一、浏览器安全

1. 什么是XSS攻击?

前端安全系列（一）：如何防止XSS攻击?

（1）概念

XSS攻击指的是跨站脚本攻击，是一种代码注入攻击。攻击者通过在网站注入恶意脚本，使之在用户的浏览器上运行，从而盗取用户的信息如cookie等。

XSS的本质是因为网站没有对恶意代码进行过滤，与正常的代码混合在一起了，浏览器没有办法分辨哪些脚本是可信的，从而导致了恶意代码的执行。

攻击者可以通过这种攻击方式进行以下操作：

- 获取页面的数据，如DOM、cookie、localStorage
- DOS攻击，发送合理请求，占用服务器资源，从而使用户无法访问服务器
- 破坏页面结构
- 流量劫持（将链接指向某网站）

（2）攻击类型

XSS可以分为存储型、反射型和DOM型：

- 存储型指的是恶意脚本会存储在目标服务器上，当浏览器请求数据时，脚本从服务器传回执行。
- 反射型指的是攻击者诱导用户访问一个带有恶意代码的URL后，服务器端接收数据后处理，然后把带有恶意的代码的数据发送到浏览器端，浏览器端解析这段带有XSS代码的数据后当做脚本执行，

最终完成XSS攻击
- DOM型指的通过修改页面的DOM节点形成XSS。

（3）如何防御XSS攻击?

- 可以从浏览器的执行来进行预防，一种是使用纯前端的方式，不用服务器端拼接后返回（不使用服务端渲染）。另一种是对需要插入到HTML中的代码做好充分的转义。对于DOM型的攻击，主要是前端脚本的不可靠而造成的，对于数据获取渲染和字符串拼接的时候应该对可能出现的恶意代码情况进行判断。
- 使用CSP（内容安全策略），CSP的本质是建立一个白名单，告诉浏览器哪些外部资源可以加载和执行，从而防止恶意代码的注入攻击。
- 对一些敏感信息进行保护，比如cookie使用 http-only ，使得脚本无法获取。也可以使用验证码，避免脚本伪装成用户执行一些操作。

cookie和localstorage怎样去防范一个黑客向页面注入恶意代码?

- 对于cookie来说，在http头部上配置，set-cookie，有两个属性可以防止XSS攻击：
  - **http-only**：这个属性可以禁止JavaScript访问cookie，故可以保护cookie不被嵌入的恶意代码所获取。在脚本中，`document.cookie`无法获取该cookie值，对XSS攻击有防御作用，但对网络拦截还是会泄露。
  - **secure**：这个属性告诉客户端浏览器仅在当前https请求时发送cookie
  - 也可以在cookie中添加校验信息，这个校验信息和当前用户外置环境有些关系，比如 ip、user agent等有关，这样当cookie被人劫持冒用时，在服务器端校验时，发现校验值发生了变化，因此会要求用户重新登录，可以规避cookie劫持。

2. 什么是CSRF攻击?

（1）概念

CSRF攻击指的是 **跨站请求伪造攻击**，攻击者诱导用户进入一个第三方网站，然后该网站向被攻击网站发送跨站请求。如果用户在被攻击网站中保存了登录状态，那么攻击者可以利用这个登录状态，绕过后台的用户验证，冒充用户向服务器执行一些操作。

CSRF攻击的 **本质是利用Cookie会在同源请求中携带发送给服务器的特点，以此来冒充用户。**

（2）攻击类型

- GET类型的CSRF攻击，比如在网站中的一个img标签里构建一个请求，当用户打开这个网站时就会自动发起提交
- POST类型的CSRF攻击，比如构建一个表单，然后隐藏它，当用户进入页面时，自动提交这个表单
- 链接类型的CSRF攻击，比如在a标签的href属性里构建一个请求，然后诱导用户区点击。

（3）如何防御CSRF攻击

- 验证码：对敏感操作加入验证码，强制用户与网站进行交互。
- **进行同源检测。**服务器根据http请求头中origin或者Referer 信息来判断请求是否为允许访问的站点，从而对请求进行过滤。
- 使用CSRF Token 进行验证，在请求头中加入token验证字段，浏览器并不会自动携带token去请求，且token可以携带一段加密的jwt用作身份认证，这样进行CSRF的时候仅传递了cookie，并不能表明用户身份，网站即拒绝攻击请求
- 对cookie进行双重验证
- 在设置cookie属性的时候设置 Samesite，该属性表示cookie不随着跨域请求发送，限制cookie不能作为被第三方使用。
-

3. 什么是中间人攻击？如何防范中间人攻击?

中间人 (Man-in-the-middle attack, MITM) 是指攻击者与通讯的两端分别创建独立的联系, 并交换其所收到的数据, 使通讯的两端认为他们正在通过一个私密的连接与对方直接对话, 但事实上整个会话都被攻击者完全控制。在中间人攻击中，攻击者可以拦截通讯双方的通话并插入新的内容

二、进程与线程

1. 进程与线程的概念

从本质上说，进程和线程都是CPU工作时间片的一个描述：

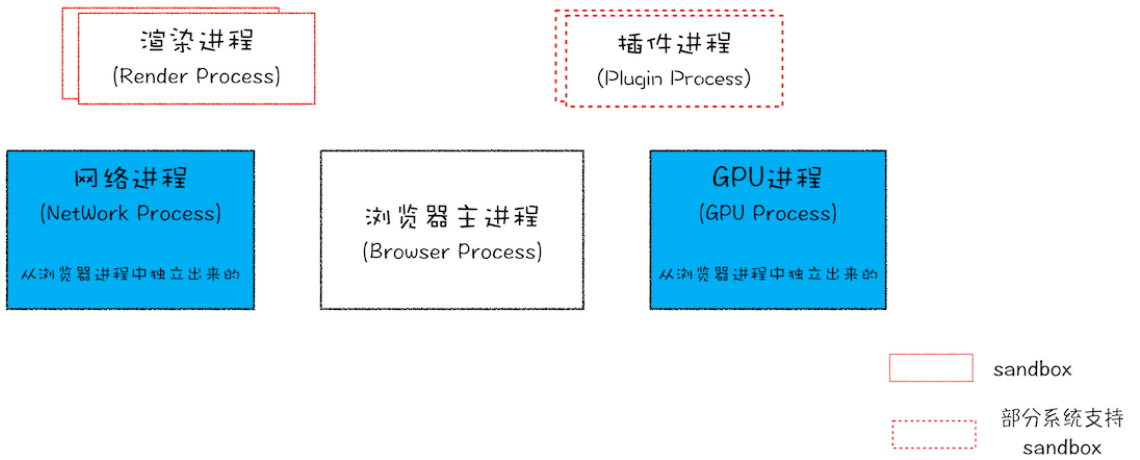
- 进程描述了CPU在运行指令及加载和保存上下文所需的时间，放在应用上来说就代表了一个程序。
- 线程是进程中的更小单位，描述了执行一段指令所需的时间。

进程是资源分配的最小单位，线程是CPU调度的最小单位。

进程和线程之间的关系有以下四个特点：

1. 进程中的任意一线程执行出错，都会导致整个进程崩溃
2. 线程之间共享进程中的数据
3. 当一个进程关闭后，操作系统会回收进程所占用的内存。
4. 进程之间的内容相互隔离

Chrome浏览器的架构图



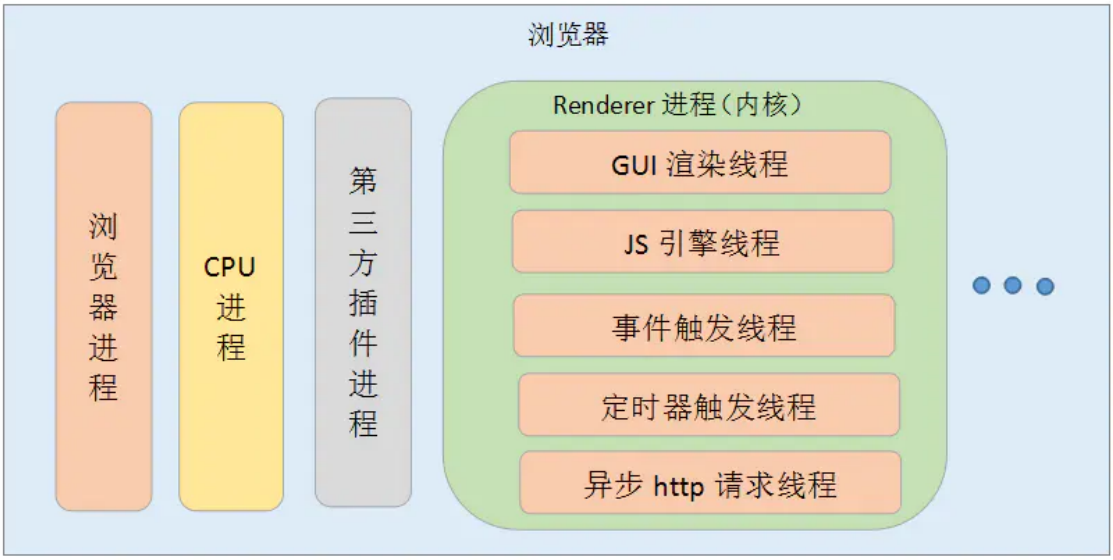
打开一个网页，至少需要四个进程：1个网络进程、1个浏览器进程、1个GPU进程和1个渲染进行。

2. 进程和线程的区别

- 进程可以看做独立应用，线程不能
- 资源：进程是CPU资源分配的最小单位（是能拥有资源和独立运行的最小单位）；线程是CPU调度的最小单位（线程是建立在进程的基础上的一次程序运行单位，一个进程中可以有多个线程）
- 通信方面：线程间可以通过直接共享同一进程中的资源，而进程通信需要借助 进程间通信
- 调度：进程切换比线程切换的开销要大。线程切换不会引起进程切换，但某个进程中的线程切换到另一个进程中的线程时，会引起进程切换。
- 系统开销：进程的创建或撤销要大于线程。

3. 浏览器渲染进程的线程有哪些

浏览器的渲染进程的线程共有5种：



（1）GUI渲染进程

负责渲染浏览器页面，解析HTML、css、构建DOM树、构建CSSOM树，构建渲染树和绘制页面；当界面需要**重绘**或由于某种操作引发**回流**时，该线程执行。

> 注意：GUI渲染引擎线程和JS引擎线程是互斥的，当JS引擎执行时GUI线程会被挂起，GUI更新会保存在一个队列中等到JS引擎空闲时立即被执行。

（2）JS引擎线程

JS引擎线程也称为JS内核，负责处理JavaScript脚本程序，解析JavaScript脚本，运行代码；JS引擎线程一直等待着任务队列中任务的到来，如果然后加一处理，一个Tab页中无论什么时候都只有一个Js引擎线程在运行Js程序。

> 注意：GUI渲染线程与JS线程是互斥的，如果JS 执行时间过长，会造成页面的渲染不连贯，导致页面渲染加载阻塞。

（3）事件触发线程

事件触发线程 属于浏览器而不是JS引擎，用来控制事件循环；当Js引擎执行代码块如setTimeOut时（也可是来自浏览器内核的其他线程，如鼠标点击，ajax异步请求等），会将对应任务添加到事件触发线程中；当对应的事件符合条件被触发时，该线程会把事件添加到待处理队列的队尾，等待Js引擎的处理。

> 注意：由于JS的单线程关系，所以这些待处理队列中的事件都得排队等待JS引擎处理（Js引擎空闲时才会去执行）

（4）定时器触发线程

定时器触发线程即setInterval 与 setTimeOut 所在线程；浏览器定时器计数器并不是由JS引擎计数的，因为JS引擎是单线程的，如果处于阻塞线程状态就会影响计时的准确性；因此使用单独线程来计时并触发定时器，计时完毕后，添加到事件队列中，等待JS引擎空闲后执行，所以定时器中的任务在设定的时间点不一定能够准时执行，定时器只是在指定时间点将任务添加到事件队列中。

> 注意：W3C在HTML标准中规定，定时器的定时时间不能小于4ms，如果是小于4ms，则默认为4ms。

（5）异步http请求线程

- XMLHTTPRequest连接后通过浏览器新开一个线程请求
- 检测到状态变更时，如果设置有回调函数，异步线程就产生状态变更事件，将回调函数放入事件队列中，等待JS引擎空闲后执行。

4. 死锁问题

什么是死锁

死锁就是指多个进程在运行过程中因争夺资源而造成的一种僵局，当进程处于这种僵持状态时，若无外力作用，他们都将无法再向前推进。

产生死锁的原因

（1）竞争资源

- **竞争不可剥夺资源**（例：系统中只有一台打印机，可供进程P1使用，假定P1已占用了打印机，若P2继续要求打印机打印将阻塞）
- **竞争临时资源**（临时资源包括硬件中断、信号、消息、缓冲区内的消息等），通常消息顺序进行不当，则会产生死锁。

（2）进程间推进顺序不当

例如，当P1运行到P1：Request（R2）时，将因R2已被P2占用而阻塞；当P2运行到P2：Request（R1）时，也将因R1已被P1占用而阻塞，于是发生进程死锁

产生死锁的必要条件

- 互斥条件：进程要求对所分配的资源进行排它性控制，即在一段时间内某资源仅为一进程所占用。
- 请求和保持条件：当进程因请求资源而阻塞时，对已获得的资源保持不放。
- 不剥夺条件：进程已获得的资源在未使用完之前，不能剥夺，只能在使用完时由自己释放。
- 环路等待条件：在发生死锁时，必然存在一个进程-------资源的环形链。

预防死锁的方法

- 资源一次性分配：一次性分配所有资源，这样就不会再有请求了（破坏请求条件）
- 只要有一个资源得不到分配，也不给这个进程分配其他的资源（破坏请求保持条件）
- 可剥夺资源：即当某进程获得了部分资源，但得不到其他资源，则释放已占有的资源（破坏不可剥夺条件）
- 资源有序分配法：系统给每类资源赋予一个编号，每一个进程按编号递增的顺序请求资源，释放则相反（破坏环路等待条件）

三、浏览器缓存

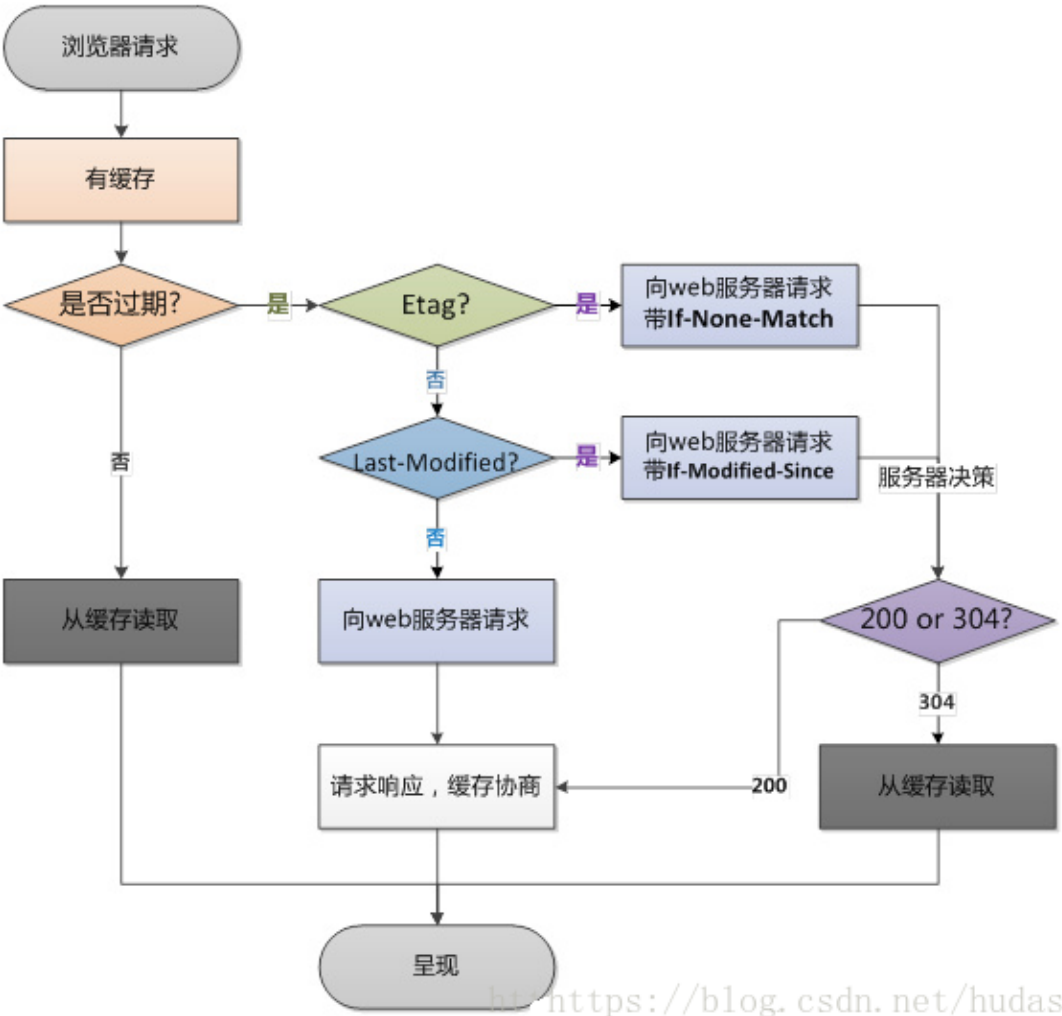
1. 对浏览器的缓存机制的理解

浏览器缓存的全过程：

- 浏览器第一次加载资源，服务器返回200，浏览器从服务器下载资源文件，并缓存资源文件与`response header`，以供下次加载时使用；
- 下一次加载资源时，由于强制缓存优先级较高，先比较当前时间与上一次返回200时的时间差，如果没有超过`cache-control`设置的`max-age`，则没有过期，并**命中强缓存**，直接从本地读取资源。如果浏览器不支持http1.1，则使用`expires`头判断是否过期。
- 如果资源已过期，则表明强制缓存没有被命中，则开始协商缓存，向服务器发送带有`if-None-Match`和`if-Modified-Since`的请求；
- 服务器受到请求后，优先根据`ETag`的值判断被请求的文件有没有做修改，`ETag`值一致则没有修改，**命中协商缓存**，返回304；如果不一致有改动，直接返回新的资源文件带上新的`ETag`值并返回200

- 如果服务器收到的请求没有`ETag`值，则将`if-Modified-Since`和被请求文件的最后修改时间做对比，一致则命中协商缓存，返回304；不一致则返回新的`last-modified`和文件并返回200.

> 注释：
>
> - `Etag`：ETag为"被请求变量的实体值"。另一种说法是，Etag是一个可以与web资源关联的记号（token），典型的web资源可以是一个web页，也可是JSON或XML文档。例：ETag:"50b1c1d4f775c61:df3"
> - `Last-Modified`：浏览器第一次请求URL时，标记文件在服务端最后被修改的时间。例：Last-Modified : Fri , 12 May 2018 18:53:33 GMT
> - `If-Modified-Since`： 客户端第二次请求URL时，浏览器会向服务器传送If-Modified-Since报头，询问该时间之后文件是否有被修改过。例：If-Modified-Since : Fri , 12 May 2018 18:53:33 GMT
> - **Expires**：Expires是RFC 2616（HTTP/1.0）协议中和网页缓存相关字段。用来控制缓存的失效日期，要注意的是，HTTP/1.0有一个功能比较弱的缓存控制机制：Pragma，使用HTTP/1.0的缓存将忽略Expires和Cache-Control头。
> - **Cache-control**：Cache-Control 是最重要的规则。这个字段用于指定所有缓存机制在整个请求/响应链中必须服从的指令。



很多网站的资源后面都加了版本号，这样做的目的是：每次升级了js文件或css文件后，为了防止浏览器进行缓存，强制改变版本号，客户端浏览器就会重新下载新的js或CSS文件，以保证用户能够及时获得网站的最新更新。

2. 浏览器资源缓存的位置有哪些?

资源缓存的位置一共有3种，按优先级从高到低分别是：

- **Service Worker**：Service Worker 运行在 JavaScript 主线程之外，虽然由于脱离了浏览器窗体无法直接访问 DOM，但是它可以完成离线缓存、消息推送、网络代理等功能。
- **Memory Cache**：内存缓存。它的效率最快，**但是内存缓存虽然读取高效，可是缓存持续性很短，会随着进程的释放而释放。**一旦我们关闭 Tab 页面，内存中的缓存也就被释放了。
- **Disk Cache**：磁盘缓存。

3. 强缓存和协商缓存

HTTP强缓存和协商缓存 - SegmentFault 思否

（1）强缓存

使用强缓存策略时，如果缓存资源有效，则直接使用缓存资源，不必再向服务器发送请求。

强缓存策略可以通过两种方式来设置，分别是http头信息中的 `Expires` 属性和 `Cache-Control` 属性

Cache-Control可以设置的字段：

- `public`：设置了该字段值的资源表示可以被任何对象缓存。
- `private`：设置了该字段值的资源只能被用户浏览器缓存，不允许任何代理服务器缓存。
- `no-cache`：设置了该字段需要先和服务端确认返回的资源是否发生了变化，如果资源未发生变化，则直接使用缓存好的资源。
- `no-store`：设置了该字段表示禁止任何缓存，每次都会向服务器发起新的请求，拉取最新的资源。
- `max-age=`：设置缓存的最大有效期，单位为秒
- `s-maxage=`：优先级高于max-age=，仅适用于共享缓存(CDN)，优先级高于max-age或者Expires头；
- `max-state[=]`：设置了该字段表明客户端愿意接收已经过期的资源，但是不能超过给定的时间限制。

一般来说只需要设置其中一种方式就可以实现强缓存策略，当两种方式一起使用时，**Cache-Control 的优先级要高于 Expires。**

（2）协商缓存

如果命中强制协商，我们无需发起新的请求，直接使用缓存内容，如果没有命中强制缓存，如果设置了协商缓存，这个时候协商缓存就会发挥作用了。

上面已经说到了，命中协商缓存的条件有两个：

- `max-age=xxx` 过期了
- 值为 `no-store`

使用协商缓存策略时，会先向服务器发送一个请求，如果资源没有发生修改，则返回一个 304 状态，让浏览器使用本地的缓存副本。如果资源发生了修改，则返回修改后的资源。

总结：

强制缓存策略和协商缓存策略在缓存命中时都会直接使用本地的缓存副本，区别只在于协商缓存会向服务器发送一次请求。他们缓存不命中时，都会向服务器发送请求来获取资源。在实际的缓存机制中，他们是一起合作使用的。浏览器首先会根据请求的信息判断，强制缓存是否命中，如果命中则直接使用资源。如果不命中则根据头信息向服务器发起请求，使用协商缓存，如果协商缓存命中的话，则服务器不返回资源，浏览器直接使用本地资源的副本，如果协商缓存不命中，则浏览器返回最新的资源给浏览器。

4. 为什么需要浏览器缓存?

所谓的 **浏览器缓存** 指的是浏览器将用户请求过的静态资源，存储到电脑本地磁盘中，当浏览器再次访问时，就可以直接从本地加载，不需要再去服务端请求了。

使用浏览器缓存的优点：

- 减少了服务器的负担，提高了网站的性能
- 加快了客户端网页的加载速度
- 减少了多余网络数据传输

5. 点击刷新按钮或者按 F5、按 Ctrl+F5 （强制刷新）、地址栏回车有什么区别?

- **点击刷新按钮或者按 F5：**浏览器直接对本地的缓存文件过期，但是会带上If-Modifed-Since，If-None-Match，这就意味着服务器会对文件检查新鲜度，返回结果可能是 304，也有可能是 200。
- **用户按 Ctrl+F5（强制刷新）：**浏览器不仅会对本地文件过期，而且不会带上 If-Modifed-Since，If-None-Match，相当于之前从来没有请求过，返回结果是 200。
- 地址栏回车： 浏览器发起请求，按照正常流程，本地检查是否过期，然后服务器检查新鲜度，最后返回内容。

四、浏览器组成

1. 对浏览器的理解

浏览器可以分为两个部分：shell 和 内核

- **shell** 是指浏览器的外壳：例如菜单，工具栏等。主要是提供给用户界面操作，参数设置等等。它是调用内核来实现各种功能的。
- **内核** 是浏览器的核心，内核是基于标记语言显示内容的程序或模块。

2. 对浏览器内核的理解

浏览器内核主要分为两部分：

- 渲染引擎的职责就是渲染，即在浏览器窗口中显示所请求的内容。默认情况下，渲染引擎可以显示HTML、XML文档及图片，它也可以借助插件显示其他类型数据，例如使用PDF阅读器插件，可以显示PDF格式。
- JS引擎：解析和执行JavaScript来实现网页的动态效果。

最开始渲染引擎和js引擎并没有区分的很明确，后来js引擎越来越独立，内核就倾向于只指渲染引擎。

4. 常见浏览器所用内核

（1） IE 浏览器内核：Trident 内核，也是俗称的 IE 内核；

（2） Chrome 浏览器内核：统称为 Chromium 内核或 Chrome 内核，以前是 Webkit 内核，现在是 Blink内核；

（3） Firefox 浏览器内核：Gecko 内核，俗称 Firefox 内核；

（4） Safari 浏览器内核：Webkit 内核；

（5） Opera 浏览器内核：最初是自己的 Presto 内核，后来加入谷歌大军，从 Webkit 又到了 Blink 内核；

（6） 360浏览器、猎豹浏览器内核：IE + Chrome 双内核；

（7） 搜狗、遨游、QQ 浏览器内核：Trident（兼容模式）+ Webkit（高速模式）；

（8） 百度浏览器、世界之窗内核：IE 内核；

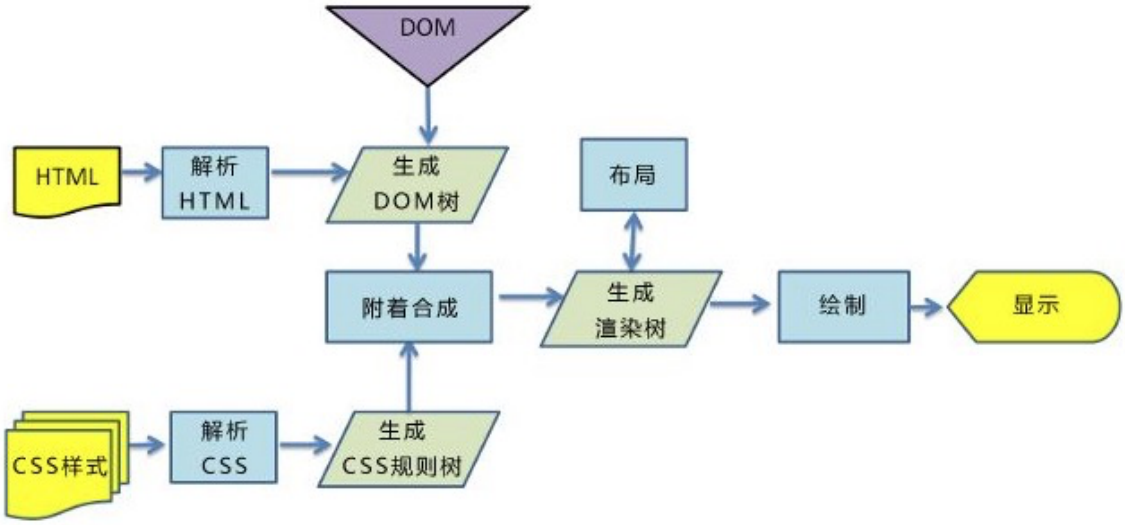
（9） 2345浏览器内核：好像以前是 IE 内核，现在也是 IE + Chrome 双内核了；

（10）UC 浏览器内核：这个众口不一，UC 说是他们自己研发的 U3 内核，但好像还是基于 Webkit 和 Trident ，还有说是基于火狐内核。

五、浏览器渲染原理

1. 浏览器渲染过程

1. 浏览器先对得到的HTML进行解析，之后进行网络资源的预处理，将以后要发送的请求提前加进请求队列中。
2. 浏览器将HTML转换为一个个的标记（标记化Tokenization），之后通过标记来构建DOM树；CSS同理，先进行标记化，再进行CSS样式树的构建。
3. 浏览器将DOM树和CSS样式树结合，生成渲染树。
4. 布局（回流）：浏览器根据渲染树，获取每个渲染树对象在屏幕上的位置和尺寸。
5. 绘制：将计算好的像素点绘制到屏幕
6. 渲染层合成：多个绘制后的渲染层按照恰当的重叠顺序进行合并，而后生成位图，最终通过显卡展示到屏幕上。



> **注意**：这个过程是逐步完成的，为了更好的用户体验，渲染引擎将会尽可能早的将内容呈现到屏幕上，并不会等到所有的html都解析完成之后再去构建和布局render树。它是解析完一部分内容就显示一部分内容，同时，可能还在通过网络下载其余内容。

2. 浏览器渲染优化

（1）针对JavaScript

JavaScript会阻塞HTML的解析，也会阻塞css的解析。

- **尽量将JavaScript文件放在body的最后面**
- **body中间尽量不要写 `<script>` 标签**
- `<script>` 标签的引入资源方式有三种，有一种就是常用的直接引入，还有两种就是使用 `async` 属性和 `defer` 属性来**异步引入**，两者都是去异步加载外部的JS文件，不会阻塞DOM的解析（**尽量使用异步加载**）。
    - **script**：立即停止渲染页面去加载资源文件，当资源加载完毕后立即执行js代码，js代码执行完毕后继续渲染页面；

- **async**：是在下载完成之后，立即异步加载，加载好后立即执行，多个async属性的标签，不能保证加载的顺序
- **defer**：是在下载完成之后，立即异步加载。加载好后，如果DOM树还没构建好，则先等待DOM树解析好后再执行；如果DOM树已经准备好，则立即执行。多个带defer属性标签，按照顺序执行。

（2）针对CSS

- **导入外部样式使用link，而不用@import。**
- **如果css少，尽可能采用内嵌样式，直接写在style标签中。**

一般使用css有三种方式：使用 **link、@import、内联样式**，其中link和@import都是导入外部样式。区别如下：

- **link**：浏览器会派发一个新的线程（HTTP线程）去加载资源文件，与此同时GIUI渲染线程会继续向下渲染代码
- **@import**：GUI渲染线程会暂时停止渲染，去服务器加载资源文件，资源文件没有返回之前不会继续渲染（阻碍浏览器渲染）。
- **style**：GUI直接渲染

> 外部样式如果长时间没有加载完毕，浏览器为了用户体验，会使用浏览器的默认样式，确保首次渲染的速度。所以CSS一般写在header中，让浏览器尽快发请求去获取CSS样式。

（3）针对DOM树、CSSOM树

- HTML文件的代码层级尽量不要太深
- 使用语义化的标签，来避免不标准语义化的特殊处理
- 减少CSSD代码的层次，因为选择器是从左向右进行解析的。

（4）减少回流与重绘

渲染队列：浏览器会将所有的回流、重绘的操作放在一个队列中，当队列中的操作到了一定的数量或者到了一定的时间间隔，浏览器就会对队列进行批处理。这样就会让多次回流、重绘变成一次回流重绘。

- 操作DOM时，尽量在低层级的DOM节点进行操作
- 不要使用table布局
- 使用CSS的表达式
- 不要频繁的操作元素的样式，对于静态页面，可以修改类名，而不是样式。
- 使用absolute或fixed，使元素脱离文档流，这样他们发生变化就不会影响其他元素。
- 将元素设置`display:none`，操作结束后再把它显示出来。因为在display属性为none的元素上进行DOM操作不会引发回流和重绘。
- 使用CSS3中的`transform，opacity，filters`属性，启动GPU加速，这些属性的改变不会引发回流或重绘。

3. 什么是文档的预解析?

Webkit 和 Firefox都做了这个优化，当执行JavaScript脚本时，另一个线程解析剩下的文档，并加载后面需要通过网络加载的资源。这种方式可以使资源并行加载从而使整体速度更快。

需要注意的是，预解析并不改变DOM树，它将这个工作留给主解析过程，自己只解析外部资源引用，比如外部脚本、样式表以及图片。

4. css如何阻塞文档解析?

理论上，既然样式表不改变DOM树，也就没有必要停下文档的解析等待他们。然而，存在一个问题，JavaScript脚本执行时可能在文档的解析过程中请求样式信息，如果样式还没有加载和解析，脚本将得到错误的值，显然这将会导致很多问题。所以如果浏览器尚未完成CSSOM的下载和构建，而我们却想在此时运行脚本，那么浏览器将延迟JavaScript脚本执行和文档的解析，直到其完成CSSOM的下载和构建。

也就是说，在这种情况下，浏览器会先下载和构建CSSOM，然后再执行JavaScript，最后再继续文档的解析。

六、浏览器本地存储

1. 浏览器本地存储方式以及使用场景

① Cookie 4KB

Cookie是最早被提出来的本地存储方式，在此之前，服务端是无法判断网络中的两个请求是否是同一用户发起的，为解决这个问题，Cookie就出现了。Cookie的大小只有4KB，它是一种纯文本文件，每次发起http请求都会携带Cookie

Cookie的特性：

- Cookie一旦创建成功，名称就无法修改
- **Cookie是无法跨域名的**，也就是说a域名和b域名下的Cookie是无法共享的，这也是由Cookie的隐私安全性决定的，这样能够阻止非法获取其他网站的Cookie。
- 每个域名下的Cookie的数量不能超过20个，每个Cookie的大小不超过4kb
- 有安全问题，如果Cookie被拦截了，那就可获得session的所有信息，即使加密也于事无补。
- Cookie在请求一个新的页面的时候都会被发送过去。

如果需要域名之间跨域共享Cookie，有两种方法：

1. 使用Nginx代理
2. 在一个站点登录之后，往其他网站写Cookie。服务端的Session存储到一个节点，Cookie存储SessionId。

Cookie的使用场景：

- 最常见的使用场景就是cookie和session结合使用，我们将sessionId存储到Cookie中，每次发送请求都会携带这个sessionId，这样服务器就知道是谁发起的请求，从而响应相应的信息。
- 可以用来统计页面的点击次数。

② LocalStorage 5M

LocalStorage是html5新引入的特性，由于有时候存储的信息量较大，Cookie就不能满足我们的需求，这时候LocalStorage就可以使用了。

优点：

- 大小一般为5M，可以存储更多的信息
- 是**持久性存储**，并不会随着页面的关闭而消失，除非主动清除，不然会永久存在。
- **仅存储在本地**，不像Cookie那样每次HTTP请求都会被携带

缺点：

- 存在浏览器兼容问题，IE8以下版本的浏览器不支持
- 如果浏览器设置为隐私模式，那么我们将无法读取LocalStorage
- LocalStorage**受到同源策略的限制**，即端口、协议、主机地址有任何一个不相同，都不会访问。

LocalStorage常用API：

```
//保存数据到LocalStorage
LocalStorage.setItem('key','value');
//从localStorage获取数据
let data = localStorage.getItem('key');
//从LocalStorage删除保存的数据
localStorage.removeItem('key');
//从localStorage删除所有保存的数据
localStorage.clear();
//获取某个索引的key
localStorage.key(index)
```

使用场景:

- 有些网站有换肤的功能，这时候就可以将换肤的信息存储在本地的LocalStorage中，当需要换肤的时候，直接操作LocalStorage即可。
- 在网站中的用户浏览信息也会存储在LocalStorage中，还有网站的一些不常变动的个人信息等也可以存储在本地的LocalStorage中

③ SessionStorage 5M

也是HTML5提出来的存储方案，sessionStorage主要用于**临时保存**同一窗口（或标签页）的数据，刷新页面是不会删除，关闭窗口或标签页之后将会删除这些数据。

SessionStorage和LocalStorage对比:

- SessionStorage和LocalStorage都在**本地进行数据存储**
- SessionStorage也有同源策略的限制，但是SessionStorage **只有在同一浏览器的同一窗口下才能够共享**;
- 两者都不能被爬虫爬取。

SessionStorage的常用API:

```
// 保存数据到 sessionStorage
sessionStorage.setItem('key', 'value');
// 从 sessionStorage 获取数据
let data = sessionStorage.getItem('key');
// 从 sessionStorage 删除保存的数据
sessionStorage.removeItem('key');
// 从 sessionStorage 删除所有保存的数据
sessionStorage.clear();
// 获取某个索引的Key
sessionStorage.key(index)
```

使用场景:

由于sessionStorage具有时效性，所以可以用来存储一些网站的游客登录的信息，还有**临时**的浏览记录的信息。当关闭网站之后，这些信息也就随之消除了。

2. cookie有哪些字段?

- **Name**：名称
- **Value**：值，对于认证cookie，value值包括web服务器所提供的访问令牌
- **Size**：大小
- **Path**：可以访问cookie的页面路径
- **Secure**：指定是否使用HTTPs安全协议发送Cookie。

- **Domain**:可以访问该cookie的域名，cookie机制并未遵循严格的同源策略，允许一个子域可以设置或获取其父域的Cookie。
- **HTTP**：该字段包含`HTTPOnly`属性，该属性用来设置cookie能否通过脚本来访问，默认为空，即可以通过脚本访问。
- **Expires/Max-size**：此cookie的超时时间。

总结：

服务端可以使用 **Set-Cookie** 的响应头部来配置 cookie信息。一条cookie包括了5个属性值 **expires**、**domain**、**path**、**secure**、**httpOnly**。

- **Expires**：指定cookie的失效时间
- **domain和path**：限制cookie能够哪些url访问。
- **secure**：规定了Cookie只能在确保安全的情况下传输。
- **httpOnly**：规定这个cookie只能被服务器访问，不能使用js脚本访问

3. 前端存储的方式有哪些?

- **Cookie**：由服务器设置，在客户端存储
- **localStorage**：永久性存储，在本地存储
- **SessionStorage**：临时存储，在本地存储
- **Web SQL**：2010年被W3C废弃的本地数据库存储方案。
- **IndexedDB**：是被正式纳入HTML5标准的数据库存储方案，它是NoSQL数据库，用键值对进行存储，可以进行快速读取操作，非常适合web场景，同时用JavaScript进行操作会非常方便。

七、浏览器同源策略

1. 什么是同源策略

跨域问题其实就是浏览器的同源策略引起的。

同源：指的是 协议（protocol）、端口（port）、域名（domain）必须一致。

同源策略：只有浏览器才受到同源策略的限制。即不同源的脚本在没有授权的情况下，不能读写对方的资源。比如我们打开了`A.com`，之后又在没有许可的情况下向`B.com`发送请求，通常这个请求是失败的。想要摆脱同源策略的限制，就要使用跨域的手段。

同源策略主要限制了三个方面：

- 当前域下的js脚本不能够访问其他域下的cookie、localStorage、indexDB。
- 在当前域下的js脚本不能操作访问其他域下的DOM
- 当前域下ajax无法发送跨域请求。

2. 如何解决跨域问题

> 最常用的是 CORS 和 反向代理
>
> 参考理解

CORS

> `CORS`（Cross-origin ResourceSharing）跨域资源共享机制，它使用额外的http头来告诉浏览器让运行在一个origin（domain）上的web应用被准许访问来自不同源服务器上的指定资源。当一个资源从该资源本身所在的服务器不同的域、协议或端口请求一个资源时，资源会发起一个跨域http请求。

CORS的关键是服务器，只要服务器实现了CORS请求，就可以跨源通信了。

只需要后端在响应头设置 `Access-Control-Allow-Origin: *`，*表示任意origin，也可以指定Origin。

使用CORS时默认不发送Cookie，想要发送Cookie需要：

1. 设置 `Access-Control-Allow-Credentials:true`
2. 此时 `Access-Control-Allow-Origin` 不能设置为*，必须指定Origin

浏览器将CORS分为 **简单请求**和 **非简单请求**：

简单请求不会触发CORS **预检请求。**

简单请求：

> 在简单请求中，在服务器内，至少需要设置字段： `Access-Control-Allow-Origin`

- 请求方法是：HEAD、GET、POST
- HTTP的头信息不超出以下几种字段：
  - Accept
  - Accept-Language
  - Content-Language
  - Last-Event-ID
  - Content-Type：只限制于三个值 `application/x-www-form-urlencoded`、`multipart/form-data`、`text/plain`

<u>若不满足以上条件，就属于非简单请求了</u>

（1）简单请求过程：

对于简单请求，浏览器会直接发出CORS请求，它会在请求头信息中增加一个Origin字段，该字段用来说明本次请求来自哪个源（协议+端口+域名），服务器会根据这个值来决定是否同意这次请求。如果Origin指定的域名在许可范围之内服务器返回的响应就会多出以下信息头：

```
Access-Control-Allow-Origin: http://api.bob.com  // 和Orign一直
Access-Control-Allow-Credentials: true   // 表示是否允许发送Cookie
Access-Control-Expose-Headers: FooBar   // 指定返回其他字段的值
Content-Type: text/html; charset=utf-8   // 表示文档类型
```

如果Origin指定的域名不在许可范围之内，服务器会返回一个正常的http回应，浏览器发现没有上面的Access-Control-Allow-Origin头部信息，就知道出错了。这个错误时无法通过状态码识别，因为返回的状态码可能是200.

（2）非简单请求

非简单请求的CORS请求会在正式通信之前进行一次HTTP查询请求，称为 **预检请求。**

预检请求使用的方法是 **OPTIONS**，表示这个请求是来询问的。他的头信息中包括以下字段：

- **Access-Control-Request-Origin**：表示请求来自哪个源。
- **Access-Control-Request-Method**：该字段是必须的，用来列出浏览器的CORS请求会用到哪些HTTP方法。
- **Access-Control-Request-Headers**：该字段是一个逗号分隔的字符串，指定浏览器CORS请求会额外发送的头信息字段。

服务器在收到浏览器的预检请求之后，会根据头信息的三个字段来进行判断，服务器回应的CORS字段如下：

```
Access-Control-Allow-Origin: http://api.bob.com  // 允许跨域的源地址
Access-Control-Allow-Methods: GET, POST, PUT // 服务器支持的所有跨域请求的方法
Access-Control-Allow-Headers: X-Custom-Header  // 服务器支持的所有头信息字段
Access-Control-Allow-Credentials: true   // 表示是否允许发送Cookie
Access-Control-Max-Age: 1728000  // 用来指定本次预检请求的有效期，单位为秒
```

如果返回的头信息中有 Access-Control-Allow-Origin这个字段就是允许跨域请求。

CORS跨域的判定流程：

1. 浏览器先根据同源策略对前端页面和后台交互地址做匹配，若同源，则直接发送数据请求；若不同源，则发送跨域请求。
2. 服务器收到浏览器跨域请求后，根据自身配置返回对应文件头。若未配置过任何允许跨域，则文件头里不包含Access-Control-Allow-Origin字段，若配置过域名，则返回Access-Control-Allow-origin + 对应配置规则里的域名的方式。
3. 浏览器根据接受到的 响应头里的 Access-Control-Allow-origin 字段做匹配，若无该字段，说明不允许跨域，从而抛出一个错误；若有该字段，则对字段内容和当前域名做比对，如果同源，则说明可以跨域，浏览器接受该响应；若不同源，则说明该域名不可跨域，浏览器不接受该响应，并抛出一个错误。

JSONP

jsonp的原理就是利用 `<script>` 标签没有跨域限制，通过 `<script>` 标签src属性，发送带有callback参数的GET请求，服务端将接口返回的数据拼凑到callback函数中，返回给浏览器，浏览器解析执行，从而前端拿到callback函数返回的数据。

```
<script>
    var script = document.createElement('script');
    script.type = 'text/javascript';
    // 传参一个回调函数名给后端，方便后端返回时执行这个在前端定义的回调函数
    script.src = 'http://www.domain2.com:8080/login?
user=admin&callback=handleCallback';
    document.head.appendChild(script);
    // 回调执行函数
    function handleCallback(res) {
        alert(JSON.stringify(res));
    }
 </script>
```

服务端返回如下（返回时即执行全局函数）：

```
handleCallback({"success": true, "user": "admin"})
```

缺点：仅支持get方法，不安全，可能会受到XSS攻击。

postMessage 跨域

postMessage原先是用来实现线程与主线程之间的消息通信的。

postMessage是HTML5 XMLHttpRequest Level2中的API。允许来自不同源的脚本采用异步方式进行有效通信，可以实现跨文本文档，多窗口，跨域消息传递，多用于窗口间数据通信，这也使它成为跨域通信的一种有效的解决方案。

用法：`postMessage(data,origin)`

```
<!--domain1.com/a.html-->
<iframe id="iframe" src="http://www.domain2.com/b.html" style="display:none;">
</iframe>
<script>
  var iframe = document.getElementById('iframe');
  iframe.onload = function() {
    var data = {
      name: 'aym'
    };
    // 向domain2传送跨域数据
    iframe.contentWindow.postMessage(JSON.stringify(data),
'http://www.domain2.com');
  };
  // 接受domain2返回数据
  window.addEventListener('message', function(e) {
    alert('data from domain2 ---> ' + e.data);
  }, false);
</script>

<!--domain2.com/b.html-->
<script>
  // 接收domain1的数据
  window.addEventListener('message', function(e) {
    alert('data from domain1 ---> ' + e.data);
    var data = JSON.parse(e.data);
    if (data) {
      data.number = 16;
      // 处理后再发回domain1
      window.parent.postMessage(JSON.stringify(data), 'http://www.domain1.com');
    }
  }, false);
</script>
```

反向代理

由于同源策略是浏览器的策略。

`A.com:80` 不能向 `B.com:3000` 发送请求。那么我们可以在 `A.com:8080` 设置一个代理服务器来代理请求，之后发请求就是 `A.com:80 -> A.com:8080 ->B.com:3000`,此时请求可以成功发过去。

> 注：关于这里为什么80可以向8080发送请求，而没有导致跨域的原因，是因为这个代理服务器设置了允许跨域。可能又会问，那为什么不直接在目标服务器上设置允许跨域呢？其实如果直接在目标服务器上设置允许跨域那就是CORS请求咯，但是使用代理服务器的好处就是可以隐藏真实的服务器或者是隐藏客户端 并且还可以实现负载均衡等功能啊。
>
> 所有这两种方法怎么用还是得看场景咯。

通常我们本地开发项目是使用 `webpack-dev-server`，而它自带了代理服务器的功能（只需要我们在配置文件中加上 `proxy`），所以可以轻松解决跨域问题。除此之外我们也可以使用 `nginx` 来反向代理。

nginx代理跨域

nginx代理跨域，实质和CORS跨域原理一样，通过配置文件设置请求响应头Access-Control-Allow-Origin...等字段

（1）nginx配置解决iconfont跨域

浏览器跨域访问js、css、img等常规静态资源被同源策略许可，但iconfont字体文件（eot|otf|ttf|woff|svg）例外，此时可在nginx的静态资源服务器中加入以下配置

```
location / {
    add_header Access-Control-Allow-Origin *;
}
```

（2）nginx反向代理接口跨域

跨域问题：同源策略仅是针对浏览器的安全策略。服务器端用HTTP接口只是使用HTTP协议，不需要同源策略，也就不存在跨域问题。

实现思路：通过Nginx配置一个代理服务器域名与domain1相同，端口不同）做跳板机，反向代理访问domain2接口，并且可以顺便修改cookie中domain信息，方便当前域cookie写入，实现跨域访问。

```
#proxy服务器
server {
    listen       81;
    server_name www.domain1.com;
    location / {
        proxy_pass  http://www.domain2.com:8080;  #反向代理
        proxy_cookie_domain www.domain2.com www.domain1.com; #修改cookie里域名
        index  index.html index.htm;
        # 当用webpack-dev-server等中间件代理接口访问nignx时，此时无浏览器参与，故没有同源限制，下面的跨域配置可不启用
        add_header Access-Control-Allow-Origin http://www.domain1.com;  #当前端只跨域不带cookie时，可为*
        add_header Access-Control-Allow-Credentials true;
    }
}
```

document.domain

此方案仅限主域相同，子域不同的跨域应用场景。

实现原理：两个页面都通过JS强制设置document.domain为基础主域，就实现了同域。

Cookie 是服务器写入浏览器的一小段信息，只有同源的网页才能共享。但是，两个网页一级域名相同，只是二级域名不同，浏览器允许通过设置`document.domain`共享 Cookie。

如a.example.com和b.example.com。

此时两个网站都设置`document.domain = "example.com"`，那么两个网页就可以共享Cookie了。

```
//a.example.com
document.cookie = 'aaa'
//b.examlpe.com
console.log(document.cookie) //'aaa'
```

location.hash + iframe 跨域

实现原理：a欲与b跨域相互通信，通过中间页c来实现。三个页面，不同域之间利用iframe的location.hash传值，相同域之间直接js来访问

具体实现：A域（a.html）-> B域（b.html）-> A域（c.html）。a与b不同域只能通过hash值单向通信，b与c也不同域也只能单向通信，但c与a同域，所以可以通过parent.parent访问a页面所有对象。

（1）a.html：（domain1.com/a.html)

```
<iframe id="iframe" src="http://www.domain2.com/b.html" style="display:none;">
</iframe>
<script>
    var iframe = document.getElementById('iframe');
    // 向b.html传hash值
    setTimeout(function() {
        iframe.src = iframe.src + '#user=admin';
    }, 1000);

    // 开放给同域c.html的回调方法
    function onCallback(res) {
        alert('data from c.html ---> ' + res);
    }
</script>
```

（2）b.html：(.domain2.com/b.html)

```
<iframe id="iframe" src="http://www.domain1.com/c.html" style="display:none;">
</iframe>
<script>
    var iframe = document.getElementById('iframe');
    // 监听a.html传来的hash值，再传给c.html
    window.onhashchange = function () {
        iframe.src = iframe.src + location.hash;
    };
</script>
```

3）c.html：(http://www.domain1.com/c.html)

```
<script>
    // 监听b.html传来的hash值
    window.onhashchange = function () {
        // 再通过操作同域a.html的js回调，将结果传回
        window.parent.parent.onCallback('hello: ' +
location.hash.replace('#user=', ''));
    };
</script>
```

window.name

这个方法主要用于**父窗口和iframe窗口的通信。**

如果父窗口和iframe窗口是不同源的，则通常无法进行通信。

```
<html>
    <body>
        <!-- 我是父窗口 -->
        <iframe src='xxx.com'>
            <!-- 我是子窗口 -->
        </iframe>
    </body>
</html>
```

`window.name`特点：无论是否同源，只要在同一个窗口里，前一个网页设置了这个属性，后一个网页可以读取它。

例如，我们在a.com页面下设置

```
window.name = '123'
location.href = 'b.com'
```

然后在b.com也能获取到`window.name`的值。

实现跨域：

使用时，先设置`iframe`的`src`为我们想要通信的目标页面。当目标页面的`window.name`修改时，将我们的`iframe`的`src`修改为一个和父窗口同源的页面。

本质：

iframe内的目标页面 <=> iframe内的一个和父窗口同源的页面 <=> 父窗口

总结

- CORS
- JSONP
- 反向代理
- postMessage
- document.domain
- location.hash + iframe
- window.name

3. 正向代理和反向代理的区别

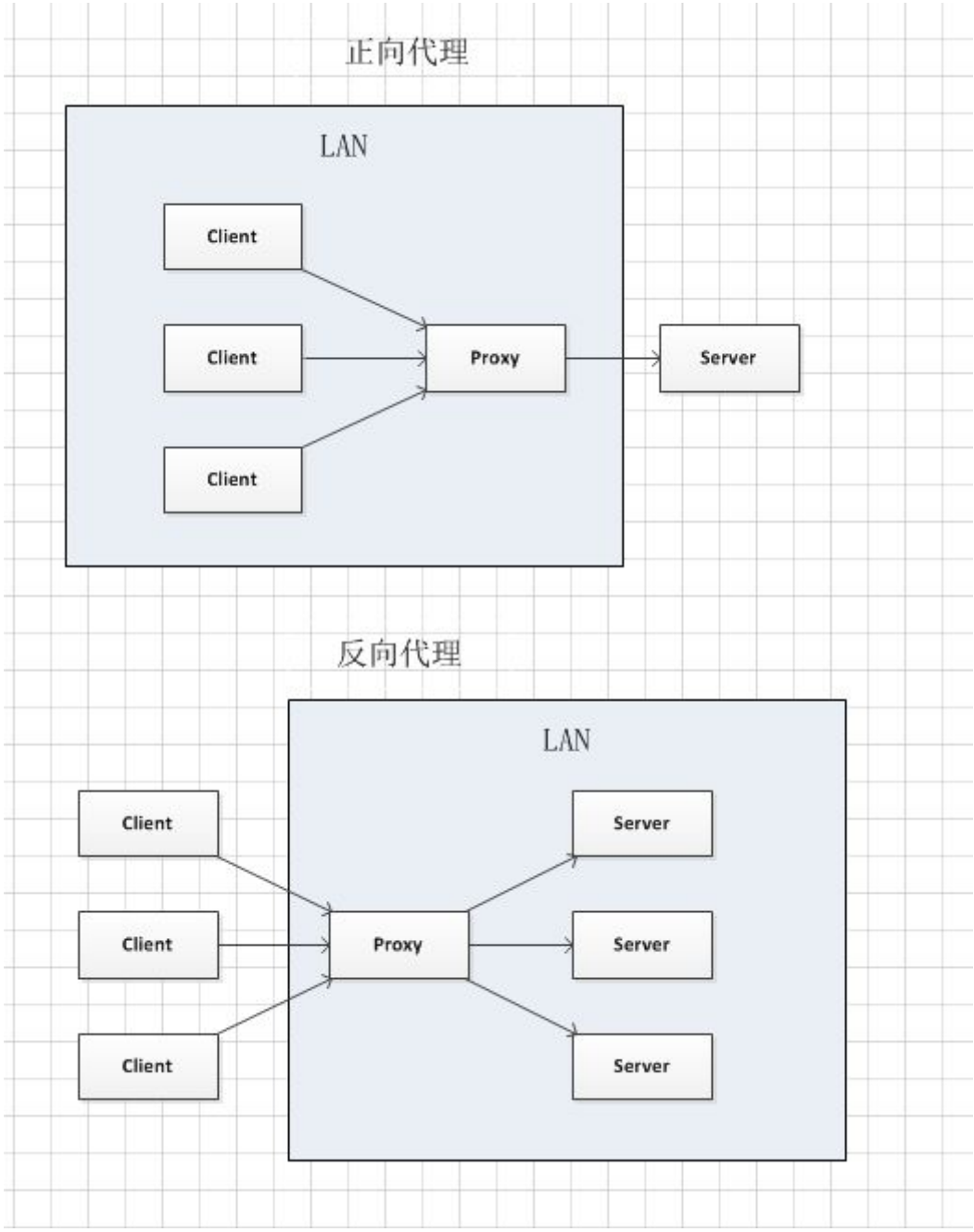
正向代理

客户端想要获取一个服务器的数据，但是因为种种原因无法直接获取。于是客户端设置了一个代理服务器，并且指定目标服务器，之后代理服务器向目标服务器转交请求并将获得的内容发送给客户端。这样本质上起到了对真实服务器隐藏真实客户端的目的。**实现正向代理需要修改客户端，比如修改浏览器配置。**

反向代理

服务器为了能够将工作负载分发到多个服务器来提高网站性能（负载均衡）等目的，当其受到请求后，会首先根据转发规则来确定请求应该被转发到哪个服务器上，然后将请求转发到对应的真实服务器上。这样本质上起到了对客户端隐藏真实服务器的作用。

一般使用反向代理后，需要通过修改DNS让 域名解析到代理服务器IP，这时浏览器无法察觉到真正的服务器的存在，当然也就不需要修改配置了。



> 正向代理和反向代理的结构是一样的，都是client-proxy-server的结构，他们主要的区别就在于中间这个proxy是哪一方设置的。在正向代理中，proxy是client设置的，用来隐藏client；在反向代理中，proxy是server设置的，用来隐藏server。

八、浏览器事件机制

1. 事件是什么？事件模型?

事件是用户操作网页时发生的交互动作，比如click/move，事件除了用户触发的动作外，还可以是文档加载，窗口滚动和大小调整。事件被封装成一个event对象，包含了该事件发生时的所有相关信息（event 的属性）以及可以对事件进行的操作（event方法）。

事件是用户操作网页时发生的交互动作或者网页本身的一些操作，现代浏览器一共有三种事件模型：

- **DOM 0级事件模型**，这种模型不会传播，所以没有事件流的概念，但是现在有的浏览器支持以冒泡的方式实现，它可以在网页中直接定义监听函数，也可以通过js属性来指定监听函数。所有浏览器都兼容这种方式。直接在dom对象上注册事件名称，就是DOM0 写法。

- **IE 事件模型**，该事件模型中，一次事件共有两个过程，**事件处理**阶段和**事件冒泡**阶段。事件处理阶段会首先执行目标元素绑定的监听事件。然后是事件冒泡阶段，冒泡指的是事件从目标元素冒泡到document，依次检查经过的节点是否绑定了事件监听函数，如果有则执行。这种模型通过attachEvent来添加监听函数，可以添加多个监听函数，会按顺序依次执行。
- **DOM2级事件模型**，在该事件模型中，一次事件共有三个过程，第一个过程是**事件捕获**阶段。捕获指的是事件从document一直向下传播到目标元素，依次检查经过的节点是否绑定了事件监听函数，如果有则执行。后面两个阶段和IE事件模型的两个阶段相同。这种事件模型，事件绑定的函数是addEventListener，其中第三个参数可以指定事件是否在捕获阶段执行。

2. 如何阻止事件冒泡

- 普通浏览器使用：`event.stopPropagation()`
- IE浏览器使用：`event.cancelBubble = true`

3. 对事件委托的理解

（1）事件委托的概念

事件委托本质上是利用了 **浏览器事件冒泡** 的机制。因为事件在冒泡过程中会上传到父节点，父节点可以通过事件对象获取到目标节点，因此可以把子节点的监听函数定义在父节点上，由父节点的监听函数统一处理多个子元素的事件，这种方式称为**事件委托（事件代理）**。

使用事件委托可以不必要为每一个子元素都绑定一个监听事件，这样减少了内存上的消耗。并且使用事件代理还可以实现事件的动态绑定，比如说新增了一个子节点，并不需要单独为它添加一个监听事件，它绑定的事件会交给父元素中的监听函数来处理。

（2）事件委托的特点

- 减少内存消耗
- 动态绑定事件

例子：如果有一个列表，列表之中有大量的列表项，需要在点击列表项的时候响应一个事件，我们可以把点击事件绑定在他的父级，也就是ul上。

```
<ul id="list">
    <li>item 1</li>
    <li>item 2</li>
    ...
    <li>item n</li>
</ul>
<script>
//给父层元素绑定事件
document.getElementById('list').addEventListener('click',function(e){
    //兼容性处理
    var event = e || window.event;
    var target = event.target || event.srcElement;
    //判断是否匹配目标元素
    if(target.nodeName.toLocaleLowerCase === 'li'){
        console.log('the content is:',target.innerHTML);
    }
})
</script>
```

在上述代码中，target元素则是在#list元素之下具体被点击的元素，然后通过判断target的一些属性（比如：nodeName，id等等）可以更精确地匹配到某一类#list li 元素之上。

（3）事件委托的局限性

focus、blur之类的事件没有事件冒泡机制，所以无法实现事件委托；mousemove、mouseout这样的事件，虽然有事件冒泡，但是只能不断通过位置去计算定位，对性能消耗高，因此也是不适合于事件委托的。

事件委托会影响页面的性能，主要影响因素有：

- 元素中，绑定事件委托的次数；
- 点击最底层的元素，到绑定事件元素之间的DOM层数

在必须使用事件委托的地方，可以进行如下处理：

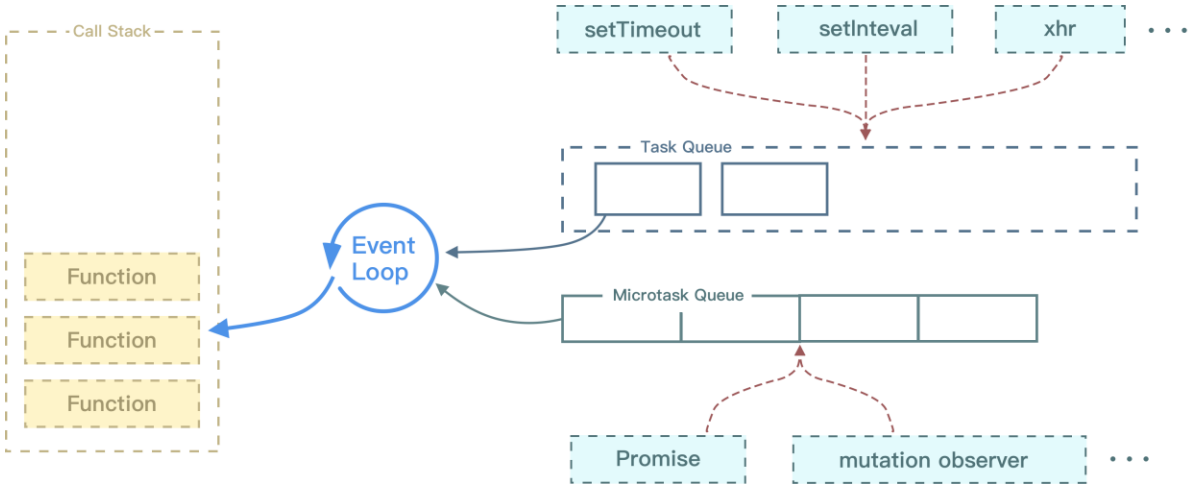
- 只在必须的地方，使用事件委托，比如：ajax的局部刷新区域
- 尽量的减少绑定的层级，不在body元素上进行绑定
- 减少绑定的次数，如果可以，那么把多个事件绑定，合并到一次事件委托中去，由这个事件委托的回调，来进行分发。

4. 同步和异步的区别

- **同步**指的是当一个进程在执行某个请求时，如果这个请求需要等待一段时间才能返回，那么这个进程会一直等待下去，直到消息返回为止再继续向下执行。
- **异步**指的是当一个进程在执行某个请求时，如果这个请求需要等待一段时间才能返回，这个时候进程会继续往下执行，不会阻塞等待消息的返回，当消息返回时系统再通知进程进行处理。

5. 对事件循环的理解

因为js是单线程运行的，在代码执行时，通过将不同函数的执行上下文压入执行栈中来保证代码的有序执行。在执行同步代码时，如果有遇到异步事件，js引擎并不会一直等待其返回结果，而是会将整个事件挂起，继续执行执行栈中的其他任务。当异步事件执行完毕后，在将异步事件对应的回调加入到一个任务队列中等待执行。任务队列可分为宏任务队列和微任务队列，当当前执行栈中的事件执行完毕后，js引擎会首先判断微任务队列中是否有任务可以执行，如果有就将微任务队首的事件压入栈中执行。当微任务队列中的任务都执行完成后再去执行宏任务队列中的任务。



事件循环（event Loop）执行顺序如下所示：

- 首先执行同步代码，这属于宏任务（也就是所说的script脚本中的代码）
- 当执行完所有同步代码后，执行栈为空，查询是否有异步代码需要执行
- 执行所有微任务
- 当执行完所有微任务后，如有必要会渲染页面
- 然后开始下一轮event Loop，执行宏任务中的异步代码

6. 什么是执行栈?

可以把执行栈认为是一个存储函数调用的**栈结构**，遵循先进后出的原则。

当开始执行js代码时，根据先进后出的原则，后执行的函数会先弹出栈。

```
function foo(b) {
    var a = 5;
    return a * b + 10;
}

function bar(x) {
    var y = 3;
    return foo(x * y);
}

console.log(bar(6));
```

输出：

7. 事件触发的过程是怎样的?

事件触发有三个阶段：

- `window`往事件触发处传播，遇到注册的捕获事件会触发
- 传播到事件触发处时触发注册的事件
- 从事件触发处往`window`传播，遇到注册的冒泡事件会触发。

事件触发一般来说会按照上面的顺序执行，但是也有特例，**如果给一个body中的子节点同时注册冒泡和捕获事件，事件触发会按照注册的顺序执行。**

九、浏览垃圾回收机制

- 浏览器垃圾回收机制
- 深入理解Chrome V8垃圾回收机制
- 图解v8垃圾回收机制

对于栈的内存空间，只保存简单数据类型的内存，由操作系统自动分配和自动释放。而堆空间中的内存，由于大小不固定，系统无法进行自动释放，这个时候就需要js引擎来手动的释放这些内存。当我们的代码没有按照正确的写法时，会使得js引擎的垃圾回收机制无法正确的对内存进行释放（内存泄漏），从而使得浏览器占用的内存不断增加，进而导致JavaScript和应用、操作系统性能下降。

1. 什么是内存泄漏？哪些操作会造成内存泄漏?

`内存泄漏`**是指你向系统申请分配内存进行使用，然后系统在堆内存中给这个对象申请一块内存空间，但当我们使用完了却没有归系统，导致这个不使用的对象一直占据内存单元，造成系统将不能再把它分配给需要的程序。**

- 第一种是由于使用未声明的变量，而意外的创建了一个全局变量，而使这个变量一直留在内存中无法被回收。
- 第二种情况是设置了 setInterval 定时器，而忘记取消它，如果循环函数有对外部变量的引用的话，那么这个变量会被一直留在内存中，而无法被回收。
- 第三种情况是获取一个DOM元素的引用，而后面这个元素被删除，由于我们一直保留了这个元素的引用，所以它也无法被回收
- 第四种情况是不合理的使用闭包，从而导致某些变量一直被留在内存中。