# HTML、HTTP、HTTPS、浏览器相关

## H5新特性

* 新增语义化标签（标签有自己的含义，页面内容结构化、便于阅读理解维护）：<footer>、<article>、<nav>、<section>、<header>等。
* 用于绘画的<canvas>标签。
* 用于媒体回放的<video>和<audio>标签。
* 存储：提供了sessionStorage和localStorage。
* 新增input类型：color、url、email、date等。
* 新增表单控件：placeholder（文本默认提示文字）、autofocus（自动获取焦点）、autocomlpete（联想关键词）等。

## 元素类型

### 行内元素display:inline;。

#### 特点

* 排在一行内，不会自动换行。
* 宽高无效。
* padding对于水平方向正常有效，垂直方向只有效果，对其他元素无任何影响。
* margin对于水平方向有效，垂直方向无效。

#### 常见的行内元素

<a>、<span>、<i>、<strong>、<img>、<input>、<button>等。

### 块级元素display:block;。

#### 特点

* 独占一行，自动换行。
* 可以指定宽高。
* margin和padding在四个方向上都有效。

#### 常见的块级元素

<div>、<aside>、<nav>、<form>、<ul>、<li>等

### 行内块级元素display:inline-block;。

#### 特点

* 可以指定宽高。
* margin和padding在四个方向上都有效。
* 元素排在一行，但是会有空白间隙。

#### 常见的行内块级元素

<button>、<img>、<input>、<select>、<textarea>、<iframe>。

## Canvas相关

只是一个二维网格的图形容器，绘制图形要用js来定位绘制。

比如画一条线，要先找到canvas元素，创建context对象，用moveTo和lineTo定位，再用stroke绘制。

var c=document.getElementById("myCanvas");   
var ctx=c.getContext("2d");   
ctx.moveTo(0,0);   
ctx.lineTo(200,100);   
ctx.stroke();

再比如画圆，也要先找到canvas元素，创建context对象，用beginPath开始路径，用arc确定圆的位置和大小，然后closePath关闭路径，最后再用stroke或fill绘制。

var c=document.getElementById("myCanvas");  
var ctx=c.getContext("2d");  
ctx.beginPath();  
ctx.arc(95,50,40,0,2\*Math.PI);  
ctx.closePath();  
ctx.stroke();

## HTTP与HTTPS的区别

http是超文本传输协议，而https相当于安全版的http。

具体区别：

* http是明文传输，而https是有着ssl加密传输协议的。
* http的端口是80，而https的端口是443。
* https需要ssl证书，费用比较昂贵,而http不需要。

## HTTP1.0、HTTP1.1、HTTP2.0的特点

### HTTP1.0的特点

* 协议版本信息会随着每个请求发送，即HTTP 1.0被追加到了GET行。
* 引入请求头，在发起请求时候会通过HTTP请求头告诉服务器它期待服务器返回什么类型的文件、采取什么形式的压缩、提供什么语言的文件以及文件的具体编码。
* 引入响应头，服务器以请求头中信息准备数据，并以响应头的信息告诉客户端数据采用何种格式返回，倘若遇到不支持的格式，只能返回服务器支持的格式，并在响应头中体现，也就是说最终浏览器是以响应头的信息解析数据。
* 引入状态码，状态码会在响应开始时发送，使浏览器能了解请求执行成功或失败，并相应调整行为。
* 引入了缓存机制，通过状态码与If-Modified-Since、Expires等控制更新或使用本地缓存。
* 引入了Content-Type头，使HTTP具备了传输除纯文本HTML文件以外其他类型文档的能力。

### HTTP1.1的特点

* 缓存处理，HTTP 1.1引入了更多的缓存控制策略，例如Entity tag、If-Unmodified-Since、If-Match、If-None-Match等更多可供选择的缓存头来控制缓存策略。
* 带宽优化以及网络连接的使用，在请求头中引入了range，它允许只请求资源的某一个部分，即返回206状态码，这样方便了开发者自由选择以便充分利用带宽和链接，并且可以使用Range和Content-Range制作断点续传功能。
* 错误通知的管理，在HTTP 1.1中新增了24个错误状态码。
* 增加Host请求头，能够使不同域名配置在同一个IP地址的服务器上。
* 支持长连接，HTTP 1.1支持长连接，在一个TCP连接上可以传输多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟，在HTTP 1.1中默认开启Connection：keep-alive，一般浏览器对于同一个域名允许同时建立6个长链接。
* 增加管线化技术，允许在第一个应答被完全发送之前就发送第二个请求，以改善队头阻塞问题，但响应的顺序还是会按照请求的顺序返回。
* 支持响应分块，通过设置Transfer-Encoding: chunked进行分块响应，允许响应的数据可以分成多个部分，配合服务端尽早释放缓冲可以获得更快的响应速度。

### HTTP2.0的特点

* 二进制分帧，HTTP 2.0是二进制协议而不是文本协议，将所有传输的信息分割为更小的消息和帧,并对它们采用二进制格式的编码。
* 多路复用，并行的请求能在同一个链接中处理，在同一域名下所有访问都是从同一个TCP连接中走，HTTP消息被分解为独立的帧，服务端根据标识符和首部将消息重新组装起来，移除了HTTP 1.1中顺序和阻塞的约束。
* 压缩header，header在一系列请求中常常是相似的，其移除了重复和传输重复数据的成本。
* 服务端推送，服务器可以主动地向客户端推送资源，而无需客户端明确的请求。

## HTTP请求方法

|  |  |
| --- | --- |
| 请求方法 | 方法含义 |
| get | 请求页面信息返回响应主体 |
| post | 提交数据 |
| head | 获取响应头信息，常用于查看客户端性能 |
| options | 请求服务器返回该资源所支持的所有请求方法，常用于客户端查看服务器的性能 |
| trace | 请求服务器回显其收到的请求信息，常用于HTTP请求的测试或诊断 |
| put | 指定资源位置上传最新内容 |
| delete | 删除所请求URI标识的资源 |
| connect | 将连接改为管道方式的代理服务器，常用于SSL加密服务器与非加密的HTTP代理服务器的通信 |

## Get和Post的区别

* get参数放在URL中传递、post参数放在request body中传递。
* get只能进行URL编码，而post可以用多种方式编码。
* get能够使用缓存，post不行。
* get传递参数有长度限制，post没有。
* get产生一个TCP包，而post产生两个TCP包（get把header和data一并发送，而post先发送header，响应100后再继续发送data）。
* get参数会被保留在历史记录里，而post不会。
* get浏览器回退无害，post会再次请求。
* get满足幂等性，post不满足。（幂等性：一次请求和多次请求某一资源具有同样的副作用，put、delete都满足）

## HTTP状态码

1XX代表通知，2XX代表操作成功，3XX代表重定向，4XX代表客户端错误，5XX代表服务端错误。

|  |  |
| --- | --- |
| 状态码 | 含义 |
| 200 | 客户端请求成功 |
| 301 | 永久移动，资源移动到新的URI中，之后用新的URI请求资源 |
| 302 | 暂时移动，资源临时移动，可以用旧的URI请求资源 |
| 304 | 未修改，请求的资源没有改动，不返回任何资源 |
| 400 | 客户端请求语法错误 |
| 401 | 需要用户验证 |
| 404 | 请求资源不存在 |
| 500 | 服务器发生错误 |
| 502 | 服务器作为网关或代理，从上游服务器收到无效响应。 |

## HTTP头部

### 请求消息Request

请求消息一般包括请求行、请求头部、空行和请求数据四个部分。

POST / HTTP1.1  
Host:www.wrox.com  
User-Agent:Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 2.0.50727; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022)  
Content-Type:application/x-www-form-urlencoded  
Content-Length:40  
Connection: Keep-Alive  
  
name=Professional%20Ajax&publisher=Wiley

### 响应消息Response

响应消息一般包括状态行、消息包头、空行和响应正文四个部分。

HTTP/1.1 200 OK  
Date: Fri, 22 May 2009 06:07:21 GMT  
Content-Type: text/html; charset=UTF-8  
  
<html>  
 <head></head>  
 <body>  
 <!--body goes here-->  
 </body>  
</html>

## 输入URL后发生了什么

1. 寻找服务器ip，现在缓存中找，然后在本地hosts文件里找，最后查询DNS服务器。
2. 建立TCP连接，发送http请求，依次经过传输层、网络层、数据链路层和物理层到达服务器，服务器解析请求返还相应的html，关闭TCP连接。
3. 浏览器根据返回的html构建DOM树，中途如果遇到图片、视频等资源会并行下载，如果遇到js脚本，会先下载相应的js脚本，这会造成阻塞，然后根据样式构建CSSOM树，接着两棵树合成render树。
4. 最后进行布局绘制。

\*window.onload在页面载入完成时执行，DOMContentLoaded在DOM树构建完成时执行。

## HTTPS加密方式

### 加密过程

1. 浏览器通过url向服务器发送请求，要求建立SSL连接。
2. 服务器接受请求后返回公钥证书。
3. 浏览器验证公钥证书是否有效，如果有效就生成会话密钥，并用公钥加密会话密钥发送给服务器。
4. 服务器通过自己的私钥解密会话密钥。此时两方都有了相同的会话密钥。
5. 服务器与浏览器通过会话密钥加密双方的通信。

### 对称加密与非对称加密

对称加密：加密和解密使用同一个密钥。

非对称加密：发送端使用公钥加密，接收端使用私钥解密。

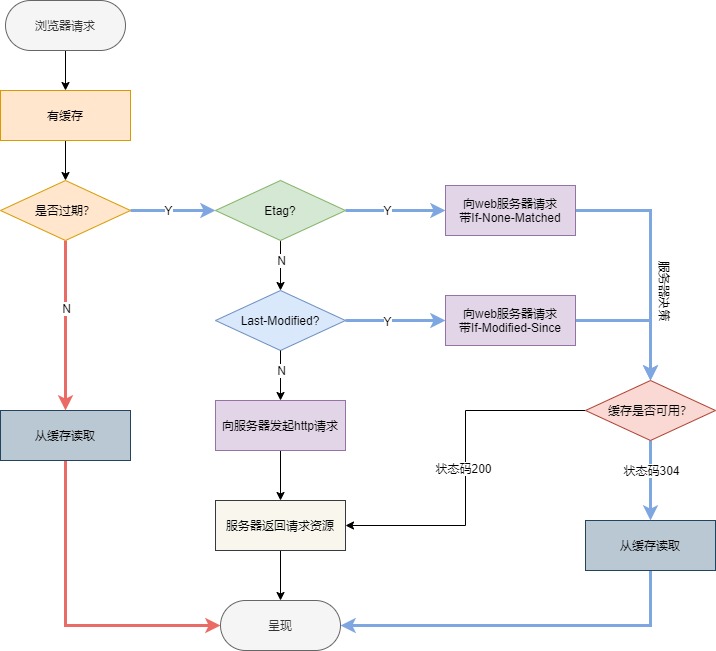
* RSA算法：两个大素数的乘积作为公钥，两个大素数作为私钥。（乘积因式分解困难）

HTTPS使用非对称加密传输对称密钥，使用对称密钥传输数据。

## HTTP缓存（浏览器缓存）

* 强缓存：直接从本地缓存读取资源，返回状态码200。
* 协商缓存：向服务器发送请求，由服务器通知缓存是否可以使用，返回状态码304。

浏览器首先查看头部信息，如果cache-control值为no-cache或者资源时间（max-age）失效会命中协商缓存，这时会发送一个请求到服务器，服务器比较if-None-Match和之前的Etag值是否相等，如果没有Etag值就再比较if-Modify-Since和Last-Modify值是否相等，如果相等那返回304，否则返回最新的资源。



## 会话跟踪

### Cookie

cookie是服务器发送到浏览器并保存在本地的一小块数据，对于同源的每个请求都会自动携带cookie，于是服务端就可以判断用户身份。

### Session

session代表着服务器与客户端一次会话的过程。对于客户端的每个会话，都有一个唯一的SESSIONID与其对应，服务端将用户数据存储进SESSIONID对应的文件或者是内存中，只要客户端每次请求将SESSIONID交予服务端，服务端就能区别用户进行会话跟踪。

### Cookie和Session的区别

* cookie保存在客户端，session保存在服务器端。
* cookie只能保存ASCII码，session可以保存任意数据类型。
* cookie相较于session不安全。
* cookie只有4K左右大小，session可存取数据远高于cookie。
* cookie可以长时间保存，session一般客户端关闭或者session超时都会失效。

### Cookie和Session的配合

用户第一次请求服务器时，服务器会根据用户提交的信息创建对应的session，然后会把SESSIONID返回给浏览器，浏览器将SESSIONID存入cookie，同时记录SESSIONID的域名。

当用户再次请求服务器时，请求会自动判断当前域名是否存在cookie，如果存在就将cookie一起发送给服务器，服务器会从中获取SESSIONID，再通过SESSIONID查找相应的session，如果没有找到就说明用户没有登陆或者登录失效，如果找到就执行之后的操作。

### 浏览器禁用Cookie

* 每个请求都带有SESSIONID参数信息。
* Token机制：当用户第一次登录后，服务器根据提交的用户信息生成一个 Token，响应时将 Token 返回给客户端，以后客户端只需带上这个 Token 前来请求数据即可，无需再次登录验证。

### 分布式Session（共享Session）

将session存在数据库里，保障分发到每一个服务器的响应结果都一致。

## 浏览器存储

### Cookie

属于DOM树根节点document，在请求头上带着数据，大小限制4K。

* expires：过期时间。
* path：路径，相同路径的页面可以共享cookie。
* domain：主机名。

### Session Storage

BOM对象window，以键值对的形式存在，存储类型是String类型，存储大小大约为5M。在用户关闭浏览器后数据就会失效。

### Local Storage

BOM对象window，以键值对的形式存在，存储类型是String类型，存储大小大约为5M。永久存在，除非手动删除。

### Cookie、Local Storage和Session Storage的区别

* cookie只有4k左右的大小，而sessionStorage和localStorage有5MB的大小。
* cookie在时间过期前都有效，sessionStorage窗口关闭之前有效，而localStorage永久有效。
* cookie和localSorage可以同源窗口共享，而sessionStorage不行。
* cookie在浏览器和服务器之间来回传递，localStorage和sessionStorage仅在本地保存。

\*两个同源页面交互可以使用localStorage，不同源的话就两个页面嵌入相同的iframe实现交互。

## XSS攻击

页面被注入恶意脚本，使之在用户浏览器上运行，利用这些脚本获取用户敏感信息从而危害数据安全。

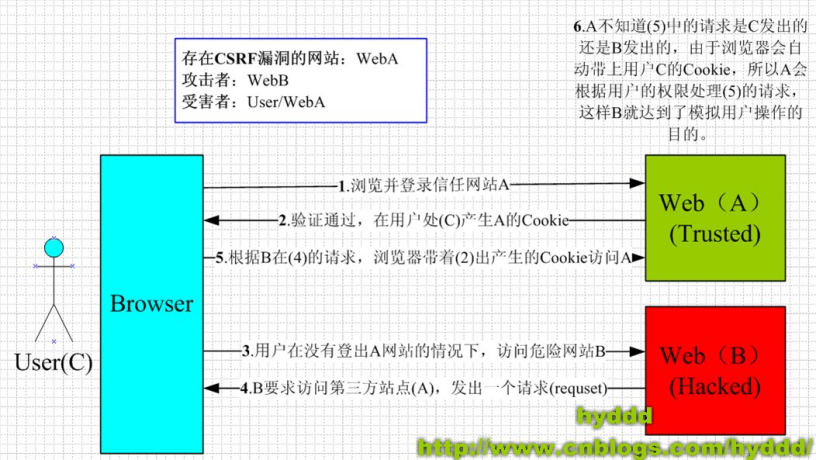
分类：

* 存储型：恶意代码存在数据库中，用户打开网站时，服务器取出恶意代码，拼接在html中返回给浏览器，然后解析执行。
* 反射型：构造包含恶意代码的URL，用户打开URL后，服务器将恶意代码取出，拼接在html中返回给浏览器，然后解析执行。
* DOM型：构造包含恶意代码的URL，用户打开URL后，浏览器收到响应后，前端js取出恶意代码解析执行。

防御：cookie设置httponly和secure；进行特殊字符过滤；对用户的输入进行检查。

## CSRF攻击

可以理解为攻击者盗用了用户的身份，以用户的名义发送了恶意请求。



防御：使用验证码、使用token。

## 跨域

原因：浏览器同源策略：非同源的不能够交互。（协议、域名、端口）

解决方案：

* CORS
  + 简单请求：请求中有origin头部，其中包含请求页面的源信息，服务器根据这个头部信息来决定是否响应，如果服务器认为这个请求可以接受就在Access-Control-Allow-Origin头部中会发相同的源信息，如果源信息不匹配就驳回请求。
  + 非简单请求：发送真正的请求前会发送一个Preflight请求给服务器，该请求使用OPTIONS方法，发送Origin、Access-Control-Request-Method头部，然后服务器决定是否允许这种类型的请求，通过在响应中发送头部与浏览器沟通，一旦服务器通过Preflight请求允许该请求后，浏览器就可以正常的CORS请求了。
  + 当Access-Control-Allow-Credentials的值为true时可以携带cookie。
* JSONP
  + js不受浏览器同源策略的影响，可以通过script标签进行跨域请求。
  + 只支持Get请求。（请求js文件只能用Get）
* 服务端代理：有跨域的请求操作时发送请求给后端，让后端帮你代为请求，然后最后将获取的结果发送给你。

## 前端性能优化

1. 降低请求量：合并CSS、JS和图片（精灵图）。
2. 使用缓存：http缓存（浏览器缓存）。
3. 加快请求速度：预解析DNS、CDN分发（本质缓存）。
4. 将CSS样式表放在顶部（CSS全部下载完才对页面进行渲染），JS脚本放在底部（加载JS后立即执行可能会阻塞整个页面）。
5. 图片懒加载：访问页面时先把图片替换成一张占位图，当图片出现在浏览器可视区域时（如果图片顶部到页面顶部的距离<=可视区域高度+滚动区域高度的和时代表进入可视区域），才显示真正的图片内容。
6. 减少cookie的传输（太大会严重影响数据传输，避免请求静态资源时发送cookie）。

## 重排重绘

* 重排：部分或整个渲染树需要重新分析并且节点尺寸需要重新计算，比如浏览器窗口大小改变、元素尺寸或位置改变。
* 重绘：由于节点的样式发生改变比如字体颜色改变，屏幕上的部分内容需要更新。

## CDN

内容分发网络，能够通过DNS查找离用户最近的CDN节点的IP，然后通过IP访问实际资源，如果CDN上没有缓存资源，就会到源站请求资源并缓存到CDN，这样用户下一次访问，就直接用CDN缓存的资源了。cache-control设置为public（任何情况下都要缓存该资源），提升缓存量。

## DNS

域名系统、将域名和IP地址相互映射的一个分布式数据库。

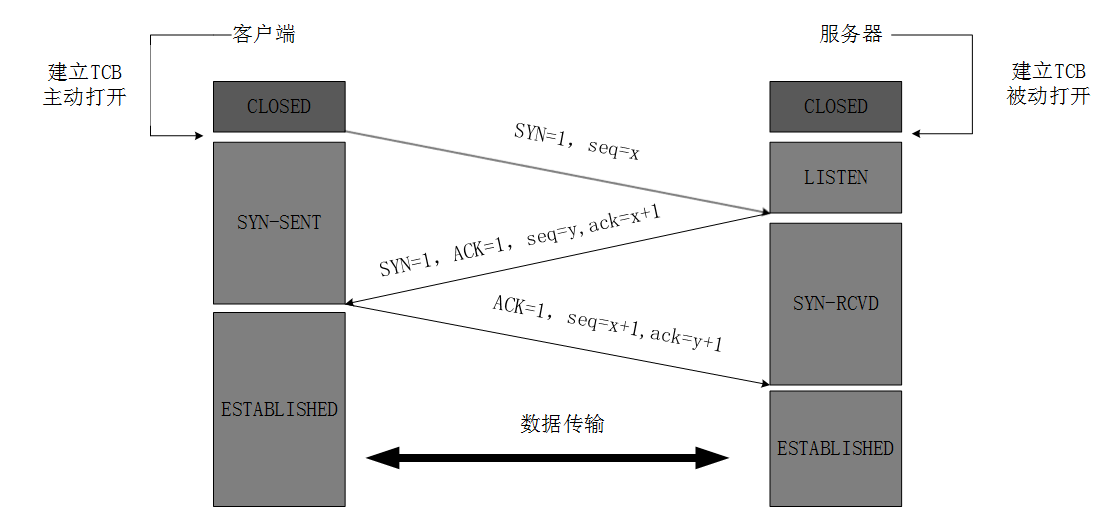
* 主机向本地域名服务器的查询一般都是采用递归查询。如果主机所询问的本地域名服务器不知道被查询域名的IP地址时，本地域名服务器就以DNS客户的身份向其他根域名服务器继续发送查询请求报文，而不是让该主机自己进行下一步的查询。
* 本地域名服务器向根域名服务器的查询通常是采用迭代查询。当根域名服务器收到本地域名服务器发出的迭代查询请求报文时，要么给出所要查询的IP地址，要么告诉本地域名服务器下一步应当向哪一个域名服务器进行查询，然后让本地域名服务器进行后续的查询。逐步按照域树的路径向下走直到叶节点，得到了所要解析的域名的IP地址，然后把这个结果返回给发起查询的主机。当然本地域名服务器也可以采用递归查询，这取决于最初的查询请求报文的设置是要使用哪一种查询方式。

## TCP相关

* TCP是面向连接的协议，提供全双工通信。
* TCP使用校验、确认和重传机制来保证可靠传输。
* TCP首部最小有20字节。
* TCP只能是一对一通信。
* TCP面向字节流通信。
* TCP 使用滑动窗口机制来实现流量控制，通过动态改变拥塞窗口的大小进行拥塞控制。

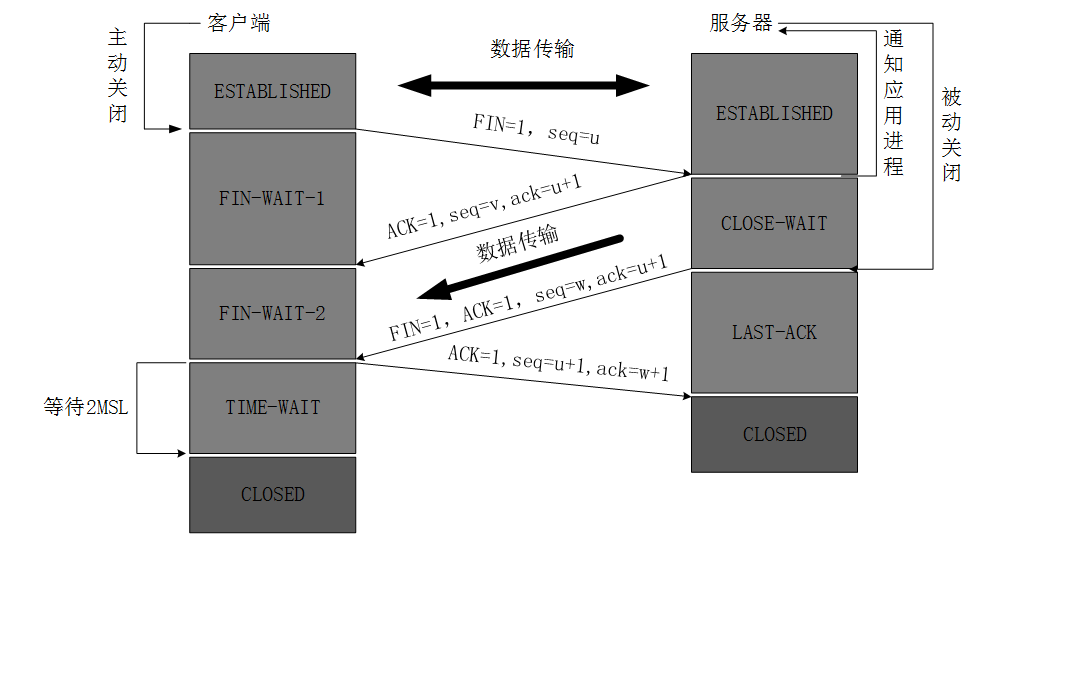
### 三次握手

1. 第一次握手：建立连接。客户端发送连接请求报文段，将SYN位置为1，seq随机的为x；然后，客户端进入SYN\_SEND状态，等待服务器端的确认。服务器端由SYN=1知道，客户端请求建立连接；
2. 第二次握手：服务器端收到SYN报文段，需要确认联机信息，设置ack为x+1(客户端的seq+1)；同时，自己还要发送SYN请求信息，将SYN位置为1，seq再随机为y；此时服务器进入SYN\_RECV状态；
3. 第三次握手：客户端收到后检查ack是否正确。即第一次发送的seq+1,以及位码ack是否为1，若正确，客户端会再发送ack=(服务器的seq+1),ack=1，服务器收到后确认seq值与ack=1，则连接建立成功。客户端和服务器端都进入ESTABLISHED状态，完成TCP三次握手。



### 四次挥手

1. 第一次挥手：客户端发送一个FIN，用来关闭客户端到服务器端的数据传送，客户端进入FIN\_WAIT\_1状态。
2. 第二次挥手：服务器端收到FIN后，发送一个ACK给客户端，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号），服务器端进入CLOSE\_WAIT状态。
3. 第三次挥手：服务器端发送一个FIN，用来关闭服务器端到客户端的数据传送，服务器端进入LAST\_ACK状态。
4. 第四次挥手：客户端收到FIN后，客户端进入TIME\_WAIT状态，接着发送一个ACK给服务器端，确认序号为收到序号+1，服务器端进入CLOSED状态，完成四次挥手。



### TCP流量控制

防止发送端发的太快，耗尽接收方的资源，从而使接收方来不及处理。机制是丢包。用动态改变滑动窗口大小实现。

滑动窗口中的数据：已发送但还未收到确认；还未发送但等待发送。

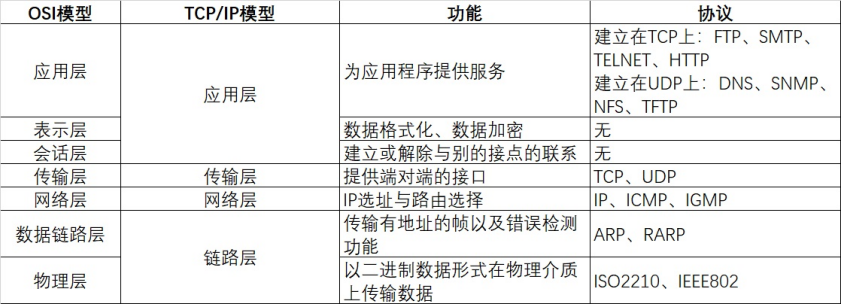
## TCP拥塞控制

防止发送方发的太快，是网络来不及处理，从而导致网络拥塞。

### 与UDP的区别

* UDP无连接，不可靠。
* UDP面向报文。
* UDP可以一对一或者一对多。
* UDP首部只有8字节。
* UDP用于高速传输和对实时性有较高要求的通信（视频、音频等多媒体通信）或广播通信。

## 网络模型



# CSS相关

## link和@import的区别

* link是html提供的标签，@import是CSS提供的。
* link没有兼容性问题，而@import有。
* link加载与页面加载同时进行，而@import要等页面加载完后加载。
* link权重高于@import。

## flex布局

弹性布局，设置该布局会导致子元素的float、clear和vertical-align失效。

|  |  |
| --- | --- |
| 属性 | 作用 |
| flex-direction | 决定主轴方向，即项目的排列方向 |
| flex-wrap | 决定是否换行 |
| justify-content | 决定项目在主轴上的对齐方式 |
| align-items | 决定项目在交叉轴上的对齐方式 |
| flex-grow | 定义项目的放大比例，默认为0 |
| flex-shrink | 定义项目的缩小比例，默认为1 |
| flex-basis | 定义在分配多余空间前，项目占据的主轴空间 |
| flex | flex-grow,flex-shrink和flex-basis的简写，默认值为0 1 auto,flex:1的意思是flex:1 1 任意数字+任意长度单位 |

## 三栏布局

* flex布局实现

<body>  
 <div class="box">  
 <div class="left">left</div>  
 <div class="center">center</div>  
 <div class="right">right</div>  
 </div>  
</body>  
  
<style>  
 .box {  
 display: flex;  
 }  
  
 .left {  
 background-color: green;  
 width: 100px;  
 }  
  
 .center {  
 flex: 1;  
 background-color: yellow;  
 }  
  
 .right {  
 background-color: red;  
 width: 100px;  
 }  
</style>

* float布局实现

<body>  
 <div class="box">  
 <div class="left">left</div>  
 <div class="right">right</div>  
 <div class="center">center</div>  
 </div>  
</body>  
  
<style>  
 .left {  
 float: left;  
 background-color: green;  
 width: 100px;  
 }  
  
 .center {  
 margin-left: 100px;  
 margin-right: 100px;  
 background-color: yellow;  
 }  
  
 .right {  
 float: right;  
 background-color: red;  
 width: 100px;  
 }  
</style>

* 绝对定位实现

<body>  
 <div class="box">  
 <div class="left">left</div>  
 <div class="center">center</div>  
 <div class="right">right</div>  
 </div>  
</body>  
  
<style>  
 .left {  
 position: absolute;  
 left: 0;  
 background-color: green;  
 width: 100px;  
 }  
  
 .center {  
 position: absolute;  
 left: 100px;  
 right: 100px;  
 background-color: yellow;  
 }  
  
 .right {  
 position: absolute;  
 right: 0;  
 background-color: red;  
 width: 100px;  
 }  
</style>

## 两列等高布局

* margin和padding正负抵消

<body>  
 <div class="box">  
 <div class="left"></div>  
 <div class="right"></div>  
 </div>  
</body>  
  
<style>  
 .box {  
 overflow: hidden;  
 }  
  
 .left {  
 width: 100px;  
 height: 200px;  
 float: left;  
 background: #149BDF;  
 padding-bottom: 5000px;  
 margin-bottom: -5000px;  
 }  
  
 .right {  
 width: 200px;  
 height: 400px;  
 margin-left: 150px;  
 background: #32CD32;  
 }  
</style>

* flex布局实现

<body>  
 <div class="box">  
 <div class="left"></div>  
 <div class="right"></div>  
 </div>  
</body>  
  
<style>  
 .box {  
 display: flex;  
 }  
  
 .left {  
 width: 100px;  
 height: 200px;  
 background: #149BDF;  
 }  
  
 .right {  
 width: 200px;  
 background: #32CD32;  
 }

* border实现

<body> <div class="box"> <div class="left"></div> <div class="right"></div> </div></body><style> .box { border-left: 100px solid #149BDF; } .left { width: 100px; height: 200px; float: left; margin-left: -100px; } .right { width: 200px; height: 400px; margin-left: 150px; background: #32CD32; }</style>

## 盒模型

box-sizing：content-box/border-box。

content-box：标准盒模型，盒子宽度 = width + padding + border。

border-box：怪异盒模型，盒子宽度 = width。

## position的值

|  |  |
| --- | --- |
| 属性 | 作用 |
| fixed | 元素位置相对于浏览器窗口是固定的，即使窗口滚动也不会移动。fixed定位使元素脱离文档流，因此不占据空间。fixed定位元素会和其他元素重叠。 |
| absolute | 元素位置相对于最近的已定位父元素，若没有则相对于html。absolute定位使元素位置脱离文档流，因此不占据空间。absolute定位元素和其他元素重叠。 |
| relative | 相对定位的元素将出现在它所在的位置上，然后可通过top、bottom、right、left让这个元素相对于它的初始位置移动。使用相对定位时无论元素是否移动仍占据原来空间，因此，移动元素会导致它覆盖其他框。 |
| static | 默认值，没有定位。元素出现在正常的流中（忽略top、bottom、right、left、z-index声明） |
| inherit | 继承父元素的position属性值 |
| sticky | 元素先按普通文档流进行定位，而后，元素定位表现为在跨越特定值前为相对定位，之后为固定定位。 |

## 隐藏元素的方法

* display:none：相当于删除元素，布局改变，不可以触发事件。
* opacity:0：元素隐藏，页面布局不变，可以触发事件。
* visibility:hidden：元素隐藏，页面布局不变，不可以触发事件。

## 伪类和伪元素

伪类用于当前元素某个状态时为其添加样式，比如:hover、:visited。

伪元素是创建不在文档树中的元素，并为其添加样式，比如::before。

## BFC

块级格式化上下文，独立容器，不会影响到外面的元素。

触发：

* float除none以外的值
* position是absolute和fixed
* display是flex/inline-block/table-cell/inline-flex/table-caption
* overflow除了visible以外的值

特性（作用）：

* 内部的Box会在垂直方向上一个接一个的放置。
* 垂直方向上的距离由margin决定。
* BFC的区域不会与float的元素区域重叠。
* 计算BFC的高度时，浮动元素也参与计算。

\*margin塌陷：父子元素设置margin-top时选择大的，如果子元素的margin-top大于父元素的，两个就会一起往下跑。BFC解决。

## 水平垂直居中

* flex实现

<body>  
 <div class="box">  
 <div class="content">content</div>  
 </div>  
</body>  
  
<style>  
 .box {  
 display: flex;  
 justify-content: center;  
 align-items: center;  
 width: 100px;  
 height: 100px;  
 background-color: green;  
 }  
  
 .content {  
 width: 50px;  
 height: 50px;  
 background-color: red;  
 }  
</style>

* margin:auto实现

<body>  
 <div class="box">  
 <div class="content">content</div>  
 </div>  
</body>  
  
<style>  
 .box {  
 position: relative;  
 width: 100px;  
 height: 100px;  
 background-color: green;  
 }  
  
 .content {  
 position: absolute;  
 margin: auto;  
 top: 0;  
 left: 0;  
 right: 0;  
 bottom: 0;  
 width: 50px;  
 height: 50px;  
 background-color: red;  
 }

* 负margin法

<body>  
 <div class="box">  
 <div class="content">content</div>  
 </div>  
</body>  
  
<style>  
 .box {  
 position: relative;  
 width: 100px;  
 height: 100px;  
 background-color: green;  
 }  
  
 .content {  
 position: absolute;  
 margin: auto;  
 top: 50%;  
 left: 50%;  
 width: 50px;  
 height: 50px;  
 margin-top: -25px;  
 margin-left: -25px;  
 background-color: red;  
 }  
</style>

## CSS3新增内容

* CSS3边框：border-radius、box-shadow、border-image
* CSS3背景：background-size、background-origin（规定背景图片定位区域）
* CSS3文字效果：text-shadow、word-wrap（对长单词进行拆分并换行）
* CSS3 2D转换：transform: translate()/rotate()/scale()
* CSS3 3D转换：transform: rotateX()/rotateY()
* CSS3 过渡：transition
* CSS3 动画：animation
* CSS3用户界面：resize（是否可由用户调整元素尺寸）、box-sizing（定义盒模型）

## PX、REM和EM的区别

* px是固定像素，无法因为页面大小而改变；
* em,rem比px灵活，是相对长度单位，更适用于响应式布局；
* em相对于父元素，rem相对于根元素。

## 选择器优先级

!important > 行内选择器 > id选择器 > class选择器 > 标签选择器

## CSS实现三角形

margin-top: 100px;  
    width: 0;  
    height: 0;  
    /\*底边\*/  
    border-top: 100px solid red;  
    /\*斜边\*/  
    border-left: 100px solid transparent;

# JS相关

## 数组常用方法

|  |  |
| --- | --- |
| push | 在末尾插入元素，返回数组长度，修改原数组 |
| pop | 删除末尾元素，返回被删除的元素，修改原数组 |
| shift | 删除头部元素，返回被删除的元素，修改原数组 |
| unshift | 在头部插入元素，返回数组长度，修改原数组 |
| map | 每个元素都会调用提供的函数，返回新数组，不会更改原数组 |
| forEach | 每个元素都会调用提供的函数，没有返回值，修改原数组 |
| reduce | 让数组前项和后项做计算，返回值为最后结果 |
| sort | 排序，返回新数组，原数组改变 |
| filter | 对所有的元素进行判断，满足条件的元素组成新数组返回，不改变原数组 |
| concat | 合并数组或元素，返回新的数组 |
| splice | 添加删除或替换元素，返回删除或替换的值，改变原数组 |
| slice | 截取复制数组指定内容，返回新数组 |
| join | 指定字符连接字符串 |

## 闭包

可以访问其它函数内部变量的函数。

作用：

* 在函数外部访问函数内部的变量。
* 让变量保存在内存中，不会随着函数的结束而销毁。

缺点：会增大内存使用量，使用不当会造成内存泄漏。

使用场景：防抖节流、回调（点击触发事件）

\*内存泄漏：已分配的内存未释放或者无法释放。比如：闭包和未清理的定时器。

## 垃圾回收

js垃圾回收器监视所有的对象，会把不可访问的对象删除。所谓的可访问就是可以从根访问到该值。

* 标记清除：垃圾回收器获取根并标记它们，然后标记它们的引用以及子孙代的引用，没有被标记的就被删除。
* 引用计数：记录每个对象被引用的次数，计数器为0就直接回收内存。无法处理相互引用的对象。

谷歌工具：performance里的JS Heap，呈上升趋势就有泄漏。

## Promise相关

构造函数接收一个函数，这个函数有两个参数resolve和reject，它们都是函数，异步操作成功就调用resolve，出错就调用reject。

Promise有自己的catch。

Promise A+规则：

* 三种状态：pending、resolved、rejected，只能从pending到resolved或者pending到rejected，并且不能再转变为其他状态。
* resolve必须有一个value值，reject必须有一个reason值，两个值都不能改变。
* 必须提供.then方法来获取value或reason，并且返回一个promise。
* .then接收两个参数，一个是onFulFilled，它在是resolved状态后调用，一个是onRejected，它在rejected状态后调用。

Promise.all：接收数组参数，等所有异步操作执行成功了才执行回调或者有一个失败了就执行回调。

Promise.race：谁先执行完成谁就先执行回调，其余的将不会再进入race的回调。

代码实现：

function mypromise(constructor) {  
 self = this;  
 self.value;  
 self.err;  
 self.status = 'pending';  
 function reslove(value) {  
 if (self.status === 'pending') {  
 self.status = 'resolved';  
 self.value = value;  
 }  
 }  
 function reject(err) {  
 if (self.status === 'pending') {  
 self.status = 'rejected';  
 self.err = err;  
 }  
 }  
 try {  
 constructor(reslove, reject);  
 }  
 catch (e) {  
 reject(e);  
 }  
 }  
 mypromise.prototype.then = function (onFullfilled, onRejected) {  
 let self = this;  
 switch (self.status) {  
 case "resolved":  
 onFullfilled(self.value);  
 break;  
 case "rejected":  
 onRejected(self.err);  
 break;  
 default:  
 }  
 }

function promiseAll(list) {  
 var finishedVal = new Array(list.length);  
 var count = 0;  
 return new Promise((resolve, reject) => {  
 for (var i = 0; i < list.length; i++) {  
 //不是promise对象,就用promise.resolve转成promise对象  
 Promise.resolve(list[i].then((val) => {  
 finishedVal[i] = val;  
 count++;  
 if (count === list.length) {  
 return resolve(finishedVal);  
 }  
 }, function (err) {  
 return reject(err);  
 }))  
 }  
 })  
 }

function promiseRace(list) {  
 return new Promise((resolve, reject) => {  
 for (var i = 0; i < list.length; i++) {  
 Promise.resolve(list[i].then((val) => {  
 return resolve(val);  
 }, function (err) {  
 return reject(err);  
 }))  
 }  
 })  
 }

## let、var和const的区别

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 类型 | 变量提升 | 暂时性死区（声明变量之前变量不可用） | 重复声明 | 初始值 | 作用域 |
| var | 存在 | 不存在 | 允许 | 不需要 | 全局作用域、函数作用域 |
| let | 不存在 | 存在 | 不允许 | 不需要 | 块级作用域 |
| const | 不存在 | 存在 | 不允许 | 需要 | 块级作用域 |

## 变量提升

作用域链：函数使用到的变量在当前作用域没有找到值，就会向创建这个函数的作用域里去查，直到查到全局作用域。

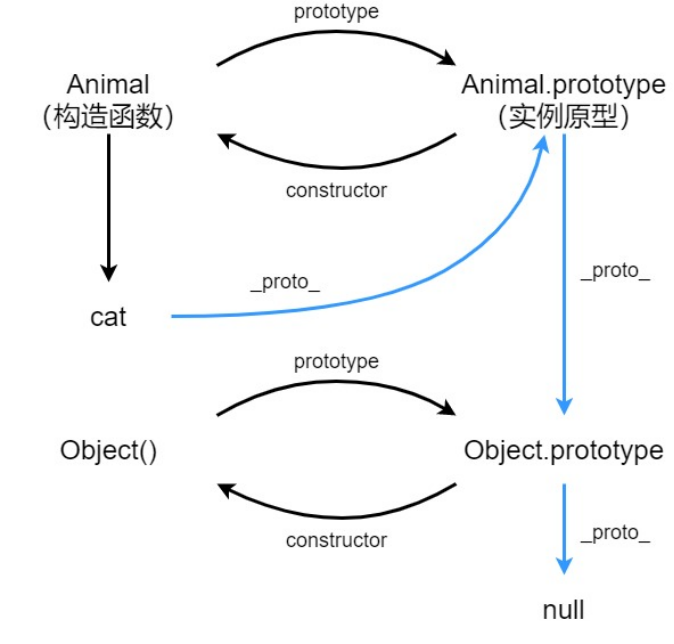
变量提升：变量声明提到函数最顶端，但是赋值不会提升。函数声明在变量声明后面。

## 事件循环

js先执行宏任务，再执行全部的微任务，然后继续新的宏任务。

* 宏任务：script(整体代码), setTimeout, setInterval, setImmediate, I/O, UI rendering。
* 微任务：process.nextTick, Promises, Object.observe等。

## 原型链



## 继承

* 原型链继承：父类实例作为子类原型
* 特点：基于原型链，既是父类的实例，又是子类的实例。  
  缺点：无法实现多继承，无法向父类构造参数传参。

function Parent() {  
 this.name = 'parent';  
}  
function Child() {  
 this.type = 'child';  
}  
Child.prototype = new Parent();

* 构造继承：使用父类构造函数增强子类实例
* 特点：可以实现多继承。  
  缺点：只能继承父类的实例属性和方法，不能继承原型上的属性和方法。

function Parent() {  
 this.name = 'parent';  
}  
function Child() {  
 Parent1.call(this);  
 this.type = 'child';  
}

* 组合继承：构造继承+原型链继承
* 特点：可以继承实例属性和方法，也可以继承原型属性和方法。  
  缺点：调用两次父类构造函数，会生成两份实例。

function Parent() {  
 this.name = 'parent';  
}  
function Child() {  
 Parent.call(this);  
 this.type = 'child';  
}  
Child.prototype = new Parent();

## new的过程

1. 创建空对象；
2. 将对象的原型指向构造函数的原型；
3. 将this指向新对象；
4. 返回这个对象。

function myNew(fn) {  
 var obj = new Object();  
 obj.\_proto\_ = fn.prototype;  
 var result = fn.apply(obj);  
 return result instanceof Object ? result : Obj;  
}

## 箭头函数

* 箭头函数不能作为构造函数。
* 箭头函数没有自己的this，从作用域链上继承。
* 箭头函数没有自己的arguments，会从外部函数上继承。
* 箭头函数没有原型。

## 防抖节流

* 防抖：事件触发n秒后再调用函数，如果n秒内再次触发事件，就重新计时。（搜索框搜索输入）

function debounce(fn, delay) {  
 var timer;  
 return function () {  
 var self = this;  
 var args = arguments;  
 if (timer) {  
 clearTimeout(timer);  
 }  
 timer = setTimeout(function () {  
 fn.apply(self, args);  
 }, delay);  
 };  
 }

节流：每隔n秒就执行一次函数，中途触发不会执行函数。（滚动加载）

function throttle(fn, delay) {  
 var timer;  
 return function () {  
 var self = this;  
 var args = arguments;  
 if (timer) {  
 return;  
 }  
 timer = setTimeout(function () {  
 fn.apply(self, args);  
 timer = null;  
 }, delay);  
 };  
 }

## bind、call和apply的区别

都可以用来改变this指向。

* call有多个参数，第一个参数是this要指向的对象，以后的参数是数组里的元素。
* apply有两个参数，第一个参数是this要指向的对象，第二个参数为数组。
* bind会返回一个新的函数，还需要调用一次。

Function.prototype.bind2 = function (context) {  
 var self = this;  
 var args = Array.prototype.slice.call(arguments, 1);  
 var fNOP = function () { };  
 var fBound = function () {  
 var bindArgs = Array.prototype.slice.call(arguments);  
 return self.apply(this instanceof fNOP ? this : context, args.concat(bindArgs));  
 }  
 fNOP.prototype = this.prototype;  
 fBound.prototype = new fNOP();  
 return fBound;  
 }

## this的指向

始终指向调用它的对象。

* 默认绑定：指向全局对象。
* 隐式绑定：调用的对象。
* 显示绑定：call、apply、bind方法改变指向。
* New绑定：指向新创建的对象。

## 数据类型

* 基本数据类型：string、number、boolean、null、undefined和symbol。
* 引用类型：object、array和function。

区别：

* 基本数据类型存放在栈中，引用类型存放在堆中。
* 基本数据类型比较的是原始值，引用类型比较的是地址。

## ==和===

[] == ![]返回true：[] == false ----> [] == 0 ----> '' == 0 ----> 0 == 0 ----> true；

{} == !{}返回false：{} == false ----> {} == 0 ----> NaN == 0 ----> false；

null == undefined返回true；



ToPrimitive(X)即使用valueOf返回原始值，或者使用toString。

## 数组类型判断

* Array.isArray()方法。
* InstanceOf Array方法。

function myInstanceOf(target1, target2) {  
 // let proto = Object.getPrototypeOf(target1);  
 // if (proto === target2.prototype) {  
 // return true;  
 // } else if (!proto) {  
 // return false;  
 // } else {  
 // return myInstanceOf(proto, target2)  
 // }  
 return target2.prototype.isPrototypeOf(target1);  
 }

* constructor属性（可以被改写，不合适）。
* Object.prototype.toString.call()方法。

## DOM0事件和DOM2事件

* DOM0级：在元素上绑定事件，解绑时设置NULL。
* DOM2级：通过事件监听的形式绑定，元素可以有多个事件处理函数，addEventListener()和removeEventListener()。有三个过程：事件捕获阶段、处于目标阶段、事件冒泡阶段。（默认是冒泡，阻止默认preventDefault行为不能阻止冒泡）

\*先冒泡后捕获：监听到捕获事件就推迟执行。

\*阻止冒泡和捕获：stopPropagation()。

## 模块化

* CommonJS：一个模块就是一个脚本文件，对外接口是exports属性，require属性第一次加载会执行脚本，然后会在内存中生成一个对象，想要再次运行模块必须清理缓存。
* ES Modules：由export和import组成。CommonJS 模块就是对象，输入时必须查找对象属性。而 ES Modules 不是对象，而是通过 export 命令显式指定输出的代码。
* AMD：异步方式加载模块，不影响他后面语句的运行，所有依赖这个模块的语句，都定义在一个回调函数中，等到加载完成之后，这个回调函数才会运行。用define定义，require引用。

## 深浅拷贝

如果B复制了A，当修改A时，看B是否会发生变化，如果B也跟着变了，说明这是浅拷贝，如果B没变，那就是深拷贝。浅拷贝是拷贝一层，深层次的对象级别的就拷贝引用；深拷贝是拷贝多层，每一级别的数据都会拷贝出来。

function deepClone(target) {  
 let result  
 if (typeof target === 'object') {  
 if (Array.isArray(target)) {  
 result = []  
 for (let i in target) {  
 result.push(deepClone(target[i]))  
 }  
 } else if (target === null) {  
 result = target  
 } else if (target.constructor === RegExp) {  
 result = target  
 } else if (target.constructor === Date) {  
 result = target  
 } else {  
 result = {}  
 for (let i in target) {  
 result[i] = deepClone(targer[i])  
 }  
 }  
 } else if (typeof target === 'function') {  
 // 如果是函数  
 result = new Function('return '+ target.toString())  
 } else {  
 // 如果是基本数据类型，如number、string、boolean、undefined  
 result = target  
 }  
 return result  
}

## ES6新特性

* let、const声明变量，实现块级作用域。
* 引入promise、async/await解决异步问题。
* 新的数据类型symbol，新的数据结构map和set。
* 引入class。

## 解决JS加载过程阻塞问题

* defer：遇到带有defer属性的script标签时，浏览器会再开启一个线程去下载js文件，同时继续解析HTML文档，等HTML全部解析完毕DOM加载完成后，再去执行加载好的js文件。
* async：会在下载完毕后立刻执行。对于多个带有async的js文件，不保证按顺序执行，哪个js文件先下载完就先执行哪个。

## JS单线程的原因

js作为浏览器脚本语言，主要用途是与用户互动，以及操作DOM，这决定了它只能是单线程，否则会带来很复杂的同步问题。比如若js同时拥有两个线程，一个线程在某个DOM节点上添加内容，另一个线程删除这个节点，那浏览器将不知所措。

## 事件委托/事件代理

不在发生地设置监听函数而是在父元素上设置监听函数，通过事件冒泡，父元素可以监听到子元素上事件的触发，从而完成响应。

优点：

* 减少DOM操作，提高性能。
* 随时可以添加子元素。

## AJAX

创建对象、设置请求函数和回调函数、获取异步对象的readyState属性、判断响应报文的状态、读取响应数据。

var xmlhttp = new XMLHttpRequest();  
 xmlhttp.open("GET", "/try/ajax/demo\_get.php", true);  
 xmlhttp.send();  
 xmlhttp.onreadystatechange = function () {  
 if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {  
 document.getElementById("mydiv").innerHTML = xmlhttp.responseText;  
 }  
 }

function promiseAjax() {  
 return new Promise((resolve,reject) => {  
 var xmlhttp = new XMLHttpRequest();  
 xmlhttp.open("GET", "/try/ajax/demo\_get.php", true);  
 xmlhttp.send();  
 xmlhttp.onreadystatechange = function () {  
 if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {  
 resolve(JSON.parse(xmlhttp.responseText));  
 } else {  
 reject(JSON.parse(xmlhttp.responseText));  
 }  
 }  
 })  
 }

# VUE相关（结合项目）

## 生命周期

1. VUE实例创建--->beforeCreate：做了一些初始化操作，此时组件还没有被挂载，data、methods等也没有绑定。
2. beforeCreate--->created：完成了数据的绑定、计算属性和方法的挂载等，此时组件还没有挂载，但可以操作data和methods等。（请求数据）
3. created--->beforeMount：完成了页面模板的解析，页面存在内存中，还没有被渲染。
4. beforeMount-->mounted： 执行页面从内存中渲染到DOM的操作，渲染页面模板的优先级是render函数>template属性>外部html。（操作DOM，监听事件，获取元素属性）
5. mounted-->beforeUpdate：数据更新时调用，此时，VUE实例中的数据是最新的，但页面上还是旧的数据。
6. beforeUpdate-->updated：DOM渲染完成后调用，此时组件的DOM已经更新，可以执行依赖于DOM的操作。
7. updated-->beforeDestory：实例销毁前调用，实例任然可以用。
8. beforeDestory-->destoryed：实例销毁后调用，所有东西都不可用。

## 双向绑定

采用数据劫持的方式实现双向数据绑定，最核心的方法是通过Object.defineProperty()实现，它可以对数据添加属性描述符getter与setter实现劫持。

首先要对数据进行劫持监听，所以我们需要设置一个监听器Observer，用来监听所有属性。如果属性发生变化了，就需要告诉订阅者Watcher看是否需要更新。因为订阅者是有很多个，所以我们需要有一个消息订阅器Dep来专门收集这些订阅者，然后在监听器Observer和订阅者Watcher之间进行统一管理的。接着，我们还需要有一个指令解析器Compile，对每个节点元素进行扫描和解析，将相关指令对应初始化成一个订阅者Watcher，并替换模板数据或者绑定相应的函数，此时当订阅者Watcher接收到相应属性的变化，就会执行对应的更新函数，从而更新视图。

## computed和watch的区别

* computed：计算属性，可以根据所依赖数据动态显示新的计算结果，它可以进行缓存，如果所依赖的数据没有发生改变就不会再次执行函数。getter函数和setter函数。每个computed属性都会用watcher包装，计算watcher里的dirty为false时直接返回value，否则重新计算。
* watch：里面的immediate属性可以控制在watch中首次绑定的时候是否执行方法，deep属性可以让watch监听到对象内部属性的改变。

## 组件通信

* 父-->子：props
* 子->父：$emit，需要通过事件的触发。
* 父子：$children和$parent
* $ref：$refs适用于父子组件通信，ref被用来给元素或子组件注册引用信息，引用信息将会注册在父组件的$refs对象上，如果在普通的DOM元素上使用，引用指向的就是DOM元素，如果用在子组件上，引用就指向组件实例。要注意的是因为ref本身是作为渲染结果被创建的，在初始渲染的时候是不能访问它们的，此时它们还不存在，另外$refs也不是响应式的，因此也不应该试图用它在模板中做数据绑定。
* VUEX

## MVVM

Model、ViewModel和View，当model更新时，ViewModel会通过数据绑定更新到View；在View触发指令时，会通过ViewModel传递消息到Model。实现了View和Model的自动同步，不需要手动操作DOM元素来改变View的显示。

## v-if和v-show的区别

* v-if是条件渲染，动态添加删除DOM元素，只有在表达式为true时渲染，适合条件不太可能改变的情况。
* v-show始终会被渲染并且保留在DOM中，只是简单切换CSS属性，适合条件频繁切换的情况。

## VUEX

实现了单向数据流。

State:状态

Getter:根据state的值计算返回值

Mutation:同步更改state

Action:异步更改state

## vue-router相关

原理：通过改变URL，在不重新请求页面的情况下，更新页面视图，从而动态加载与销毁组件，简单的说就是，虽然地址栏的地址改变了，但是并不是一个全新的页面，而是之前的页面某些部分进行了修改。

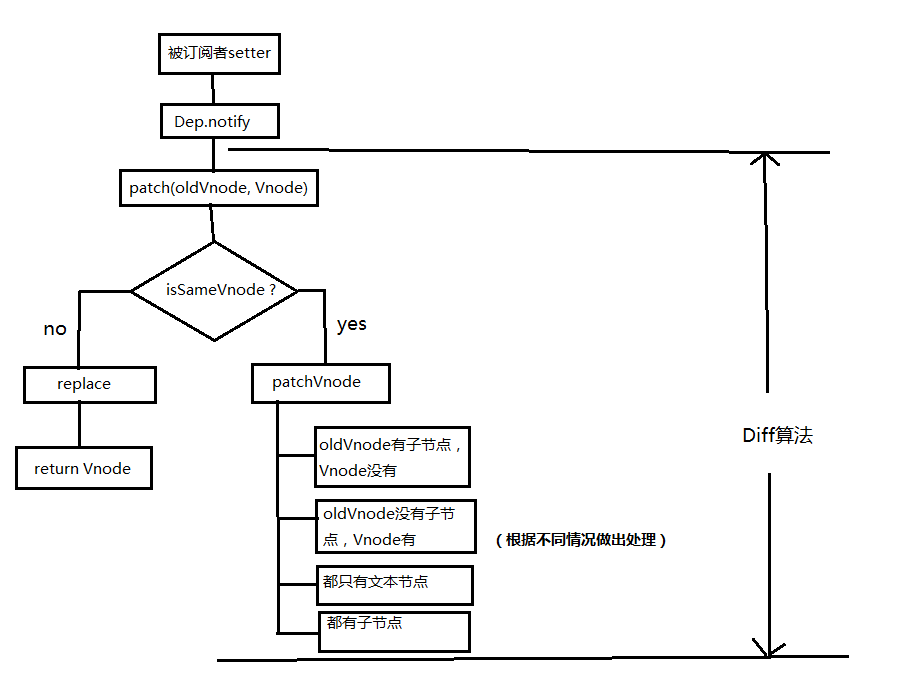
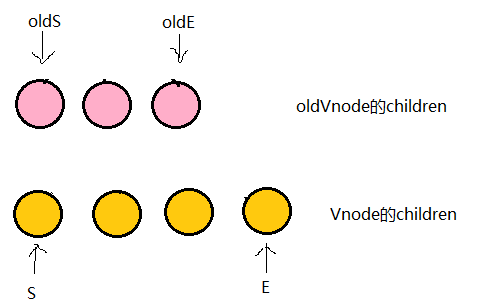
路由模式：

* hash模式：#号及后面的字符，不会向服务器端发出请求，所以不会重新加载页面，同时hash改变会触发hashchange事件。
* history模式：必须与后端设置路由相同。

$router和$route的区别：

$route可以访问当前路由的状态信息，而$router是一个实例，管理一组$route。

## diff算法

* virtual DOM 用js对象结构表示DOM树结构，然后用这个树构建一个真正的DOM树，插到文档中，当状态变更时，重新构造一颗新的对象树，然后将新的树和旧的树进行比较，记录差异，并把差异以打补丁的方式应用到真正的DOM树上来更新视图。
* 
* img
* 都有子节点的时候会执行updateChildren函数
* 
* 四种匹配方式，如果都不匹配就会看key值，如果有key值就根据key值操作。（使用key值可以更高效地更新虚拟DOM，index会改变）

## VUE模板编译原理

1. 将模板字符串循环解析转换成AST。
2. 对AST进行静态节点标记，用来做虚拟DOM的渲染优化。
3. 用AST生成render函数代码字符串。

## v-model原理

语法糖，用于表单数据的双向绑定，做了两个操作：

1. v-bind绑定一个value属性
2. v-on指令给当前元素绑定input事件

# 操作系统

## 进程与线程

* 进程：操作系统进行资源分配调度的单位。程序运行的载体。进程之间是独立的。
* 线程：进程的一个实体，是CPU调度分配的基本单位。（程序的运行依靠进程，进程的实际执行单元就是线程）
* 一个程序至少有一个进程，一个进程至少有一个线程，资源分配给进程，同一个进程下所有线程共享该进程的资源。

## 死锁

死锁是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。

产生的条件：

* 互斥条件：一个资源每次只能被一个线程使用；
* 请求与保持条件：一个线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：线程已获得的资源，在末使用完之前，不能强行剥夺；
* 循环等待条件：若干线程之间形成一种头尾相接的循环等待资源关系；

# 编译系统

1. 五个阶段：词法分析、语法分析、语义分析与中间代码生成、优化和目标代码生成（依赖于目标机）。
2. 文法：0型文法、1型文法（上下文有关）、2型文法（上下文无关）和3型文法（正规文法）。
3. 词法分析：确定的有穷自动机DFA和不确定的有穷自动机（NFA）。
4. 语法分析：自顶向下和自底向上。

# 数据结构

1. 数组：连续存储，下标访问元素，大小固定，只能存储一种数据类型。
2. 栈：栈顶操作，先进后出。空间由操作系统自动分配释放。
3. 队列：队头删除，队尾插入，先进先出。
4. 链表：非连续非顺序的存储结构，无需初始化容量，可以任意加减元素，占用空间大。
5. 树：二叉树、红黑树、B+树等。
6. 堆：由程序员分配释放，可以被看成一棵完全二叉树，可分为大顶堆和小顶堆。
7. 图：由节点的有穷集合和边的集合组成，无向图和有向图。

# 设计模式

## 观察者模式（发布订阅）

观察者模式建立了一种对象与对象之间的依赖关系，一个对象发生改变时将自动通知其他对象，其他对象将相应做出反应。所以发生改变的对象称为观察目标，而被通知的对象称为观察者，一个观察目标可以对应多个观察者，而且这些观察者之间没有相互联系，可以根据需要增加和删除观察者，使得系统更易于扩展。

function Subject() {  
 this.observers = {};  
 this.on = function(observer,handler) {  
 for(var key in onservers) {  
 if(key === observer) {  
 this.observers[key].push(handler);  
 }  
 }  
 }  
 this.off = function(observer,handler) {  
 for(var key in this.observers) {  
 if(key === observer) {  
 for(var i = 0; i < this.observers[key].length;i++) {  
 if(this.observers[key][i] === handler) {  
 this.observers[key].splice(i,1);  
 }  
 }  
 }  
 }  
 }  
 this.fire = function(observer) {  
 for(var key in this.observers) {  
 if(key === observer) {  
 this.observers[key].forEach(handler => handler.apply(this));  
 }  
 }  
 }  
}

## 单例模式

单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。

var SingleTon = function(){  
 var instance;  
 class CreateSingleTon {  
 constructor (name) {  
 if(instance) return instance;  
 this.name = name;  
 this.getName();  
 return instance = this;  
 }  
  
 getName() {  
 return this.name;  
 }  
 }  
 return CreateSingleTon;  
}();