Performance --- 前端性能监控

Web前端学习营 关注

0.754 2019.06.08 16:02:17 字数 6,240 阅读 22,225

一：什么是Performance?

Performance是前端性能监控的API。它可以检测页面中的性能，W3C性能小组引入进来的一个新的API，它可以检测到白屏时间、首屏时间、用户可操作的时间节点，页面总下载的时间、DNS查询的时间、TCP链接的时间等。因此我们下面来学习下这个API。

那么在学习之前，前端性能最主要的测试点有如下几个：

白屏时间：从我们打开网站到有内容渲染出来的时间点。

首屏时间：首屏内容渲染完毕的时间节点。

用户可操作时间节点：domready触发节点。

总下载时间：window.onload的触发节点。

我们现在在html中来简单的使用下performance的基本代码：如下代码所示：

<!DOCTYPE html>performance演示varperformance =window.performance ||window.msPerformance ||window.webkitPerformance;if(performance) {console.log(performance); }

然后在浏览器下会打印如下 performance基本信息如下：

如上可以看到，performance包含三个对象，分别为 **memory**、**navigation**、**timing** . 其中memory 是和内存相关的，navigation是指来源相关的，也就是说从那个地方跳转过来的。timing是关键点时间。下面我们来分别介绍下该对象有哪些具体的属性值。

performance.memory含义是显示此刻内存占用的情况，从如上图可以看到，该对象有三个属性，分别为：

jsHeapSizeLimit该属性代表的含义是：内存大小的限制。

totalJSHeapSize表示 总内存的大小。

usedJSHeapSize表示可使用的内存的大小。

如果 usedJSHeapSize 大于 totalJSHeapSize的话，那么就会出现内存泄露的问题，因此是不允许大于该值的。

performance.navigation含义是页面的来源信息，该对象有2个属性值，分别是：redirectCount 和 type。

redirectCount：该值的含义是：如果有重定向的话，页面通过几次重定向跳转而来，默认为0；

type：该值的含义表示的页面打开的方式。默认为0. 可取值为0、1、2、255.

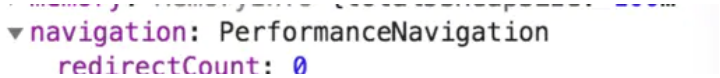
0（TYPE_NAVIGATE）：表示正常进入该页面(非刷新、非重定向)。

1（TYPE_RELOAD）：表示通过 window.location.reload 刷新的页面。如果我现在刷新下页面后，再来看该值就变成1了。

2（TYPE_BACK_FORWARD ）：表示通过浏览器的前进、后退按钮进入的页面。如果我此时先前进下页面，再后退返回到该页面后，查看打印的值，发现变成2了。

255（TYPE_RESERVED）：表示非以上的方式进入页面的。

如下图所示：



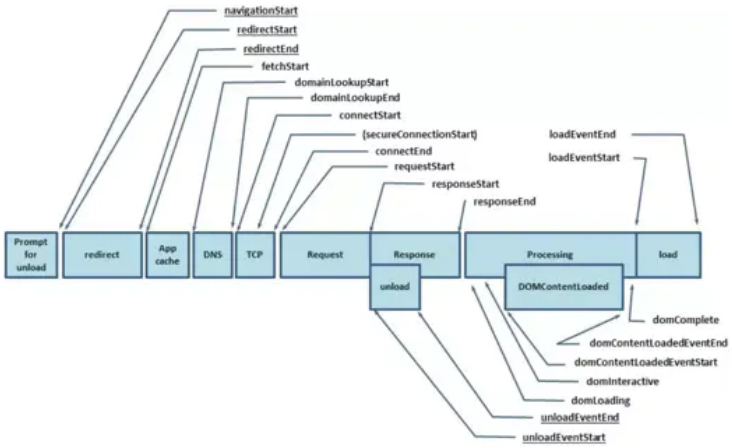
performance.onresourcetimingbufferfull; 如上截图也有这个属性的，该属性的含义是在一个回调函数。该回调函数会在浏览器的资源时间性能缓冲区满了的时候会执行的。

performance.timeOrigin：是一系列时间点的基准点，精确到万分之一毫秒。如上截图该值为：1559526951495.139，该值是

一个动态的，刷新下，该值是会发生改变的。

performance.timing：是一系列关键时间点，它包含了网络、解析等一系列的时间数据。

为了方便，从网上弄了一张图片过来，来解析下各个关键时间点的含义如下所示：



按照如上图的顺序，我们来分别看下各个字段的含义如下：

navigationStart：含义为：同一个浏览器上一个页面卸载结束时的时间戳。如果没有上一个页面的话，那么该值会和fetchStart的值相同。

redirectStart:该值的含义是第一个http重定向开始的时间戳，如果没有重定向，或者重定向到一个不同源的话，那么该值返回为0.

redirectEnd:最后一个HTTP重定向完成时的时间戳。如果没有重定向，或者重定向到一个不同的源，该值也返回为0.

fetchStart:浏览器准备好使用http请求抓取文档的时间(发生在检查本地缓存之前)。

domainLookupStart:DNS域名查询开始的时间，如果使用了本地缓存话，或 持久链接，该值则与fetchStart值相同。

domainLookupEnd:DNS域名查询完成的时间，如果使用了本地缓存话，或 持久链接，该值则与fetchStart值相同。

connectStart:HTTP 开始建立连接的时间，如果是持久链接的话，该值则和fetchStart值相同，如果在传输层发生了错误且需要重新建立连接的话，那么在这里显示的是新建立的链接开始时间。

secureConnectionStart:HTTPS 连接开始的时间，如果不是安全连接，则值为 0

connectEnd：HTTP完成建立连接的时间(完成握手)。如果是持久链接的话，该值则和fetchStart值相同，如果在传输层发生了错误且需要重新建立连接的话，那么在这里显示的是新建立的链接完成时间。

requestStart:http请求读取真实文档开始的时间，包括从本地读取缓存，链接错误重连时。

responseStart:开始接收到响应的时间(获取到第一个字节的那个时候)。包括从本地读取缓存。

responseEnd：HTTP响应全部接收完成时的时间(获取到最后一个字节)。包括从本地读取缓存。

unloadEventStart:前一个网页（和当前页面同域）unload的时间戳，如果没有前一个网页或前一个网页是不同的域的话，那么该值为0.

unloadEventEnd:和 unloadEventStart 相对应，返回是前一个网页unload事件绑定的回调函数执行完毕的时间戳。

domLoading:开始解析渲染DOM树的时间。

domInteractive:完成解析DOM树的时间（只是DOM树解析完成，但是并没有开始加载网页的资源）。

domContentLoadedEventStart：DOM解析完成后，网页内资源加载开始的时间。

domContentLoadedEventEnd:DOM解析完成后，网页内资源加载完成的时间。

domComplete:DOM树解析完成，且资源也准备就绪的时间。Document.readyState 变为 complete，并将抛出 readystatechange 相关事件。

loadEventStart:load事件发送给文档。也即load回调函数开始执行的时间，如果没有绑定load事件，则该值为0.

loadEventEnd:load事件的回调函数执行完毕的时间，如果没有绑定load事件，该值为0.

如上就是各个值的含义了，大家简单的看下，了解下就行了，不用过多的折腾。在使用这些值来计算白屏时间、首屏时间、用户可操作的时间节点，页面总下载的时间、DNS查询的时间、TCP链接的时间等之前，我们可以先看下传统方案是如何做的?

传统方案

在该API出现之前，我们想要计算出如上前端性能的话，我们需要使用时间戳来大概估计下要多长时间。比如使用:(new Date()).getTime() 来计算之前和之后的值，然后两个值的差值就是这段的时间已用的时间。但是该方法有误差，不准确。下面我们来看看传统的方案如下：

1.1 白屏时间

白屏时间：是指用户进入该网站(比如刷新页面、跳转到新页面等通过该方式)的时刻开始计算，一直到页面内容显示出来之前的时间节点。如上图我们可以看到，这个过程包括dns查询、建立tcp链接、发送首个http请求等过程、返回html文档。

比如如下代码：

<!DOCTYPE html>performance演示varstartTime = (newDate()).getTime();varendTime = (newDate()).getTime();

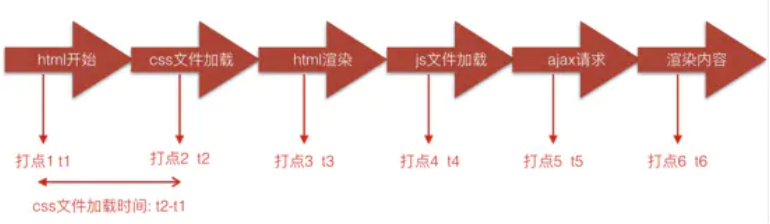
如上代码，endTime - startTime 的值就可以当作为白屏时间的估值了。

1.2 首屏时间

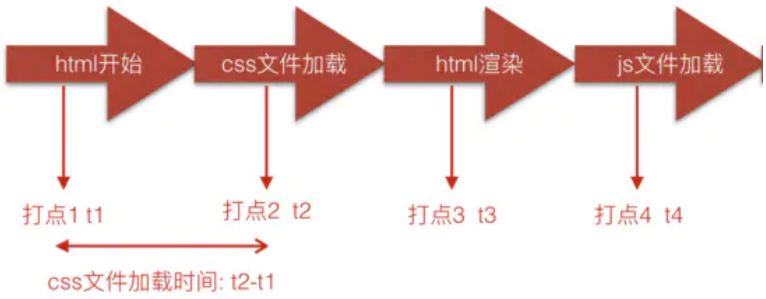
要获取首屏时间的计算，首先我们要知道页面加载有2种方式：

1. 加载完资源文件后通过js动态获取接口数据，然后数据返回回来渲染内容。

因此会有如下所示的信息图：如下所示：



2. 同构直出前端页面，如下所示：



css资源文件加载时间的计算，我们可以如上图所示：t2-t1 就是所有的css加载的时间。

因此假如我们现在的项目文件代码index.html 代码如下所示：

<!DOCTYPE html>performance演示// 获取页面开始的时间varpageStartTime = (newDate()).getTime();// 获取加载完成后的css的时间varcssEndTime = (newDate()).getTime();// 获取加载完 jquery插件的时间varjsPluginTime = (newDate()).getTime();

计算时间

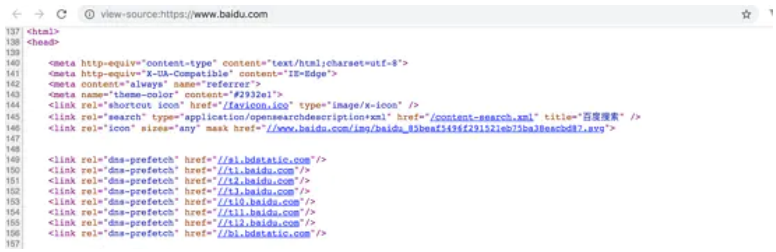
// 加载js的开始时间varJsStartTime = (newDate()).getTime();// 加载完成后的js的结束时间varJsEndTime = (newDate()).getTime();

如上代码，可以获取页面开始的时间 pageStartTime， 加载资源文件css后的时间就是cssEndTime - pageStartTime，加载jquery插件的时间 jsPluginTime - cssEndTime 了。但是js的加载时间难道就是 = JsEndTime - JsStartTime 吗？这肯定不是的，因为JS还需要有执行的时间的。比如js内部做了很多dom操作，那么dom操作需要时间的，那么js加载和执行的时间 = JsEndTime - JsStartTime; 吗？这也是不对的，因为浏览器加载资源文件是并行的，执行js文件是串行的。那如果css文件或jquery文件发起http请求后一直没有返回，那么它会阻塞后续js文件的执行的。但是此时此刻js文件加载很早就已经返回了，但是由于服务器原因或网络原因导致css文件加载很慢，所以会堵塞js文件的执行。

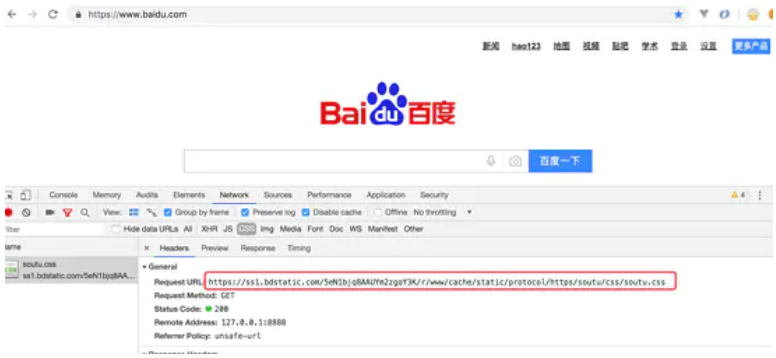
因此我们可以总结为：js加载的时间 不等于 JsEndTime - JsStartTime；同理js加载和执行的实际 也不等于 JsEndTime - JsStartTime。正因为有外链css中的http请求，它会堵塞js的执行，因此很多网站会把外链css文件改成内联css文件代码，内联css代码是串行的。比如百度，淘宝官网等。下面我们来看下这两个网站的源码：

百度源码：

我们打开百度搜索页面，然后我们右键查看网页的源码如下：

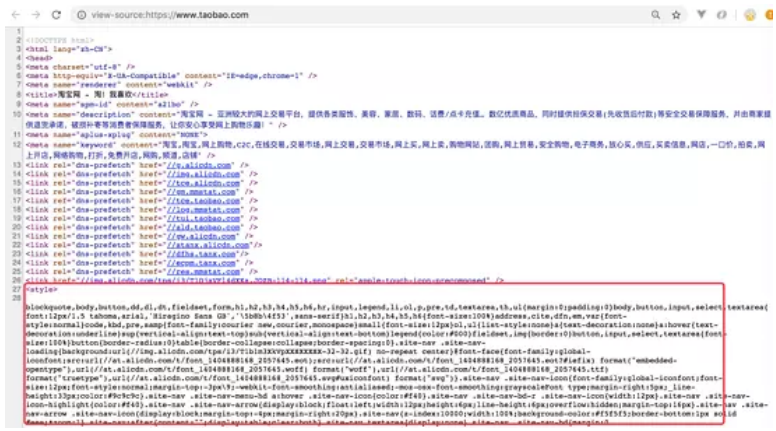


我们再来看下百度搜索页面的网络css的请求可以看到如下，同域名下根本就没有css外链操作，所有的css代码都是内联的，我们查看网络看到就仅仅有一个css外链请求，并且该css并不是百度内部的css文件，如下所示：



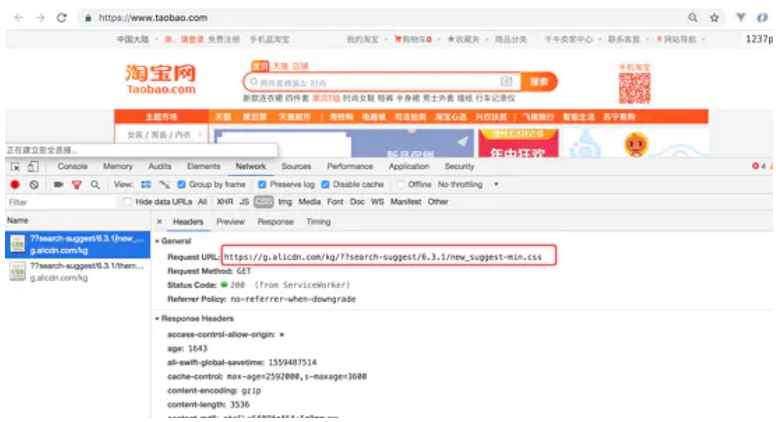
淘宝官网源码：

我们操作如上所示，打开淘宝官网源码查看如下所示：

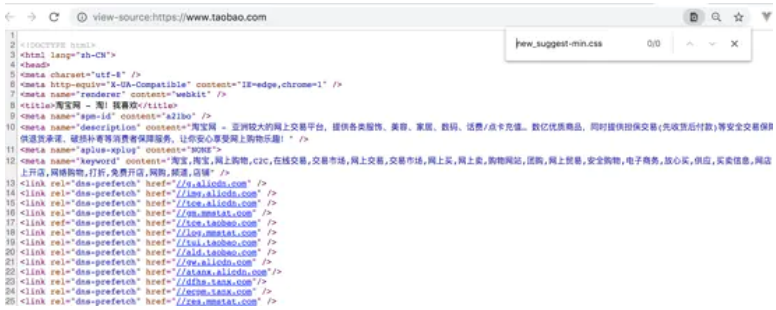


并且我们查看淘宝官网的网络请求中只看到两个css外链请求，且该两个外链请求并不是淘宝内部的同域名下的css请求，

如下所示：



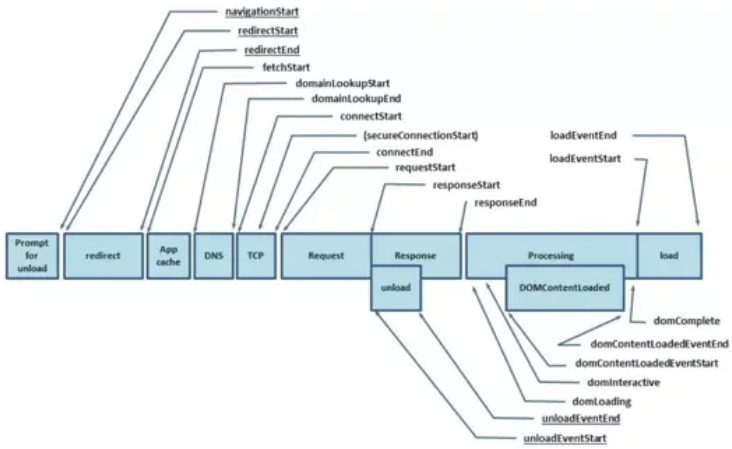
并且 该css文件名为 new_suggest-min.css 文件，我通过源码搜索下，并没有发现该css文件外链，因此可以肯定的是该css文件是通过js动态加载进去的，如下所示：



切记： 如果body下面有多个js文件的话，并且有ajax动态渲染的文件的话，那么尽量让他放在最前面，因为其他的js加载的时候会阻止页面的渲染，导致渲染js一直渲染不了，导致页面的数据会有一段时间是空白的情况。

二：使用 performance.timing 来计算值

performance 对象有一个timing属性，该属性包含很多属性值，我们还是来看看之前的示意图，如下所示：



从上面的示意图我们可以看到：

重定向耗时 = redirectEnd - redirectStart;

DNS查询耗时 = domainLookupEnd - domainLookupStart;

TCP链接耗时 = connectEnd - connectStart;

HTTP请求耗时 = responseEnd - responseStart;

解析dom树耗时 = domComplete - domInteractive;

白屏时间 = responseStart - navigationStart;

DOMready时间 = domContentLoadedEventEnd - navigationStart;

onload时间 = loadEventEnd - navigationStart;

如上就是计算方式，为了方便我们现在可以把他们封装成一个函数，然后把对应的值计算出来，然后我们根据对应的数据值来进行优化即可。

我们这边来封装下该js的计算方式, 代码如下：

functiongetPerformanceTiming(){varperformance =window.performance;if(!performance){console.log('您的浏览器不支持performance属性');return; }vart = performance.timing;varobj = {timing: performance.timing };// 重定向耗时obj.redirectTime =

t.redirectEnd - t.redirectStart;// DNS查询耗时obj.lookupDomainTime = t.domainLookupEnd - t.domainLookupStart;// TCP链接耗时obj.connectTime = t.connectEnd - t.connectStart;// HTTP请求耗时obj.requestTime = t.responseEnd - t.responseStart;// 解析dom树耗时 obj.domReadyTime = t.domComplete - t.domInteractive;// 白屏时间耗时obj.whiteTime = t.responseStart - t.navigationStart;// DOMready时间obj.domLoadTime = t.domContentLoadedEventEnd - t.navigationStart;// 页面加载完成的时间 即：onload时间 obj.loadTime = t.loadEventEnd - t.navigationStart;returnobj;}varobj = getPerformanceTiming();console.log(obj);

三：前端性能如何优化?

1. 在网页中，css资源文件尽量内联，不要外链，具体原因上面已经有说明。

2. 重定向优化，重定向有301(永久重定向)、302（临时重定向）、304（Not Modified）。前面两种重定向尽量避免。304是用来做缓存的。重定向会耗时。

3. DNS优化，如何优化呢？一般有两点：第一个就是减少DNS的请求次数，第二个就是进行DNS的预获取(Prefetching)。在PC端正常的解析DNS一次需要耗费 20-120毫秒的时间。因此我们可以减少DNS解析的次数，进而就会减少DNS解析的时间。

第二个就是DNS的预获取，什么叫预获取呢？DNS预获取就是浏览器试图在用户访问链接之前解析域名。比如说我们网站中有很多链接，但是这些链接不在同一个域名下，我们可以在浏览器加载的时候就先解析该域名，当用户真正去点击该预解析的该链接的时候，可以平均减少200毫秒的耗时(是指第一次访问该域名的时候，没有缓存的情况下)。这样就能减少用户的等待时间，提高用户体验。

我们下面可以看下淘宝的官网的代码如下就进行了DNS的Prefetch了，如下所示：

DNS Prefetch 应该尽量的放在网页的前面，推荐放在 <meta charset="UTF-8"> 后面。具体使用方法如下：

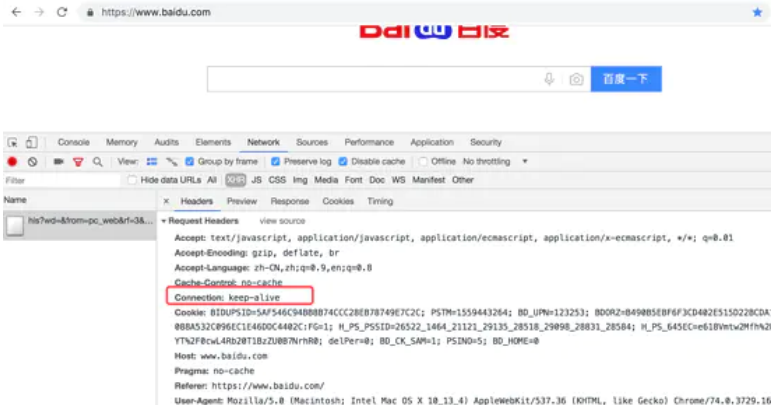
4. TCP请求优化

TCP的优化就是减少HTTP的请求数量。比如前端资源合并，图片，资源文件进行压缩等这些事情。

在http1.0当中默认使用短链接，也就是客户端和服务端进行一次http请求的时候，就会建立一次链接，任务结束后就会中断该链接。那么在这个过程当中会有3次TCP请求握手和4次TCP请求释放操作。

在http1.1中，在http响应头会加上 Connection: keep-alive，该代码的含义是：当该网页打开完成之后，链接不会马上关闭，

当我们再次访问该链接的时候，会继续使用这个长连接。这样就减少了TCP的握手次数和释放次数。只需要建立一次TCP链接即可，比如我们看下百度的请求如下所示：



5. 渲染优化

在我们做vue或react项目时，我们常见的模板页面是通过js来进行渲染的。而不是同构直出的html页面，对于这个渲染过程中对于我们首屏就会有很大的损耗，白屏的时间会增加。因此我们可以使用同构直出的方式来进行服务器端渲染html页面会比较好，或者我们可以使用一些webpack工具进行html同构直出渲染，webpack渲染可以看这篇文章。

四：Performance中方法

首先我们在控制台中打印下 performance 中有哪些方法，如下代码：

varperformance =window.performance;console.log(performance);

如下所示：

4.1 performance.getEntries()

该方法包含了所有静态资源的数组列表。

比如我现在html代码如下：

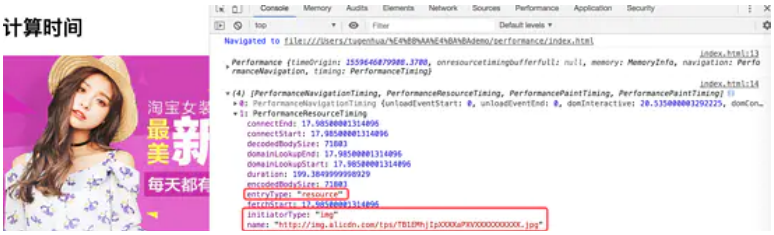
<!DOCTYPE html>performance演示

计算时间

window.onload =function(){varperformance =window.performance;console.log(performance);console.log(performance.getEntries()) }

如上代码，我页面上包含一张淘宝cdn上的一张图片，因为该方法会获取页面中所有包含了页面中的 HTTP 请求。

然后我们在浏览器中看到打印performance.getEntries()信息如下：



该对象的属性中除了包含资源加载时间，还有如下几个常见的属性：

name:资源名称，是资源的绝对路径，如上图就是淘宝cdn上面的图片路径。我们可以通过performance.getEntriesByName(name属性值)，来获取该资源加载的具体属性。

startTime:开始时间

duration:表示加载时间，它是一个毫秒数字，只能获取同域下的时间点，如果是跨域的话，那么该时间点为0。

entryType:资源类型 "resource", 还有 "navigation", "mark" 和 "measure" 这三种。

initiatorType:表示请求的来源的标签，比如 link标签、script标签、img标签等。

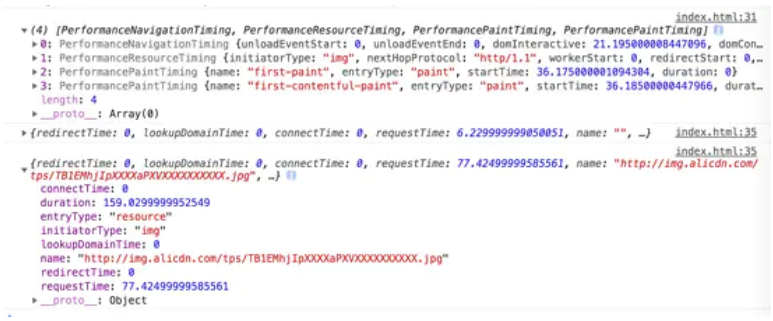
因此我们可以像 getPerformanceTiming 该方法一样，封装一个方法，获取某个资源的时间。如下封装代码方法如下：

<!DOCTYPE html>performance演示

计算时间

// 计算加载时间functiongetEntryTiming(entry){varobj = {};// 获取重定向的时间 obj.redirectTime = entry.redirectEnd - entry.redirectStart;// 获取DNS查询耗时 obj.lookupDomainTime = entry.domainLookupEnd - entry.domainLookupStart;// 获取TCP链接耗时obj.connectTime = entry.connectEnd - entry.connectStart;// HTTP请求耗时 obj.requestTime = entry.responseEnd - entry.responseStart; obj.name = entry.name; obj.entryType = entry.entryType; obj.initiatorType = entry.initiatorType; obj.duration = entry.duration;returnobj; }window.onload =function(){varentries =window.performance.getEntries();console.log(entries); entries.forEach(function(item) {if(item.initiatorType) {varcurItem = getEntryTiming(item);console.log(curItem); } }); }

然后如上会有2个console.log 打印数据，我们到控制台中看到打印信息如下：



如上图所示，我们可以看到通过 getEntryTiming 方法计算后，会拿到各对应的值。

4.2 performance.now()

该方法会返回一个当前页面执行的时间的时间戳，可以用来精确计算程序执行的实际。

比如如下，我循环100万次，然后返回一个数组，我们来看下代码如下：

<!DOCTYPE html>performance演示

计算时间

functiondoFunc(){vararrs = [];for(vari =0; i <1000000; i++) { arrs.push({'label': i,'value': i }); }returnarrs; }vart1 =window.performance.now();console.log(t1); doFunc();vart2 =window.performance.now();console.log(t2);console.log('doFunc函数执行的时间为：'+ (t2 - t1) +'毫秒');

然后我们再来打印下 doFunc() 这个函数执行了多久，如下所示：

我们也知道我们还有一个时间就是 Date.now(), 但是performance.now()与Date.now()方法不同的是：

该方法使用了一个浮点数，返回的是以毫秒为单位，小数点精确到微妙级别的时间。相对于 Date.now() 更精确，并且不会受系统程序堵塞的影响。

下面我们来看看使用 Date.now()方法使用的demo如下：

<!DOCTYPE html>performance演示

计算时间

functiondoFunc(){vararrs = [];for(vari =0; i <1000000; i++) { arrs.push({'label': i,'value': i }); }returnarrs; }vart1 =Date.now();console.log(t1); doFunc();vart2 =Date.now();console.log(t2);console.log('doFunc函数执行的时间为：'+ (t2 - t1) +'毫秒');

执行的结果如下：

注意： performance.timing.navigationStart + performance.now() 约等于Date.now();

4.3 performance.mark()

该方法的含义是用来自定义添加标记的时间, 方便我们计算程序的运行耗时。该方法使用如下：

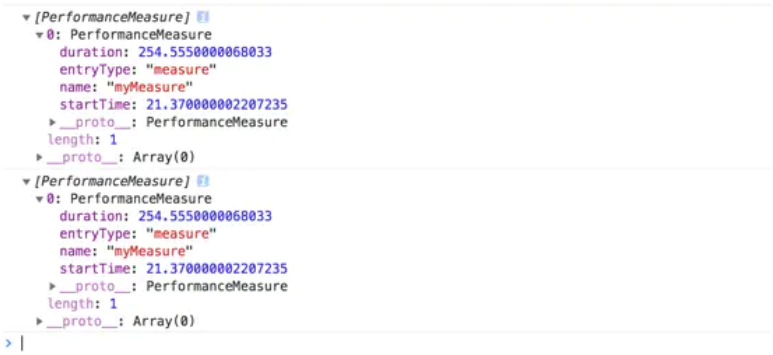
<!DOCTYPE html>performance演示

计算时间

functiondoFunc(){vararrs = [];for(vari =0; i <1000000; i++) { arrs.push({'label': i,'value': i }); }returnarrs; }// 函数执行前做个标记varmStart ='mStart';varmEnd ='mEnd';window.performance.mark(mStart); doFunc();// 函数执行之后再做个标记 window.performance.mark(mEnd);// 然后测量这两个标记之间的距离，并且保存起来varname ='myMeasure';window.performance.measure(name, mStart, mEnd);// 下面我们通过 performance.getEntriesByName 方法来获取该值 console.log(performance.getEntriesByName('myMeasure'));console.log(performance.getEntriesByType('measure'));

如上代码，我们通过 window.performance.measure(name, mStart, mEnd); 这个方法做出标记后，我们可以使用performance.getEntriesByName('myMeasure') 和 performance.getEntriesByType('measure') 获取该值。

如下图所示：



4.4 performance.getEntriesByType()

该方法返回一个 PerformanceEntry 对象的列表，基于给定的 entry type， 如上代码 performance.getEntriesByType('measure') 就可以获取该值。

4.5 performance.clearMeasures()

从浏览器的性能输入缓冲区中移除自定义添加的 measure. 代码如下所示：

<!DOCTYPE html>performance演示

计算时间

functiondoFunc(){vararrs = [];for(vari =0; i <1000000; i++) { arrs.push({'label': i,'value': i }); }returnarrs; }// 函数执行前做个标记varmStart ='mStart';varmEnd ='mEnd';window.performance.mark(mStart); doFunc();// 函数执行之后再做个标记 window.performance.mark(mEnd);// 然后测量这两个标记之间的距离，并且保存起来varname ='myMeasure';window.performance.measure(name, mStart, mEnd);// 下面我们通过 performance.getEntriesByName 方法来获取该值 console.log(performance.getEntriesByName('myMeasure'));console.log(performance.getEntriesByType('measure'));// 使用 performance.clearMeasures() 方法来清除 自定义添加的 measureperformance.clearMeasures();console.log(performance.getEntriesByType('measure'));

如上我们在最后代码中使用 performance.clearMeasures() 方法清除了所有自定义的 measure。然后我们后面重新使用 console.log(performance.getEntriesByType('measure'));打印下，看到如下信息：

4.6 performance.getEntriesByName(name属性的值)

该方法返回一个 PerformanceEntry 对象的列表，基于给定的 name 和 entry type。

4.7 performance.toJSON()

该方法是一个 JSON 格式转化器，返回 Performance 对象的 JSON 对象。如下代码所示：

<!DOCTYPE html>performance演示

计算时间

console.log(window.performance);varjs =window.performance.toJSON();console.log("json = "+JSON.stringify(js));

然后打印信息如下：

五：使用performane编写小工具

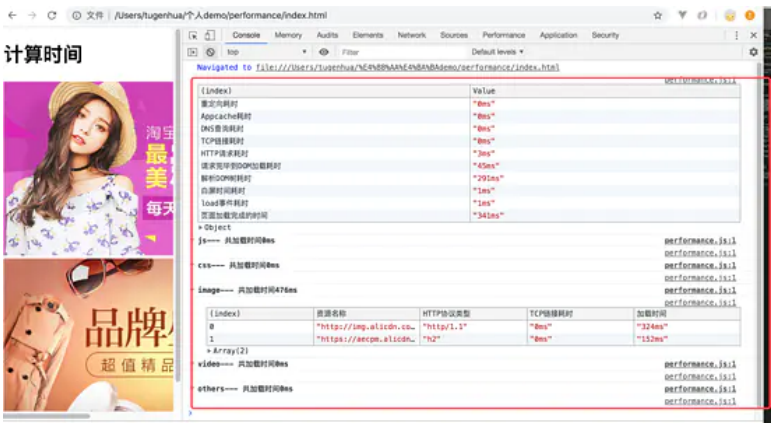
首先html使用代码如下(切记一定要把初始代码放到window.onload里面，因为确保图片加载完成)：

<!DOCTYPE html>performance演示

计算时间

window.onload =function(){window.performanceTool.getPerformanceTiming(); };

然后页面进行预览效果如下：



如上图我们就可以很清晰的可以看到，页面的基本信息了，比如：重定向耗时、Appcache耗时、DNS查询耗时、TCP链接耗时、HTTP请求耗时、请求完毕到DOM加载耗时、解析DOM树耗时、白屏时间耗时、load事件耗时、及 页面加载完成的时间。页面加载完成的时间 是上面所有时间的总和。及下面，我们也可以分别清晰的看到，js、css、image、video、等信息的资源加载及总共用了多少时间。

js基本代码如下：

src/utils.js 代码如下：

```
exportfunctionisObject(obj){returnobj !==null&& (typeofobj ==='object')}// 格式化成毫秒
exportfunctionformatMs(time){if(typeoftime !=='number') {console.log('时间必须为数字');return;  }// 毫秒转换成秒 返回if(time >1000) {return(time /1000).toFixed(2) +'s';  }// 默认返回毫秒returnMath.round(time) +'ms';}exportfunctionisImg(param){if(/\.(gif|jpg|jpeg|png|webp|svg)/i.test(param)) {returntrue;    }returnfalse;
}exportfunctionisJS(param){if(/\.(js)/i.test(param)) {returntrue;    }returnfalse;
}exportfunctionisCss(param){if(/\.(css)/i.test(param)) {returntrue;    }returnfalse;
}exportfunctionisVideo(param){if(/\.(mp4|rm|rmvb|mkv|avi|flv|ogv|webm)/i.test(name)) {returntrue;    }returnfalse;  }exportfunctioncheckResourceType(param){if(isImg(param)) {return'image';  }if(isJS(param)) {return'javascript';  }if(isCss(param)) {return'css';
}if(isVideo(param)) {return'video';    }return'other'}
```

js/index.js 代码如下：

```
varutils =require('./utils');varformatMs = utils.formatMs;varisObject = utils.isObject;varcheckResourceType = utils.checkResourceType;functionPerformance() {};Performance.prototype = {// 获取数据信息getPerformanceTiming:function(){// 初始化数据this.init();if(!isObject(this.timing)) {console.log('值需要是一个对象类型');return;    }// 过早获取 loadEventEnd值会是0varloadTime =this.timing.loadEventEnd - this.timing.navigationStart;if(loadTime <0) {      setTimeout(()=> {this.getPerformanceTiming();      },200);return;    }// 获取解析后的数据this.afterDatas.timingFormat =this._setTiming(loadTime);this.afterDatas.enteriesResouceDataFormat =this._setEnteries();this._show();  },init:function(){this.timing =window.performance.timing;// 获取资源类型为 resource的所有数据this.enteriesResouceData =window.performance.getEntriesByType('resource');  },// 保存原始数据timing: {},// 原始enteries数据enteriesResouceData: [],// 保存解析后的数据afterDatas: {timingFormat: {},enteriesResouceDataFormat: {},enteriesResouceDataTiming: {"js":0,"css":0,"image":0,"video":0,"others":0}  },_setTiming:function(loadTime){vartiming =this.timing;// 对数据进行计算vardata = {"重定向耗时": formatMs(timing.redirectEnd - timing.redirectStart),"Appcache耗时": formatMs(timing.domainLookupStart - timing.fetchStart),"DNS查询耗时": formatMs(timing.domainLookupEnd - timing.domainLookupStart),"TCP链接耗时": formatMs(timing.connectEnd - timing.connectStart),"HTTP请求耗时": formatMs(timing.responseEnd - timing.responseStart),"请求完毕到DOM加载耗时": formatMs(timing.domInteractive - timing.responseEnd),"解析DOM树耗时": formatMs(timing.domComplete - timing.domInteractive),"白屏时间耗时": formatMs(timing.responseStart - timing.navigationStart),"load事件耗时": formatMs(timing.loadEventEnd - timing.loadEventStart),"页面加载完成的时间": formatMs(loadTime)    };returndata;
},_setEnteries:function(){varenteriesResouceData =this.enteriesResouceData;varimageArrs = [],      jsArrs = [],      cssArrs = [],      videoArrs = [],      otherArrs = [];
enteriesResouceData.map(item=>{vard = {'资源名称': item.name,'HTTP协议类型': item.nextHopProtocol,"TCP链接耗时": formatMs(item.connectEnd - item.connectStart),"加载时间": formatMs(item.duration)      };switch(checkResourceType(item.name)) {case'image':this.afterDatas.enteriesResouceDataTiming.image += item.duration;
imageArrs.push(d);break;case'javascript':this.afterDatas.enteriesResouceDataTiming.js += item.duration;
jsArrs.push(d);break;case'css':this.afterDatas.enteriesResouceDataTiming.css += item.duration;
cssArrs.push(d);break;case'video':this.afterDatas.enteriesResouceDataTiming.video += item.duration;
videoArrs.push(d);break;case'others':this.afterDatas.enteriesResouceDataTiming.others += item.duration;          otherArrs.push(d);break;      }    });return{'js': jsArrs,'css': cssArrs,'image': imageArrs,'video': videoArrs,'others': otherArrs    }  },_show:function()
```

```
{console.table(this.afterDatas.timingFormat);for(varkeyinthis.afterDatas.enteriesResouceDataFormat ){console.group(key +"--- 共加载时间"+ formatMs(this.afterDatas.enteriesResouceDataTiming[key]));console.table(this.afterDatas.enteriesResouceDataFormat[key]);console.groupEnd(key);  } };varPer =newPerformance();module.exports = Per;
```