Web前端性能优化

1. **前端性能优化目的**
2. 从用户角度而言，优化能够让页面加载得更快、对用户的操作响应得更及时，能够给用户提供更为友好的体验。
3. 从服务商角度而言，优化能够减少页面请求数、或者减小请求所占带宽，能够节省可观的资源。

总之，恰当的优化不仅能够改善站点的用户体验并且能够节省相当的资源利用。

1. **前端性能分析工具**

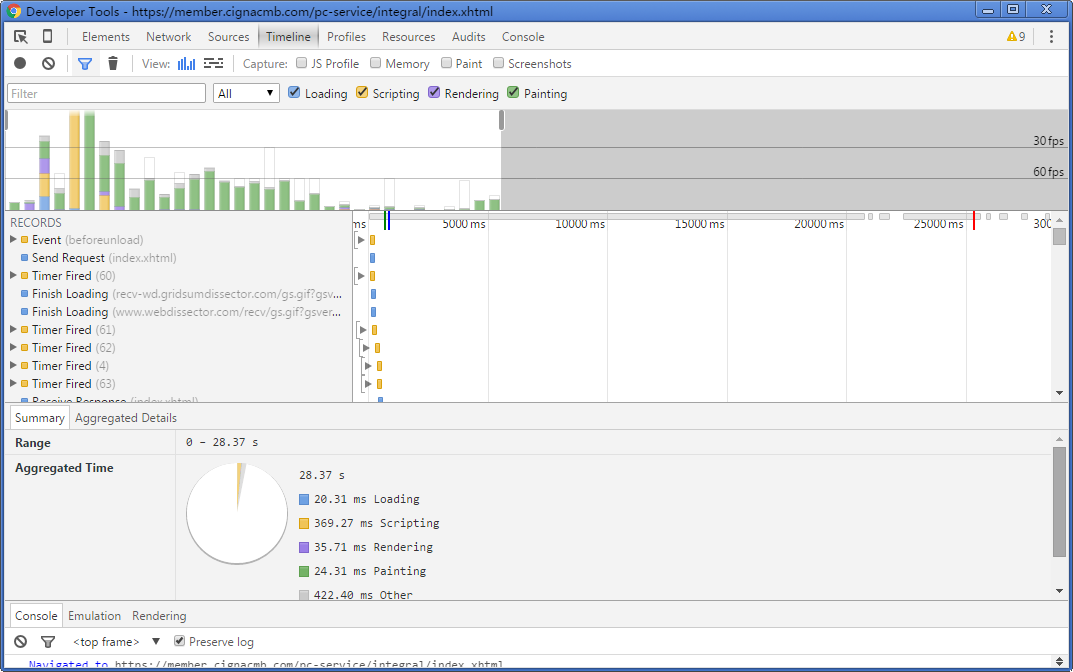
随着webpage可以承载的表现形式更加多样化，通过webpage来实现更多交互功能，构建web应用程序已经成为很多产品的首要选择。这种方式拥有非常明显的优势：跨平台、开发便捷、便于部署和维护等等，但随着功能的不断积累，web应用程序也会变得越来越复杂。但是，想要在webpage支持丰富的呈现形式的同时，让页面效果能够达到>=60fps(帧)/s的刷新频率以避免出现卡顿，就需要使用一些比较直观的方式来分析衡量页面的性能问题，为性能优化方案提供依据。

为什么是60fps?目标是保证页面要有高于每秒60fps(帧)的刷新频率，这和目前大多数显示器的刷新率相吻合(60Hz)。如果网页动画能够做到每秒60帧，就会跟显示器同步刷新，达到最佳的视觉效果。这意味着，一秒之内进行60次重新渲染，每次重新渲染的时间不能超过16.66毫秒。

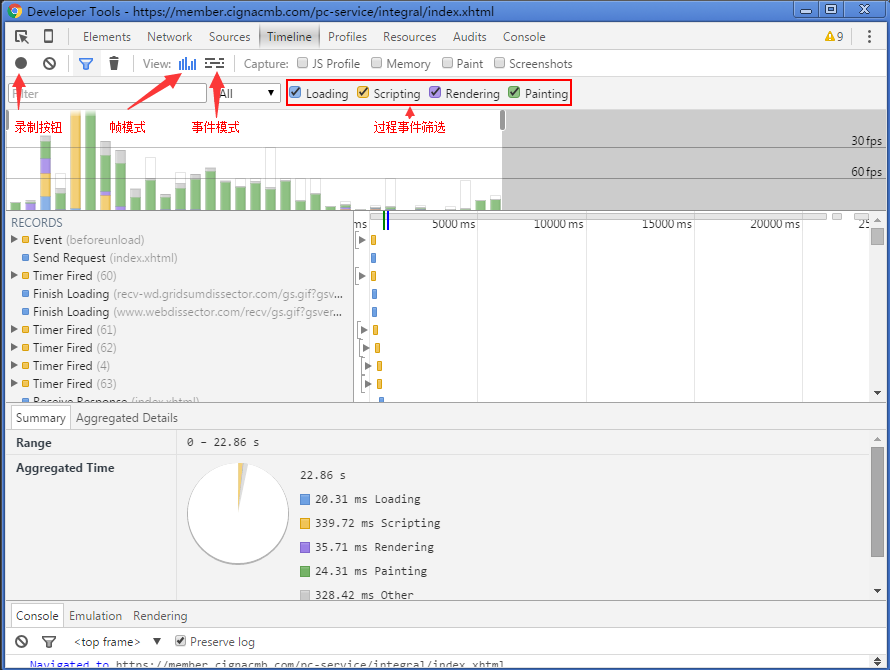
1. **Chrome DevTools的Timeline**

Chrome DevTools的Timeline是用来记录和分析应用在运行时所有的活动情况，它是用来排查应用性能瓶颈的最佳工具。

Timeline面板的预览效果：



**Timeline工具栏介绍**

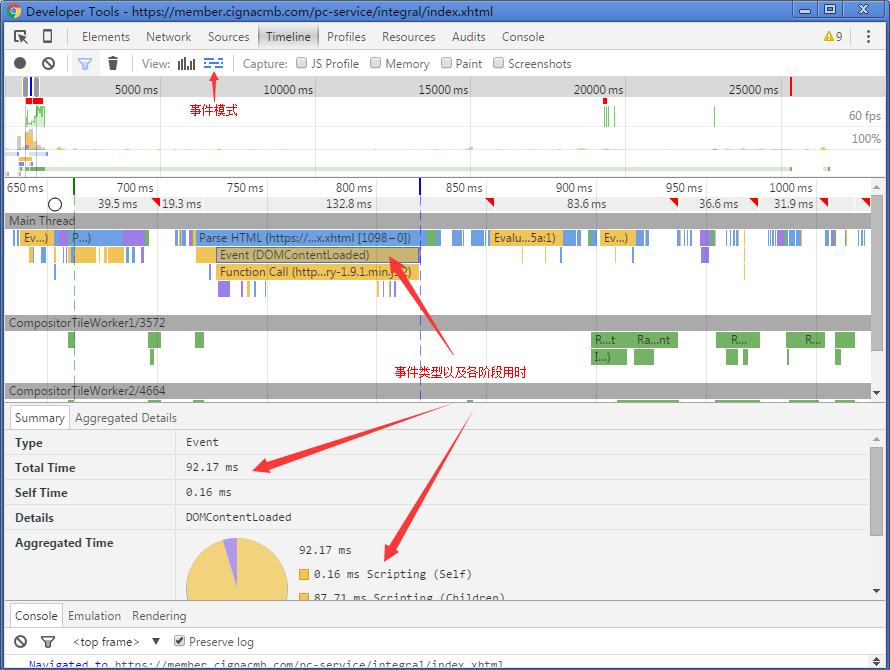
Timeline工具会详细检测出在Web应用加载的过程中时间花费情况的概览，包括下载资源、处理DOM事件、页面布局渲染、向屏幕绘制元素等。可以通过分析Timeline得到的事件、框架和实时的内存用量，找出应用的性能问题。

在分析页面前，需要首先开启录制功能，记录页面的操作和渲染记录。如上图，左上角的灰色圆点就是录制按钮，点击后会变成红色，然后在页面上进行相关操作后再次按下变成灰色完成录制，这样就完成了一次对操作及加载渲染的记录过程，随后Timeline就会开始分析操作过程中的各项性能参数。

Timeline同时提供了两种查看模式：“事件模式(Event Mode)”和“帧模式(Frame Mode)”。如上图箭头所示。

事件模式：显示重新渲染的各种事件花费的时间。帧模式：显示每一帧的时间花费情况。

**事件模式 (Event Mode)**

如果一个页面的执行效率不高，就要搞清楚导致页面性能低下的原因，到底是javascript执行出了问题，还是页面渲染出了问题。要了解这里面的执行细节，可以使用“事件模式”来进行分析。首先录制一些需要被分析的操作，录制结束后进入事件模式预览Timeline。下图是得到的事件模式的视图： 

在上图中，不同的颜色表示不同的事件。一种颜色的区块越长，说明在处理该事件的耗时就越长。单击某一区块，可以在下面的Summary概要中看到详细的事件处理过程及耗时分布。



蓝色(Loading)：网络通信和HTML解析

黄色(Scripting)：JavaScript执行

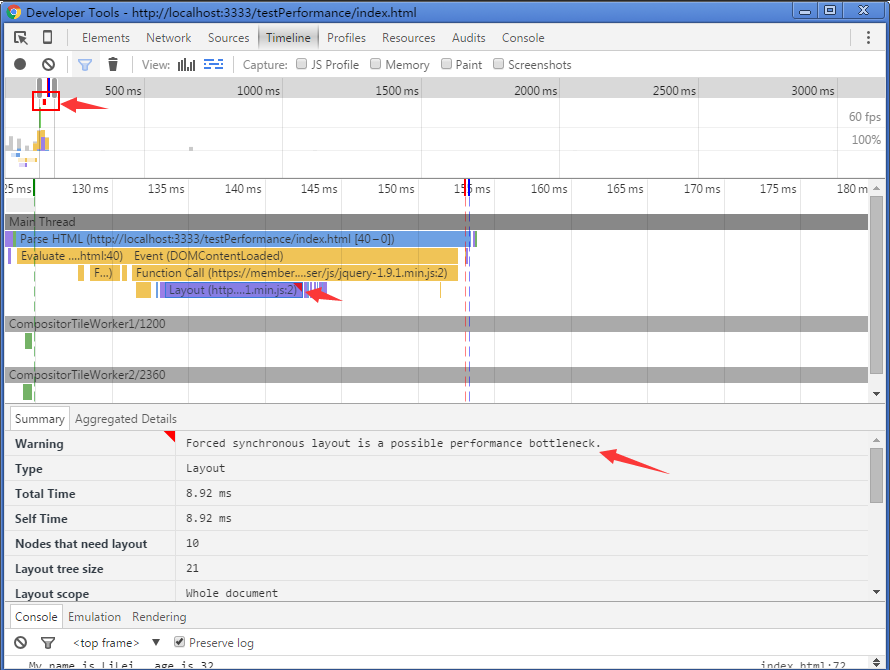
紫色(Rendering)：样式计算和布局，即重排

绿色(Painting)：重绘

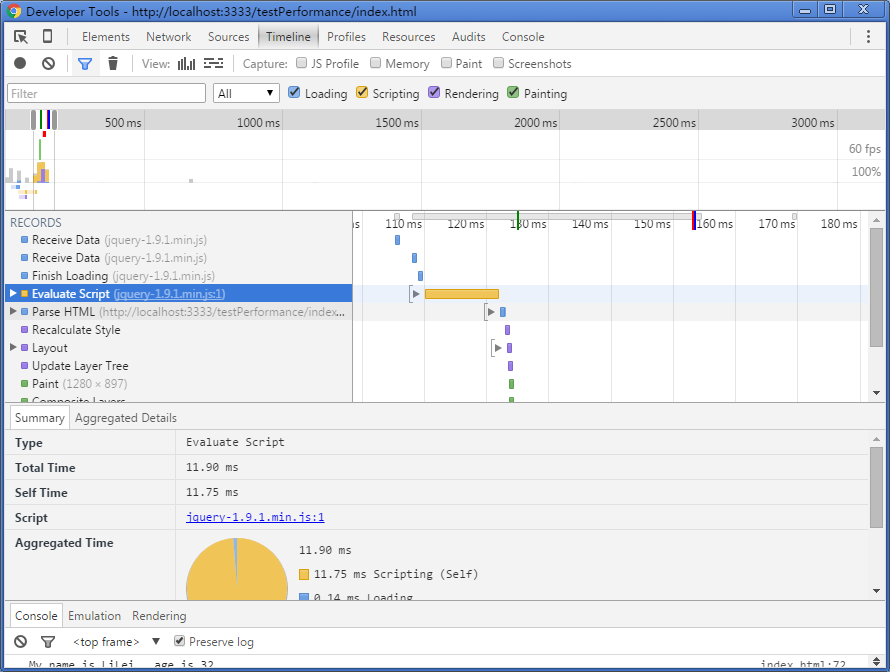
灰色(other)：其它事件花费的时间

白色(Idle)：空闲时间

在显示的记录中，浏览器也会为在检测过程中发现的一些可能导致性能问题的过程进行标注，在Mode View视图区域，可能会出现一些红色的区块段，这些红色的区块段表明，在对应的时间上执行的事件可能存在性能问题，而在对应的Main Thread视图区域，事件区块的右上角会出现红色的小三角，点击当前区块，在下面的Summary概要区域内会给出详细的警告内容以及脚本可能出现问题的行数，如下图，浏览器提示“强制同步布局可能会导致性能瓶颈”：



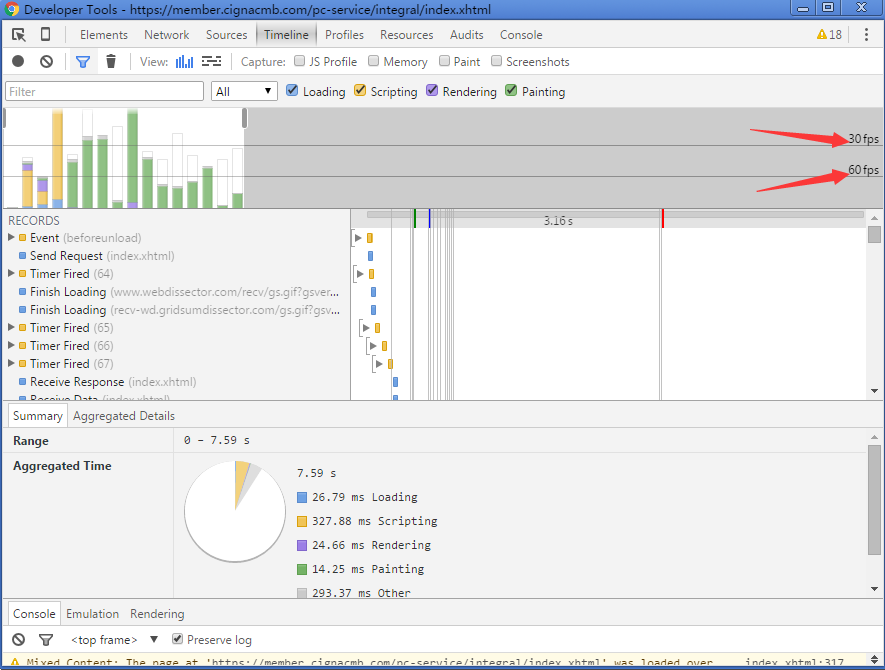
此外，在关闭Event Mode后，还可以看到Record Detail视图，详细列出一次记录中各类事件的详细内容。



Record Detail视图区域的左侧是事件标题，右侧是对应的时间线。点击每一条时间标题可以看到更多信息，如事件发生在脚本的哪一行等。

**帧模式 (Frame Mode)**

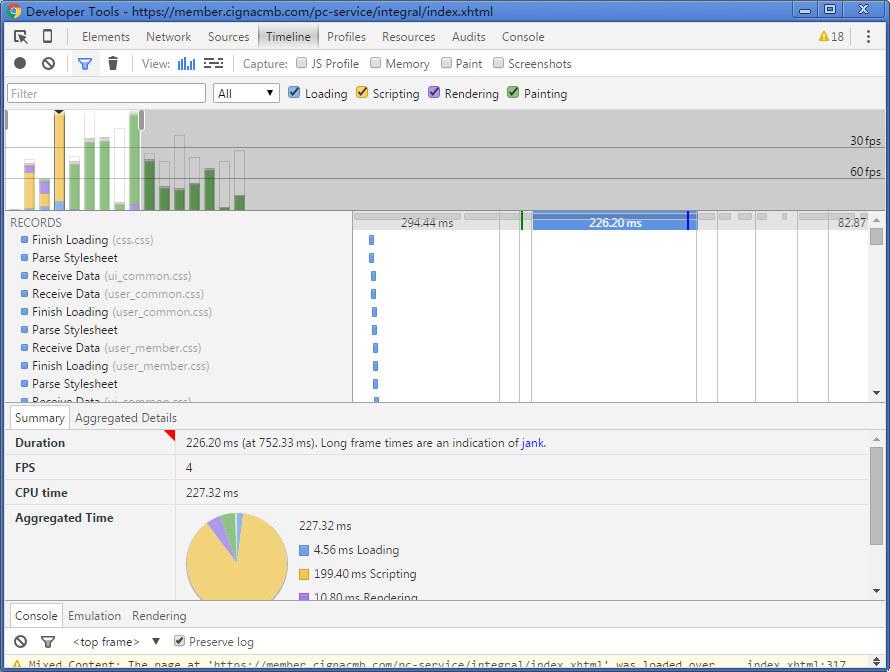
帧模式从页面渲染性能的角度提供了数据支撑，一个柱状“frame”表示渲染过程中的一帧，也就是浏览器为了渲染单个内容块而必须要做的工作，包括：执行js，处理事件，修改DOM，更改样式和布局，绘制页面等。

为保证页面有高流畅度的渲染，就要求页面要有高于每秒60fps(帧)的刷新频率。

图中在Frame视图中有两条贯穿该视图的横线，分别标识出60FPS和30FPS的基准，按照16.66ms的计算方式，可以理解为分别标识了16.6ms和33.3ms两个时间点。下面的一条是60FPS，低于这条线，可以达到每秒60帧；上面的一条是30FPS，低于这条线，可以达到每秒30次渲染。如果色柱都超过30FPS，这个网页就有性能问题了。

图中帧柱的高度表示了该帧的总耗时，帧柱中的颜色分别对应该帧中包含的不停类型的事件。每一帧柱的高度越低越好。对相关的帧进行分析时，可以点击其中某一帧查看渲染详情，也可以选择某个区域的几个帧查看渲染详情。而要找出可能影响性能的原因，点击当前问题帧，在Summary面板及Record Detail视图中的详细信息中进行逐条分析。

点击某一个帧柱还可以得到该帧的详细记录数据：



Warning: 警告信息

ScreenShot: 当前选中帧的渲染截屏

Duration: 该记录及其子记录的总耗时

FPS: 当前帧的渲染频率

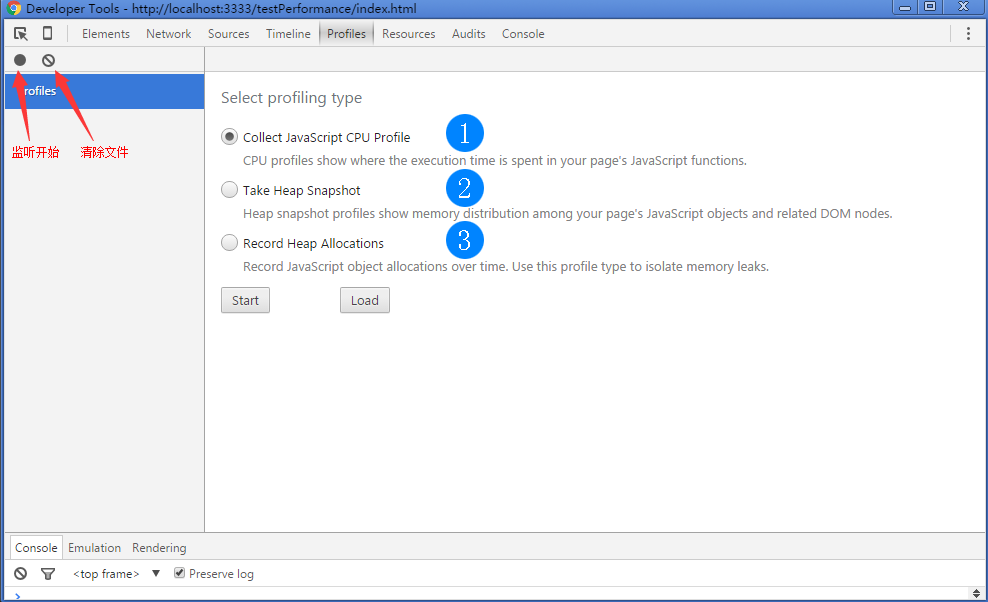
CPU Time: CPU耗时

Aggregated Time: 合计耗时分布

1. **Chrome DevTools的Profiles**

Profiles面板功能的作用主要是监控网页中各种方法执行时间和内存的变化，简单来说它就是Timeline的数字化版本。

Profiles面板

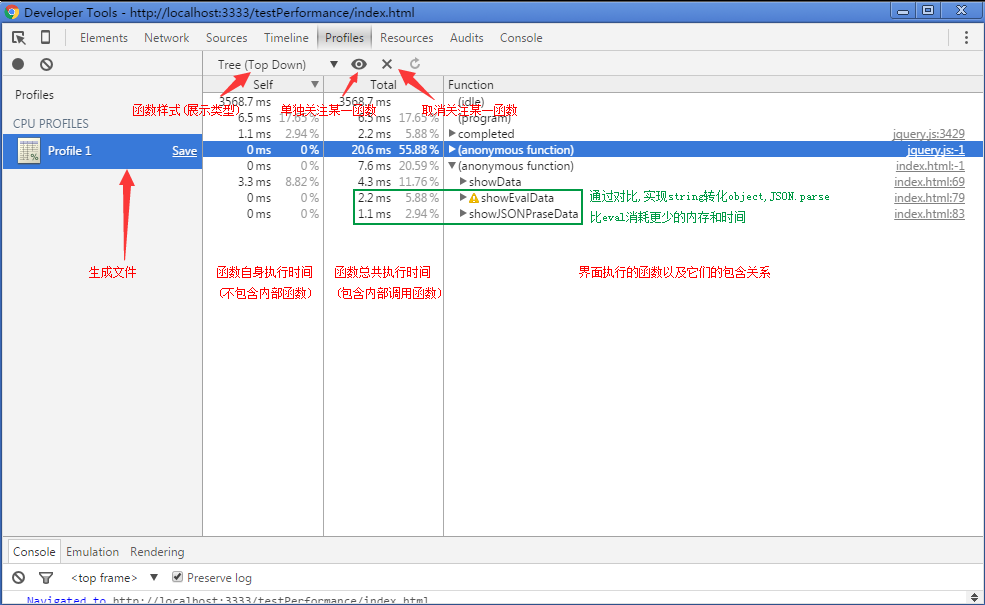


Profiles界面分为左右两个区域，左边区域是放文件的区域，右边是展示数据的区域。在开始检测之前可以看到右边区域有三个选项，它们分别代表者不同的功能:

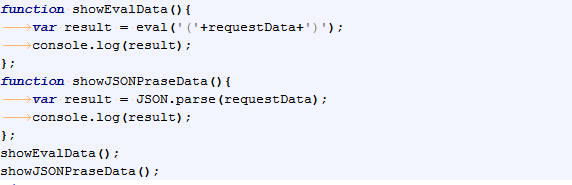
* + 1. **(Collect JavaScript CPU Profile)监控函数执行期花费的时间**
    2. **(Take Heap Snapshot)为当前界面拍一个内存快照**
    3. **(Record Heap Allocations)实时监控记录内存变化(对象分配跟踪)**

**Collect JavaScript CPU Profile（函数收集器）**

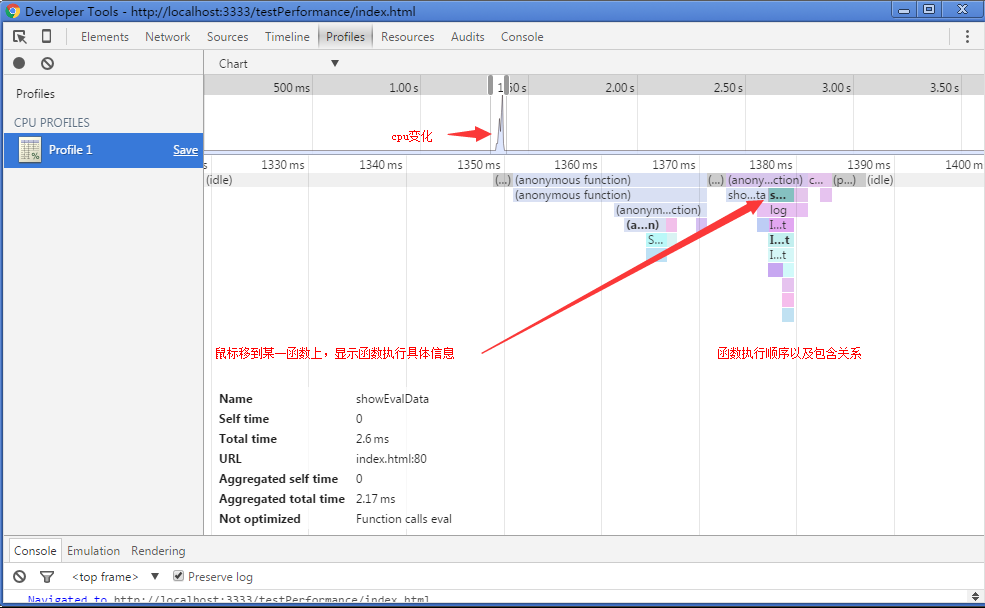
在右边区域中选择Collect JavaScript CPU Profile 选项，点击下方的Start按钮（也可以点击左边的黑色圆圈），这时候Chrome会开始记录网页的方法执行，然后我们点击界面的按钮来执行函数。最后再点击右边区域的Stop按钮（或者左边的红色圆圈），这时监控就结束了。左边Profiles会列出一个文件，单击可以看到如下界面：



以上，脚本运行了showEvalData和showJSONPraseData两个方法，通过对比，实现string转换object，JSON.parse比eval性能更好



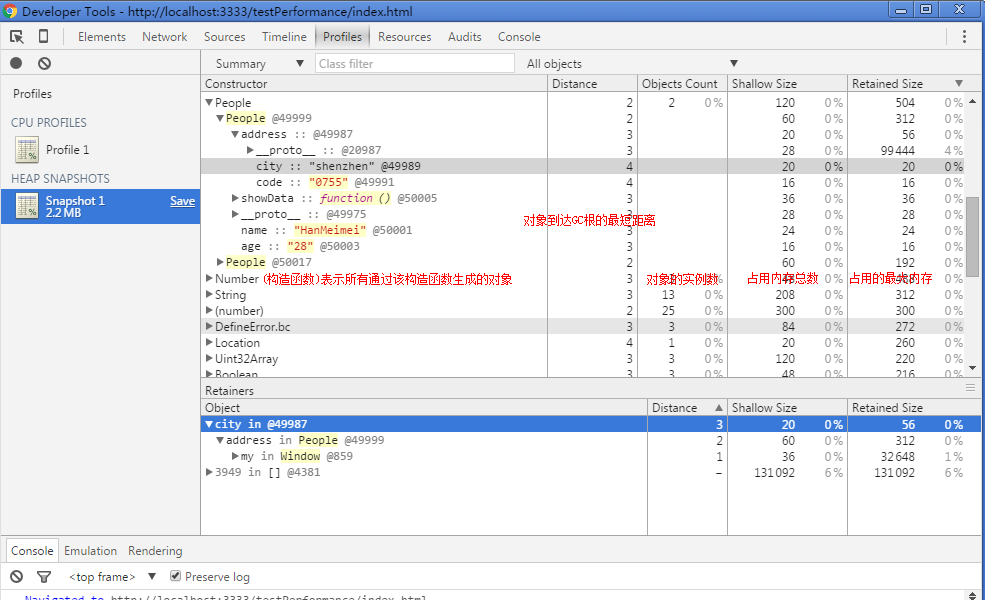
通过右边区域的类型选项可以切换数据显示的方式。有正包含关系，逆包含关系，图表类型三种选项。我们可以选择其中的图表类型：



从上图可以看到点击按钮执行的各个函数执行的时间，顺序，包含关系和CUP变化等。

**Take Heap Snapshot（内存快照）**

第二个功能是给当前网页拍一个内存快照.选择第二个拍照功能，按下 Take Snapshot 按钮，给当前的网页拍下一个内存快照，得到如下图。



可以看到左边区域生成个文件，文件名下方有数字，表示这个张快照记录到的内存大小（此时为3.2M）。右边区域是个列表，它分为五列，表头可以按照数值大小手动排序。从左到右的顺序它们分别表示:

**Constructor**(构造函数)表示所有通过该构造函数生成的对象

**Distance** 对象到达GC根的最短距离

**Objects Count** 对象的实例数

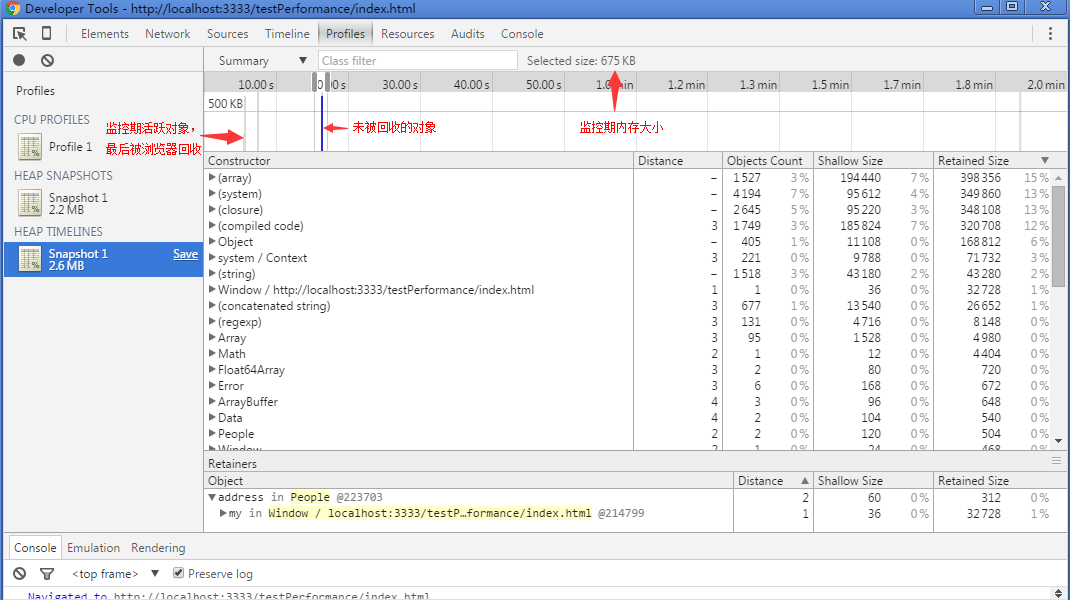
**Shallow size** 对应构造函数生成的对象的shallow sizes(直接占用内存)总数

**Retained size** 展示了对应对象所占用的最大内存

个人觉得内存快照的作用为根据对象占用内存的情况，检查代码有没有适当对没用的对象进行内存释放。

**Record Heap Allocations.（对象跟踪器）**

它的作用是为为我们拍下一系列的快照（频率为50ms），为我们检测在启用它的时候每个对象的生存情况。



1. **性能优化方法**

前端优化的途径有很多，按粒度大致可以分为两类，第一类是页面级别的优化，例如 HTTP请求数、脚本的无阻塞加载、内联脚本的位置优化等 ;第二类则是代码级别的优化，例如 Javascript中的DOM 操作优化、CSS选择符优化、图片优化以及 HTML结构优化等等。

1. **HTML部分**
   1. **减少HTTP请求**
      1. 把多个JS请求合并为一个JS请求，把多个CSS请求合并为一个CSS请求
      2. 配置ETag

在http请求中加入ETag标记，这个标记可以在请求时检验服务器端资源是否与客户端资源相同。如果相同，只返回带304响应码的响应头，不用重新从服务器端发送页面数据，而直接使用本地缓存。

* + 1. 添加Expires头

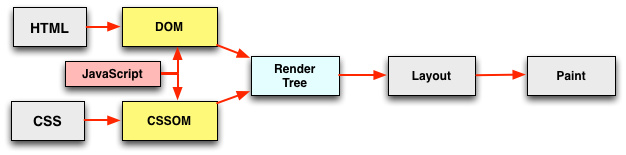
响应头中有Expires头，设置默认过期时间30分钟。即是如果当前网页未过期，浏览器不会发请求，直接查找本地页面缓存。加速前端响应速度，减少服务器端压力。

* + 1. 压缩组件

查看http请求头参数中，有一项Accept-Encoding:gzip,deflate，响应头中有一项Content-Encoding：gzip ，这里表示请求的内容采用gzip方式进行压缩传输。减少传输文件的大小，加快页面响应

* 1. **将样式表放在顶部**

如果样式文件放在页面底部可能会出现白屏或者无样式内容的闪烁



* 1. **将脚本放在底部**

浏览器的加载顺序基本是按源码从上到下加载的，把样式表放在顶部，可以加快页面样式的显示，给客户更好的体验。同理，把脚本放在底部，会避免因脚本阻塞页面内容的呈现

或者采用以下两种方法：

在<script></script>标签中使用 async或defer特性。

1. <script async>将会在HTML解析时下载该文件并在下载完成后马上执行。
2. <script defer> 将会在HTML解析式下载该文件并在HTML解析完成后执行。
   1. **使用内容分发网络(CDN)**

把网站内容分散到多个、处于不同地域位置的服务器上可以加快下载速度。

其基本思路是尽可能避开互联网上有可能影响数据传输速度和稳定性的瓶颈和环节，使内容传输的更快、更稳定。通过在网络各处放置节点服务器所构成的在现有的互联网基础之上的一层智能虚拟网络，CDN系统能够实时地根据网络流量和各节点的连接、负载状况以及到用户的距离和响应时间等综合信息将用户的请求重新导向离用户最近的服务节点上。

* 1. **尽量减少重定向**

有时为了特定需求，需要在网页中使用重定向。重定向的意思是，用户的原始请求（例如请求A）被重定向到其他的请求（例如请求B）。

但是这会造成网站性能和速度下降，因为浏览器访问网址是一连串的过程，如果访问到一半而跳到新地址，就会重复发起一连串的过程，这将浪费很多的时间。

1. **JS部分**
2. **精简Javascript**

对JS文件进行压缩，减少了JS的体积

1. **移除重复脚本**

一个HTML文件对同一个JS文件或者CSS文件，只引入一次

1. **代码方面优化**
2. 减少对DOM的操作

当解析的html文件很大时，生成DOM树占用内存较大，同时遍历（不更新）元素耗时也更长。但这都不是重点，DOM的核心问题是：DOM修改导致的页面重绘、重新排版！重新排版是用户阻塞的操作，同时，如果频繁重排，CPU使用率也会猛涨！

DOM操作会导致一系列的重绘（repaint）、重新排版（reflow）操作。为了确保执行结果的准确性，所有的修改操作是按顺序同步执行的。大部分浏览器都不会在JavaScript的执行过程中更新DOM。相应的，这些浏览器将对对 DOM的操作放进一个队列，并在JavaScript脚本执行完毕以后按顺序一次执行完毕。也就是说，在JavaScript执行的过程，直到发生重新排版，用户一直被阻塞。

一般的浏览器中（不含IE），repaint的速度远快于reflow，所以避免reflow更重要。

导致repaint、reflow的操作：

\* DOM元素的添加、修改（内容）、删除( Reflow + Repaint)

\* 仅修改DOM元素的字体颜色（只有Repaint，因为不需要调整布局）

\* 应用新的样式或者修改任何影响元素外观的属性

\* Resize浏览器窗口、滚动页面

\* 读取元素的某些属性（offsetLeft、offsetTop、offsetHeight、offsetWidth、scrollTop/Left/Width/Height、clientTop/Left/Width/Height、getComputedStyle()、currentStyle(in IE))

1. 减少循环遍历
2. 慎用with，避免用eval和Function
3. 合理封装方法，较少代码量
4. 等等…
5. **CSS部分**
6. **利用CSS3和图标字体**
7. **优化图片大小**

包括：

1. 缩小图片分辨率
2. 改变图片格式
3. 降低图片保存质量
4. **不要在HTML中使用缩放图片**
5. **利用CSS Sprites合并图片**

减少请求数的一个好办法

1. **Inline Images**

使用 data: URL scheme的方式将图片嵌入到页面或 CSS中，如果不考虑资源管理上的问题的话，不失为一个好办法。如果是嵌入页面的话换来的是增大了页面的体积，而且无法利用浏览器缓存。使用在 CSS中的图片则更为理想一些。

1. **Lazy Load Images(图片懒加载)**

这条策略实际上并不一定能减少 HTTP请求数，但是却能在某些条件下或者页面刚加载时减少 HTTP请求数。对于图片而言，在页面刚加载的时候可以只加载第一屏，当用户继续往后滚屏的时候才加载后续的图片。这样一来，假如用户只对第一屏的内容感兴趣时，那剩余的图片请求就都节省了。