十一面试题准备（1）

HTML标签的分类[熟悉]

HTML文档标签

1. `<!DOCTYPE>`：定义文档类型。
2. `<html>`： 定义HTML文档。
3. `<head>`：定义文档的头部。 `<meta>`：定义元素可提供的有关页面的元信息，比如针对搜索引擎和更新频度的描述和关键字。 `meta`标签共有两个属性， `http-equiv`属性：相当于http头文件的作用，向浏览器传回一些有用的信息，使用content规定属性的内容比如

```
1.<meta http-equiv="Set-Cookie" content="cookievalue=xxx; expires=Friday,12-Jan-2001 18:18:18
2.<meta http-equiv='expires' content='时间' >: 用于设定网页的到期时间。一旦网页过期，必须到服务器
3.<meta http-equiv="Refresh" content="5;URL">: 告诉浏览器在【数字】秒后跳转到【一个网址】
4.<meta http-equiv="content-Type" content="text/html; charset=utf-8">: 设定页面使用的字符集。
  <meta charset="utf-8">: 在HTML5中设定字符集的简写写法。
5.<meta http-equiv="Pragma" content="no-cache">: 禁止浏览器从本地计算机的缓存中访问页面内容。访问
6.<meta http-equiv="Window-target" content="_top">: 用来防止别人在iframe(框架)里调用自己的页面，
7.<meta http-equiv='X-UA-Compatible' content='IE=edge,chrome=1'> :强制浏览器按照特定的版本标准
```

`name`属性：主要用来描述网页，content中的内容主要是便于搜索引擎机器人查找信息和分类信息用的。

```
1.<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, user-sc
2.<meta name="description" content="">: 告诉搜索引擎，当前页面的主要内容是xxx。
3.<meta name="keywords" content="">: 告诉搜索引擎，当前页面的关键字。
4.<meta name="author" content="">: 告诉搜索引擎，标注网站作者是谁。
5.<meta name="copyright" content="">: 标注网站的版权信息。
```

`<base>`：定义页面上的所有链接规定默认地址或默认目标。

`<title>`：定义文档标题。

`<link>`：定义文档与外部资源的关系。

`<style>`：定义HTML文档样式信息。

4. `<body>`：定义文档的主体。（脚本在非必需情况时在的最后） `<script>`：定义客户端脚本，比如javascript。 `<noscript>`：定义脚本未被执行时的替代内容。（如文本）

按闭合特征分类：

闭合标签是指由开始标签和结束标签组成的一对标签，这种标签允许嵌套和承载内容，例如`<html></html>`、`<p></p>`等。

空标签是指没有内容的标签，在开始标签中自动闭合。常见的空标签有：`<br>`、`<hr>`、 `<img>`、`<input>`、 `<link>` `<meta>`。

按是否换行特征分类：

- 块级元素：块级元素是值本身属性为display:block的元素。
- 内联元素：内联元素是指本身属性为display:inline的元素，其宽度随元素的内容而变化。

常用块级元素：

标签	描述
div	常用块级容器，也是css和layout的主要标签
h1、h2、h3	一、二、三级标题
h4、h5、h6	四、五、六级标题
hr	水平分隔线
menu	菜单列表
ol、ul、li	有序列表、无序列表、列表项
dl、dt、dd	定义列表、定义术语、定义描述
table	表格
p	段落
form	交互表单

H5新特性（常用）：

标签	描述
<time>	定义日期或时间
<progress>	状态标签（任务过程：安装、加载）
<canvas>	定义图形，比如图标和其他图像。，canvas作容器，用脚本绘制图形。
<dialog>	定义对话框或窗口。
<video>	定义视频，像电影片段或其他视频流而不用再使用第三方插件。现在主要支持 Ogg 和 MPEG 两种视屏格式。
<audio>	定义音频，如音乐或其他音频流。
<keygen>	定义表单里一个生产的键值，加密信息传送。

HTML5新增元素[了解]

1. 标签增删 8个语义元素 `header` `section` `footer` `aside` `nav` `main` `article` `figure` 内容元素 `mark` 高亮 `progress` 进度 新的表单控件 `calander` `date` `time` `email` `url` `search` 新的input类型 `color` `date` `datetime` `datetime-local` `email` 移除过时标签 `big` `font` `frame` `frameset`
2. `canvas` 绘图，支持内联SVG。支持MathML
3. 多媒体 `audio` `video` `source` `embed` `track`
4. 本地离线存储，把需要离线存储在本地的文件列在一个manifest配置文件
5. web存储, `localStorage` 、 `SessionStorage`

cookie sessionStorage localStorage区别[精通]

Cookie属性详解

- `Name` 字段：为一个cookie的名称
- `Value` 字段：为一个cookie的值 `Domain` 字段：为可以访问此cookie的域名，二级域名可以获取顶级域名中的cookie，同二级域名之间不可相互获取，顶级域名不可获取二级域名的cookie
- `Path` 字段：为可以访问此cookie的页面路径，只有此路径下的页面可以读取此cookie
- `Expires/Max-Age` 字段：为此cookie的超时时间，若设置为一个时间，则当达到此时间之后，此cookie失效，不设置的话默认值为Session，即浏览器关闭后才会失效。
- `Size` 字段：表示此cookie的大小

- `http`字段：表示此cookie的httponly属性，若此属性设置为true，则只有在http请求头中会带有此cookie的信息，而不能通过document.cookie来访问。
- `Secure`字段：表示设置是否只能通过https来传递此条cookie。

Cookie与webstorage：

- cookie数据始终在同源的http请求中携带(即使不需要)，即cookie在浏览器和服务器间来回传递。
- cookie数据还有路径（path）的概念，可以限制。cookie只属于某个路径下存储大小限制也不同，cookie数据不能超过4K，同时因为每次http请求都会携带cookie，所以cookie只适合保存很小的数据，如回话标识。
- webStorage虽然也有存储大小的限制，但是比cookie大得多，可以达到5M或更大

sessionStorage与localStorage

数据的有效期不同：

- sessionStorage：仅在当前的浏览器窗口关闭有效；
- localStorage：始终有效，窗口或浏览器关闭也一直保存，因此用作持久数据；cookie：只在设置的cookie过期时间之前一直有效，即使窗口和浏览器关闭。

作用域不同：

- sessionStorage：不在不同的浏览器窗口中共享，即使是同一个页面；localStorage：在所有同源窗口都是共享的；
- cookie：也是在所有同源窗口中共享的。

HTTP状态码[了解]

1xx(临时响应)：表示临时响应并需要请求者继续执行操作的状态码。

- 100(继续)请求者应当继续提出请求。服务器返回此代码表示已收到请求的第一部分，正在等待其余部分。
- 101(切换协议)请求者已要求服务器切换协议，服务器已确认并准备切换。

2xx (成功)：表示成功处理了请求的状态码。

- 200(成功)服务器已成功处理了请求。通常，这表示服务器提供了请求的网页。如果是对您的 robots.txt 文件显示此状态码，则表示 Googlebot 已成功检索到该文件。
- 201(已创建)请求成功并且服务器创建了新的资源。
- 202(已接受)服务器已接受请求，但尚未处理。
- 203(非授权信息)服务器已成功处理了请求，但返回的信息可能来自另一来源。
- 204(无内容)服务器成功处理了请求，但没有返回任何内容。
- 205(重置内容)服务器成功处理了请求，但没有返回任何内容。与 204 响应不同，此响应要求请求者重置文档视图(例如，清除表单内容以输入新内容)。
- 206(部分内容)服务器成功处理了部分 GET 请求。

3xx (重定向)：要完成请求，需要进一步操作。通常，这些状态码用来重定向。

- 300(多种选择)针对请求，服务器可执行多种操作。服务器可根据请求者 (user agent) 选择一项操作，或提供操作列表供请求者选择。
- 301(永久移动)请求的网页已永久移动到新位置。服务器返回此响应(对 GET 或 HEAD 请求的响应)时，会自动将请求者转到新位置。您应使用此代码告诉 Googlebot 某个网页或网站已永久移动到新位置。
- 302(临时移动)服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来响应以后的请求。此代码与响应 GET 和 HEAD 请求的 301 代码类似，会自动将请求者转到不同的位置，但您不应使用此代码来告诉 Googlebot 某个网页或网站已经移动，因为 Googlebot 会继续抓取原有位置并编制索引。

- 303(查看其他位置)请求者应当对不同的位置使用单独的 GET 请求来检索响应时，服务器返回此代码。对于除 HEAD 之外的所有请求，服务器会自动转到其他位置。
- 304(未修改)自从上次请求后，请求的网页未修改过。服务器返回此响应时，不会返回网页内容。如果网页自请求者上次请求后再也没有更改过，您应将服务器配置为返回此响应(称为 If-Modified-Since HTTP 标头)。服务器可以告诉 Googlebot 自从上次抓取后网页没有变更，进而节省带宽和开销。
- 305(使用代理)请求者只能使用代理访问请求的网页。如果服务器返回此响应，还表示请求者应使用代理。
- 307(临时重定向)服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来响应以后的请求。此代码与响应 GET 和 HEAD 请求的 301 代码类似，会自动将请求者转到不同的位置，但您不应使用此代码来告诉 Googlebot 某个页面或网站已经移动，因为 Googlebot 会继续抓取原有位置并编制索引。

4xx(请求错误)：这些状态码表示请求可能出错，妨碍了服务器的处理。

- 400(错误请求)服务器不理解请求的语法。
- 401(未授权)请求要求身份验证。对于登录后请求的网页，服务器可能返回此响应。
- 403(禁止)服务器拒绝请求。如果您在 Googlebot 尝试抓取您网站上的有效网页时看到此状态码(您可以在 Google 网站管理员工具诊断下的网络抓取页面上看到此信息)，可能是您的服务器或主机拒绝了 Googlebot 访问。
- 404(未找到)服务器找不到请求的网页。例如，对于服务器上不存在的网页经常会返回此代码。
- 405(方法禁用)禁用请求中指定的方法。
- 406(不接受)无法使用请求的内容特性响应请求的网页。
- 407(需要代理授权)此状态码与 401(未授权)类似，但指定请求者应当授权使用代理。如果服务器返回此响应，还表示请求者应当使用代理。
- 408(请求超时)服务器等候请求时发生超时。

5xx(服务器错误)：这些状态码表示服务器在处理请求时发生内部错误。这些错误可能是服务器本身的错误，而不是请求出错。

- 500(服务器内部错误)服务器遇到错误，无法完成请求。
- 501(尚未实施)服务器不具备完成请求的功能。例如，服务器无法识别请求方法时可能会返回此代码。
- 502(错误网关)服务器作为网关或代理，从上游服务器收到无效响应。
- 503(服务不可用)服务器目前无法使用(由于超载或停机维护)。通常，这只是暂时状态。
- 504(网关超时)服务器作为网关或代理，但是没有及时从上游服务器收到请求。
- 505(HTTP 版本不受支持)服务器不支持请求中所用的 HTTP 协议版本。

前端优化方法[熟悉]

方法分类：

- 降低请求量：合并资源，减少HTTP 请求数，minify / gzip 压缩， webPlazyLoad。
- 加快请求速度：预解析DNS，减少域名数，并行加载，CDN 分发。
- 缓存：HTTP 协议缓存请求，离线缓存 manifest，离线数据缓存localStorage。
- 渲染：JS/CSS优化，加载顺序，服务端渲染，pipeline。

具体措施：

1. 减少HTTP请求次数：CSS Sprites，js、CSS源码压缩、图片大小控制合适；网页Gzip、CDN托管、data缓存、图片服务器
2. 前端模板JS+数据，减少由于HTML标签导致的带宽浪费，前端用变量保存AJAX请求结果，每次操作本地变量，不用请求，减少请求次数
3. 用innerHTML替代DOM操作，减少DOM操作次数，优化JavaScript性能。
4. 当需要设置的样式很多时，设置className而不是直接操作Style
5. 少用全局变量、缓存DOM节点查找的结果。减少IO读取

6. 避免使用CSS Expression（css表达式）又称Dynamic properties（动态属性）
7. 图片预加载，将样式表放在顶部，将脚本放在底部，加上时间戳。
8. 防止内存泄漏：内存泄漏指任何对象在不再拥有或需要它之后仍然继续存在。垃圾回收器定期扫描对象，并计算引用了每个对象的其他的数量。如果一个对象的引用数量为0（没有其他对象引用过该对象），或对该对象的唯一引用是循环的，那么该对象的内存即可回收。

优雅降级与渐进增强[熟悉]

浏览器输入网址到页面渲染全过程 **渐进增强**：一开始就针对低版本浏览器进行构建页面，完成基本的功能，然后在针对高级浏览器进行效果，交互，追加功能达到更好的体验。 **优雅降级**：一开始就构建站点的完整功能，然后针对浏览器测试和修复，比如一开始使用css3的特性构建一个应用，然后逐步针对各大浏览器进行hack使其可以在低版本浏览器上正常浏览。

post和get区别[精通]

1. get参数通过url传递，post参数放在request body中。
2. get请求在url中传递的参数是有长度限制的，而post没有。
3. get比post更不安全，因为参数直接暴露在url中，所以不能用来传递敏感信息。
4. get请求只能进行url编码，而post支持多种编码方式
5. get请求参数会被完整保留在浏览历史记录里，而post中的参数不会被保留。
6. GET和POST本质上就是TCP链接，并无差别。但是由于HTTP的规定和浏览器/服务器的限制，导致他们在应用过程中体现出一些不同。
7. GET产生一个TCP数据包；POST产生两个TCP数据包。对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200（返回数据）；而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok（返回数据）。

http与https有什么区别？[了解]

1. https协议需要ca申请认证书，一般免费的较少
2. http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议
3. http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443
4. http的连接很简单，是无状态的，https协议是由ssl+http协议构建的可进行加密传输，身份认证的网络协议，比http安全

HTTP协议中缓存的处理流程[熟悉]

缓存分为两种：强缓存和协商缓存，根据响应的header内容来决定。

- **强缓存**是利用http头中的Expires和cache-Control两个字段来控制的，用来表示资源的缓存时间。强缓存中，普通刷新会忽略它，但不会清除它，需要强制刷新。浏览器强制刷新，请求会带上cache-Control :no-cache和pragma: no-cache
- **协商缓存**就是由服务器来确定缓存资源是否可用，所以客户端与服务器端要通过某种标识来进行通信，从而让服务器判断请求资源是否可以缓存访问。
- 普通刷新会启用弱缓存，忽略强缓存。只有在地址栏或收藏夹输入网址、通过链接引用资源等情况下，浏览器才会启用强缓存，这也是为什么有时候我们更新一张图片、一个js文件，页面内容依然是旧的，但是直接浏览器访问那个图片或文件，看到的内容却是新的。这个主要涉及到两组协商缓存相关的header字段:Etag和If-None-Match、Last-Modified和If-Modified-Since。

输入URL到后发生了什么[熟悉]

简单过程：

1. DNS域名解析；
2. 建立TCP连接；
3. 发送HTTP请求；

4. 服务器处理请求；
5. 返回响应结果；
6. 关闭TCP连接；
7. 浏览器解析HTML；
8. 浏览器布局渲染；

详细分析 构建请求的过程:

1. 应用层进行DNS解析 通过DNS将域名解析成IP地址，在解析过程中，按照浏览器缓存，系统缓存，路由器缓存，ISP(运营商)DNS缓存，根域名服务器，顶级域名服务器，主域名服务器的顺序，逐步读取缓存，直到拿到IP地址。 这里使用DNS预解析，可以根据浏览器定义的规则，提前解析之后可能会用到的域名，使解析结果缓存到系统缓存中，缩短DNS解析时间，来提高网站的访问速度。
2. 应用层生成HTTP请求报文 应用层生成针对目标服务器的HTTP报文请求，HTTP请求包括起始行，首部和主体部分。首部包括域名host，keep-alive，User-Agent，Accept-Endoding，Accept-Language，cookie等信息
3. 传输层建立TCP连接 http协议使用的TCP协议。 将http请求报文按序号分为多个报文段，并对每个报文段进行封装，使用本地的随机tcp源端口简历到目标服务器的tcp80端口(https是443端口)的连接，TCP源端口和目的端口被加入到报文中。即协议数据单元，同时还会加入序列号，确认号，检验和等参数，共计添加20个字节的头部信息。 这里构建TCP连接请求会增加大量的网络延时。常用的优化方法如下： 1)资源打包，合并请求 2)多使用缓存，减少网络传输 3)使用keep-alive建立持久连接 4)使用多个域名，增加浏览器的资源并发加载数，或者使用http2的管道化连接的多路复用技术
4. 网络层使用IP协议来选择线路。 处理来自传输层的数据段segment，将数据段segment装入数据包packet，填充包头，主要添加源和目的IP地址，然后发送数据。IP协议负责选择传送的路线，称为路由功能。
5. 数据链路层实现网络相邻结点间可靠的数据通信 为了保证数据的可靠性，把数据包packet封装成帧，并按顺序传送各帧，对每个数据块计算CRC(循环冗余检验),防止数据包丢失，出错就重传。 将数据包封装成帧，分为帧头和帧尾，帧尾是CRC校验部分，帧头主要是添加数据链路层的地址，源地址和目的地址，即网络相邻结点间的MAC地址
6. 物理层传输数据 将数据链路层的帧转换为二进制形式的比特流，从网卡发送出去，再转换成电子，光学信号在网络中传输。

Js阻塞[精通]

JS可以操作DOM来改变DOM结构，修改CSSOM样式，所以浏览器在遇到`<script>`标签时，DOM构建将暂停，直到脚本执行完成，再继续构建DOM，所以`<script>`标签一般放在最后。现在可以在`<script>`标签上增加`defer`，`async`等属性，单独解析js中操作DOM和CSSOM的地方追加到DOM和CSSOM上。[具体看红宝书]

Css阻塞[精通]

CSSOM负责存储渲染信息，所以CSSOM在浏览器合成渲染树之前必须是完备的，否则就不会进入渲染过程。所以将样式放在head中，为了更快的解析css。

defer和async属性的区别[了解]

没有`defer`和`async`属性的时候浏览器在读取到`<script>`标签的时候会立即读取脚本文件，而不会加载后续的文档元素，会阻塞后续DOM的构建 文档解析时，遇到设置了`defer`的脚本，就会在后台进行下载，但是并不会阻止文档的渲染，当页面解析&渲染完毕后。会等到所有的defer脚本加载完毕并按照顺序执行，执行完毕后会触发DOMContentLoaded(dom内容加载完毕)事件。 `async`脚本会在加载完毕后执行。`async`脚本的加载不计入DOMContentLoaded事件统计 有`defer`或者`async`属性之后，脚本文件的读取和后续文档的加载是并行的也就是异步执行，js脚本的执行会等到所有元素被解析完成之后，DOMContentLoaded事件调用前执行。 存在多个`defer`的脚本文件时，他们会按照顺序`defer`

的顺序执行，而`async`的加载则是无序的，只要DOM加载完成会立即执行，不考虑脚本之间的依赖关系。

重排(layout)和重绘(repaint)[了解]

DOM的变化影响到了已渲染完成的几何属性，比如宽高等，浏览器将重新计算元素的几何属性，其他元素的几何属性也会受到影响，浏览器需要重新构造渲染树（Render树），这个过程称之为重排(也叫回流reflow)，浏览器将受到影响的部分重新绘制在屏幕上的过程称之为重绘。`display:none`会触发reflow，而`visibility: hidden`属性则并不算是不可见属性，只会触发repaint

重绘产生的原因有：

- 添加或删除可见的DOM元素，
- 元素的尺寸位置发生改变。
- 浏览器页面初始化
- 浏览器窗口大小发生变化
- 重排一定导致重绘，重绘不一定导致重排。

减少重绘重排的方法：

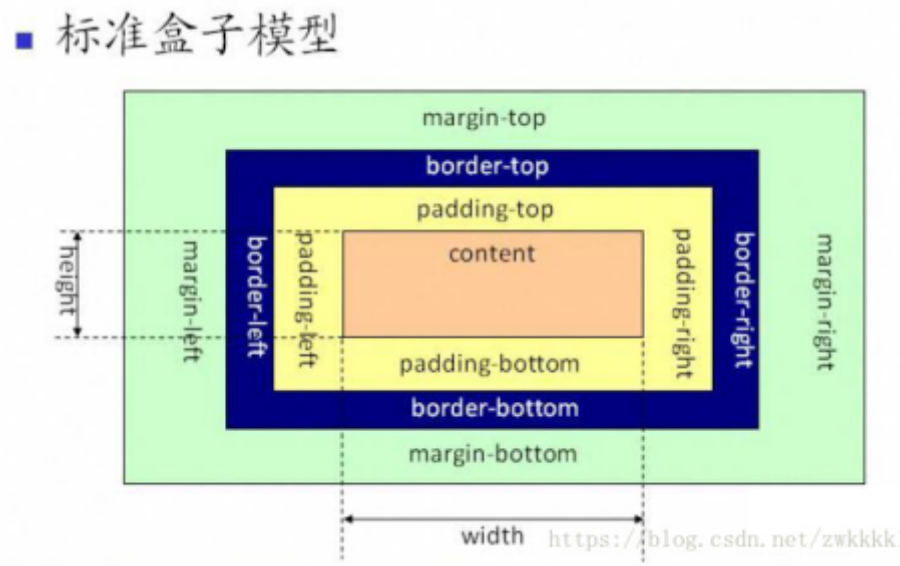
- 不在布局信息改变时做DOM查询
- 不要一条一条的修改DOM样式，使用csstext，className一次性改变属性
- 在内存中多次操作节点，完成后再添加到文档中
- 对于一个元素进行复杂的操作时，先隐藏，操作完再展示
- 需要经常获取那些引起浏览器重排的属性时，缓存到变量中
- 不要使用table布局，一个小改动会导致table重新布局
- 对于多次重排的元素，比如说动画，使用绝对定位让其脱离文档流，使其不影响其他元素，减少重绘范围

http报文构成[熟悉]

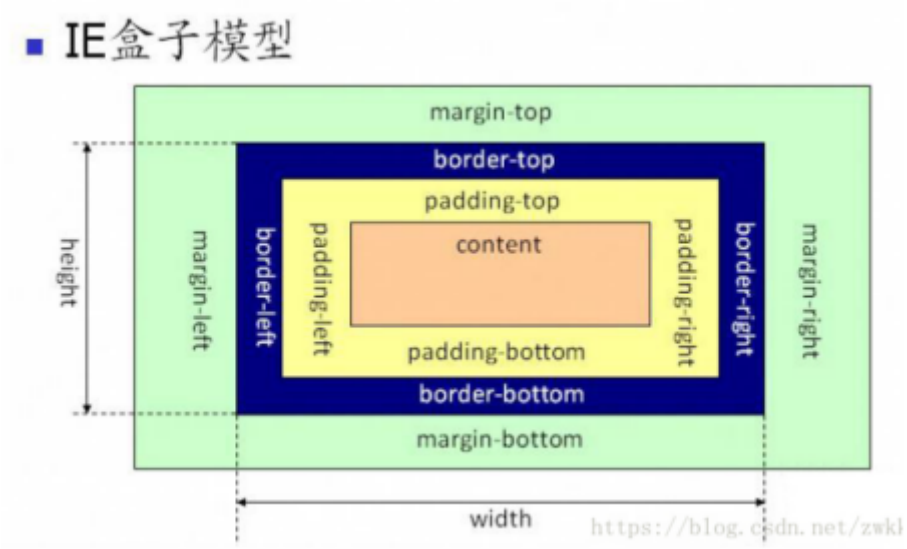
css3新增[精通]

CSS3边框如`border-radius`，`box-shadow`等；CSS3背景如`background-size`，`background-origin`等；CSS3 2D，3D转换如`transform`等；CSS3动画如`animation`等。 CSS3的新特性中，在布局方面新增了flex布局，在选择器方面新增了例如`first-of-type`,`nth-child`等选择器，在盒模型方面添加了`box-sizing`来改变盒模型，在动画方面增加了`animation`，2d变换，3d变换等，在颜色方面添加透明，rbga等，在字体方面允许嵌入字体和设置字体阴影，最后还有媒体查讯等.

css的两种盒模型[精通]



在标准的盒子模型中，width指content部分的宽度



在IE盒子模型中，width表示content+padding+border这三个部分的宽度 如果想要切换盒模型也很简单，这里需要借助css3的box-sizing属性 box-sizing: content-box 是W3C盒子模型 box-sizing: border-box 是IE盒子模型 box-sizing的默认属性是content-box

visibility=hidden, opacity=0，display:none：

- opacity=0，该元素隐藏起来了，但不会改变页面布局，并且，如果该元素已经绑定一些事件，如click事件，那么点击该区域，也能触发点击事件的方法。
- visibility=hidden，该元素隐藏起来了，但不会改变页面布局，但是不会触发该元素已经绑定的事件。
- display=none，把元素隐藏起来，并且会改变页面布局，可以理解成在页面中把该元素删除掉一样。

position属性比较[精通]

固定定位fixed： 元素的位置相对于浏览器窗口是固定位置，即使窗口是滚动的它也不会移动。Fixed定位使元素的位置与文档流无关，因此不占据空间。 Fixed定位的元素和其他元素重叠。

相对定位relative：相对于他原本的位置进行移动 如果对一个元素进行相对定位，它将出现在它所在的位置上。然后，可以通过设置垂直或水平位置，让这个元素"相对于"它的起点进行移动。 在使用相对定位时，无论是否进行移动，元素仍然占据原来的空间。因此，移动元素会导致它覆盖其它框。

绝对定位absolute：相对于他的父元素进行移动 绝对定位的元素的位置相对于最近的已定位父元素，如果元素没有已定位的父元素，那么它的位置相对于`<html>`。 absolute 定位使元素的位置与文档流无关，因此不占据空间。 absolute 定位的元素和其他元素重叠。

粘性定位sticky： 先是相对于他原本的位置进行移动，然后在超出目标区域后，固定在目标位置，即先按照relative定位方式定位，然后按照fix定位方式定位。 元素先按照普通文档流定位，然后相对于该元素在流中的flow root（BFC）和 containing block（最近的块级祖先元素）定位。而后，元素定位表现为在跨越特定阈值前为相对定位，之后为固定定位。

默认定位Static： 默认值。没有定位，元素出现在正常的流中（忽略top, bottom, left, right 或者 z-index 声明）。

继承定位inherit: 规定应该从父元素继承position 属性的值。

JS部分

JS基本数据类型[精通]

简单数据类型：Undefined、Null、Boolean、Number和String,es6xinzeng 复杂数据类型：Object 通过typeof查看数据的基本类型，其中null会返回object，理解为空对象。

```
console.log(null==undefined);    //true  因为两者都默认转换成了false
console.log(typeof undefined);    //"undefined"
console.log(typeof null);       //"object"
console.log(null===undefined);    //false   "==="表示绝对相等，null和undefined类型是不一样的，所以
```

Null和undefined的区别：

- Null表示一个无的对象，转成数值是0，一个尚未存在的对象，常用来表示函数企图返回一个不存在的对象 用法：

1. 作为函数的参数，表示该函数的参数不是对象
2. 作为对象原型链的终点

- Undefined是一个表示无的原始值，转换成数值为NaN，当声明的变量未被初始化的时候，变量的默认值为undefined 用法：

1. 变量被声明了，但没有被赋值时，就等于undefined
2. 调用函数时，应该提供的参数没有提供，该参数等于undefined
3. 对象没有赋值属性，该属性的值为undefined
4. 函数没有返回值时，默认返回undefined

一定要注意数据类型的转换

ES6新增[精通]

1.新增声明命令let和const，let表示变量，const表示常量。

- let和const都是块级作用域，以{}代码块作为作用域范围，只能在代码块里面使用。
- 不存在变量提升，只能先声明再使用，否则报错。在声明变量之前，该变量都是不可用的。
- 在同一个代码块内，不允许重复声明。
- Const声明的是一个只读常量，在声明时就需要赋值。（如果const的是一个对象，对象的值是可以被修改的，就是对象所指向的地址不能改变，而成员变量是可以修改的。）

2.模板字符串

用一对反引号作为标识，可以当做普通字符串使用，也可以用来定义多行字符串，也可在字符串中嵌入变量，js表达式或函数，变量，js表达式或函数需要写在`${}`中.

```
var str = `abc
def
gh`;
console.log(str);
let name = "小明";
function a() {
   return "ming";
}
console.log(`我的名字叫做${name}，年龄${17+2}岁，性别${'男'}，游戏ID：${a()}`);
```

3.函数相关扩展方法

函数的默认参数 ES6为参数提供了默认值，在定义函数时便初始化这个值，以便在参数没有传递进去时使用。

箭头函数 写法：函数名=(形参)=>{.....}，当函数体中只有一个表达式时，{}和return可以省略，当函数体中形参只有一个时，()可以省略。

特点：

- 箭头函数是匿名函数，不能作为构造函数，不能使用new
- 箭头函数不绑定arguments，用rest参数...代替
- 箭头函数不绑定this，会捕获其所在上下文的this值，作为自己的this值
- 任何方法都不能改变箭头函数中this的指向，箭头函数通过call(),apply()方法调用一个函数时，只传入参数，不改变this指向 箭头函数没有原型属性
- 箭头函数不能当做Generator函数,不能使用yield关键字
- 与普通函数的区别:普通函数中的this指向的始终是最后调用它的对象(比如在对象中定义的函数，这个函数内部的this,指向的是这个对象),如果没有直接调用对象，会指向undefined或者window，一般都会指向window，在严格模式下才会指向undefined。

普通函数中的this的四种调用模式：

1. 函数式调用模式 如果一个函数不是一个对象的属性时，就是被当做一个函数来进行调用的。此时this指向了window
2. 方法式调用 当一个函数被保存为对象的一个属性时，我们称之为一个方法。当一个方法被调用时，this被绑定到当前调用对象
3. 构造函数式调用 如果函数是通过new关键字进行调用的，此时this被绑定到创建出来的新对象上。
4. 上下文调用（借用方法模式） 上下文调用模式也叫方法借用模式，分为apply与call，使用方法：函数.call() 或者函数.apply()
5. 几种特殊的this指向 定时器中的this指向了window，因为定时器的function最终是由window来调用的。 事件中的this指向的是当前的元素，在事件触发的时候，浏览器让当前元素调用了function

4. 对象相关的扩展方法

1. 对象的解构取值与解构赋值
2. Object.keys()方法，获取对象的所有属性名或方法名（不包括原型的内容），返回一个数组。
3. Object.assign ()，assign方法将多个原对象的属性和方法都合并到了目标对象上面。可以接收多个参数，第一个参数是目标对象，后面的都是源对象。 **第一级属性深拷贝，以后级别属性浅拷贝**

5.for of循环

是遍历所有数据结构的统一的方法。for...of循环可以使用的范围包括数组、Set 和 Map 结构、某些类似数组的对象（比如arguments对象、DOM NodeList 对象）、Generator 对象，以及字符串。**不能遍历对象，因为对象没有内置的迭代器**

6.import和export

- ES6标准中，JavaScript原生支持模块(module)了。这种将JS代码分割成不同功能的小块进行模块化，将不同功能的代码分别写在不同文件中，各模块只需导出公共接口部分，然后通过模块的导入的方式可以在其他地方使用。
- export用于对外输出本模块（一个文件可以理解为一个模块）变量的接口。
- import用于在一个模块中加载另一个含有export接口的模块。
- import和export命令只能在模块的顶部，不能在代码块之中。

7.解构赋值

- ES6 允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构（Destructuring）。
- 数组的解构赋值，数组中的值会自动被解析到对应接收该值的变量中，数组的解构赋值要一一对应如果有对应不上的就是undefined

```
var [name, pwd, sex]=["小周", "123456", "男"];
console.log(name) //小周
console.log(pwd)//123456
console.log(sex)//男
```

- 对象的解构赋值，对象的解构与数组有一个重要的不同。数组的元素是按次序排列的，变量的取值由它的位置决定；而对象的属性没有次序，变量必须与属性同名，才能取到正确的值。

```
var obj={name:"小周", pwd:"123456", sex:"男"}
var {name, pwd, sex}=obj;
console.log(name) //小周
console.log(pwd)//123456
console.log(sex)//男

//如果想要变量名和属性名不同，要写成这样
let { foo: foz, bar: baz } = { foo: "aaa", bar: "bbb" };
console.log(foz) // "aaa"
console.log(foo) // error: foo is not defined
```

8.set数据结构

Set数据结构，类似数组，所有数据都是唯一的，没有重复的值，它本身是一个构造函数 属性和方法：

- size 数据的长度
- add() 添加某个值，返回 Set 结构本身。
- delete() 删除某个值，返回一个布尔值，表示删除是否成功。
- has() 查找某条数据，返回一个布尔值。
- clear() 清除所有成员，没有返回值。
- 应用：数组去重
- 注意：new Set(arr)的返回结果是object，可以使用Array.from(new Set(arr))或[...new Set(arr)]来返回数组。

9.map数据结构

Map结构是键值对集合(Hash结构) 在JS中的默认对象的表示方式为{}，即一组键值对，但是键必须是字符串。 为了使用Number或者其他数据类型作为键，ES6规范引入了新的数据类型Map。 Map是一组键值对的结构，具有极快的查找速度。初始化Map需要一个二维数组，或者直接初始化一个空Map。

map和object的区别

- Obj的键只能用字符串、数字或者Symbol等简单数据类型当作键
- Map的键可以是任意的数据类型
- Map不存在同名碰撞问题，每个对象都是单独的一块内存地址
- Map实现了迭代器，可用for...of遍历，而Object不行
- Map可以直接拿到长度，而Object不行。
- 填入Map的元Map可以使用省略号语法展开，而Object不行。填入元素会保持原有的顺序，而Object无法做到。

10.展开运算符

promise对象[精通]

romise是异步编程的一种解决方案，将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。 promise构造函数是同步执行的，then方法是异步执行的。 它有三种状态，分别是pending-进行中、resolved-已完成、rejected-已失败。 Promise 构造函数包含一个参数和一个带有 resolve（解析）和 reject（拒绝）两个参数的回调。在回调中执行一些操作（例如异步），如果一切都正常，则调用 resolve，否则调用 reject。对于已经实例化过promise对象可以调用 promise.then() 方法，传递 resolve 和 reject 方法作为回调。then()方法接收两个参数：onResolve和onReject，分别代表当前 promise 对象在成功或失败时。

Promise的实现原理

首先promise有三种状态

Pending 对象实例创建时候的初始状态

Fulfilled 可以理解为成功的状态

Rejected可以理解为失败的状态

构造一个Promise实例需要给Promise构造函数传入一个函数。传入的函数需要有两个形参，两个形参都是function类型的参数。分别是resolve和reject。

Promise上还有then方法，then 方法就是用来指定Promise 对象的状态改变时确定执行的操作，resolve 时执行第一个函数（onFulfilled），reject时执行第二个函数（onRejected）

当状态变为resolve时便不能再变为reject，反之同理。

Promise的缺点:

1. 无法取消，一旦新建就会立即执行，无法中途取消，
2. 不设置回调函数的情况下，promise内部抛出的错误不会反映到外部
3. 当处于pending状态时，无法得知目前进展到哪一个阶段，刚开始还是快结束

Promise.all() 将promise对象数组按照异步顺序全部完成后在then的第一个函数中传入完成的结果，这个list参数是arr这个promise对象数组中所有异步的then中返回的img按顺序组合成的一个数组。

也可以理解为将arr中的每个promise对象完成后的then中的img按顺序加入到一个数组中，在全部完成后返回这个数组list

Promise.all(requestPromises).then(...).catch(...) 会在所有requestPromises都resolve时才会进then方法，并且把所有结果以一个数组返回。只要有一个失败，就会进catch。如果在单个请求中定义了catch方法，那么就不会进Promise.all的catch方法。因此，可以在单个的catch中将失败的promise放入一个list，待一轮请求完成后，再去请求失败的请求。

```
Promise.all(arr).then(function(list){
    // console.log(list);//所有图片
    list.forEach(item=>{
        console.log(item.width,item.src);
    })
})
```

Promise.race()

的作用是将arr这个promise对象数组中最先执行完成的一个promise中then中返回的参数传入img中，也就是promise.race().then()的第一个函数的参数中。

```
    Promise.race(arr).then(function(img){
    console.log(img.src);
})
```

ES6与 CommonJS 模块化的区别[精通]

什么是js模块化?

js一开始并没有模块化的概念，直到ajax被提出，前端能够像后端请求数据，前端逻辑越来越复杂，就出现了许多问题：全局变量，函数名冲突，依赖关系不好处理。当时使用子执行函数来解决这些问题，比如经典的jquery就使用了匿名自执行函数封装代码，将他们全都挂载到全局变量jquery下边。

CommonJs CommonJs通过nodeJs发扬光大，每个js文件就是一个模块，每个模块有单独的作用域。模块以module.exports为出口，输出一个对象。使用require方法读取文件，并返回其内部的module.exports对象。 CommonJs的问题在于，他的加载是同步的，这在服务端很正常，但是在充满了异步的浏览器里，就不适用了。为了适应浏览器，社区内部发生了分歧。

AMD（异步模块定义） AMD规范通过define方法去定义模块，通过require方法去加载模块。RequireJS实现了这种规范。 AMD只有一个接口：define(id?,dependencies?,factory); 它要在声明模块的时候制定所有的依赖(dep)，并且还要当做形参传到factory中。要是没什么依赖，就定义简单的模块（或者叫独立的模块）

CMD（通用模块定义） CMD是SeaJS 在推广过程中对模块定义的规范化产出。

AMD和CMD的区别：

1. 对于依赖的模块，AMD 是提前执行，CMD 是延迟执行。不过 RequireJS 从 2.0 开始，也改成可以延迟执行（根据写法不同，处理方式不同）。CMD 推崇 as lazy as possible（尽可能的懒加载，也称为延迟加载，即在需要的时候才加载）。
2. CMD 推崇依赖就近（依赖项可以使用的时候定义），AMD 推崇依赖前置(依赖项必须在一开始就写好)。
3. AMD 的 API 默认是一个当多个用，CMD 的 API 严格区分，推崇职责单一。比如 AMD 里，require 分全局 require 和局部 require，都叫 require。CMD 里，没有全局 require，而是根据模块系统的完备性，提供 seajs.use 来实现模块系统的加载启动。CMD 里，每个 API 都简单纯粹。

es6中的模块化

在es6没有出来之前，社区制定了一些模块加载方案，最主要的有 CommonJS 和 AMD 两种，前者用于服务器，后者用于浏览器，ES6 在语言标准的层面上，实现了模块功能，而且实现得相当简单，完全可以取代 CommonJS 和 AMD 规范，成为浏览器和服务器通用的模块解决方案。

es6中的模块化有一个比较大的特点，就是实现尽量的静态化。

ES6 模块不是对象，而是通过export命令显式指定输出的代码，再通过import命令输入。

模块功能主要由两个命令构成：export和import，export命令用于规定模块的对外接口，import命令用于输入其他模块提供的功能。

一般来说，一个模块就是一个独立的文件，该文件内部的所有变量，外部无法获取，如果你希望外部能够读取模块内部的某个变量，就必须使用export关键字输出该变量。

Javascript 模块化主要有三种方案：

1. CommonJS

```
// module add.js
module.exports = function add (a, b) { return a + b; }

// main.js
var {add} = require('./math');
// i hate sync
console.log('1 + 2 = ' + add(1,2);
```

2. AMD / CMD

```
// module add.js
define(function () {
  return {
    add: function (a, b) add { return a + b; }
  };
});

// main.js
require(['add'], function (add) {
  //i hate callback
  console.log('1 + 2 = ' + add(1,2);
});
```

3. ES6

```
// module add.js
export function add (a, b) { return a + b; }

// main.js
// i hate static
import {add} from 'add.js';
```

ES6模块与CommonJS的区别:

- CommonJS 模块输出的是一个值的拷贝，ES6 模块输出的是值的引用。
- CommonJS 模块是运行时加载，ES6 模块是编译时输出接口。
- CommonJs模块输出的是值的拷贝，也就是说，一旦输出一个值，模块内部的变化不会影响到这个值。

回调函数[精通]

回调函数： 一个函数被作为参数传递给另一个函数（在这里我们把另一个函数叫做“otherFunction”），回调函数在otherFunction中被调用。

softeem · 杨标

```
//注意到click方法中是一个函数而不是一个变量
//它就是回调函数
$("#btn_1").click(function() {
    alert("Btn 1 Clicked");
});
```

softeem · 杨标

```
//或者
function click() { // 它就是回调函数
    alert("Btn 1 Clicked");
}
$("#btn_1").click(click);
```

回调函数并不会马上被执行。它会在包含它的函数内的某个特定时间点被“回调”（就像它的名字一样）实现回调函数的基本原理

使用命名函数或者匿名函数作为回调

像之前的例子一样，第一种方法就是匿名函数作为回调（使用了参数位置定义的匿名函数作为回调函数）。第二种方式就是命名函数作为回调（定义一个命名函数并将函数名作为变量传递给函数）

什么是回调地狱

说起回调地狱，首先想到的是异步，在js中我们经常会大量使用异步回调，例如使用ajax请求

我们来看下面这段代码：

softeem · 杨标

```
function a(functionb){
    c(function d(){
    })
})
```

我们发现上面代码大量使用了回调函数（将一个函数作为参数传递给另个函数）并且有许多 })结尾的符号，使得代码看起来很混乱。

如何解决回调地狱呢?

第一种使用ES6中的Promise,中文翻译过来承诺,意思是在未来某一个时间点承诺返回数据给你。

Promise有三种状态：pending/reslove/reject 。pending就是未决，resolve可以理解为成功，reject可以理解为拒绝。

同时Promise常用的三种方法 then 表示异步成功执行后的数据状态变为reslove catch 表示异步失败后执行的数据状态变为reject all表示把多个没有关系的Promise封装成一个Promise对象使用then返回一个数组数据。

但是如果过多的使用then也会照成新的执行流程问题。所以我们的另一位主角登场了，那就是ES6中的Generator（生成器）

Generator（生成器）是一种有效利用内存的机制，一边循环一边计算生成数值的机制。通过配合Promise可以更加优雅的写异步代码

```
// 首先声明一个生成器函数
function *main() {
    console.log('starting *main()');
    yiled; // 打住，不许往下走了
    console.log('continue yield 1');
    yield; // 打住，又不许往下走了
    console.log('continue yield 2');
}
// 构造处一个迭代器it
let it = main();

// 调用next()启动*main生成器，表示从当前位置开始运行，停在下一个yield处
it.next(); // 输出 starting *main()

// 继续往下走
it.next(); // 输出 continue yield 1

// 再继续往下走
it.next(); // 输出 continue yield 2

```

Iterator(迭代器)：当我们实例化一个生成器函数之后，这个实例就是一个迭代器。可以通过next()方法去启动生成器以及控制生成器的是否往下执行。

yield/next：这是控制代码执行顺序的一对好基友。

通过yield语句可以在生成器函数内部暂停代码的执行使其挂起，此时生成器函数仍然是运行并且是活跃的，其内部资源都会保留下来，只不过是处在暂停状态。

在迭代器上调用next()方法可以使代码从暂停的位置开始继续往下执行。

当然生成器不是最完美的，它的语法让人难以理解，所以ES7推出了async/await (异步等待),多么贴切。yield + Promise的写法需要我们对拿到的promise的决议进行人工处理(区分成功或失败)，在ES7中提供了async/await帮我们省掉了这个步骤

```
function getCallSettings() {
    return utils.ajax({
        url: '/dialer/dialerSetting',
        method: "GET",
    });
}
async function dealData() {
    try {
        let settingInfo = await getCallSettings(); // await会暂停在这，直到promise决议(请求返回)
        // do something...
    }
    catch(err) {
        console.log(err);
    }
}
dealData();
```

明确概念

- async函数就是generator函数的语法糖。
- async函数，就是将generator函数的`*`换成async，将yield替换成await。

async函数对generator的改进

- 内置执行器，不需要使用next()手动执行。
- await命令后面可以是Promise对象或原始类型的值，yield命令后面只能是Thunk函数或Promise对象。
- 返回值是Promise。返回非Promise时，async函数会把它包装成Promise返回。(Promise.resolve(value))

作用 异步编程的终极解决方案。

通俗理解（个人理解） async/await，就是异步编程回调函数写法的替代方法。（使代码以同步方式的写法完成异步操作）

原理(执行顺序) 函数执行时，一旦遇到await就会返回。等到触发的异步操作完成（并且调用栈清空），再接着执行函数体内后面的语句。【个人理解】

- await语句后面的代码，相当于回调函数。（即：await的下一行开始，都视作回调函数的内容）
- 回调函数会被压入microtask队列，当主线程调用栈被清空时，去microtask队列里取出各个回调函数，逐个执行。
- await只是让当前async函数内部、后面的代码等待，并不是所有代码都卡在这里。遇到await就先返回，执行async函数之后的代码。

闭包[熟悉]

1. 什么是闭包： 闭包是指有权访问另外一个函数作用域中的变量的函数。 闭包就是函数的局部变量集合，只是这些局部变量在函数返回后会继续存在。闭包就是就是函数的“堆栈”在函数返回后并不释放，我们也可以理解为这些函数堆栈并不在栈上分配而是在堆上分配。当在一个函数内定义另外一个函数就会产生闭包。
2. 为什么要用：

- 匿名自执行函数：我们知道所有的变量，如果不加上var关键字，则默认的会添加到全局对象的属性上去，这样的临时变量加入全局对象有很多坏处，比如：别的函数可能误用这些变量；造成全局对象过于庞大，影响访问速度(因为变量的取值是需要从原型链上遍历的)。除了每次使用变量都是用var关键字外，我们在实际情况下经常遇到这样一种情况，即有的函数只需要执行一次，其内部变量无需维护，可以用闭包。
- 结果缓存：我们开发中会碰到很多情况，设想我们有一个处理过程很耗时的函数对象，每次调用都会花费很长时间，那么我们就需要将计算出来的值存储起来，当调用这个函数的时候，首先在缓存中查找，如果找不到，则进行计算，然后更新缓存并返回值，如果找到了，直接返回查找到的值即可。闭包正是可以做到这一点，因为它不会释放外部的引用，从而函数内部的值可以得以保留。
- 封装：实现类和继承等。

闭包的应用： 实现每隔一秒输出1,2,3,4.... 实现方式

- ES6 let 块级作用域
- ES5 闭包 匿名函数and函数自动执行,(function(i){})(i) 其中第一个()返回一个匿名函数，第二()起到立即执行的作用

```
for(let i = 1; i <= 10; i++){
    setTimeout(function () {
        console.log(i);
    }, 1000 * i);
}
```

```
for(var i = 1; i <= 10; i++){
    (function (i) {
        setTimeout(function () {
            console.log(i);
        }, 1000 * i)
    })(i);
}
```

实现add(1)(2)(3)输出6

闭包的优点

- 可以重复使用变量，并且不会造成变量污染
- 全局变量可以重复使用，但是容易造成变量污染。局部变量仅在局部作用域内有效，不可以重复使用，不会造成变量污染。闭包结合了全局变量和局部变量的优点。
- 可以用来定义私有属性和私有方法。 闭包的缺点
- 比普通函数更占用内存，会导致网页性能变差，在IE下容易造成内存泄露（分配了一些'顽固的'内存，浏览器无法进行回收，就会导致后面所用内存不足）。

Js内存泄漏：

内存泄漏是指由于疏忽或者错误造成程序未能释放已经不再使用的内存，内存泄漏并非指内存在物理内存上的消失，而是应用程序在分配某段内存之后，由于设计错误，导致在释放该段内存之前就失去了对该段内存的控制，造成了内存的浪费。

1. 意外的全局变量，js对未声明变量会在全局最高对象上创建它的引用
2. console.log
3. 闭包
4. DOM泄漏，我们一般将常用的DOM。我们会采用变量引用的方式会将其缓存在当前环境
5. 被遗忘的timers

Js垃圾回收机制[了解]

JS的内存生命周期：

1. 分配你所需要的内存
2. 使用分配到的内存（读、写）
3. 不需要时将其释放、归还

js垃圾回收机制：

1. 自动垃圾回收机制就是找出那些不再继续使用的值，然后释放其占用的内存空间。垃圾回收器每隔固定的时间段就执行一次释放操作。
2. js最常用的是通过标记清除的算法来找到哪些对象是不再继续使用的，上面例子中的a = null 其实就是做了一个释放引用的操作，让a原本对应的值失去引用，脱离执行环境，这个值会在下一次垃圾收集器执行操作时被找到并释放。因此，在适当的时候解除引用，是为页面获的更好性能的一个重要方式。

标记清除

这是javascript中最常用的垃圾回收方式。当变量进入执行环境是，就标记这个变量为"进入环境"。当变量离开环境时，则将其标记为"离开环境"。从逻辑上讲，永远不能释放进入环境的变量所占用的内存，因为只要执行流进入相应的环境，就可能会用到他们。 垃圾收集器在运行的时候会给存储在内存中的所有变量都加上标记。然后，它会去掉环境中的变量及被其引用的变量的标记。而在此之后再被加上标记的变量将被视为准备删除的变量，原因是环境中的变量已经无法访问到这些变量了。最后。垃圾收集器完成内存清除工作，销毁那些带标记的值，并回收他们所占用的内存空间。

引用计数

另一种不太常见的垃圾回收策略是引用计数。引用计数的含义是跟踪记录每个值被引用的次数。当声明了一个变量并将一个引用类型赋值给该变量时，则这个值的引用次数就是1。相反，如果包含对这个值引用的变量又取得了另外一个值，则这个值的引用次数就减1。当这个引用次数变成0时，则说明没有办法再访问这个值了，因而就可以将其所占的内存空间给收回来。这样，垃圾收集器下次再运行时，它就会释放那些引用次数为0的值所占的内存。

Ajax，fetch，Axios的区别[精通]

```
<script>
    //步骤一:创建异步对象
    var xhr = new XMLHttpRequest();
    //步骤二:设置请求的基本参数
    xhr.open('get','test.php');
    //步骤三:发送请求
    xhr.send();
    //步骤四:注册onreadystatechange监听事件，只要状态改变就会调用
    xhr.onreadystatechange = function () {
        if (ajax.readyState == 4 && ajax.status == 200) {
         //步骤五 如果能够进到这个判断 说明数据完美到手
            console.log(ajax.responseText);//操作返回内容
        }
    }
</script>
```

ajax使用步骤

1. 创建XmlHttpRequest对象
2. 调用open方法设置基本请求信息
3. 设置发送的数据，发送请求
4. 注册监听的回调函数
5. 拿到返回值，对页面进行更新 可这里的readyState的状态码代表什么呢？这里记录如下： 0：未初始化，但是已经创建了XHR实例 1：调用了open()函数 2：已经调用了send()函数，但还未收到服务器回应 3：正在接受服务器返回的数据 4：完成响应

```
ajax({
    type:"post",
    url:"test.php",
    data:"name=lan&pwd=123456",
    success:function(data){
        console.log(data);
    }
});
```

```
fetch('http://www.mozotech.cn/bangbang/index/user/login', {
  method: 'post',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded'
  },
  body: new URLSearchParams([["username", "Lan"],["password", "123456"]])
})
.then(res => {
    console.log(res);
    return res.text();
})
.then(data => {
    console.log(data);
})
```

```
axios({
    method: 'post',
    url: '/abc/login',
    data: {
        userName: 'Lan',
        password: '123'
    }
})
.then(function (response) {
    console.log(response);
})
.catch(function (error) {
    console.log(error);
});
```

传统Ajax值得是XMLHttpRequest(XHR)，最早出现的发送后端请求技术，隶属于原生js中，核心是使用XMLHttoRequest对象，多个请求之间如果有先后关系的话，就会出现回调地狱。

优点：局部更新，原生支持，不需要插件

缺点：可能破坏浏览器的后退功能，产生回调地狱

Jquery ajax是对原生XHR的封装，此外还添加了对JSONP的支持。

优点：很多

缺点：还是没有解决回调地狱的问题。

Fetch是ajax的替代品，是ES6中出现的，使用了ES6中的promise对象，代码结构比ajax简单，参数类似Jquery ajax，但是fetch不是ajax的进一步封装，而是原生js，没有使用XHR对象

优点：语法简洁，易于理解。基于promise实现，支持async/await。更加底层，丰富的API，脱离了XHR，是ES规范里的新实现方式

缺点：只对网络请求报错，400,500都会当做成功的请求。默认不带cookie，需添加配置项。不支持超时控制。不能监测请求的进度。

Axios是一个基于promise用于浏览器和node.js的http客户端，本质上是对XHR的封装，不过是基于promise实现，符合ES最新规范。

优点：从浏览器中创建XHR，支持promise的API，浏览器端支持防止CSRF(跨站请求伪造)，提供了并发请求接口，从node.js创建http请求，拦截请求和相应，转换请求和响应数据，自动转换json数据。

JS跨域[精通]

跨域产生的原因：跨域是由于浏览器的同源策略引起的，是指页面请求的接口地址，必须与页面url地址处于同域（即域名，端口，协议相同）上，是为了防止某域名下的接口被其他域名下的网页非法调用，是浏览器对js施加的安全限制。

后端解决跨域的方式通常有两个： **JSONP**： 利用script标签可跨域的特点，在跨域脚本中可以直接回调当前脚本的函数，即通过动态创建script，再请求一个带参的网址实现跨域通信。 在jQuery中如何通过JSONP来跨域获取数据?

第一种方法是在ajax函数中设置dataType为'jsonp'：

第二种方法是利用getJSON来实现，只要在地址中加上`callback=?`参数即可：

```
$.getJSON('http://www.a.com/user?id=123&callback=?', function(data){
        //处理data数据
});
```

当你需要的跨域时候，创建一个script标签，加到head中，设置这个script的src值为其它域的url，最好加上一个随机参数，以防WEB缓存机制。如生成这样的一个标签：

```
<script type="text/javascript" src="http://abc.com/js.php?t=1234"></script>
```


当然这应该通过document.createElement来完成。

```
function( url ) jsonHandle{
    var script = document.createElement("script");
    script.setAttribute("src",url);
    document.getElementsByTagName("body")[0].appendChild(script);
}
//JS加入之后就可以处理数据了
```

CORS 服务器设置http响应头中`Access-Control-Allow-Origin`值。

```
'Access-Control-Allow-Origin': 'http://127.0.0.1:8080',//设置允许跨域的域名,*为所有
'Access-Control-Allow-Methods': 'OPTIONS,HEAD,DELETE,GET,PUT,POST',//设置允许跨域的方法
'Access-Control-Allow-Headers': 'x-requested-with, accept, origin, content-type',//指示哪些 HTTP 头的名称能在响应中列出。
'Access-Control-Max-Age':10000,//指示预请求的结果能被缓存多久
'Access-Control-Allow-Credentials':true//通过使用该响应头就可以让cookies包含在CORS请求中
```

关于cookie如何在跨域请求携带的问题：

```
$.ajax({
        url : 'http://remote.domain.com/corsrequest',
        data : data,
        dataType: 'json',
        type : 'POST',
        xhrFields: {
            withCredentials: true
        },
        crossDomain: true,
```

通过设置 withCredentials: true ，发送Ajax时，Request header中便会带上 Cookie 信息。 服务器端通过在响应 header 中设置`Access-Control-Allow-Credentials = true` 来允许携带cookie。`Access-Control-Allow-Origin:""`和`Access-Control-Allow-Credentials", "true"`同时设置的时候`Access-Control-Allow-Origin`不能设置为"*"，只能设置为具体的域名。

以上两个方法严重依赖后端的协助。

如何用前端的方式实现跨域?

1. 正向代理（forward）意思是一个位于客户端和原始服务器 (origin server) 之间的服务器，为了从原始服务器取得内容，客户端向代理发送一个请求并指定目标 (原始服务器)，然后代理向原始服务器转交请求并将获得的内容返回给客户端。 正向代理是为客户端服务的，客户端可以根据正向代理访问到它本身无法访问到的服务器资源。 正向代理对客户端是透明的，对服务端是非透明的，即服务端并不知道自己收到的是来自代理的访问还是来自真实客户端的访问。
2. 反向代理（Reverse Proxy）方式是指以代理服务器来接受 internet 上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给 internet 上请求连接的客户端，此时代理服务器对外就表现为一个反向代理服务器。 反向代理是为服务端服务的，反向代理可以帮助服务器接收来自客户端的请求，帮助服务器做请求转发，负载均衡等。 反向代理对服务端是透明的，对客户端是非透明的，即客户端并不知道自己访问的是代理服务器，而服务器知道反向代理在为他服务。

深拷贝与浅拷贝：

深复制和浅复制最根本的区别在于是否是真正获取了一个对象的复制实体，而不是引用。

深拷贝和浅拷贝是只针对Object和Array这样的引用数据类型的。

- 浅复制 —-只是拷贝了基本类型的数据，而引用类型数据，复制后也是会发生引用，我们把这种拷贝叫做“（浅复制）浅拷贝”，换句话说，浅复制仅仅是指向被复制的内存地址，如果原地址中对象被改变了，那么浅复制出来的对象也会相应改变。
- 深复制 —-在计算机中开辟了一块新的内存地址用于存放复制的对象。如果源对象发生了改变，复制的对象并不会发生变化。 通俗一点理解就是浅拷贝出来的数据并不独立，如果被复制的对象改变了，那么浅拷贝的对象也会改变，深拷贝之后就会完全独立，与拷贝对象断绝关系。 深拷贝的实现方式：

1. JSON.parse(JSON.stringify()) 原理： 用JSON.stringify将对象转成JSON字符串，再用JSON.parse()把字符串解析成对象，一去一来，新的对象产生了，而且对象会开辟新的栈，实现深拷贝。这种方法虽然可以实现数组或对象深拷贝，但不能处理函数。
2. 手写递归方法 递归方法实现深度克隆原理：遍历对象、数组直到里边都是基本数据类型，然后再去复制，就是深度拷贝。

Js中的同步与异步[精通]

Js中的事件委托[精通]

事件委托就是利用事件冒泡，只指定一个事件处理程序，就可以管理某一类型的所有事件。

事件委托的原理：事件委托是利用事件的冒泡来实现的，冒泡就是事件从最深的节点开始，然后逐步向上传播事件。举个例子，页面上有这么一个结点树div>ul>li>a，比如给最里面的a加一个click点击事件，那么这个事件就会一层一层的往外执行，执行顺序a>li>ul>div，有这样一个机制，那么我们给最外面的div添加点击事件，那么里面的ul，li，a触发点击事件的时候，都会冒泡到最外成的div上，所以都会触发，这就是事件委托，委托他们父级代为执行事件。

浏览器事件机制

事件传播的三个阶段：捕获，目标对象，冒泡。

1. 其中捕获（Capture）是 事件对象(event object) 从 window 派发到 目标对象父级的过程。
2. 目标（Target）阶段是 事件对象派发到目标元素时的阶段，如果事件类型指示其不冒泡，那事件传播将在此阶段终止。
3. 冒泡（Bubbling）阶段 和捕获相反，是以目标对象父级到 window 的过程。 在任一阶段调用 stopPropagation 都将终止本次事件的传播。

绑定在被点击元素的事件是按照代码的顺序发生的，其他非绑定的元素则是通过冒泡或者捕获的触发。按照W3C的标准，先发生捕获事件，后发生冒泡事件。所以事件的整体顺序是：非目标元素捕获 -> 目标元素代码顺序 -> 非目标元素冒泡。

Js执行栈[精通]

Js执行上下文： 执行上下文就是JavaScript 在被解析和运行时环境的抽象概念，JavaScript 运行任何代码都是在执行上下文环境中运行的，执行上下文包括三个周期：创建——运行——销毁，重点说一下创建环节。 创建环节（函数被调用，但未未被执行）会执行三件事情

1. 创建变量对象，首先初始化函数的arguments对象，提升函数声明和变量声明，从近到远查找函数运行所需要的变量。
2. 创建作用域链，作用域就是一个独立的地盘，让变量不会相互干扰，当前作用域没有定义的变量，这成为 自由变量。自由变量会向上一直寻找，要到创建这个函数的那个作用域中取值——是“创建”，而不是“调用”，如果最终没有就为undefined。这种层层之间的调用关系就构成了作用域链。

3. 确定this指向，this、apply、call的指向 **Js执行栈** JavaScript 引擎创建了执行栈来管理执行上下文，可以把执行栈认为成一个储存函数调用的栈结构，遵循先进后出的原则。 Js中有三种执行上下文

- 全局执行上下文，默认的，在浏览器中是window对象，并且this在非严格模式下指向它。
- 函数执行上下文，JS的函数每当被调用时会创建一个上下文。
- Eval执行上下文，eval函数会产生自己的上下文，这里不讨论。 Js执行栈工作过程 1 JavaScript引擎是单线程执行，所有代码都是排队执行。

1. 一开始执行的是全局代码，首先创建全局的执行上下文，然后将该执行上下文压入执行栈中。
2. 每当执行一个函数，就会创建该函数的执行上下文，然后将其压入执行栈的顶部，函数执行完成后，执行上下文从底部退出，等待垃圾回收。
3. 浏览器js总是访问执行栈顶层的执行上下文。
4. 全局上下文只有唯一的一个，它在浏览器关闭时出栈

Js数组去重方法：

1. Array.filter() + indexOf

```
function distinct(a,b) {
    let arr = a.concat(b);
    return arr.filter((item,index) => {
        return arr.indexOf(item) === index;
    })
}
```

2. 使用 for...of + includes()

```
function distinct(a,b) {
    let arr = a.concat(b)
        let result = []
        for (let i of arr) {
            !result.includes(i) && result.push(i)
        }
        return result
}
```

3. 双重 for 循环
4. Array.sort()，然后比较相邻元素是否相等，从而排除重复项。
5. for...of + Object 利用对象的属性不能相同的特点进行去重

```
for (let i of arr) {
    if (!obj[i]) {
        result.push(i)
        obj[i] = 1
    }
}

return result
```

6. ES6的new Set()

```
function distinct(a, b) {
    return Array.from(new Set([...a, ...b]))
}
```

数组去重要么使用for...of + Object方式，要么使用ES6的 new Set()方式。从执行结果看for...of + Object的效率应该是最高的（只在当前量级的计算结果来看）。

Js防抖和节流[精通]

> 关于本章是一个扩展点，同学们一定要在我的博客里面去自己看一下
>
> 防抖与节流

Js防抖的基本思路：当持续触发事件时，一定时间段内没有再触发事件，事件处理函数才会执行一次，如果设定的时间到来之前，又一次触发了事件，就重新开始延时。

```
function debounce(fn,delaytime){
  let timer = null
  return function(){
      if(timer){
        //进入这里说明当前存在一个执行过程，并且同时又执行了一个相同事件，故取消当前的执行过程
        clearTimeout(timer)
      }
      timer = setTimeout(fn,delaytime)
  }
}
function show_scrollPosition(){
  var scrollPosition = document.body.scrollTop || document.documentElement.scrollTop;
  console.log("当前滚动条位置为：",scrollPosition);
}

window.onscroll = debounce(show_scrollPosition,1000)
```

JS中节流的基本思路是：规定一个期限时间，在该时间内，触发事件的回调函数只能执行一次，如果期限时间内回调函数被多次触发，则只有一次能生效。

```
function throttle(fn, delay) {
  let last_time
  let timer = null
  return function () {
    let cur_time = new Date().getTime()
    if (last_time && cur_time < last_time + delay) {
      //若为真，则表示上次执行过，且在期限值范围内
      clearTimeout(timer)
      timer = setTimeout(() => {
        fn();
        last_time = cur_time
      }, delay)
    } else {
      last_time = cur_time;
      fn();
    }
  }
}
function show_scrollPosition() {
  var scrollPosition = document.body.scrollTop || document.documentElement.scrollTop;
  console.log("当前滚动条位置为：", scrollPosition);
}
window.onscroll = throttle(show_scrollPosition, 1000)
```