Devtools 入门

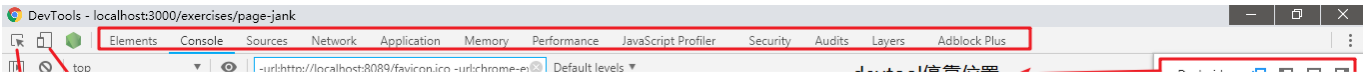
相关

- 本文作者：ScarSu
- 进阶系列文链接
- 本文基于 chrome 浏览器版本 73.0.3683.103（正式版本）总结
- 本文目的：关于【devtools 能做什么】建立完善的知识结构，至于怎么做，请查阅官方文档；另工具类知识需要实践，建议阅读本文时打开 sample用 devtools 操作一遍
- 参考 1：google developers 官方文档
- 参考 2：来自作者 Jon Kuoerman 在 FrontEndMaster 的 Mastering Chrome Developer Tools v2 课程
- 参考 3：来自 作者 Tomek Sułkowski 在 medium 的系列文章
- 系列文脑图.xmind
- 脑图.png

web devtool 历史

- view-source + alert 调试法
- Live DOM Viewer
- Firebug

Chrome Devtools 界面概览



Tips and Tricks

- 快捷键：ctrl shift p：执行命令
- 快捷键：ctrl p：打开文件
- 快捷键：esc：显示/隐藏 drawer(第二行面板

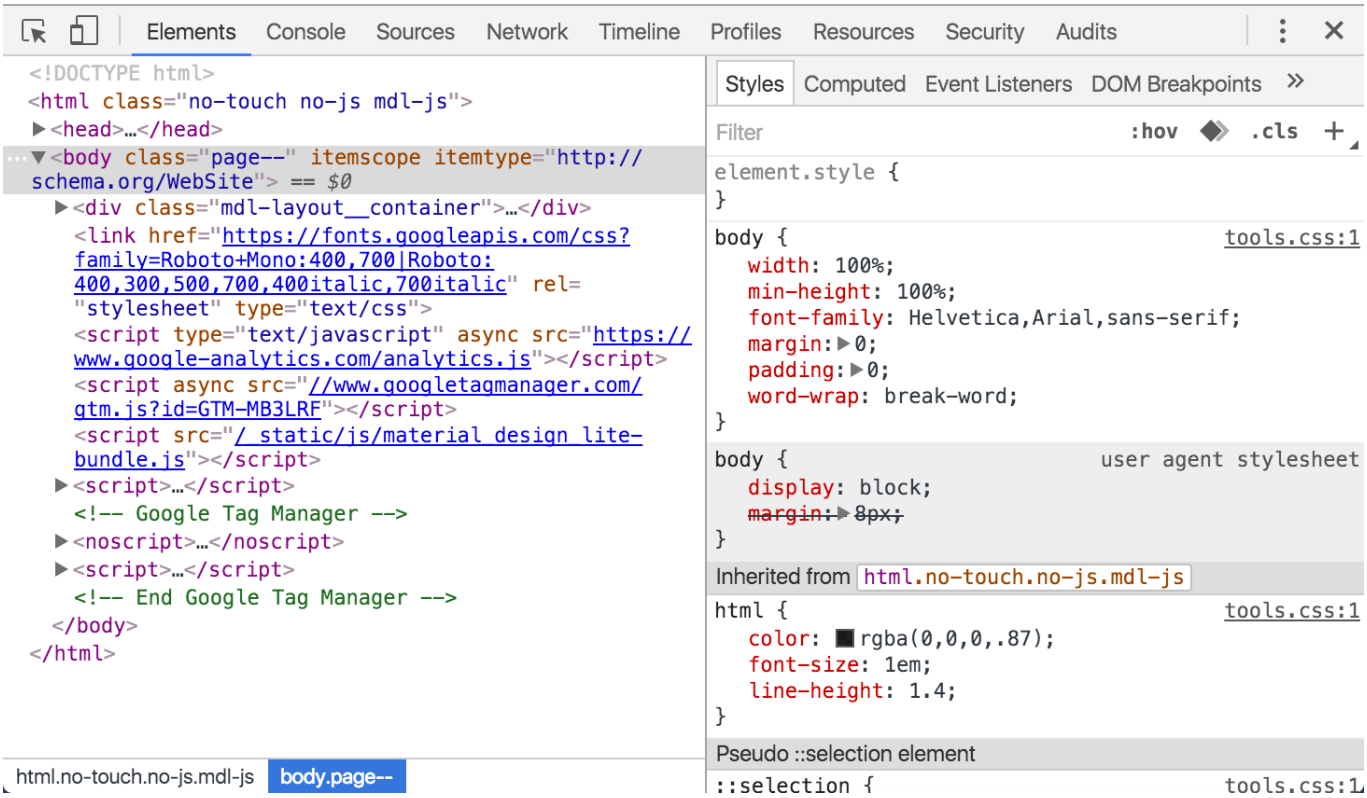
- 快捷键：ctrl shift c：选择元素
- more -> focus debugee：切换至正在被调试的页面
- more -> more tools：全部面板
- 无痕模式打开网页 —> 更纯净的调试环境，无扩展代码干扰
- 实验性功能：

```
打开url     chrome://flags/
搜索dev
打开Experimental Extension APIs开关
在settings中找到experiments可以找到相关实验性功能
shift按七次，显示隐藏的实验性功能（比如terminal
```

- 或者使用金丝雀版 chrome - Canary - 开发者专用的每日更新版

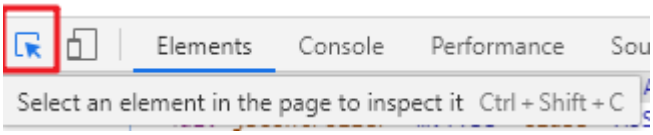
Elements 面板

使用 Chrome DevTools 的 Elements 面板检查和实时编辑页面的 HTML 与 CSS



Inspect Mode

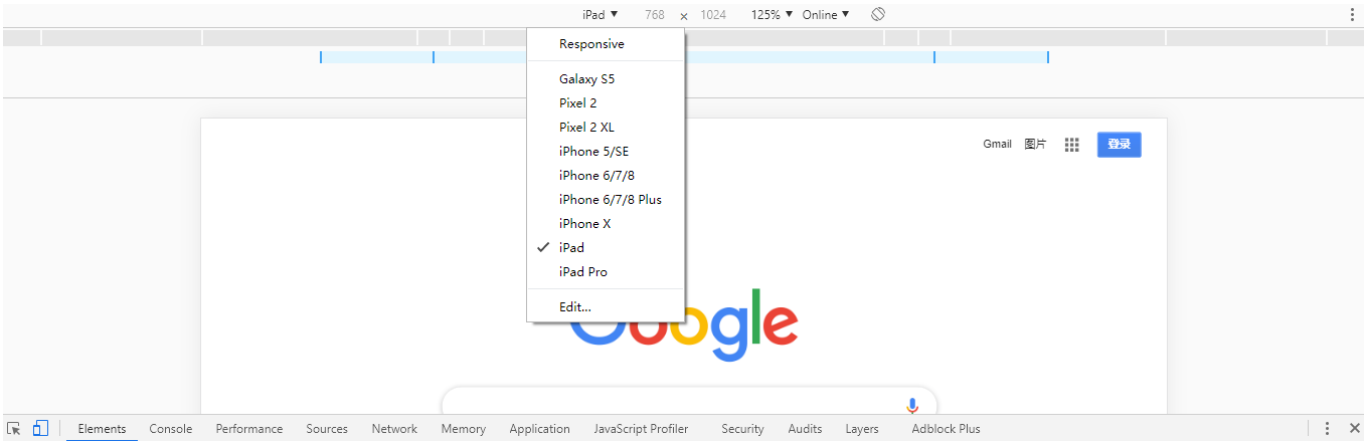
快捷键 ctrl shift c/点击面板左上角的按钮，进入元素选择模式



在新版本 chrome 中，选择元素时会显示更多元素信息

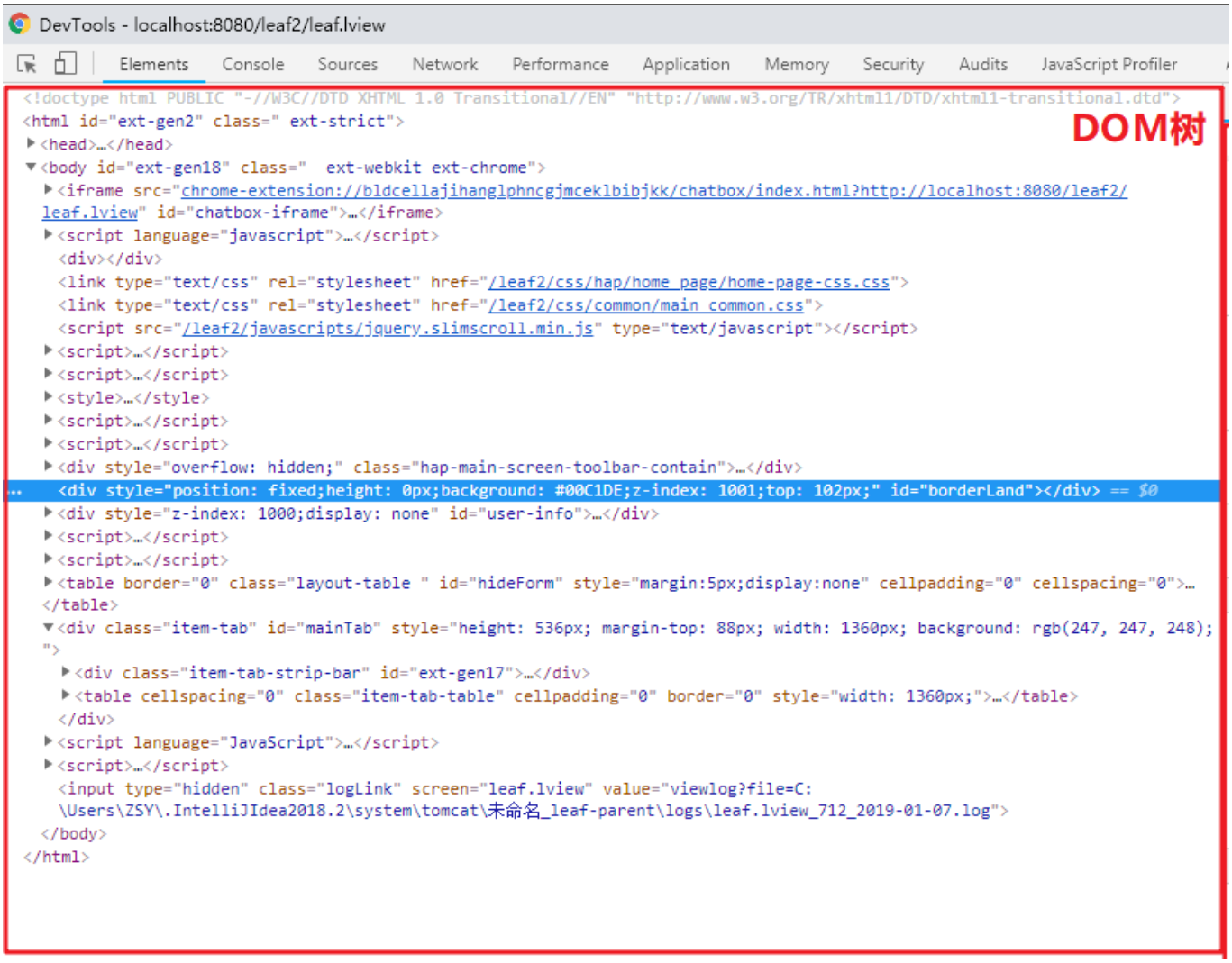
Device Mode 设备模式

- 模拟不同尺寸移动端设备下，网页的表现。
- 是自适应网页调试利器。
- 内置/可配置既有设备属性，例如 iphone/ipad
- 支持调试媒体查询 media-query。



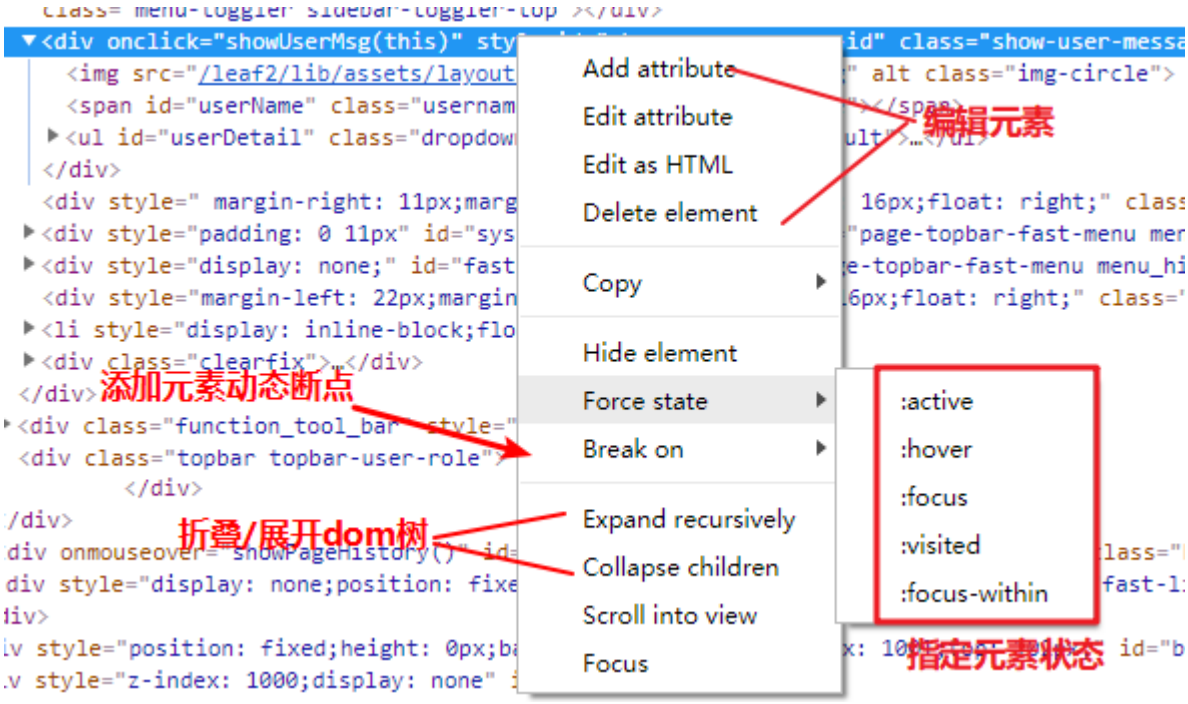
DOM 树

在元素面板左侧是当前页的 DOM 树



在 DOM 树中你可以：

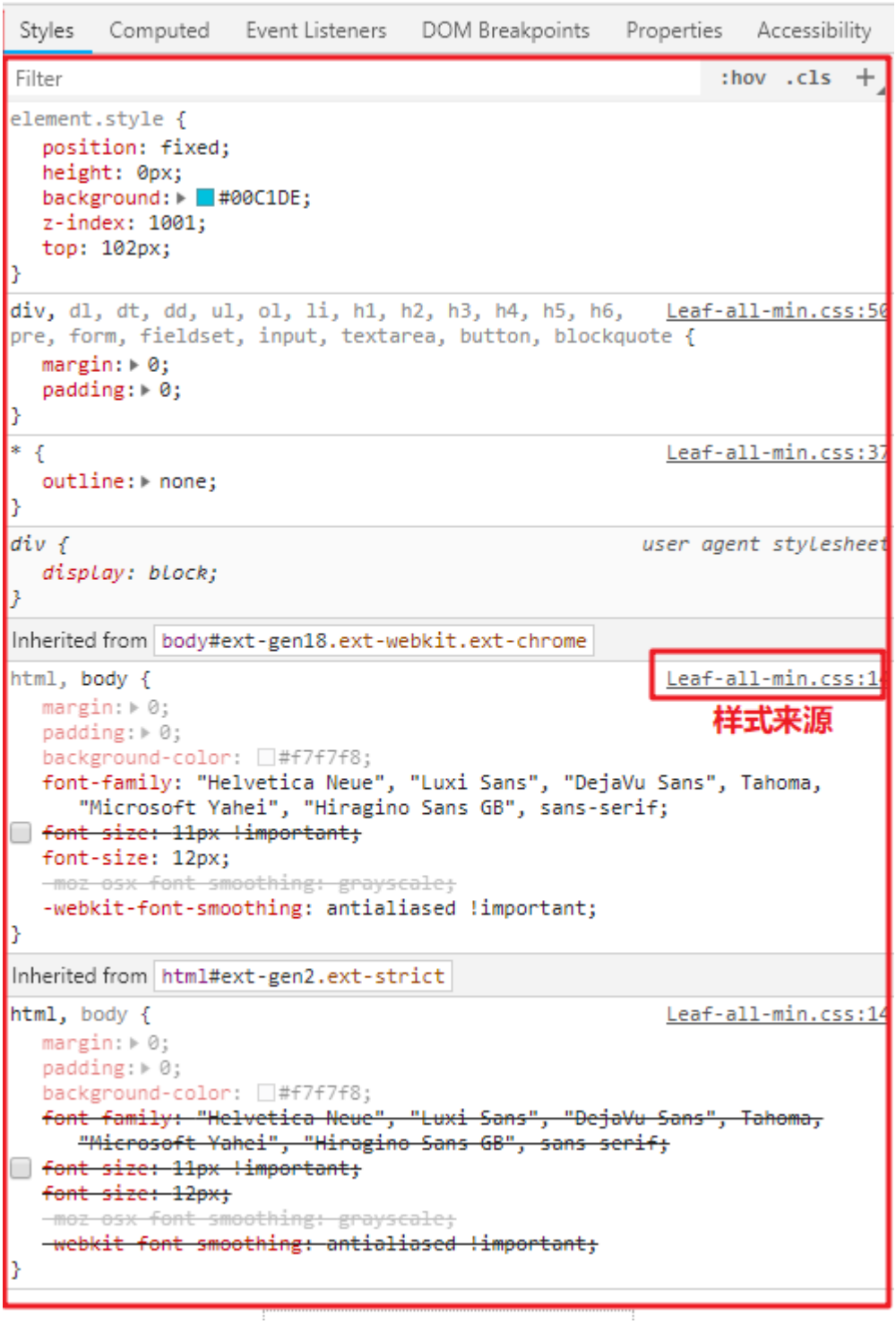
- 直接增/删/改/复制/拖放移动 DOM 元素，查看实时效果(非持久化)
- 添加元素断点(节点移除断点，属性变更断点，子树变更断点)
- 模拟元素 focus/hover/actice 等状态
- 选中元素后通过右键“Scroll into view”突出显示当前元素在页面的位置
- 按快捷键**h**来快速隐藏/显示元素当前元素及其后代元素(原理是 visibility 设为 hidden,不影响其他元素,不引起重绘)
- 按住 alt 键 点击 dom 元素前的箭头：全部折叠/展开当前元素及其后代元素



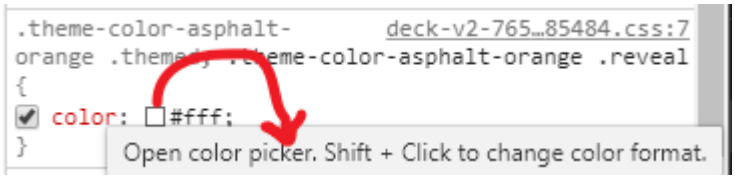
Styles

在面板右侧 Styles 窗格中：

- 会显示节点的各级样式
- 每级样式的来源
- 每条样式属性是否命中
- 可以直接增/删/改元素样式，查看实时效果(非持久化)



color picker



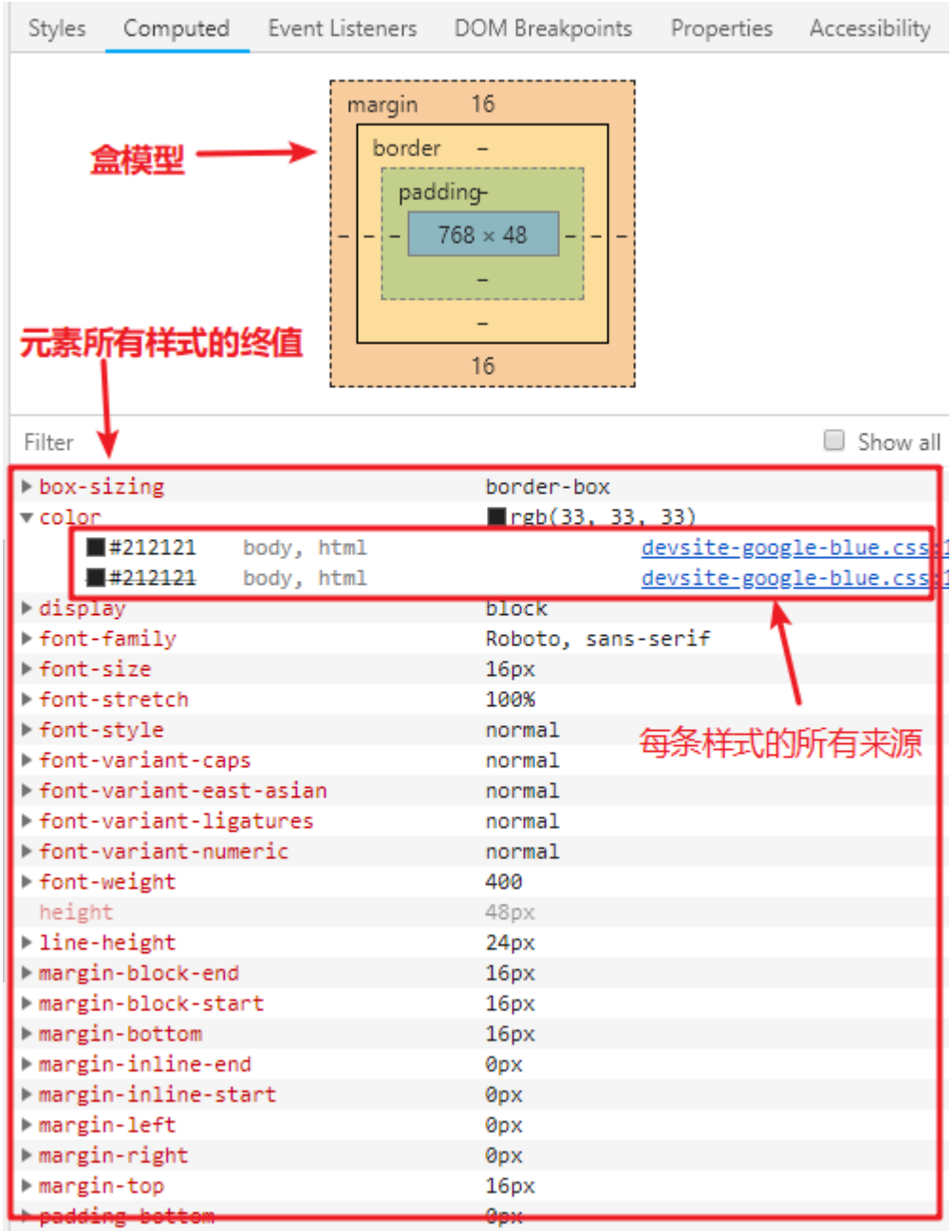
- 在样式窗格中，devtools 给所有颜色属性值前添加了 color picker 工具
- 按住 shift 点击色块，快速切换颜色格式 rgb/hsl/hex

- page colors：color picker 中会列出页面所有的颜色
- material colors：color picker 中会列出 google 设计推荐色系

Computed

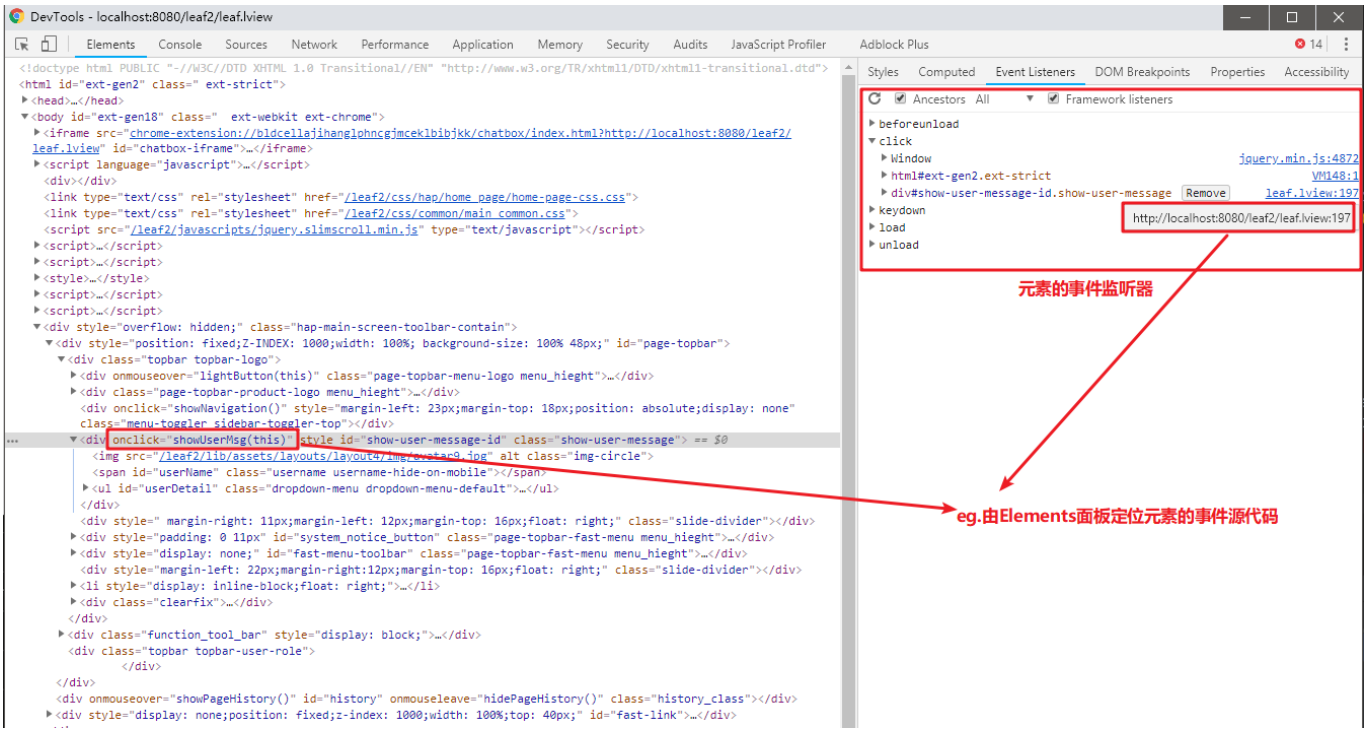
在 Styles 右侧的 Computed 窗格中可以查看：

- 元素的盒模型(双击值可编辑)
- 元素所有样式的**计算后最终值**(即最终实际应用到元素的值)
- 点开每一条最终值，可以看到所有该条样式的规则，以及代码来源
- 勾选**show all**选项，会同时列出元素**继承 / 默认**样式



Event Listeners

- 在 Event Listeners 窗格中，可以看到元素的事件监听器
- 例如"load","DOMContentLoaded","click"等，以及每个事件对应的事件处理函数



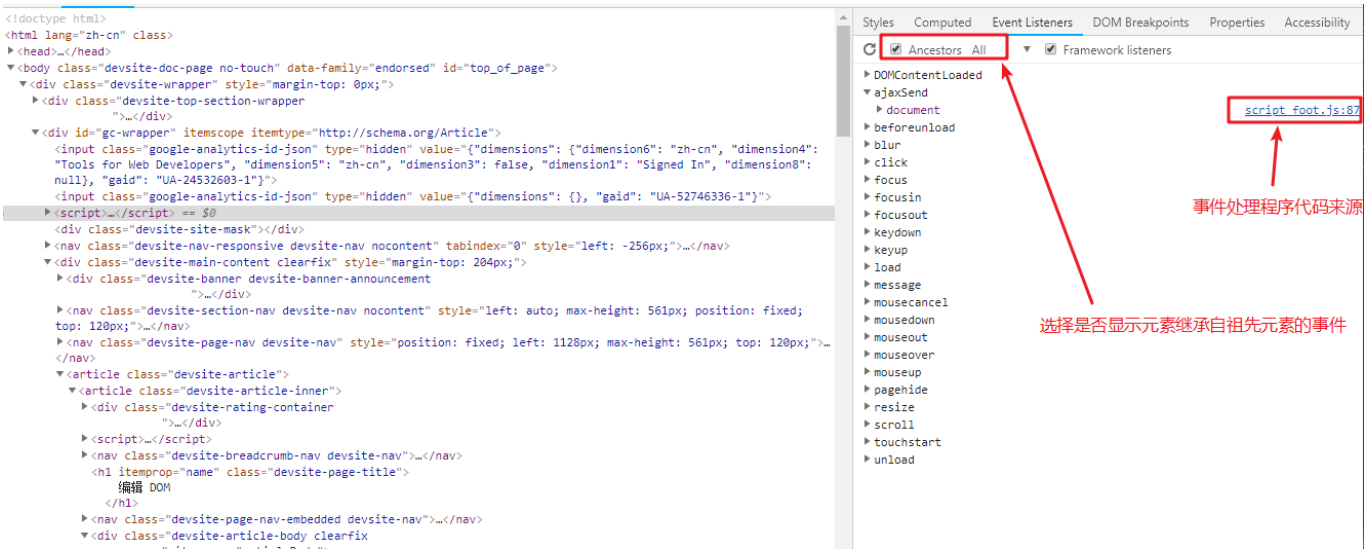
在源代码中加 行 **debugger 断点**，或者**debug(函数)断点**(Sources 面板会提及这两种断点)，是需要代码维护成本的，有时候还会忘记删除；

或者你想调试别人开发的 你不拥有源码的 网页；

这些时候可以利用 Event Listeners 窗格快速定位当前元素被绑定的所有的**事件函数代码**并调试。

DOM Breakpoints

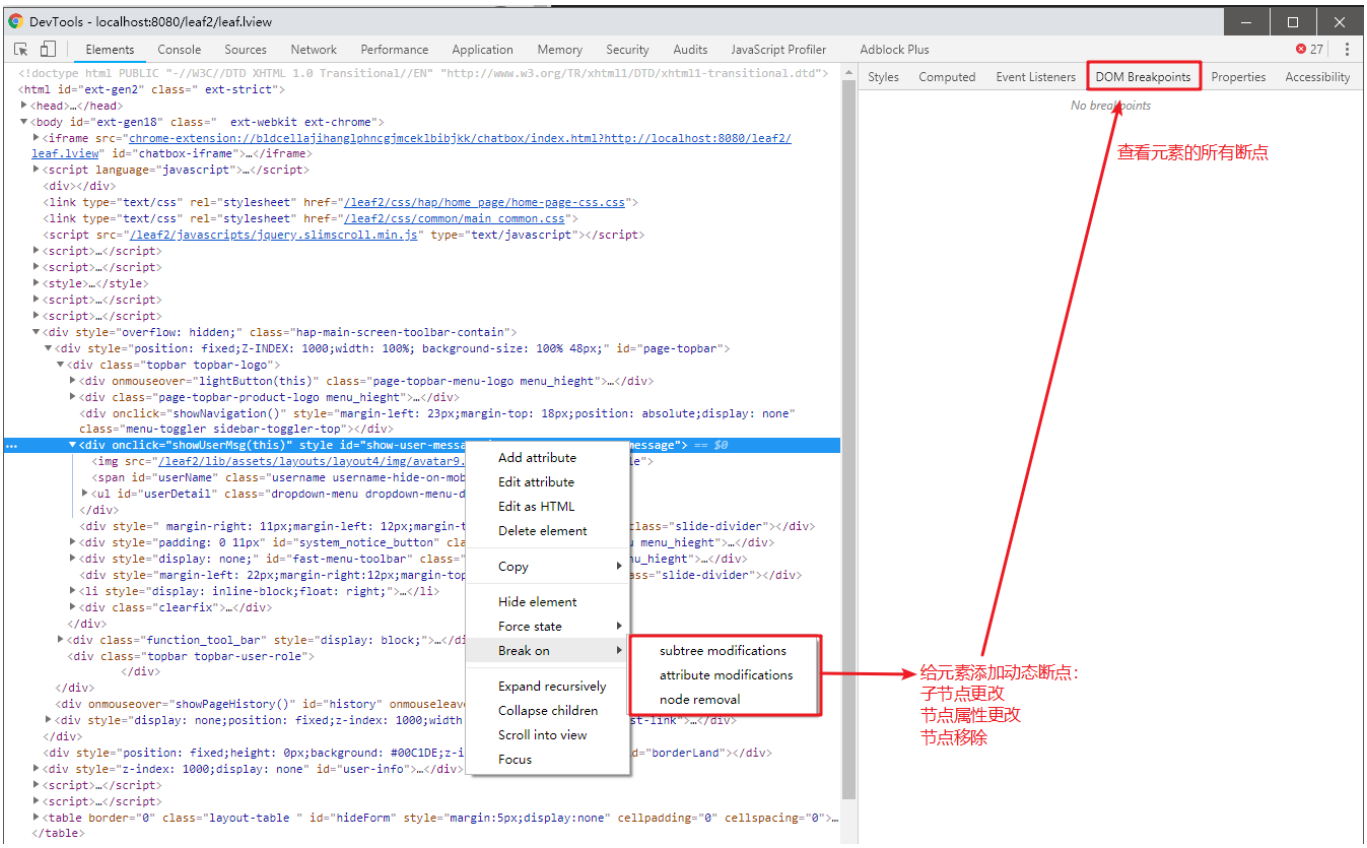
在面板右侧 DOM Breakpoints 中，可以查看**元素断点**



相应的在左侧 DOM 树右键点击元素，可以给元素添加断点

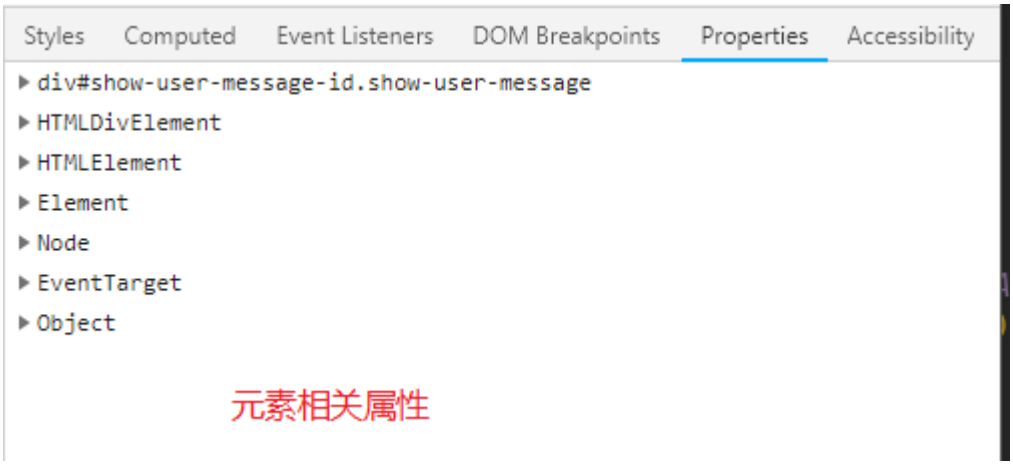
元素断点有三种类型：属性变更，子树变更，节点删除

例如添加“node removal”断点，就会在 有代码移除当前节点时，在当前行代码执行前暂停执行，并自动转换到 Sources 面板，以便做进一步调试



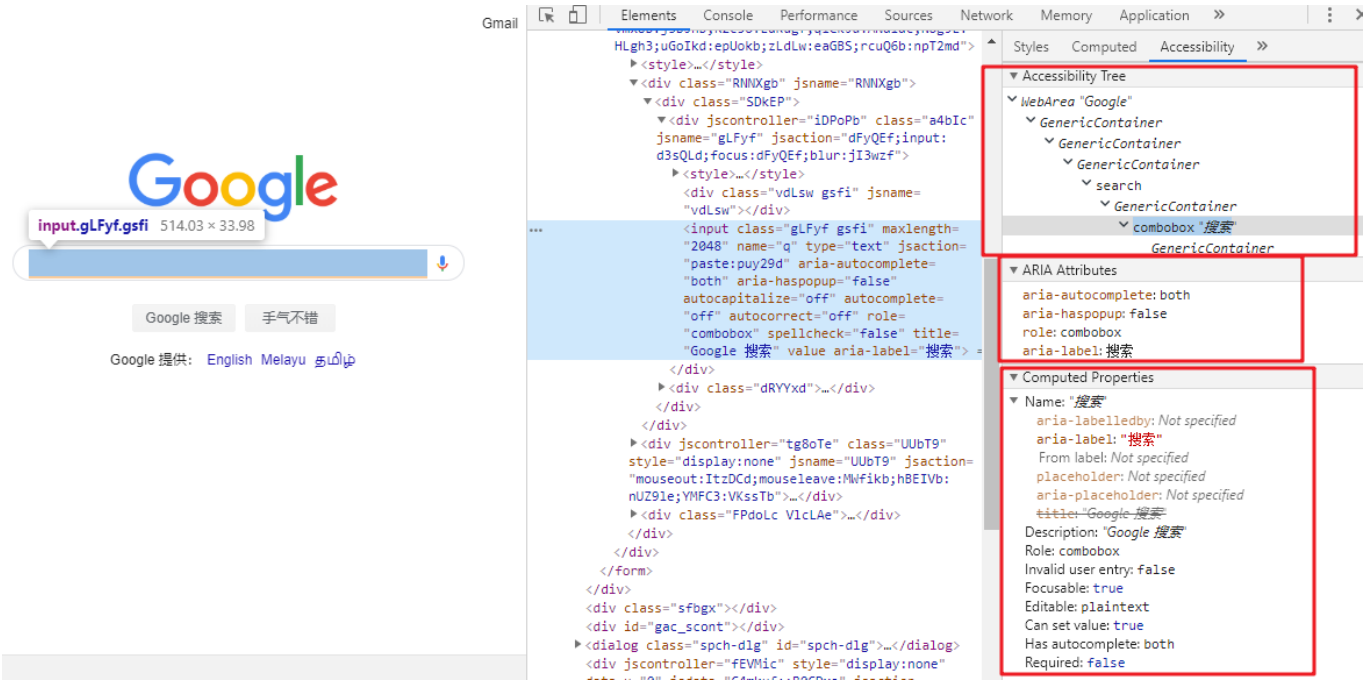
Properties

Properties 面板会列出元素 DOM 底层相关属性



Accessibility(可访问性)

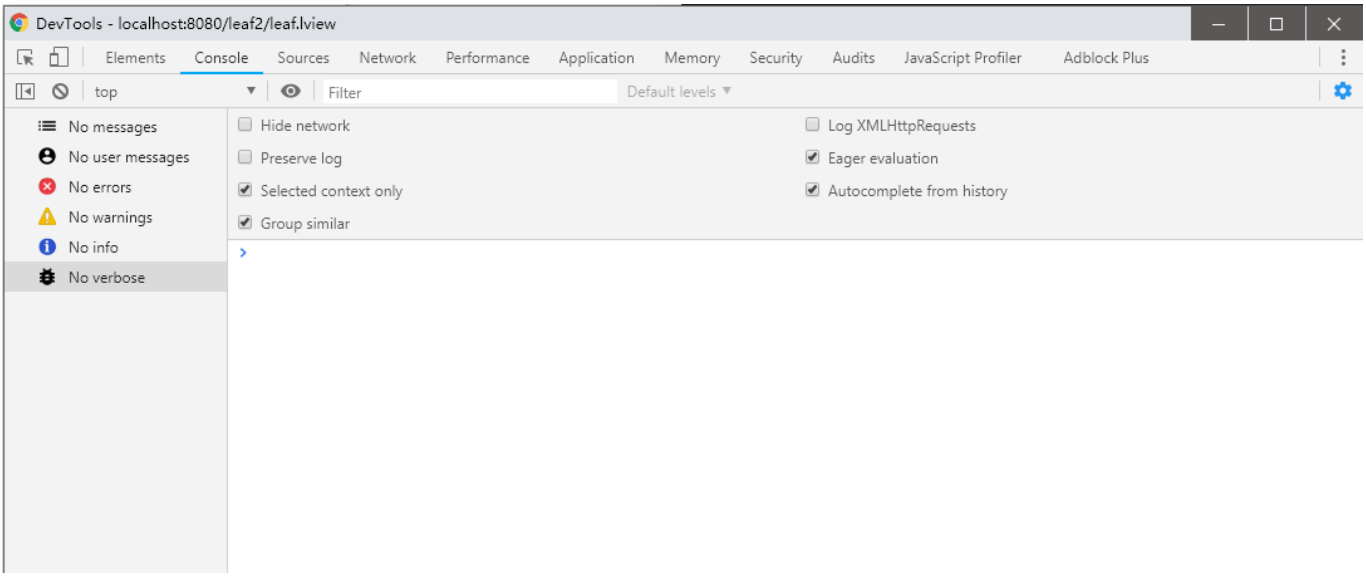
- 在辅助功能树中查看元素的位置(可访问性树/无障碍树是 DOM 树的子集。它只包含来自 DOM 树的元素，这些元素可以展示在屏幕阅读器中页面的内容。
- 查看元素的 ARIA 属性(ARIA 属性确保屏幕阅读器具有所需的所有信息，以便正确表示页面的内容。
- 查看元素的计算辅助功能属性(某些辅助功能属性由浏览器动态计算。可以在“ 辅助功能”窗格的“ 计算属性”部分中查看这些属性



Console 面板

Console 面板是浏览器的控制台，也是 Devtools 的灵魂。

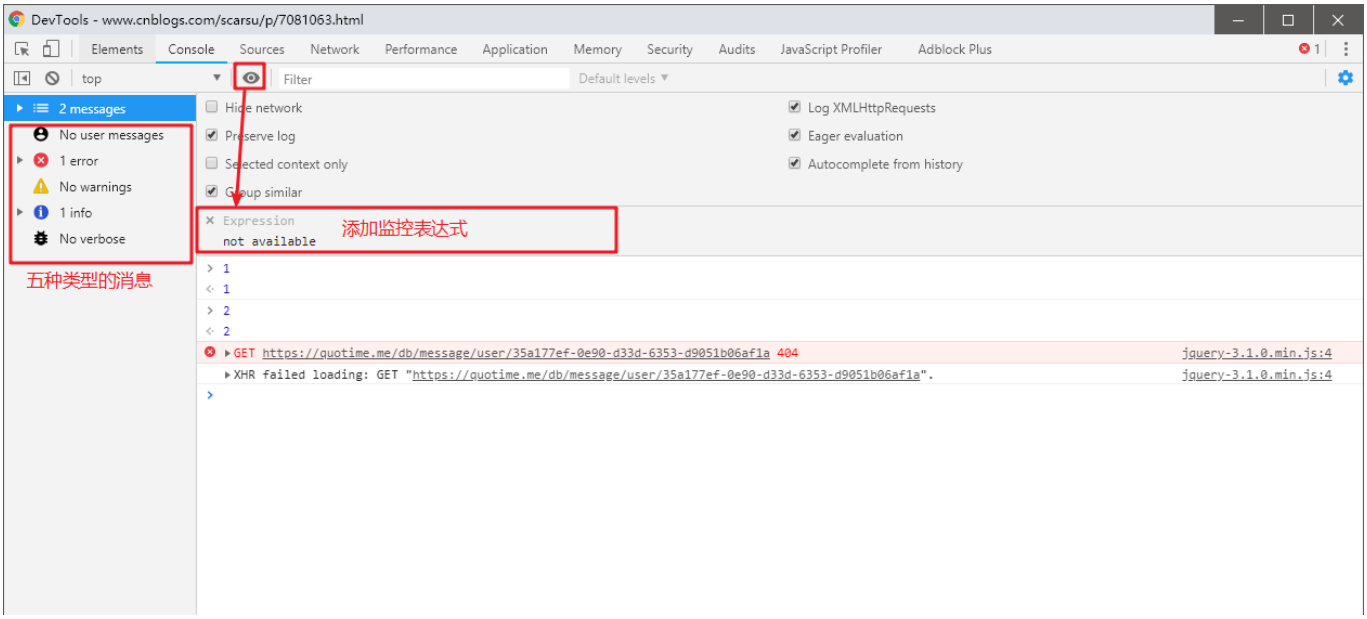
可以通过**设置->Show Console drawer**或者**Esc 快捷键**让 Console 在每个面板都能显示。



message

- 在 console 中，可以看到来自**浏览器/代码**的五种类型的信息：
  - user message
  - error
  - warning
  - info
  - verbose
- 相同的消息默认是堆叠的，可以通过 ctrl+shift+p 输入 time 命令或者设置中找到**timestamps**命令，给消息加上时间戳
- 通过选项**Log XMLHttpRequest**选择是否输出所有 XMLHttp 请求日志(可以监控页面所有 ajax 请求 定位其代码调用栈)

- 通过**Hide network**选择显示/隐藏网络请求的错误信息(例如 GET xxx 404)
- 通过**Preserve log**选择保留历史记录，即刷新页面后是否还显示先前的消息

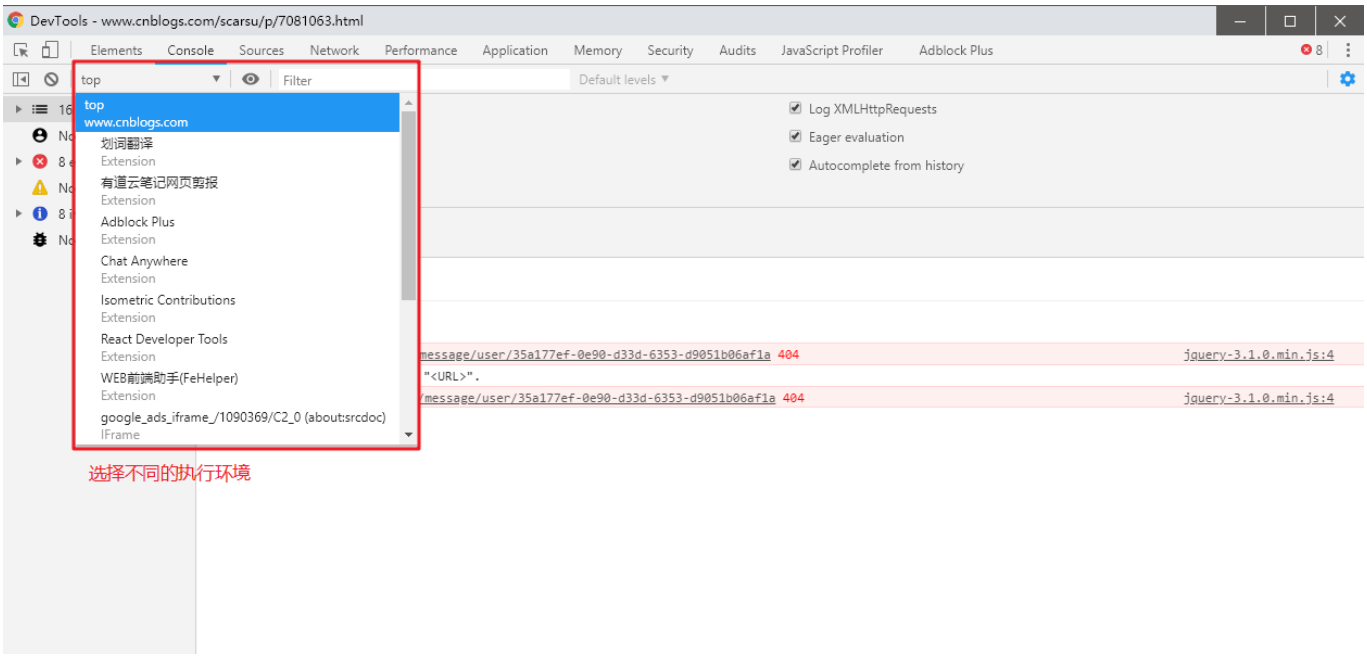


javascript 实时执行环境

- console 除了能输出调试信息，也是一个 javascript 实时执行环境。
- 可以直接在这里输入任何**全局变量名**/内置对象名/函数名，会得到相应的值输出;在调试环境下的断点内，可以获取**局部变量**值
- 右键选**Store as global variable**，可以将输出值存储为一个临时的全局变量
- 双击对象的属性值，可以直接更改这个对象（持久化的更改,因为 console 存储下来的是对象的引用）
- console 中输出的 dom 元素 -> 右键 -> reveal in elements：快速定位到元素面板中的当前元素

选择执行环境

可以通过左上的下拉列表，选择不同的**执行环境**



top 是最外层的顶级页面,其他的是 iframe 子页面

默认情况下

- 子 frame 中：

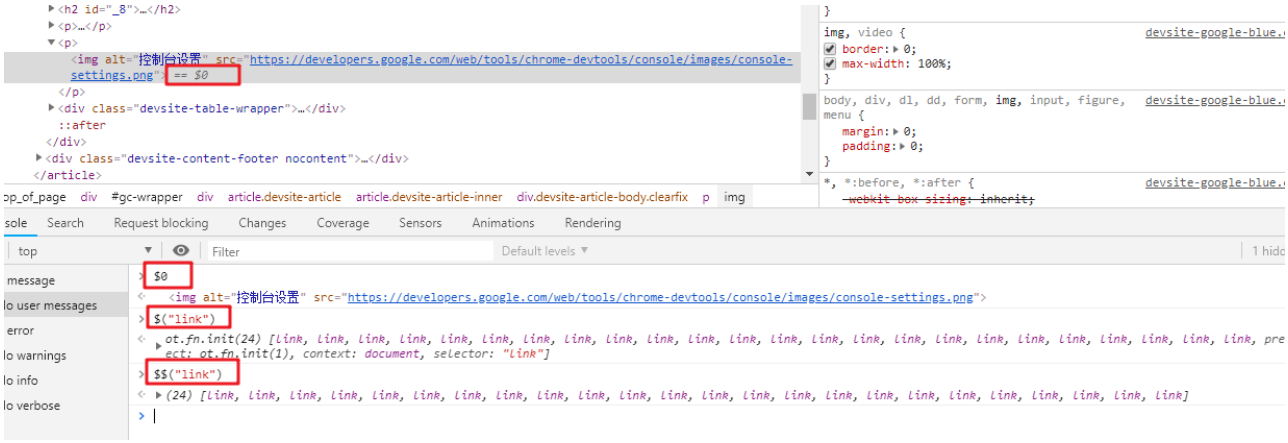
```
(window === self) === self.window;
//top.window是顶级页面top的全局变量window
```

- top frame 中：

```
(((window === self) === self.window) === top) === top.window;
```

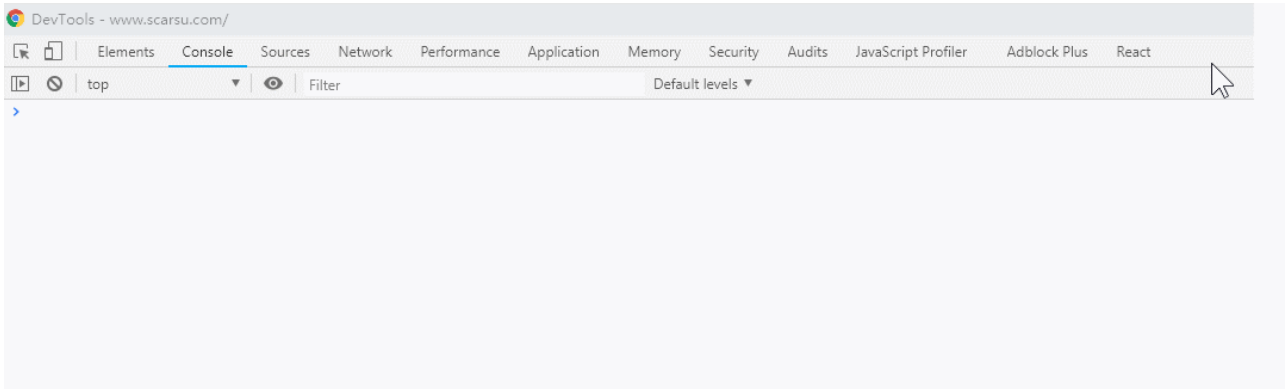
console 中的$符号：

- 可以通过$0，获取当前在 Elements 面板所选中的元素节点
- 如果 $ 在当前页面没被占用，可以用来替代 document.querySlector 方法使用
- $$ 是 document.querySelectorAll 方法的更佳替代，因为 document.querySelectorAll 返回的是 nodeList(NodeList)，而$$能直接返回数组(Array)



- $_可以引用上一次执行的结果

- 如果需要使用 npm 的包，可以安装 Console Importer 插件，用$i方法引入 npm 中的库



console 下的方法：

- 除了被用烂了的console.log()（当然 console.log 也有特别一点的用法）

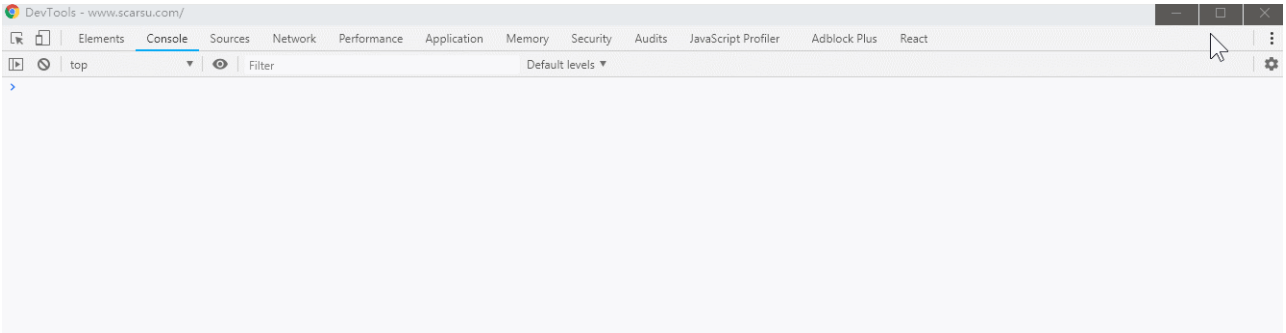
```
> 20:14:17.627 console.log("%cWatch this message!","color:red;background:green;font-size:3em")
  20:14:17.627 Watch this message!
< 20:14:17.632 undefined
>
```

给log方法要输出的字符串前加上%c，第二个参数就可以自定义输出样式

- console 对象还有 console.error/clear/debug/count/time/table/tarce 等等方法
- 直接在 console 中输入 console，可以看到 console 对象下的所有方法

```
> console
< ▼console {debug: ƒ, error: ƒ, info: ƒ, log: ƒ, warn: ƒ, …}
    ▶assert: ƒ assert()
    ▶clear: ƒ clear()
    ▶context: ƒ context()
    ▶count: ƒ count()
    ▶countReset: ƒ countReset()
    ▶debug: ƒ debug()
    ▶dir: ƒ dir()
    ▶dirxml: ƒ dirxml()
    ▶error: ƒ error()
    ▶group: ƒ group()
    ▶groupCollapsed: ƒ groupCollapsed()
    ▶groupEnd: ƒ groupEnd()
    ▶info: ƒ info()
    ▶log: ƒ log()
      memory: (...)
    ▶profile: ƒ profile()
    ▶profileEnd: ƒ profileEnd()
    ▶table: ƒ table()
    ▶time: ƒ time()
    ▶timeEnd: ƒ timeEnd()
    ▶timeStamp: ƒ timeStamp()
    ▶trace: ƒ trace()
    ▶warn: ƒ warn()
      Symbol(Symbol.toStringTag): "Object"
    ▶get memory: ƒ ()
    ▶set memory: ƒ ()
    ▶__proto__: Object
```

- 例如 console.table(obj)可以把你的对象以可视化的表格形式进行输出



- 例如可以使用 console.time()和 console.timeEnd()方法来测量时间差

```
> 19:40:30.426 console.time();
               let sum=0;
               for(let i=0;i<10000;i++){
                 a+=i;
               }
               console.timeEnd();
  19:40:30.432 default: 1.656005859375ms
```

- 另:console 命令行还内置了一些 API 方法，例如 queryObjects(),可以返回指定类型的对象下所有的实例化的对象


```
queryObjects(Object); //返回所有object对象
queryObjects(Function); //返回所有函数
queryObjects(Array); //返回所有数组
```

- monitor(function)方法来追踪函数调用信息，当函数被执行，会输出追踪信息

- monitorEvents(el,eventType)方法来追踪事件

Sources 源文件面板

在 Sources 面板你可以：

- Debug : 在源代码面板中可以设置**断点**来调试 JavaScript ， 比 console.log()调试更快速高效
- Devtools as IDE : 通过 Workspaces（工作区）连接本地文件来使用开发者工具的**实时编辑器**

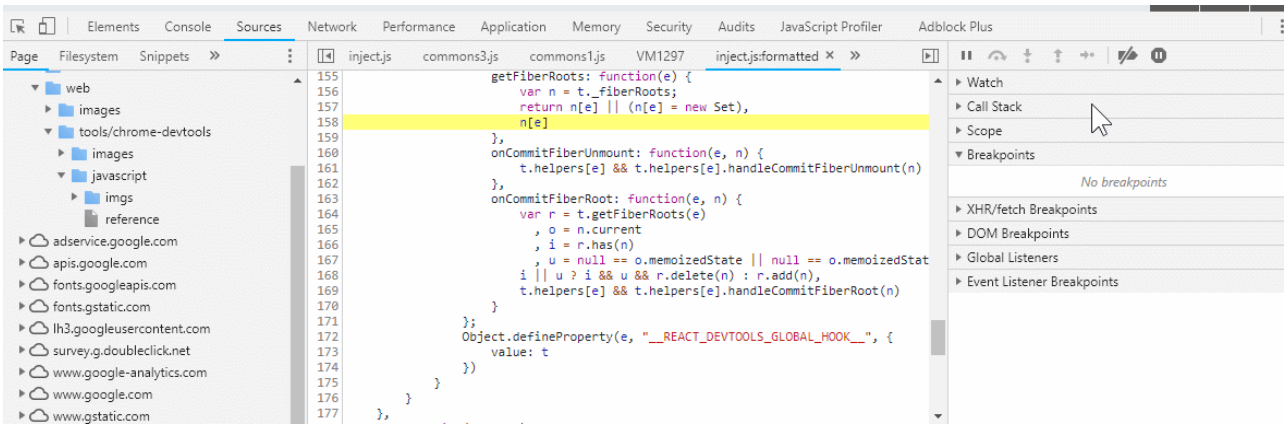
七种断点类型

1. 行断点：代码运行到当前行之前暂停执行

> 在源代码添加debugger关键字
> 或者
> 点击Sources面板中的源代码的行号

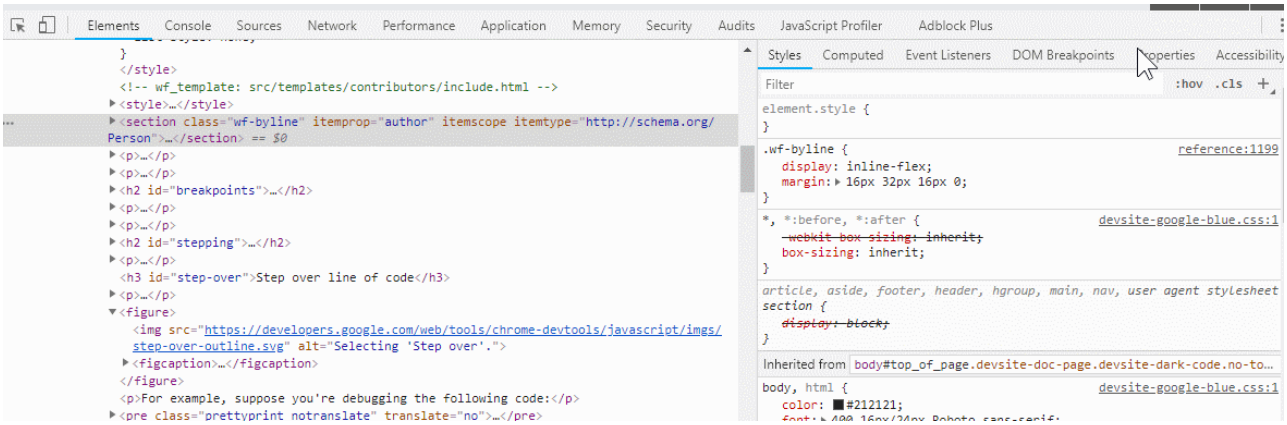
2. 条件行断点：当满足条件时才会触发该断点

> 右击Sources面板中的源代码的行号
> 选择“Add conditional breakpoint”



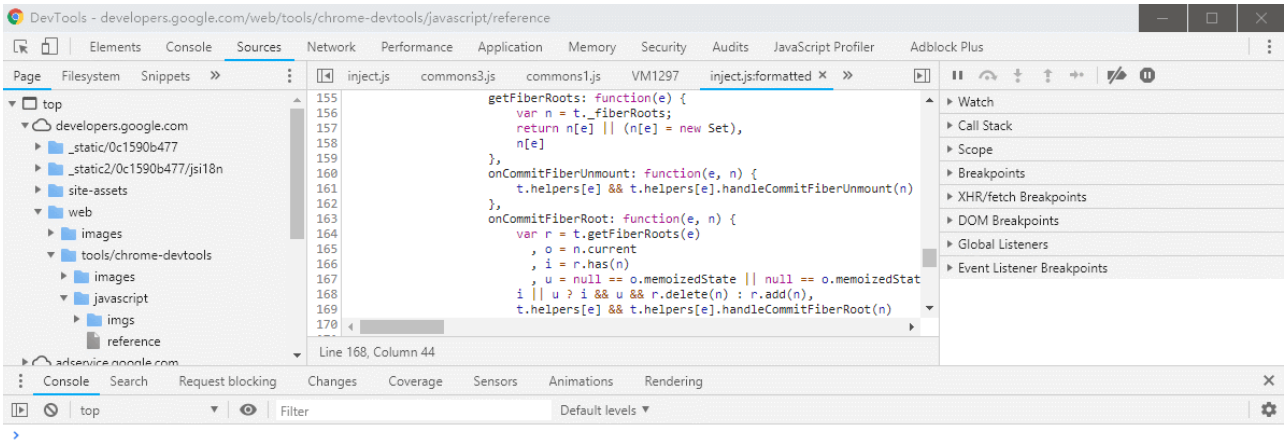
3. DOM 断点

> 即Elements面板提及过的三种DOM断点：
> - 节点属性断点
> - 节点删除断点
> - 子树变更断点



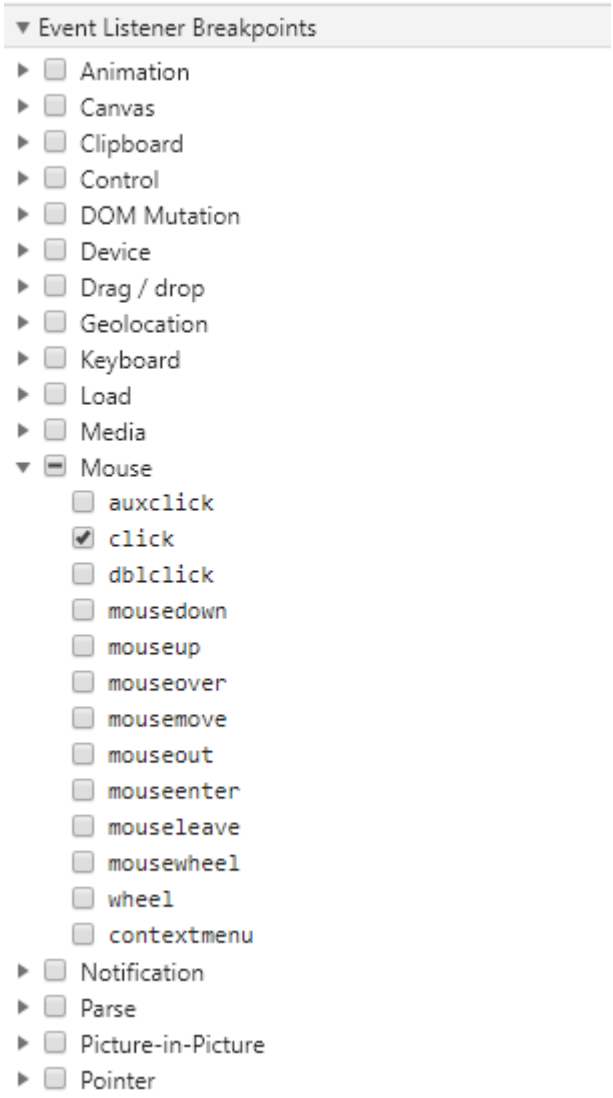
4. XHR/Fetch 断点

> 在页面发出XHR或Fetch请求前加断点

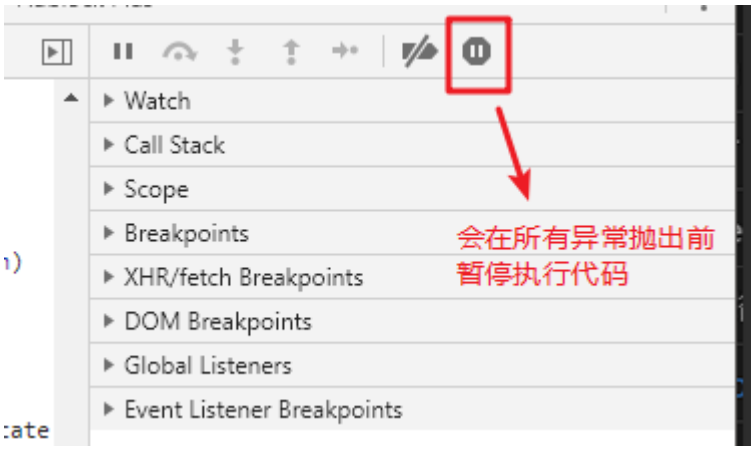


5. Event Listener 事件监听断点

> 可以在所有类型的事件函数被出发前加断点

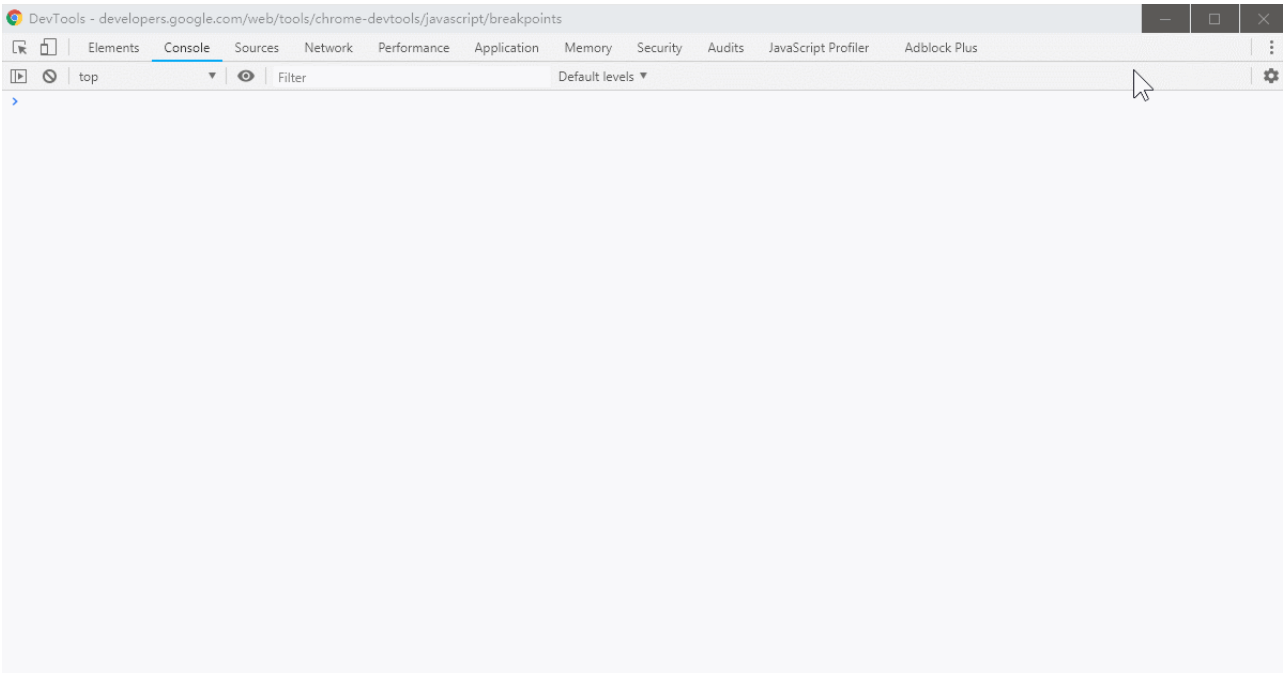


6. Exception 异常断点



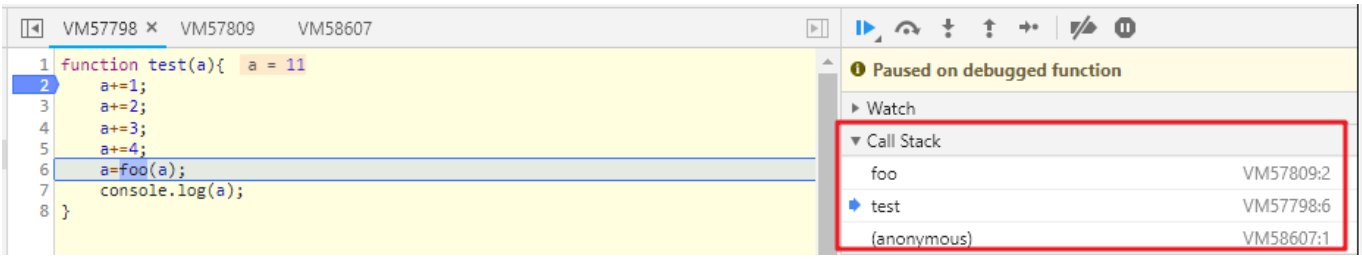
7. Function 函数断点

> 把想调试的函数名作为参数，调用debug()函数，可以在每次执行该函数前暂停执行代码

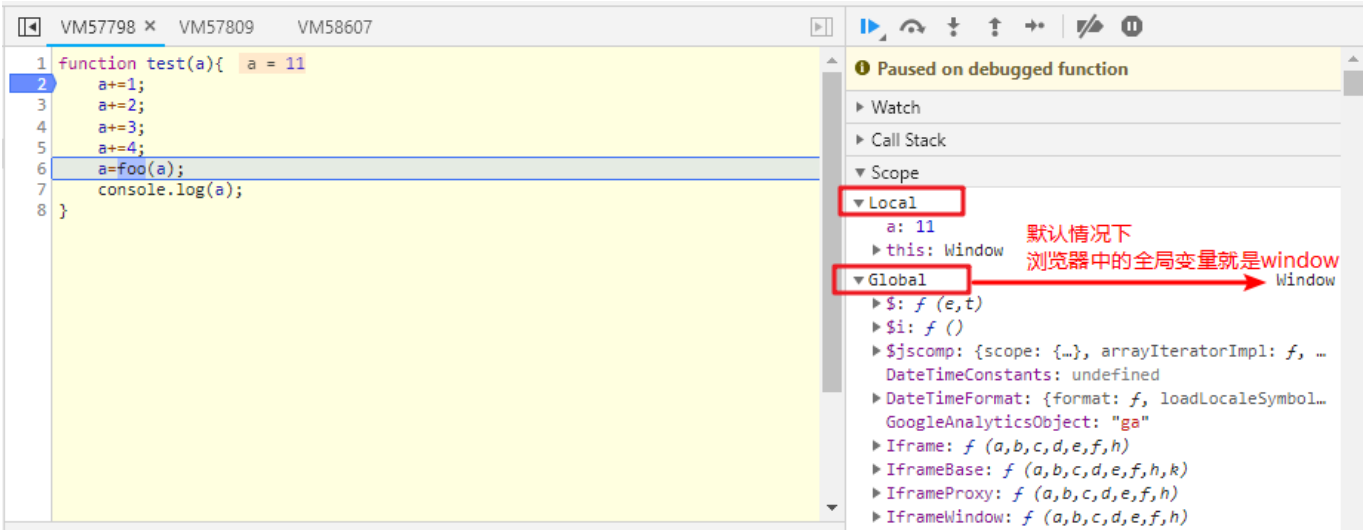


Debug

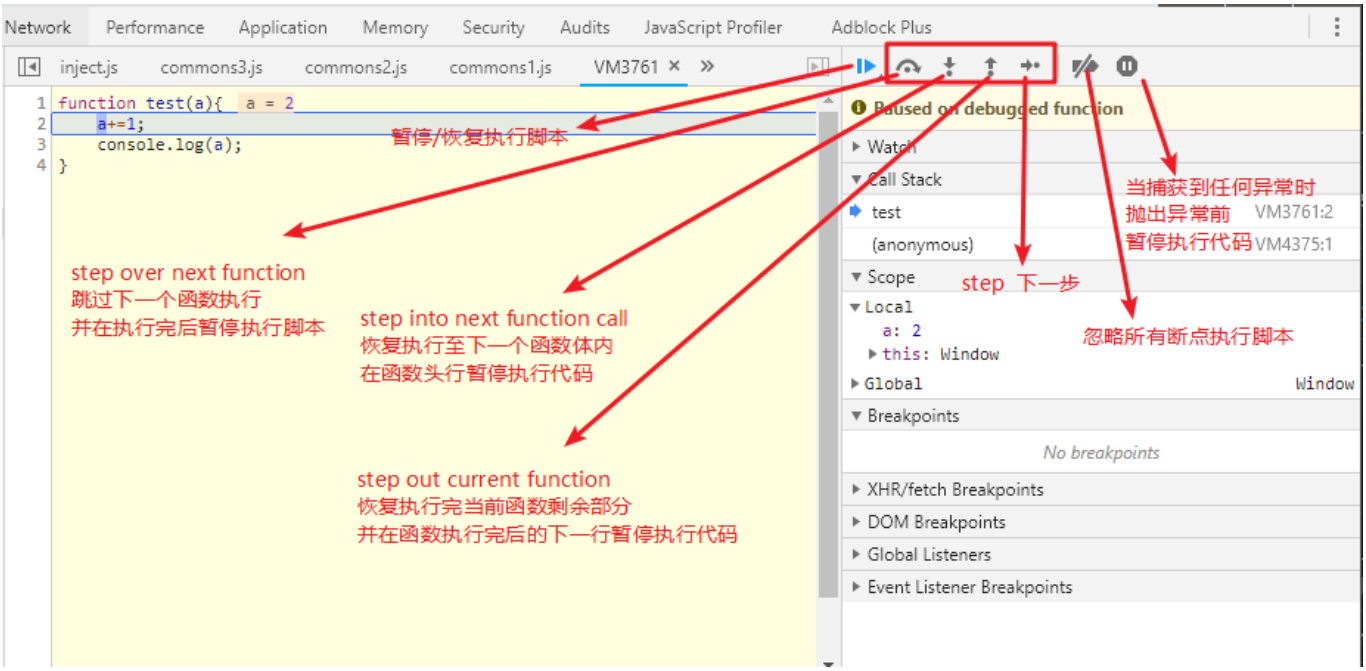
- 函数调用栈 Call Stack：Call Stack 是 time traveling 的，即点击栈中的任一节点，当前的作用域和局部变量等信息，都会模拟至该节点执行时的状态



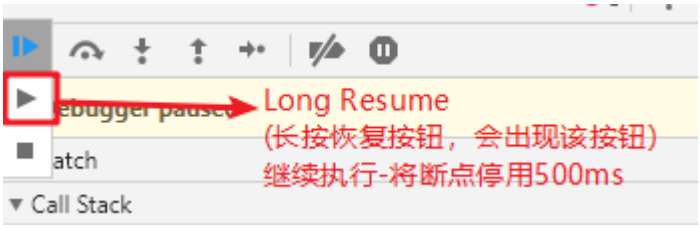
- 全局作用域 Global ，局部作用域 Local ，闭包作用域 Closure



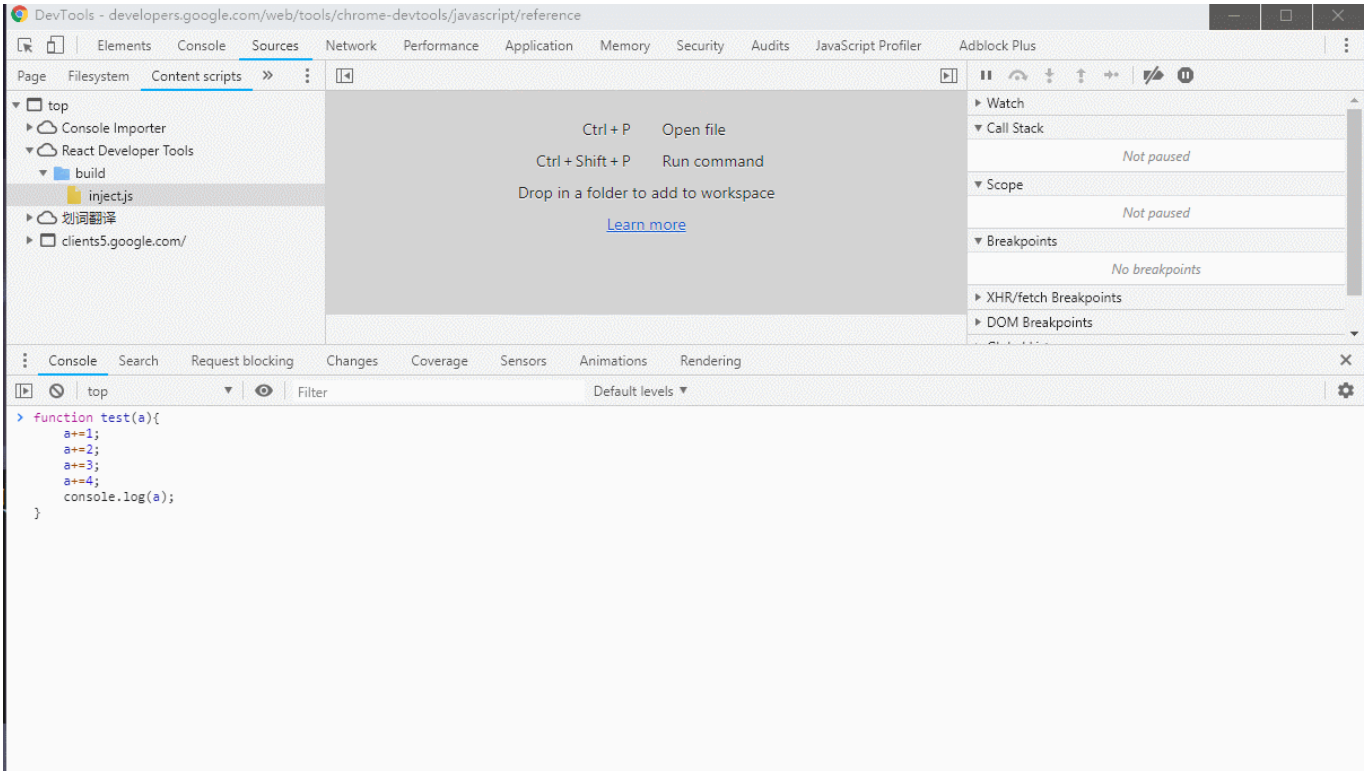
- step over next function
- step into next function
- step out current function
- step (与 step over/into 的区别就是，step 会优先尝试 step into，当没有可步入的代码时，就会执行 step over)



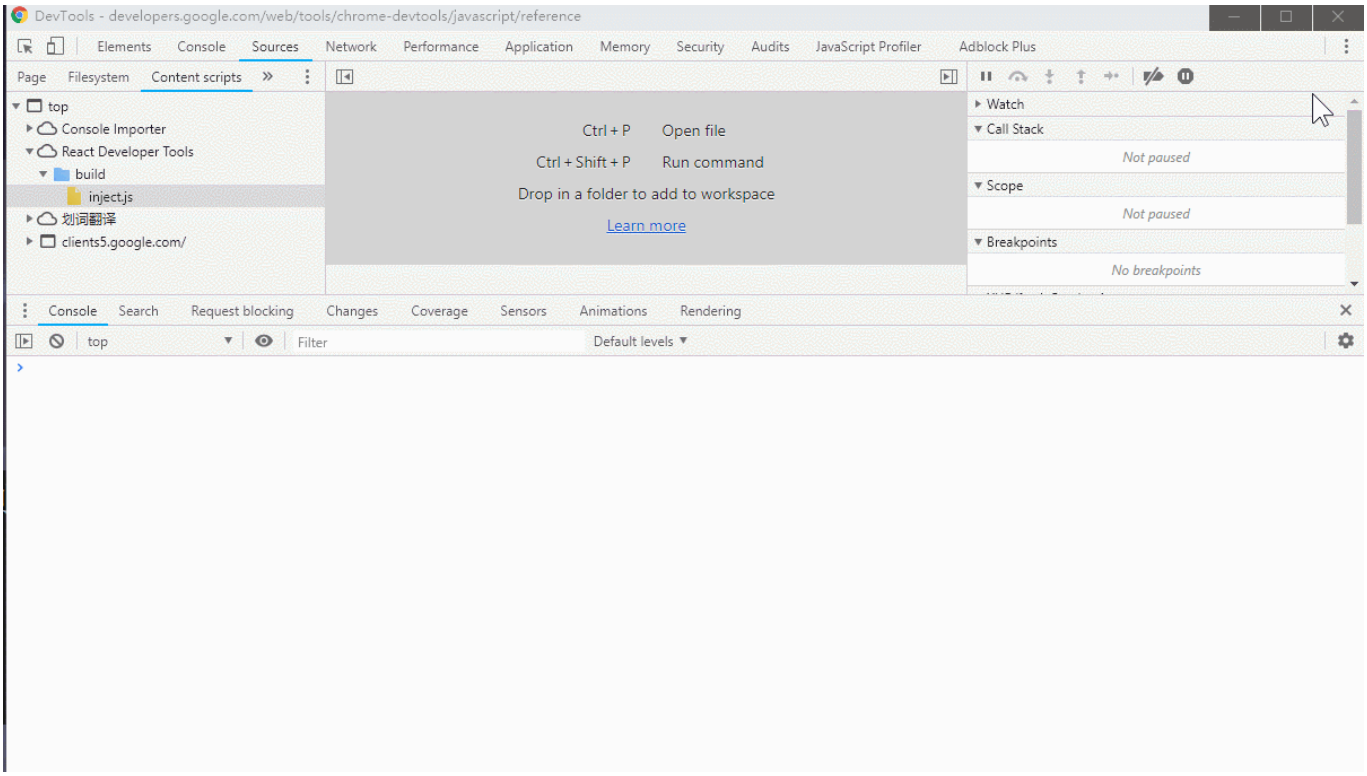
- long resume：恢复执行，并将断点停用 500ms



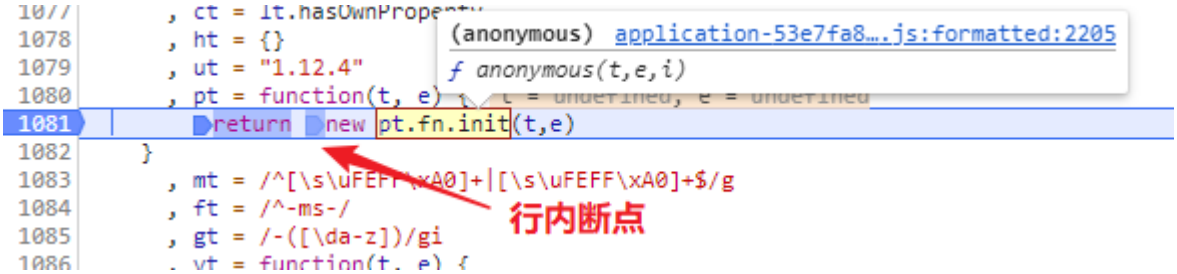
- Continue to here：继续执行至此行



- Restart Frame：重新执行函数调用堆栈中的某一帧

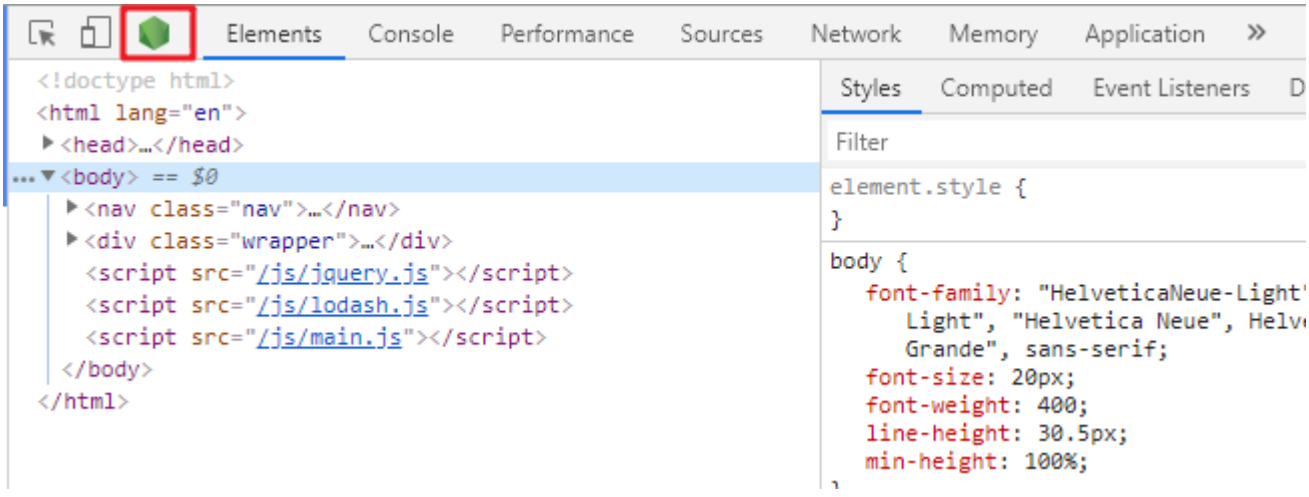


- 行断点内的多个箭头：行内断点（行内的，可 step into 的 执行点



Devtools Nodejs debug

- node 执行 js 文件，文件名前加--inspect 标志，启用浏览器 nodejs 调试



- 点击 devtools 中，左上角的 devices mode 右侧的绿色按钮，即可启用 node 服务端中的脚本调试
- 更多相关

BlackBox

- BlackBox 的用途：

  “BlackBox Script”可以在调试中忽略某些脚本(此处的 BlackBox 为动词)，在 Call Stack 堆栈中会将该脚本隐藏，单步调试时也不会步入脚本中的任何函数

```
function animate() {
prepare();
lib.doFancyStuff(); // A
render();
}
```

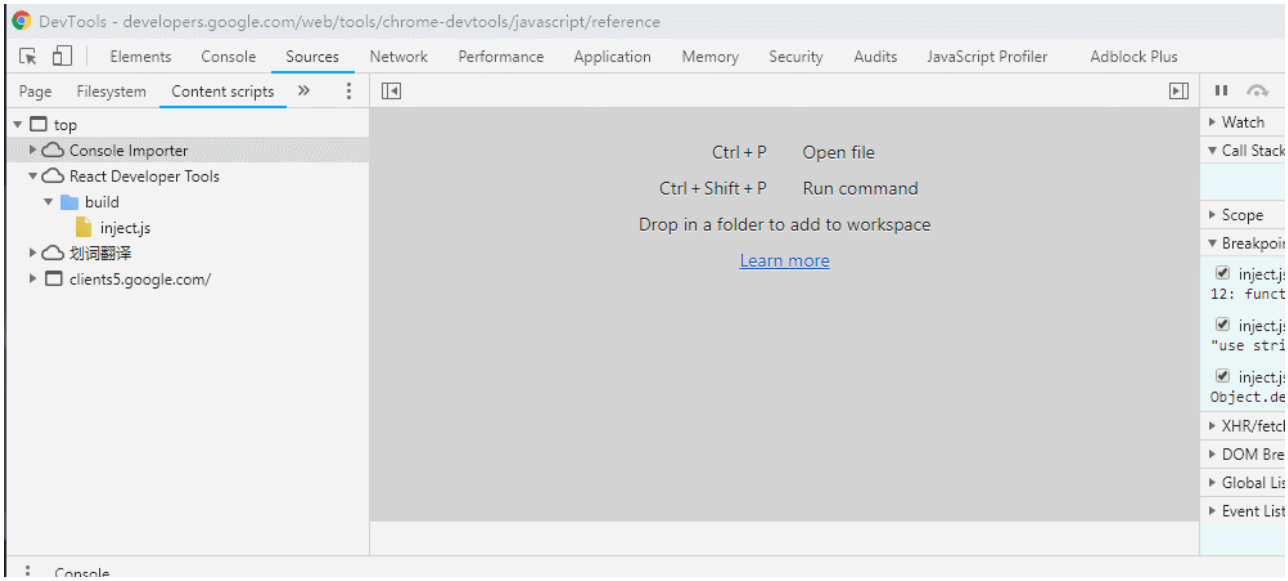
  例如以上代码的 A 行，调用的是第三方库的 doFancyStuff 函数

  如果我确认该第三方库没有 bug

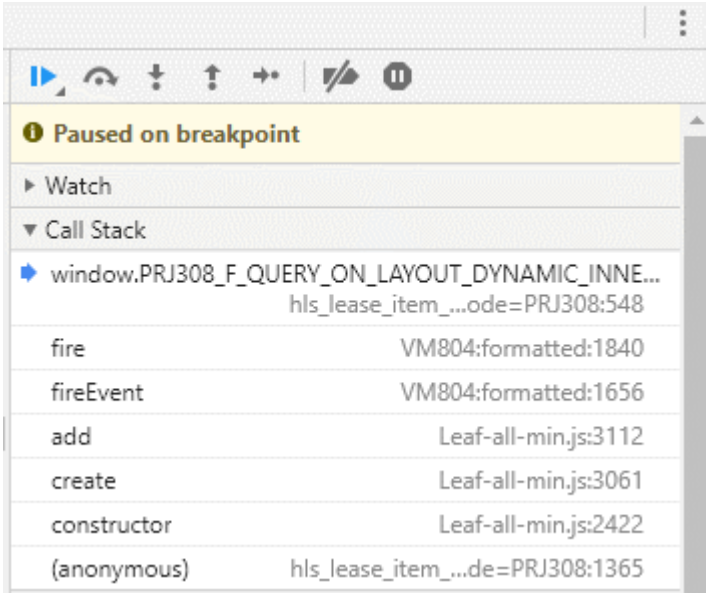
  就可以 BlackBox 整个第三方库的 js 脚本，在调试中跳过这些代码的执行

- 三种添加 BlackBox 的方法：

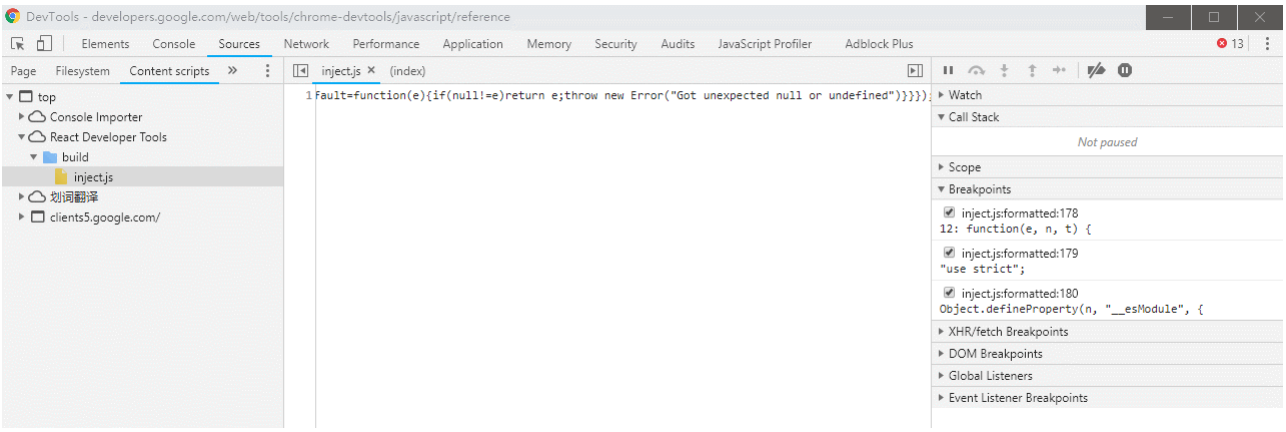
1. 在源代码窗格右键，选择"BlackBox Script"



2. 在 Call Stack 中右键某一帧，选择"BlackBox Script"



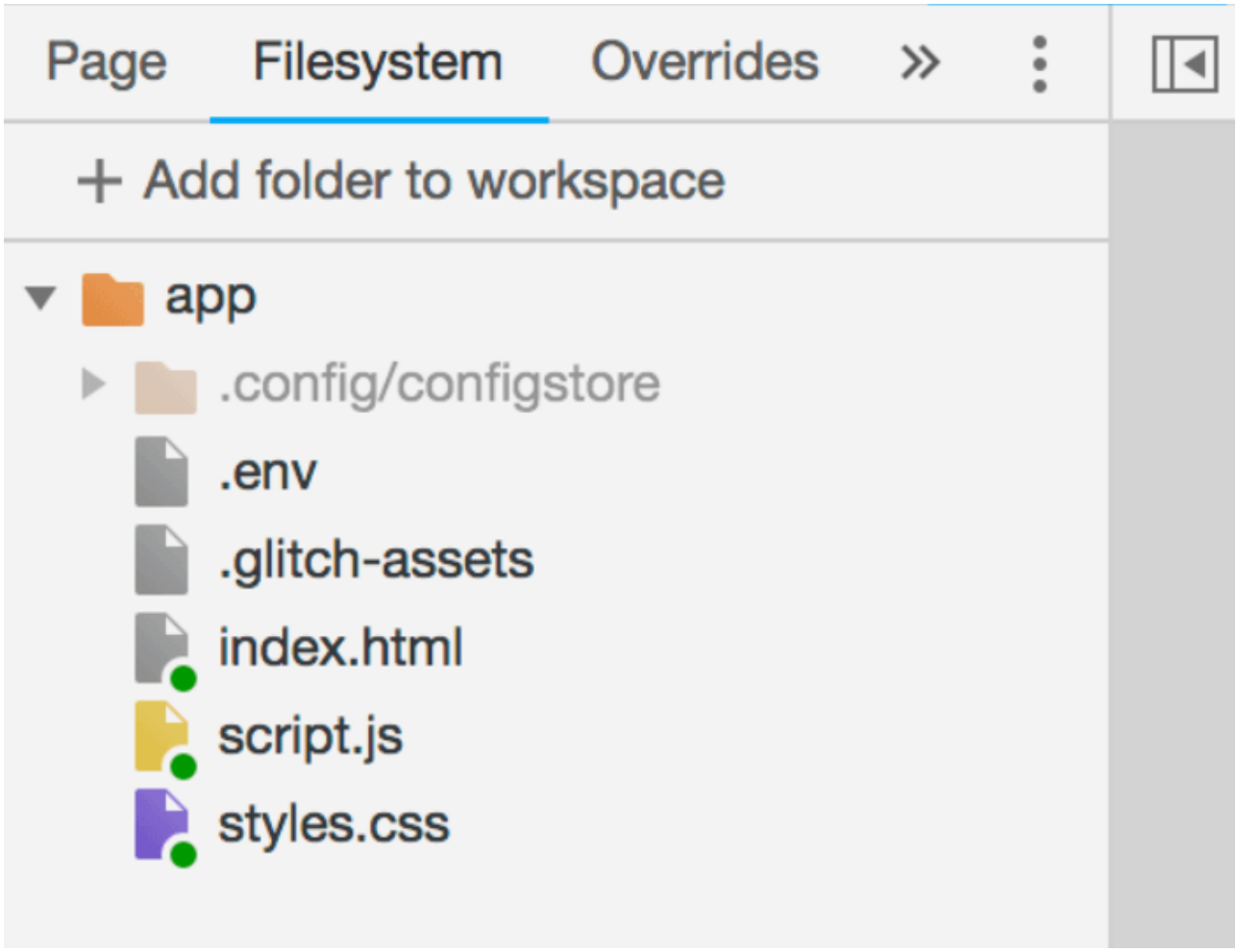
3. 在设置中的 Blackboxing 面板添加正则表达式匹配文件名

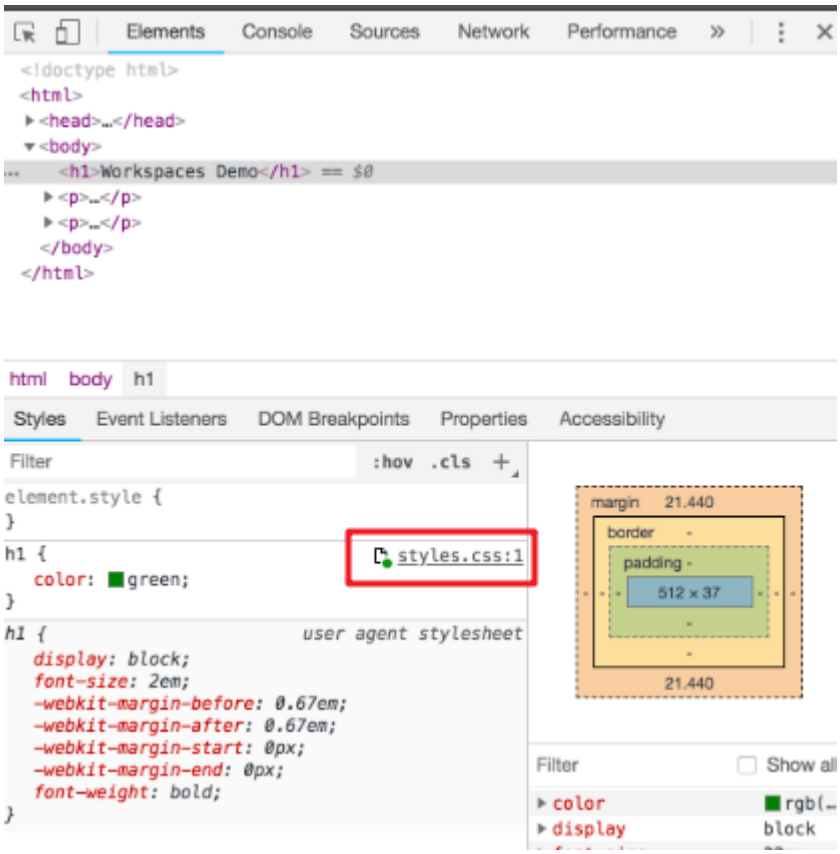


Workspace：Devtools as IDE 将更改持久化

- 在 sources 左侧的面板中选择`Filesystem`，点击`Add folder to workspace`，将你本地运行的站点的相关源文件添加到 Devtools 的工作区，会自动识别 Page 下和工作区下相对应的文件，在 devtools 更改文件并保存，即持久化保存（目前只支持自动识别，不支持添加映射）

- 绿标文件：成功的映射到本地的文件，在 Styles 和 Sources 中的文件名前，都会添加绿色圆点作为标识



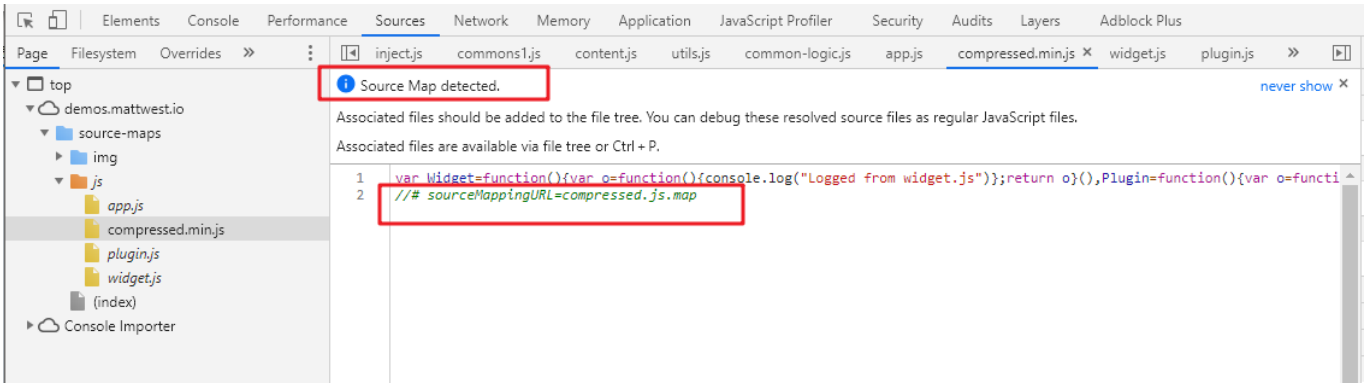


- 目前 Devtools 已经支持 sass/scss、UglifyJS、Grunt、Coffescript、Closure 等等，暂时还不支持 webpack，和其他现代的复杂框架，如 react
- 所有 sources 面板的文件，都可以右键选择`local modifications`，查看所有更改

- 对 DOM 树的更改不会持久化至 html 文件：因为 dom 的最终表现，受到 html、css、javascript 的共同影响，DOM 树 !== HTML，因此可以在 sources 中直接更改 html 文件并保存

Source Map

- 组合/压缩 css,js 文件是常见的性能优化方案，但是会对开发调试造成困扰
- Source Map 用于将生产代码映射至源代码，Chrome 和 firefox 都内置了对 Source Map 的支持
- 在 Chorme devtools 中，settings -> preferen -> sources 中，选中`Enable Javascript source maps`和`Enable CSS source maps`
- source map 映射信息存在 json 对象中，保存在 .map 文件中，可以由编译程序添加注释`//# sourceMappingURL=/path/to/script.js.map`至生产文件末尾，也可以由服务端在响应头中添加`X-SourceMap: /path/to/script.js.map`，将 map 文件与生产文件对应。更多关于 source map 的介绍

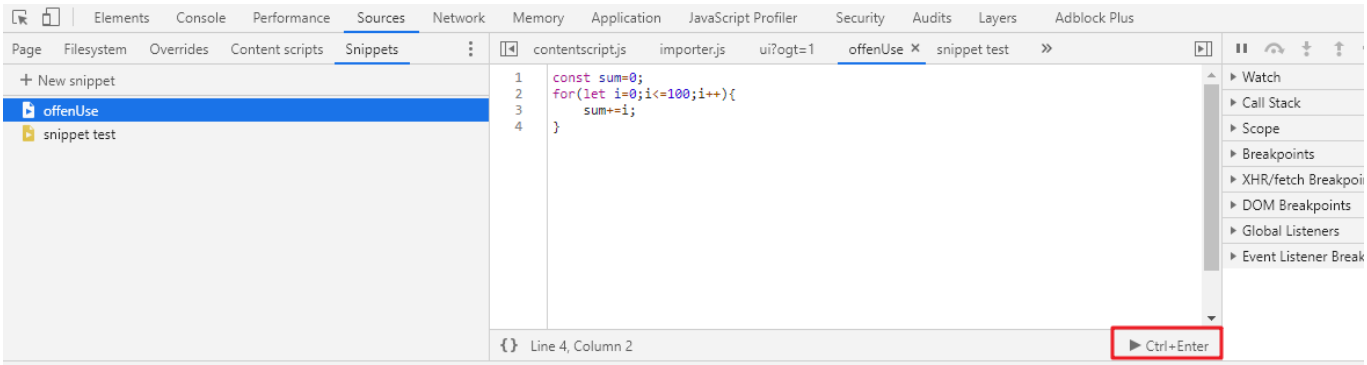


Local Overrides

- 通过 Local Overrides，可以在 DevTools 中进行更改，并在页面加载后保留这些更改
- 在 Sources 面板左侧选择 Overrides，指定 DevTools 应保存更改的目录，当在 DevTools 中进行更改时，DevTools 会将修改后的文件的副本保存到所选的本地目录中，重新加载页面时，DevTools 提供本地修改的文件，而不是请求的网络资源。
- 与 Workspace 相似的，不支持保存对 DOM 树的更改，需要直接更改 html 源文件。

Snippets 代码片段

- 在 Sources 面板左侧选择 Snippets，或`crlt shift p`输入 snippet 打开 Snippets 面板，可以创建并保存常用的代码片段，和用 gist 类似
- snippets 中，选中代码并`ctrl enter`，或点击右下角的执行按钮，即可执行代码片段



Content scripts

- 这部分脚本是浏览器插件的脚本，在特定网页的上下文中运行。（与插件运行在服务端的脚本，页面上引用的脚本，页面上 script 中的内嵌脚本都不同
- 插件在服务端的脚本可以访问所有 WebExtension JavaScript API，但它们无法直接访问网页内容。
- Content scripts 只能访问 WebExtension API 的一小部分，但它们可以使用消息传递系统与后台脚本进行通信，从而间接访问 WebExtension API。
- 如果有浏览器插件相关的工作，可以更深入研究，不赘述。

Network 面板

- 默认情况下，只要 DevTools 处于打开状态，DevTools 就会在 Network 面板中记录所有网络请求。
- 左上红点按钮：停止记录网络请求
- 第二个按钮：清空请求记录
- 录像按钮：页面加载时捕获屏幕截图
- 过滤按钮：显示/隐藏 过滤条件行
- View 中的两个按钮：第一个是切换请求列表中每行的显示样式（大小请求行），第二个是显示/隐藏瀑布图
- Group By Frame：是否根据不同的 frame 分类显示请求
- Preserve Log：保存显示跨页面的加载请求
- Disable Cache：禁用浏览器缓存，模拟新用户打开页面的体验
- Offline 是模拟断网离线的状态，其后的下拉框可以选择模拟其他网络状况，比如 2G,3G

筛选请求

- filter 文本框中可输入请求的属性 对 请求进行过滤，多个属性用空格分隔
- 支持过滤的属性：
    - domain。 仅显示来自指定域的资源。 可以使用通配符字符 (*) 纳入多个域。 例如，*.com 将显示来自以 .com 结尾的所有域名的资源。 DevTools 会使用其遇到的所有域填充自动填充下拉菜单。

    - has-response-header。 显示包含指定 HTTP 响应标头的资源。 DevTools 会使用其遇到的所有响应标头填充自动填充下拉菜单。
    - is。 使用 is:running 可以查找 WebSocket 资源。
    - larger-than。 显示大于指定大小的资源（以字节为单位）。 将值设为 1000 等同于设置为 1k。
    - method。 显示通过指定 HTTP 方法类型检索的资源。 DevTools 会使用其遇到的所有 HTTP 方法填充下拉菜单。
    - mime-type。 显示指定 MIME 类型的资源。 DevTools 会使用其遇到的所有 MIME 类型填充下拉菜单。
    - mixed-content。 显示所有混合内容资源 (mixed-content:all)，或者仅显示当前显示的资源 (mixed-content:displayed)。
    - scheme。 显示通过未保护 HTTP (scheme:http) 或受保护 HTTPS (scheme:https) 检索的资源。
    - set-cookie-domain。 显示具有 Set-Cookie 标头并且 Domain 属性与指定值匹配的资源。 DevTools 会使用其遇到的所有 Cookie 域填充自动填充下拉菜单。
    - set-cookie-name。 显示具有 Set-Cookie 标头并且名称与指定值匹配的资源。 DevTools 会使用其遇到的所有 Cookie 名称填充自动填充下拉菜单。
    - set-cookie-value。 显示具有 Set-Cookie 标头并且值与指定值匹配的资源。 DevTools 会使用其遇到的所有 Cookie 值填充自动填充下拉菜单。
    - status-code。 仅显示 HTTP 状态代码与指定代码匹配的资源。 DevTools 会使用其遇到的所有状态代码填充自动填充下拉菜单。
- 例如：mime-type:image/gif larger-than:1K 显示大于一千字节的所有 GIF
- `Hide Data URLs`：隐藏 data 类型的 url

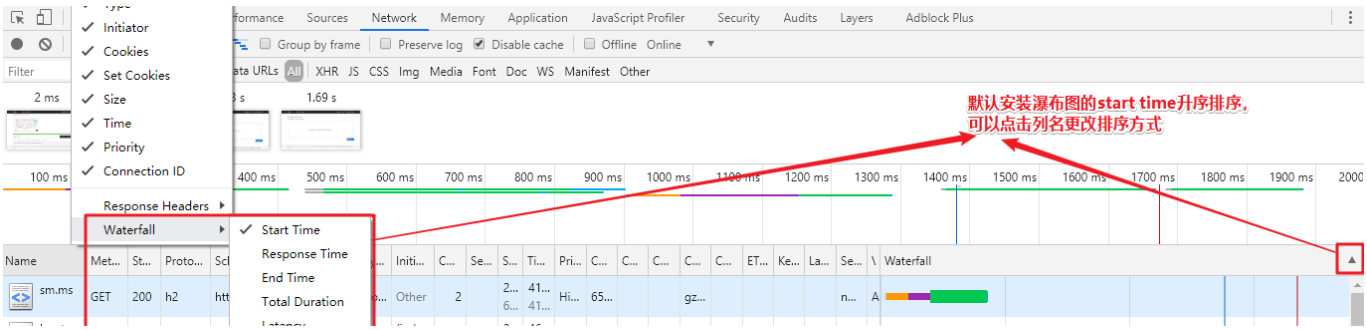
瀑布图

- 瀑布图按时间线展示所有请求
- 可以用鼠标拖动选中一段时间，只查看改时间线内的请求
- 瀑布图中有两条竖线，一条蓝色，代表DOMContentLoaded事件发生的事件，一条红色代表load事件发生的时间点

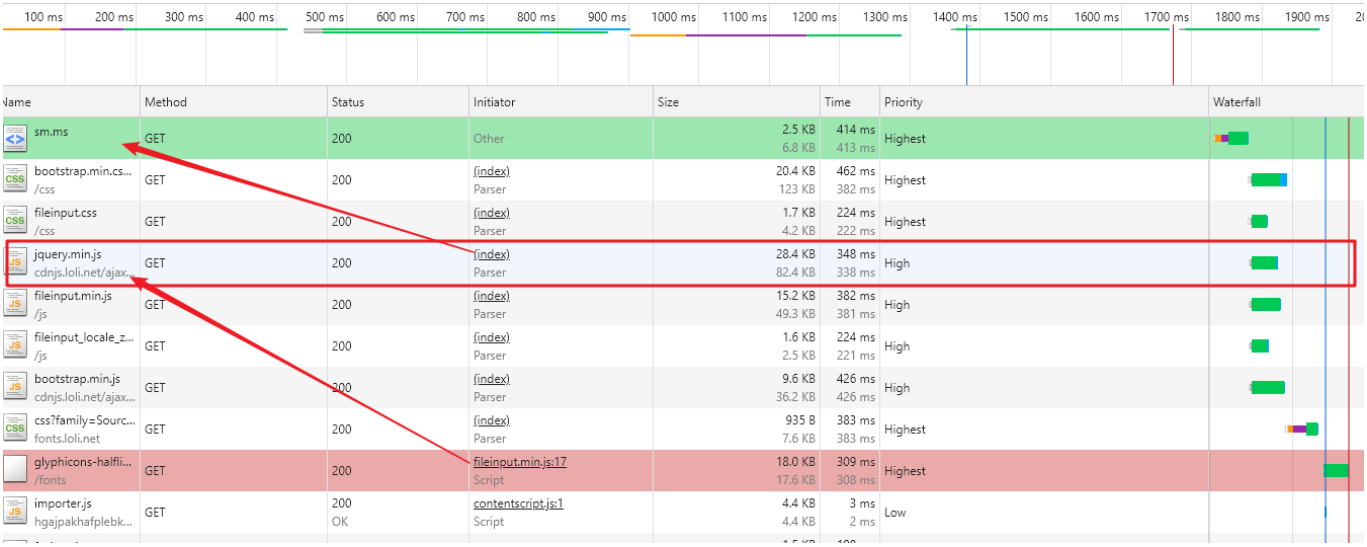
分析请求/请求列表

- 重播请求：右键点击 Requests 表格中的请求 -> `Replay XHR`
- 手动清除浏览器缓存：右键点击 Requests 表格中的任意位置 -> 选择 `Clear Browser Cache`
- 手动清除浏览器 Cookie：右键点击 Requests 表格中的任意位置 -> 选择 `Clear Browser Cookies`
- 自定义列表中展示的列

- 请求行排序，默认按照瀑布图 start time 升序排序，即请求发起的时间点：



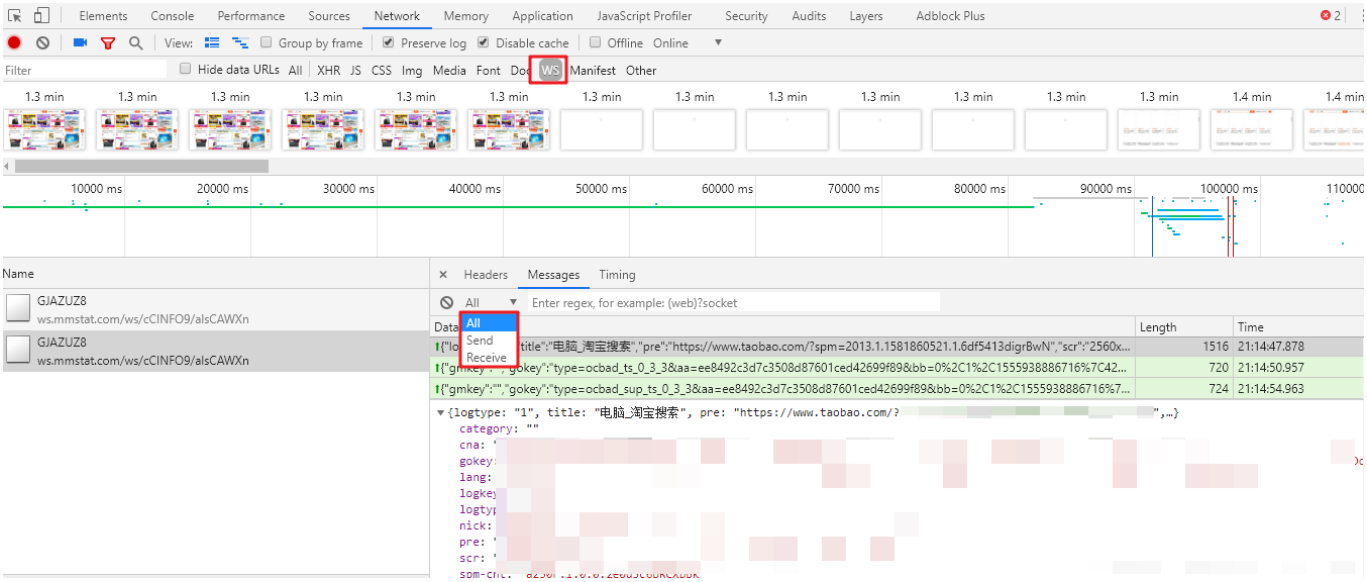
- 每条请求，可以看到网络请求以及被清华求资源的全部信息：
    - 请求的一般信息：url、HTTP 方法(GET POST 等)、状态码、ip 地址
    - 请求相关：请求头、Initiator、Priority
    - 响应相关：响应头、响应内容
- Initiator：请求的来源/发起者。parser：一般来自解析器解析到的 html 页面内的请求；script：来自脚本文件的请求。鼠标悬浮到 Initiator 列中的文件名上，可以看到发起当前请求的堆栈轨迹，点击文件名，可以定位到直接发起请求的代码
- 两个 size：在 size 列中，有两个数值，上面的较小值代表下载到的资源的大小，下面的较大值是资源解压后的大小。（例如 在 Content-Encoding 中可以看到的 gzip 和 br）
- 按住`shift`鼠标悬浮在请求行上，变绿色的行是当前行的发起者，红色的行是当前行的依赖项。



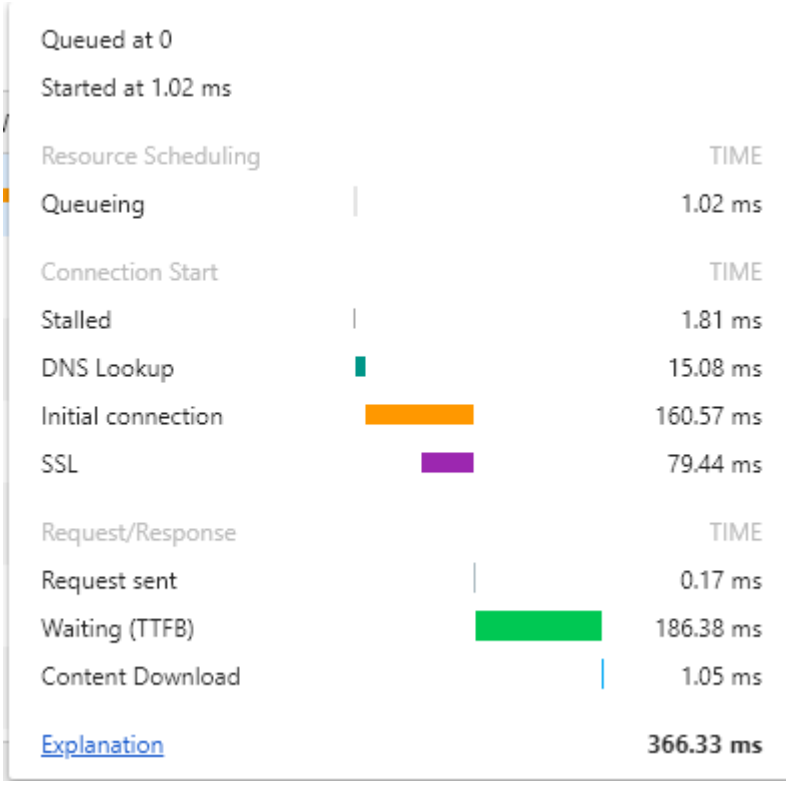
- Priority：High,Highest,Low。根据时间线中的蓝线和红线（DOMContentLoaded 和 load），以及请求的优先级，可以从结果的角度观察浏览器的加载流程。

Websocket

- 在 network 的 filter 条件后，选择ws类型的请求，即可看到所有 Websocket 请求
- 在请求详情的 Message 栏中，可以看到 wensocket 全双工通信中客户端接收和发送的信息



Color Code：瀑布图中的几种颜色与代码



- Queueing 排队，请求未发出，正在等待。 浏览器在以下情况下对请求排队：
  - 存在更高优先级的请求。
  - 此源已打开六个 TCP 连接，达到限值。 仅适用于 HTTP/1.0 和 HTTP/1.1（在 HTTP1 下浏览器一次最允许 6 个 TCP 连接，超出 6 个，就要 queue 排队)(优化 web 性能->避免 queue->合并资源请求）
  - 浏览器正在短暂分配磁盘缓存中的空间
- Stalled/Blocking 停滞/阻塞，请求仍未发出。请求可能会因 Queueing 中描述的任何原因而停止。
- DNS Lookup dns 查找，浏览器正在解析请求的 IP 地址，每次有指向新 domian 的请求时，会有 dns 查找的时间消耗。
- Proxy negotiation 代理协商。 浏览器正在与代理服务器协商请求。
- initial connection/connecting 正在初始化连接 或 正在连接，包含 tcp 的三次握手的时间
- SSL 完成 SLL 握手所需要的时间
- Request sent/senting 正在发送请求，发请求所占的时间，通常只有几分之一毫秒。
- ServiceWorker Preparation。 浏览器正在启动 Service Worker。
- Request to ServiceWorker。 正在将请求发送到 Service Worker。
- Waiting (TTFB)。 浏览器正在等待响应的第一个字节。 TTFB 表示 Time To First Byte（至第一字节的时间）。 此时间包括 1 次往返延迟时间及服务器准备响应所用的时间。
- Content Download。 浏览器正在接收响应。
- Receiving Push。 浏览器正在通过 HTTP/2 服务器推送接收此响应的数据。
- Reading Push。 浏览器正在读取之前收到的本地数据。

DOMContentLoaded 和 load 事件

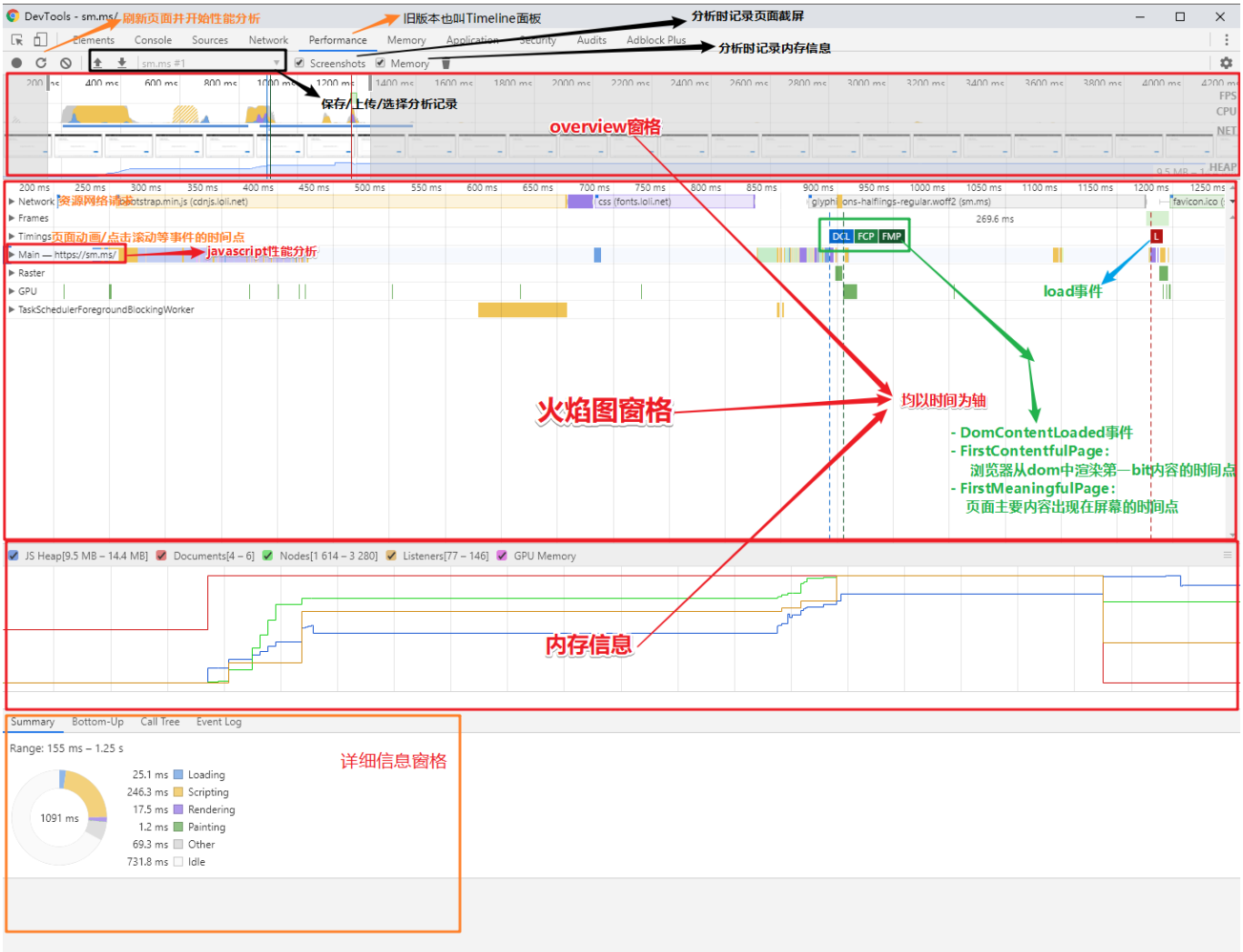
- DOMContentLoaded — 浏览器已经完全加载了 HTML，DOM 树已经构建完毕，但是像是 `<img>` 和样式表等外部资源可能并没有下载完毕。
- load — 浏览器已经加载了所有的资源（图像，样式表等）。
- beforeunload/unload -- 当用户离开页面的时候触发。
- 更多

data URLs

- 即前缀为 data: 协议的的 URL，其允许内容创建者向文档中嵌入小文件，例如浏览器 API canvas 支持的 base64 编码格式图片，更多相关

Performance 性能面板

- performance 面板可以用于分析运行时性能(运行时强调的是与页面加载性能相区分)
- 以隐身模式打开网页 （隐身模式可确保 Chrome 以干净的状态运行。例如，排除扩展对性能测量的影响
- Janky Animation demo ： 性能测试 demo
- 视图 overview：



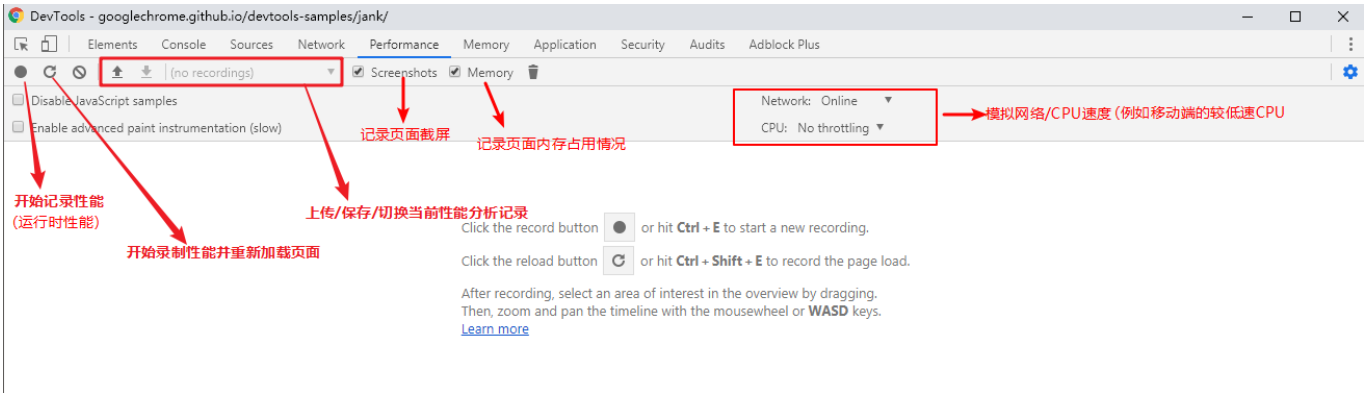
RAIL 模型

- RAIL 模型是一种性能模型，定义了四个维度的性能分析指标
- Response：在100 毫秒以内响应（例如从点按到绘制）
- Animation： 每秒生成 60 帧，每个帧的工作（从 JS 到绘制）完成时间小于 16 毫秒,达到人眼顺滑（例如滚动 拖动都是动画类型）（因为浏览器需要花费时间将新帧绘制到屏幕上，只有 10 毫秒来执行代码）
- Idle：利用空闲时间完成推迟的工作（要实现第一条 response 在 100ms 内响应，Main 主线程 JS 工作应该小于 50ms，剩余的时间将主线程的控制从 js 返回给浏览器执行其像素管道、对用户输入作出反应

等，因此最佳实践是将 js 的工作分成不大于 50 毫秒的块,如果用户开始交互，优先级最高的事项是响应用户。

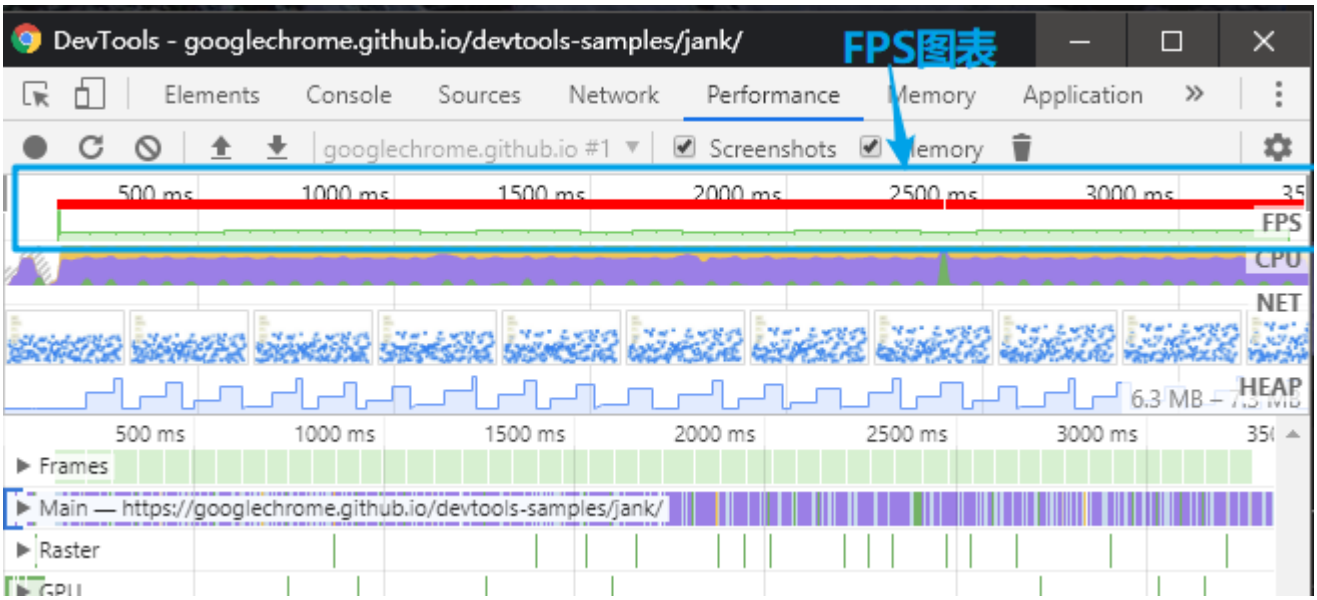
- `Load`：在 `1000 毫秒`以内呈现内容（无需完整加载，启用渐进式渲染，将非必需的加载推迟到空闲时间段
- 通过 performance 面板，可以得到这四个维度的分析数据

控制区



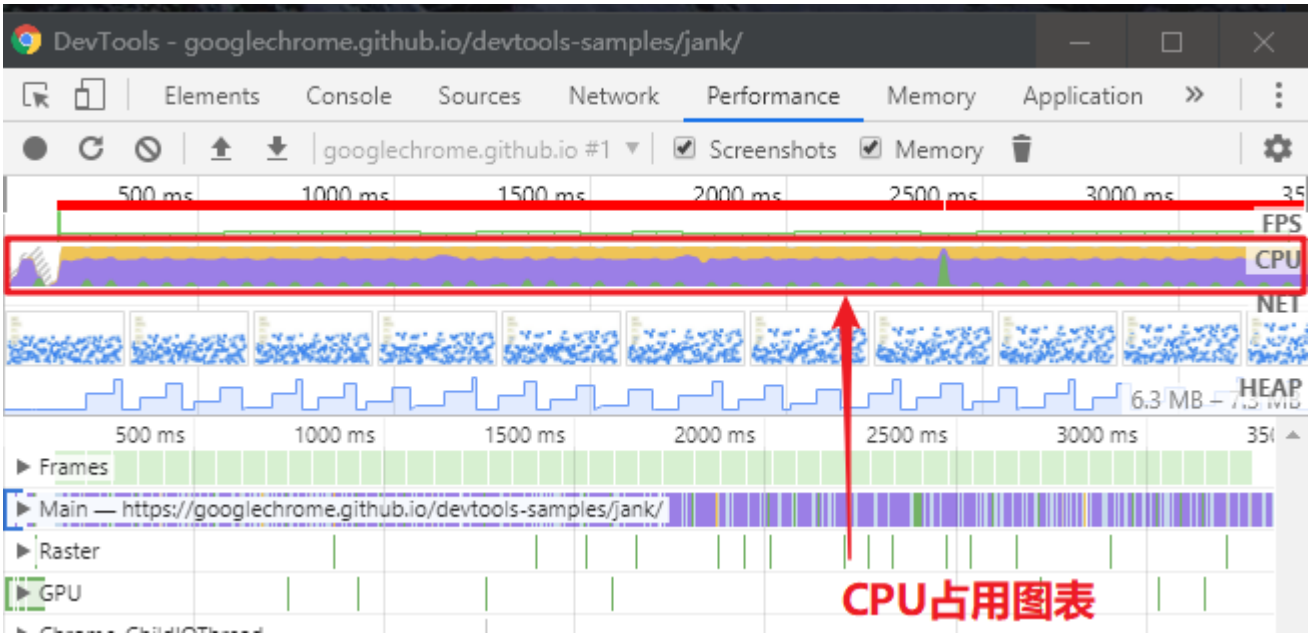
- 点击`录制按钮`或者`开始录制并刷新页面按钮`,可以在控制区下方得到全部性能分析结果
- 其中除了最下方的详细信息窗格以外，分析结果都是以时间为轴
- 可以在 overview 窗格拖动鼠标，选择某段时间的分析结果
- 滚动鼠标滚轮，缩放/移动选中事件
- 在火焰图窗格，按住`shift`，滚动鼠标滚轮，可以上下
- 在火焰图窗格，也可以直接左右拖动图表
- 或者用`W A S D`按键控制缩放移动
- `Disable JavaScript samples`默认情况，在`Main`主线程的火焰图中，会详细记录 js 函数之间的调用栈，可以开启此选项禁用调用栈记录
- `Enable advanced paint instrumentation`启用高级绘图工具，可以在分析结果的`Frames`中的每一帧的详细结果中看到`Layer`选项卡，其中有选中帧的详细图层信息；也可以在`Main`主线程火焰图中选中绿色的`Paint`事件，在最底部详细信息的`Paint Profile`选项卡中，看到详细的页面绘制过程分析
- `Collect garbage`控制器最右的垃圾桶图标，是强制执行垃圾回收，对于监控内存比较有用

FPS 图表 - Frames Per Seconds

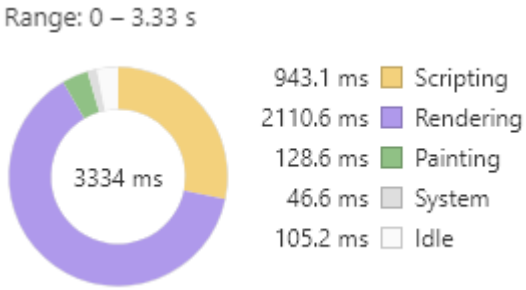


- FPS 图表中，绿色代表帧率高低，参考RAIL模型，帧率>=60 时，用户能体验的顺滑的网页
- 红色出现 代表有掉帧情况

CPU 图表



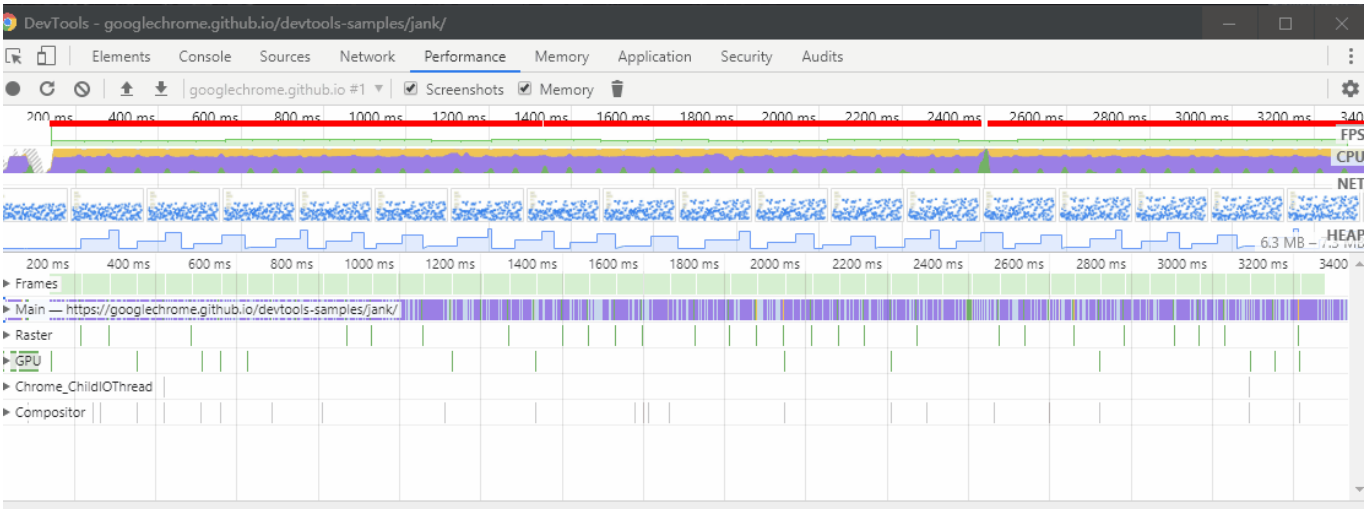
- CPU 图表中，不同的颜色代表不同事件对 CPU 的占用，颜色信息如图



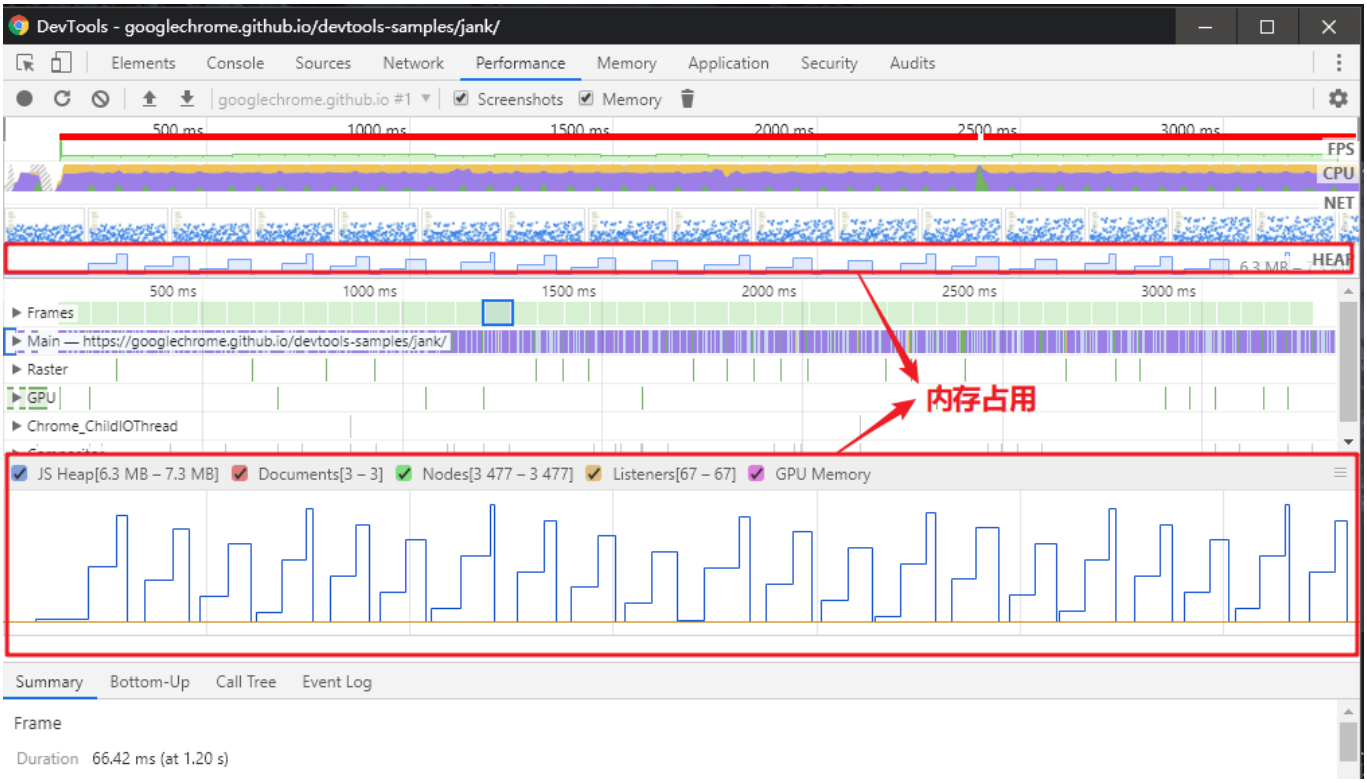
- 当 CPU 长时间被占满，就是当前网页性能需要优化的信号

SCREENSHOTS

- 鼠标在FPS,CPU,NET图表悬浮时，会展示出鼠标对应时间点的网页截屏，左右移动鼠标可以看到网页变化的重播效果



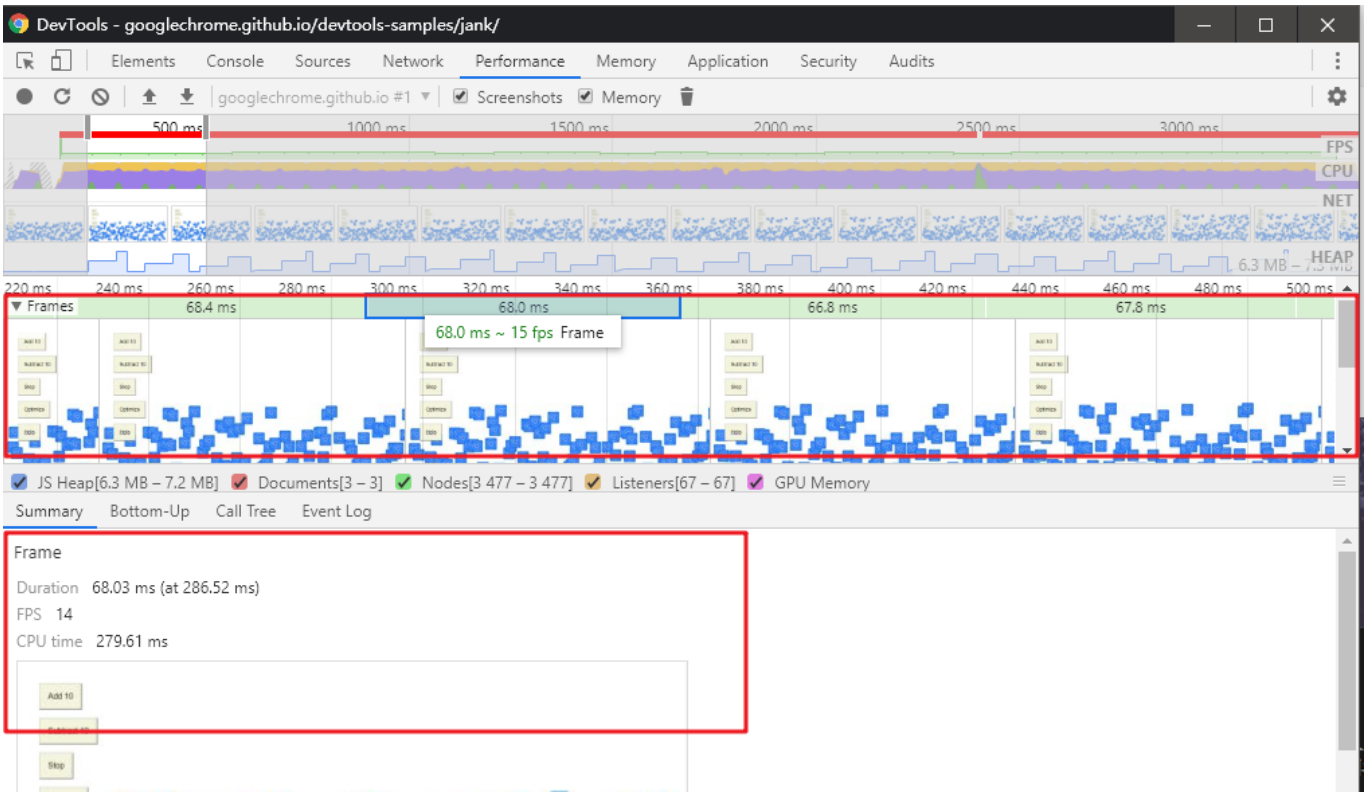
HEAP



- 在 HEAP 图表中可以看到 JS 内存占用情况，与下方的 memory 窗格中的JS Heap相对应
- 在 Memory 窗格还可以看到 Document 文档、Nodes DOM 节点、监听器、GPU 内存的习份内存统计

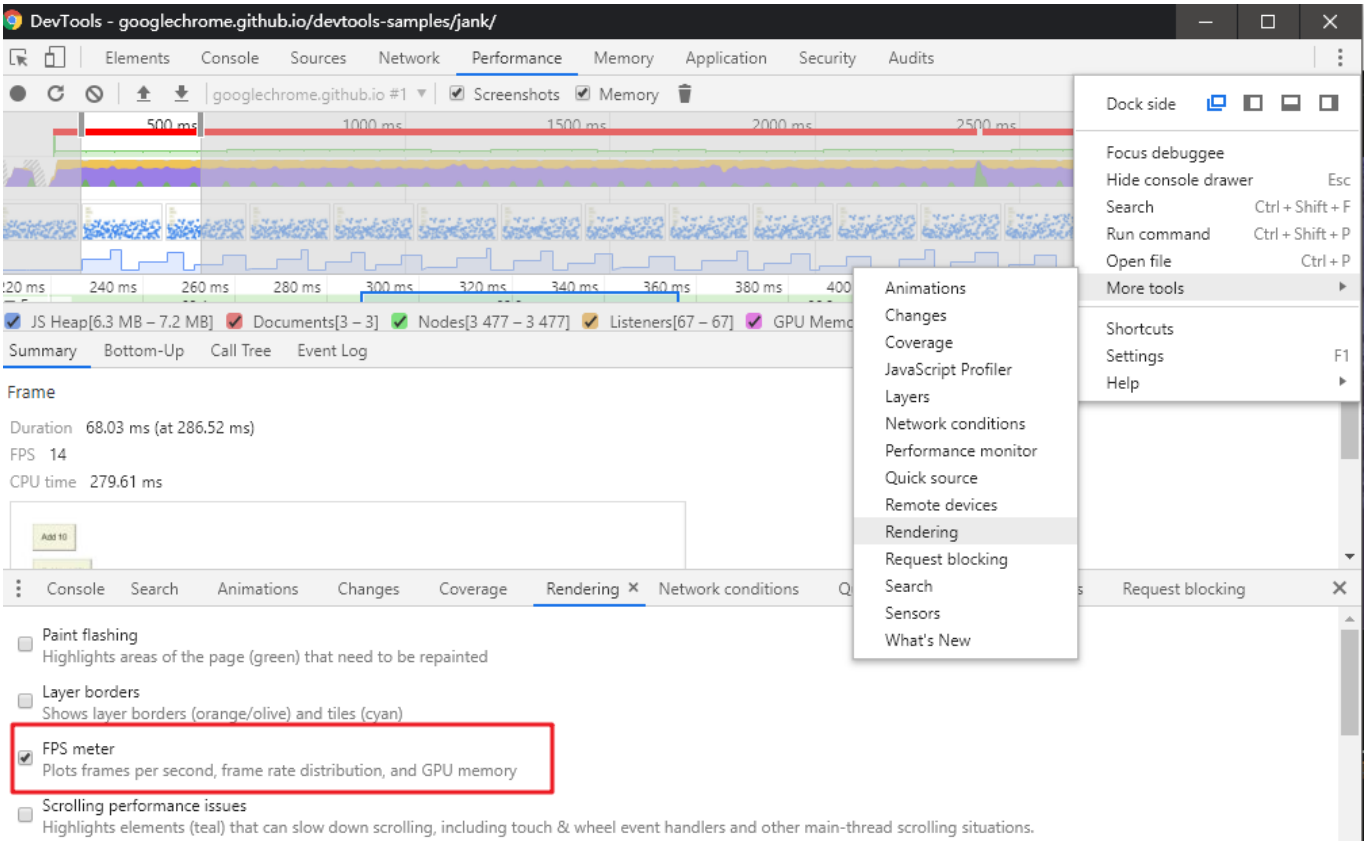
Frames

- 点击三角箭头展开Frames区域，鼠标悬浮/点击绿色方块，可以看到该特定帧的帧率和渲染耗时，当 FPS 低于 60，表明当前帧的渲染效率较低

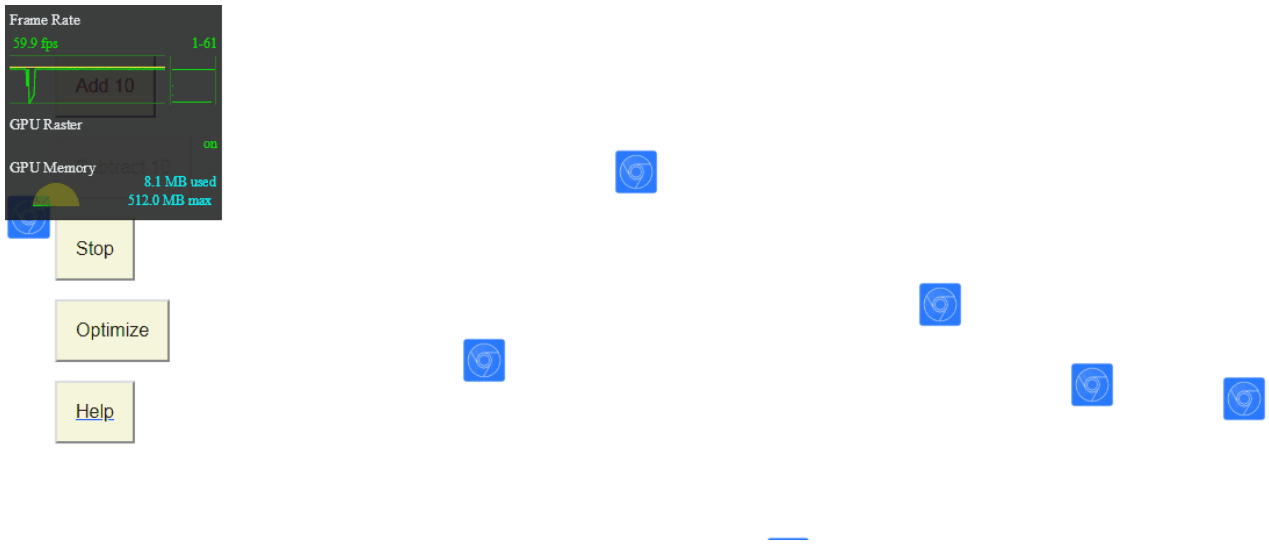


FPS 仪表工具

- 通过`more -> more tools -> Rendering` 或者 `ctrl+shift+p -> rendering` 打开`Rendering`面板

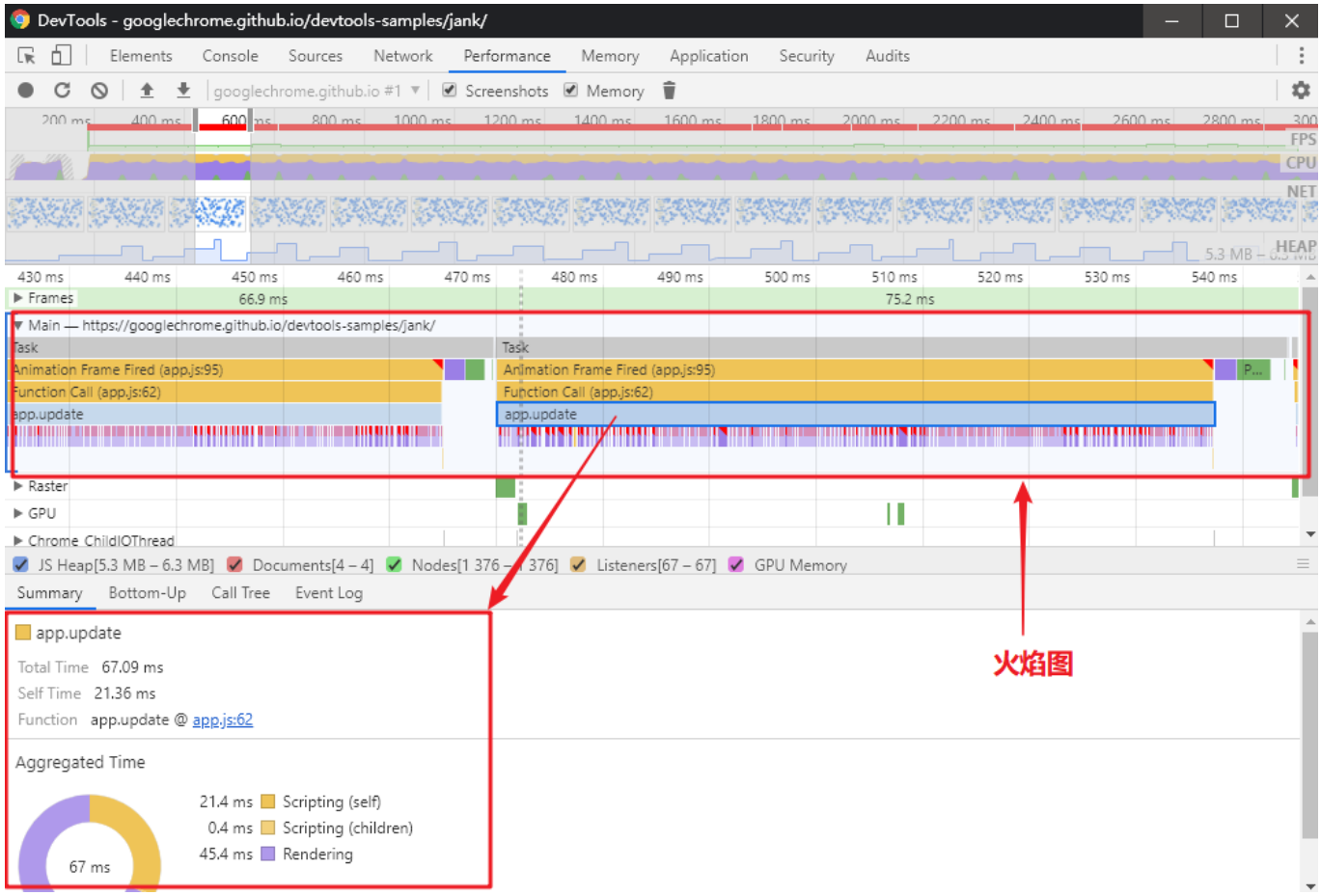


- 启用`FPS meter`，即可看到的页面实时帧率

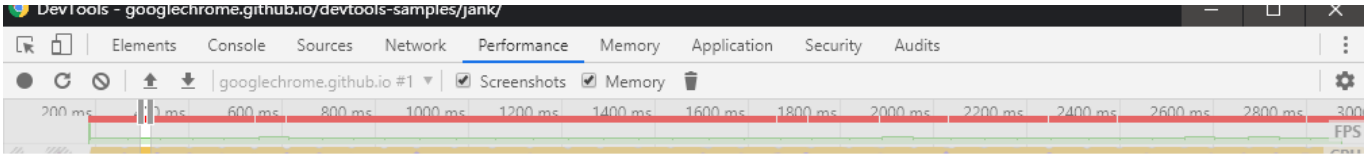


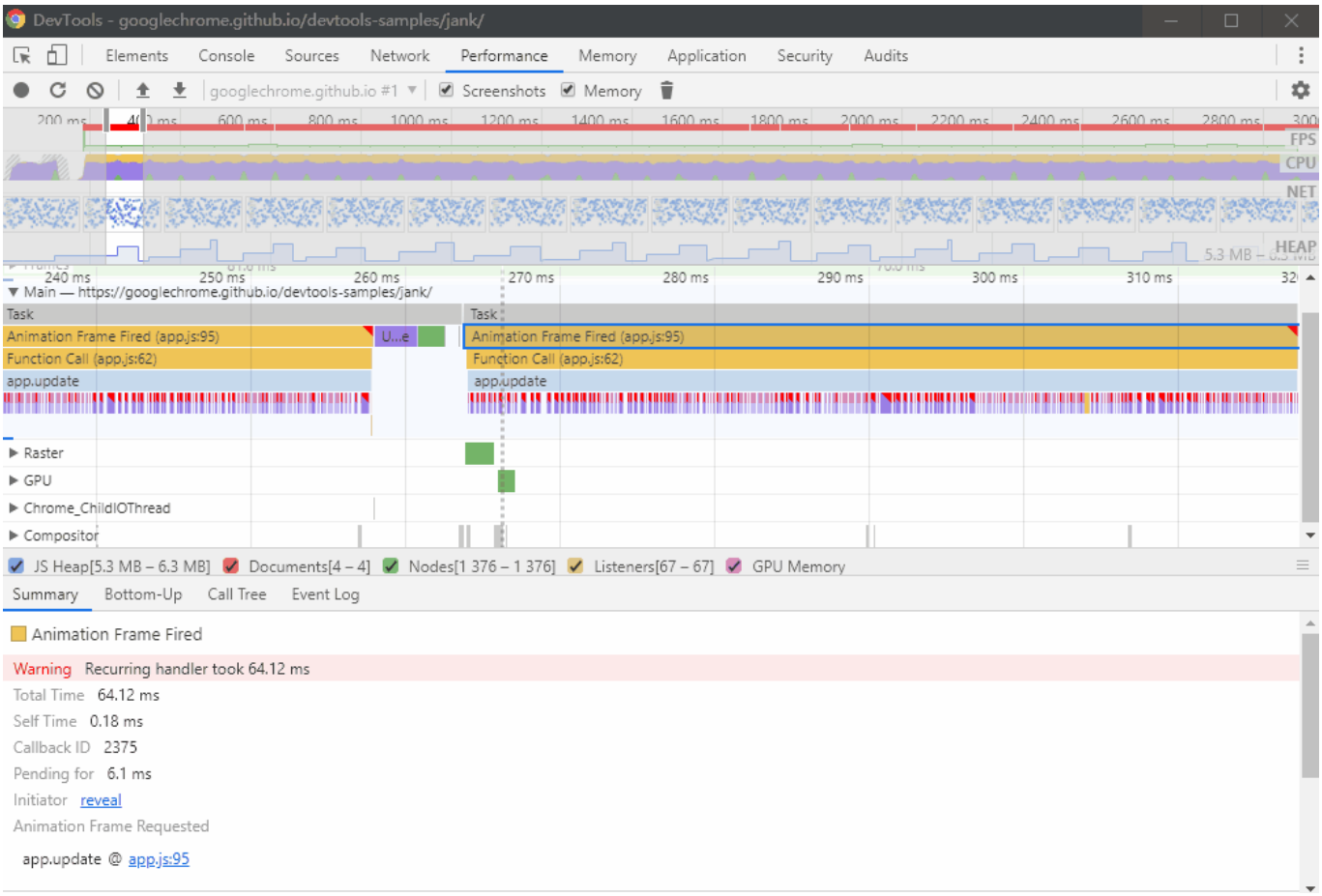
Mian

- 点击三角箭头展开Main区域，可以看到主线程上事件的火焰图
- x 轴是时间，每一块代表一个事件，y 轴代表堆栈，事件的上下堆叠，代表上层事件引发/调用了下层事件

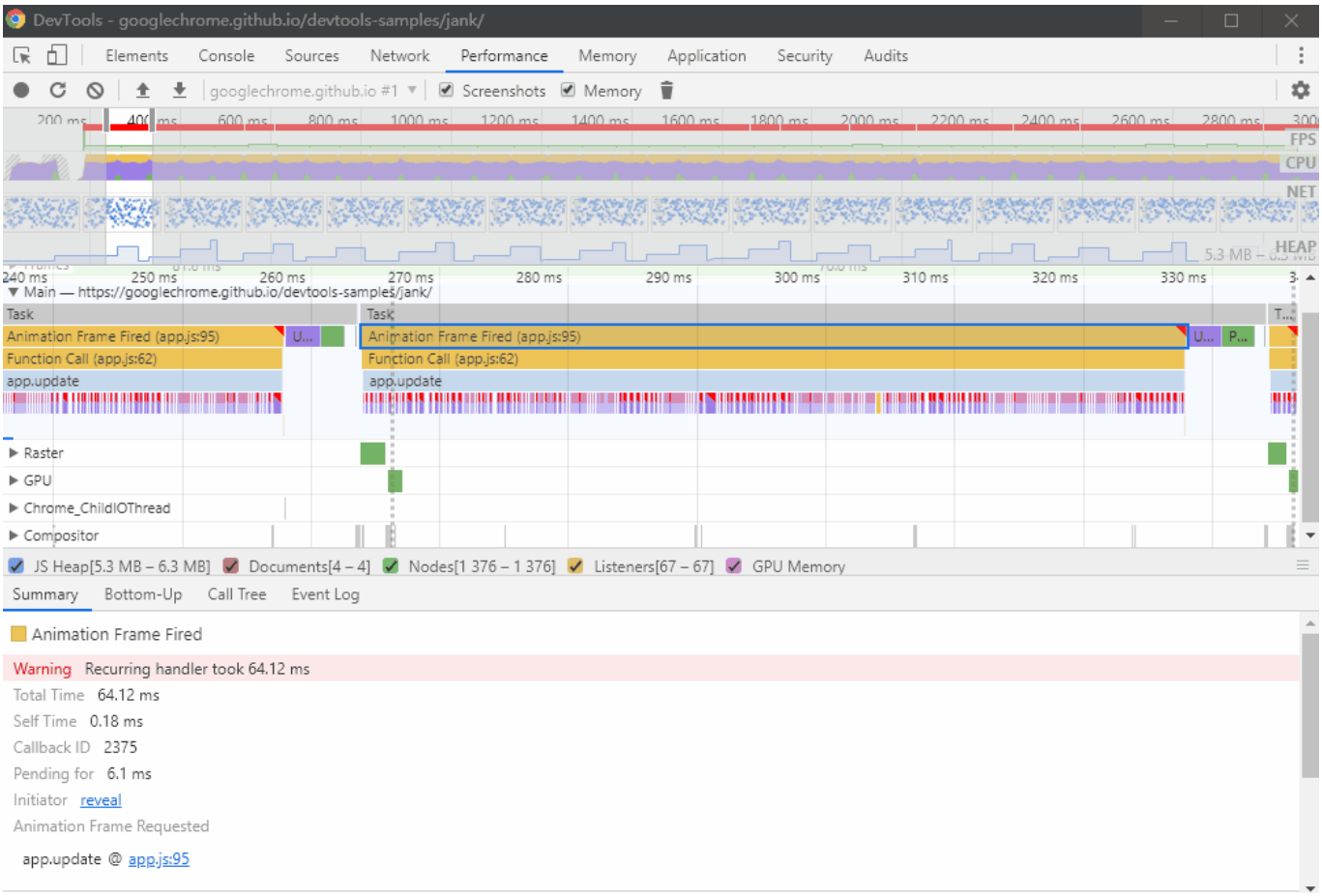


- 通过调用堆栈，可以找出导致低性能的事件及其源码位置
- 当事件块出现红色三角，可以点击三角查看该事件的性能相关警告信息，并定位到引起警告的代码





- 点击`Animation Frame Fired`事件，可以在最下方`Summary`窗格查看触发动画事件的详细信息，点击`Initiator`后的`reveal`链接，会高亮到引起动画事件的事件



性能相关扩展

- 网页性能-性能模型/加载/渲染/审计/优化
- the-anatomy-of-a-frame - 一个帧的剖析
- 常见的时间线事件参考

Memory 内存面板

内存 && 内存泄露

内存占用：

1. allocate 分配内存(eg 声明变量
2. 使用内存
3. release 释放内存

内存泄漏：

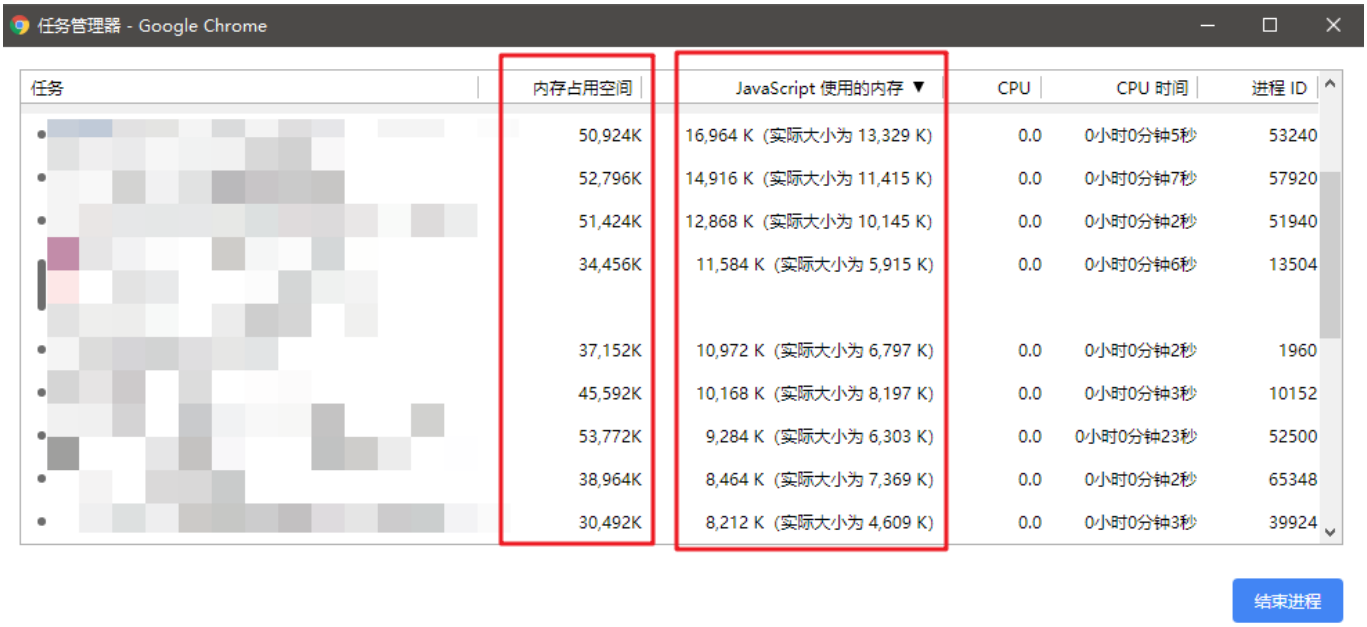
- 内存泄露-Memory Leak：内存被占用后无法被 release，且无法被垃圾回收器回收
- 内存泄漏会引起性能问题，且时间越久越严重，因为被占用且无法回收的内存只会增加不会减少
- 垃圾回收-Garbage Collect-GC：浏览器收回内存。 浏览器决定何时进行垃圾回收。 回收期间，所有脚本执行都将暂停。因此，如果浏览器经常进行垃圾回收，脚本执行就会被频繁暂停

造成内存泄露常见原因

- fogotten timer被遗忘的计时器：例如调用 setInterval()方法一定要加结束条件
- Dettached HTMLElement分离的 dom 节点：在 dom 被移除后，dom 变量仍然存在

内存监控 1-Task manager 任务管理器

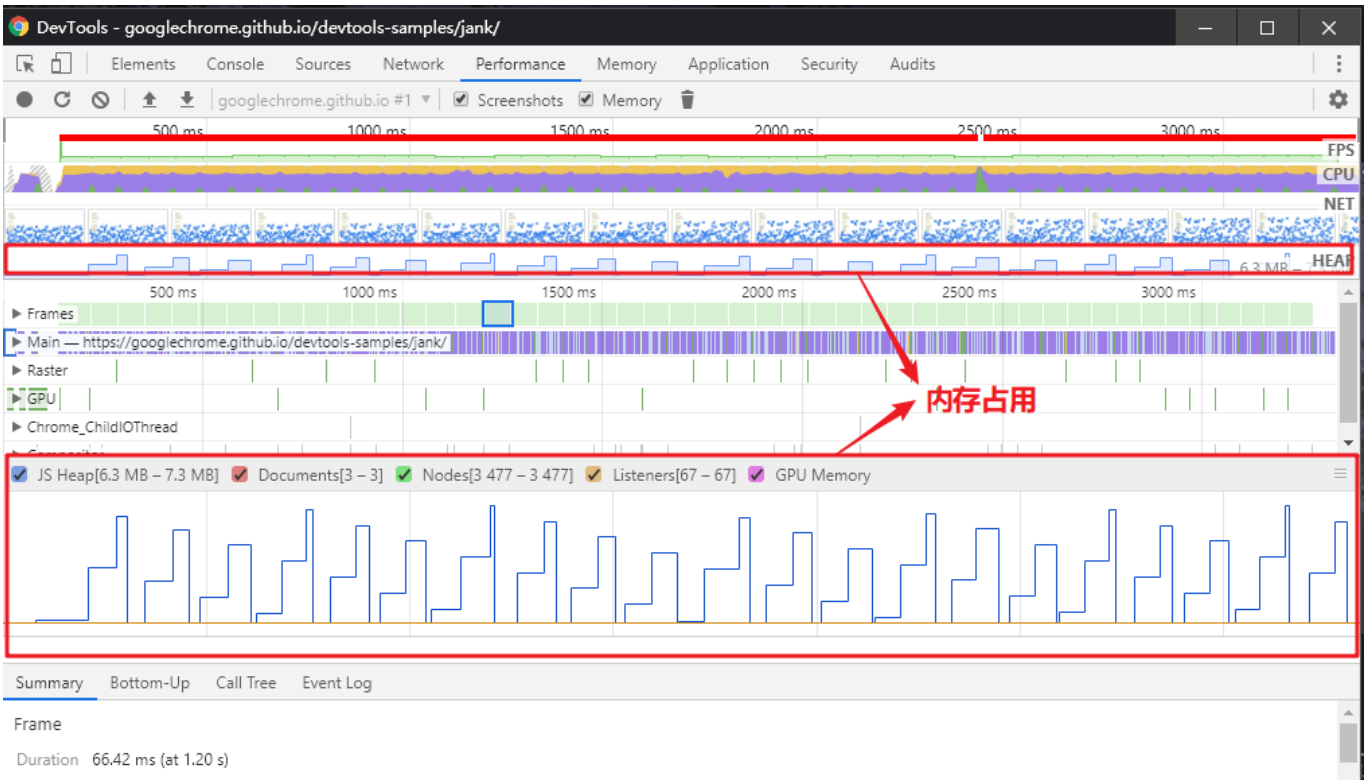
- chorme 浏览器 -> task manager 任务管理器工具中，可以监控每个 tab 页的 js 内存占用大小



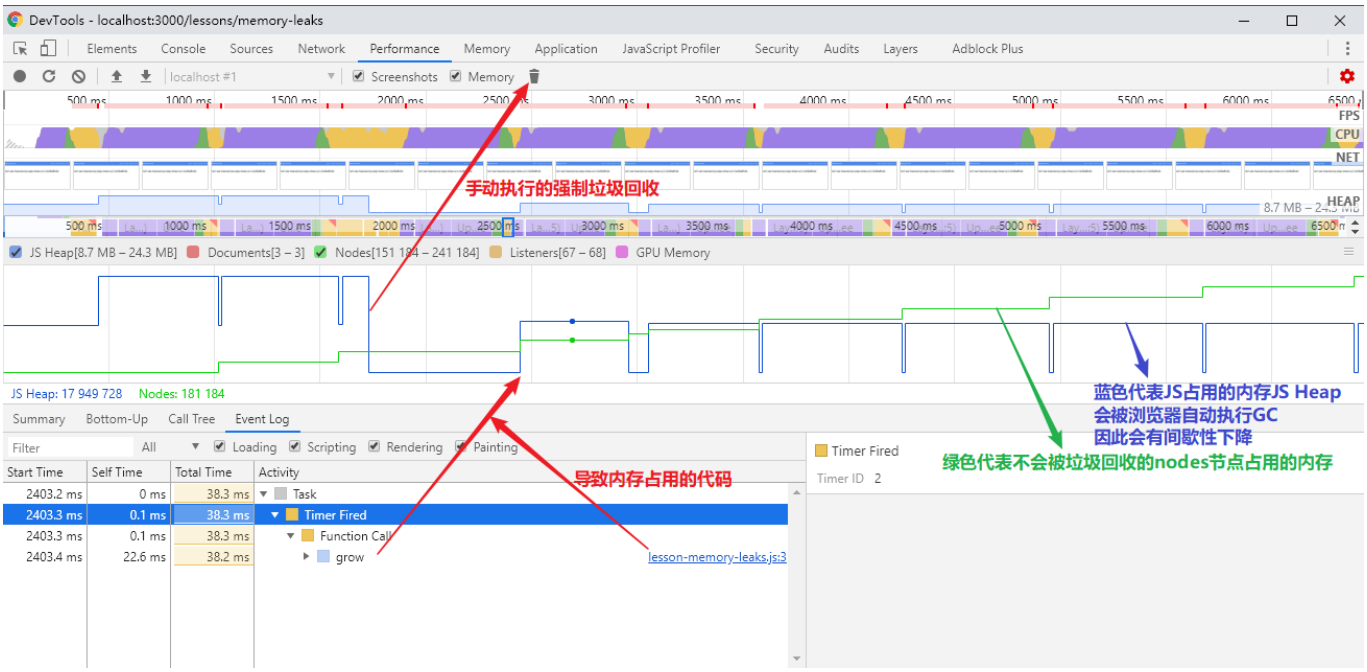
- `Memory` 列表示原生内存。DOM 节点存储在原生内存中。 如果此值正在增大，则说明正在创建 DOM 节点。
- `JavaScript Memory`列表示 JS 堆。此列包含两个值。 实际大小表示页面上的对象正在使用的内存量。如果此数字在增大，要么是正在创建新对象，要么是现有对象正在增长。

内存监控 2-Devtools Performance 面板

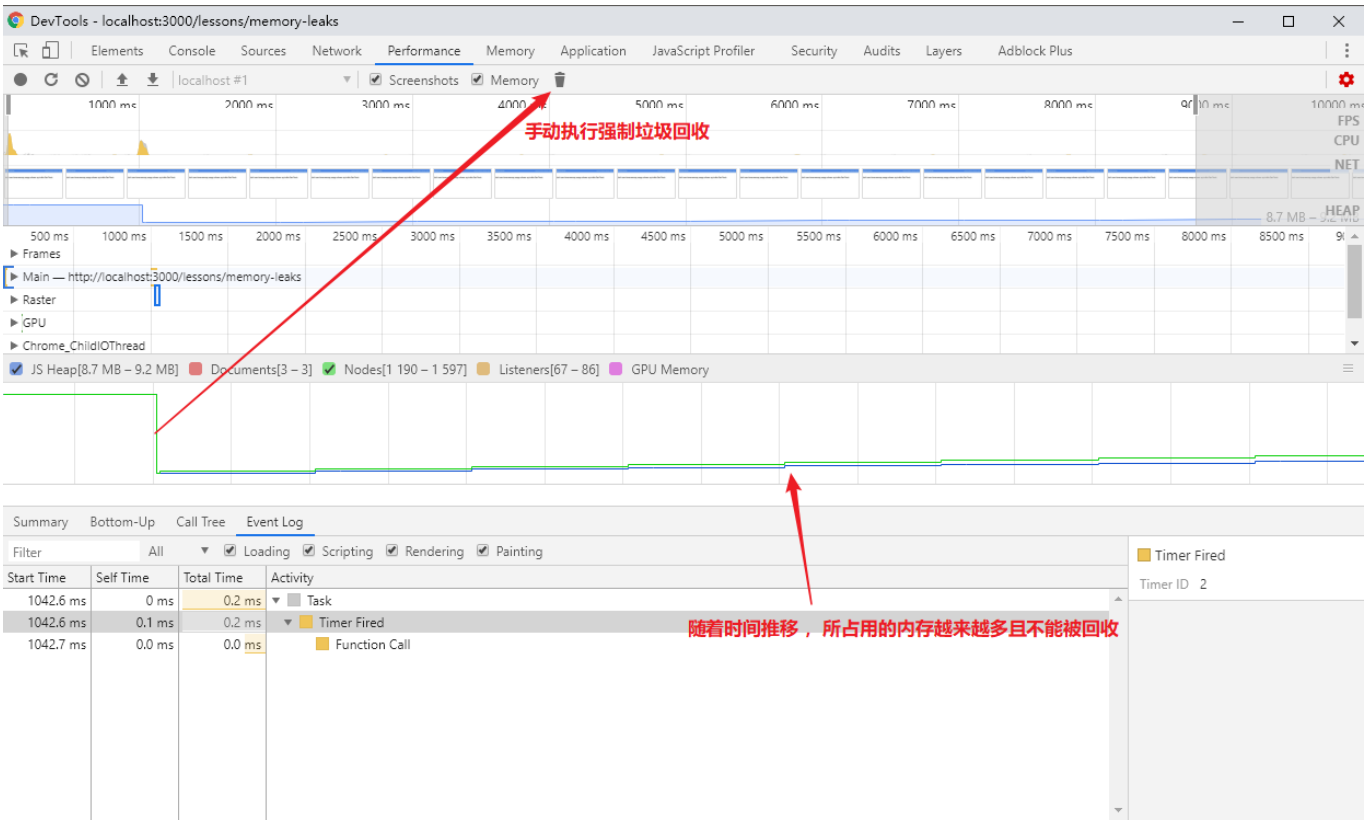
- 在`Performance`面板记录性能时，勾选`memory`即可在分析结果中看到 memory 占用情况



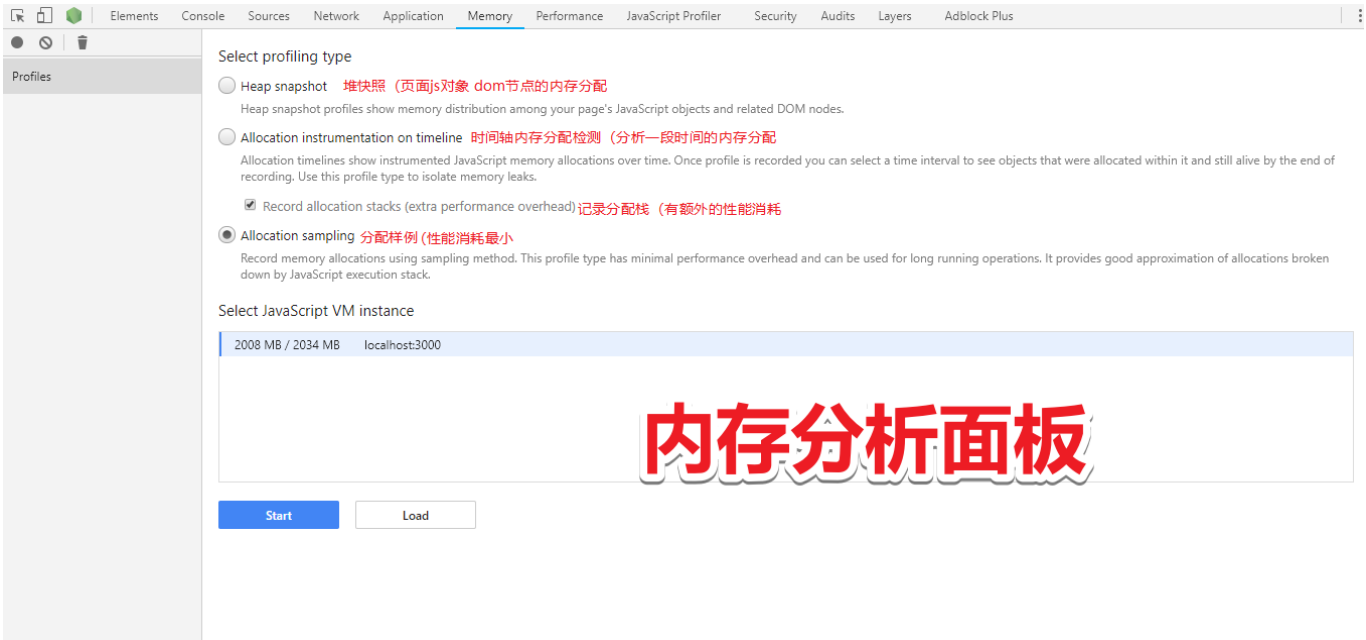
```
//示例1:正常的内存占用与GC

var x = [];

function grow() {
    for (var i = 0; i < 10000; i++) {
        document.body.appendChild(document.createElement("div"));
    }
    x.push(new Array(1000000).join("x"));
}

setInterval(grow, 100);
```




```
//示例2:不可被GC的内存泄漏

function grow() {
    // for (var i = 0; i < 10000; i++) {
    //     document.body.appendChild(document.createElement("div"));
    // }
    // x.push(new Array(1000000).join("x"));
    var ul = document.createElement("ul");
    for (var i = 0; i < 10; i++) {
        var li = document.createElement("li");
        ul.appendChild(li);
    }
    detachedTree = ul;
}

setInterval(grow, 1000);
```



内存监控 3-Devtools Memory 面板

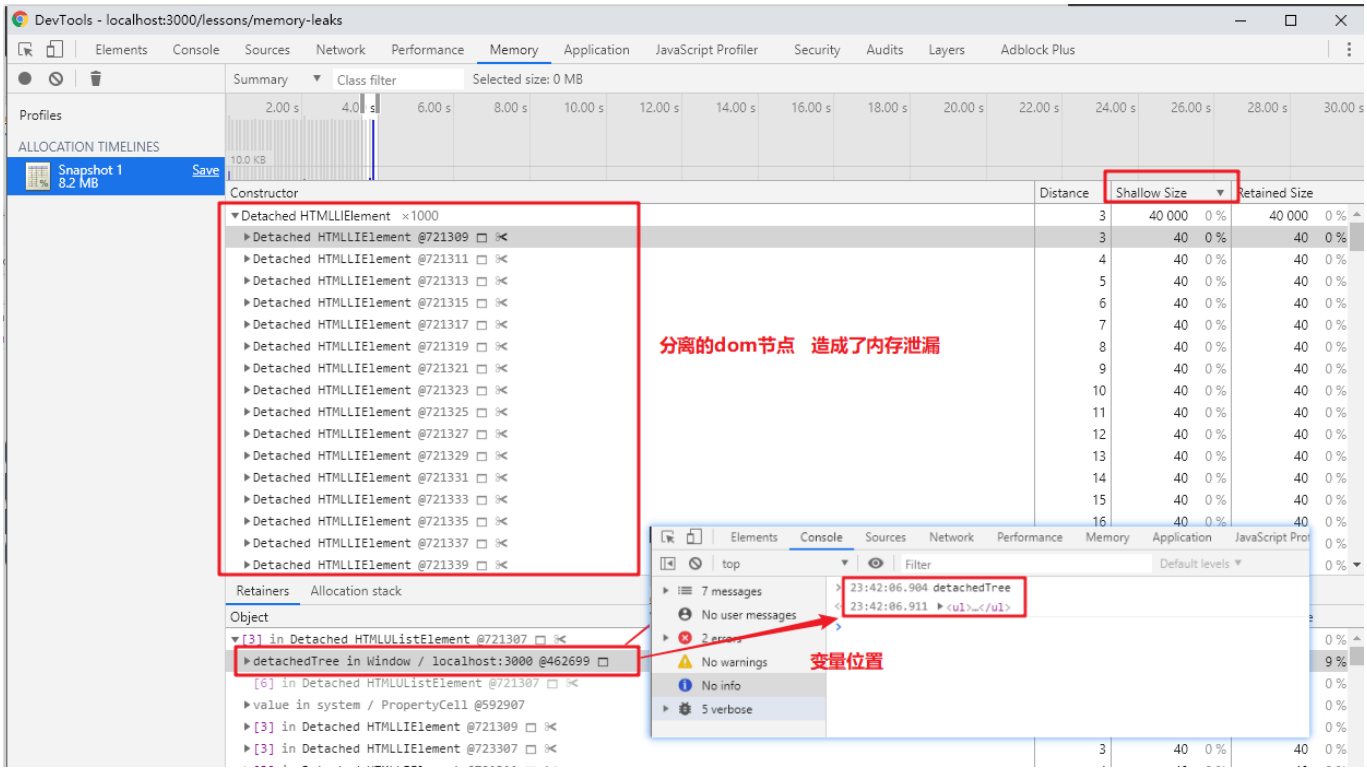


- 如上图所示，在右侧三种内存分析模式选择一种后，即可点击左上角`record`开始记录内存

1. `Heap snapshot`堆快照，记录当前时间点内存中页面 js 对象和 dom 节点的分配情况
2. `Allocation instrumentation on timeline`按时间轴记录内存，可以选记录内存分配调用栈(可以帮助定位到具体分配内存的源码)
3. `Allocation sampling`使用抽样方法记录内存分配。具有最小的性能开销，可用于长时间运行的操作。提供了由 JavaScript 执行堆栈细分的良好近似分配。

- 左上角的垃圾桶图标`Collect garbage`是强制执行一次垃圾回收，内存监控的最佳实践是在监控内存前执行一次强制垃圾回收

- 利用上述示例 2 代码，执行时间线 Memory 分析：



扩展

- 内存相关术语
- 深入内存分析