

22 | DOM树：JavaScript是如何影响DOM树构建的？

2019-09-24 李兵

浏览器工作原理与实践

[进入课程 >](#)



讲述：李兵

时长 12:56 大小 14.81M



在[上一篇文章](#)中，我们通过开发者工具中的网络面板，介绍了网络请求过程的几种**性能指标**以及对页面加载的影响。

而在渲染流水线中，后面的步骤都直接或者间接地依赖于 DOM 结构，所以本文我们就继续沿着网络数据流路径来**介绍 DOM 树是怎么生成的**。然后再基于 DOM 树的解析流程介绍两块内容：第一个是在解析过程中遇到 JavaScript 脚本，DOM 解析器是如何处理的？第二个是 DOM 解析器是如何处理跨站点资源的？

什么是 DOM

从网络传给渲染引擎的 HTML 文件字节流是无法直接被渲染引擎理解的，所以要将其转化为渲染引擎能够理解的内部结构，这个结构就是 DOM。DOM 提供了对 HTML 文档结构

化的表述。在渲染引擎中，DOM 有三个层面的作用。

从页面的视角来看，DOM 是生成页面的基础数据结构。

从 JavaScript 脚本视角来看，DOM 提供给 JavaScript 脚本操作的接口，通过这套接口，JavaScript 可以对 DOM 结构进行访问，从而改变文档的结构、样式和内容。

从安全视角来看，DOM 是一道安全防护线，一些不安全的内容在 DOM 解析阶段就被拒之门外了。

简言之，DOM 是表述 HTML 的内部数据结构，它会将 Web 页面和 JavaScript 脚本连接起来，并过滤一些不安全的内容。

DOM 树如何生成

在渲染引擎内部，有一个叫**HTML 解析器 (HTMLParser)** 的模块，它的职责就是负责将 HTML 字节流转换为 DOM 结构。所以这里我们需要先要搞清楚 HTML 解析器是怎么工作的。

在开始介绍 HTML 解析器之前，我要先解释一个大家在留言区问到过好多次的问题：

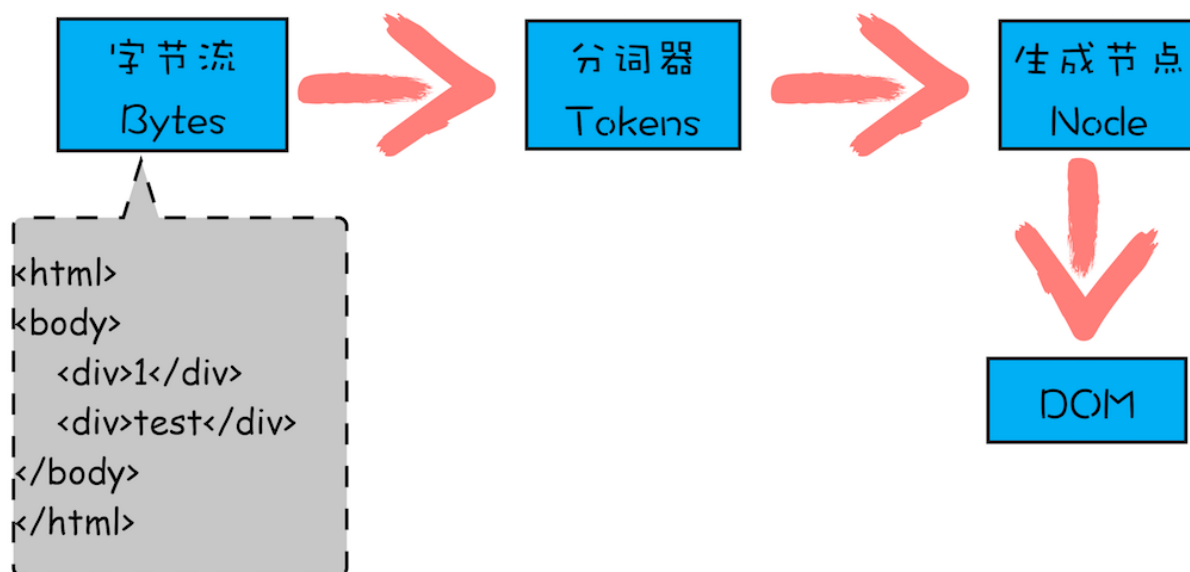
HTML 解析器是等整个 HTML 文档加载完成之后开始解析的，还是随着 HTML 文档边加载边解析的？

在这里我统一解答下，HTML 解析器并不是等整个文档加载完成之后再解析的，而是**网络进程加载了多少数据，HTML 解析器便解析多少数据**。

那详细的流程是怎样的呢？网络进程接收到响应头之后，会根据响应头中的 content-type 字段来判断文件的类型，比如 content-type 的值是 “text/html”，那么浏览器就会判断这是一个 HTML 类型的文件，然后为该请求选择或者创建一个渲染进程。渲染进程准备好之后，**网络进程和渲染进程之间会建立一个共享数据的管道**，网络进程接收到数据后就往这个管道里面放，而渲染进程则从管道的另外一端不断地读取数据，并同时读取的数据“喂”给 HTML 解析器。你可以把这个管道想象成一个“水管”，网络进程接收到的字节流像水一样倒进这个“水管”，而“水管”的另外一端是渲染进程的 HTML 解析器，它会动态接收字节流，并将其解析为 DOM。

解答完这个问题之后，接下来我们就可以来详细聊聊 DOM 的具体生成流程了。

前面我们说过代码从网络传输过来是字节流的形式，那么后续字节流是如何转换为 DOM 的呢？你可以参考下图：



字节流转换为 DOM

从图中你可以看出，字节流转换为 DOM 需要三个阶段。

第一个阶段，通过分词器将字节流转换为 Token。

前面 [《14 | 编译器和解释器：V8 是如何执行一段 JavaScript 代码的？》](#) 文章中我们介绍过，V8 编译 JavaScript 过程中的第一步是做词法分析，将 JavaScript 先分解为一个个 Token。解析 HTML 也是一样的，需要通过分词器先将字节流转换为一个个 Token，分为 Tag Token 和文本 Token。上述 HTML 代码通过词法分析生成的 Token 如下所示：



生成的 Token 示意图

由图可以看出，Tag Token 又分 StartTag 和 EndTag，比如<body>就是 StartTag，</body>就是 EndTag，分别对于图中的蓝色和红色块，文本 Token 对应的绿色块。

至于后续的第二个和第三个阶段是同步进行的，需要将 Token 解析为 DOM 节点，并将 DOM 节点添加到 DOM 树中。

HTML 解析器维护了一个**Token 栈结构**，该 Token 栈主要用来计算节点之间的父子关系，在第一个阶段中生成的 Token 会被按照顺序压到这个栈中。具体的处理规则如下所示：


如果压入到栈中的是**StartTag Token**，HTML 解析器会为该 Token 创建一个 DOM 节点，然后将该节点加入到 DOM 树中，它的父节点就是栈中相邻的那个元素生成的节点。

如果分词器解析出来是**文本 Token**，那么会生成一个文本节点，然后将该节点加入到 DOM 树中，文本 Token 是不需要压入到栈中，它的父节点就是当前栈顶 Token 所对应的 DOM 节点。

如果分词器解析出来的是**EndTag 标签**，比如是 EndTag div，HTML 解析器会查看 Token 栈顶的元素是否是 StartTag div，如果是，就将 StartTag div 从栈中弹出，表示该 div 元素解析完成。

通过分词器产生的新 Token 就这样不停地压栈和出栈，整个解析过程就这样一直持续下去，直到分词器将所有字节流分词完成。

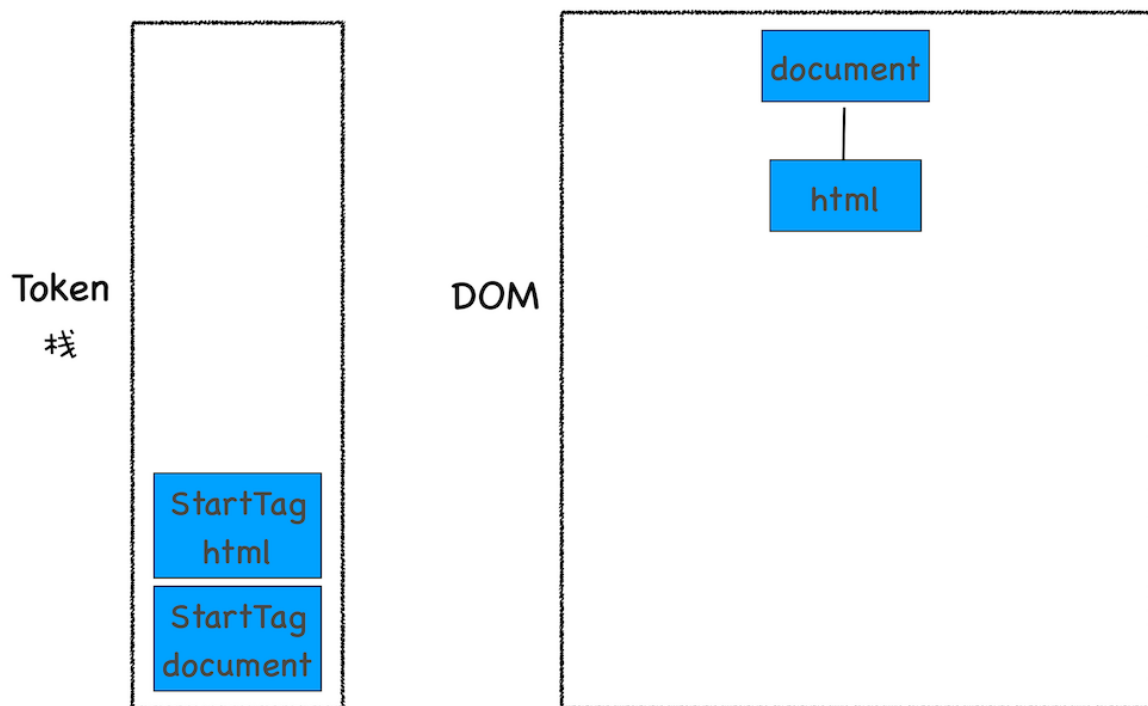
为了更加直观地理解整个过程，下面我们结合一段 HTML 代码（如下），来一步步分析 DOM 树的生成过程。

 复制代码

```
1 <html>
2 <body>
3     <div>1</div>
4     <div>test</div>
5 </body>
6 </html>
```

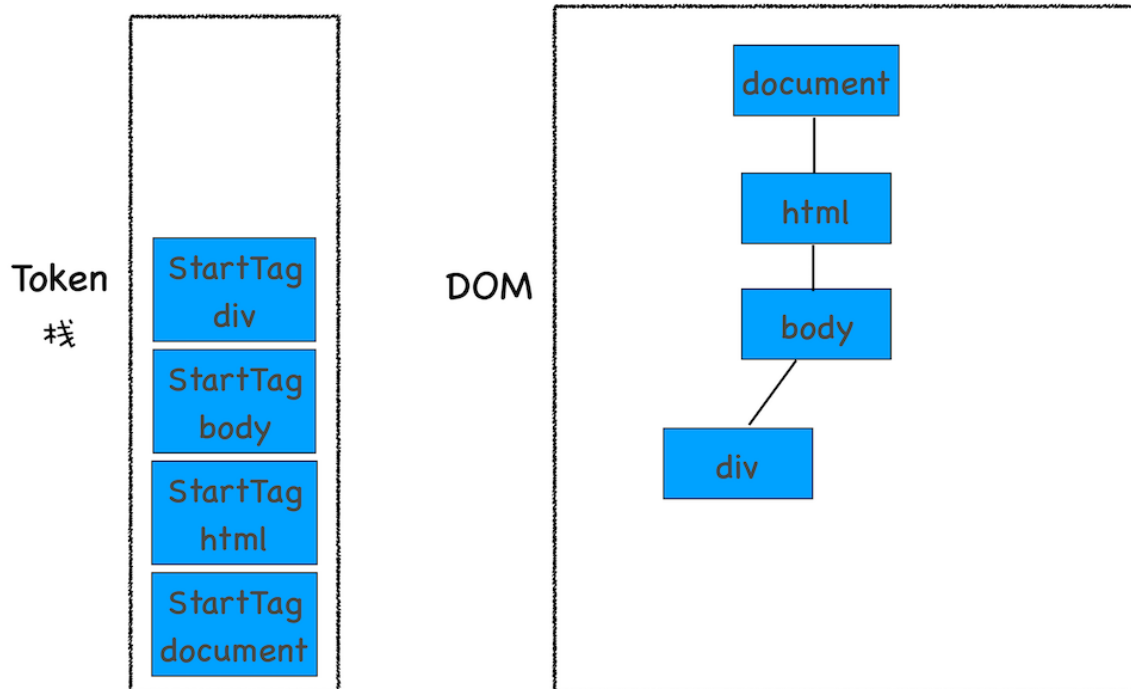
这段代码以字节流的形式传给了 HTML 解析器，经过分词器处理，解析出来的第一个 Token 是 StartTag html，解析出来的 Token 会被压入到栈中，并同时创建一个 html 的 DOM 节点，将其加入到 DOM 树中。

这里需要补充说明下，HTML 解析器开始工作时，会默认创建了一个根为 document 的空 DOM 结构，同时会将一个 StartTag document 的 Token 压入栈底。然后经过分词器解析出来的第一个 StartTag html Token 会被压入到栈中，并创建一个 html 的 DOM 节点，添加到 document 上，如下图所示：



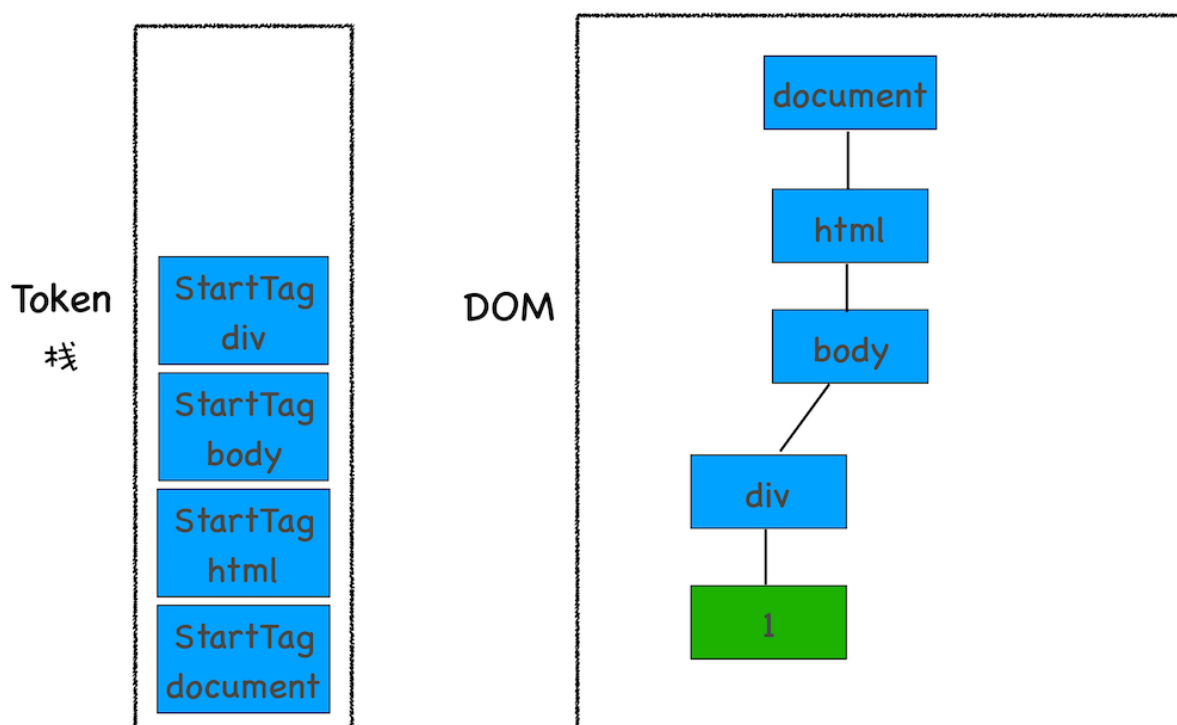
解析到 StartTag html 时的状态

然后按照同样的流程解析出来 StartTag body 和 StartTag div，其 Token 栈和 DOM 的状态如下图所示：



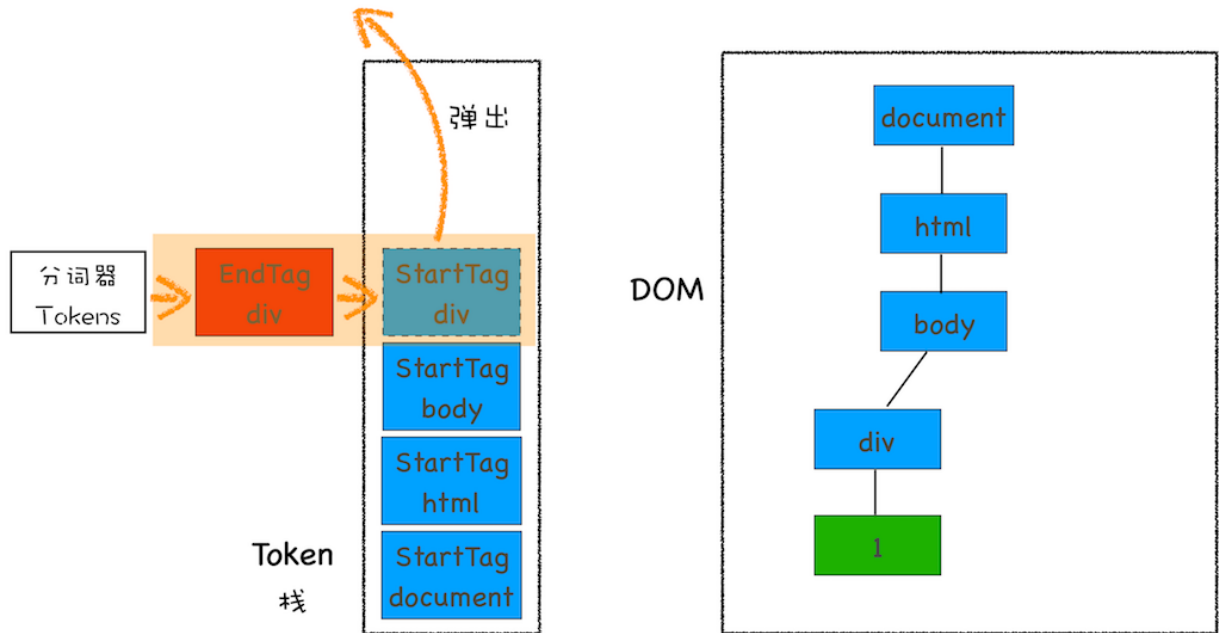
解析到 `StartTag div` 时的状态

接下来解析出来的是第一个 `div` 的文本 Token，渲染引擎会为该 Token 创建一个文本节点，并将该 Token 添加到 DOM 中，它的父节点就是当前 Token 栈顶元素对应的节点，如下图所示：



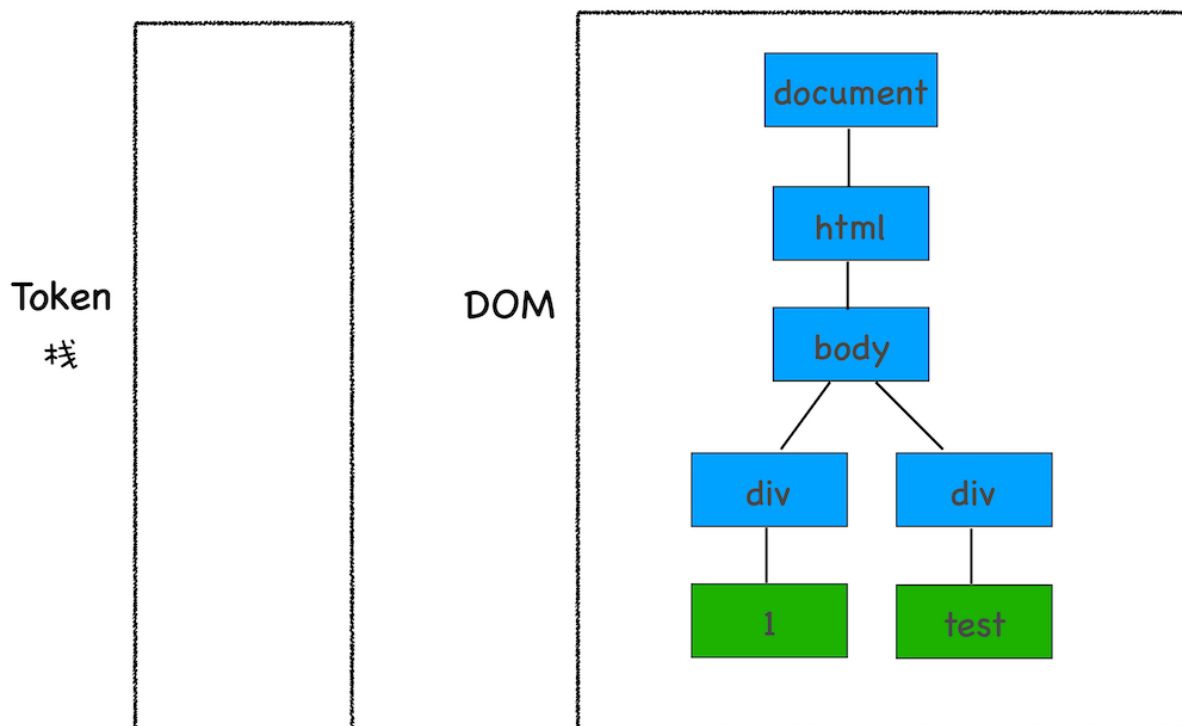
解析出第一个文本 Token 时的状态

再接下来，分词器解析出来第一个 EndTag div，这时候 HTML 解析器会去判断当前栈顶的元素是否是 StartTag div，如果是则从栈顶弹出 StartTag div，如下图所示：



元素弹出 Token 栈示意图

按照同样的规则，一路解析，最终结果如下图所示：




最终解析结果

通过上面的介绍，相信你已经清楚 DOM 是怎么生成的了。不过在实际生产环境中，HTML 源文件中既包含 CSS 和 JavaScript，又包含图片、音频、视频等文件，所以处理过程远比上面这个示范 Demo 复杂。不过理解了这个简单的 Demo 生成过程，我们就可以往下分析更加复杂的场景了。

JavaScript 是如何影响 DOM 生成的

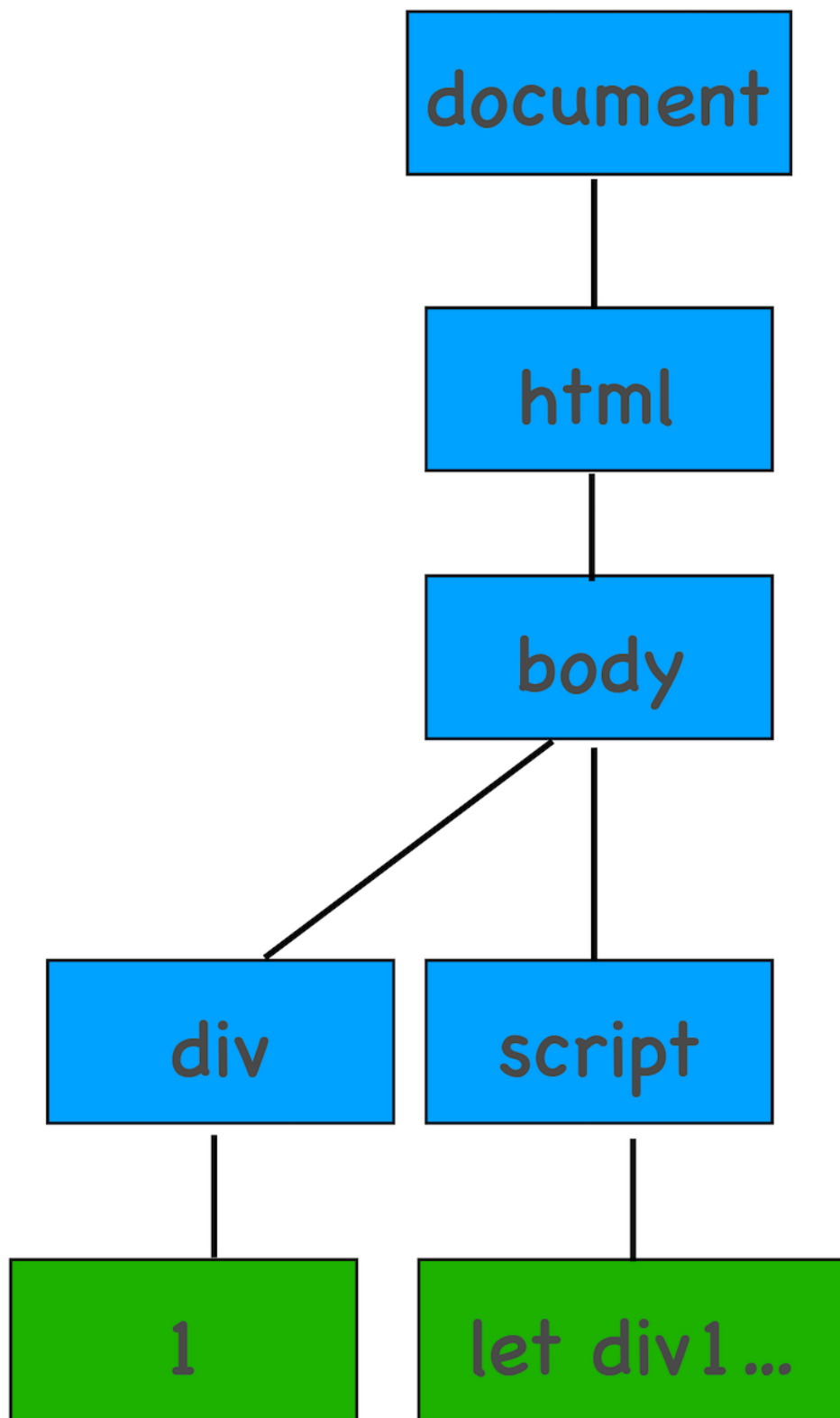
我们再来看看稍微复杂点的 HTML 文件，如下所示：

 复制代码

```
1 <html>
2 <body>
3   <div>1</div>
4   <script>
5     let div1 = document.getElementsByTagName('div')[0]
6     div1.innerText = 'time.geekbang'
7   </script>
8   <div>test</div>
9 </body>
10 </html>
```


我在两段 div 中间插入了一段 JavaScript 脚本，这段脚本的解析过程就有点不一样了。<script>标签之前，所有的解析流程还是和之前介绍的一样，但是解析到<script>标签时，渲染引擎判断这是一段脚本，此时 HTML 解析器就会暂停 DOM 的解析，因为接下来的 JavaScript 可能要修改当前已经生成的 DOM 结构。

通过前面 DOM 生成流程分析，我们已经知道当解析到 script 脚本标签时，其 DOM 树结构如下所示：




执行脚本时 DOM 的状态


这时候 HTML 解析器暂停工作，JavaScript 引擎介入，并执行 script 标签中的这段脚本，因为这段 JavaScript 脚本修改了 DOM 中第一个 div 中的内容，所以执行这段脚本之后，

div 节点内容已经修改为 time.geekbang 了。脚本执行完成之后，HTML 解析器恢复解析过程，继续解析后续的内容，直至生成最终的 DOM。

以上过程应该还是比较好理解的，不过除了在页面中直接内嵌 JavaScript 脚本之外，我们还通常需要在页面中引入 JavaScript 文件，这个解析过程就稍微复杂了些，如下面代码：

 复制代码

```
1 //foo.js
2 let div1 = document.getElementsByTagName('div')[0]
3 div1.innerText = 'time.geekbang'
```


 复制代码

```
1 <html>
2 <body>
3   <div>1</div>
4   <script type="text/javascript" src='foo.js'></script>
5   <div>test</div>
6 </body>
7 </html>
```

这段代码的功能还是和前面那段代码是一样的，不过这里我把内嵌 JavaScript 脚本修改成了通过 JavaScript 文件加载。其整个执行流程还是一样的，执行到 JavaScript 标签时，暂停整个 DOM 的解析，执行 JavaScript 代码，不过这里执行 JavaScript 时，需要先下载这段 JavaScript 代码。这里需要重点关注下载环境，因为**JavaScript 文件的下载过程会阻塞 DOM 解析**，而通常下载又是非常耗时的，会受到网络环境、JavaScript 文件大小等因素的影响。

不过 Chrome 浏览器做了很多优化，其中一个主要的优化是**预解析操作**。当渲染引擎收到字节流之后，会开启一个预解析线程，用来分析 HTML 文件中包含的 JavaScript、CSS 等相关文件，解析到相关文件之后，预解析线程会提前下载这些文件。

再回到 DOM 解析上，我们知道引入 JavaScript 线程会阻塞 DOM，不过也有一些相关的策略来规避，比如使用 CDN 来加速 JavaScript 文件的加载，压缩 JavaScript 文件的体积。另外，如果 JavaScript 文件中没有操作 DOM 相关代码，就可以将该 JavaScript 脚本设置为异步加载，通过 async 或 defer 来标记代码，使用方式如下所示：

 复制代码


```
1 <script async type="text/javascript" src='foo.js'></script>
```

 复制代码


```
1 <script defer type="text/javascript" src='foo.js'></script>
```

async 和 defer 虽然都是异步的，不过还有一些差异，使用 async 标志的脚本文件一旦加载完成，会立即执行；而使用了 defer 标记的脚本文件，需要在 DOMContentLoaded 事件之前执行。

现在我们知道了 JavaScript 是如何阻塞 DOM 解析的了，那接下来我们再来结合文中代码看看另外一种情况：

 复制代码

```
1 //theme.css
2 div {color:blue}
```

 复制代码

```
1 <html>
2   <head>
3     <style src='theme.css'></style>
4   </head>
5   <body>
6     <div>1</div>
7     <script>
8       let div1 = document.getElementsByTagName('div')[0]
9       div1.innerText = 'time.geekbang' // 需要 DOM
10      div1.style.color = 'red' // 需要 CSSOM
11    </script>
12    <div>test</div>
13 </body>
14 </html>
```

该示例中，JavaScript 代码出现了 `div1.style.color = 'red'` 的语句，它是用来操纵 CSSOM 的，所以在执行 JavaScript 之前，需要先解析 JavaScript 语句之上所有的 CSS 样式。所以如果代码里引用了外部的 CSS 文件，那么在执行 JavaScript 之前，还需要等待外部的 CSS 文件下载完成，并解析生成 CSSOM 对象之后，才能执行 JavaScript 脚本。

而 JavaScript 引擎在解析 JavaScript 之前，是不知道 JavaScript 是否操纵了 CSSOM 的，所以渲染引擎在遇到 JavaScript 脚本时，不管该脚本是否操纵了 CSSOM，都会执行 CSS 文件下载，解析操作，再执行 JavaScript 脚本。

所以说 JavaScript 脚本是依赖样式表的，这又多了一个阻塞过程。至于如何优化，我们在下篇文章中再来深入探讨。

通过上面的分析，我们知道了 JavaScript 会阻塞 DOM 生成，而样式文件又会阻塞 JavaScript 的执行，所以在实际的工程中需要重点关注 JavaScript 文件和样式表文件，使用不当会影响到页面性能的。

总结

好了，今天就讲到这里，下面我来总结下今天的内容。

首先我们介绍了 DOM 是如何生成的，然后又基于 DOM 的生成过程分析了 JavaScript 是如何影响到 DOM 生成的。因为 CSS 和 JavaScript 都会影响到 DOM 的生成，所以我们又介绍了一些加速生成 DOM 的方案，理解了这些，能让你更加深刻地理解如何去优化首次页面渲染。

额外说明一下，渲染引擎还有一个安全检查模块叫 XSSAuditor，是用来检测词法安全的。在分词器解析出来 Token 之后，它会检测这些模块是否安全，比如是否引用了外部脚本，是否符合 CSP 规范，是否存在跨站点请求等。如果出现不符合规范的内容，XSSAuditor 会对该脚本或者下载任务进行拦截。详细内容我们会在后面的安全模块介绍，这里就不赘述了。

思考时间

看下面这样一段代码，你认为打开这个 HTML 页面，页面显示的内容是什么？

```

1 <html>
2 <body>
3   <div>1</div>
4   <script>
5     let div1 = document.getElementsByTagName('div')[0]
6     div1.innerText = 'time.geekbang'
7
8     let div2 = document.getElementsByTagName('div')[1]
9     div2.innerText = 'time.geekbang.com'
10  </script>
11  <div>test</div>
12 </body>
13 </html>

```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。



浏览器工作原理与实践

>>> 透过浏览器看懂前端本质

李兵

前盛大创新院高级研究员



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 21 | Chrome开发者工具：利用网络面板做性能分析

下一篇 23 | 渲染流水线：CSS如何影响首次加载时的白屏时间？

精选留言 (16)

写留言



Luke

2019-09-25

CSS不阻塞dom的生成。

CSS不阻塞js的加载，但是会阻塞js的执行。

js会阻塞dom的生成，也就是会阻塞页面的渲染，那么css也有可能阻塞页面的渲染。

如果把CSS放在文档的最后面加载执行，CSS不会阻塞DOM的生成，也不会阻塞JS，但是浏览器在解析完DOM后，要花费额外时间来解析CSS，而不是在解析DOM的时候，并行...

展开 ∨

1

6



Angus

2019-09-24

会显示time.geekbang和test，JavaScript代码执行的时候第二个div还没有生成DOM节点，所以是获取不到div2的，页面会报错Uncaught TypeError: Cannot set property 'innerText' of undefined。

另外复习了下async和defer： ...

展开 ∨

1

4



mfist

2019-09-24

开始看文章的时候就在想如果js获取的dom还没有解析出来，会如何处理，结果思考题就是这个。

会两行显示，一行是time.geekbang 另外一行是test。原因是script脚本执行的时候获取想不到第二个div，所以不会对后来的div有影响。 ...

展开 ∨

1

3



Peter Cheng

2019-09-25

针对文章中js和css加载我有一个疑问。

<head>

<link ref="a.css">

<script src="b.js"></script>

<link ref="c.css">...

展开 ▾



1



王大可

2019-09-24

time.geekbang

test

把script标签包裹的代码放入一个js文件中，在引入该文件

1. 放入第一个div之前页面显示

1...

展开 ▾



1



蓝配鸡

2019-09-28

思考题看法：

会生成

time.geekbang

Test

...

展开 ▾



木瓜777

2019-09-27

您好，网络进程接收到响应头之后，会根据请求头中的 content-type 字段来判断文件的类型，比如 content-type 的值是 "text/html" ！

这个地方应该是根据响应头判断文件类型吧？

作者回复： 嗯 是响应头，我改过来



叫我大胖就好了

2019-09-26

我看MDN写的是defer在DOMContentLoaded 前执行

展开 ▾

作者回复： 你是对的，我写错了。



Hurry

2019-09-26

Token 栈，遇到结束标签，但是栈顶刚好不是对应开始标签，这种错误，解析器，如何处理？



充电中

2019-09-25

2点疑惑望解答

1.既然是先进行html解析，那么当html解析器遇到style标签和行内样式时，会怎么做，是一个怎样的流程，或者说什么时候进行样式计算呢

2.文中有一段是这样说的：而 JavaScript 引擎在解析 JavaScript 之前，是不知道 JavaScript 是否操纵了 CSSOM 的，所以渲染引擎在遇到 JavaScript 脚本时，不管该脚本是否操...

展开 ∨



歌在云端

2019-09-25

time.geekbang 和 test

展开 ∨



ytd

2019-09-25

第一行是time.geekbang，第二行不会变，仍是test。原因就是浏览器是边加载边解析html的，而且遇到js会停止dom的解析执行js，js执行完毕后再接着解析dom。上面的代码，js执行时第2个div并未被解析为dom，所以js中获取不到，js会抛出错误TypeError，但js抛出错误并未影响html的继续解析。所以，第2个div保持原来的状态被解析出来。

展开 ∨



HB

2019-09-24

每节课都能学到东西，如果能更新快一点就好了。比如二四六改成一三五七



柒月



2019-09-24

time.geekbang

test

get了，一直以为CSS的解析不会阻塞DOM的解析呢

展开 ▾



Geek_Jamorex

2019-09-24

老师，上面defer那里有点问题

实测defer会阻塞DOMContentLoaded事件，也就是DOMContentLoaded事件之后触发defer

async不会，两个都会阻塞load事件



Chao

2019-09-24

浏览器管道化解析，需要chunked的支持吗？

展开 ▾

