Chrome DevTools
性能

安佳

FE@搜索前端

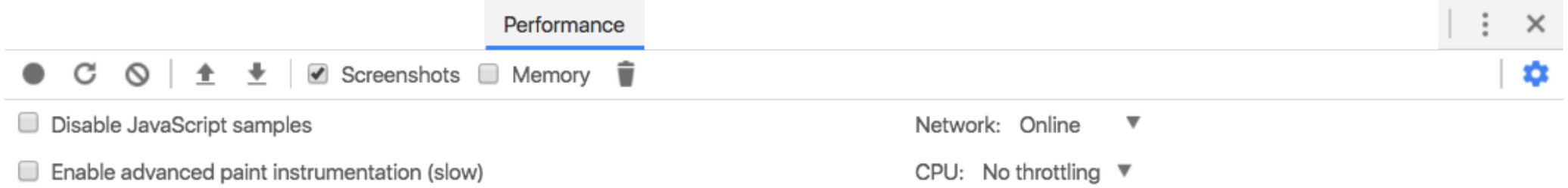
2017.11.16

github.com/anjia

Google Chrome

Google Chrome 已是最新版本

版本 62.0.3202.94（正式版本） （64 位）



Click the record button ● or hit ⌘ **E** to start a new recording.

Click the reload button ↻ or hit ⌘ ⇧ **E** to record the page load.

After recording, select an area of interest in the overview by dragging.
Then, zoom and pan the timeline with the mousewheel or **WASD** keys.
Learn more

The **Performance** panel provides the combined functionality of Timeline and JavaScript CPU profiler. Learn more

The JavaScript CPU profiler will be removed shortly. Meanwhile, it's available under ⋮ → More Tools → JavaScript Profiler.

Performance

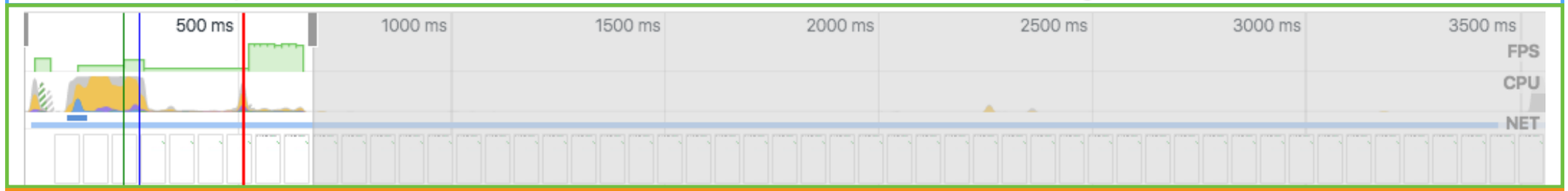
Screenshots Memory

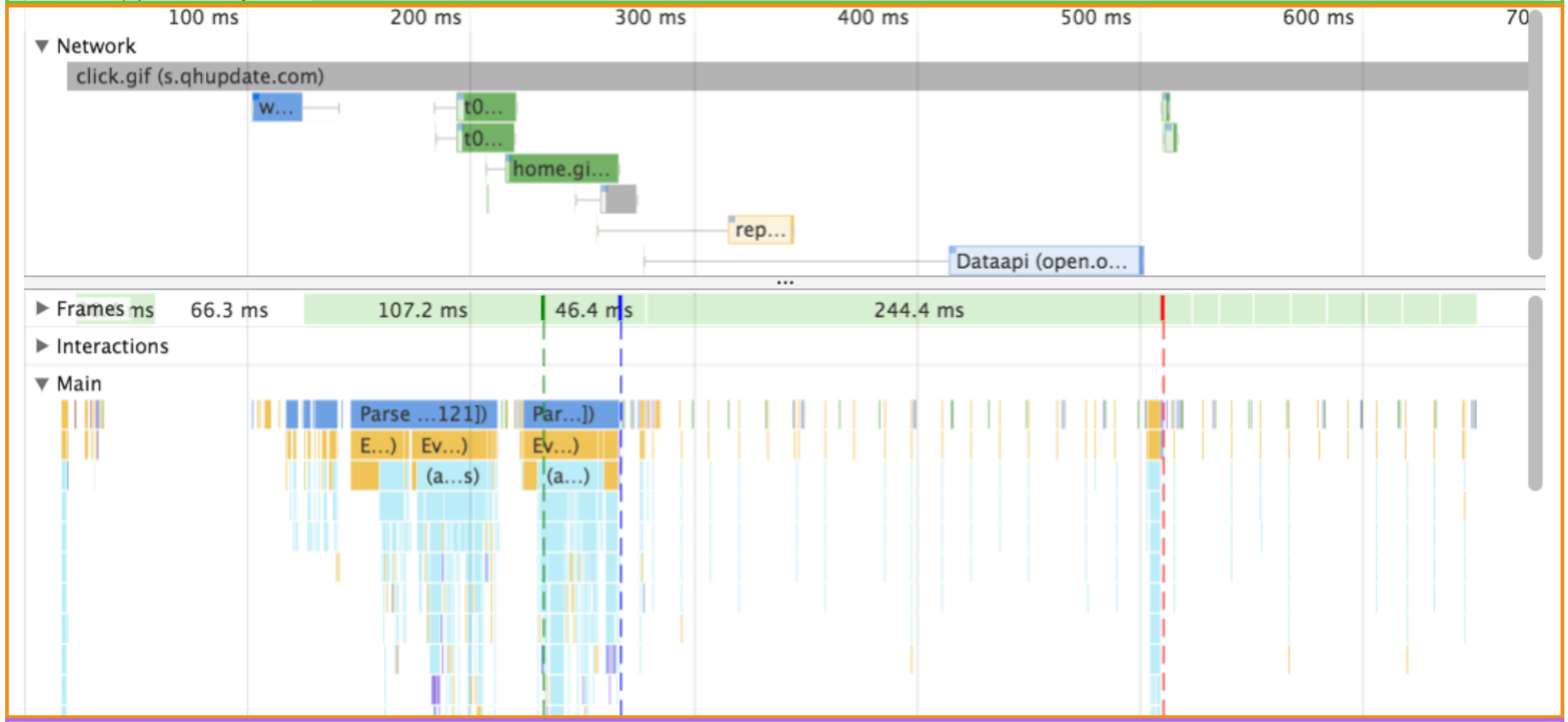
Disable JavaScript samples

Enable advanced paint instrumentation (slow)

Network: Online

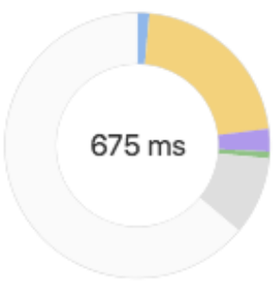
CPU: No throttling





Summary Bottom-Up Call Tree Event Log

Range: 0 – 675 ms



9.9 ms	Loading
145.1 ms	Scripting
18.7 ms	Rendering
5.5 ms	Painting
64.3 ms	Other
431.1 ms	Idle

目录

- Performance 面板的使用
- 分析数据：浏览器的工作原理 & DevTools
- 常见的性能问题及解决思路

演示

DevTools/Performance 面板的使用

使用手册

注意事项

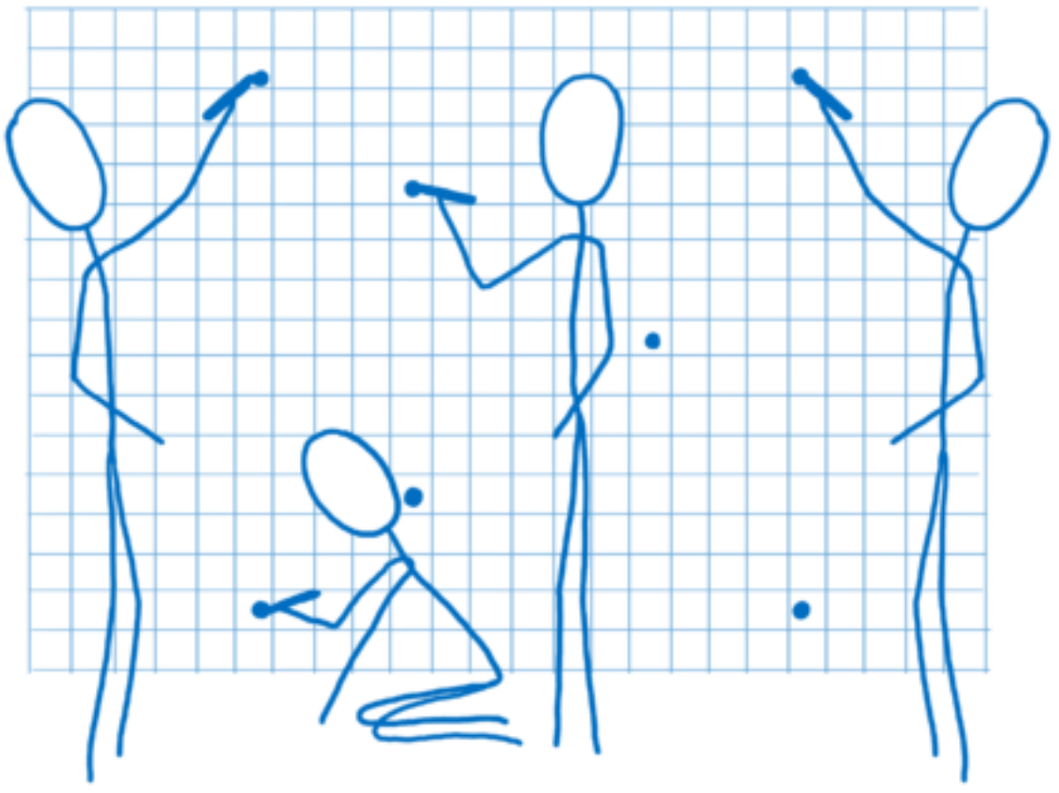
- ✓ 去除其它干扰
  - 去除所有扩展：禁用所有扩展 / 隐身模式
  - 避免无关&不必要的操作
- ✓ 模拟首次访问
  - Network 禁用缓存
  - Clear Storage
- ✓ 记录时长：尽量简短，2~4s
- ✓ 节流：CPU、网络

练习

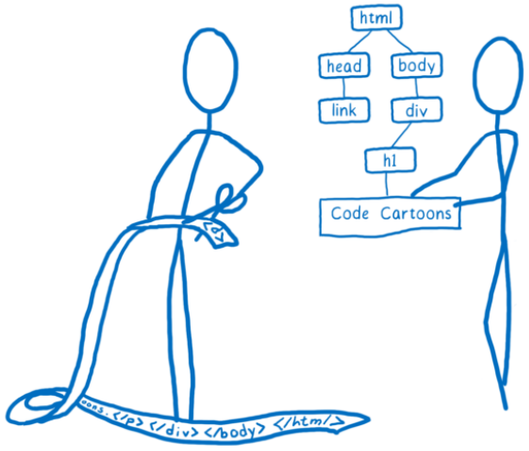
找一个熟悉的网站，尽量复杂点，
实战使用手册中提到的各个部分

分析数据

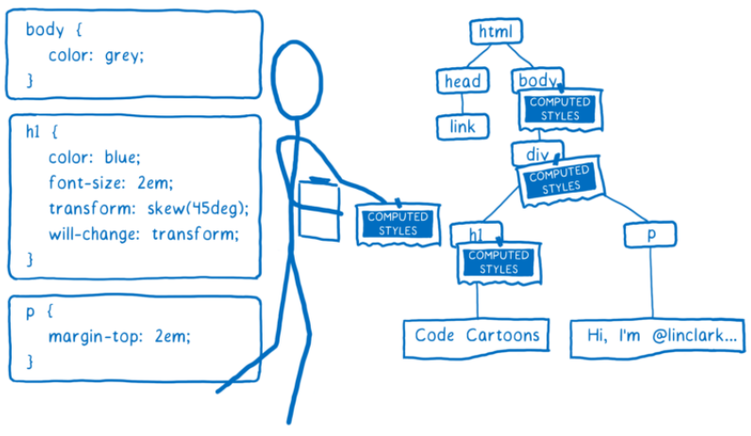
- 浏览器：如何将 HTML、CSS、JS 解析成页面上的像素点的



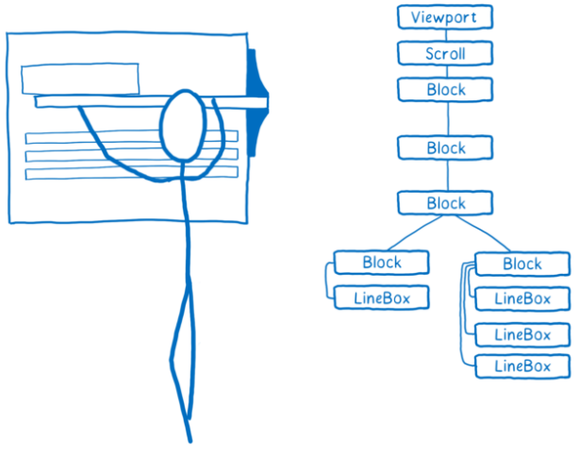
PARSE



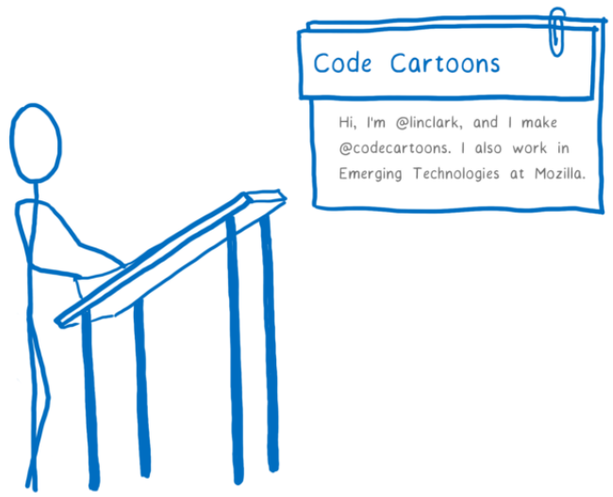
STYLE



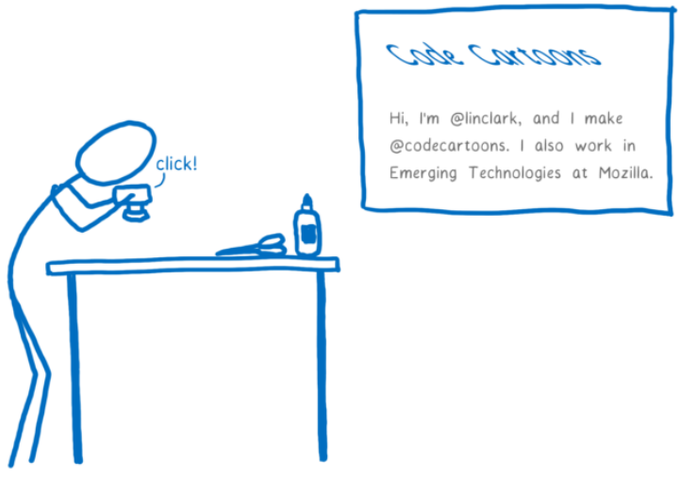
LAYOUT

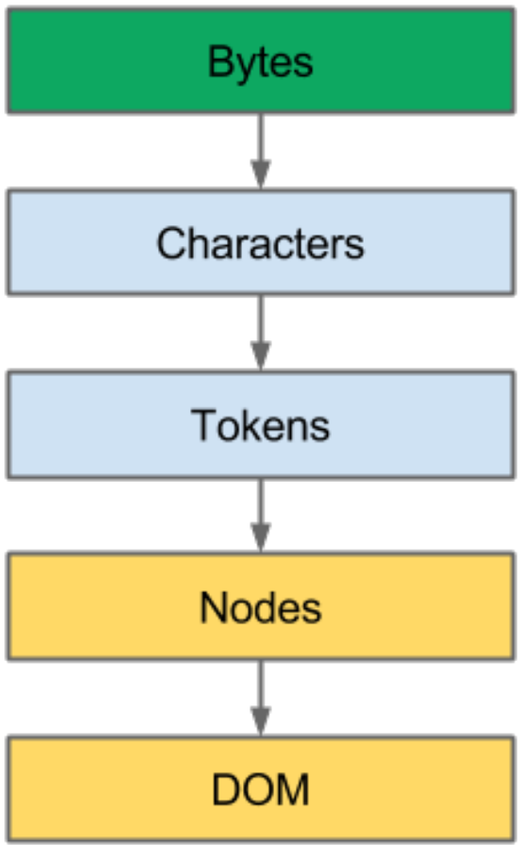


PAINT



COMPOSITE & RENDER



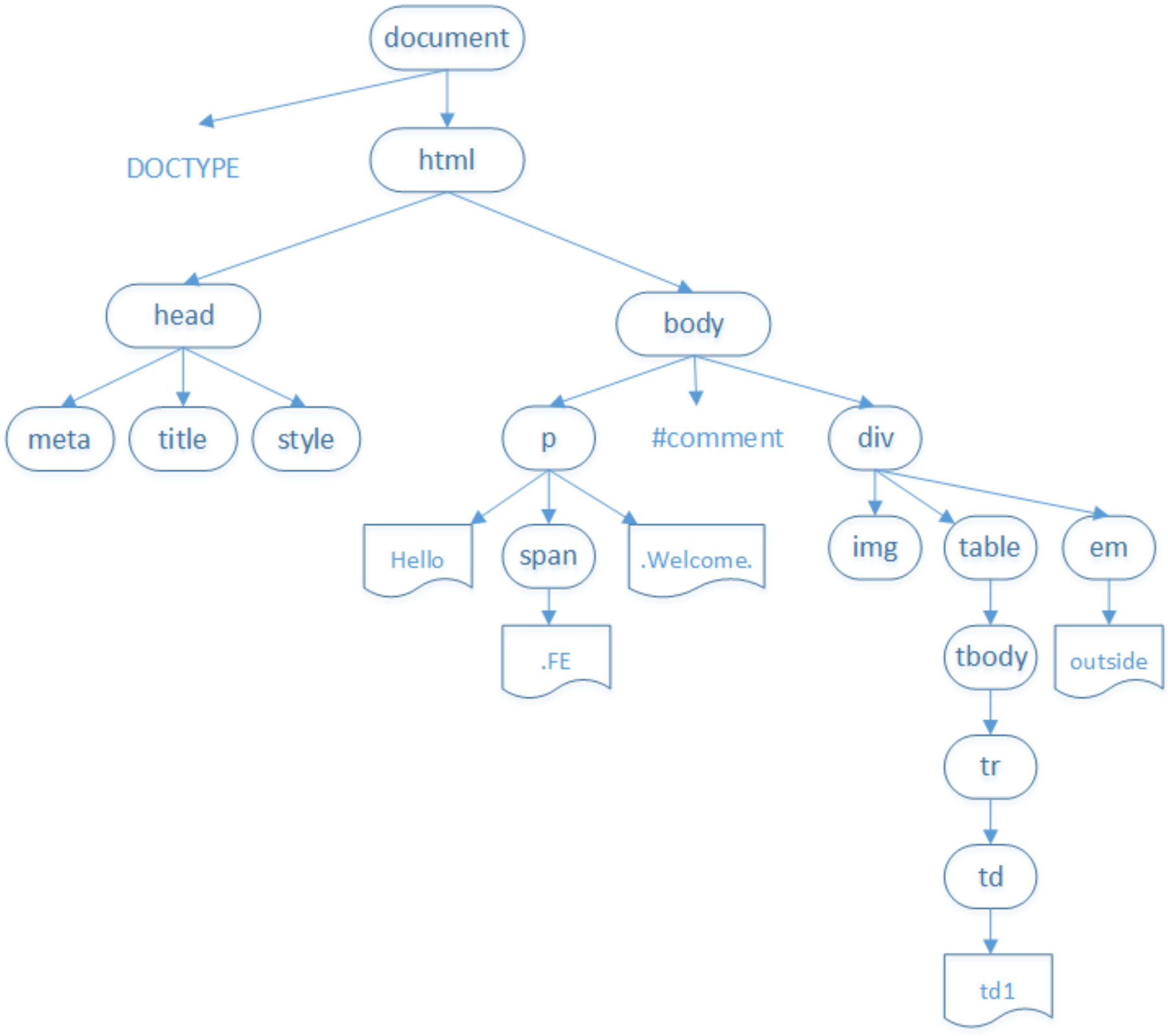

Bytes
Characters
Tokens
Nodes
DOM

练习

2. 画出示例页面的DOM树

示例页面

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>DevTools</title>
    <style>
        body {
            font-size: 16px;
        }
        p {
            font-weight: bold;
        }
        span {
            color: red;
            font-size: 25px;
        }
        p span {
            display: none;
        }
        img {
            float: right;
        }
        table {
            display: none;
        }
    </style>
</head>
<body>
    <p>
        Hello<span>,FE</span>.Welcome.
    </p>
    <!-- pic: hello world -->
    <div>
        <img src="img/hello.jpeg" width="320" height="200">
    <table>
        <td>td1</td>
    </table>
</body>
    <em>outside</em>
</html>
```


document
DOCTYPE
html
head
body
meta
title
style
p
#comment
div
Hello
span
.Welcome.
.FE
img
table
em
tbody
outside
tr
td
td1


Bytes
3C 62 6F 64 79 3E 48 65 6C 6C 6F 2C 20 3C 73 70 61 6E 3E 77 6F 72 6C 64 21 3C 2F 73 70 61 6E 3E 3C 2F 62 6F 64 79 3E
Characters
<html><head>...</head><body><p>Hello web performance...
Tokens
StartTag: html
StartTag: head
...
EndTag: head
StartTag: body
StartTag: p
Hello
...
Nodes
DOM

Tokens

6种类型的Tokens

DOCTYPE

start tag

attributes

end tag

comment

character

end-of-file

tokenization

start tag
end tag
text
comment

1. Void elements

area, base, br, col, embed, hr, img, input, keygen, link, meta, param, source, track, wbr

只有 start tag

2. Raw text elements

script, style

start tag 和 end tag

3. Escapable raw text elements

textarea, title

start tag 和 end tag

4. Foreign elements

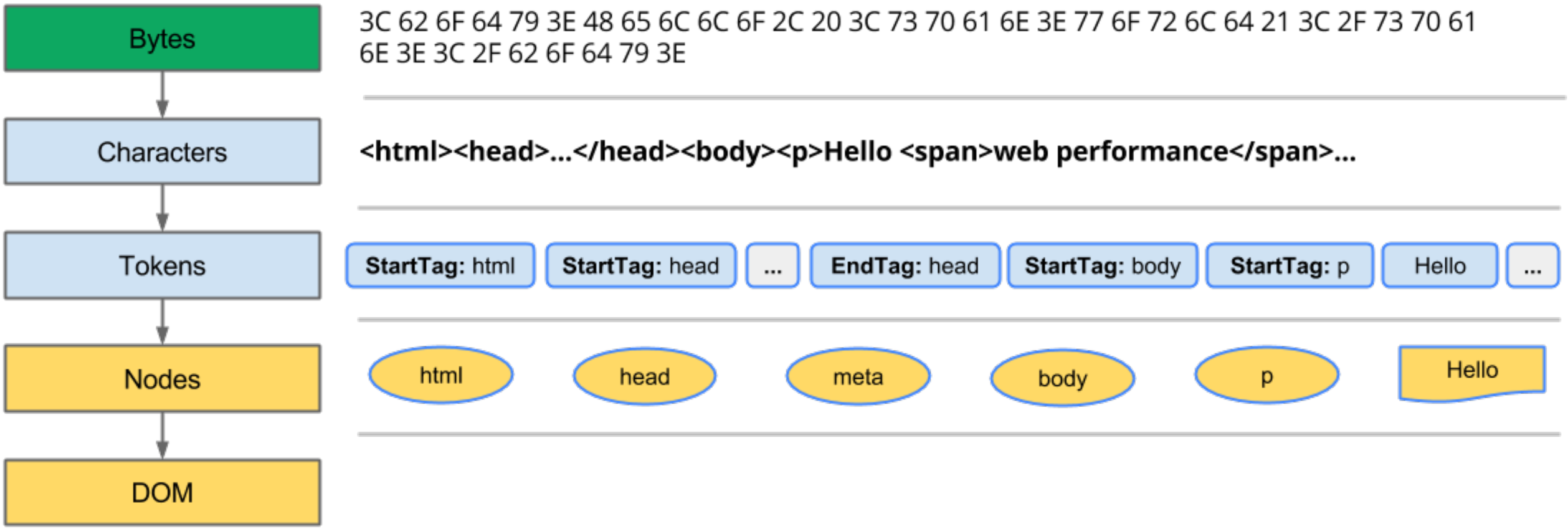
elements from the MathML namespace and the SVG namespace.

start tag 和 end tag | 只有自闭合的 start tag

5. Normal elements

all other allowed HTML elements

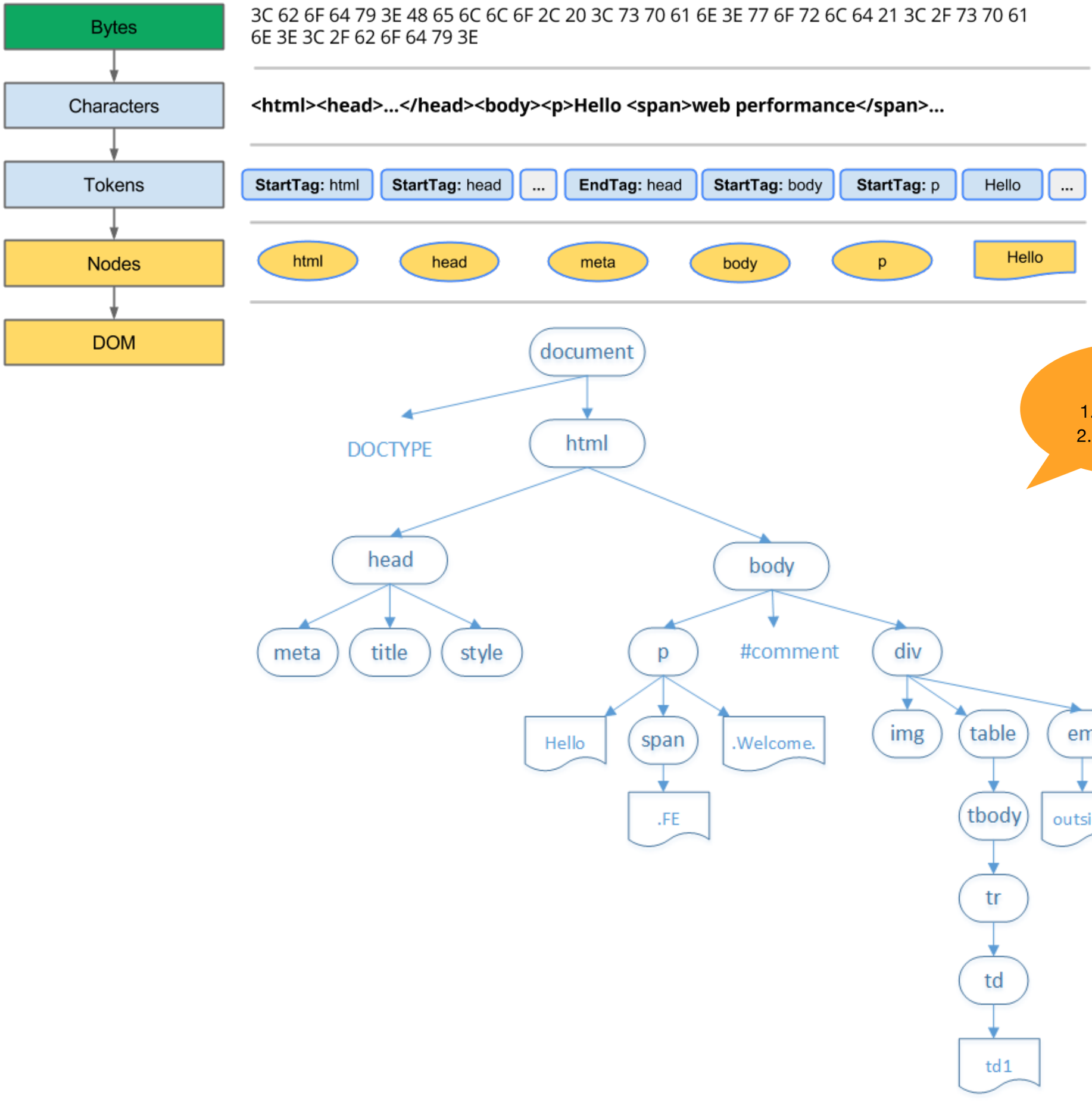
start tag 和 end tag | 特定情形下可以省略tag


Bytes
3C 62 6F 64 79 3E 48 65 6C 6C 6F 2C 20 3C 73 70 61 6E 3E 77 6F 72 6C 64 21 3C 2F 73 70 61 6E 3E 3C 2F 62 6F 64 79 3E
Characters
<html><head>...</head><body><p>Hello web performance...
Tokens
StartTag: html
StartTag: head
...
EndTag: head
StartTag: body
StartTag: p
Hello
...
Nodes
html
head
meta
body
p
Hello
DOM

12－3=9种节点

```
interface Node : EventTarget {
  const unsigned short ELEMENT_NODE = 1;
  const unsigned short ATTRIBUTE_NODE = 2;
  const unsigned short TEXT_NODE = 3;
  const unsigned short CDATA_SECTION_NODE = 4;
  const unsigned short ENTITY_REFERENCE_NODE = 5; // historical
  const unsigned short ENTITY_NODE = 6; // historical
  const unsigned short PROCESSING_INSTRUCTION_NODE = 7;
  const unsigned short COMMENT_NODE = 8;
  const unsigned short DOCUMENT_NODE = 9;
  const unsigned short DOCUMENT_TYPE_NODE = 10;
  const unsigned short DOCUMENT_FRAGMENT_NODE = 11;
  const unsigned short NOTATION_NODE = 12; // historical
  readonly attribute unsigned short nodeType;
  readonly attribute DOMString nodeName;
```

Node


Bytes
Characters
Tokens
Nodes
DOM
3C 62 6F 64 79 3E 48 65 6C 6C 6F 2C 20 3C 73 70 61 6E 3E 77 6F 72 6C 64 21 3C 2F 73 70 61 6E 3E 3C 2F 62 6F 64 79 3E
<html><head>...</head><body><p>Hello web performance...
StartTag: html
StartTag: head
...
EndTag: head
StartTag: body
StartTag: p
Hello
...
html
head
meta
body
p
Hello
document
DOCTYPE
html
head
meta
title
style
body
p
#comment
div
Hello
span
.Welcome.
.FE
img
table
em
tbody
outside
tr
td
td1
省略了
1.空格 换行
2. Attr Node

```
DOCTYPE: html
HTML lang="en"
  HEAD
    #text:
    META charset="UTF-8"
    #text:
    TITLE
      #text: DevTools
    #text:
    STYLE
      #text: body { font-size: 16px; } p { font-weight: bold; } span { color: red; font-size: 25px; } p span { display: none; } img { float: right; }
    #text:
  #text:
  BODY
    #text:
    P
      #text: Hello
      SPAN
        #text: ,FE
      #text: .Welcome.
    #text:
    #comment: pic: hello world
    #text:
    DIV
      #text:
      IMG src="img/hello.jpeg" width="320" height="200"
      #text:
      TABLE
        #text:
        TBODY
          TR
            TD
              #text: td1
            #text:
      #text:
      EM
        #text: outside
      #text:
```

空格和回车：有效的 Text Node

在顶级，有两种特殊情况：

1. <head>之前的，会被忽略
2. </body>之后的，会被自动移动到<body>里面

看demo

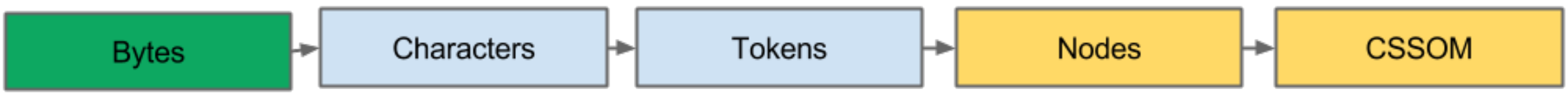
Live DOM Viewer

关于DOM

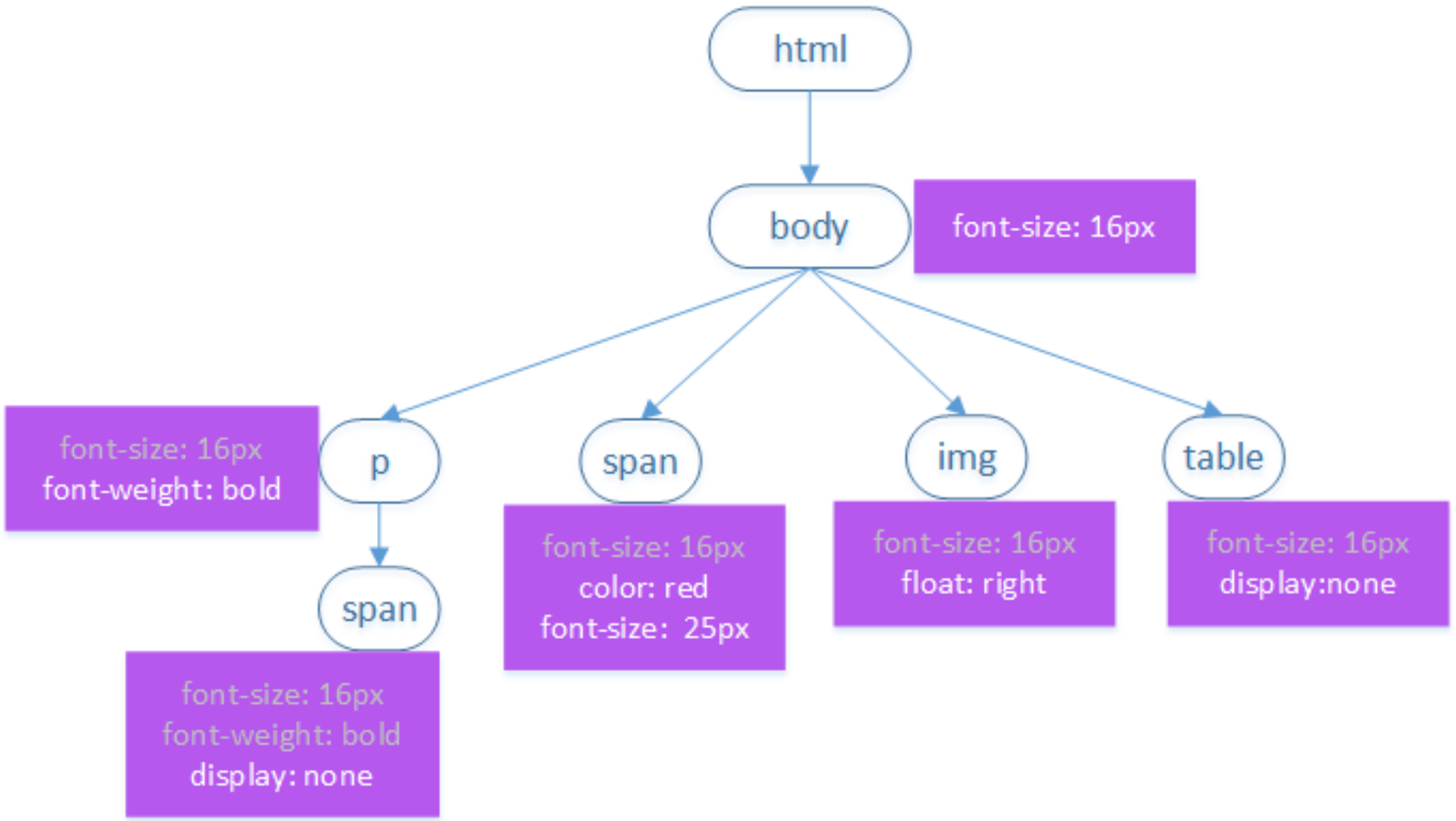
- 根元素是 document
- 规则：只要在.html文档里的，必须在DOM树里
  - eg. DOCTYPE, 注释
- 浏览器自动更正：自动补全、自动闭合、标签嵌套
  - 可以合法省略的标签
  - 规定：内容都在<body>里

练习

2. 续：画出CSSOM树



```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>DevTools</title>
    <style>
        body {
            font-size: 16px;
        }
        p {
            font-weight: bold;
        }
        span {
            color: red;
            font-size: 25px;
        }
        p span {
            display: none;
        }
        img {
            float: right;
        }
        table {
            display: none;
        }
    </style>
</head>
<body>
    <p>
        Hello<span>,FE</span>.Welcome.
    </p>
    <!-- pic: hello world -->
    <div>
        <img src="img/hello.jpeg" width="320" height="200">
    <table>
        <td>td1</td>
    </table>
</body>
    <em>outside</em>
</html>
```

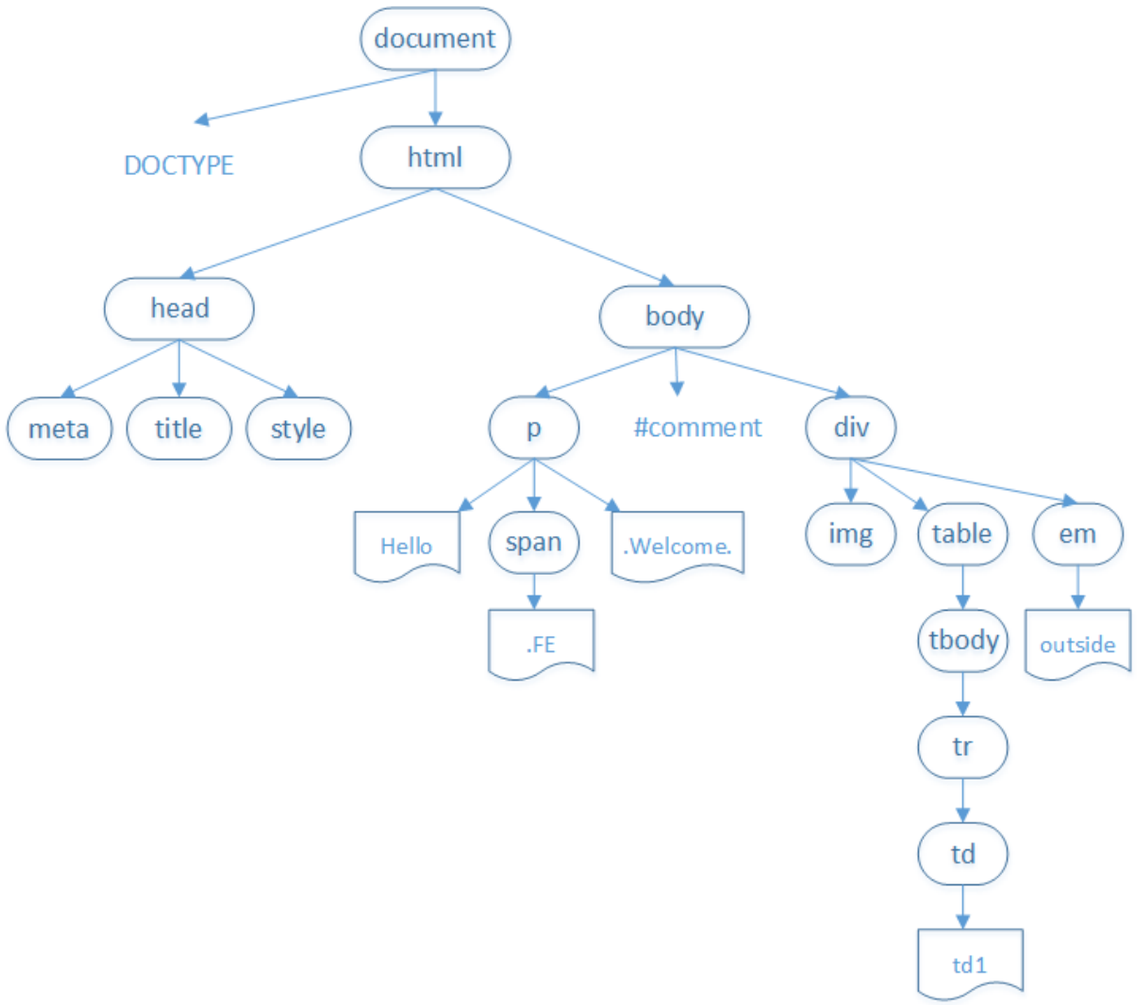

html
body
font-size: 16px
font-size: 16px
font-weight: bold
p
span
font-size: 16px
font-weight: bold
display: none
span
font-size: 16px
color: red
font-size: 25px
img
font-size: 16px
float: right
table
font-size: 16px
display:none

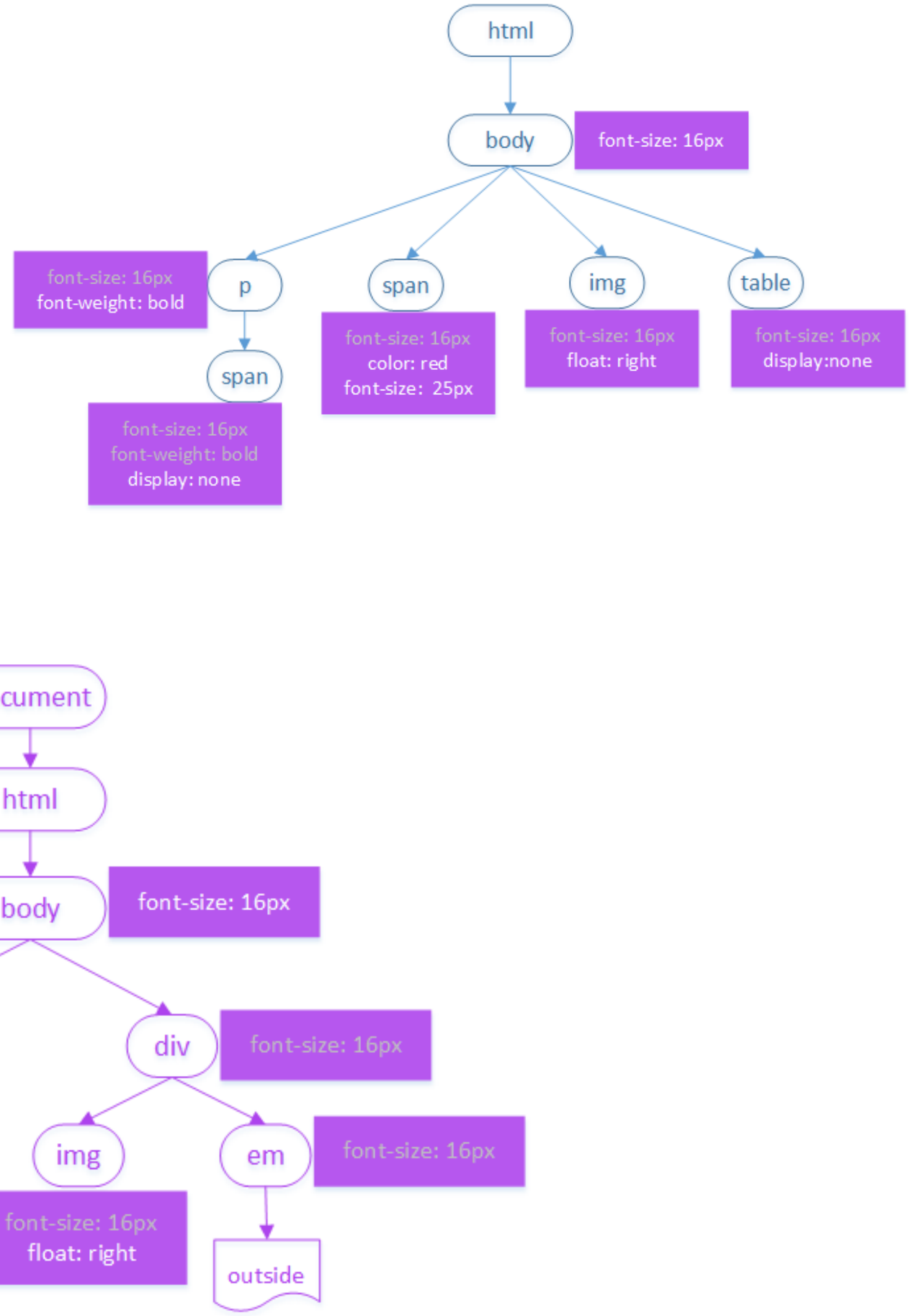
练习

2. 续：画出 RenderTree

步骤：

1. 从DOM树的根节点开始，遍历每一个“可见节点”
2. 对于每个可见的节点，在CSSOM树中找到对应的规则，并应用它们
3. 对于所有可见的节点，计算好内容和样式

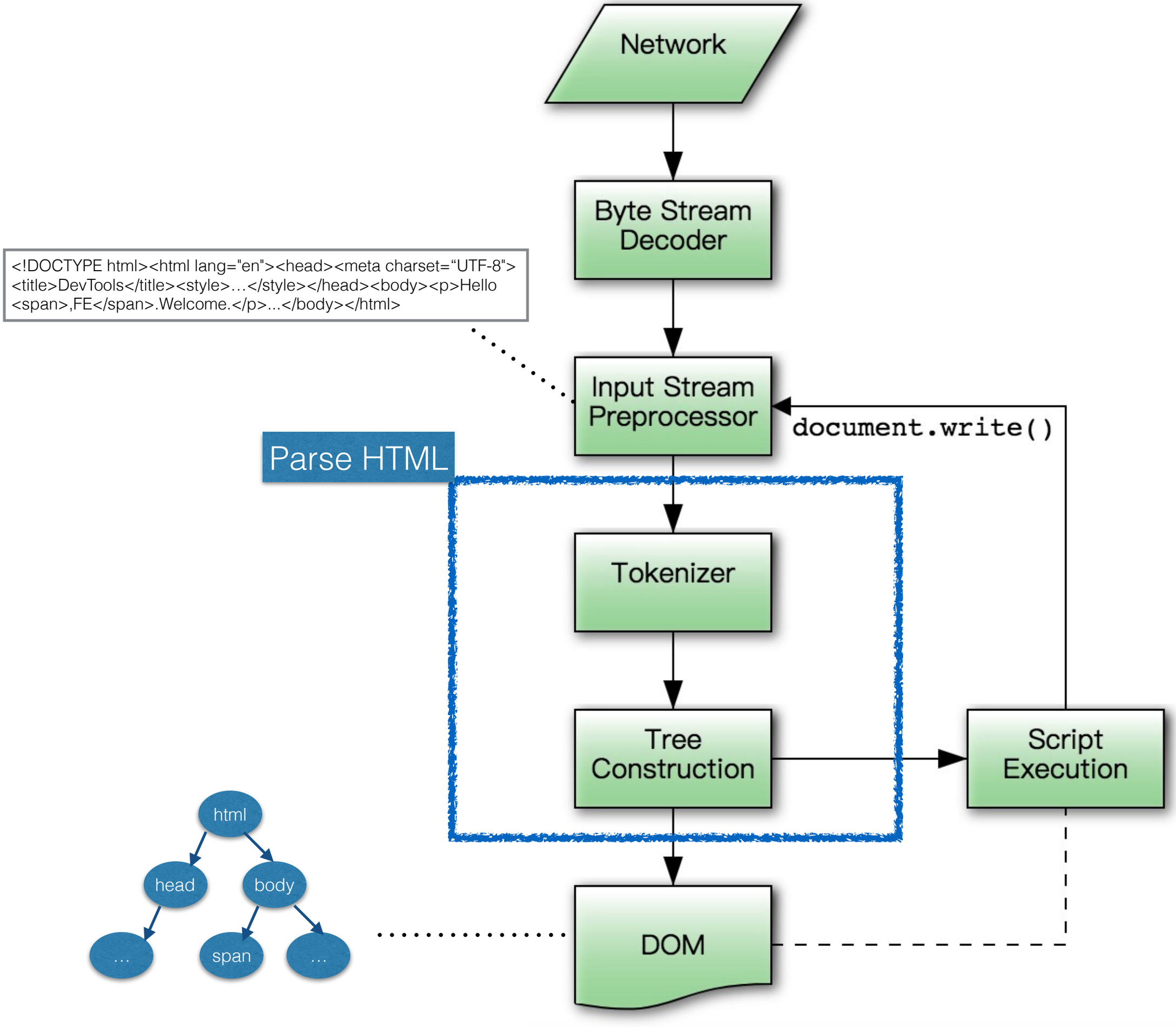

document
DOCTYPE
html
head
body
meta
title
style
p
#comment
div
Hello
span
.Welcome.
.FE
img
table
em
tbody
outside
tr
td
td1

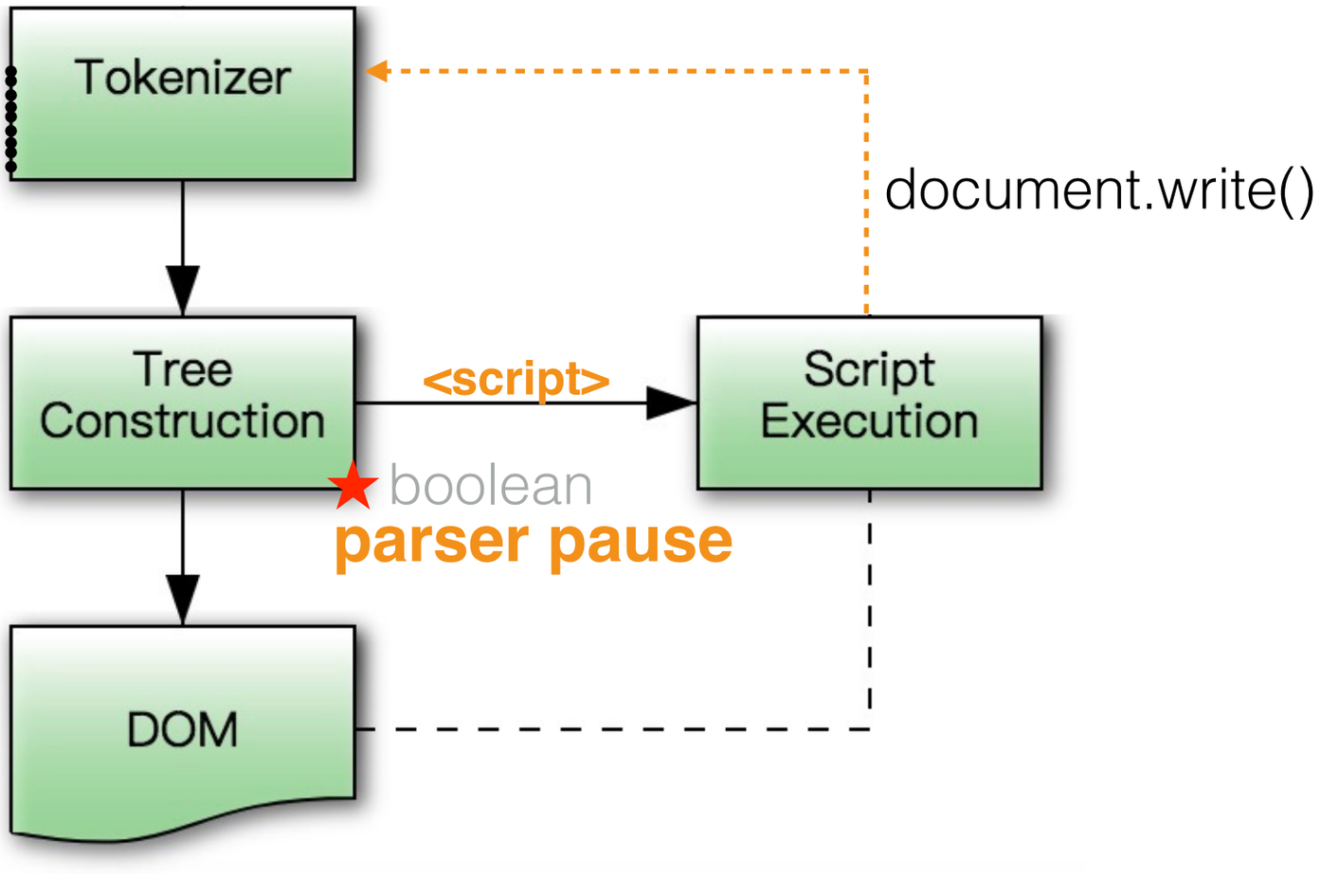

html
body
font-size: 16px
font-size: 16px
font-weight: bold
p
span
font-size: 16px
font-weight: bold
display: none
font-size: 16px
color: red
font-size: 25px
img
font-size: 16px
float: right
table
font-size: 16px
display:none
document
html
body
font-size: 16px
font-size: 16px
font-weight: bold
p
div
font-size: 16px
Hello
.Welcome.
img
em
font-size: 16px
font-size: 16px
float: right
outside

节点是否“可见”

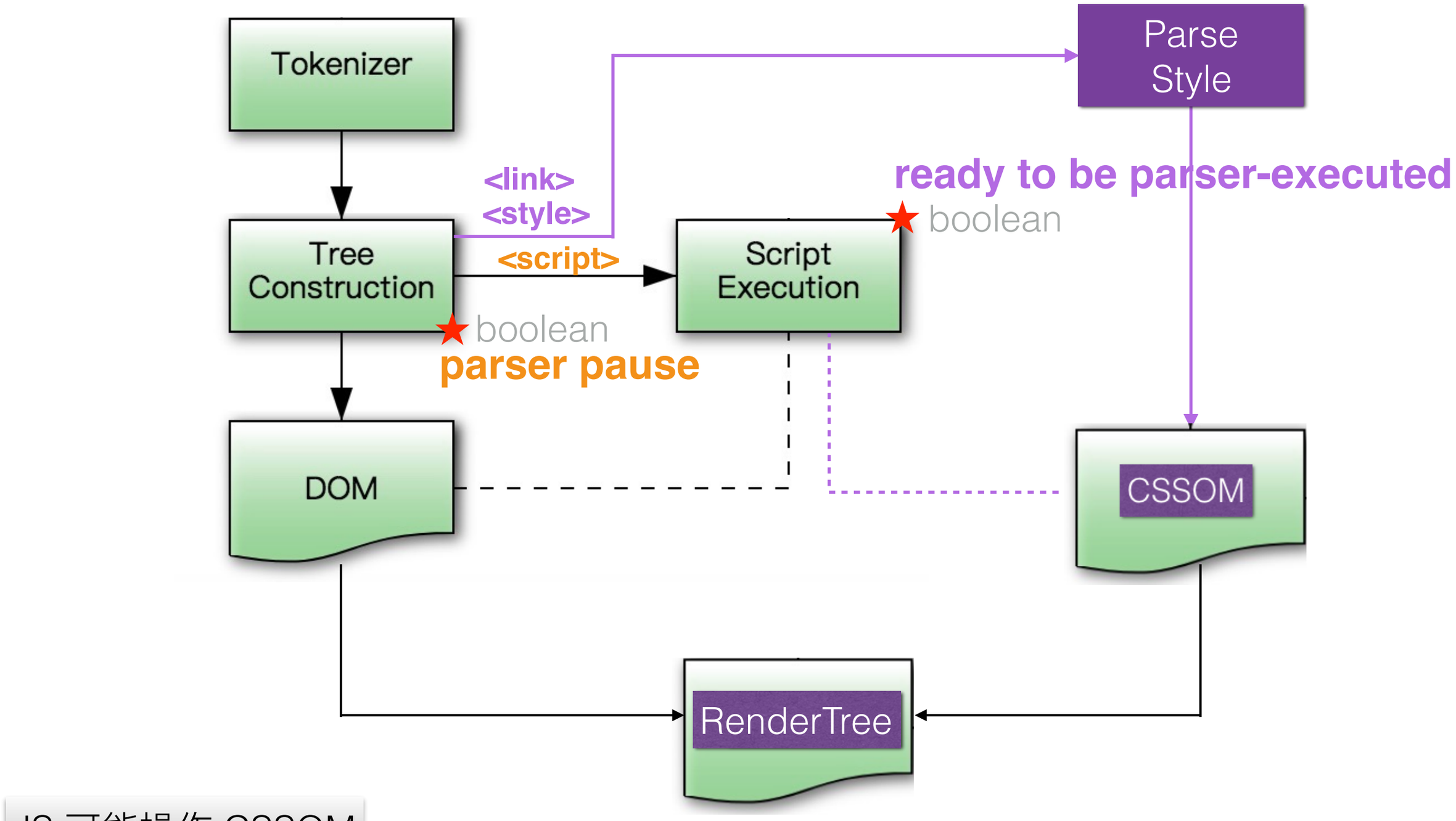
- “不可见”节点：本身不可见＋被CSS隐藏的
  - <head> <meta> <link> <comment>
  - <script> <style>
  - display:none 的元素
- “可见”节点
  - visibility:hidden 的元素

HTML、CSS、JS 之间的关系...


Network
Byte Stream Decoder
<!DOCTYPE html><html lang="en"><head><meta charset="UTF-8">
<title>DevTools</title><style>…</style></head><body> Hello
,FE.Welcome. ...</body></html>
Input Stream Preprocessor
document.write()
Parse HTML
Tokenizer
Tree Construction
Script Execution
html
head
body
…
span
…
DOM



JS 是 Parsing-block 的 ------▶ JS 可能阻塞 DOM 树的构建



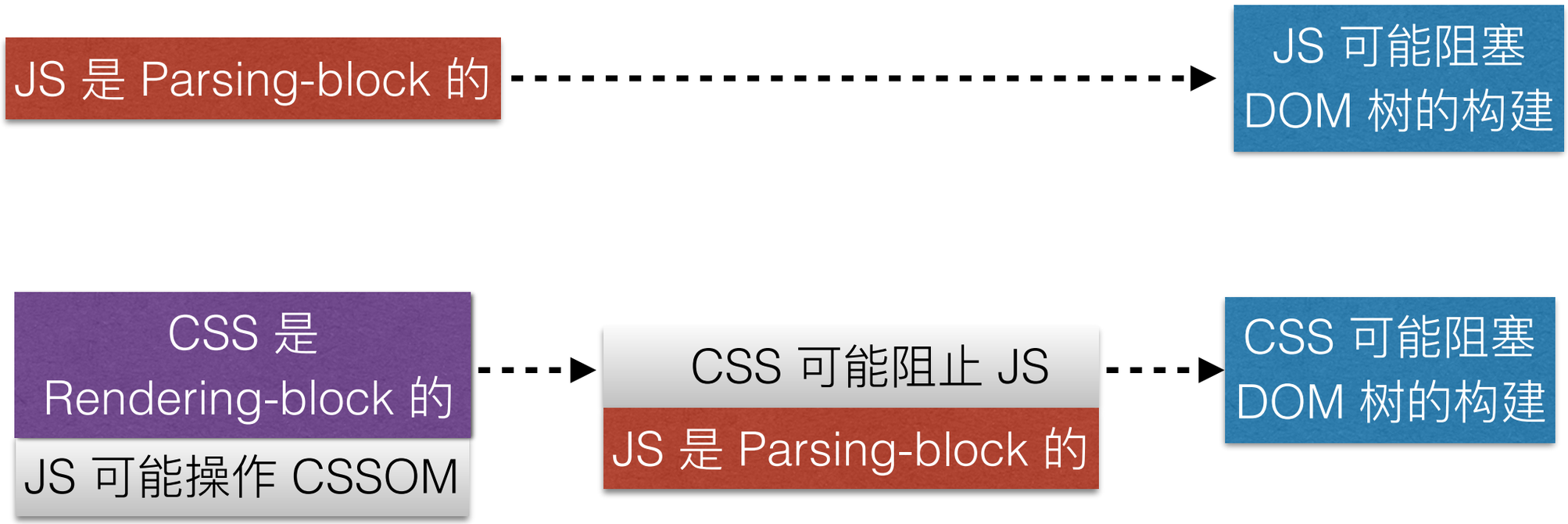
JS 可能操作 CSSOM

CSS 是 Rendering-block 的

CSS 可能阻止 JS

JS 是 Parsing-block 的

CSS 可能阻塞 DOM 树的构建



通过 DevTools / Performance 面板查看
DOM构建活动 和 资源是如何“阻塞”的,
优化＋优化后的数据

演示

例子1：纯文本 Hello World

代码示例

hello-plaintext.html

录制时长的重要性


200 ms
400 ms
600 ms
800 ms
1000 ms
1200 ms
1400 ms
1600 ms
1800 ms
2000 ms
2200 ms
2400 ms
2600 ms
2800 ms
3000 ms
FPS
CPU
NET

50 ms
100 ms
150 ms
200 ms
250 ms
300 ms
350 ms
400 ms
450 ms
500 ms
550 ms
600 ms
650 ms
700 ms
750 ms
800 ms
850 n
FPS
CPU
NET

20 ms
40 ms
60 ms
80 ms
100 ms
120 ms
160 ms
180 ms
200 ms
220 ms
240 ms
260 ms
280 ms
300 ms
320 ms
340 ms
360 ms
380 ms
400
FPS
CPU
NET

事件分组

Filter All Loading Scripting Rendering Painting

Start Time	Self Time	Total Time	Activity
24.4 ms	0 ms	0 ms	Send Request
40.6 ms	0 ms	0 ms	Receive Response
41.4 ms	0 ms	0 ms	Receive Data
51.4 ms	0 ms	0 ms	Finish Loading
52.0 ms	1.5 ms	1.5 ms	Parse HTML

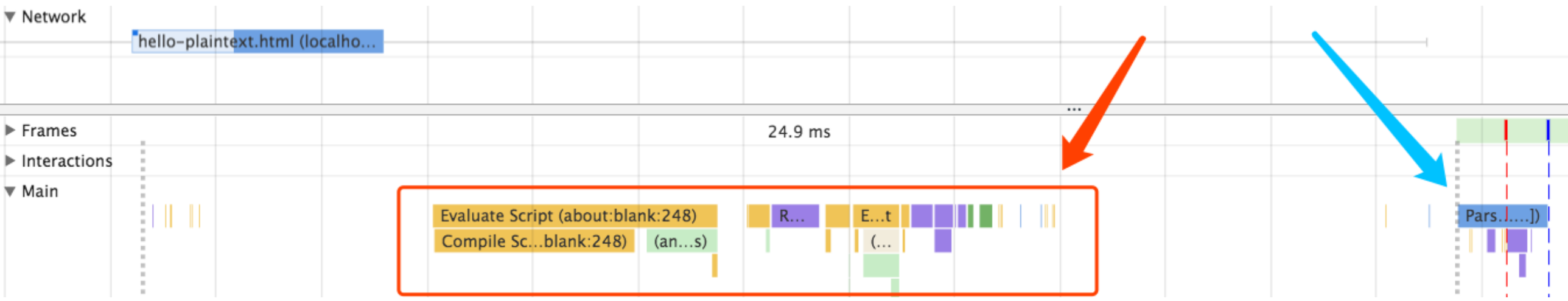
Filter All Loading Scripting Rendering Painting

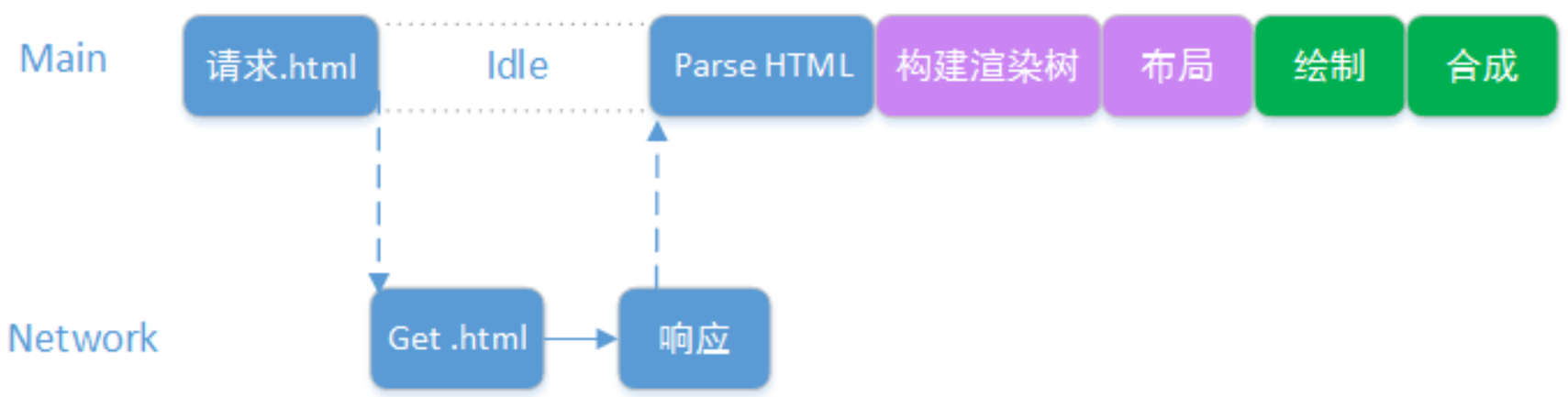
Start Time	Self Time	Total Time	Activity
52.5 ms	0.2 ms	0.2 ms	Recalculate Style
52.9 ms	0.2 ms	0.3 ms	▶ Layout
53.3 ms	0.0 ms	0.0 ms	Recalculate Style
54.0 ms	0.1 ms	0.1 ms	Update Layer Tree

Filter All Loading Scripting Rendering Painting

Start Time	Self Time	Total Time	Activity
62.5 ms	0.1 ms	0.1 ms	Paint
63.1 ms	0.2 ms	0.2 ms	Composite Layers

开始的一段是浏览器的准备
可忽略





演示

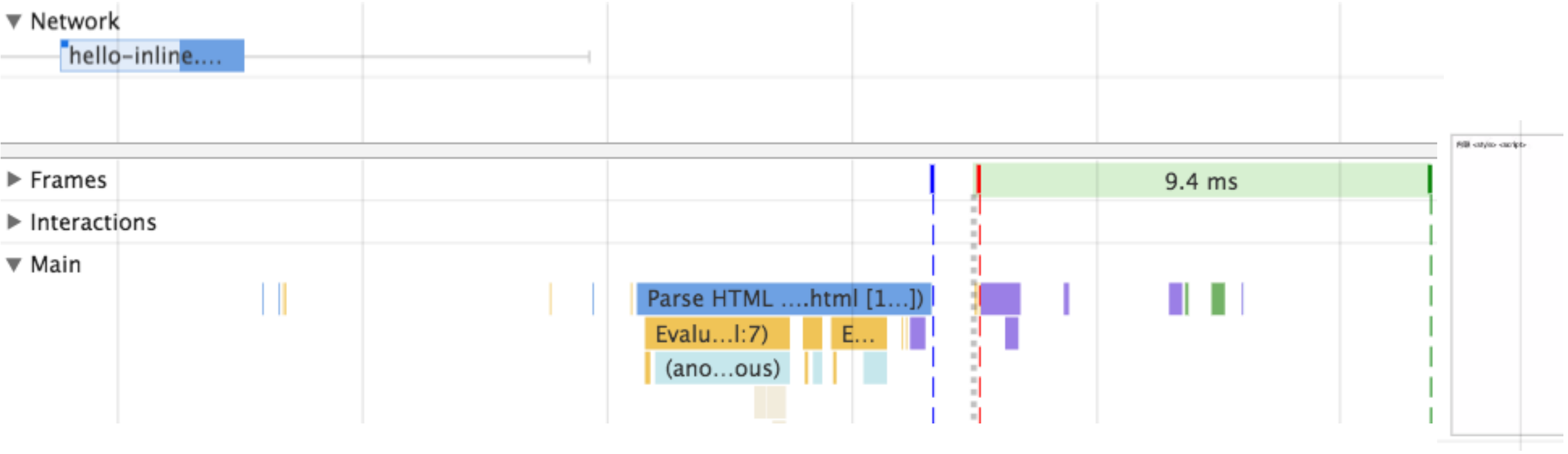
例子2:

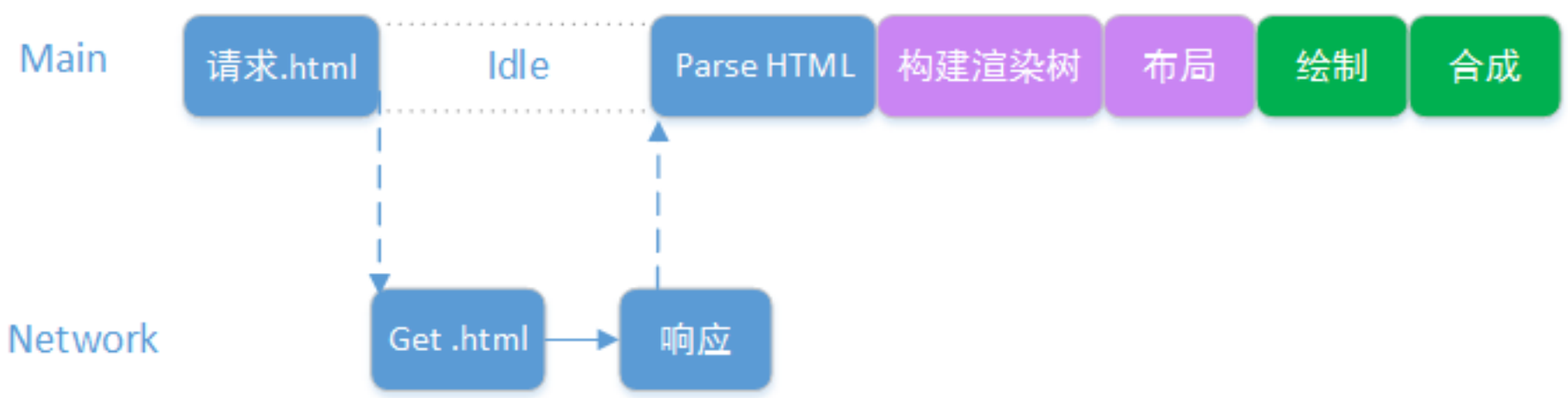
2.1 内联<style> <script>

代码示例

hello-inline.html

1. 一段<script>对应一段 Evaluate Script
2. 阻塞的表现：<script>阻塞ParseHTML
3. 构建CSSOM：内联的<style>不显示


Network
hello-inline....
Frames
9.4 ms
Interactions
Main
Parse HTML ....html [1...])
Evalu...l:7)
E...
(ano...ous)


Main
请求.html
Idle
Parse HTML
构建渲染树
布局
绘制
合成
Network
Get .html
响应

演示

例子2:

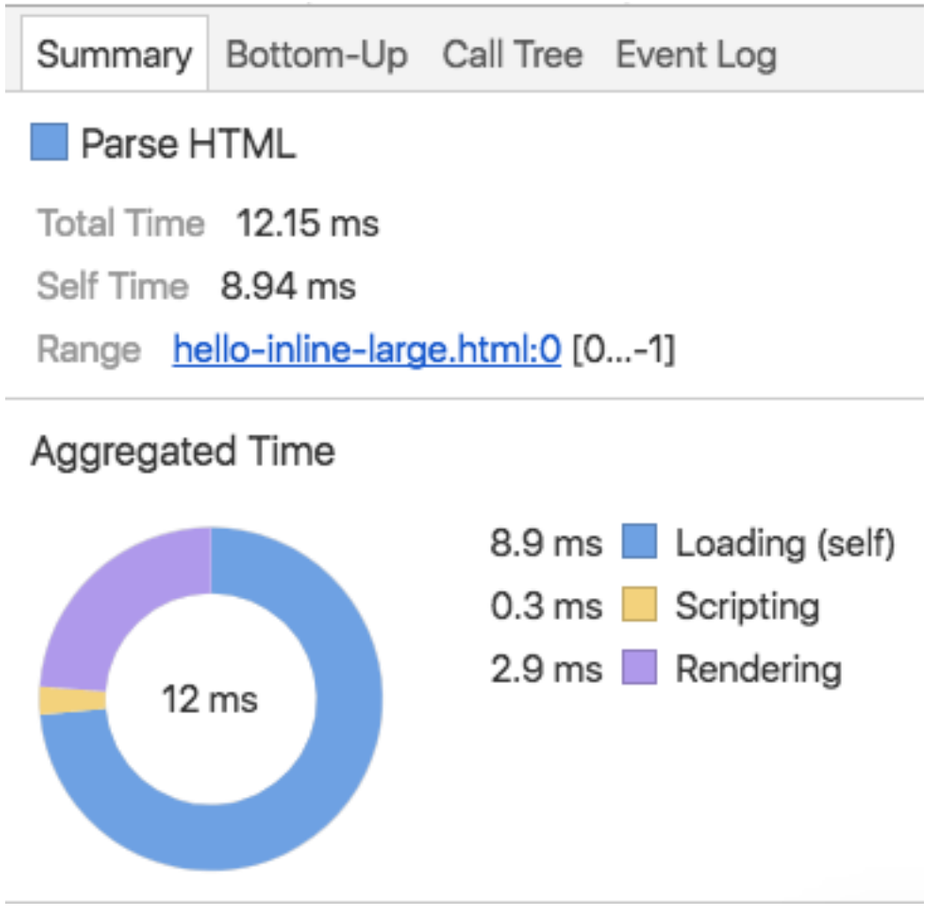
2.2 内联<style> <script>，当文件变大时

代码示例

hello-inline-large.html
hello-inline-large-source.html

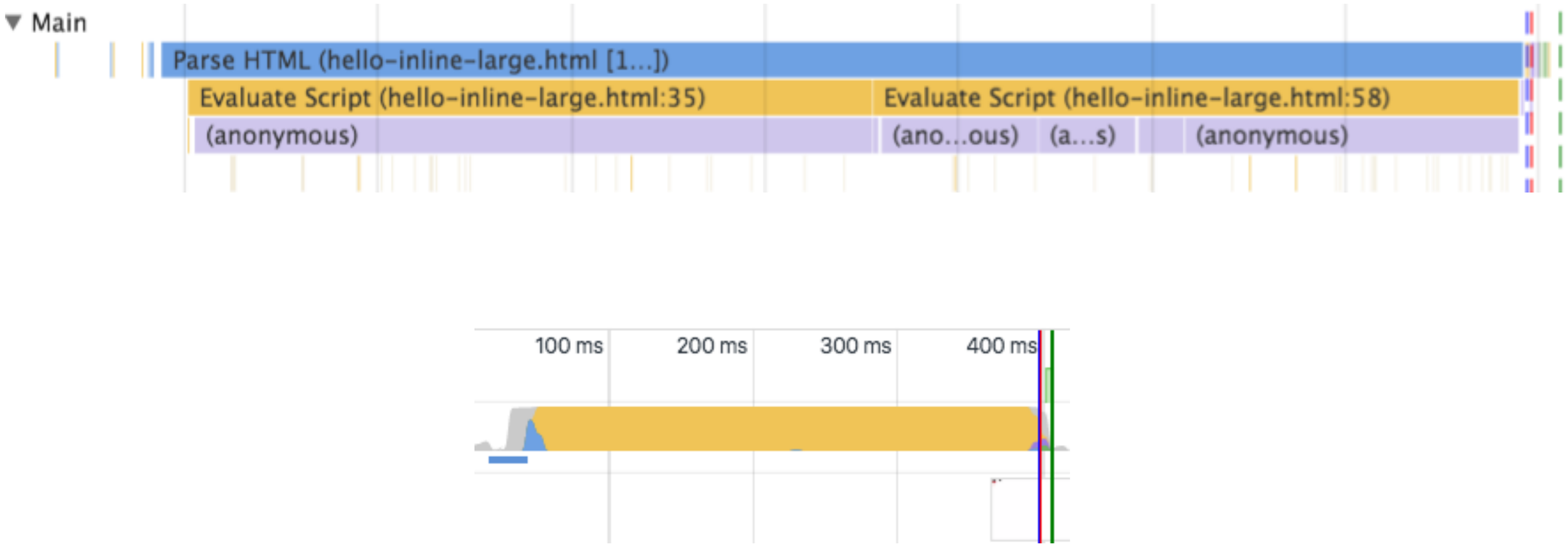
白屏

内联过多的CSS，会导致ParseHTML时间过长
～构建CSSOM～



白屏

内联过多的JS，会导致白屏时间过长
～阻塞ParseHTML～



1. 内联的CSS：删减无用的选择器
2. 内联的JS：删减的脚本量

演示

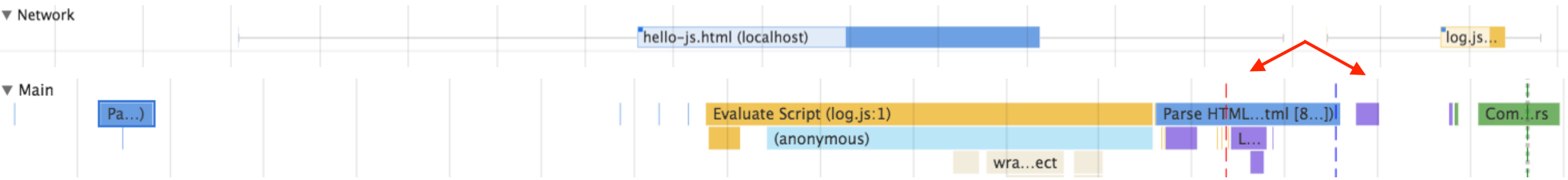
例子3：外链 Script

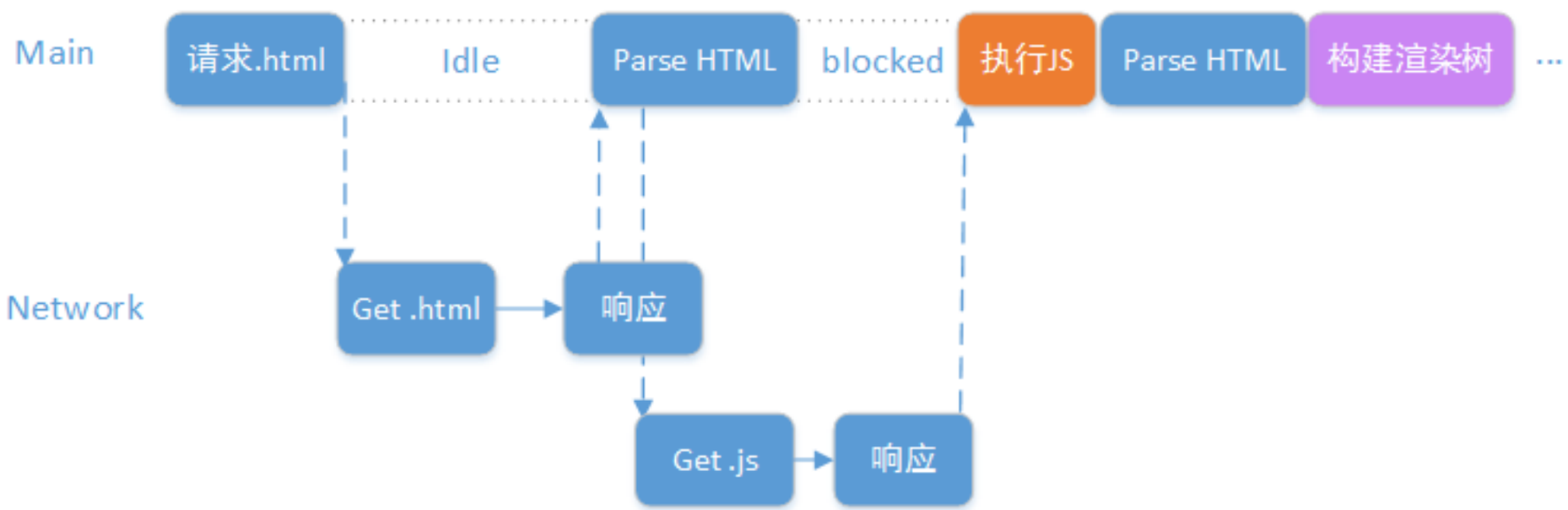
代码示例

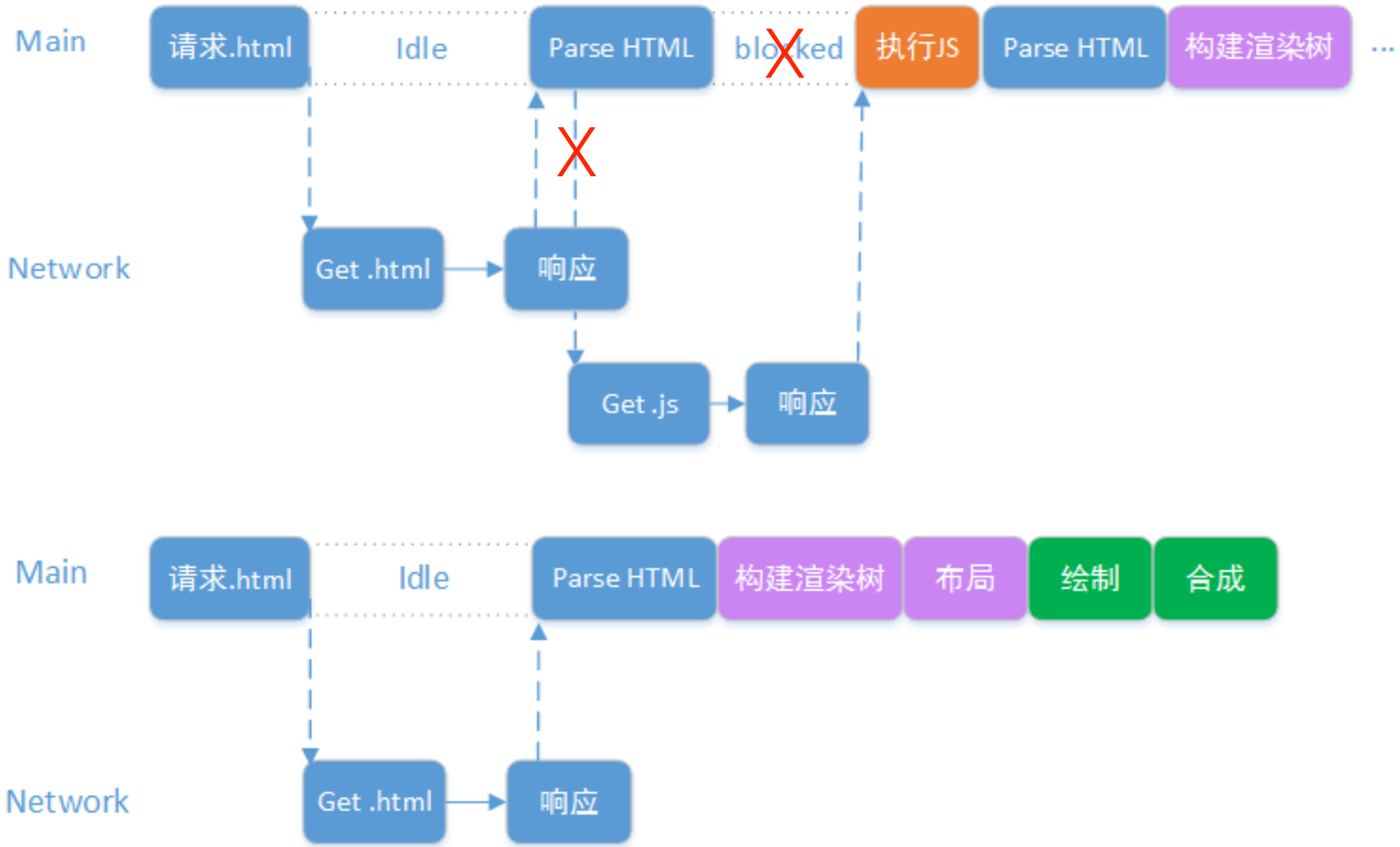
hello-js.html

外链JS：阻塞

```
<head>
    <script src="js/log.js"></script>
</head>
<body>
    Hello<span>,FE</span>.Welcome.
</body>
```







1. 不发请求，用内联JS，放底部
2. 外链JS，放底部
3. 外链JS，加defer/async属性

外链 defer 和 async：消除阻塞


```
<head>
    <script src="js/log.js" async></script>
</head>
<body>
    Hello<span>,FE</span>.Welcome.
</body>
```

```
<body>
    Hello<span>,FE</span>.Welcome.
    <script src="js/log.js" async></script>
</body>
```

defer 和 async

- 相同点
  1. 加载脚本时，不阻塞页面渲染
  2. 对于inline的script无效
  3. 脚本中不能调用 document.write() 方法
- 不同点
  - 1. 执行时机
    - defer：文档解析完毕之后，DOMContentLoaded之前
    - async：下载后立即执行，window.load 之前
  - 2. 彼此间的执行顺序
    - defer：按序，先定义先执行
    - async：无序，先回来先执行

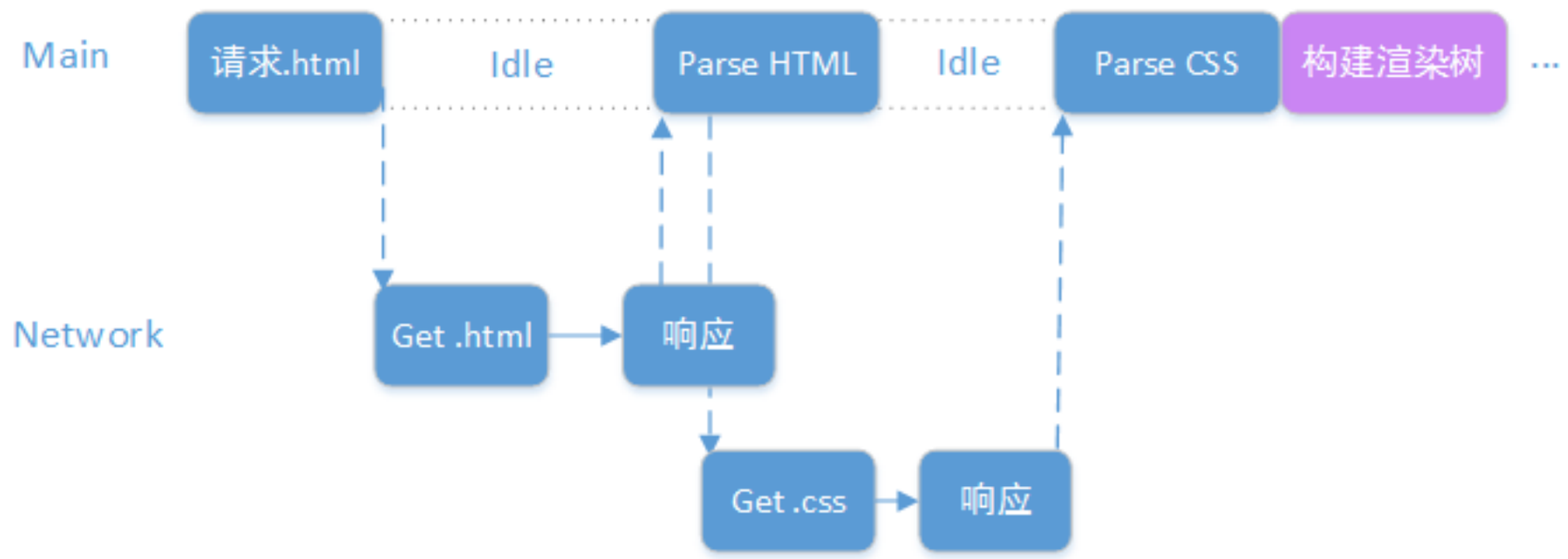
演示

例子4：外链CSS

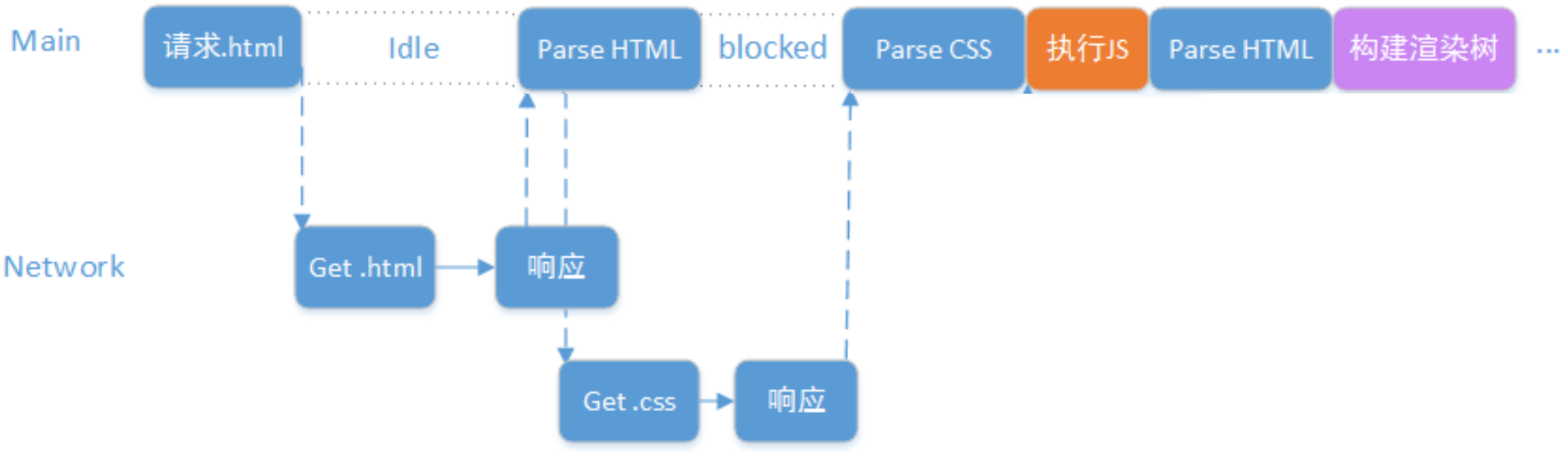
代码示例

hello-css.html

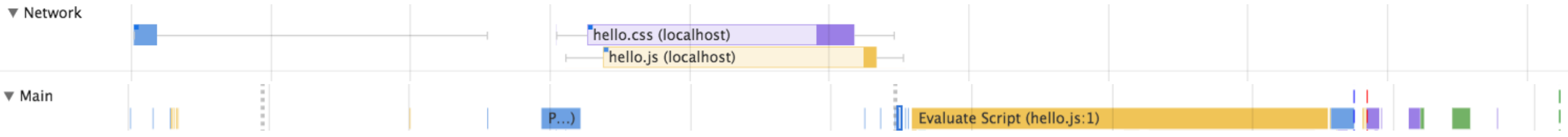
```html
<head>
    <link rel="stylesheet" href="css/hello.css">
</head>
<body>
    Hello, <span>FE.</span>
</body>
```

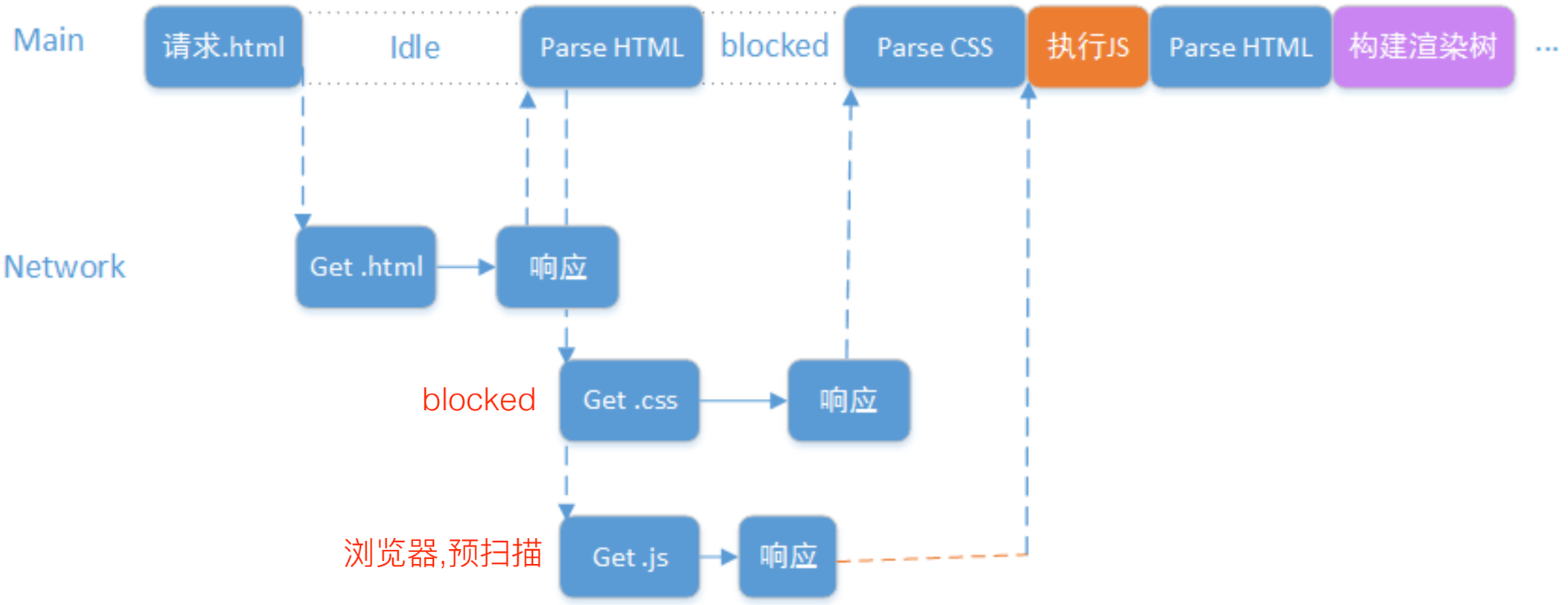


```html
<head>
    <link rel="stylesheet" href="css/hello.css">
</head>
<body>
    Hello, <span>FE.</span>
    <script>
        let span = document.querySelector('span');
        let color = window.getComputedStyle(span).color;
        console.log(color);
    </script>
</body>
```



```html
<head>
    <link rel="stylesheet" href="css/hello.css">
    <script src="js/hello.js"></script>
</head>
<body>
    Hello, <span>FE.</span>
</body>
```





如何消除 link 的 render-blocking?

1. 内联，且尽早执行
2. 利用媒体类型和媒体查询

```
<link href="style.css"    rel="stylesheet">
<link href="style.css"    rel="stylesheet" media="all">
<link href="portrait.css" rel="stylesheet" media="orientation:portrait">
<link href="print.css"    rel="stylesheet" media="print">
```

第3个：动态媒体查询，加载页面时计算

第4个：只在打印网页时应用

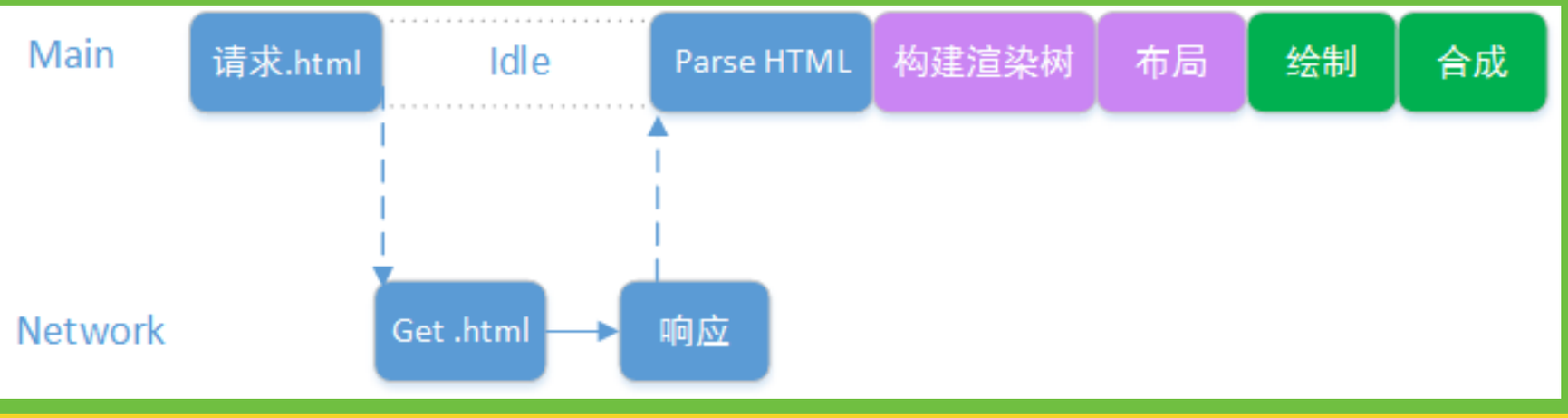
小结

- Parse HTML
  - 构建 DOM、CSSOM、RenderTree
  - 两个阻塞及解决方案
    - Parsing-blocking 的 JS
    - Render-blocking 的 CSS

- DevTools/Performance 的应用
  - Main、Network、Summary及兄弟页签
  - Frame、截图、Overview

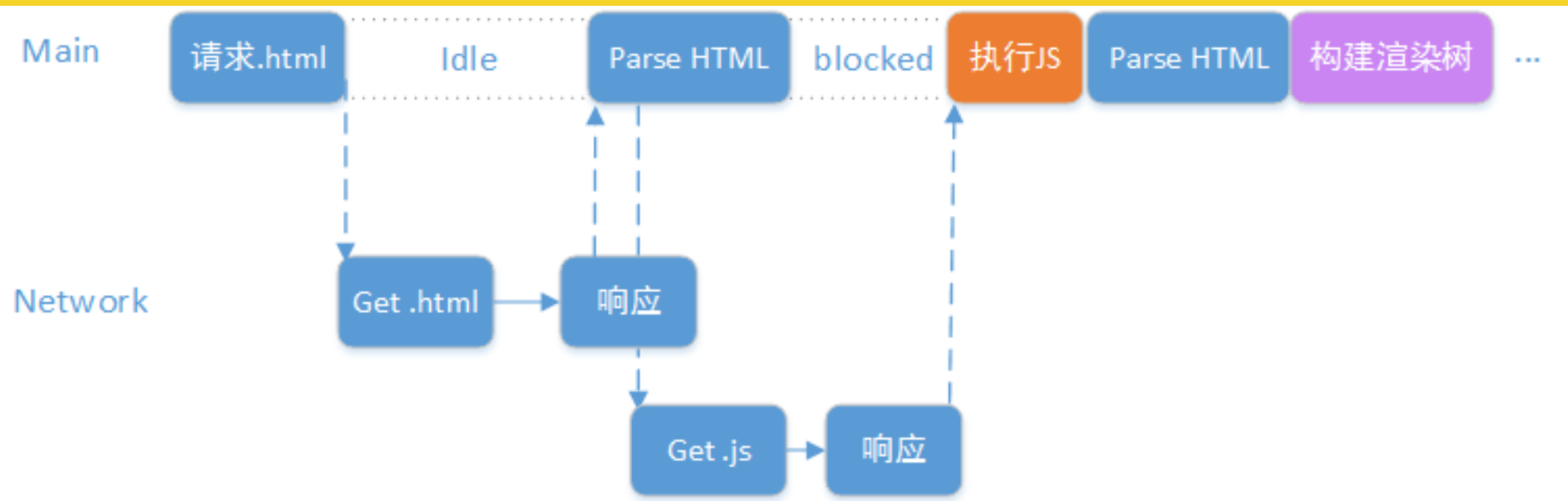
首屏绘制时间

- 关键渲染路径 CRP
- Critical Rendering Path

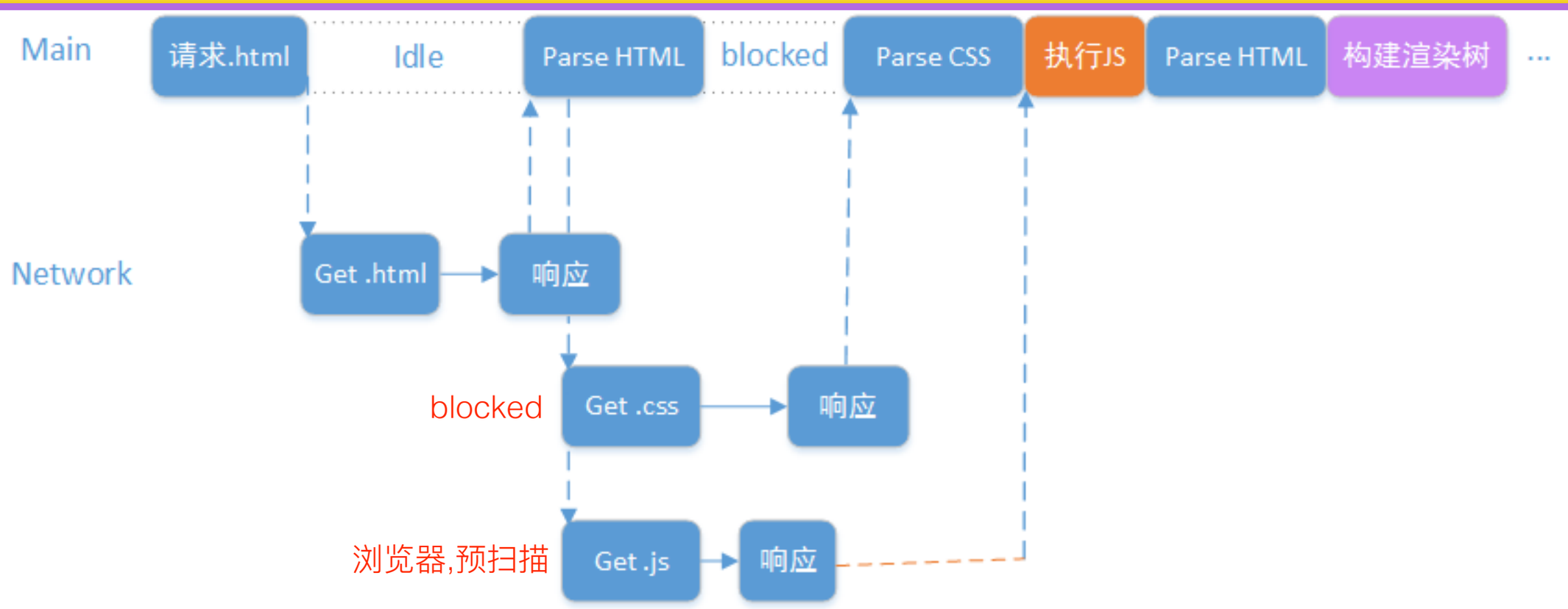


关键资源个数 / 大小 / 网络来回

1 / a / 1



2 / a+b / 2



3 / a+b+c / 2

练习

找一个网站，稍微简单点的

1. 画出关键渲染路径图
2. 算出关键渲染路径的三个指标
3. 是否可优化？优化后的指标

最后…

症状	可导致	思路	解决方法
绿条太晚	首屏白屏	CRP	1.关键资源个数 parse HTML HTTP 2.关键资源大小 HTTP 内容压缩 3.关键资源网络来回 HTTP
FPS太红	不流畅	～看提示～	1.JS的问题 2.Paint的问题
JS问题	～都可能～	看Main	1.若不停导致重绘：则先读后写 2.若要动画，则 requestAnimationFrame 3.若大量纯计算，则 web worker 4.若长时间执行的大任务，则拆成微任务 5.内存问题… JS
Paint问题	不流畅	～工具～	1.使用成本低的CSS: csstriggers.com CSS 2.减少重绘的影响面: More tools > Rendering render 3.提升layer: More tools > Layers render 4. FE：CSS新属性 CSS
资源请求太慢 …	～都可能～	～工具～ 看网络	1.灰线太长（左侧） HTTP • HTTP1.0/1.1，域分片 • HTTP2 2.绿条太长 RD • 网慢：CND / 服务提供商 • 服务器慢：缓存 / 配置 / 算法 3.蓝条长 • 网慢：CND / 服务提供商 • 压缩内容：不同内容不同策略 内容压缩 5.其它 • Main线程的使用情况 • FE：HTML新属性 HTML

FE Browser HTTP

先测量，再优化

- Chrome DevTools
- Performance API
- PageSpeed
- Lighthouse

Q&A

安佳
github.com/anjia
anjia.github.io/

FE@搜索前端

～谢谢～