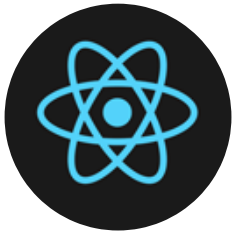
100G前端干货教程


React 中文文档

跟牛人学前端


CSS3参考手册

前端最新干货

web在线直播课

JavaScript 代码片段

Parcel 中文文档

CSS参考手册(包含css3) jQuery API中文文档 / 更新说明 Chrome DevTools 中文文档 webpack 中文文档(v4.15.1) Lodash中文文档
Sass中文文档 CSS3代码生成(演示) React 中文文档（V16.4.1） TypeScript 中文文档(v2.4) jQuery UI API中文文档 Zepto.js 中文文档
WEB开发速查表 HTML 2 JS 工具 Backbone.js 中文文档 Bootstrap 中文文档 Underscore.js 中文文档 JSDoc JSDoc中文文档
Javascript格式化工具 CSS整理与优化工具 更多前端文档及工具

您的位置：首页 » 分类: HTML+CSS & JavaScript & 前端资源 » 文章: 页面重绘和回流以及优化

搜索…

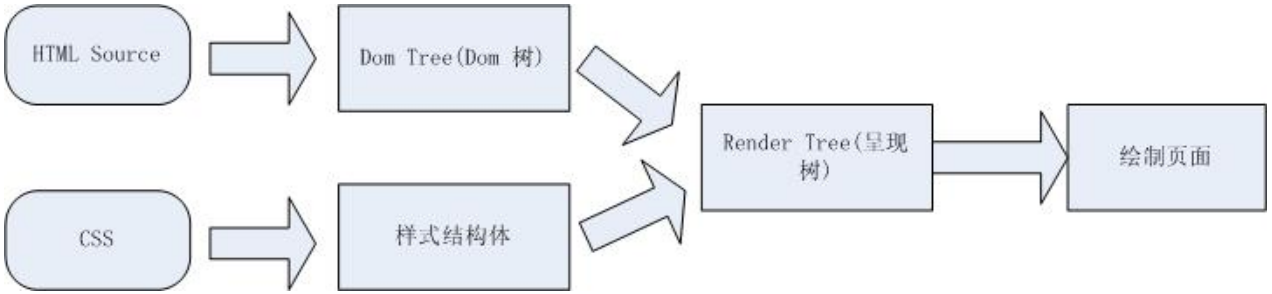
WEB前端开发公众号
关注国内外最新最好的前端开发技术干货，获取最新前端开发资讯

分类目录

HTML+CSS (313)

小编推荐：掘金是一个面向程序员的高质量技术社区，从 一线大厂经验分享到前端开发最佳实践，无论是入门还是进阶，来掘金你不会错过前端开发的任何一个技术干货。

在讨论页面重绘、回流之前。需要对页面的呈现流程有些了解，页面是怎么把html结合css等显示到浏览器上的，下面的流程图显示了浏览器对页面的呈现的处理流程。可能不同的浏览器略微会有些不同。但基本上都是类似的。



1. 浏览器把获取到的HTML代码解析成1个DOM树，HTML中的每个tag都是DOM树中的1个节点，根节点就是我们常用的document对象。DOM树里包含了所有HTML标签，包括display:none隐藏，还有用JS动态添加的元素等。

2. 浏览器把所有样式(用户定义的CSS和用户代理)解析成样式结构体，在解析的过程中会去掉浏览器不能识别的样式，比如IE会去掉-moz开头的样式，而FF会去掉_开头的样式。

3、DOM Tree 和样式结构体组合后构建render tree, render tree类似于DOM tree，但区别很大，render tree能识别样式，render tree中每个NODE都有自己的style，而且 render tree不包含隐藏的节点 (比如display:none的节点，还有head节点)，因为这些节点不会用于呈现，而且不会影响呈现的，所以就不会

4. 一旦render tree构建完毕后，浏览器就可以根据render tree来绘制页面了。

回流与重绘

1. 当render tree中的一部分(或全部)因为元素的规模尺寸，布局，隐藏等改变而需要重新构建。这就称为回流(reflow)。每个页面至少需要一次回流，就是在页面第一次加载的时候。在回流的时候，浏览器会使渲染树中受到影响的部分失效，并重新构造这部分渲染树，完成回流后，浏览器会重新绘制受影响的部分到屏幕中，该过程成为重绘。

2. 当render tree中的一些元素需要更新属性，而这些属性只是影响元素的外观，风格，而不会影响布局的，比如background-color。则就叫称为重绘。

注意：回流必将引起重绘，而重绘不一定会引起回流。

回流何时发生：

当页面布局和几何属性改变时就需要回流。下述情况会发生浏览器回流：

1、添加或者删除可见的DOM元素；

2、元素位置改变；

3、元素尺寸改变——边距、填充、边框、宽度和高度

4、内容改变——比如文本改变或者图片大小改变而引起的计算值宽度和高度改变；

5、页面渲染初始化；

例》

黄金投资理财发表在《React 教程：函数作为子组件(Function as Child Components)》

react实现多行文本超出加省略号 _ IT人知识库发表在《多行文本溢出显示省略号(…)全攻略》

郝冰发表在《更多关于Flexbox布局如何工作的 – 用大彩图和GIF动画解释》

小白发表在《CSS引用(link)和导入(@import)的区别(2009年4月13日更新)》

标签

Node.js 前端开发 ES2017 webpack 插件 Promise ES6

```
js 代码:

1. var s = document.body.style;
2. s.padding = "2px"; // 回流+重绘
3. s.border = "1px solid red"; // 再一次 回流+重绘
4. s.color = "blue"; // 再一次重绘
5. s.backgroundColor = "#ccc"; // 再一次 重绘
6. s.fontSize = "14px"; // 再一次 回流+重绘
7. // 添加node，再一次 回流+重绘
8. document.body.appendChild(document.createTextNode('abc!'));
```

说到这里大家都知道回流比重绘的代价要更高，回流的花销跟render tree有多少节点需要重新构建有关系，假设你直接操作body，比如在body最前面插入1个元素，会导致整个render tree回流，这样代价当然会比较高，但如果是指body后面插入1个元素，则不会影响前面元素的回流。

聪明的浏览器

从上个实例代码中可以看到几行简单的JS代码就引起了6次左右的回流、重绘。而且我们也知道回流的花销也不小，如果每句JS操作都去回流重绘的话，浏览器可能就会受不了。所以很多浏览器都会优化这些操作，浏览器会维护1个队列，把所有会引起回流、重绘的操作放入这个队列，等队列中的操作到了一定的数量或者到了一定的时间间隔，浏览器就会flush队列，进行一个批处理。这样就会让多次的回流、重绘变成一次回流重绘。

虽然有了浏览器的优化，但有时候我们写的一些代码可能会强制浏览器提前flush队列，这样浏览器的优化可能就起不到作用了。当你请求向浏览器请求一些 style信息的时候，就会让浏览器flush队列，比如：

4. width,height

5. 请求了getComputedStyle(), 或者 IE的 currentStyle

当你请求上面的一些属性的时候，浏览器为了给你最精确的值，需要flush队列，因为队列中可能会有影响到这些值的操作。即使你获取元素的布局和样式信息跟最近发生或改变的布局信息无关，浏览器都会强行刷新渲染队列。

如何减少回流、重绘

减少回流、重绘其实就是需要减少对render tree的操作（合并多次多DOM和样式的修改），并减少对一些style信息的请求，尽量利用好浏览器的优化策略。具体方法有：

1. 直接改变className，如果动态改变样式，则使用cssText（考虑没有优化的浏览器）

```
js 代码:

// 不好的写法
var left = 1;
var top = 1;
el.style.left = left + "px";
el.style.top = top + "px";// 比较好的写法
el.className += " className1";
// 比较好的写法
el.style.cssText += ";
left: " + left + "px;
top: " + top + "px;";
```

2. 让要操作的元素进行”离线处理”，处理完后一起更新

React 教程：函数作为子组件 (Function as Child Components) - 2,667

前端性能优化：2018年JavaScript开销及优化工具和方法 - 2,618

深入理解 React 高阶组件（Higher Order Component，简称：HOC） - 2,612

React 组件模式 - 2,421

React 组件模式-有状态组件 x 无状态组件、容器组件 x 展示组件、高阶组件 x 渲染回调(函数作为子组件) - 640

3.不要经常访问会引起浏览器flush队列的属性，如果你确实要访问，利用缓存

```
js 代码:

1. // 别这样写，大哥
2. for(循环) {
3. el.style.left = el.offsetLeft + 5 + "px";
4. el.style.top = el.offsetTop + 5 + "px";
5. }
6. // 这样写好点
7. var left = el.offsetLeft,
8. top = el.offsetTop,
9. s = el.style;
10. for (循环) {
11. left += 10;
12. top += 10;
13. s.left = left + "px";
14. s.top = top + "px";
15. }
```

4. 让元素脱离动画流，减少回流的Render Tree的规模

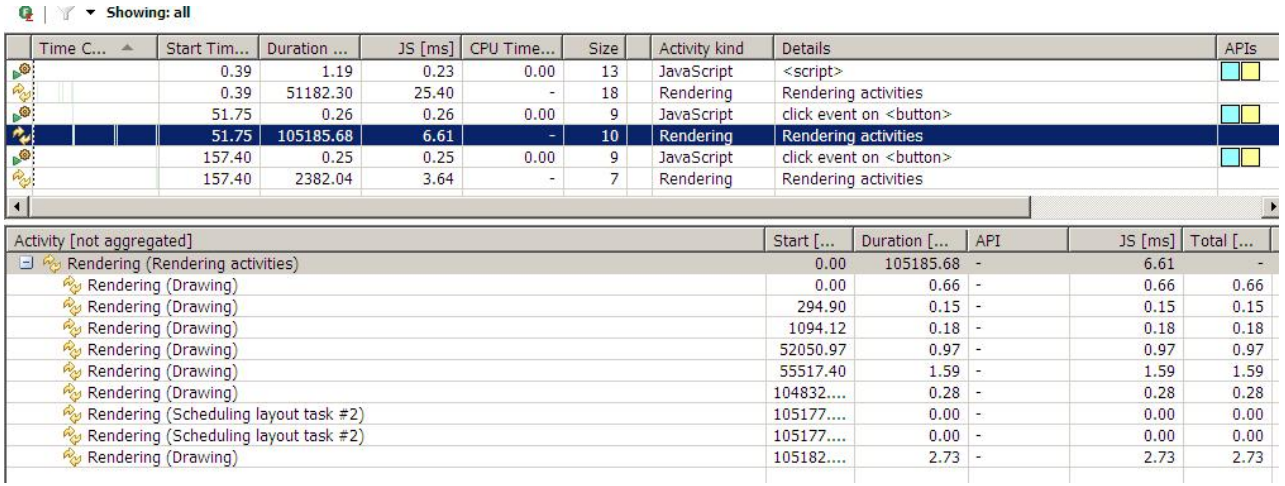
```
js 代码:

1. $("#block1").animate({left:50});
2. $("#block2").animate({marginLeft:50});
```

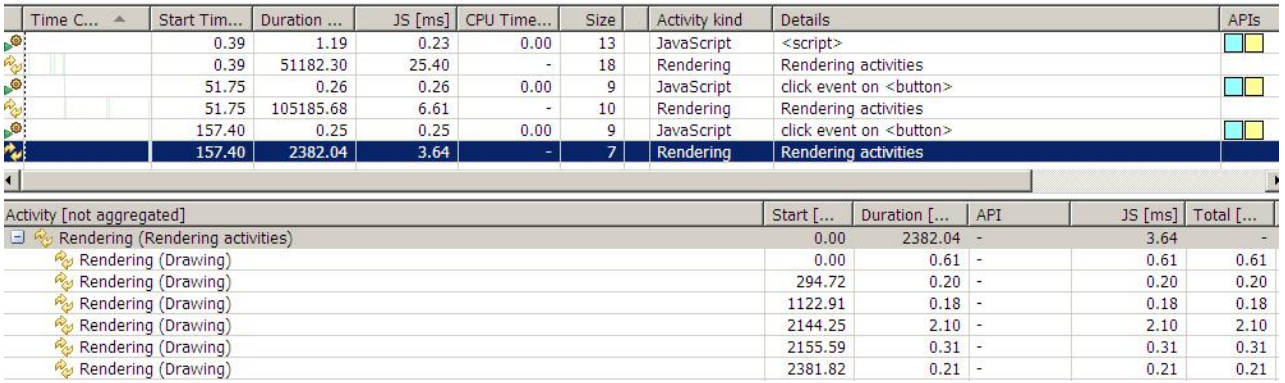
实例测试

html 代码:

```
<body>
<script type="text/javascript">
var s = document.body.style;
var computed;
if (document.body.currentStyle) {
computed = document.body.currentStyle;
} else {
computed = document.defaultView.getComputedStyle(document.body, '');
}
function testOneByOne(){
s.color = 'red';;
tmp = computed.backgroundColor;
s.color = 'white';
tmp = computed.backgroundImage;
s.color = 'green';
tmp = computed.backgroundAttachment;
}
function testAll() {
s.color = 'yellow';
s.color = 'pink';
s.color = 'blue';
tmp = computed.backgroundColor;
tmp = computed.backgroundImage;
tmp = computed.backgroundAttachment;
}
</script>
color test <br />
<button onclick="testOneByOne()">Test One by One</button>
<button onclick="testAll()">Test All</button>
</body>
```



上图可以看到我们执行了2次button的click事件，每次click后都跟一次rendering，2次click函数执行的时间都差不多, 0.25ms,0.26ms，但其后的rendering时间就相差一倍多。(其实很多时候，前端的性能瓶颈并不在于JS的执行，而是在于页面的呈现，这种情况在富客户端中更为突出)。我们再看图的下面部分，这是第一次rendering的详细信息，可以看到里面有2行是 Scheduleing layout task，这个就是我们前面讨论过的浏览器优化过的队列，可以看出我们引发2次的flush。

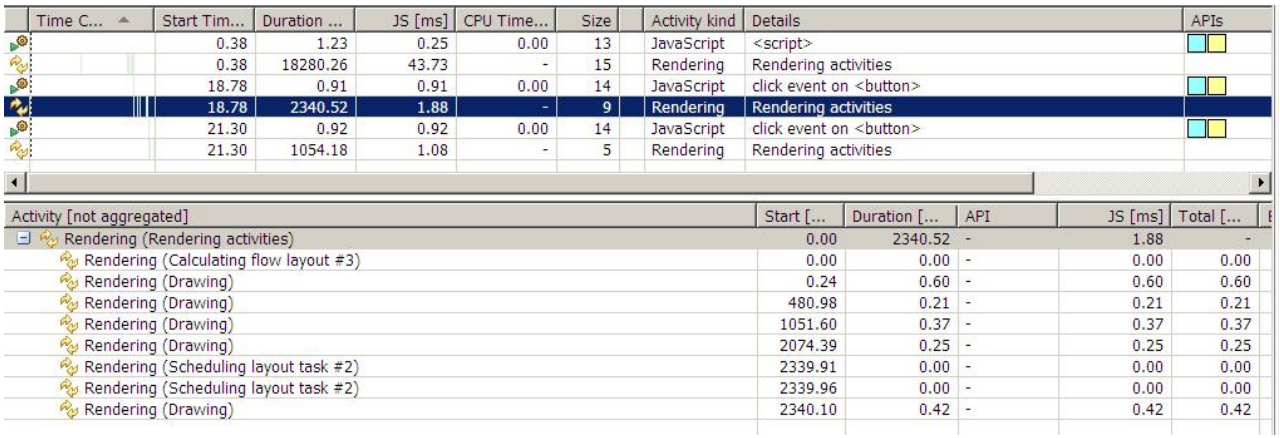


html 代码:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1
/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
</head>
<body>
<script type="text/javascript">
var s = document.body.style;
var computed;
if (document.body.currentStyle) {
computed = document.body.currentStyle;
} else {
computed = document.defaultView.getComputedStyle(document.body, '');
}
function testOneByOne(){
s.color = 'red';
s.padding = '1px';
tmp = computed.backgroundColor;
s.color = 'white';
s.padding = '2px';
tmp = computed.backgroundImage;
s.color = 'green';
s.padding = '3px';
tmp = computed.backgroundAttachment;
}
function testAll() {
s.color = 'yellow';
s.padding = '4px';
s.color = 'pink';
s.padding = '5px';
s.color = 'blue';
```

```
38. color test <br />
39. <button onclick="testOneByOne()">Test One by One</button>
40. <button onclick="testAll()">Test All</button>
41. </body>
42. </html>
```

用dynaTrace监控如下：



Time C...	Start Tim...	Duration ...	JS [ms]	CPU Time...	Size	Activity kind	Details	APIs
	0.38	1.23	0.25	0.00	13	JavaScript	<script>	
	0.38	18280.26	43.73	-	15	Rendering	Rendering activities	
	18.78	0.91	0.91	0.00	14	JavaScript	click event on <button>	
	18.78	2340.52	1.88	-	9	Rendering	Rendering activities	
	21.30	0.92	0.92	0.00	14	JavaScript	click event on <button>	
	21.30	1054.18	1.08	-	5	Rendering	Rendering activities	

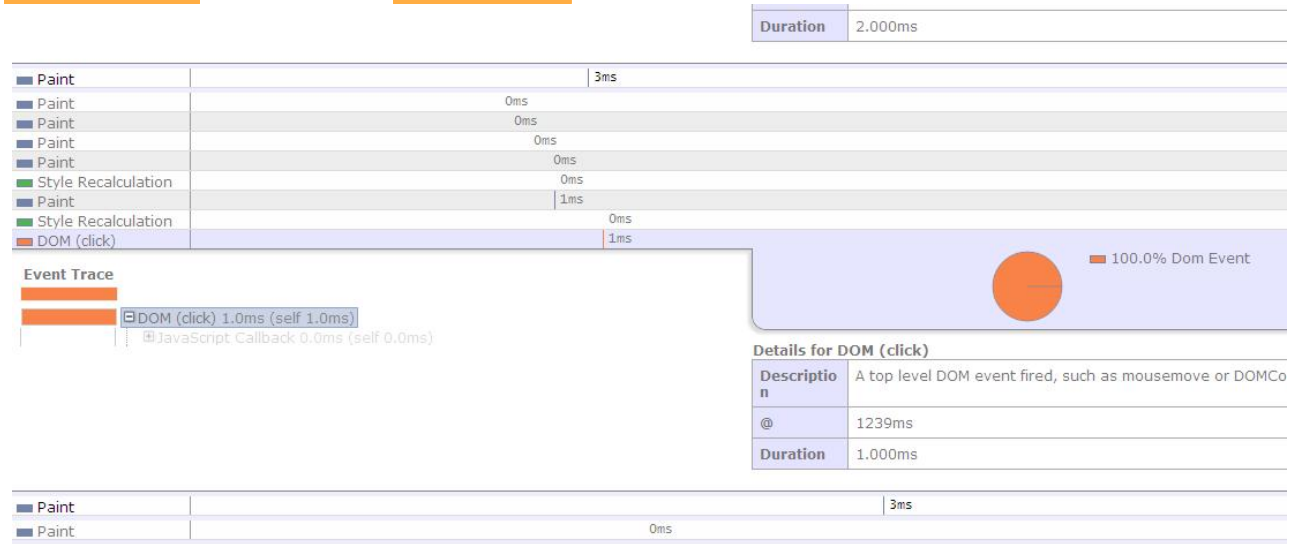
Activity [not aggregated]	Start [...	Duration [...	API	JS [ms]	Total [...
Rendering (Rendering activities)	0.00	2340.52	-	1.88	-
Rendering (Calculating flow layout #3)	0.00	0.00	-	0.00	0.00
Rendering (Drawing)	0.24	0.60	-	0.60	0.60
Rendering (Drawing)	480.98	0.21	-	0.21	0.21
Rendering (Drawing)	1051.60	0.37	-	0.37	0.37
Rendering (Drawing)	2074.39	0.25	-	0.25	0.25
Rendering (Scheduling layout task #2)	2339.91	0.00	-	0.00	0.00
Rendering (Scheduling layout task #2)	2339.96	0.00	-	0.00	0.00
Rendering (Drawing)	2340.10	0.42	-	0.42	0.42

这图可以看出，有了回流后，rendering的时间相比之前的只重绘，时间翻了3倍了，可见回流的高成本性啊。

大家看到时候注意明细处相比之前的多了个 Calcalating flow layout。

最后再使用Speed Tracer测试一下，其实结果是一样的，只是让大家了解下2个测试工具：

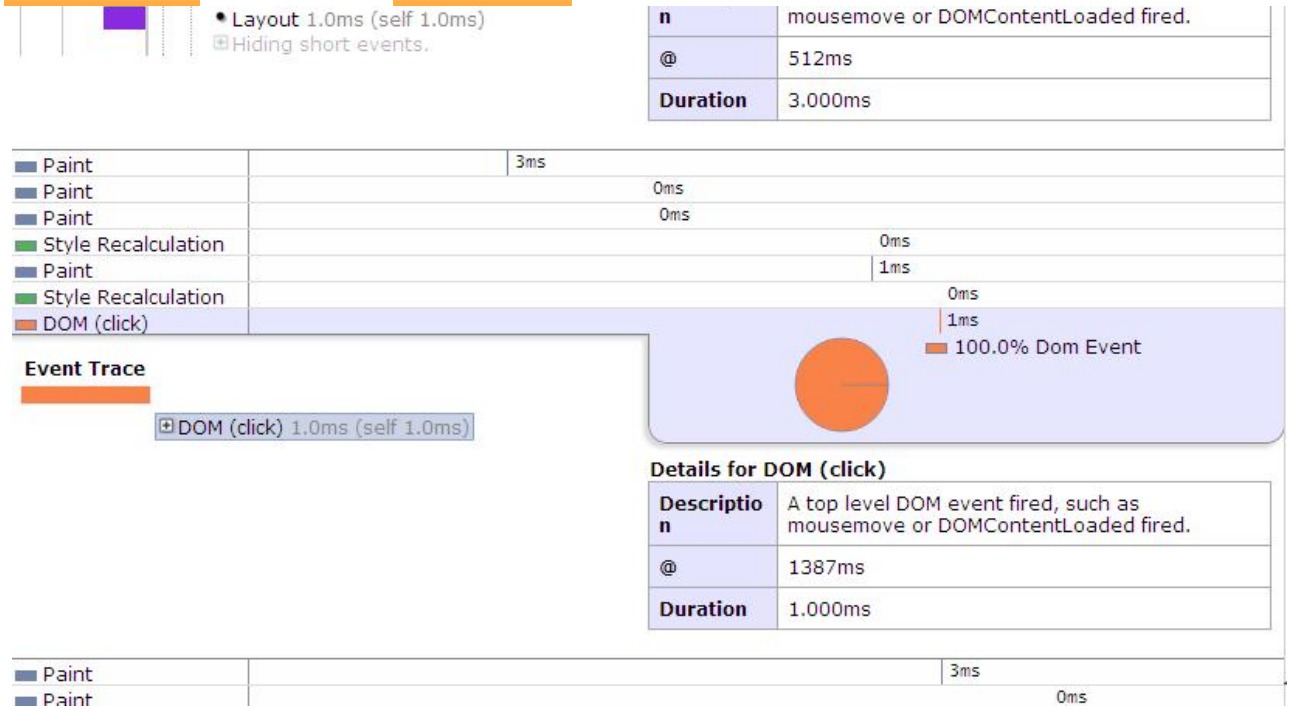
测试1：



图上第一次点击执行2ms (其中有50% 用于style Recalculation), 第二次1ms，而且第一次click后面也跟了2次style Recalculation,而第二次点击却没有style Recalculation。

但是这次测试发现paint重绘的时间竟然是一样的，都是3ms，这可能就是chrome比IE强的地方吧。

测试2：



从图中竟然发现第二次的测试结果在时间上跟第一次的完全一样，这可能是因为操作太少，而chrome又比较强大，所以没能测试明显结果出来。

但注意图中多了1个紫色部分，就是layout的部分。也就是我们说的回流。

参考文章：

浏览器的重绘[repaints]与重排[reflows]

高性能WEB开发–页面呈现、重绘、回流

所属分类： HTML+CSS, JavaScript, 前端资源 | 标签： 优化, 回流, 重绘 | 书签： permalink.

您可能感兴趣的文章

JavaScript性能优化技巧：概述 11-29
CSS优化：使用 DevTools 优化动画性能 11-27
使用 webpack 3 构建高性能的应用程序 07-20
2017年成为全栈开发工程师的权威指南 06-15
降低样式计算的范围和复杂度 12-22
优化JavaScript的执行效率 12-20
浏览器的重绘(repaints)与重排(reflows) 08-11
window resize和scroll事件的基本优化 04-08
前端性能优化：2018年JavaScript开销及优化工具和方法(0) 08-28
使用顶级 VSCode 扩展来加快开发 JavaScript(0) 08-24

← 浏览器的重绘(repaints)与重排(reflows)
cookie窃取和session劫持 →

Pingback： 高性能JavaScript编程阅读简记（二） – 李小白要天天向上

Pingback： 页面的重绘与回流及优化 | Codeba

Pingback： CSS Animation性能优化 - 小戒

Pingback： 页面的重绘与回流及优化 - 南山老幺 - – 引力一族

Pingback： 28-重绘和回流 – 静修小栈

发表评论

电子邮件地址不会被公开。 必填项已用*标注

评论

发表评论