

debug

[手动调试](#)

[工具调试](#)

[chrome调试](#)

[文件同步](#)

[断点调试](#)

[more tools](#)

[vscode断点调试](#)

[nodejs调试](#)

[npm scripts调试](#)

[非node命令调试](#)

[总结](#)

代码的编写工作其实只占开发者时间的一小部分，为了更高效地开发，我们必须了解和掌握调试技巧、如何进行debug。花一点时间学习各种调试技巧，能帮助我们更好更快地完成工作。我总结了一些常用的调试方法和调试经验，接下来分享一些核心概念，并辅助具体的例子进行演示。

手动调试

打日志应该是程序员最常用、最烂熟于心的调试方式。原理是直接在控制台打印出结果，用来检查数据是否正确。

有以下几种不同类型的`console`形式：

1. `console.log`

`console.log`无疑是程序员使用频率最高的`console`形式。可以直接使用`console.log(a)`即可完成变量a的打印，下面主要介绍一下其他非常便利的技巧：

对json对象进行格式化：

使用`console.log(JSON.stringify(obj, null, 2))`可以结构化打印出`obj`对象，方便更好地观察json数据结构，由于是字符串，所以不用担心引用产生的问题。

```
> console.log(JSON.stringify(obj,null,2))
{
  "componentName": "FormItem",
  "props": {
    "name": "attachType",
    "label": "模式：",
    "style": {
      "fontSize": 12,
      "color": "#0364ff",
      "lineHeight": 18,
      "float": "none"
    },
    "asterisk": true,
    "rules": [
      {
        "required": true,
        "message": "请选择模式"
      }
    ]
  }
}
```

使用{}包裹变量：

在使用`console.log()`的时候，可以使用大括号({})将变量包围起来，这样的优点是不仅会把变量的值打印，同时还会将变量名打印出来。如将`console.log(id,name,content)`改为

```
console.log({id,name,content})。
```

```
> console.log(id,name,content)
1 "张三" "我就是张三啊,一起来要!"                               fundebug.2.0.0.min.js:1
< undefined
> console.log({id,name,content})
                                                fundebug.2.0.0.min.js:1
▶ {id: 1, name: "张三", content: "我就是张三啊,一起来要!"}
< undefined
```

给`console.log`添加颜色：

当遇到一个具有大量log输出的大型应用时，除了使用过滤器（filter）以外，我们还可以使用颜色来加以区分。在一些重要的地方输入带样式的log，可以在控制台快速地定位到它，不至于被大量的log信息淹没。

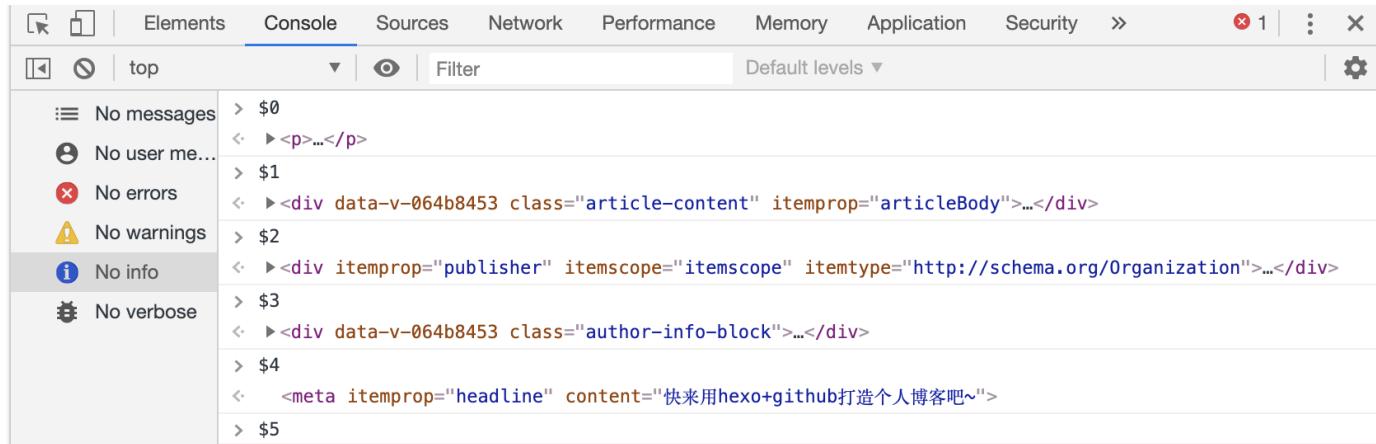
```
> console.log('%c我是绿色文字', 'color: green');
我是绿色文字                                         fundebug.2.0.0.min.js:1
< undefined
> console.log('%c我是绿色文字,%c我是红色文字,%c我是蓝色文字', 'color:
green','color:red','color:blue');
我是绿色文字,我是红色文字,我是蓝色文字           fundebug.2.0.0.min.js:1
< undefined
```

具体的格式如下：

- 单色：`console.log ('%c+内容', style样式)`
- 多色：`console.log ('%c+内容1, %c+内容2, %c+内容3', style样式1, style样式2, style样式3)`

快速获取引用对象

`$0、$1、$2、$3、$4` 指令相当于在Elements面板最近选择过的五个引用。例如在某网页对应的Elements面板中，我随机点击了5个DOM节点，则`$4`是我第一个点击的，而`$0`是我第五个点击的（最后一个点击的），使用此方法可以快速在console面板中调试选中的DOM节点。



在console面板中，还存在以下这些非常好用的方法，可以在调试的过程中直接使用：

方法名	作用
<code>\$_</code>	返回上一个被执行过的值
<code>\$</code>	<code>document.querySelector()</code>
<code>?</code>	<code>document.querySelectorAll()</code>
<code>keys</code>	<code>Object.keys</code>
<code>values</code>	<code>Object.values</code>
<code>copy</code>	拷贝一个对象为字符串表示方式到剪切板
<code>getEventListeners</code>	获取注册到一个对象上的所有事件监听器

2. console.dir

使用 `console.dir` 命令，可以打印出对象的结构，而 `console.log` 仅能打印返回值，在打印 `document` 属

性时尤为有用。

```
> console.dir(document.getElementsByTagName('div'))  
VM2163:1  
▼ HTMLCollection(65) 1  
► 0: div#container  
► 1: div#wrap  
► 2: div.nav_container  
► 3: div.fundebbug_nav_dropdown  
► 4: div.fundebbug_nav_dropdown_menu  
► 5: div#toggle.menuButton_mobile  
► 6: div#mobile_bg_toggle.mobile_bg  
► 7: div.outer  
► 8: div.article-meta  
► 9: div.article-inner  
► 10: div.article-entry
```

3. console.table

以表格形式打印，对于对象数组尤为合适。如果要打印的变量是一个数组，每一个元素都是一个对象，那么建议使用console.table来进行打印，表格化的形式会变得更加美观易读。

```
< ► (8) [{...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}]  
> console.table(arr)
```

VM2808:1

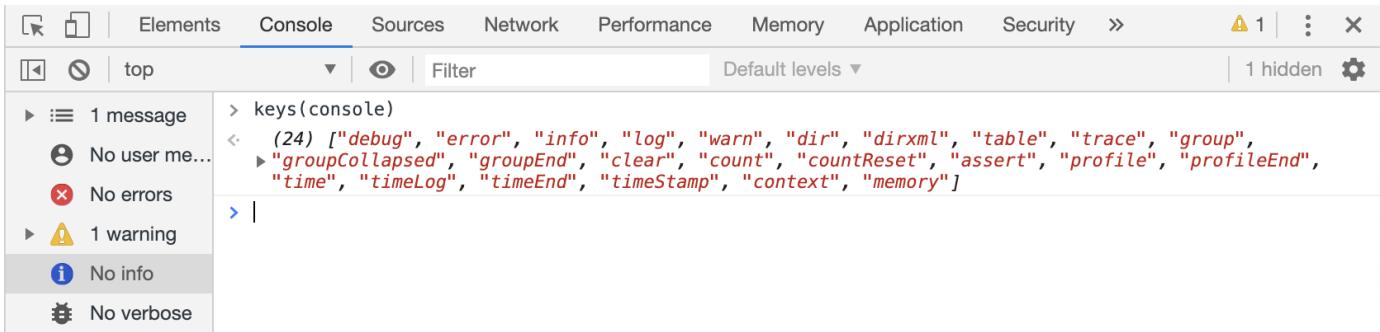
(index)	title	dataIndex
0	"id"	"id"
1	"类目名称"	"name"
2	"商品维度"	"dimension"
3	"作用对象"	"object"
4	"数量"	"num"
5	"开始时间"	"startTime"
6	"结束时间"	"endTime"
7	"操作人"	"operator"

4. console.info、console.debug、console.warn、console.error

console.info、console.debug 的作用与 console.log 相同，都是打印信息。

console.warn 是打印警告，console.error 是打印错误。

当然，`console` 对象远不止以上这些方法，我们可以在实际工作中，根据不同的场景去使用合适的方法。



工具调试

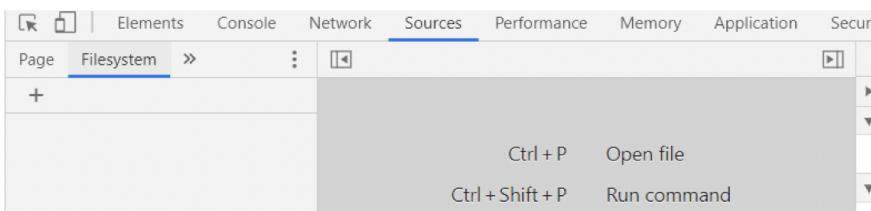
调试的第一步就是打断点。断点的目的是，代码运行到你想要开始调试的地方就停下来。这个时候就可以查看当前上下文信息，比如全局变量、局部变量的值，函数的输入是否正确，请求的返回值是否正常等。可以判断问题具体发生的地方，从而能更好地解决。

chrome调试

文件同步

在开发过程中，我们一般会直接打开chrome浏览器开发者工具调试，可能会在调试面板中修改css、js等，直到页面呈现我们想要的效果。这时，编辑源文件并同步到本地文件夹中是很重要的操作，在开发中能够提供很大的便利。我们可以在 `Sources` 面板下找到 `Filesystem` 选项，点击 `Add folder to workspace`，添加本地文件夹，接下来在 `Element` 中修改css样式、`Sources` 中修改js代码的同时，本地文件会同步更新，这样我们就不需要不断粘贴代码到编辑器了。

如果站点已经发布，需要对在线的资源做调整并更新到本地的文件系统，则同样找到 `Filesystem` 选项，点击 `Add folder to workspace` 添加本地文件夹，对相应文件做 `map` 操作，调整后即可同步到本地文件。

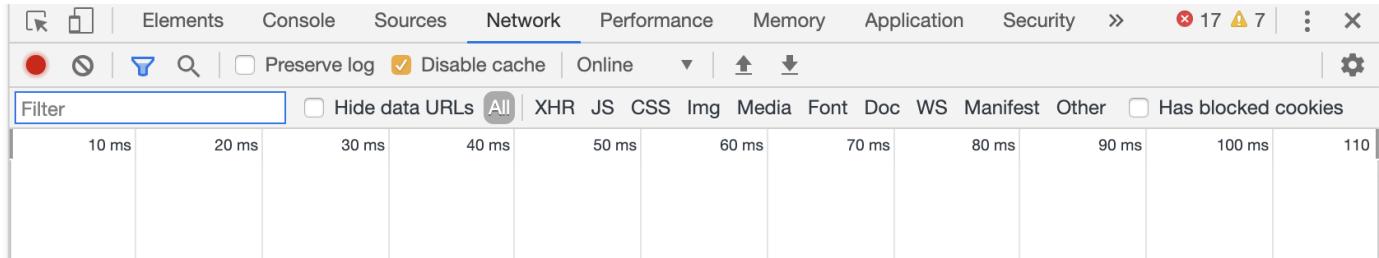


断点调试

chrome调试面板的三种打开方式：

1. Ctrl+Shift+I/F12 (Windows) 或 Cmd+Opt+I (Mac)
2. 页面右键选择“检查”
3. Chrome菜单中选择“更多工具 > 开发者工具”

Chrome调试面板一共分为八个功能块：



每一个图标点击后都会打开对应的调试面板，能获取各种不同的信息，如 DOM 树、资源占用情况、页面相关的脚本等。每个模块及其主要功能为：

Element 标签页：用于查看和编辑当前页面中的 HTML 和 CSS 元素。

Network 标签页：用于查看 HTTP 请求的详细信息，如请求头、响应头及返回内容等。

Source 标签页：用于查看和调试当前页面所加载的脚本的源文件。

TimeLine 标签页：用于查看脚本的执行时间、页面元素渲染时间等信息。

Profiles 标签页：用于查看 CPU 执行时间与内存占用等信息。

Resource 标签页：用于查看当前页面所请求的资源文件，如 HTML, CSS 样式文件等。

Audits 标签页：分析页面加载的过程，进而提供减少页面加载时间、提升响应速度的方案，用于优化前端页面，加速网页加载速度等。

Console 标签页：用于显示脚本中所输出的调试信息，或运行测试脚本等。

这里讲解一下常见的断点调试方法：

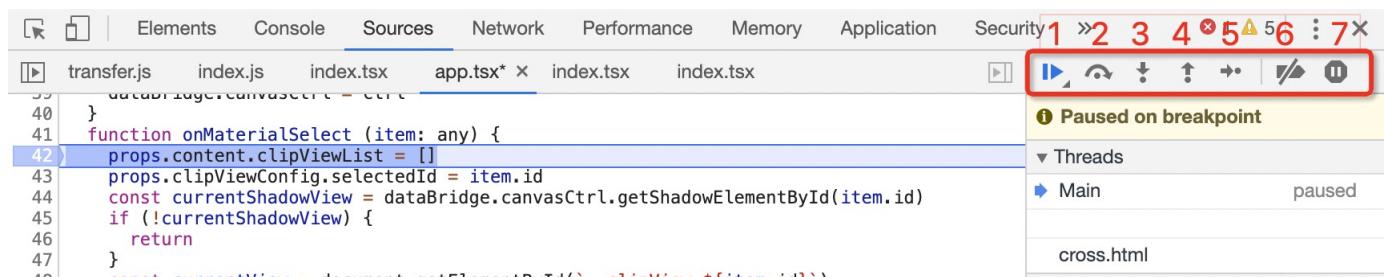
代码行断点

我们可以在浏览器开发工具中找到source面板，通过左侧的目录定位到需要调试的源码文件，找到对应源码在js代码，在左侧的数字上加上断点即可。可以通过 `ctrl+o` 打开搜索框输入文件名称进行搜索，`ctrl+shift+o` 快速跳转至指定函数。之后代码运行到断点处时，就会进入到调试状态。

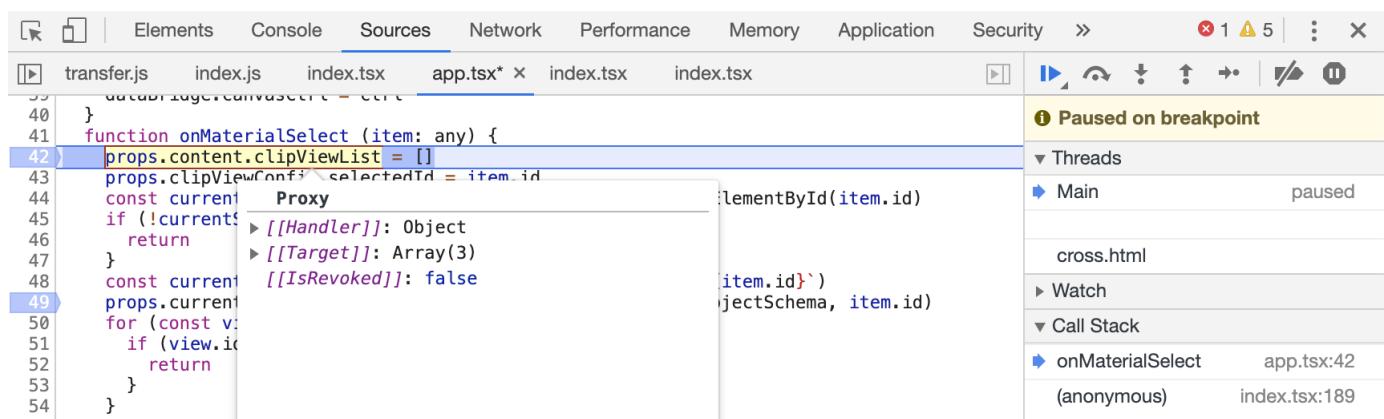
除了点击数字添加断点，我们还可以直接在需要调试的地方加上 `debugger` 关键字，这个操作相当于以上的代码行断点只是不需要的DevTools界面中设置，代码运行到此处时同样会进入到调试状态。

在右侧会出现一系列的调试按钮，从左向右依次为：

1. Pause/Resume script execution: 暂停/恢复脚本执行 (程序执行到下一个断点停止)
2. Step over next function call: 执行到下一步的函数调用 (执行但不进入)
3. Step into next function call: 进入当前函数 (一步步执行每行代码)
4. Step out of current function: 跳出当前执行函数
5. step: 与Step into next function call一样
6. Deactive/Active all breakpoints: 关闭/开启所有断点
7. 在Pause on exceptions: 异常情况自动断点设置



在参数区可以添加想要监听的参数或表达式的变化，也可以直接在文件里通过鼠标悬停、选中表达式的方式查看它们的变化。



条件代码断点: 如果知道需要调查的确切代码区域，但只想在其他一些条件成立时进行暂停，则可使用条件代码行断点。同样找到source面板，通过左侧的目录定位到需要调试的源码文件，找到对应源码，右键点击左侧的行数，选择 Add conditional breakpoint，输入需要满足的条件表达式即可。当代码运行到这里满足此条件时，才会进入到调试状态。

```

72 const addProject = (): void => {
73   setVisible(true)
    Line 73: Conditional breakpoint ▾
    Expression to check before pausing, e.g. x > 5
  
```

DOM 更改断点

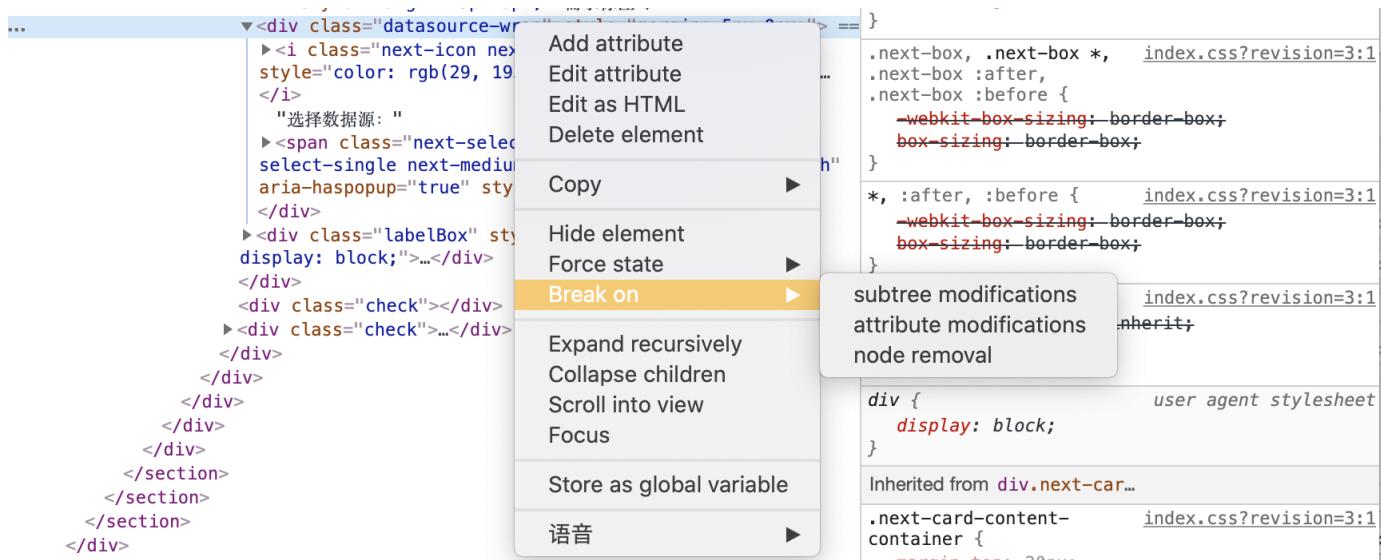
监听dom节点或其子节点变化时用到的断点，具体步骤如下：

1. 打开Elements面板

2. 找到需要打断点的element位置

3. 右键点击选中的element，在菜单栏选择Break on，有3种属性可供选择——Subtree

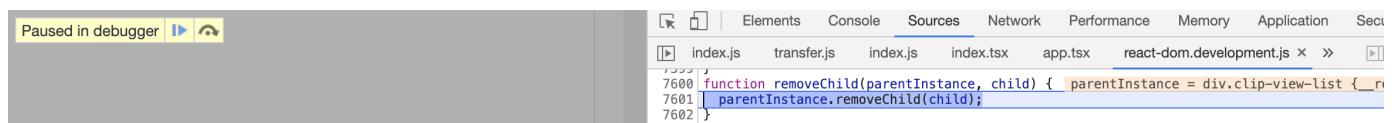
modifications、Attributes modifications、Node Removal，分别是在DOM子节点结构改变、DOM属性改变、DOM节点移除时触发断点，右侧面板直接调至对应的js处，可以迅速定位代码位置进行检查和修改



1) Subtree modifications

在当前节点添加、删除、改变子节点时触发，在dom元素右键，选择（Break on subtree modifications），可以在此dom被修改时触发断点，在不确定dom被哪段js脚本修改时可能有用。

在打断点的DOM区域发生变化时，会自动中断，调至指定js代码：



2) Attributes modifications

在当前选定的节点上添加或删除属性时或属性值更改时触发，与Subtree modifications相似，通过js改变当前节点属性时，断点会跳转到对应的js代码处。

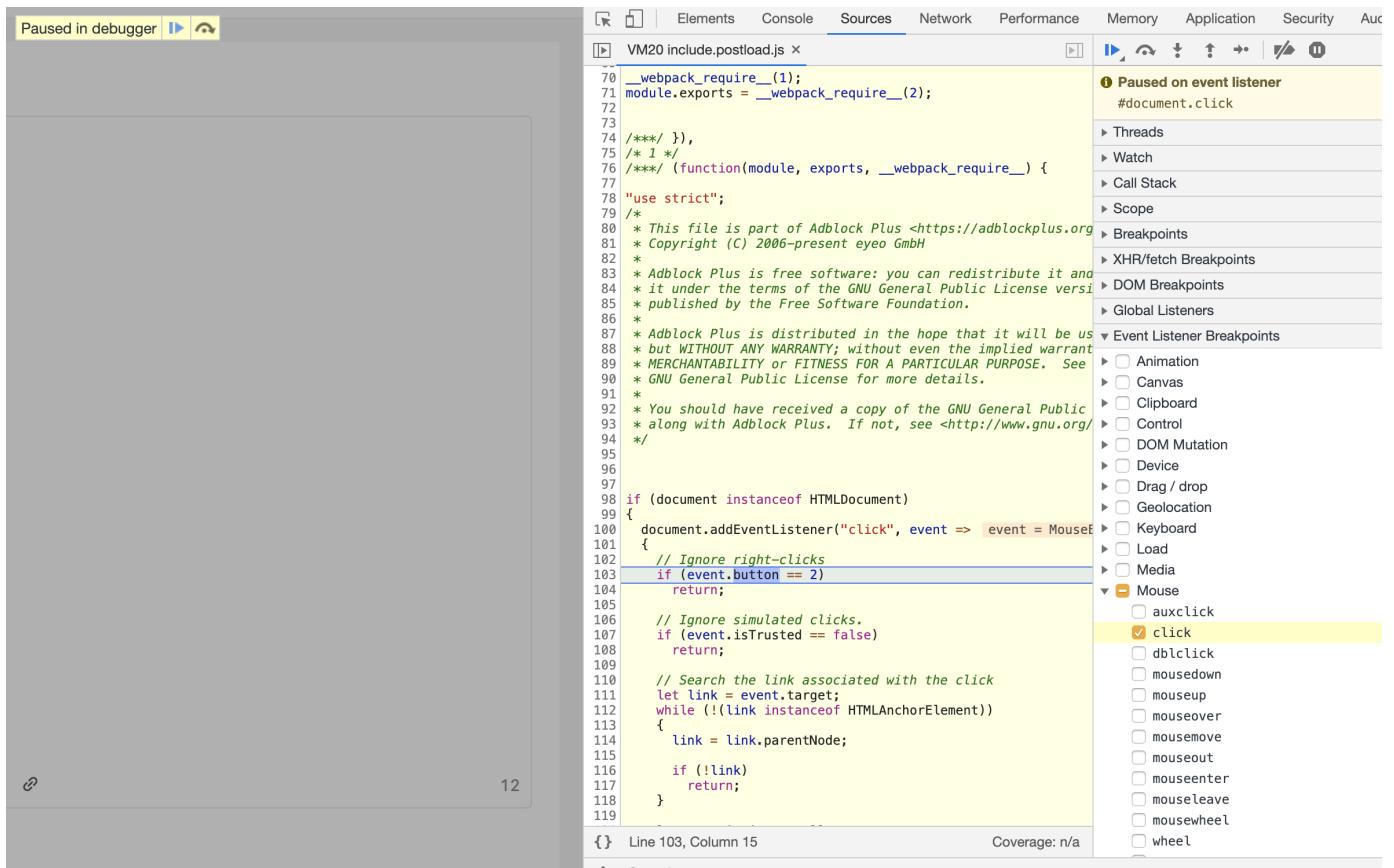
3) Node Removal

删除当前选定的节点时触发，与Subtree modifications相似，通过js删除当前节点属性时，断点会跳转到对应的代码处。

Event listener 断点

Event Listener Breakpoints是一大神器，可以根据事件的名称设置断点。当事件被触发时，断点到事件绑定的位置。事件监听器断点，列出了所有页面及脚本事件，如鼠标、键盘、动画、定时器、XHR等，能极大程度降低业务逻辑在事件层面的调试难度。具体的步骤如下：

1. 打开Sources tab
2. 进入Event listener pane
3. 任选一个Event listener



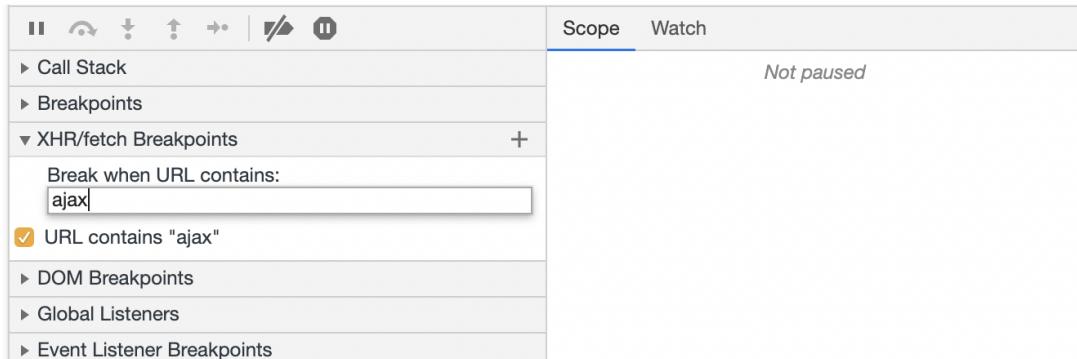
如上图，可以看到，在没有打断点的情况下，勾选了Event listener Breakpoints下的click，当点击页面按钮时，代码会在addEventListener触发click事件的代码行中断。如果使用的不是原生的addEventListener，而是使用了库，比如jquery，Event listener 断点会在jquery内部监听click事件的代码行中断。

XHR 断点

前端与XMLHttpRequest对象有着千丝万缕的联系，不论是原生的ajax，还是经过封装后的axios、request等请求库，实际上都使用了XMLHttpRequest对象，“XHR Breakpoints”正是为异步准备的断点

调试功能。

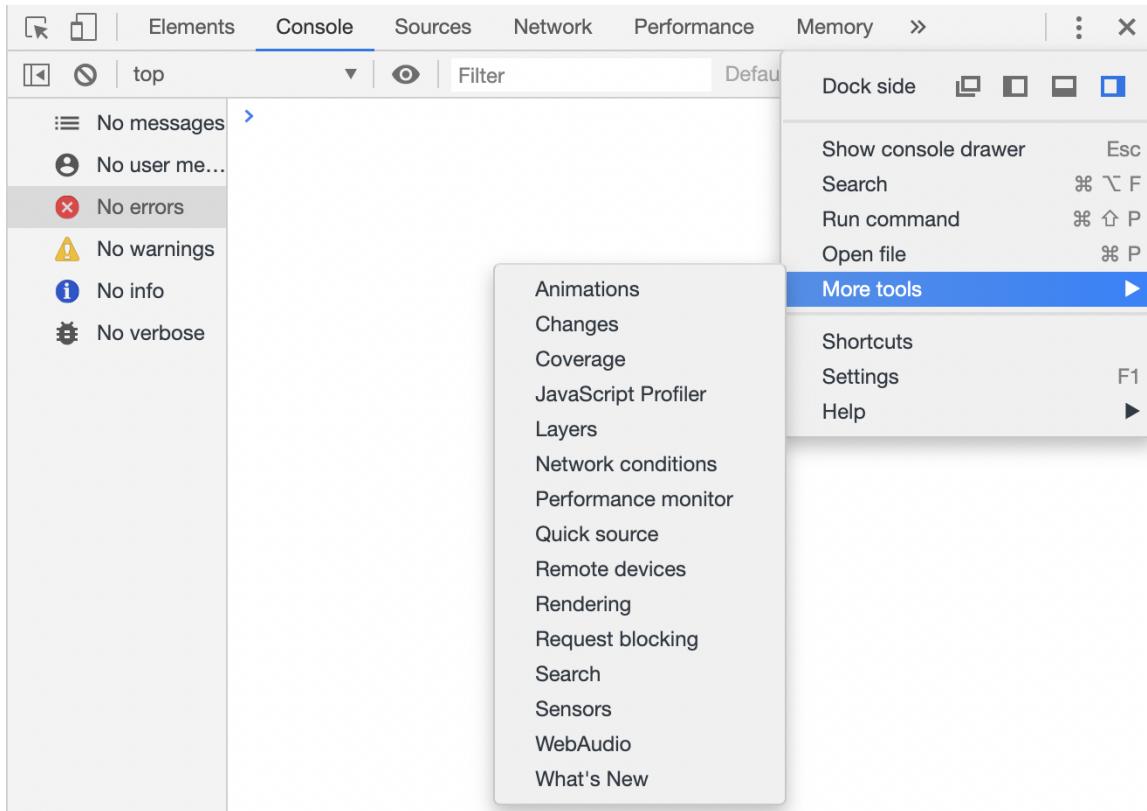
具体的步骤是：打开sources面板，展开XHR Breakpoints，点击Add breakpoint。输入要对其设置断点的字符串。DevTools会在XHR的请求网址的任意位置显示此字符串时暂停。我们也可以添加断点条件表达式，当异步请求触发时的URL满足此条件，js逻辑则会自动产生断点。



XHR断点可以自定义断点规则，我们可以针对某一个或者某一批的请求设置断点。但是目前在开发过程中我们用到XHR断点不多，而更多地采用了Network面板进行请求检。因为目前前端极少直接封装Ajax方法，大多都是直接采用了成熟的请求库，往往这些库都进行了代码压缩，使得XHR断点十分难追踪。而通过Network面板，可以直接看到发送了哪些请求、请求状态、请求返回值等等内容，十分便利。

more tools

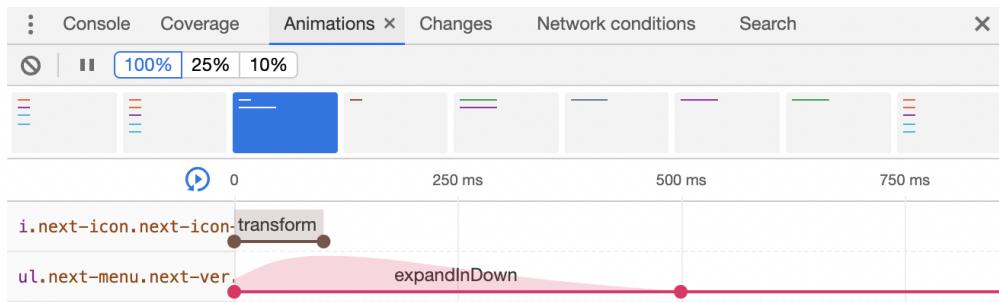
more tools是chrome devtools提供的更多调试的方案，如下图方式打开：



Animations 监听页面动画的变更

在开发过程中，我们可能会使用到css或js书写的animation动画，那么该如何监听动画的变更以便更好地调试与监控呢？例如使用css开发的特殊动画效果，调试起来就会非常复杂。

我们可以打开 `Animation` 面板，通过更改动画时间、延迟、持续时间或关键帧偏移修改动画。动画检查器支持 CSS 动画、CSS 过渡和网络动画。当前不支持 `requestAnimationFrame` 动画。我们可以清除所有当前捕捉的动画组，或者更改当前选定动画组的速度。选择好动画组后，可以在动画窗格中直接进行检查和修改，可以修改对应动画的动画持续时间、关键帧时间、开始延迟时间，可快速完成动画的调整。



Changes 代码变更检测

用于跟踪在DevTools中本地进行的更改，即在chrome中使用Overrides进行线上即时修改后，可以通过Changes页面查看diff结果。

The screenshot shows the Chrome DevTools interface. The top navigation bar includes Elements, Console, Sources, Network, Performance, and Memory. The Sources tab is active, showing a file tree under 'demo3.html'. The right pane displays the contents of 'demo3.js' with the following code:

```
1 const card = document.querySelector('.card-link');
2 card.onclick = function() {
3     card.text = 'hello';
4 }
```

Below the code, it says 'Line 4, Column 2'. The bottom section of the Sources panel shows the Call Stack and Breakpoints, both of which are currently empty.

A separate 'Changes' panel is open at the bottom, showing the file 'demo3.css' with the following CSS rules:

```
1 body {
2     background: #ddd;
3     background: red;
4     font-size: 22px;
5 }
6 .container {
```

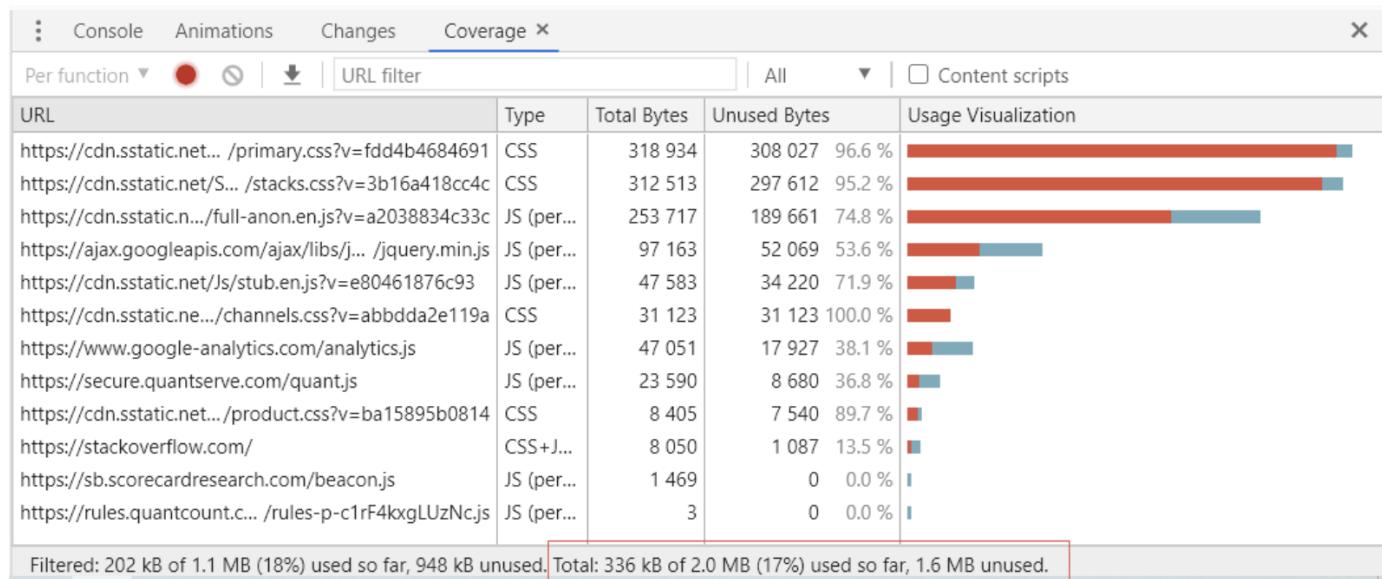
The 'Changes' panel indicates '2 insertions (+), 1 deletion (-)'. The background colors of the CSS rules in the 'Changes' panel correspond to the changes made in the screenshot: the first two lines are red, and the third line is green.

Coverage 代码覆盖率检测

随着时间的推移，每个应用程序通常会累积大量JS和CSS代码，并且其中很大一部分通常由不再使用的代码组成。可以用来检测代码覆盖率，用来优化死代码、懒加载代码，这将加快页面的加载时间并使得代码不再那么复杂，这将使应用程序的维护和进一步开发变得更加轻松。

打开Coverage面板后，只需要按下 `Instrument Coverage`，然后开始执行所需的用例--例如开始浏览应用程序，单击按钮，填写表格等。完成后，只需再按一次按钮即可停止记录并显示覆盖率结果。这是

浏览stackoverflow.com的覆盖结果的示例：



该图显示了在会话期间下载的所有JS和CSS文件，在状态栏的最底部，有一个摘要，可以发现83%的内容都没有使用，即1.6MB。双击摘要表中的一行可以显示该文件的每行报告：

Screenshot of the Chrome DevTools Coverage tab showing the execution coverage of primary.css.

The Coverage tab displays the following information:

- Filesystem** workspace selected.
- Coverage** tab selected.
- URL filter**: `https://cdn.sstatic.net/primary.css?v=fdd4b4684691`
- Usage Visualization** table:

URL	Type	Total Bytes	Unused Bytes	Usage Percentage
<code>https://cdn.sstatic.net/primary.css?v=fdd4b4684691</code>	CSS	318 934	307 683	96.5 %
<code>https://cdn.sstatic.net/stacks.css?v=3b16a418cc4c</code>	CSS	312 513	297 612	95.2 %
<code>https://cdn.sstatic.net/full-anon.en.js?v=a2038834c33c</code>	JS (per...)	253 717	189 661	74.8 %
<code>https://ajax.googleapis.com/ajax/libs/jquery/jquery.min.js</code>	JS (per...)	97 163	51 772	53.3 %
<code>https://cdn.sstatic.net/Js/stub.en.js?v=e80461876c93</code>	JS (per...)	47 583	33 586	70.6 %
<code>https://cdn.sstatic.net/channels.css?v=abbdda2e119a</code>	CSS	31 123	31 123	100.0 %
<code>https://www.google-analytics.com/analytics.js</code>	JS (per...)	47 051	17 927	38.1 %
<code>https://secure.quantserve.com/quant.js</code>	JS (per...)	23 590	8 680	36.8 %
<code>https://cdn.sstatic.net/product.css?v=ba15895b0814</code>	CSS	8 405	7 540	89.7 %
<code>https://stackoverflow.com/</code>	CSS+J...	8 050	1 087	13.5 %
<code>https://sb.scorecardresearch.com/beacon.js</code>	JS (per...)	1 469	0	0.0 %
<code>https://rules.quantcount.com/rules-p-c1rF4kxgLUzNc.js</code>	JS (per...)	3	0	0.0 %

绿线表示该行代码已经完全执行，红线表示该行代码在录制会话时并没有使用，红/绿线表示仅使用部分代码但未全部执行。

JavaScript Profiler 查看原生函数的执行性能

Chrome长期目标是将所有人迁移到新的工作流程。将来的DevTools版本中可能会删除此工作流程：

The screenshot shows the Chrome DevTools JavaScript Profiler interface. On the left, there's a sidebar titled 'Profiles' with a section for 'Record JavaScript CPU Profile'. A red arrow points from this section down to the 'Start' button at the bottom of the main panel. Another red arrow points from the 'Start' button to the right-hand table, which displays a detailed list of CPU profile data. The table has columns for 'Self Time', 'Total Time', and 'Function'. The data includes various JavaScript functions and their execution times, such as '(idle)', '(program)', and 'pushFactorSet'. The right side of the interface also shows tabs for Elements, Console, Sources, Network, Performance, and JavaScript Profiler.

Self Time	Total Time	Function
33616.3 ms	33616.3 ms	(idle)
1736.5 ms	42.39 %	(program)
322.6 ms	7.88 %	▸ (anonymous)
271.3 ms	6.62 %	▸ pushFactorSet
269.2 ms	6.57 %	▸ (garbage collector)
125.8 ms	3.07 %	▸ 6.1 ms
118.1 ms	2.88 %	▸ pushWorkflow
28.4 ms	0.69 %	▸ initFactorSet
26.5 ms	0.65 %	▸ transformExpression
26.1 ms	0.64 %	▸ _callee\$
25.3 ms	0.62 %	▸ clearFactorSet
25.2 ms	0.62 %	▸ mark
24.7 ms	0.60 %	▸ setFocus
22.3 ms	0.54 %	▸ measure
		▸ ./node_modules/_core-js@3.6.5@core.js
		inspect-source.js

Layers 查看图层和页面重绘

在Layers面板中，可以查看网站中当前存在的所有图层——这些图层以元素周围的边框表示或可以3D模式查看，这也有助于了解页面的堆叠上下文。与元素进行交互时，会更新页面图层，展示操作如何影响网站以及页面中必须重绘的那些部分，通过此面板的检查可以降低关键点的重绘成本。

The screenshot shows the Chrome DevTools Layers panel. On the left, there's a sidebar with a list of CSS selectors and their bounding boxes. In the center, there's a visual representation of the page's layers, each outlined with a colored border (pink, blue, yellow). Below this, there's a 'Details' section with various performance metrics. A red arrow points from the 'Slow scroll regions' section in the details to the corresponding area in the visual representation.

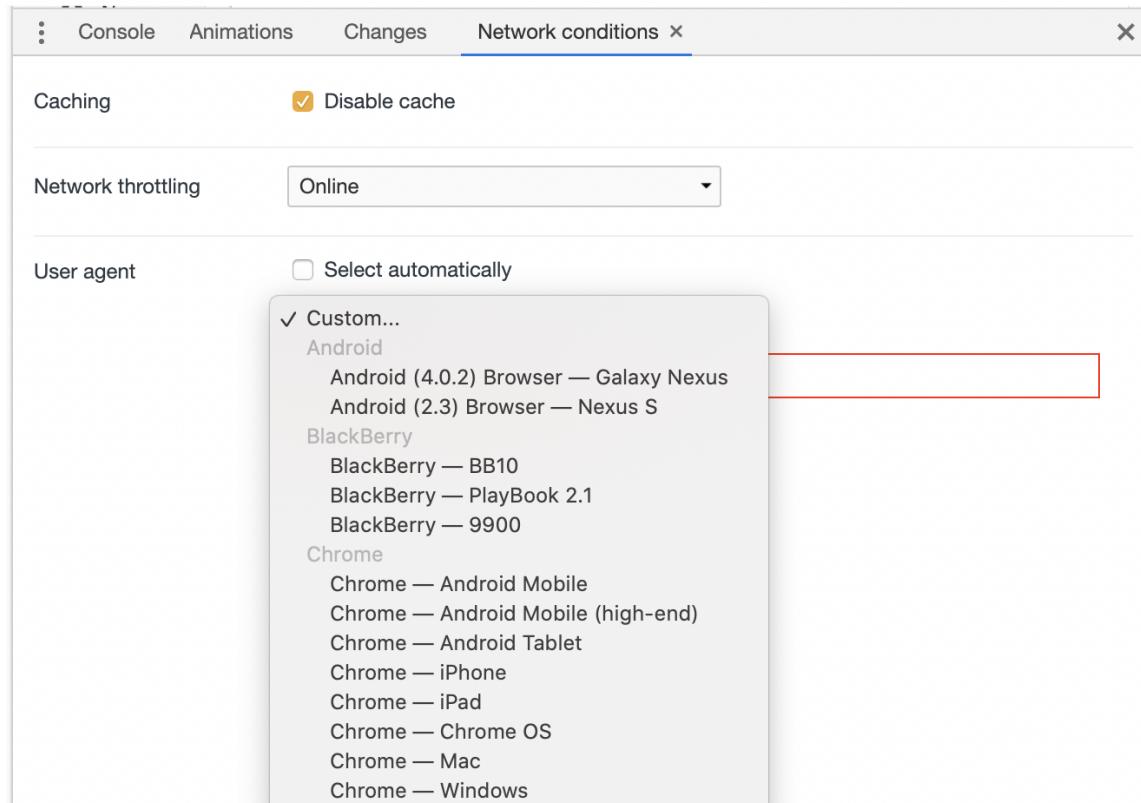
Size	847 x 1210 (at 0,0)
Compositing Reasons	Has a clip that needs to be known by compositor because of composited descendants.
Memory estimate	4.1 MB
Paint count	13
Slow scroll regions	Touch event handler 847 x 1210 (at 0, 0)
Sticky position constraint	

由于Layers面板运行时的耗费的性能较多，因此在进行检查时可能会卡住或者崩溃，但它还是可以发现一些难以发现的性能瓶颈问题。

Network conditions 修改网络环境

User-Agent 首部包含了一个特征字符串，用来让网络协议的对端来识别发起请求的用户代理软件的应用类型、操作系统、软件开发商以及版本号。

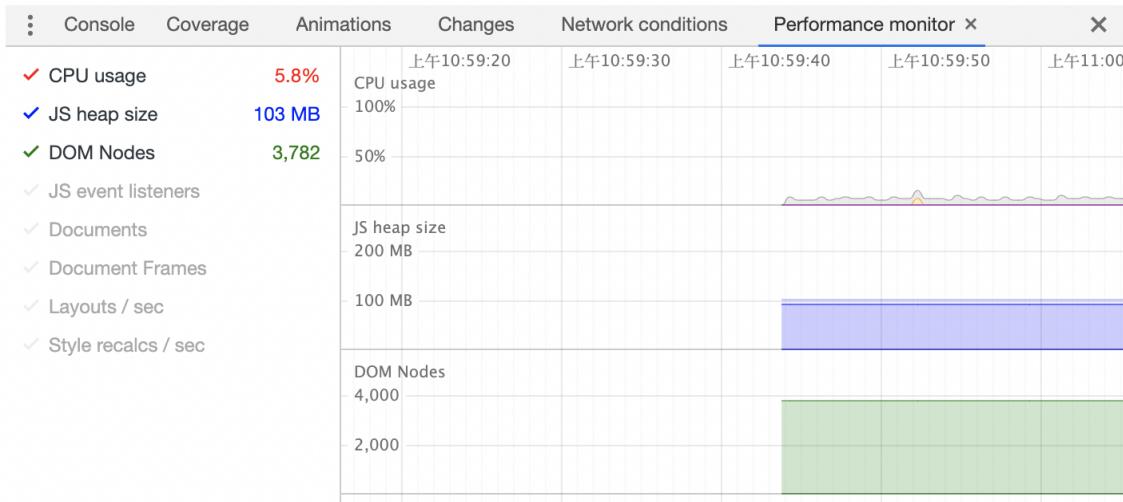
在此面板中，可以设置 User agent 模拟手机、浏览器等不同平台的请求情况。这样，在调试不同类型的浏览器、不同的浏览器版本、乃至手机端时，可以迅速切换完成调试，而不需要频繁地更换设备。去勾选 User agent 中的 Select automatically，在下拉框中选择待调试的 User agent。



Performance monitor 性能监控器

使用性能监视器可以实时查看页面负载或运行时性能的各个方面，包括：

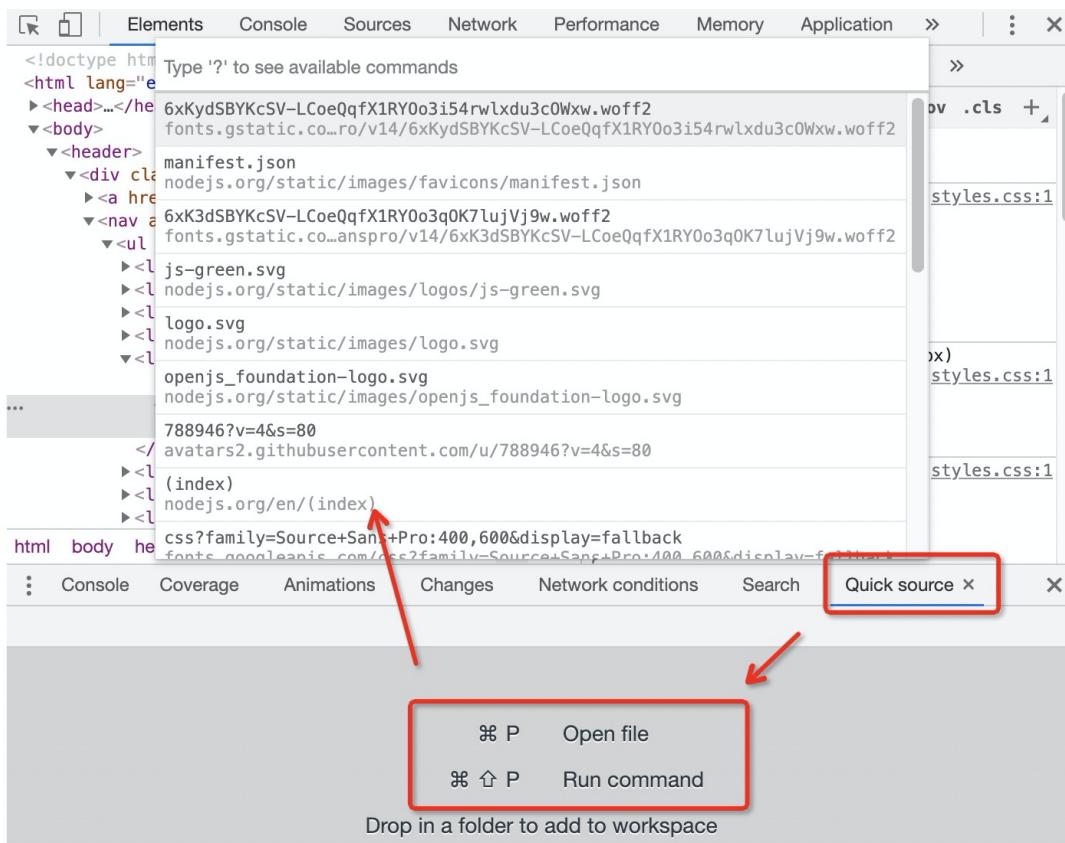
- CPU 使用率
- JavaScript 堆大小
- 页面上的 DOM 节点
- JavaScript 事件侦听器
- 文档和框架的总数
- 每秒重新计算布局和样式



单击一个指标可以显示或隐藏它。在图中，显示了CPU使用率，JS堆大小和Dom节点的性能图表。

Quick source 快速源窗格

可快速地查找所有资源，例如可更快地进行CSS编辑。打开Quick source可立即了解样式页面中的CSS（也可与Sass一起使用）与源文件CSS之间的映射。并且可以添加，修改和删除“快速源”窗格中反映的CSS或者添加新的样式规则。



Remote devices 远程调用

可以用于远程连接移动设备如Android或IOS进行调试：

This panel has been deprecated in favor of the <chrome://inspect/#devices> interface, which has equivalent functionality.

直接通过chrome://inspect/#devices访问即可

DevTools Devices

Devices

Pages

Extensions

Apps

Shared workers

Service workers

Other

Discover USB devices

Discover network targets

Port forwarding...

Configure...

Open dedicated DevTools for Node

Remote Target #LOCALHOST

Rendering 监控页面重绘的方法

如果我们发现web页面非常卡顿，我们可以打开Rendering面板，监听是页面的哪些区域进行了重绘动作。

Console Coverage Animations Changes Network conditions Search Rendering

Paint flashing 高亮页面重绘区域
 Highlights areas of the page (green) that need to be repainted. May not be suitable for people prone to photosensitive epilepsy.

Layout Shift Regions 高亮重绘区域边界
 Highlights areas of the page (blue) that were shifted. May not be suitable for people prone to photosensitive epilepsy.

Layer borders 展示层边界
 Shows layer borders (orange/olive) and tiles (cyan).

FPS meter 记录FPS帧率
 Plots frames per second, frame rate distribution, and GPU memory.

Scrolling performance issues
 Highlights elements (teal) that can slow down scrolling, including touch & wheel event handlers and other main-thread scrolling situations.

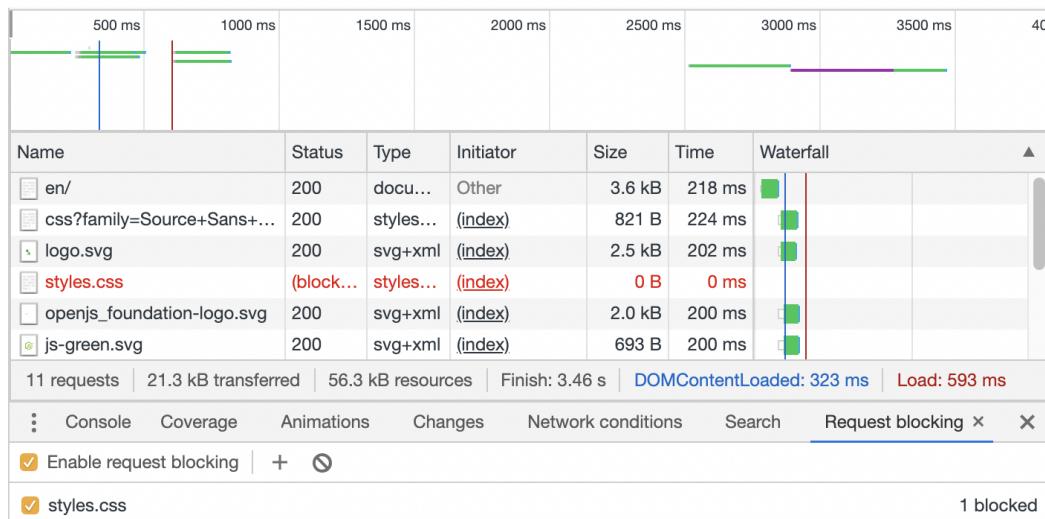
Highlight ad frames
 Highlights frames (red) detected to be ads.

Hit-test borders
 Shows borders around hit-test regions.

Emulate CSS media type
Forces media type for testing print and screen styles

Request blocking

当某些脚本、样式文件缺少或者其他资源加载失败时，可以添加规则并阻塞相应请求以看到网页的样子。



Search

用于快速从源代码中搜索内容。

```
▼ index.js — 127.0.0.1:3333/js/index.js
4062 ...rt */ var _src_global_scss__WEBPACK_IMPORTED_MODULE_13__ = __webpack_require__(/*! ..../src/global...
4063 ...rt */ var _src_global_scss__WEBPACK_IMPORTED_MODULE_13__ default = /*#__PURE__*/ __webpack_r...
6263 ... 支持 JS 和 CSS。
6264 ...g} url JS / CSS 的资源地址。
6268 ...ion} o.error 超时或发生错误时回调函数。当资源文件为 CSS 文件时不支持;
6269 ...r} o.timeout 单位为秒，默认无限大。超时后触发 error 回调。当资源文件为 CSS 文件时不支持。
6393 ...odules/_@ali_ice-ajax@0.3.5@@ali/ice-ajax/lib/main.scss":
6395 ...ules/_@ali_ice-ajax@0.3.5@@ali/ice-ajax/lib/main.scss ***!
6400 ... mini-css-extract-plugin
6403 var cssReload = __webpack_require__(/*! ../../css-hot-loader@1.4.4@css-hot-loader/hotModuleReplace...
6404 module.hot.dispose(cssReload);
6405 ...pt(undefined, cssReload);
6471 ...re_/*! ./main.scss */ ./node_modules/_@ali_ice-ajax@0.3.5@@ali/ice-ajax/lib/main.scss");
6898 ...yle = rgba["css-rgba"];
6908 ...stop.position, rgba["css-rgba"]);
6951 ...rgba["css-rgba"]);
6962 ...stop.position, rgba["css-rgba"]);
7016 ...olor = utils_1.colorToJSON(shadow.color)["css-rgba"];
7683 ...= colorObj['css-rgba'];

Show 1285 more
```

Sensors

可以模拟Geolocation地理位置、orientation方向、touch触摸。

The screenshot shows the Sensors tab in the developer tools. It includes sections for Geolocation (set to 'No override'), Orientation (set to 'Off'), and Touch (set to 'Device-based'). Each section has input fields for latitude, longitude, and timezone ID, as well as alpha, beta, and gamma values for orientation.

WebAudio

用于检查AudioContexts的详细信息–Web Audio API的一部分。音频检查器显示以下信息：

- 状态（运行中）
- 采样率
- 回调缓冲区大小
- 最大输出通道
- 当前播放时间
- 回调间隔
- 渲染能力

The screenshot shows the WebAudio tab in the developer tools. It displays an AudioContext object with the identifier '8a19c81b-4146-4912-af46-34899f948c3d'. Below it, detailed configuration parameters are listed:

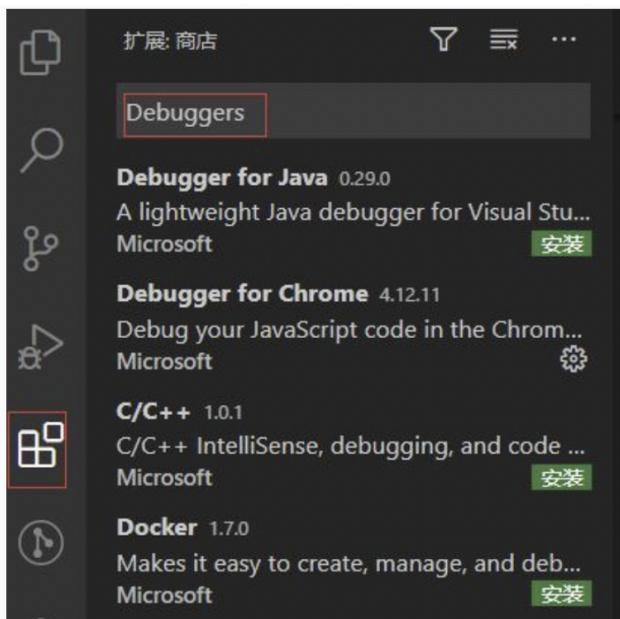
State	running
Sample Rate	48000 Hz
Callback Buffer Size	256 frames
Max Output Channels	2 ch

vscode断点调试

使用vscode直接进行断点调试也是非常方便的。具体内容可以参考[vscode官网之调试](#)。

VS Code具有对Node.js运行时的内置调试支持，并且可以调试js、ts或任何其他可转换为js语言。若需要

调试其他语言，如Python、C++、Go等，可以在VS Code Marketplace查找Debuggers扩展。

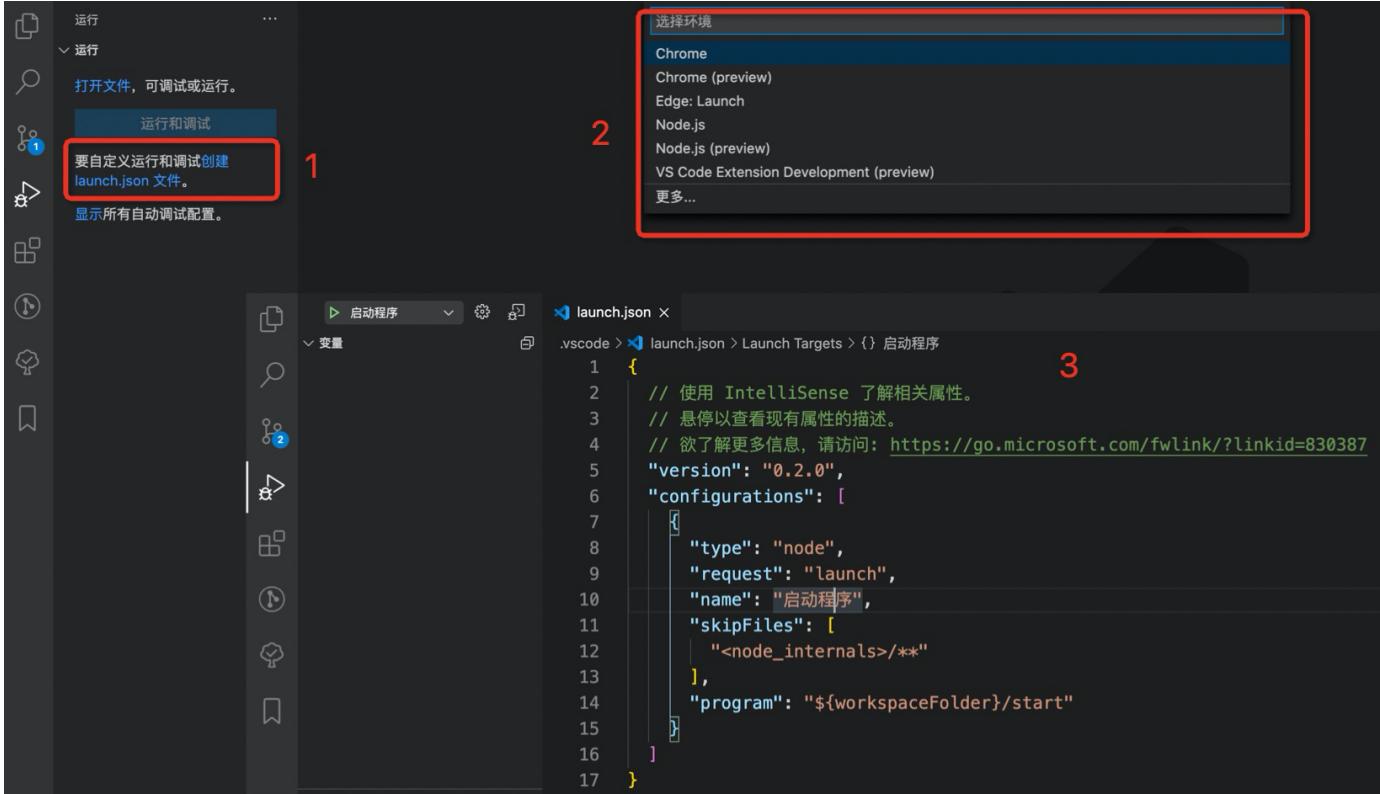


nodejs调试

要在VS Code中运行或调试一个简单的应用程序，请按F5键，VS Code将尝试运行您当前处于活动状态的文件。

但是，对于大多数调试方案，创建启动配置文件是有好处的，因为它使您可以配置和保存调试设置详细信息。VS Code将调试配置信息保留在工作空间launch.json中的.vscode文件夹（项目根文件夹）中或用户设置或工作空间设置中的文件中。

要创建launch.json文件，在vscode中打开项目文件夹（File > Open Folder），然后在Run视图点击“创建launch.json文件”。选择好对应环境后，将在根目录下创建一个.vscode文件夹，其中包含了launch.json文件。



我们看到 `.vscode/launch.json` 的 `configurations.program` 属性为 `${workspaceFolder}/start`，表示调试的入口文件，其中 `workspaceFolder` 是 vscode 资源管理器的根目录。

点击绿色箭头，即可启动调试。

npm scripts 调试

在实际项目中，命令基本上都是放到了 npm scripts 中。

首先添加 npm scripts，其中 6666 是任意指定的调试端口号：

```

18 "scripts": {
19     "debug": "node --inspect-brk=6666 index.js"
20 }

```

打开 `.vscode/launch.json`，删除 `program` 属性，增加以下 3 个配置项：`runtimeExecutable`、`runtimeArgs`、`port`。

The screenshot shows the VS Code interface with two tabs open: package.json and launch.json. The launch.json tab contains the following configuration:

```
10 "name": "启动程序",
11 "skipFiles": [
12   "<node_internals>/**"
13 ],
14 "runtimeExecutable": "npm",
15 "runtimeArgs": [
16   "run-script",
17   "debug"
18 ],
19 "port": 6666
20 }
21 ]
22 }
```

A red box highlights the runtimeExecutable, runtimeArgs, and port sections. Below the tabs, the Variables sidebar is visible, showing the Local scope with several properties listed.

最后启动调试的方法同以上的nodejs调试。点击绿色箭头，即可启动调试。

非node命令调试

npm run会自动添加`node_module/.bin` 到当前命令所用的环境变量中，例如：

```
1 {
2   "scripts": {
3     "build": "webpack"
4   }
5 }
```

运行`npm run build`实际上是调用`node_modules/.bin/webpack`，会根据当前环境调用对应的脚本，内部实际上调用的是`node ./node_modules/webpack/bin/webpack.js`。具体的调试步骤如

下：

1. 修改 scripts

```
18 "scripts": {  
19     "debug": "node --inspect-brk=3000 ./node_modules/webpack/bin/webpack.js"  
20 },
```

2. 启动调试

3. 跳转至 node_modules/webpack/bin/webpack.js

```
1  #!/usr/bin/env node  
2  
3 // @ts-ignore  
4 process.exitCode = 0;  
5  
6 /**  
7 * @param {string} command process to run  
8 * @param {string[]} args commandline arguments  
9 * @returns {Promise<void>} promise  
10 */  
11 const runCommand = (command, args) => {  
12     const cp = require("child_process");  
13     return new Promise((resolve, reject) => {
```

总结

以上的调试、测试方法并不完备，只能起到一个抛砖引玉的作用。整理的过程中发现调试技巧非常繁多，无法一一列举，只列举了一些日常工作中可能常使用的一些方法。希望大家在未来的工作之中，能够根据自己的业务场景，掌握相关的调试技巧和测试技巧，让工作变得更加高效。