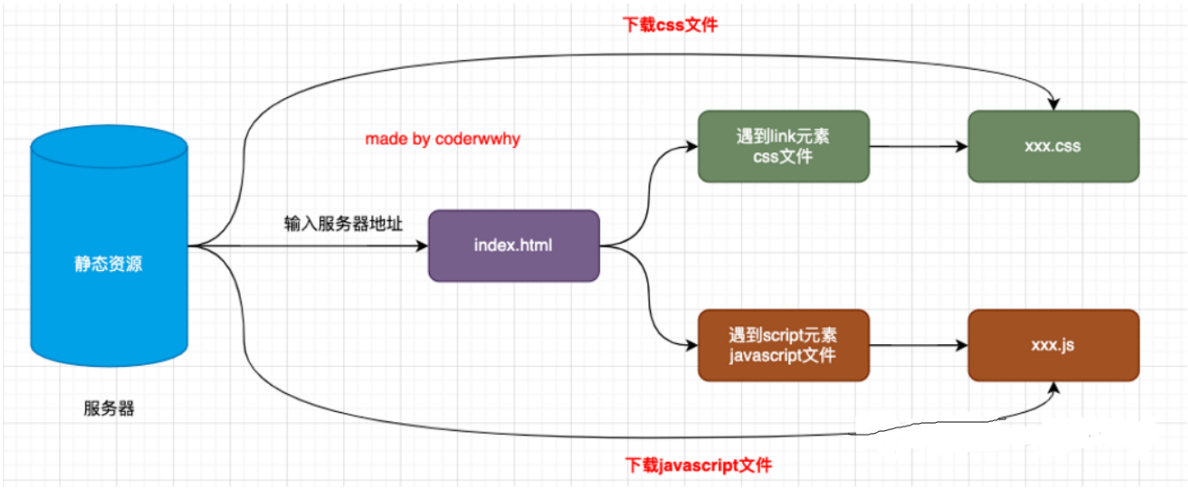
标题1 理论体系背景

一 网页的整体解析流程

1.1 网页被下载的过程



1.2 浏览器内核的理解

浏览器解析网页的过程通常是由浏览器内核完成的

- 浏览器内核是指浏览器中负责解析HTML、CSS JavaScript等文件的核心组成，也被称为渲染引擎

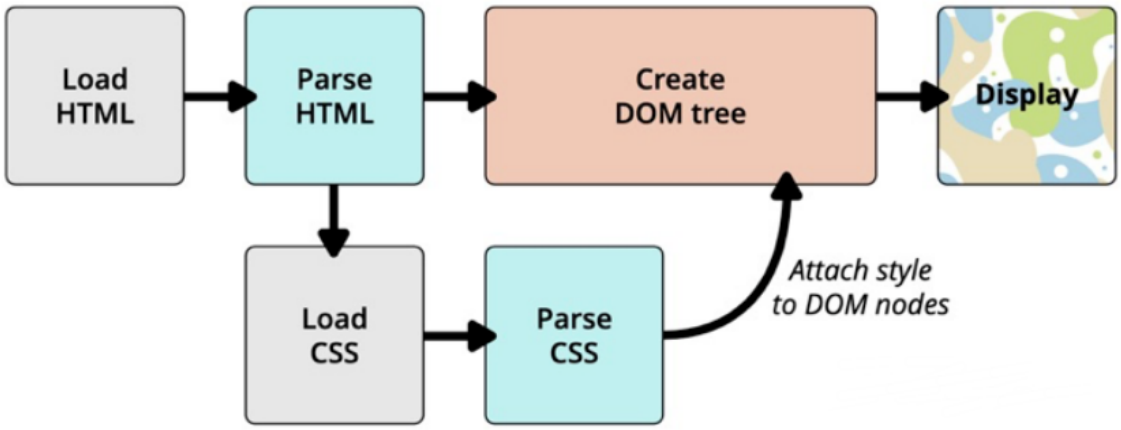
当用户在浏览器中输入网址时，浏览器内核会根据协议类型（如HTTP或HTTPS）向服务器发送请求，下载网站的HTML文件

- 在下载完成之后，浏览器内核会开始解析HTML文件，构建DOM树，根据DOM树和CSS样式表构建渲染树，最终将页面呈现给用户

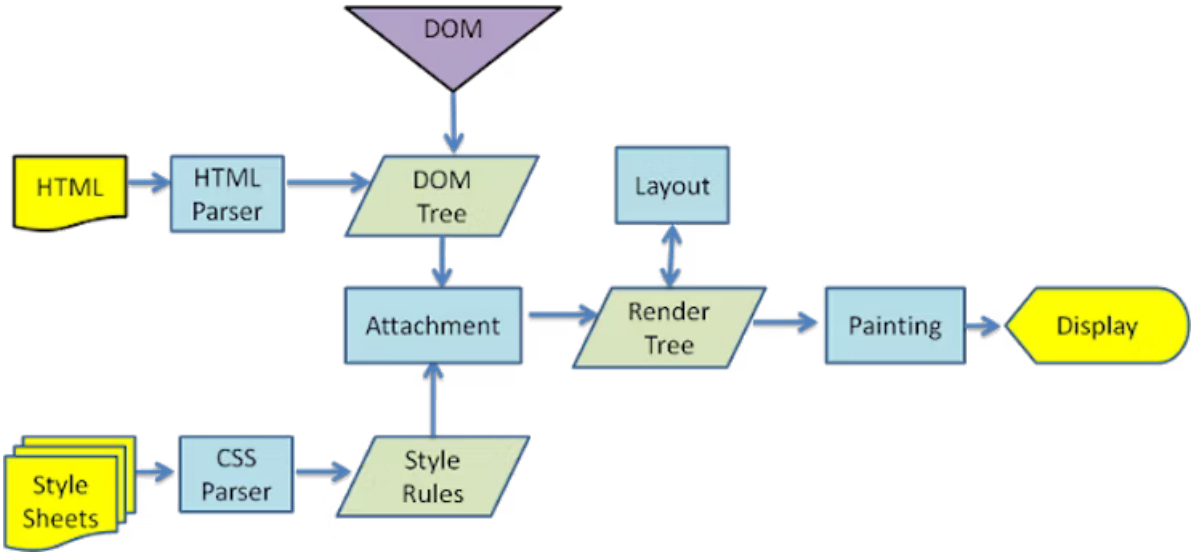
浏览器内核指的是浏览器的排版引擎：排版引擎（layout engine），也称为浏览器引擎（browser engine）、页面渲染引擎（rendering engine）或样版引擎。

1.3 页面渲染整体流程

渲染引擎在拿到一个页面后，如何解析整个页面并且最终呈现出我们的网页呢?



整体流程图



详细流程图

二 页面的详细解析流程

2.1 HTML的解析过程

HTML解析过程，构建DOM树，是浏览器渲染的第一步

- 因为默认情况下服务器会给浏览器返回index.html文件，所以解析HTML是所有步骤的开始：

1. **获取 HTML 文件** 当用户在浏览器中输入网址时，浏览器会向服务器发送请求，请求下载网站的 HTML 文件。
2. **HTML 标记识别** 浏览器会将 HTML 文件解析成一个个标记（tag），如 div、p、img 等等。解析的过程中，浏览器会忽略一些不合法的标记，如没有闭合标签、属性值没有使用引号等等。
3. **DOM 树构建** 浏览器会将解析后的标记转化成一个个 DOM 节点（Node），构建成一棵 DOM 树（Document Object Model）。DOM 树是一个树形结构，根节点是 document，其他节点代表 HTML 文档中的元素、属性、文本等等。

在构建 DOM 树的过程中，浏览器会按照 HTML 文档的层次结构，将文档分成一个个的块（block），如文本块、段落块、表格块等等。每个块都会被转换成一个 DOM 节点，节点之间的关系由 HTML 标记之间的关系来确定。

2.2. 解析生成CSS规则

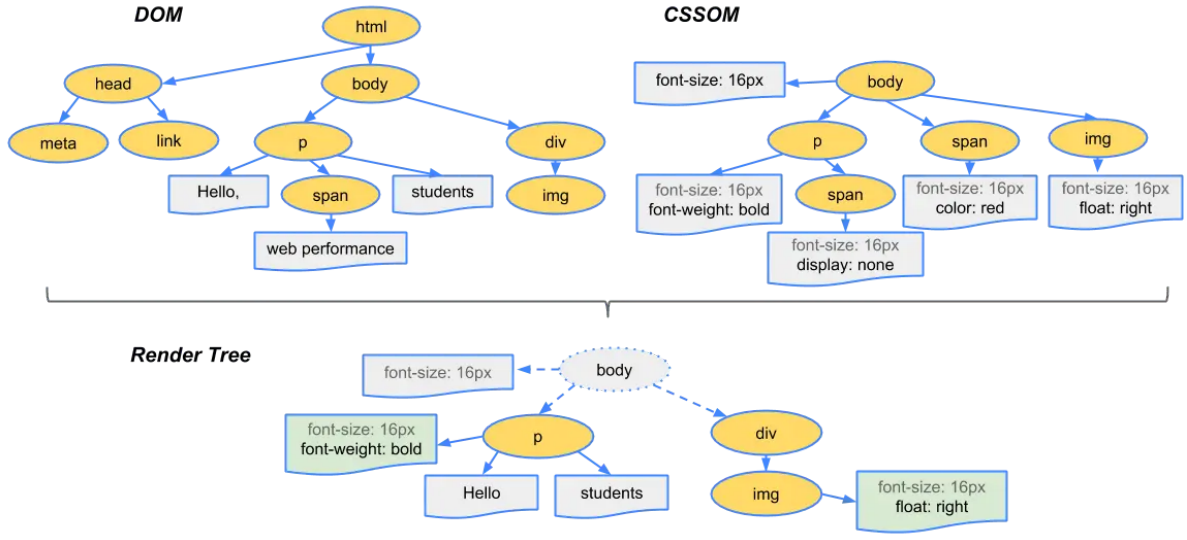
生成 CSS 规则，是浏览器解析 HTML 文件的一部分。

1. 在解析 HTML 文件的过程中，如果遇到 CSS 的 link 元素，浏览器会下载对应的 CSS 文件。
2. 下载完成后，浏览器会对 CSS 文件进行解析，解析出对应的规则树。

- 这个树形结构称为 CSSOM（CSS Object Model，CSS 对象模型），它描述了 HTML 文档中各元素的样式和布局信息。
    - 在CSSOM中，每个节点代表一个CSS规则，包括选择器和声明
    - 选择器指定了那些元素会被应用这个规则，声明则制定了这些元素的样式属性和值
    - CSSOM树的构建过程类似于DOM树的构建过程，也是一个逐步解析的过程

2.3 构建RenderTree

当有了DOM Tree和CSSOM Tree后，就可以两个结合构建RenderTree了



注意一：

- 需要注意的是，link元素不会阻塞DOM树的构建过程，但会阻塞Render Tree的构建过程
- 这是因为Render Tree在构建时需要对应的CSSOM Tree

注意二：

- 同时，需要注意的是Render Tree和DOM Tree并不是一一对应的关系
- 例如，对于display为none的元素，它不会在Render Tree中出现。这是因为该元素被隐藏了，不会影响页面的呈现，因此也不需要在渲染树中进行渲染

2.4. 布局和绘制的过程

布局

- 浏览器会在渲染树（Render Tree）上运行布局（Layout）以计算每个节点的几何体,得到一个
- 渲染树表示显示哪些节点以及其他样式，但不表示每个节点的尺寸、位置等信息。布局时确定呈现树中所有节点的宽度、高度和位置信息

绘制

- 浏览器将每个节点绘制到（Paint）屏幕上
- 在绘制阶段，浏览器将布局阶段计算的每个frame转换为屏幕上的像素点
- 这个部分包括将元素可见的部分进行绘制，例如文本、颜色、边框、阴影、替换元素（例如img）等等

2.5. 回流和重绘的问题

这里还有两个比较重要的概念，也是会对浏览器渲染过程中引发性能问题的两个重要概念：**回流和重绘。**

2.5.1. 回流的解析

理解回流reflow：（也可以称之为重排）

- 第一次确定节点的大小和位置，称之为布局（layout）
- 之后对节点的大小、位置修改重新计算称之为回流。

也就是说回流是指浏览器必须 重新计算渲染树中部分或全部元素的集合信息（位置和大小）

触发回流的情况有很多，常见的包括：

1. DOM结构的变化，比如添加、删除、移动元素等操作
2. 改变元素的布局，比如修改元素的宽高、padding、margin、border、position、display 等属性；
3. 页面尺寸的变化，比如浏览器窗口大小的变化，或者文档视口的变化
4. 获取元素的集合属性比如调用 getComputedStyle() 方法获取元素的尺寸、位置等信息

回流的代价比较高，因为它会涉及到大量的计算和页面重排，这会导致页面的性能和响应速度下降。

2.5.2. 重绘的解析

理解重绘repaint：

- 第一次渲染内容称为绘制（paint）
- 之后重新渲染称之为重绘

重绘是指浏览器不需要重新计算元素的几何信息，而只需要重新绘制元素的内容的过程。

触发重绘的情况有很多，常见的包括：

1. 修改元素的颜色、背景色、边框颜色、文本样式等属性；
2. 修改元素的 box-shadow、text-shadow、outline 等属性；
3. 使用 CSS3 transform 和 opacity 等属性；
4. 添加、移除、修改元素的 class；
5. 使用 JavaScript 直接修改样式。

重绘的代价比较小，因为它不涉及到元素的位置和大小等计算，只需要重新绘制元素的内容即可。

回流一定会引起重绘，所以回流是一件很消耗性能的事情。

2.5.3. 页面性能优化

避免回流是提高页面性能的重要手段之一，以下是一些常用的优化方法：

1. 尽量一次性修改样式 可以使用 cssText 属性、添加 class 等方式，一次性修改元素的样式，避免多次修改引起页面的回流。比如，可以将需要修改的样式保存在一个对象中，然后一次性设置给元素，避免多次触发回流。
2. 避免频繁的操作 DOM 可以使用 DocumentFragment 或者父元素来操作 DOM，避免频繁地操作 DOM 元素，从而减少页面的回流次数。
3. 避免使用 getComputedStyle() 获取元素信息 因为 getComputedStyle() 方法会强制浏览器进行回流操作，从而影响页面的性能。如果需要获取元素的信息，可以在修改样式之前先保存到变量中，避免多次触发回流。
4. 使用 position:absolute 或者 position:fixed 属性 使用 position:absolute 或者 position:fixed 属性可以将元素从文档流中脱离出来，从而避免影响其他元素的位置和大小。虽然这也会引起回流，但是相对于修改文档流中的元素来说，开销较小，对页面的性能影响较小。

总之，避免回流是一项重要的优化技巧，可以帮助我们提高页面的性能和响应速度。可以采用一些常用的优化方法，减少回流的次数，从而提高页面的渲染速度和性能。

2.6. composite合成

绘制的过程，可以将布局后的元素绘制到多个合成图层中。

- 这是浏览器的一种优化手段；

默认情况下，标准流中的内容都是被绘制在同一个图层（Layer）中的；

而一些特殊的属性，会创建一个新的合成层（ CompositingLayer ），并且新的图层可以利用GPU来加速绘制；

- 因为每个合成层都是单独渲染的；

有些属性可以触发合成层的创建，包括：

1. 3D 变换（3D Transforms）：如 rotateX、rotateY、translateZ 等属性，可以创建一个新的合成层。
2. video、canvas、iframe 等标签：这些标签会创建一个新的合成层。
3. opacity 动画转换时：当元素透明度发生变化时，会创建一个新的合成层。
4. position: fixed：将元素定位为固定位置时，也会创建一个新的合成层。
5. will-change 属性：可以通过这个实验性的属性，告诉浏览器元素可能会发生哪些变化，从而预先创建合成层。
6. 动画（Animation）或过渡（Transition）设置了 opacity、transform 属性时，也会创建一个新的合成层。

需要注意的是，过度使用合成层也会带来一些问题，如占用更多的内存、增加页面的复杂度等。

- 因此，在使用合成层时需要谨慎，避免滥用。

2.7 script元素

- 事实上，浏览器在解析HTML的过程中，遇到了script元素是不能继续构建DOM树的
- 他会停止继续构建，首先下载JavaScript代码，并且执行JavaScript的脚步
- 只有等到JavaScript脚本执行结束后，才会继续解析HTML，构建DOM树
    - 只要遇到js文件就会停止解析，会执行完js之后再进行继续解析

原因

- 这是因为JavaScript的作用之一就是操作DOM，并且可以修改DOM
- 如果等到DOM树构建完成并且渲染再执行JavaScript，会造成严重的回流和重绘，影响页面的性能
- 遇到script元素时，优先下载和执行JavaScript代码，再继续构建DOM树

标题2 应用场景

2.1 减少回流和重绘

1. 例如，多次修改一个把元素布局的时候，我们很可能会如下操作

```
const el = document.getElementById('el')
// 这里循环判定比较简单，实际中或许会拓展出比较复杂的判定需求
for(let i=0;i<10;i++) {
    el.style.top  = el.offsetTop  + 10 + "px";
    el.style.left = el.offsetLeft + 10 + "px";
}
```

每次循环都需要获取多次`offset`属性，比较糟糕，可以使用变量的形式缓存起来，待计算完毕再提交给浏览器发出重计算请求

放大后会有很多重复的recaculate style（重新计算样式）和layout（布局）

2. 避免逐条改变样式，使用类名去合并样式

比如我们可以把这段单纯的代码：

```
const container = document.getElementById('container')
container.style.width = '100px'
container.style.height = '200px'
container.style.border = '10px solid red'
container.style.color = 'red'
```

优化成一个有 class 加持的样子：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
  <style>
    .basic_style {
      width: 100px;
      height: 200px;
      border: 10px solid red;
      color: red;
    }
  </style>
</head>
<body>
  <div id="container"></div>
  <script>
  const container = document.getElementById('container')
  container.classList.add('basic_style')
  </script>
</body>
</html>
```

前者每次单独操作，都去触发一次渲染树更改（新浏览器不会），

都去触发一次渲染树更改，从而导致相应的回流与重绘过程

避免逐条改变样式，使用类名去合并样式

合并之后，等于我们将所有的更改一次性发出

2.2 浏览器优化机制

- “width、height、border是几何属性，各触发一次回流；color只造成外观的变化，会触发一次重绘。”

```
let container = document.getElementById('container')
container.style.width = '100px'
container.style.height = '200px'
container.style.border = '10px solid red'
container.style.color = 'red'
```

每次 DOM 操作都即时地反馈一次回流或重绘，那么性能上来说是扛不住的。于是它自己缓存了一个 flush 队列，把我们触发的回流与重绘任务都塞进去，待到队列里的任务多起来、或者达到了一定的时间间隔，或者“不得已”的时候，再将这些任务一口气出队。因此我们看到，上面就算我们进行了 4 次 DOM 更改，也只触发了一次 Layout 和一次 Paint。

- 由于每次重排都会造成额外的计算消耗，因此大多数浏览器都会通过队列化修改并批量执行来优化重排过程。
- 浏览器会将修改操作放入到队列里，直到过了一段时间或者操作达到了一个阈值，才清空队

- 当你获取布局信息的操作的时候，会强制队列刷新，包括前面讲到的`offsetTop`等方法都会返回最新的数据
- 因此浏览器不得不清空队列，触发回流重绘来返回正确的值

标题3 其他相关技术体系

常见的浏览器内核

1. Trident （三叉戟） Trident 内核最初是由 Microsoft 开发的，用于 Internet Explorer 浏览器。
2. - 后来，一些国内的浏览器厂商（如 360安全浏览器、搜狗高速浏览器、百度浏览器、UC浏览器）也采用了 Trident 内核。
3. Gecko （壁虎） Gecko 内核最初是由 Mozilla 开发的，用于 Firefox 浏览器。
4. - Gecko 内核的优势在于支持 HTML5、CSS3 等最新的 Web 标准，并且具有较高的性能和稳定性。
5. Presto（急板乐曲）-> Blink （眨眼） Presto 内核最初是由 Opera 开发的，用于 Opera 浏览器。
6. - 后来，Opera 采用了 Blink 内核，Blink 内核基于 WebKit 内核进行了改进和优化，能够更快地渲染页面，并且支持更多的 HTML5、CSS3 特性。
7. Webkit WebKit 内核最初是由 Apple 开发的，用于 Safari 浏览器。
8. - 现在，很多国内的浏览器厂商（如 360极速浏览器、搜狗高速浏览器）也采用了 WebKit 内核。
   - 除了桌面浏览器，WebKit 内核在移动设备上也得到了广泛的应用，如 iOS 和 Android 系统的浏览器。
9. Webkit -> Blink Blink 内核最初也是由 Google 开发的，用于 Chrome 浏览器。
10. - Blink 内核基于 WebKit 内核进行了改进和优化，并且具有更高的性能和更好的兼容性。
    - 现在，Microsoft Edge 也采用了 Blink 内核。

标题4 在EC项目/Jetlinks上的应用

jetlinks的网页
参考网站：http://doc.jetlinks.cn/

标题5 总结与展望

此次分享主要是关于浏览器页面的解析流程，对于JavaScript的详细运行过程可以作为后续的了解继续学习。

标题6 参考文献

（注明是内部文献还是外部文献，内部文献是以往学长学姐或同学们已经讲过的内容，外部文献是上网搜索的文献，包括参考的视频、博客等来源。）

https://juejin.cn/post/6982531924023754783

https://developer.aliyun.com/article/1284630

https://juejin.cn/post/6850418121548365831