

# Tbrowser Remote DevTools 使用说明

王勇望

2018 年 5 月 2 日

# 1 引言

工欲善其事，必先利其器。优秀的工具辅助，通常能使工作效率大幅度提升。而我们今天所要介绍的 Tbrowser Remote DevTools（Tbrowser 远程页面调试工具）就是这样的利器。它能使 TV 上页面调试工作拥有和 PC 上几乎一样的体验。话不多说，让我们尽快开始 Tbrowser Remote DevTools 学习之旅吧。

## 2 打开 Tbrowser Remote DevTools

为了保证浏览器性能，Tbrowser Remote DevTools 是默认关闭的。不过，只需要简单的操作就可以打开此功能。我们提供两种方法打开 Tbrowser 调试功能：

- 通过 tcli 命令打开

需要执行的命令是：/tvos/bin/tcli tbrw2.enableddevtools.pid

pid 是浏览器的进程号，可以通过 /tvos/bin/tcli help | grep tbrw2 命令查看。

这种打开方式不需要重启 TV，直接可以使用。但重启之后失效，还需再次执行命令。

```
# /tvos/bin/tcli help | grep tbrw2
tbrw2.clearcache.627      -      clear cache
tbrw2.destorypage.627     -      destory page
tbrw2.enableddevtools.627 -      enable devtools
tbrw2.goback.627          -      go back
tbrw2.goforward.627       -      go forward
tbrw2.inputstr.627        -      input string
tbrw2.openurl.627         -      set page's url
tbrw2.reload.627          -      reload page
tbrw2.setactive.627       -      set page visible
tbrw2.setbgtransparent.627 -      set background transparent
tbrw2.setcookieenable.627 -      set Cookie enable
tbrw2.setdonottrack.627   -      set do not track
tbrw2.setdrawfocus.627   -      set draw focus
tbrw2.setloadjsextpath.627 -      set load jsext path
tbrw2.setpluginfilepath.627 -      set plugin file path
tbrw2.setrequestlanguage.627 -      set http request language
tbrw2.setuseragent.627    -      set UserAgent
tbrw2.setvisible.627      -      set page visible
tbrw2.setzindex.627       -      set zindex
# /tvos/bin/tcli tbrw2.enableddevtools.627
tcli _threadLoop start
Run tbrw2.enableddevtools.627
enableddevtools
Run tbrw2.enableddevtools.627 end ,ret = 0
tcli _threadLoop proxy_exit start
tcli _threadLoop proxy_exit end
tcli pthread_join,wait thread,ret=0
tcli proxy_serv return 0
#
```

图 2.1: 通过 tcli 打开过程

- 通过添加启动参数

需要添加的参数是：-remote-debugging-port=9222

修改启动 Tbrowser 的 run\_weblauncher\_server.sh 脚本，添加上述的启动参数，重启 TV 即可。此种打开方式会一直打开 Tbrowser Remote DevTools，重启依然有效，所以请不要在 SQA 的机器上这样修改。

Tbrowser Remote DevTools 功能打开后，Tbrowser 会监听指定的端口，创建一个 http 服务，供 PC 端访问。可以通过 netstat -anp 命令查看系统的端口使用情况。如果 Tbrowser Remote DevTools 功能开启，你将看到图 2.2 中的信息。其中显示的 IP 是你机器的 IP 地址。如果使用 tcli 打开，端口默认是 9222；如果添加了启动参数，那么端口就是参数中指定的端口号。

```
# netstat -anp
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 192.168.1.101:49152    0.0.0.0:*               LISTEN      1511/nscreenService
tcp        0      0 192.168.1.101:9222    0.0.0.0:*               LISTEN      627/weblauncher_ser
tcp        0      0 0.0.0.0:9999         0.0.0.0:*               LISTEN      1511/nscreenService
tcp        0      0 0.0.0.0:56789        0.0.0.0:*               LISTEN      1115/--type=child
tcp        0      0 0.0.0.0:6550         0.0.0.0:*               LISTEN      1511/nscreenService
tcp        0      0 0.0.0.0:56790        0.0.0.0:*               LISTEN      1115/--type=child
tcp        0      0 0.0.0.0:6553         0.0.0.0:*               LISTEN      1511/nscreenService
tcp        0      0 0.0.0.0:5978         0.0.0.0:*               LISTEN      1687/netflix
tcp        0      0 0.0.0.0:4123         0.0.0.0:*               LISTEN      1511/nscreenService
tcp        0      0 0.0.0.0:6557         0.0.0.0:*               LISTEN      1514/MonitorServer
tcp        1      0 192.168.1.101:50024    217.175.69.232:80      CLOSE_WAIT  622/sitatvservice
tcp        1      0 192.168.1.101:50025    217.175.69.232:80      CLOSE_WAIT  622/sitatvservice
udp        0      0 0.0.0.0:54795        0.0.0.0:*               1511/nscreenService
udp        0      0 192.168.1.101:58963    192.168.88.1:53        ESTABLISHED 1687/netflix
udp        0      0 127.0.0.1:46923       0.0.0.0:*               1511/nscreenService
udp        0      0 0.0.0.0:1900         0.0.0.0:*               1511/nscreenService
udp        0      0 239.255.255.250:1900  0.0.0.0:*               1115/--type=child
udp        0      0 0.0.0.0:6537         0.0.0.0:*               1511/nscreenService
```

图 2.2: netstat -anp 命令输出结果

到这里就非常接近成功了，后续要做的事情就是将 PC 连接到 TV 所在的网络（必须是直接连接，使用代理是不行的）。打开 PC 上的 Chrome 浏览器，输出 Tbrowser 监听的地址，如上图应该是 <http://192.168.1.101:9222>。如果一切顺利，你将看到图 2.3 中显示的内容。其中 BBC iPlayer、index.html、livetv、systemUi 为打开的链接 title。如果此时浏览器没有打开任何页面，将没有任何链接。

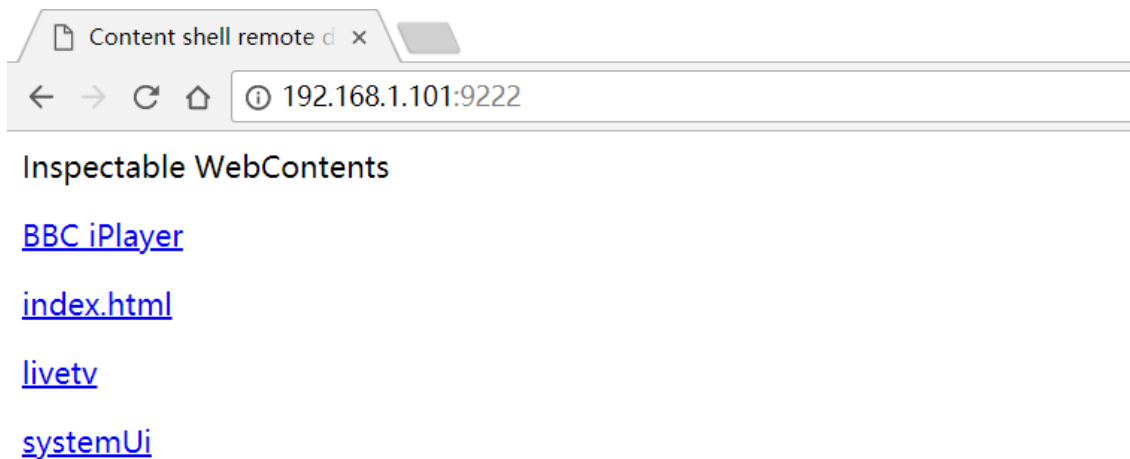


图 2.3: Tbrower Remote DevTools 首页

你可以点击其中的一个链接，进入调试页面，例如我打开 BBC iPlayer 链接，显示的效果如图 2.4 所示。至此，Tbrowser Remote DevTools 功能已经准备妥当，下一步就可以进行页面调试了。

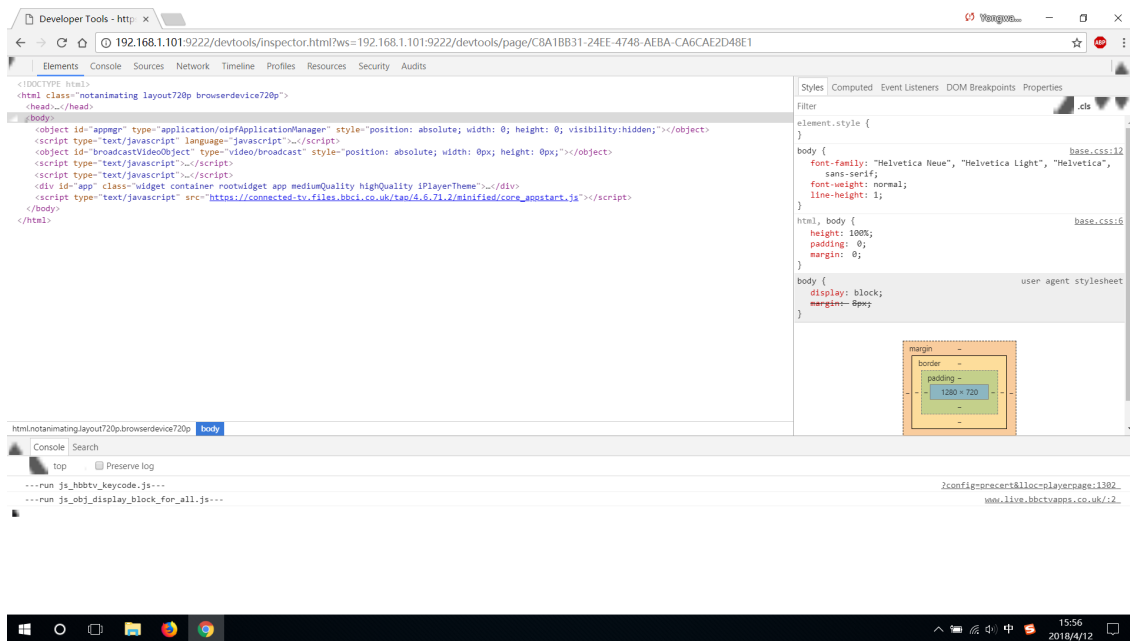


图 2.4: Tbrower Remote DevTools 调试页面

### 3 使用 Tbrowser Remote DevTools

目前的 Tbrowser2.0 是基于 Chromium49 开发的,所以这里看到的调试页面是 Chromium49 版的调试页面。有些功能和 PC 上的开发者工具不同,但大多数常用的功能还是一致的。这里我们按照模块来研究一下 Tbrowser Remote DevTools 的具体使用情况。

#### 3.1 Elements 面板

Elements 面板功能和最新的 Chrome DevTools 几乎没有区别,对于页面开发者来说这个模块的功能也应该非常熟悉。Elements 主要用于检查和实时编辑页面的 HTML 与 CSS。

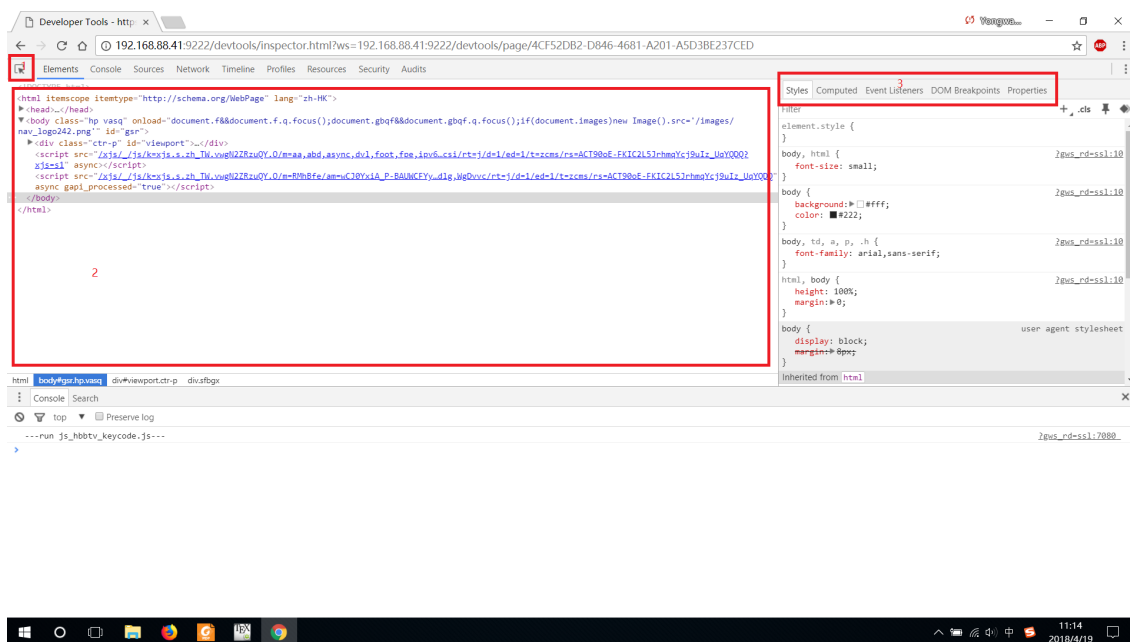


图 3.1: DevTools Elements Panel

如图 3.1所示,是 Elements 面板的界面。其中常用到的功能区已经在图中用红框标记出来了。中间最大的红框 2 标记的区域是用来展示 DOM 树视图,这里的 DOM 树视图是经过 JS 执行之后的结果。当鼠标移动到代码中的某个 element 上时,在 TV 上会使用不同的背景色高亮这些元素,并显示该元素的大小与位置坐标。

```

<div class="sfbg nojsv" style="margin-top:-20px">...
<form class="tsf" action="/search" style="overflow
<div id="tophf">...</div>
<div class="tsf-p">
  <div class="nojsv logocont" id="logocont">...</di
  <div class="sfibbbc">...</div>
  <div class="jsb" style="padding-top:18px">
    <center>
      <input value="Google 搜尋" aria-label="Goog
      <input value="试一试" aria-label="好手氣" na
    </center>
  </div>
</div>

```

图 3.2: 修改 Google 页面代码



图 3.3: 修改 Google 页面效果

双击 HTML 代码可以进行编辑文本，或者右键选择 Edit as HTML（快捷键是 F2）进行编辑，编辑后的页面会同步更新。如图??就是将 google 页面中的“好手气”改为“试一试”之后的效果。

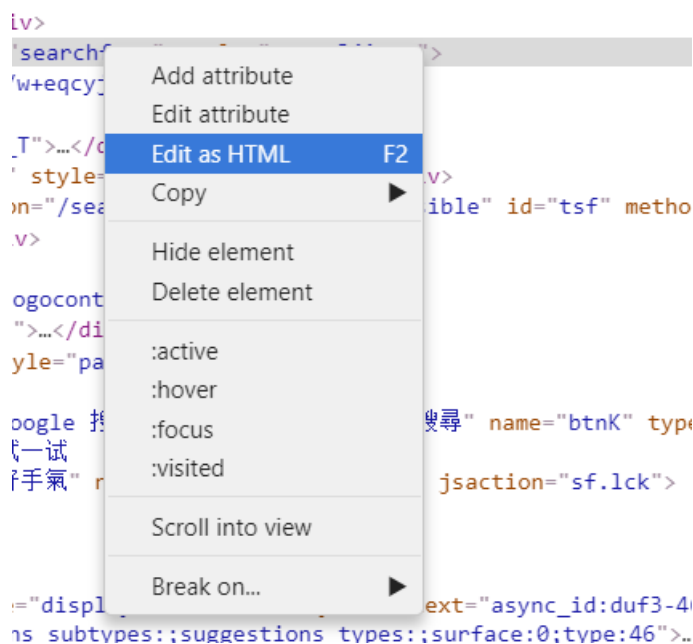


图 3.4: Elements 右键菜单

另外，如图 3.4 在这个区域的右键菜单中还提供了一些其他功能，下面也做出一些简要说明：

- Add attribute: 为选中的元素添加一个属性。
- Edit attribute: 编辑鼠标所在位置的属性，与双击鼠标作用相同。
- Edit as HTML: 编辑 HTML 代码，会打开一个编辑框进行代码编辑。点击编辑框以外的部分或者 Esc 键可以退出编辑模式。
- Copy: 在远程调试中 Copy 是不可用的。在调试过程中，可以复制文本，但不能使用

任何的 Copy 或 Save 功能。因为这些命令会发送到 TV 端执行，而 Server 端禁止执行这些命令。

- **Hide element:** 隐藏或取消隐藏选中的元素。
- **Delete element:** 删除选中的元素。注意：在远程调试中 **element** 一旦删除，无法撤销，只有重新加载页面才能恢复。
- **:active :hover :focus :visited :** 强制修改页面伪状态。
- **Scroll into view:** 控制选中的 **element** 对应的页面元素滚动到可见区域。
- **Break on ...:** 添加 Dom Breakpoint。关于 Dom Breakpoint 会在讲解 Sources 模块时详细说明。

图 3.1 中左上角的红框 1 标记出图标的功能是通过点击页面元素，来定位其在代码中的位置。这个功能和 Chrome 的 Devtools 的功能一样。只不过有一个前提就是打开这个页面的应用要响应鼠标事件。此功能的快捷键是 **Ctrl+Shift+C**。

图 3.1 中右上角的红框 3 标记出来的一系列 Tabs 是用来查看代码区选中 **element** 的各项属性的。下面简要介绍一下各个标签的作用：

- **Styles** 用于展示对选中的 **element** 的 CSS 属性。其中包含对其自身或者其父节点的 CSS 属性设置。没有生效的属性会使用横线划掉表示。在这个标签内可以勾选每条属性前的复选框来改变页面中的节点 CSS 属性，也可以直接编辑。
- **Computed** 用于展示选中 **element** 计算之后的 CSS 属性，这里会列出该 **element** 所有被设置的属性。展开其中的一条属性可以看到在这个节点上对该属性所有设置的值和设置的代码位置。没有生效的值会使用横线划掉。**Computed** 标签内的值不能修改。
- **Event Listeners** 用于展示选中的 **element** 监听的 JS 事件。展开其中一条事件可以看到事件监听方法在代码中的位置，点击代码路径可以调整到对应的函数。这个标签用于调试 JS 事件的处理情况非常方便。
- **Dom Breakpoints** 用于展示和管理 Dom Breakpoint。关于 Dom Breakpoint 会在讲解 Sources 模块时详细说明。
- **Properties** 用于展示选中的 **element** 所有的 **property**。其中包含了 **element** 自身和它的原型链的所有 **property**。

更多信息可以查看 Chrome Devtools 说明文档：<https://developers.google.com/web/tools/chrome-devtools/inspect-styles/?hl=zh-cn>

## 3.2 Console 面板

Chrome DevTools 的 Console 面板主要有两个功能：查看页面 log 信息和执行命令。但是目前 Tbrowser Remote DevTools 只支持查看页面 log 的功能，暂时不支持命令的执行。

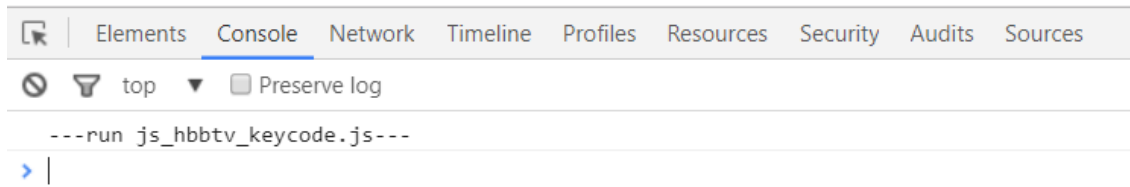


图 3.5: Console 界面

如图 3.5所示，是 Console 的界面。其中上面的一排按钮的功能分别为：

- Clear console 清除 log，用于清除当前 Console 界面中的 log。快捷键是 Ctrl+L。
- Filter 用于根据输入的关键字过滤 log 信息，支持正则表达式。
- iframe 选择 iframe，如果页面使用多个 iframe，可以通过这里切换 iframe。top 是 main frame；默认为当前操作的 iframe。
- Preserve log 保持 log，如果勾选此选项，那么在页面切换或者重新加载是不会清除以前的 log 信息。

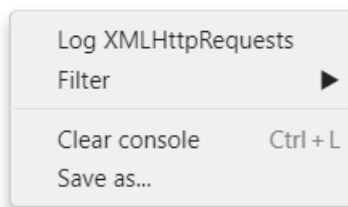


图 3.6: Console 界面

如图 3.6所示，在 Console 的右键菜单中还提供了一些功能：

- log XMLHttpRequest 如果勾选此选项会在执行 XMLHttpRequest 时输出 log。
- Filter 此功能和按钮的功能不一样，这里是用于隐藏某个文件中所有的 log。
- Clear console 和按钮的功能一致。
- Save as... 此功能不可用。



更多信息可以查看 Chrome Devtools 说明文档:<https://developers.google.com/web/tools/chrome-devtools/console/?hl=zh-cn>

### 3.3 Sources 面板

Sources 面板主要用于页面代码的浏览器与调试，主要是对 JavaScript 代码的调试使用较多。

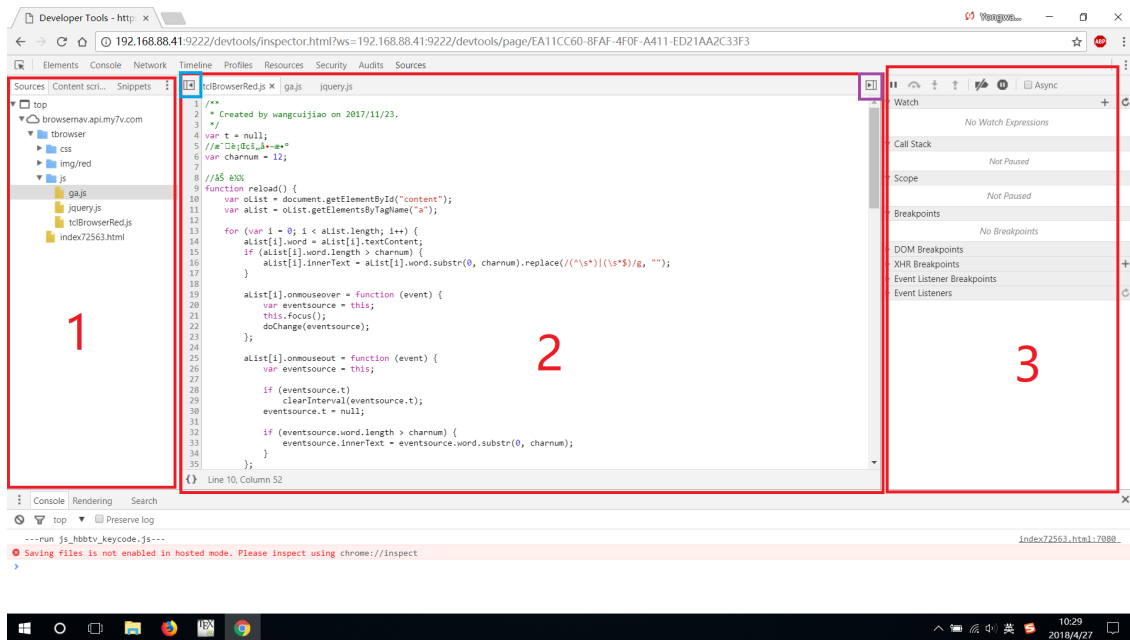


图 3.7: Sources 界面

如图 3.8 所示，是 Sources 界面。如图中的红框标记，Sources 界面主要分为三个区域。

- 红框 1 标记出的左侧工具栏用于文件浏览，默认是按照文件夹的结构显示的。
- 红框 2 标记出的中间区域用于代码浏览和调试。其中左上角和右上角的两个按钮用于显示和隐藏左右工具栏。左下角的大括号图标用于格式化 JS 代码。
- 红框 3 标记出的右侧工具栏用于查看调试过程中的变量、堆栈和管理断点。

在开始熟悉 JavaScript 调试之前，了解一下 JavaScript 调试的入门知识非常有好处。Chrome Devtools 说明文档中《在 Chrome DevTools 中调试 JavaScript 入门》就是一个不错的教程，链接是：<https://developers.google.com/web/tools/chrome-devtools/javascript/?hl=zh-cn>。为了方便我将其内容添加到这里。

# 在 Chrome DevTools 中调试 JavaScript 入门



By Kayce Basques

(<https://developers.google.com/web/resources/contributors/kaycebasques?hl=zh-cn>)

Technical Writer for Chrome DevTools

本交互式教程循序渐进地教您在 Chrome DevTools 中调试 JavaScript 的基本工作流程。虽然教程介绍的是如何调试一种具体问题，但您学到的一般工作流程对调试各种类型的 JavaScript 错误均有帮助。

如果您使用 `console.log()` 来查找和修正代码中的错误，可以考虑改用本教程介绍的工作流程。其速度快得多，也更有效。

## 第 1 步：重现错误

重现错误始终是调试的第一步。“重现错误”是指找到一系列总是能导致错误出现的操作。

您可能需要多次重现错误，因此要尽量避免任何多余的步骤。

请按照以下说明重现您要在本教程中修正的错误。

1. 点击 **Open Demo**。演示页面在新标签中打开。

**OPEN DEMO** (<https://googlechrome.github.io/devtools-samples/debug-js/get-started>)

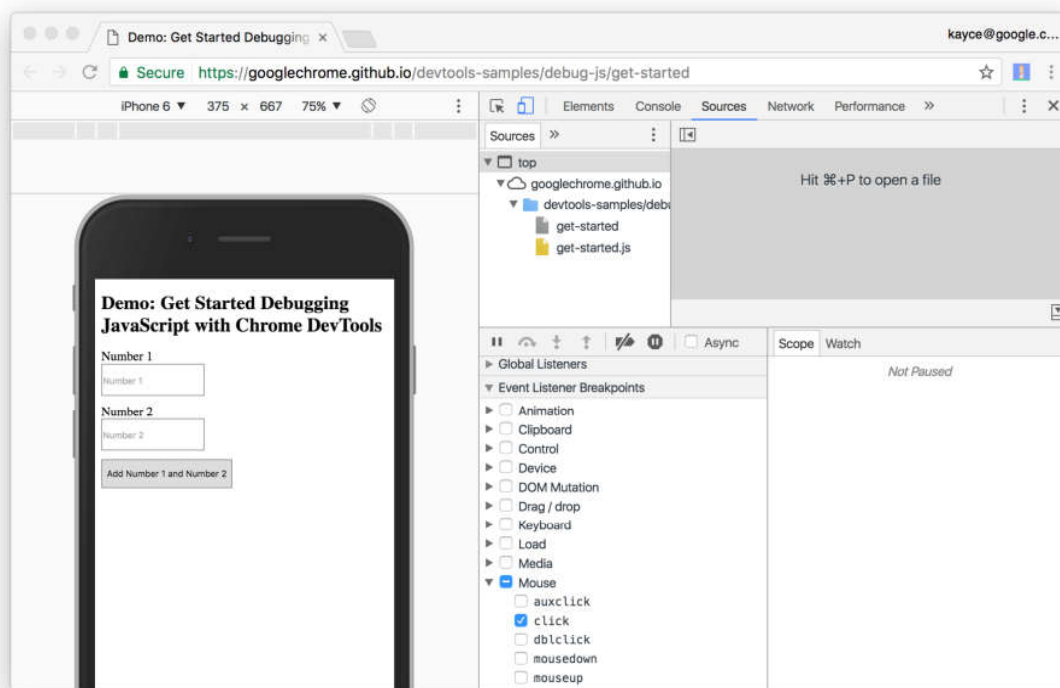
2. 在演示页面上，输入 5 作为 **Number 1**。
3. 输入 1 作为 **Number 2**。
4. 点击 **Add Number 1 and Number 2**。
5. 查看输入和按钮下方的标签。上面显示的是  $5 + 1 = 51$ 。

啊呜。这个结果是错误的。正确结果应为 6。这就是您要修正的错误。

## 第 2 步：使用断点暂停代码

DevTools 让您可以暂停执行中的代码，并对暂停时刻的**所有**变量值进行检查。用于暂停代码的工具称为**断点**。立即试一试：

1. 按 **Command+Option+I** (Mac) 或 **Ctrl+Shift+I** (Windows、Linux) 在演示页面上打开 DevTools。
2. 点击 **Sources** 标签。
1. 点击 **Event Listener Breakpoints** 将该部分展开。DevTools 显示一个包含 **Animation** 和 **Clipboard** 等可展开事件类别的列表。
1. 在 **Mouse** 事件类别旁，点击 **Expand** ►。DevTools 显示一个包含 **click** 等 Mouse 事件的列表，事件旁有相应的复选框。
2. 选中 **click** 复选框。



**图 1：** DevTools 在演示页面上打开，Sources 面板获得焦点，click 事件侦听器断点处于启用状态。如果 DevTools 窗口较大，则 **Event Listener Breakpoints** 窗格位于右侧，而不是像屏幕截图中那样位于左下方。

3. 返回至演示页面，再次点击 **Add Number 1 and Number 2**。DevTools 暂停演示并在 **Sources** 面板中突出显示一行代码。DevTools 突出显示的是下面这行代码：

```
function onClick() {
```

当您选中 **click** 复选框时，就是在所有 **click** 事件上设置了一个基于事件的断点。点击了**任何**节点，并且该节点具有 **click** 处理程序时，DevTools 会自动暂停在该节点 **click** 处理程序的第一行。


注：这不过是 DevTools 提供的众多断点类型中的一种。应使用的断点类型取决于您要调试的问题类型。

## 第 3 步：单步调试代码

一个常见的错误原因是脚本执行顺序有误。可以通过单步调试代码一次一行地检查代码执行情况，准确找到执行顺序异常之处。立即试一试：

1. 在 DevTools 的 **Sources** 面板上，点击 **Step into next function call** ，一次一行地单步调试 `onClick()` 函数的执行。DevTools 突出显示下面这行代码：

```
if (inputsAreEmpty()) {
```

2. 点击 **Step over next function call** 。DevTools 执行 `inputsAreEmpty()` 但不进入它。请注意 DevTools 是如何跳过几行代码的。这是因为 `inputsAreEmpty()` 求值结果为 `false`，所以 `if` 语句的代码块未执行。


这就是单步调试代码的基本思路。如果您看一下 `get-started.js` 中的代码，就能发现错误多半出在 `updateLabel1()` 函数的某处。您可以不必单步调试每一行代码，而是使用另一种断点在靠近错误位置的地方暂停代码。

## 第 4 步：设置另一个断点

代码行断点是最常见的断点类型。如果您想在执行到某一行代码时暂停，请使用代码行断点。立即试一试：

1. 看一下 `updateLabel1()` 中的最后一行代码，其内容类似于：

```
label.textContent = addend1 + ' + ' + addend2 + ' = ' + sum;
```

2. 在这行代码的左侧，可以看到这行代码的行号：**32**。点击 **32**。DevTools 会在 **32** 上放置一个蓝色图标。这意味着这行代码上有一个代码行断点。DevTools 现在总是会在执行这行代码之前暂停。
3. 点击 **Resume script execution** 。脚本继续执行，直至到达您设置了断点的代码行。
4. 看一下 `updateLabel1()` 中已执行的代码行。


DevTools 打印输出 `addend1`、`addend2` 和 `sum` 的值。

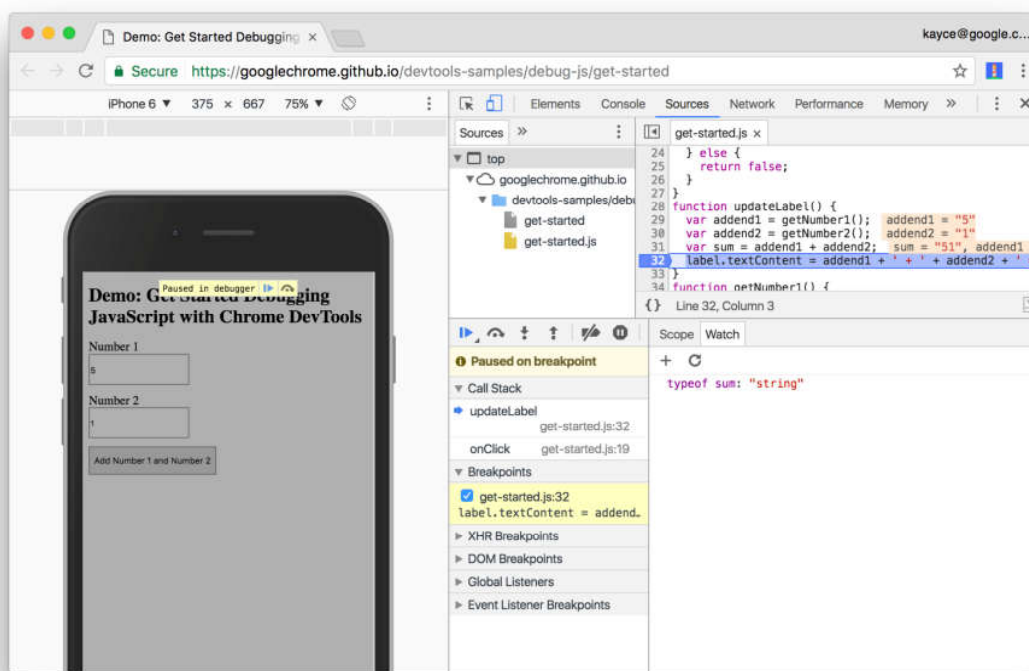
`sum` 的值疑似有问题。其求值结果本应是数字，而实际结果却是字符串。这可能就是造成错误的原因。

## 第 5 步：检查变量值

另一种常见的错误原因是，变量或函数产生的值异常。许多开发者都利用 `console.log()` 来了解值随时间变化的情况，但 `console.log()` 可能单调乏味而又效率低下，原因有两个。其一，您可能需要手动编辑大量调用 `console.log()` 的代码。其二，由于您不一定知晓究竟哪一个变量与错误有关，因此可能需要对许多变量进行记录。

DevTools 为 `console.log()` 提供的其中一个替代工具是监视表达式。可以使用监视表达式来监视变量值随时间变化的情况。顾名思义，监视表达式的监视对象不仅限于变量。您可以将任何有效的 JavaScript 表达式存储在监视表达式中。立即试一试：

1. 在 DevTools 的 **Sources** 面板上，点击 **Watch**。该部分随即展开。
2. 点击 **Add Expression** .
3. 键入 `typeof sum`。
4. 按 Enter。DevTools 显示 `typeof sum: "string"`。冒号右侧的值就是监视表达式的结果。



**图 1：**创建 `typeof sum` 监视表达式后的“监视表达式”窗格（右下方）。如果 DevTools 窗口较大，则“监视表达式”窗格位于右侧，**Event Listener Breakpoints** 窗格的上方。

正如猜想的那样，`sum` 的求值结果本应是数字，而实际结果却是字符串。这就是演示页面错误的原因。

DevTools 为 `console.log()` 提供的另一个替代工具是 Console。可以使用 Console 对任意 JavaScript 语句求值。开发者通常利用 Console 在调试时覆盖变量值。在您所处的情况下, Console 可帮助您测试刚发现的错误的潜在解决方法。立即试一试:

1. 如果您尚未打开 Console 抽屉, 请按 `Escape` 将其打开。它会在 DevTools 窗口底部打开。
2. 在 Console 中, 键入 `parseInt(addend1) + parseInt(addend2)`。
3. 按 `Enter`。DevTools 对语句求值并打印输出 6, 即您预料演示页面会产生的结果。

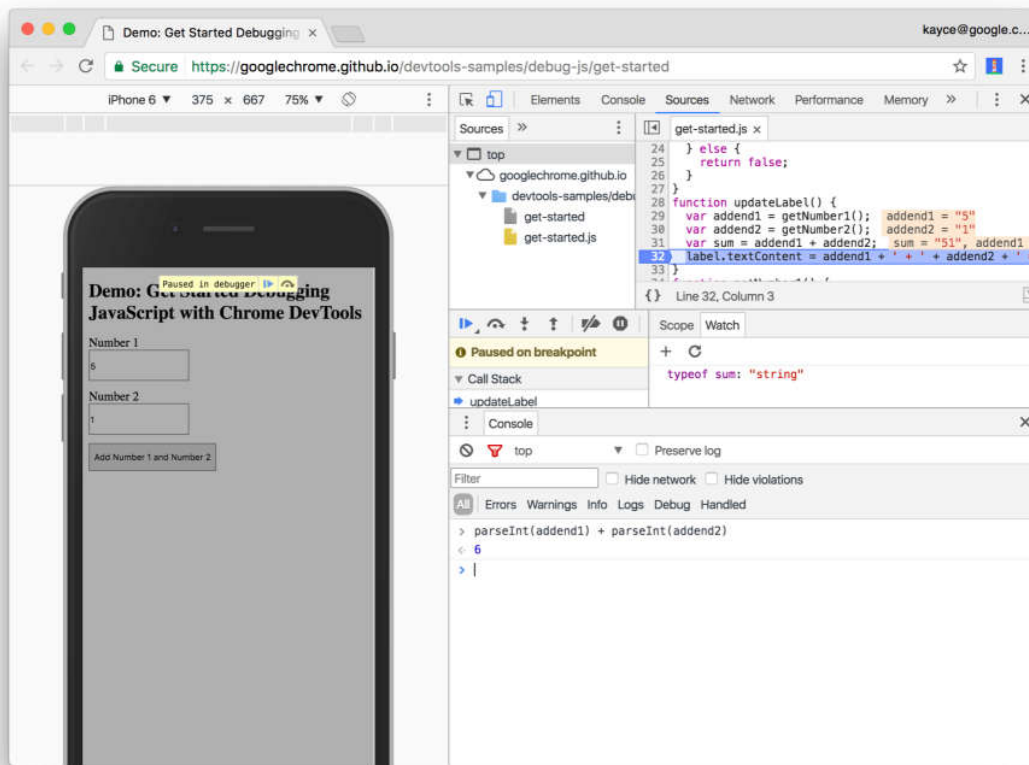




图 1: 对 `parseInt(addend1) + parseInt(addend2)` 求值后的 Console 抽屉。

## 第 6 步: 应用修正

您已找到错误的潜在解决方法。剩下的工作就是编辑代码后重新运行演示页面来测试修正效果。您不必离开 DevTools 就能应用修正。您可以直接在 DevTools UI 内编辑 JavaScript 代码。立即试一试:

1. 在 DevTools 的 **Sources** 面板上的代码编辑器中, 将 `var sum = addend1 + addend2` 替换为 `var sum = parseInt(addend1) + parseInt(addend2);`。它就是您当前暂停位置上面那行代码。

2. 按 **Command+S** (Mac) 或 **Ctrl+S** (Windows、Linux) 保存更改。代码的背景色变为红色，这表示在 DevTools 内更改了脚本。
3. 点击 **Deactivate breakpoints** 。它变为蓝色，表示处于活动状态。如果进行了此设置，DevTools 会忽略您已设置的任何断点。
4. 点击 **Resume script execution** 。
5. 使用不同的值测试演示页面。现在演示页面应能正确计算求和。

切记，此工作流程只对运行在浏览器中的代码应用修正。它不会为所有运行您的页面的用户修正代码。要实现该目的，您需要修正运行在提供页面的服务器上的代码。

## 后续步骤

恭喜！现在您已掌握了在 DevTools 中调试 JavaScript 的基础知识。

本教程只向您介绍了两种设置断点的方法。DevTools 还提供了许多其他方法，其中包括：

- 仅在满足您指定的条件时触发的条件断点。
- 发生已捕获或未捕获异常时触发的断点。
- 当请求的网址与您提供的子字符串匹配时触发的 XHR 断点。

[为我演示所有断点](https://developers.google.com/web/tools/chrome-devtools/javascript/add-breakpoints?hl=zh-cn)

(<https://developers.google.com/web/tools/chrome-devtools/javascript/add-breakpoints?hl=zh-cn>)

有几个代码单步执行控件在本教程中未予说明。请点击以下链接，了解有关它们的更多信息。

[我想要掌握代码单步调试知识](https://developers.google.com/web/tools/chrome-devtools/javascript/step-code?hl=zh-cn#stepping_in_action)

([https://developers.google.com/web/tools/chrome-devtools/javascript/step-code?hl=zh-cn#stepping\\_in\\_action](https://developers.google.com/web/tools/chrome-devtools/javascript/step-code?hl=zh-cn#stepping_in_action))

## 反馈

请通过回答下列问题帮助我们改进本教程。



下面的内容，我们重点了解一下如何添加断点。在 Tbrowser Remote Devtools 中支持以下七种断点：

Line-of-code	在代码中的确切位置添加断点
Conditional line-of-code	在代码中的确切位置添加条件断点，只有条件表达式运算结果为 <b>true</b> 才会生效
DOM	当更改或者移除特定的 DOM 节点或者其子节点时触发的断点
XHR	当一个 XHR 请求的 URL 包含指定的字符串时触发的断点
Event listener	当一个事件（如 <b>keydown</b> 、 <b>click</b> ）发生时触发的断点
Exception	当捕获到异常信息时发出的断点
Function	当特定函数执行时触发的断点

### 3.3.1 Line-of-code Breakpoints

确切断点一般用于知道要调试代码的确切位置时使用。在代码中的某一行添加断点后，当执行到这一行之前 JavaScript 代码的执行就会被暂停。

添加确切断点的方法有两种：

- 在 JavaScript 代码的左侧行号位置鼠标左键单击。行号背景色会变为蓝色，断点就添加成功了。
- 在代码中调用 **debugger**。调用 **debugger** 后，正常打开页面不会暂停代码执行。需要在 Devtool 打开的情况下，执行到 **debugger** 语句时就会暂停执行不需手动添加断点。

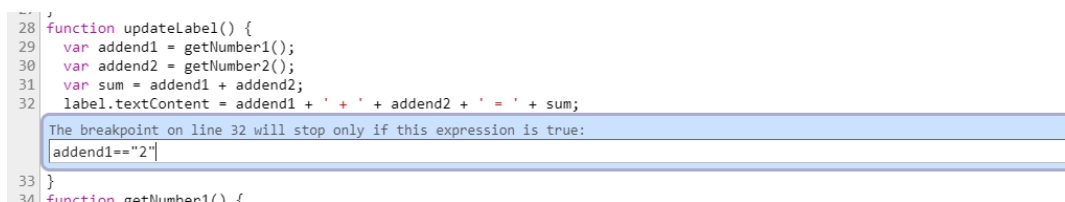


图 3.8: 添加条件断点

```
28 function updateLabel() {
29   var addend1 = getNumber1();
30   var addend2 = getNumber2();
31   var sum = addend1 + addend2;
32   label.textContent = addend1 + ' + ' + addend2 + ' = ' + sum;
33 }
34 function getNumber1() {
```

图 3.9: 添加断点成功

如果想要只在某种条件下才出发断点，可以使用条件断点。添加条件断点的步骤是：



1. 在要添加条件断点的行号处右键，选择 **Add conditional breakpoint...**。
2. 在条件编辑框内输入触发的条件。输入 **Tab** 键或者鼠标点击其他区域完成编辑。
3. 此时对应行号的背景色会变为橙色，表明条件断点已经添加成功。
4. 在条件断点处右键可以编辑条件和删除断点。

在右侧工具栏中的 **Breakpoints** 标签中可以查看和管理所有的确定断点。其中常用的操作有：

- 停用/启动断点。勾选断点前的复选框可以停用/启用单个断点；右键菜单中可以停用/启用所有断点。
- 删除断点。右键菜单中可以删除单个或者全部断点；单击已经添加的断点的行号也可以删除断点。
- 定位断点所在位置。单击单个断点可跳转断点在代码中的位置。

### 3.3.2 DOM Breakpoints

Dom 断点是当页面指定的节点或者其子节点被删除或更改时触发的断点。代码将暂停在修改指定节点的 JavaScript 代码中。添加 Dom 断点的步骤是：

1. 打开 **Elements** 面板，选择要检测的元素。
2. 在右键菜单中选择 **Break on...**。
3. 再在子菜单中选择添加断点的类型。
4. 添加成功后会在 Dom 树的左侧有一个蓝色的小点标记。

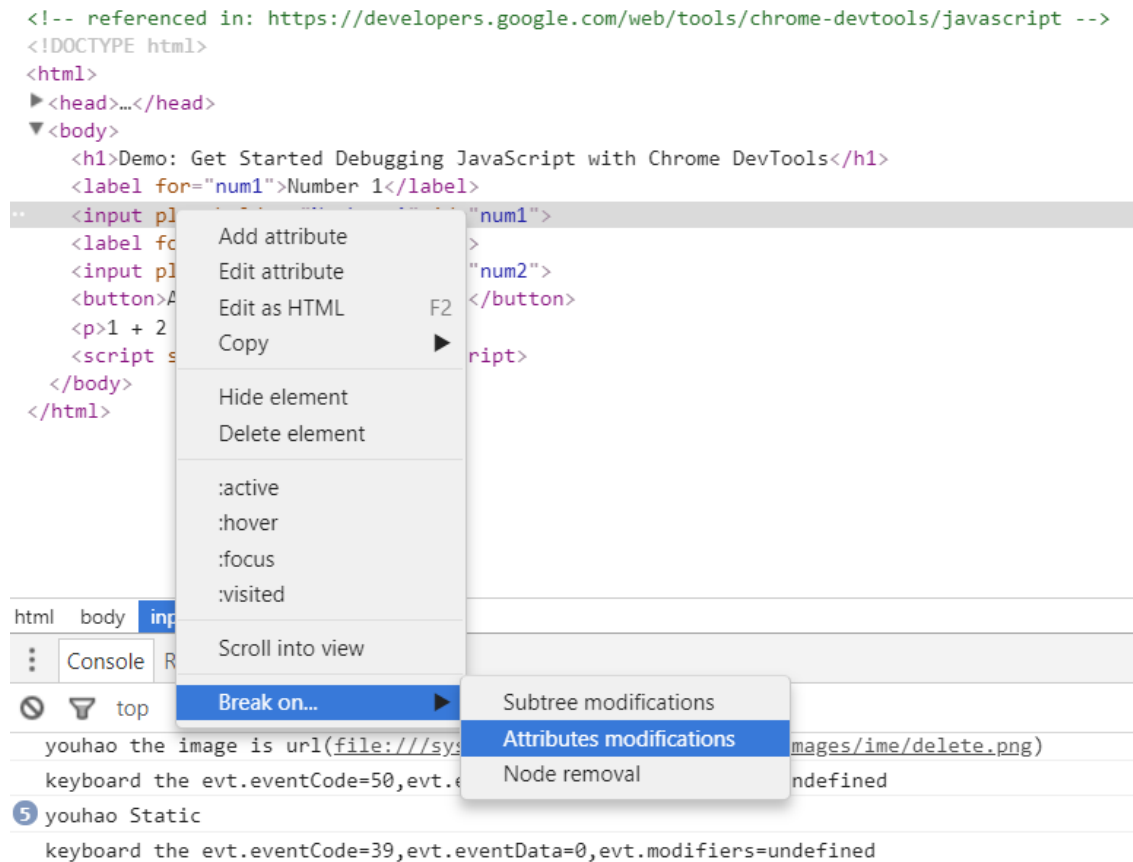


图 3.10: 添加 Dom 断点的过程

Devtools 支持三种 Dom 断点:

- **Subtree modifications:** 当选中的节点的子节点被删除或添加, 或者子节点的内容发生改变时触发断点。当子节点属性或选中节点属性和内容发生变化时不会触发断点。
- **Attributes modifications:** 当选中节点属性被添加、删除或者属性值发生变化时触发断点。
- **Node Removal:** 当选中的节点被删除时触发断点。

### **3.3.3 XHR Breakpoints**

## **3.4 Network 模块**

## **3.5 Timeline 模块**

## **3.6 Profiles 模块**

## **3.7 Resource 模块**

## **3.8 Security 模块**

## **3.9 Audits 模块**