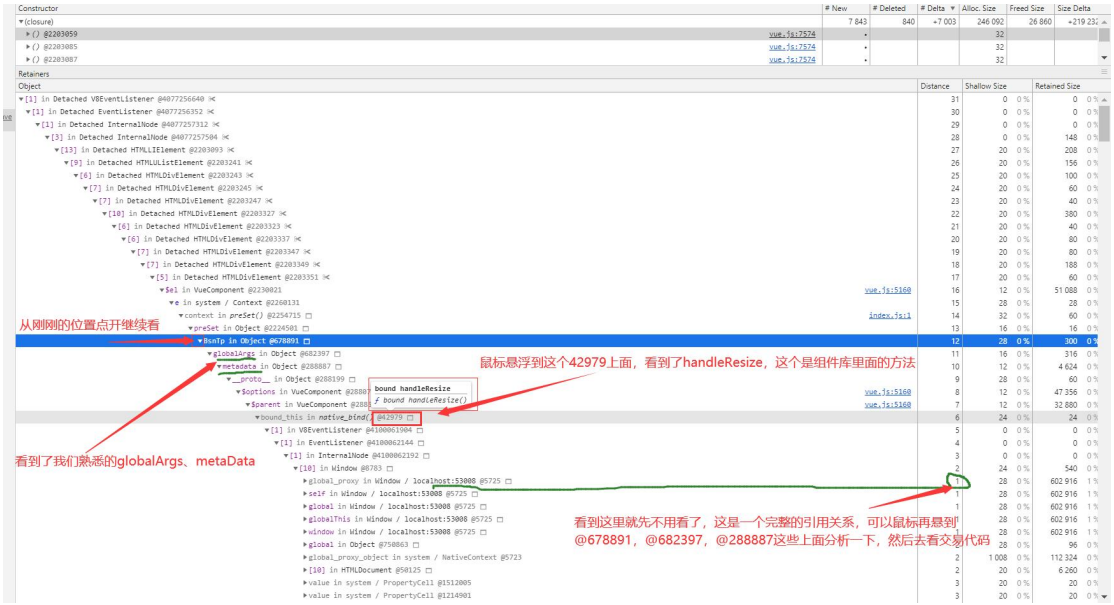

图 6

接着图 5 继续展开，可以看到了关键的 aui-select 里面的 handleResize

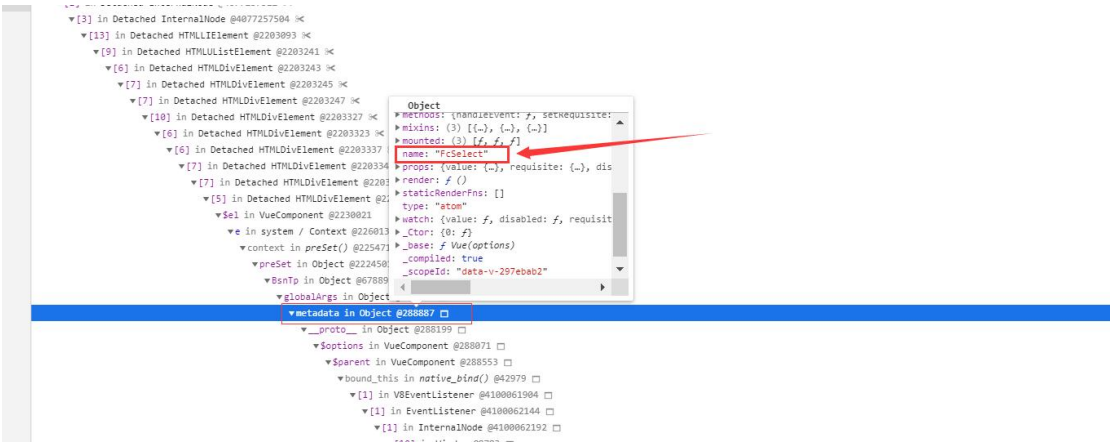

图 7

然后鼠标再悬到其他对象 id 上面找到了关键的 fc-select

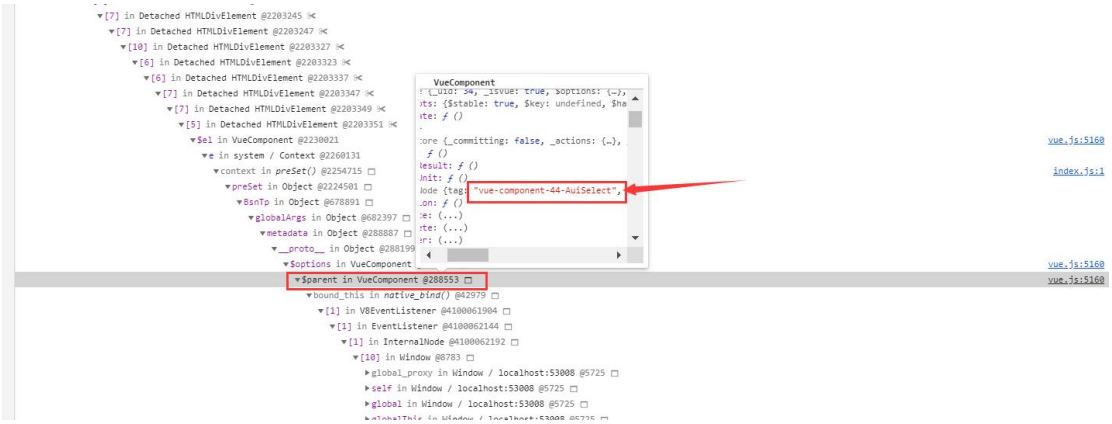

图 8

鼠标再悬到这个上面找到 aui-select

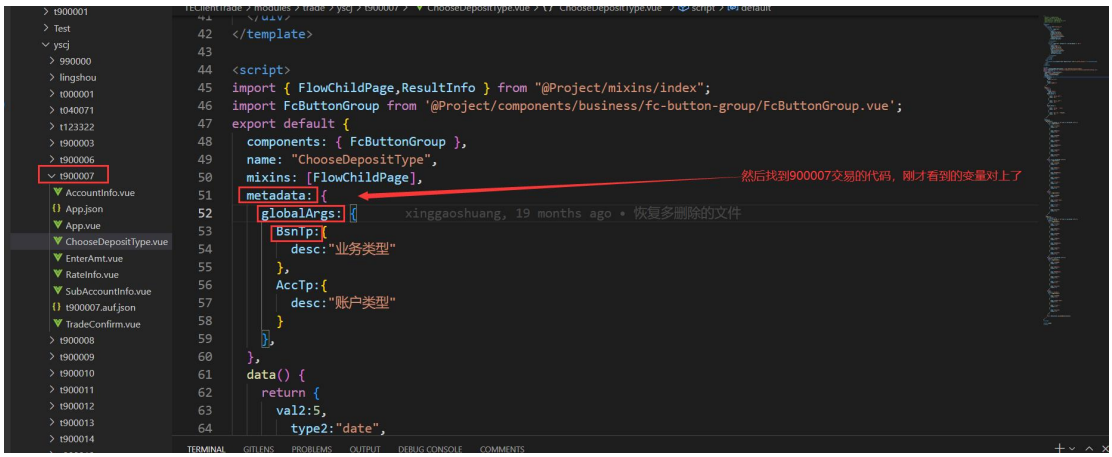

图 9

然后找到 900007 的交易代码，找到了 dm 上的 BsnTp

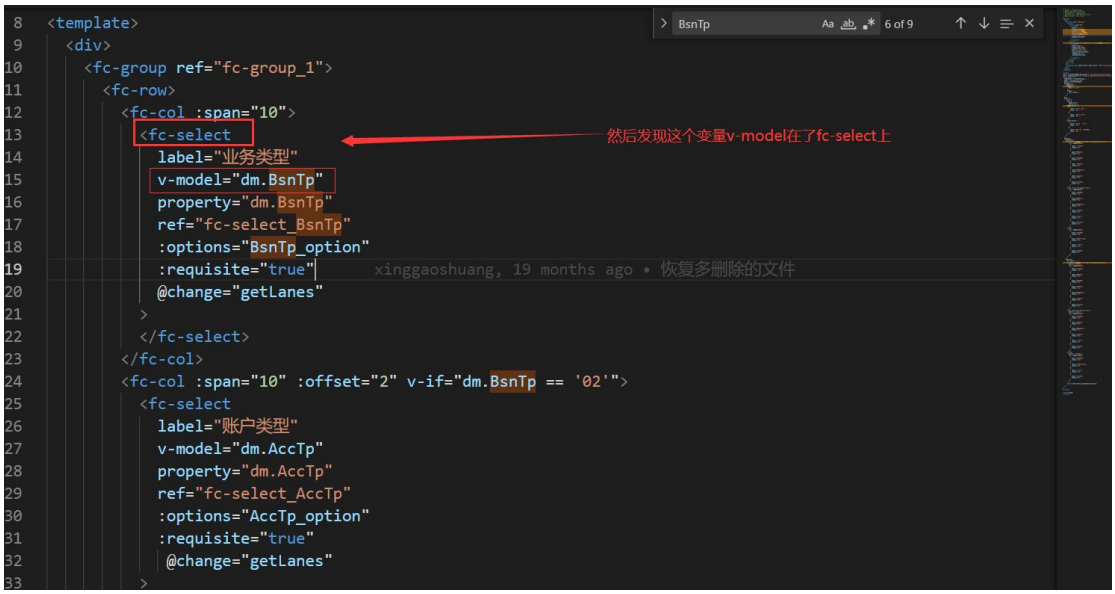

图 10

BsnTp 是 v-model 在 fc-select 上的

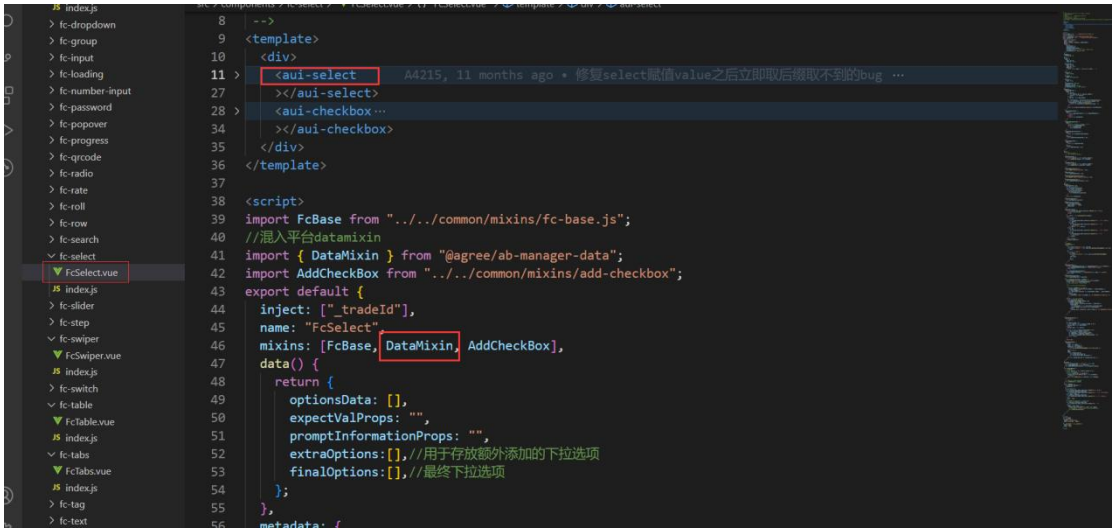

图 11

然后去看 fc-select 的实现，里面是包了 aui-select，并且混入了 DataMixin

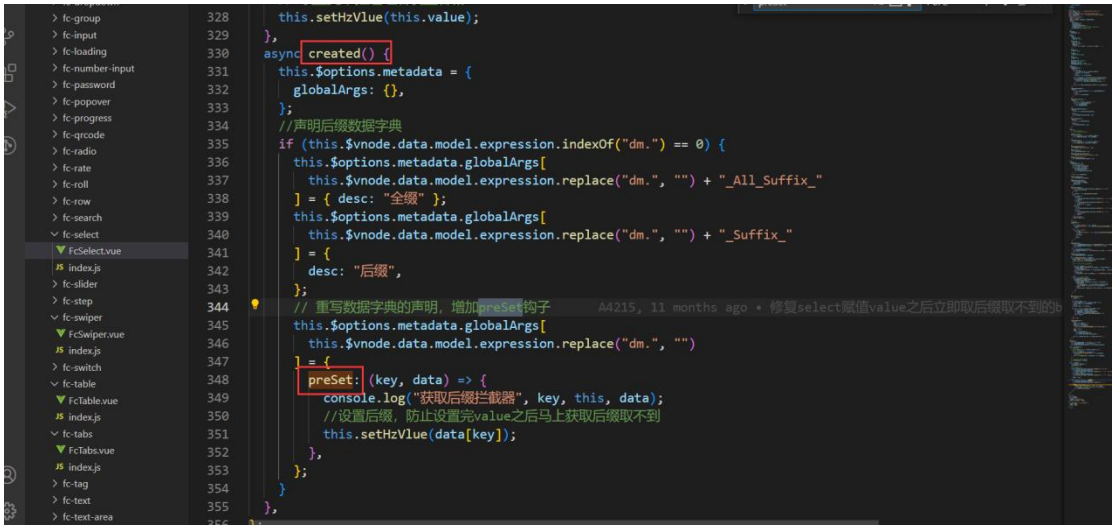

图 12

在 fc-select 的 created 生命周期里面会帮所有 fc-select 生成 preset，preset 会被 @agree/ab-manager-data 里面调用

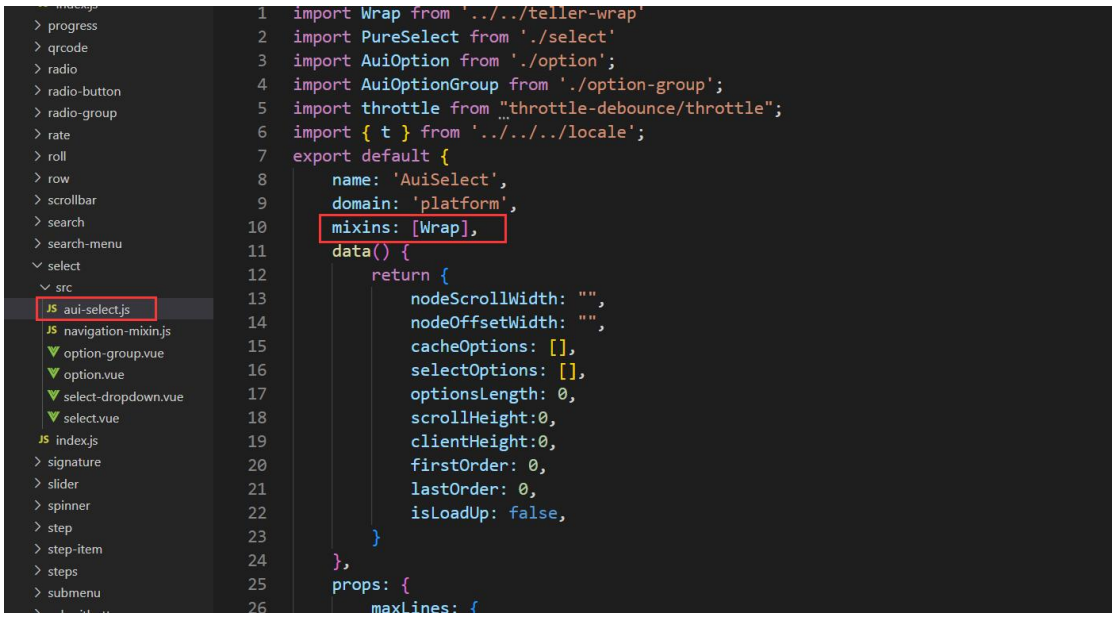

图 13

再来看 aui-select 的源码，根据刚刚鼠标悬上@288553 看到的应该是 select 混入的 wrap 里面的 handleResize 方法，所以找到 wrap 里面的此方法

图 14

图 15

图 16

看着这个 setPopoverWidth 方法除了定义了一个 el 赋值了一个 dom 节点使用完没有解除引用，其他的没看出通过 handleResize 是怎么泄漏的，所以我先把 el 使用完的引用断开，然后再看看效果

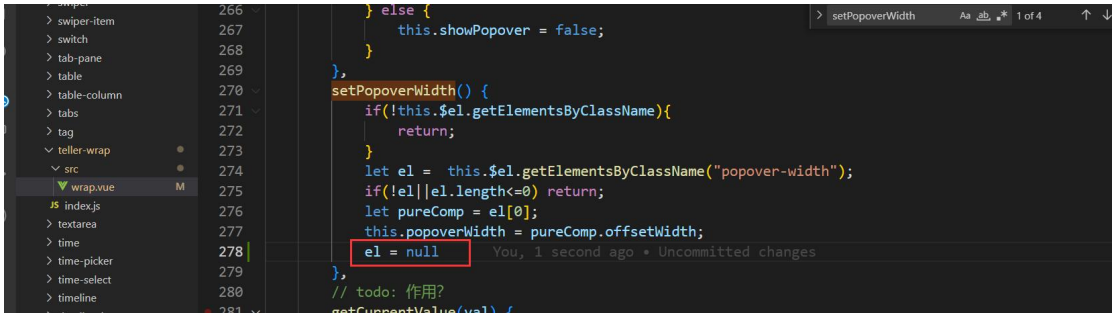

图 17

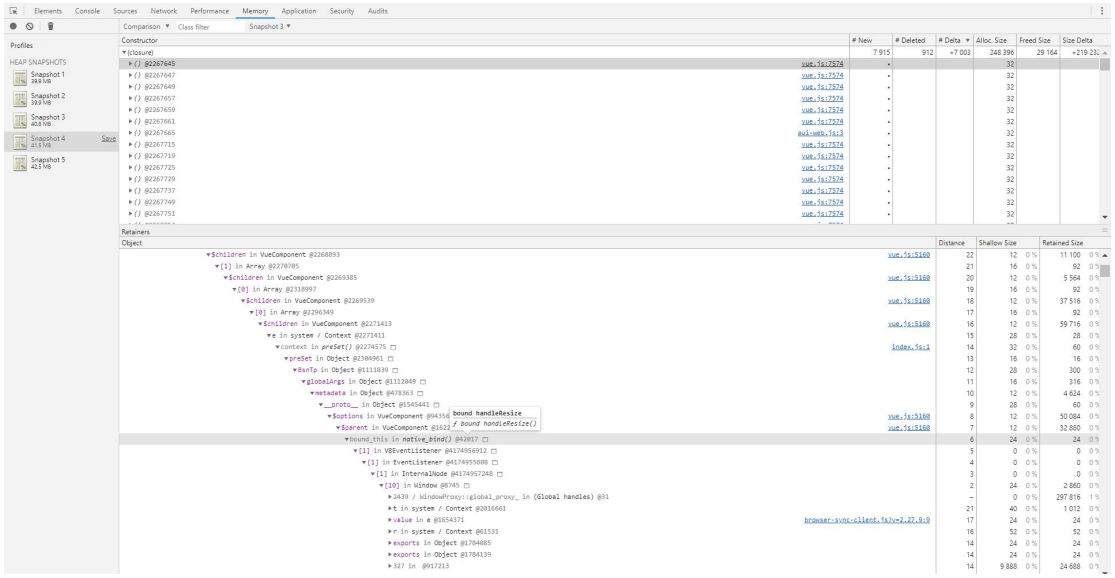

图 18

我又重复上面的操作抓了五次对比，然后发现好像没什么变化，还是增 7000 多个，还是 handleResize，找不到 handleResize 的泄漏原因，于是我把相关代码先注释了

图 19

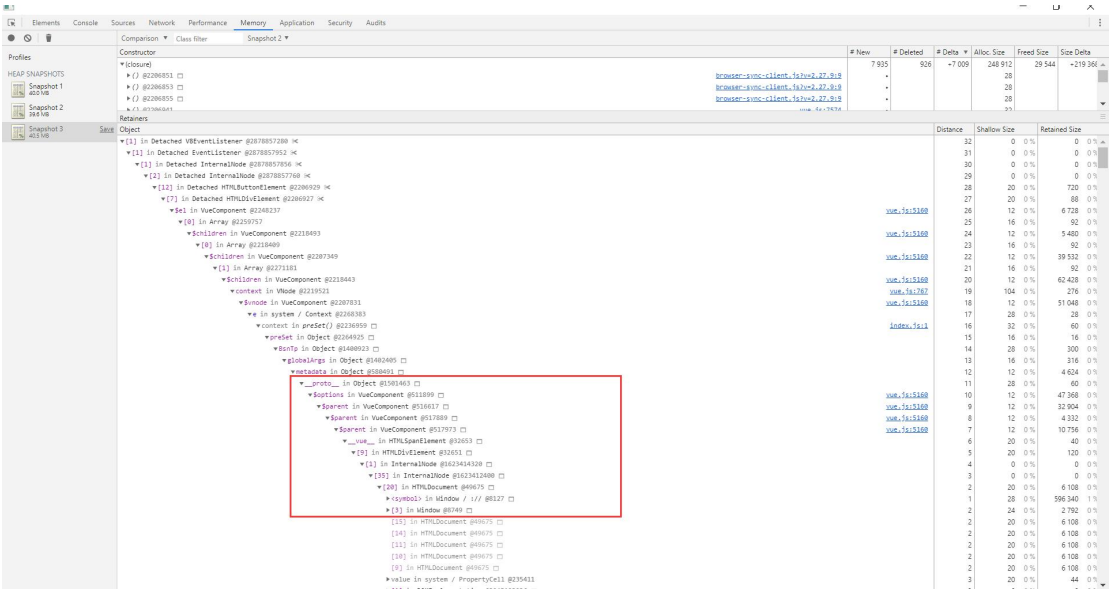

图 20

这次注释后我又重复之前的操作抓了三次，对比快照 3 和快照 2 发现已经不显示 bound_this in Native_bind()了，但是又陷入了新的难题

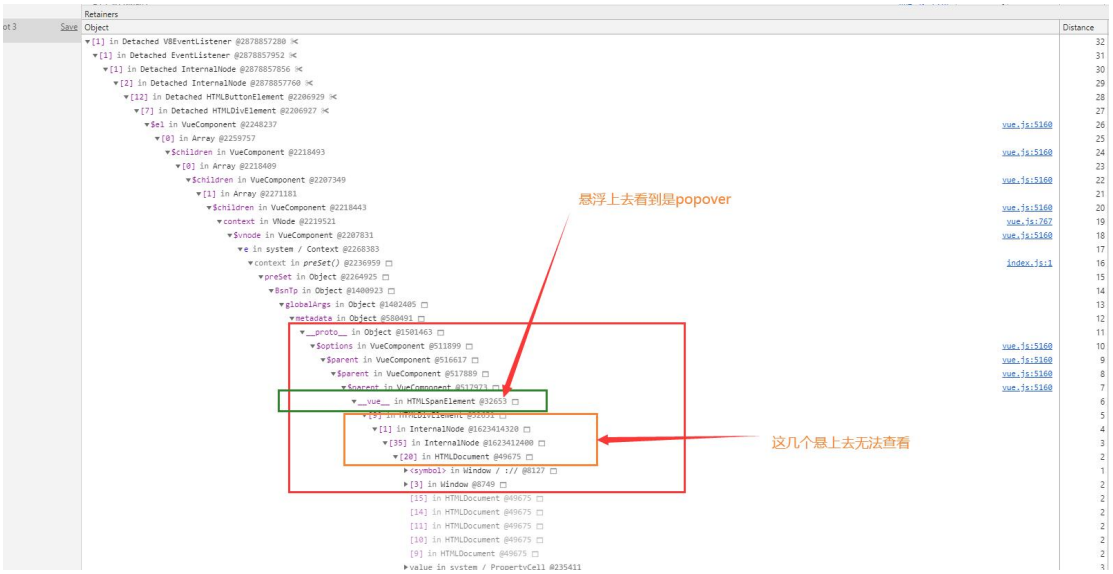

图 21

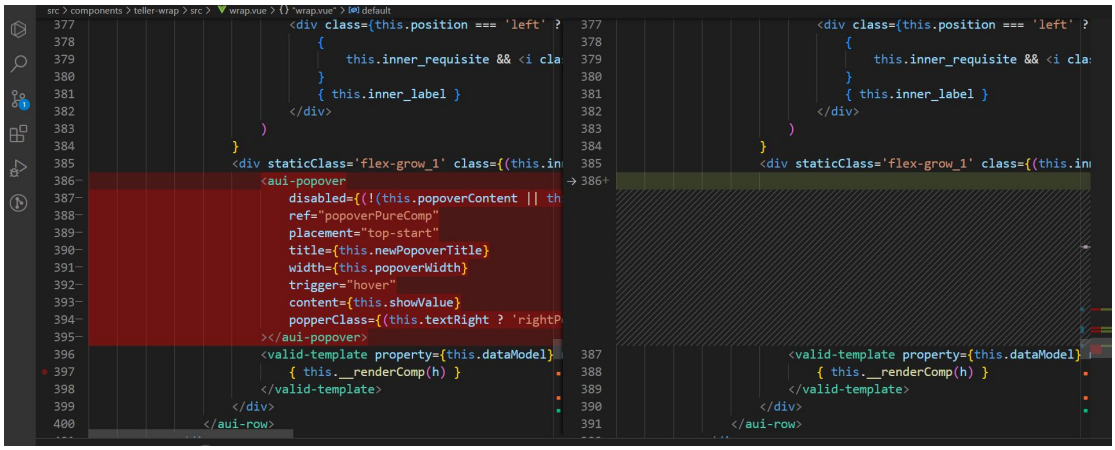

图 22

然后我把 select 里面的 popover 去掉重新重复上面的操作

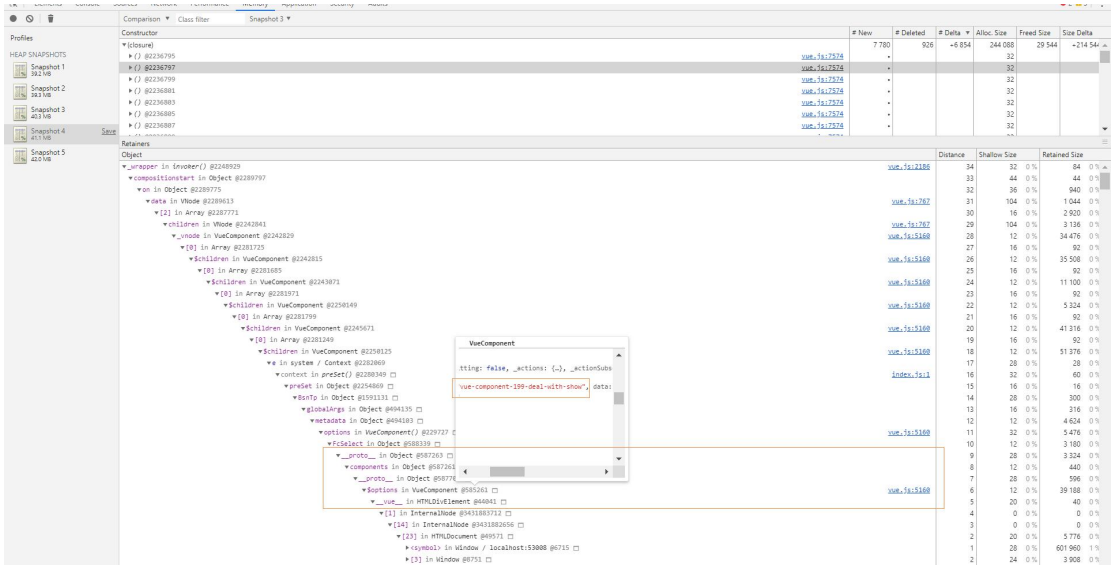

图 23

这次新增的(closure)只有 6854 个了，但是问题还是没有解决，不知道引用关系怎么又去到了侧边栏里面的 DealWithShow 组件里面。

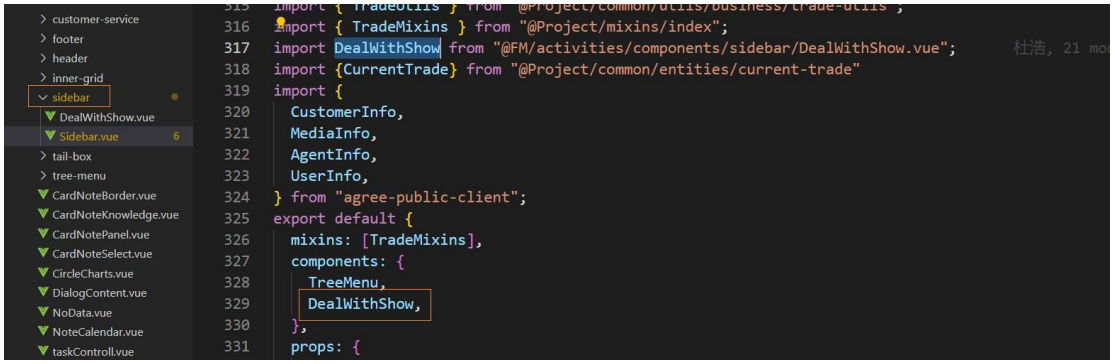

图 24

通过 Memory 面板录制时间线查看

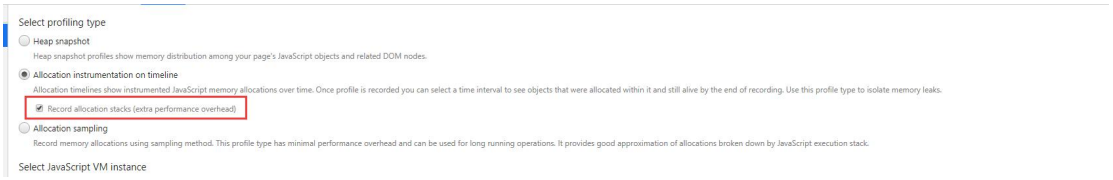

图 25

勾选这个是为了可以看到分配的栈，但是会有额外的性能消耗，交易比较复杂的话勾选这个录制过程中一般就白屏了

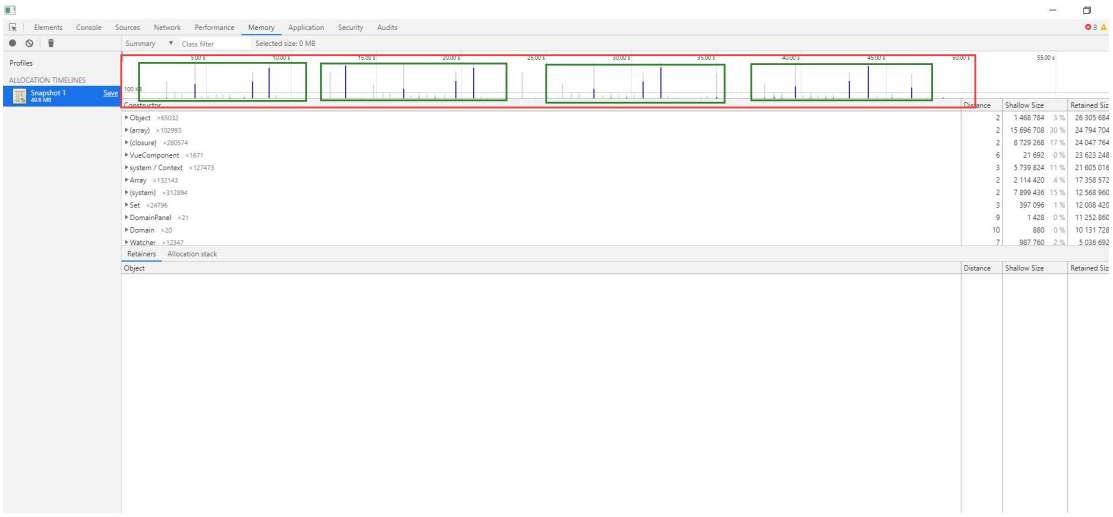

图 26

录制之前还是手动 GC 一次，然后点击录制按钮后，还是和刚刚一样打开 900007 交易，然后直接关闭，我是重复了 4 次，然后结束录制，这 4 次对应各自的时间线范围我用绿框大概框了起来，时间线里面灰色的线条代表分配后回收了的，蓝色的线条代表分配后一直没有销毁，最终还在内存中存在，不是说看到蓝色的线一定就是内存泄漏，但是重复多次操作后蓝色的线有规律的出现一般就是发生了内存泄漏

图 27

可以拖动两根滑块在查看某一时间段内分配的对象，我是放在了最后一次打开关闭交易的位置，然后点开上面的构造器选择一个对象进行查看

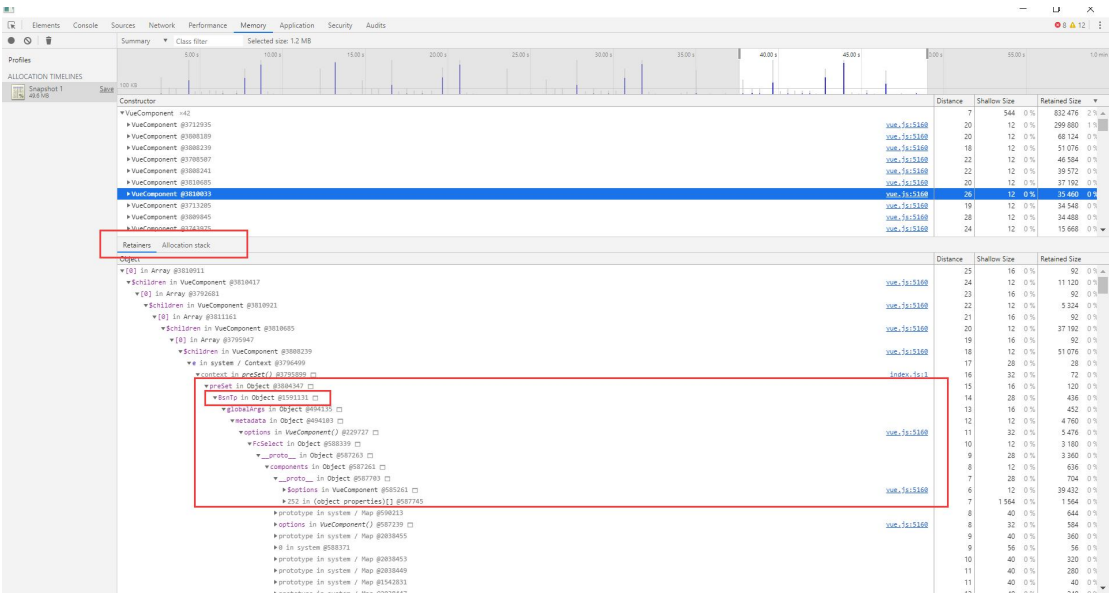

图 28

点开这 42 个 VueComponent 进行查看，发现大部分还是刚刚在快照里面的看到的那个

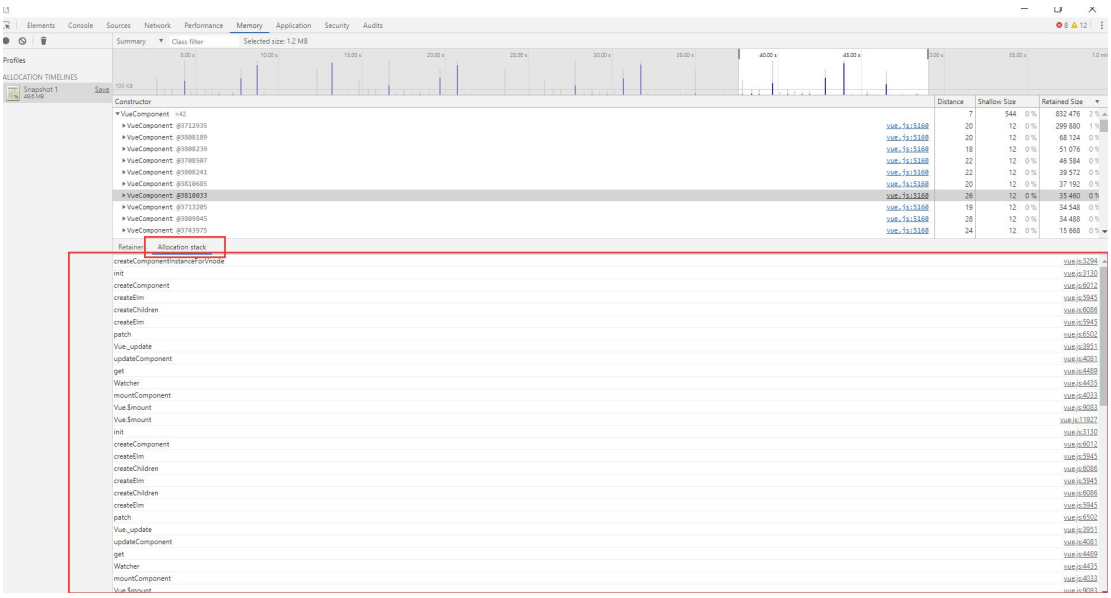

图 29

由于我们刚刚在录制时间线的时候勾选了分配栈，所以可以从保留者视图切换到分配栈视图，里面有完整的过程，有时候可能会看到一些你熟悉的函数来帮忙查找问题，但即使你录制的时候勾选了分配栈也不是每个构造器下的每一个对象都能看到它的分配栈

图 30

有时会是这样的

堆快照中 constructor 列常见项含义

表 1 堆快照中 constructor 列常见项含义

Constructor	Description
(global property)	全局对象和它引用的对象之间的中间对象(如 window)，如果一个对象是使用构造函数 Person 创建的，并且由一个全局对象持有，则保留路径表示为[global] > (global property) > Person，这与对象直接相互引用的规范形成对比。存在中间对象是为了性能。全局变量会定期修改，属性访问优化对非全局变量对象做得很好，但不适用于全局变量
(roots)	保留视图中的根是具有对所选对象的引用的实体，这些也可以是引擎为自己创建的引用。引擎有引用对象的缓存，但所有此类引用都是弱引用，并且不会阻止对象被回收，因为没有真正的强引用
(closure)	通过函数闭包对一组对象的引用的计数
(array),(string),(number) (regexp)	具有引用数组、字符串、数字、正则表达式的属性的对象列表
(complied code)	与编译代码相关的一切，脚本类似于一个函数，但对应一个<script>主体。SharedFunctions(SFI)是介于函数和编译代码之间的对象。函数通常有上下文，而 SFI 则没有
HTMLDIvElement HTMLAnchorElement DocumentFragment And so on	对代码引用的特定类型的元素或文档对象的引用
(object shape)	对 V8(Javascript 引擎)用来理解和索引类对象中的属性的隐藏类和描述符数组的引用
(Bigint)	对 Bigint 对象的引用，该对象用于表示和操作太大而无法由 Number 对象表示的值
Detched Elements	一个 DOM 节点只有在没有被页面的 DOM 树或者 Javascript 引用时，才会被垃圾回收。当一个节点处于”detched”状态，表示它已经不在 DOM 树上了，但 javascript 仍旧对它有引用，所以暂时没有被回收。
(System), System/Context	表示引擎自己创建的以及上下文创建的一些引用，这些通常不需要太关注
Dep、Observer、VNode、Watcher、VueComponent	这些都是 vue 特有的 constructor

其他没有列出来的构造函数就都按字面意思去翻译。constructor 下面那么多个，难道要每个都点开去看吗？一般是两个快照比较，看比较视图，增量不是 0 的都需要看，因为不能一下清晰具体的定位到问题，就需要每个都点开看看，可以先找到熟悉的看，比如 DetchedHTMDIVElement、VueComponent。

(array)和 Array 的区别是什么：(array)是任何类型对象的集合，这些对象具有引用实际数组的属性，而 Array 是实际 Array 对象的列表

常见的几种内存泄漏

1. 定义的全局变量可能会造成泄漏(比如定义了没有使用，或者隐式的定义了一个)
2. addEventListener 一定要记得移除，有时候可能写了 removeEventListener 还是会造成泄漏，可能是写法有问题，可能是代码里某个操作直接 removeChild 了你监听的那个 dom 而没有移除 listener
3. 定时器:setTimeout/setInterval 用完没有清除也会导致泄漏
4. 保存的 dom 节点，使用完记得断开引用
5. IntersectionObserver, ResizeObserver,MutationObserver 等等，这些新的 API 非常方便，如果你在组件内部创建一个，并且它附加到一个全局可用的元素，没有调用 disconnect()来清理它们，也会造成泄漏
6. 如果 Promise 从未被 resolve 或者 reject，它可能会泄漏，在这种情况下，任何.then()附加到它的回调都会泄漏
7. 不恰当的使用闭包(闭包函数内的基本数据类型的变量或者引用数据类型的数据被保存到外面一直没有释放)

目前遇到的几个难题

在甘肃农信工程里面打开关闭 030201 交易每次会泄漏 17MB 左右的内存，这些泄漏点里面大都会指向两个地方(1,2)：

1. 指向 window 上面的 requireJS
2. 指向 home 页自定义的热键 ctrl + s
3. 有很多被 feedback_cell 引用的到底需不需要关心
4. 堆快照的 summary 视图的 constructor 里有许多 window,这些 window 的来源和区别

其他查找内存泄漏的工具

1. alibaba/JS-Memory-Analysor

   已有 5 年没有维护，需要把抓好的快照放进去会生成一个分析的统计图，还会把一些重点大尺寸对象红色标出，提示几个可疑对象的 id，和我们自己看堆内存快照比较分析区别不大，没有太大的作用
2. Memlab
3. Fuite

   2 和 3 都是目前最新的查找工具，需要给一个可以在浏览器正常打开的 url 地址，本地跑起来的 localhost:53008/index.html 也可以,支持自定义写类似于测试案例的自定义操作过程，然后会给出一个分析报告，不是很适合市场的交易工程