# 从输入URL到页面加载完成发生了什么

1.DNS解析

2.TCP连接

3.发送HTTP请求

4.服务器处理请求并返回HTTP报文

5.浏览器解析渲染页面

具体过程：

1.DNS解析

DNS解析的过程就是寻找哪台机器上有你需要资源的过程。它实际上充当了一个翻译的角色，实现了网址到IP地址的转换。

解析过程：DNS解析是一个递归查询的过程。首先在本地域名服务器中查询IP地址，如果没有找到的情况下，本地域名服务器会向根域名服务器发送一个请求，如果根域名服务器也不存在该域名时，本地域名会向com顶级域名服务器发送一个请求，依次类推下去。直到最后本地域名服务器得到google的IP地址并把它缓存到本地，供下次查询使用。可以看出网址的解析是一个从右向左的过程: com -> google.com -> www.google.com

2.TCP连接

3.发送HTTP请求

发送HTTP请求的过程就是构建HTTP请求报文并通过TCP协议中发送到服务器指定端口(HTTP协议80/8080, HTTPS协议443)。HTTP请求报文是由三部分组成: 请求行, 请求报头和请求正文。

请求行：

Method Request-URL HTTP-Version CRLF

eg: GET index.html HTTP/1.1

常用的方法有: GET, POST, PUT, DELETE, OPTIONS, HEAD。

请求报头：

请求报头允许客户端向服务器传递请求的附加信息和客户端自身的信息。

常见的请求报头有: Accept, Accept-Charset, Accept-Encoding, Accept-Language, Content-Type, Authorization, Cache-Control, Connection, Cookie, User-Agent等。

Accept用于指定客户端用于接受哪些类型的信息，Accept-Encoding与Accept类似，它用于指定接受的编码方式。Connection设置为Keep-alive用于告诉客户端本次HTTP请求结束之后并不需要关闭TCP连接，这样可以使下次HTTP请求使用相同的TCP通道，节省TCP连接建立的时间。

请求正文：

当使用POST, PUT等方法时，通常需要客户端向服务器传递数据。这些数据就储存在请求正文中。在请求包头中有一些与请求正文相关的信息，例如: 现在的Web应用通常采用Rest架构，请求的数据格式一般为json。这时就需要设置Content-Type: application/json。

4.服务器处理请求并返回HTTP报文

后端从在固定的端口接收到TCP报文开始，这一部分对应于编程语言中的socket。它会对TCP连接进行处理，对HTTP协议进行解析，并按照报文格式进一步封装成HTTP Request对象，供上层使用。这一部分工作一般是由Web服务器去进行，我使用过的Web服务器有Tomcat, Jetty和Netty等等。

HTTP响应报文也是由三部分组成: 状态码, 响应报头和响应报文。

状态码：

状态码是由3位数组成，第一个数字定义了响应的类别，且有五种可能取值:

1xx：指示信息–表示请求已接收，继续处理。

2xx：成功–表示请求已被成功接收、理解、接受。

3xx：重定向–要完成请求必须进行更进一步的操作。

4xx：客户端错误–请求有语法错误或请求无法实现。

5xx：服务器端错误–服务器未能实现合法的请求。

平时遇到比较常见的状态码有:200, 204, 301, 302, 304, 400, 401, 403, 404, 422, 500

响应报头：

常见的响应报头字段有: Server, Connection...。

响应报文：

服务器返回给浏览器的文本信息，通常HTML, CSS, JS, 图片等文件就放在这一部分。

5.浏览器解析渲染页面

···解析HTML文件，构建DOM树，并行请求css/image/js

···css文件下载完成，开始构建css树

···css树构建结束后，和DOM一起生成渲染树（render tree）

···布局（layout）：计算出每个节点在屏幕中的位置

···绘制（painting）：通过显卡把页面画到屏幕上

# CSS

## 盒子模型

盒子模型包括：content+ border + padding + margin

标准盒子模型：width=内容的宽度（content）

低版本IE盒子模型：width=（content+border+padding）

**box-sizing属性？**

用来控制元素的盒子模型的解析模式，默认为content-box

context-box：W3C的标准盒子模型，设置元素的 height/width 属性指的是content部分的高/宽

border-box：IE传统盒子模型。设置元素的height/width属性指的是border + padding + content部分的高/宽

* 在 CSS 盒子模型 (Box Model) 规定了元素处理元素的几种方式： **内容、内边距、边框、外边距**。
* 在 CSS的**标准盒子模型**中，**width 和 height 指的是内容区域**的宽度和高度。增加内边距、边框和外边距不会影响内容区域的尺寸，但是会增加元素框的总尺寸。**IE盒子模型**中，**width 和 height 指的是内容区域+border+padding**的宽度和高度。

## 三栏布局

1、浮动布局

左栏左浮动，右栏右浮动，主体直接放后面，再设置一个左右的margin，就实现了自适应。

2、绝对定位

左右两栏采用绝对定位，分别固定于页面的左右两侧，中间的主体栏用左右margin值撑开距离。于是实现了三栏自适应布局。

3、flex布局

父元素设置display：flex；左右盒子定宽，中间盒子设置flex-grow：1；（放大比例，当宽度改变时，左右盒子不变，中间盒子自适应）

4、表格布局

父元素设置display：table；三个盒子设置display: table-cell;（使div作为表格单元显示），左右盒子设置固定宽度，中间盒子自动填满剩余区域

5、双飞翼布局

**双飞翼布局，**中间的主体要使用双层标签。外层div宽度100%显示，并且浮动（本例左浮动，下面所述依次为基础），内层div为真正的主体内容，含有左右210像素的margin值。左栏与右栏都是采用margin负值定位的，左栏左浮动，margin-left为-100%，由于前面的div宽度100%与浏览器，所以这里的-100%margin值正好使左栏div定位到了页面的左侧；右侧栏也是左浮动，其margin-left也是负值，大小为其本身的宽度即200像素。

6、圣杯布局

**圣杯布局**首先用一个外层div包裹主体和左右两栏，外层div设置左右padding:210px,主体和左右两栏左浮动，左栏与右栏都是采用margin负值定位的，左栏左浮动，margin-left为-100%，由于前面的div宽度100%与浏览器，所以这里的-100%margin值正好使左栏div定位到了主题的左侧，再相对定位position:relative;left:-210px;将左栏移动到外层的padding区域，以保证不遮挡主体区域；右侧栏也是左浮动，其margin-left也是负值，大小为其本身的宽度即200像素，同时也需要设置相对定位，position:relative;right:-210px;以保证不遮挡主体区域。

## CSS选择器的权重与优先规则

!important 的作用是提升优先级，换句话说。加了这句的样式的优先级是最高的（比内联样式的优先级还高)。

1. 第一等：代表行间样式，如: style=””，权值为1000。
2. 第四等：代表标签选择器和伪元素选择器，如div p，权值为1。
3. 第二等：代表ID选择器，如：#content，权值为100。
4. 第三等：代表类，伪类和属性选择器，如.content，权值为10。

## 垂直水平居中

宽高已知：（宽高固定）

1、定位 +负的margin （盒子宽高已知）

position: absolute;

left: 50%; top: 50%;

margin-left:-自身一半宽度;

margin-top: -自身一半高度;

负的margin的transform的区别：前者要知道宽度，后者不需要

宽高未知（不固定）：

2、table-cell布局

父级 display: table-cell; vertical-align: middle;

子级 margin: 0 auto;

3、定位 + transform ; 适用于 子盒子 宽高不定时； （这里是本人常用方法）

position:absolute;

<!--设置元素的定位位置，距离上、左都为50%-->

left:50%;top:50%;

<!--设置元素的相对于自身的偏移度为负50%(也就是元素自身尺寸的一半)-->

transform:translate(-50%,-50%); 注意这里启动了3D硬件加速哦 会增加耗电量的 （至于何是3D加速 请看浏览器进程与线程篇）

4、纯定位

position:absolute;

left:0; right:0; top:0; bottom:0;

margin:auto;

5、flex 布局

父级：

display: flex;

justify-content: center; 实现水平居中 align-items: center; 实现垂直居中

## rem

浏览器默认字体尺寸：16px

**px，em，rem的区别：**

**一、px**

px像素(Pixel)。相对长度单位。像素px是相对于显示器屏幕分辨率而言的。

**二、em**

em（font size of the element）是指相对于父元素的字体大小的单位

em做弹性布局的缺点在于牵一发而动全身，一旦某个节点的字体大小发生变化，那么其后代元素都得重新计算

**三、rem（root em）**

rem（font size of the root element）是指相对于根元素的字体大小的单位。

它计算的方式依赖于根元素，就是先给予html元素一个font-size，然后我们所有的rem就根据这个font-size来计算，如html{ font-size:100px;}，那么1rem = 100px；

(function(doc, win) {

var docEl = doc.documentElement, //取到的是根节点<html>

resizeEvt = 'orientationchange' in window ? 'orientationchange' : 'resize',

recalc = function() {

var clientWidth = docEl.clientWidth;

if(!clientWidth) return;

if(clientWidth >= 768) {

docEl.style.fontSize = '100px';

} else {

docEl.style.fontSize = 100 \* (clientWidth / 768) + 'px';

}

};

if(!doc.addEventListener) return;

win.addEventListener(resizeEvt, recalc, false);

doc.addEventListener('DOMContentLoaded', recalc, false);

})(document, window);

rem就是根元素（即：html）的字体大小。html中的所有标签样式凡是涉及到尺寸的（如： height,width,padding,margin,font-size。甚至，left,top等）你都可以放心大胆的用rem作单位。

**多屏适配布局问题**

移动端布局，为了适配各种大屏手机，目前最好用的方案莫过于使用相对单位rem。

基于rem的原理，我们要做的就是: **针对不同手机屏幕尺寸动态的改变根节点html的font-size大小(基准值)**。

## 如何隐藏一个元素

Opacity：0

Display：none

Visibility：hidden

position: absolute;

   top: -9999px;

   left: -9999px;

## flex布局

一，适用情况

适用于在子元素的尺寸未知或者动态的情况下，对子元素的对齐方式、排列方式以及排序顺序进行控制展示

一般情况下，当我们对一部分内容进行布局的时候，会使用盒模型。常见的布局手段即是结合padding与margin设置元素的display、float和position。但是这三种方法在设置元素位置的时候，有时会相对死板并且使css语句过多。

二、特性如下：

可以在垂直或者水平方向以任意一种对齐方式进行样式布局

可以在单一轴线上布局也可以在多行内布局

随可用空间的变化进行响应式布局（能够调整其子元素的宽度或者高度以使其能在不同分辨率的屏幕下能用最好的方式去填充可用空间）

三、弹性容器flex container的属性

1、flex-direction：主轴的方向（即项目的排列方向）

row（默认值） 主轴为水平方向，起点在左端。

row-reverse 主轴为水平方向，起点在右端。

column 主轴为垂直方向，起点在上沿。

column-reverse 主轴为垂直方向，起点在下沿。

2、flex-wrap：如果一条轴线排不下，如何换行

nowrap（默认）：不换行

wrap：换行，第一行在上方

wrap-reverse：换行，第一行在下方。

3、flex-flow：flex-direction属性和flex-wrap属性的简写

默认值为row nowrap

4、justify-content ：定义了项目在主轴上的对齐方式

它可能取5个值，具体对齐方式与轴的方向有关。下面假设主轴为从左到右。

flex-start（默认值） 左对齐

flex-end 右对齐

center 居中

space-between 两端对齐，项目之间的间隔都相等。

space-around 每个项目两侧的间隔相等。所以，项目之间的间隔比项目与边框的间隔大一倍。

5、align-items：定义项目在交叉轴上如何对齐

它可能取5个值。具体的对齐方式与交叉轴的方向有关，下面假设交叉轴从上到下。

flex-start 交叉轴的起点对齐。

flex-end 交叉轴的终点对齐。

center 交叉轴的中点对齐。

baseline 项目的第一行文字的基线对齐。

stretch（默认值） 如果项目未设置高度或设为auto，将占满整个容器的高度。（可实现等高）

6、align-content：多根轴线在垂直方向上的对齐方式，一根轴该属性无效

flex-start 与交叉轴的起点对齐。

flex-end 与交叉轴的终点对齐。

center 与交叉轴的中点对齐。

space-between 与交叉轴两端对齐，轴线之间的间隔平均分布。

space-around 每根轴线两侧的间隔都相等。所以，轴线之间的间隔比轴线与边框的间隔大一倍。

stretch（默认值） 轴线占满整个交叉轴。

四、弹性子元素flex item的属性

1、order ：项目的排列顺序(数字) 数值越小，排列越靠前，默认为0 类似z-index

2、flex-grow：放大比例，默认为0-如果存在剩余空间也不放大

如果所有项目的flex-grow属性都为1，则它们将等分剩余空间（如果有的话）。

如果一个项目的flex-grow属性为2，其他项目都为1，则前者占据的剩余空间将比其他项多一倍。

3、flex-shrink：缩小比例，默认为1-如果空间不足项目将缩小

如果所有项目的flex-shrink属性都为1，当空间不足时，都将等比例缩小。

如果一个项目的flex-shrink属性为0，其他项目都为1，则空间不足时，前者不缩小。

负值对该属性无效。

4、flex-basis：在分配多余空间之前，项目占据的主轴空间（main size）。浏览器根据这个属性，计算主轴是否有多余空间。它的默认值为auto，即项目的本来大小

如果设为跟width或height属性一样的值（比如350px），则项目将占据固定空间。

flex-basis:flex-basis可以理解为我们给子元素设置的宽度。

默认值是auto,宽度设置为auto时，盒子的宽度取决你们元素的宽度。

5、flex：flex-grow, flex-shrink 和 flex-basis的简写，默认值为0 1 auto。后两个属性可选

6、align-self：允许单个项目有与其他项目不一样的对齐方式，可覆盖align-items属性。默认值为auto，表示继承父元素的align-items属性，如果没有父元素，则等同于stretch。

五，flex：1代表什么意思？

当 flex 取值为一个非负数字，则该数字为 flex-grow 值，flex-shrink 取 1，flex-basis 取 0%，

当 flex 取值为一个长度或百分比，则视为 flex-basis 值，flex-grow 取 1，flex-shrink 取 1

## BFC

BFC(Block formatting context)直译为"块级格式化上下文"。它**是一个独立的渲染区域**，只有**Block-level box**参与（在下面有解释）， 它规定了内部的Block-level Box如何布局，并且与这个区域外部毫不相干。

定位方案：

1. 内部的Box会在垂直方向上一个接一个放置。
2. Box垂直方向的距离由margin决定，属于同一个BFC的两个相邻Box的margin会发生重叠。
3. 每个元素的margin box 的左边，与包含块border box的左边相接触。
4. BFC的区域不会与float box重叠。
5. BFC是页面上的一个隔离的独立容器，容器里面的子元素不会影响到外面的元素。
6. 计算BFC的高度时，浮动元素也会参与计算。

满足下列条件之一就可触发BFC

1. 根元素，即html
2. float的值不为none（默认）
3. overflow的值不为visible（默认）
4. display的值为inline-block、table-cell、table-caption
5. position的值为absolute或fixed

**BFC有哪些作用：**

1. 自适应两栏布局
2. 可以阻止元素被浮动元素覆盖
3. 可以包含浮动元素——清除内部浮动
4. 分属于不同的BFC时可以阻止margin重叠

# BOM

## 浏览器渲染过程

* 浏览器的渲染过程：
  + 解析HTML构建 DOM(DOM树)，并行请求 css/image/js
  + CSS 文件下载完成，开始构建 CSSOM(CSS树)
  + CSSOM 构建结束后，和 DOM 一起生成 Render Tree(渲染树)
  + 布局(Layout)：计算出每个节点在屏幕中的位置
  + 显示(Painting)：通过显卡把页面画到屏幕上
* DOM树 和 渲染树 的区别：
  + DOM树与HTML标签一一对应，包括head和隐藏元素
  + 渲染树不包括head和隐藏元素，大段文本的每一个行都是独立节点，每一个节点都有对应的css属性

## 重绘和回流

* 重绘：当渲染树中的元素外观（如：颜色）发生改变，不影响布局时，产生重绘
* 回流：当渲染树中的元素的布局（如：尺寸、位置、隐藏/状态状态）发生改变时，产生重绘回流
* 注意：JS获取Layout属性值（如：offsetLeft、scrollTop、getComputedStyle等）也会引起回流。因为浏览器需要通过回流计算最新值
* 回流必将引起重绘，而重绘不一定会引起回流
* display：none 不显示对应的元素，在文档布局中不再分配空间（回流+重绘）
* visibility：hidden 隐藏对应元素，在文档布局中仍保留原来的空间（重绘）

**如何最小化重绘(repaint)和回流(reflow)？**

* 需要要对元素进行复杂的操作时，可以先隐藏(display:"none")，操作完成后再显示
* 需要创建多个DOM节点时，使用DocumentFragment创建完后一次性的加入document
* 缓存Layout属性值，如：var left = elem.offsetLeft; 这样，多次使用 left 只产生一次回流
* 尽量避免用table布局（table元素一旦触发回流就会导致table里所有的其它元素回流）
* 避免使用css表达式(expression)，因为每次调用都会重新计算值（包括加载页面）
* 尽量使用 css 属性简写，如：用 border 代替 border-width, border-style, border-color
* 批量修改元素样式：elem.className 和 elem.style.cssText 代替 elem.style.xxx

## cookie相关

**cookie** : 用于客户端数据的存储，在客户端请求服务端，使用response向客户端浏览器发送一个cookie,当客户端再次访问该网站的时候，浏览器会把请求的地址和cookie数据一起提交给服务器，服务器检查该cookie，用来判断客户的登录状态，存储大小为4KB。

//js读取

document.cookie;

//js写入

document.cookie = 'name=value';

//js删除

找到相对name对应的值，将存储时间设置为过去时间： expires=过去时间；

//设置作用域 domain设置有效作用域 path限制有效路径

cookie.domain = 'www.baidu.com';

//www主机下所有位置可以获取

cookie.domain = 'baidu.com'；

//baidu.com域下所有主机的所有位置都可以获取

cookie.domain = 'www.baidu.com/test';

cookie.path = '/test'; //主机的test下可以获取

Web Storage 其中包含localStorage和sessionStorage两种机制，这两种机制是通过Window.sessionStorage 和 Window.localStorage ，调用其中任意对象会创建storage对象，通过对storage对象设置，获取和删除进行控制。两者存储的形式只能是字符串类型的键值对。

//js读取

localStorage.getItem('name');

sessionStorage.getItem('name');

//js写入

localStorage.setItem('name','value');

sessionStorage.setItem('name','value');

//js删除

localStorage.removeItem('name');

sessionStorage.removeItem('name');

//删除所有key/value

localStorage.clear();

sessionStorage.clear();

localStorage：永久存储，在浏览器关闭后，重新打开后数据仍然存在，需要手动删除，不然一直存在。不同浏览器中支持存储的大小不同，一般为5M。

sessionStorage：在浏览器页面会话期间可用（包换页面重新加载和恢复），会话结束就被删除。

**描述 cookies、sessionStorage 和 localStorage 的区别？**

* 与服务器交互：
  + cookie 是网站为了标示用户身份而储存在用户本地终端上的数据（通常经过加密）
  + cookie 始终会在同源 http 请求头中携带（即使不需要），在浏览器和服务器间来回传递
  + sessionStorage 和 localStorage 不会自动把数据发给服务器，仅在本地保存
* 存储大小：
* cookie 数据根据不同浏览器限制，大小一般不能超过 4k
* sessionStorage 和 localStorage 虽然也有存储大小的限制，但比cookie大得多，可以达到5M或更大
* 有期时间：
  + localStorage 存储持久数据，浏览器关闭后数据不丢失除非主动删除数据
  + sessionStorage 数据在当前浏览器窗口关闭后自动删除
  + cookie 设置的cookie过期时间之前一直有效，与浏览器是否关闭无关

## 跨域

一、同源策略

同源策略限制不同源之间执行特定操作。

其中，同源是指：协议、域名、端口相同。

特定操作是指：①读取cookie、localstorage、indexDB

②获取DOM元素

③发送ajax请求

它的含义是指，A网页设置的 Cookie，B网页不能打开，除非这两个网页"同源"。

举例来说，http://www.example.com/dir/page.html这个网址，协议是http://，域名是www.example.com，端口是80（默认端口可以省略）。它的同源情况如下。

http://www.example.com/dir2/other.html：同源

http://example.com/dir/other.html：不同源（域名不同）

http://v2.www.example.com/dir/other.html：不同源（域名不同）

http://www.example.com:81/dir/other.html：不同源（端口不同）

同源策略的目的，是为了保证用户信息的安全，防止恶意的网站窃取数据。"同源策略"是必需的，否则 Cookie 可以共享，互联网就毫无安全可言了。

虽然这些限制是必要的，但是有时很不方便，合理的用途也受到影响。下面，将详细介绍，如何规避上面三种限制。

二、跨域方式

1、document.domain （cookie 跨域）

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。

举例来说，A网页是http://w1.example.com/a.html，B网页是http://w2.example.com/b.html，那么只要设置相同的document.domain，两个网页就可以共享Cookie。

document.domain = 'example.com';

现在，A网页通过脚本设置一个 Cookie。

document.cookie = "test1=hello";

B网页就可以读到这个 Cookie。

var allCookie = document.cookie;

注意，这种方法只适用于 Cookie 和 iframe 窗口，LocalStorage 和 IndexDB 无法通过这种方法，规避同源政策，而要使用下文介绍的PostMessage API。

另外，服务器也可以在设置Cookie的时候，指定Cookie的所属域名为一级域名，比 如.example.com。

Set-Cookie: key=value; domain=.example.com; path=/

这样的话，二级域名和三级域名不用做任何设置，都可以读取这个Cookie。

2、window.postMessage （iframe跨域）

这个API为window对象新增了一个window.postMessage方法，允许跨窗口通信，不论这两个窗口是否同源。

主窗口通过postMessageAPI向异域的窗口发送数据：

// window\_1 域名为 http://winodow1.com:8080

window.postMessage("Hi, How are you!", "http://window2.com:8080")

postMessage函数接收两个参数，第一个为要发送的信息（可以是任何JavaScript类型数据，但部分浏览器只支持字符串格式），第二个为信息发送的目标地址。也可以设为\*，表示不限制域名，向所有窗口发送。

子窗口向父窗口发送消息的写法类似。

window.opener.postMessage('Nice to see you', 'http://aaa.com');

父窗口和子窗口都可以通过message事件，监听对方的消息。

window.addEventListener('message', function(e) {

console.log(e.data);

},false);

message事件的事件对象event，提供以下三个属性。

event.source：发送消息的窗口

event.origin: 消息发向的网址

event.data: 消息内容

event.origin属性可以过滤不是发给本窗口的消息。

window.addEventListener('message', receiveMessage);

function receiveMessage(event) {

if (event.origin !== 'http://aaa.com') return;

if (event.data === 'Hello World') {

event.source.postMessage('Hello', event.origin);

} else {

console.log(event.data);

}

}

3、ajax跨域方法

同源政策规定，AJAX请求只能发给同源的网址，否则就报错。

除了架设服务器代理（浏览器请求同源服务器，再由后者请求外部服务），有三种方法规避这个限制。

JSONP

CORS

WebSocket

JSONP

原理：利用浏览器对src属性的资源引用没有同源限制，所有具有src属性的HTML标签都可以实现跨域，包括<script>、<img>、<link>

①首先在客户端注册一个callback，然后把callback的名字传给服务器。

②服务器先生成json数据，将json数据直接以参数的方式，放置到callback中，这样就生成了一段js语法的文档，返回给客户端。

③客户端浏览器，解析script标签，并执行返回的js代码，此时数据作为参数，传入到了客户端预先定义好的callback函数里。（动态执行回调函数)

　　JSONP只支持GET请求而不支持POST等其它类型的HTTP请求,它只支持跨域HTTP请求这种情况，不能解决不同域的两个页面之间如何进行JavaScript调用的问题，JSONP的优势在于支持老式浏览器，弊端也比较明显：需要客户端和服务端定制进行开发，服务端返回的数据不能是标准的Json数据，而是callback包裹的数据。

function addScriptTag(src) {

var script = document.createElement('script');

script.setAttribute("type","text/javascript");

script.src = src;

document.body.appendChild(script);

}

window.onload = function () {

addScriptTag('http://example.com/ip?callback=foo');

}

function foo(data) {

console.log('Your public IP address is: ' + data.ip);

};

CORS

CORS是现代浏览器支持跨域资源请求的一种方式，全称是"跨域资源共享"（Cross-origin resource sharing），当使用XMLHttpRequest发送请求时，浏览器发现该请求不符合同源策略，会给该请求加一个请求头：Origin，后台进行一系列处理，如果确定接受请求则在返回结果中加入一个响应头：Access-Control-Allow-Origin;浏览器判断该相应头中是否包含Origin的值，如果有则浏览器会处理响应，我们就可以拿到响应数据，如果不包含浏览器直接驳回，这时我们无法拿到响应数据。

　　CORS与JSONP的使用目的相同，但是比JSONP更强大，CORS支持所有的浏览器请求类型，承载的请求数据量更大，开放更简洁，服务端只需要将处理后的数据直接返回，不需要再特殊处理。

浏览器将CORS请求分成两类：简单请求（simple request）和非简单请求（not-so-simple request）。

简单请求：

①请求方法只属于HEAD，GET，POST请求的其中一种；

②HTTP的头信息只限于以下字段：

Accept

Accept-Language

Content-Language

Last-Event-ID

Content-Type（只能为application/x-www-form-urlencoded，multipart/form-data和text/plain其中一种）

凡是不同时满足上面两个条件，就属于非简单请求。浏览器对这两种请求的处理，是不一样的。

当浏览器检测到一个简单的跨域AJAX请求，就自动在头信息之中，添加一个Origin字段，Origin字段用来说明，本次请求来自哪个源（协议 + 域名 + 端口）。服务器根据这个值，决定是否同意这次请求。

如果的确是服务端认可的域，那么服务端会在响应报文中添加如下字段：

（必须）：该字段用来告知浏览器服务端接受的能够发送跨域AJAX Access-Control-Allow-Origin请求的域，它的值要么是该次AJAX请求报头中由浏览器自动添加的Origin值，要么还可以是一个\*号，表示可以接受任意的域名请求；

Access-Control-Allow-Credentials（可选）：该字段用来告知浏览器是否允许客户端向服务端发送Cookie。默认情况下，CORS规范会阻止跨域AJAX向服务端发送Cookie，因此该字段默认值为false，当你显式的将该字段值设置为true时，则表示允许此次跨域AJAX向服务端发送Cookie。

Access-Control-Expose-Headers（可选）：该字段用来向客户端暴露可获取的响应头；

非简单请求：

与简单AJAX跨域请求不同，“复杂“的AJAX跨域请求一共会发送两次HTTP请求，其中第一次为”查询请求“，第二次才是我们正式的”AJAX跨域请求“。

非简单请求是那种对服务器有特殊要求的请求，比如请求方法是PUT或DELETE，或者Content-Type字段的类型是application/json。

非简单请求的CORS请求，会在正式通信之前，增加一次HTTP查询请求，称为"预检"请求（preflight）。

浏览器先询问服务器，当前网页所在的域名是否在服务器的许可名单之中，以及可以使用哪些HTTP动词和头信息字段。只有得到肯定答复，浏览器才会发出正式的XMLHttpRequest请求，否则就报错。

"预检"请求用的请求方法是OPTIONS，表示这个请求是用来询问的。头信息里面，关键字段是Origin，表示请求来自哪个源。

除了Origin字段，"预检"请求的头信息包括两个特殊字段。

（1）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，上例是PUT。

（2）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，

预检请求的回应

服务器收到"预检"请求以后，检查了Origin、Access-Control-Request-Method和Access-Control-Request-Headers字段以后，确认允许跨源请求，就可以做出回应。

HTTP回应中，

首先，Access-Control-Allow-Origin字段向浏览器说明了发起AJAX请求的域是被服务器认可的（注意这个字段的值也可以为一个“\*”号）；

其次，Access-Control-Allow-Methods字段向浏览器说明了服务器接收跨域AJAX的请求方式；

最后，Access-Control-Allow-Headers字段向浏览器说明了服务器允许跨域AJAX额外发送的报头信息；

当浏览器收到服务端这样的表示同意请求的响应后，就会正常发送接下来的跨域AJAX请求，而服务器也会正常的回应。值的一提的是，在服务端与客户端整个跨域AJAX请求的交互中，Access-Control-Allow-Origin头信息自始至终都是必须携带的。

而当服务器在检查“查询请求”后，如果不同该请求，则会返回一个正常的HTTP响应，报文中包含任何与CORS规范有关的报头字段，此时，浏览器就会心领神会的明白服务器拒绝接收发出的跨域AJAX请求，因此会返回一个错误状态（可以被XML对象实例使用onerror回调函数捕获）并在控制台打印一条错误信息：

WebSocket

WebSocket是一种通信协议，使用ws://（非加密）和wss://（加密）作为协议前缀。该协议不实行同源政策，只要服务器支持，就可以通过它进行跨源通信。

下面是一个例子，浏览器发出的WebSocket请求的头信息（摘自维基百科）。

GET /chat HTTP/1.1

Host: server.example.com

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==

Sec-WebSocket-Protocol: chat, superchat

Sec-WebSocket-Version: 13

Origin: http://example.com

上面代码中，有一个字段是Origin，表示该请求的请求源（origin），即发自哪个域名。

正是因为有了Origin这个字段，所以WebSocket才没有实行同源政策。因为服务器可以根据这个字段，判断是否许可本次通信。如果该域名在白名单内，服务器就会做出如下回应。

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=

Sec-WebSocket-Protocol: chat

WebSocket 是 HTML5中的一种新的协议。它实现了浏览器与服务器全双工通信，能更好的节省服务器资源和带宽并达到实时通讯，它建立在 TCP 之上，同 HTTP 一样通过 TCP 来传输数据，但是它和 HTTP 最大不同是：

WebSocket 是一种双向通信协议，在建立连接后，WebSocket 服务器和 浏览器都能主动的向对方发送或接收数据，就像 Socket 一样；

WebSocket 需要类似 TCP 的客户端和服务器端通过握手连接，连接成功后才能相互通信。

WebSocket 协议本质上是一个基于 TCP 的协议。

为了建立一个 WebSocket 连接，客户端浏览器首先要向服务器发起一个 HTTP 请求，这个请求和通常的 HTTP 请求不同，包含了一些附加头信息，其中附加头信息"Upgrade: WebSocket"表明这是一个申请协议升级的 HTTP 请求，服务器端解析这些附加的头信息然后产生应答信息返回给客户端，客户端和服务器端的 WebSocket 连接就建立起来了，双方就可以通过这个连接通道自由的传递信息，并且这个连接会持续存在直到客户端或者服务器端的某一方主动的关闭连接。

补充：

Websocket是一个持久化的协议，**而HTTP是不支持持久连接的**

Websocket其实是一个新协议，跟HTTP协议基本没有关系，只是为了兼容现有浏览器的握手规范而已，也就是说它是HTTP协议上的一种补充

HTTP 协议有一个缺陷：通信只能由客户端发起。

举例来说，我们想了解今天的天气，只能是客户端向服务器发出请求，服务器返回查询结果。HTTP 协议做不到服务器主动向客户端推送信息。

这种单向请求的特点，注定了如果服务器有连续的状态变化，客户端要获知就非常麻烦。我们只能使用轮询：每隔一段时候，就发出一个询问，了解服务器有没有新的信息。最典型的场景就是聊天室。

轮询的效率低，非常浪费资源（因为必须不停连接，或者 HTTP 连接始终打开）。因此，工程师们一直在思考，有没有更好的方法。WebSocket 就是这样发明的。

## 垃圾回收机制

**哪些操作会造成内存泄漏？**

* JavaScript 内存泄露指对象在不需要使用它时仍然存在，导致占用的内存不能使用或回收
* 意外的全局变量 （“使用use strict”）
* 闭包函数(Closures) （用完之后用null解除引用）
* 循环引用(两个对象相互引用)
* 控制台日志(console.log)
* 移除存在绑定事件的DOM元素(IE)

**javascript垃圾回收方法**

* **标记清除（mark and sweep）**
* 这是JavaScript最常见的垃圾回收方式，当变量进入执行环境的时候，比如函数中声明一个变量，垃圾回收器将其标记为“进入环境”，当变量离开环境的时候（函数执行结束）将其标记为“离开环境”
* 垃圾回收器会在运行的时候给存储在内存中的所有变量加上标记，然后去掉环境中的变量以及被环境中变量所引用的变量（闭包），在这些完成之后仍存在标记的就是要删除的变量了

**引用计数**(reference counting)

在低版本IE中经常会出现内存泄露，很多时候就是因为其采用引用计数方式进行垃圾回收。引用计数的策略是跟踪记录每个值被使用的次数，当声明了一个 变量并将一个引用类型赋值给该变量的时候这个值的引用次数就加1，如果该变量的值变成了另外一个，则这个值得引用次数减1，当这个值的引用次数变为0的时 候，说明没有变量在使用，这个值没法被访问了，因此可以将其占用的空间回收，这样垃圾回收器会在运行的时候清理掉引用次数为0的值占用的空间

# DOM

## 事件和事件流

**DOM事件流**

“DOM2级事件”规定的事件流包括三个阶段：事件捕获阶段、处于目标阶段和事件冒泡阶段。首先发生的是事件捕获，然后处于目标阶段，最后才事件冒泡。依然以上面的代码为例子，单击div时，首先document获得点击事件，依次到<html>、<body>，之后事件捕获结束。事件捕获的意义在于，能够在目标获得点击事件之前截获事件，并对其作出相应的处理。处于目标阶段时，div能够执行绑定的事件处理程序，之后再到事件冒泡阶段。

**事件冒泡**

IE在处理上述事件时候，是由事件开始最具体的元素（文档中嵌套层次最深的那个节点）接收，然后逐级向上传播到根节点。这就是事件冒泡。

阻止冒泡：

标准浏览器：event.stopPropagation()

低版本IE： event.cancelBubble = true;

**事件捕获**

Netscape Communicator团队提出了一种与IE完全相反的一种解决办法，接收事件的顺序为根节点到具体的节点，这种方法就是事件捕获。也就是在上面的这个例子中，接收点击事件的顺序为document、<html>、<body>、<div>。

**事件处理程序**

**事件绑定**

要想让 JavaScript 对用户的操作作出响应，首先要对 DOM 元素绑定事件处理函数。所谓事件处理函数，就是处理用户操作的函数，不同的操作对应不同的名称。

在JavaScript中，有三种常用的绑定事件的方法：

* 在DOM元素中直接绑定；
* 在JavaScript代码中绑定；
* 绑定事件监听函数。

**在DOM中直接绑定事件**

我们可以在DOM元素上绑定onclick、onmouseover、onmouseout、onmousedown、onmouseup、ondblclick、onkeydown、onkeypress、onkeyup等。

**在JavaScript代码中绑定事件**

在JavaScript代码中（即 script 标签内）绑定事件可以使JavaScript代码与HTML标签分离，文档结构清晰，便于管理和开发。

**使用事件监听绑定事件**

绑定事件的另一种方法是用 addEventListener() 或 attachEvent() 来绑定事件监听函数。下面详细介绍，事件监听。

**事件监听**

关于事件监听，W3C规范中定义了3个事件阶段，依次是捕获阶段、目标阶段、冒泡阶段。

起初Netscape制定了JavaScript的一套事件驱动机制（即事件捕获）。随即IE也推出了自己的一套事件驱动机制（即事件冒泡）。最后W3C规范了两种事件机制，分为捕获阶段、目标阶段、冒泡阶段。IE8以前IE一直坚持自己的事件机制（前端人员一直头痛的兼容性问题），IE9以后IE也支持了W3C规范。

**W3C规范**

语法：

element.addEventListener(event, function, useCapture)

event : （必需）事件名，支持所有 [DOM事件](http://www.runoob.com/jsref/dom-obj-event.html) 。

function：（必需）指定要事件触发时执行的函数。

useCapture：（可选）指定事件是否在捕获或冒泡阶段执行。true，捕获。false，冒泡。默认false。

**IE标准**

语法：

element.attachEvent(event, function)

event：（必需）事件类型。需加“on“，例如：onclick。

function：（必需）指定要事件触发时执行的函数。

**事件监听的优点**

1、可以绑定多个事件。

常规的事件绑定只执行最后绑定的事件。标准浏览器的事件监听可以按顺序执行所有事件，低版本IE的事件监听按倒序执行所有事件

2、可以解除相应的绑定

removeEventListener() 、 detachEvent()

**事件委托**

[**https://www.cnblogs.com/liugang-vip/p/5616484.html**](https://www.cnblogs.com/liugang-vip/p/5616484.html)

在传统的事件处理中，你按照需要为每一个元素添加或者是删除事件处理器。然而，每个函数都是对象，都会占用内存；内存中的对象越多，性能越差。

事件委托就是利用冒泡的原理，把事件加到父元素或祖先元素上，等待事件从它的子级元素里冒泡上来，触发执行效果。

**事件委托优点**

1、提高JavaScript性能。事件委托可以显著的提高事件的处理速度，减少内存的占用

**传统写法**

<ul id="list">

<li id="item1" >item1</li>

<li id="item2" >item2</li>

<li id="item3" >item3</li>

</ul>

item1.onclick = function(){

alert("hello item1");

}

item2.onclick = function(){

alert("hello item2");

}

item3.onclick = function(){

alert("hello item3");

}

**事件委托**

document.addEventListener("click",function(event){

var target = event.target;

if(target == item1){

alert("hello item1");

}else if(target == item2){

alert("hello item2");

}else if(target == item3){

alert("hello item3");

}

})

ulEl.addEventListener('click', function(e){ var target = event.target || event.srcElement; if(!!target && target.nodeName.toUpperCase() === "LI"){ console.log(target.innerHTML); } }, false);

# JavaScript

## ES6语法

### 箭头函数

* 代替匿名函数表达式，语法更加简洁
* 注意点：没有this，必须通过查找作用域链来决定其值，如果箭头函数被非箭头函数包含，则this绑定的是最近一层非箭头函数的this，否则，this的值会被设为undefined，并且在函数生命周期内this值不可更改，
* 不能通过new关键字调用，否则会抛出错误（因为箭头函数没有[[construct]]方法，所以不能被用作构造函数，也没有原型）
* 不支持arguments对象

### Promise

* Promise 就是一个对象，用来表示并传递异步操作的最终结果
* Promise 最主要的交互方式：将回调函数传入 then 方法来获得最终结果或出错原因
* Promise 代码书写上的表现：以“链式调用”代替回调函数层层嵌套（回调地狱）
  + pending: 初始状态, 非 fulfilled 或 rejected.
  + fulfilled: 成功的操作.
  + rejected: 失败的操作.
* 构造一个 Promise，最基本的用法如下：

var promise = new Promise(function(resolve, reject) { if (...) { // succeed resolve(result); } else { // fails reject(Error(errMessage)); } });

* Promise 实例拥有 then 方法（具有 then 方法的对象，通常被称为thenable）。它的使用方法如下：

promise.then(onFulfilled, onRejected)

* 接收两个函数作为参数，一个在 fulfilled 的时候被调用，一个在rejected的时候被调用，接收参数就是 future，onFulfilled 对应 resolve, onRejected 对应 reject
* <https://segmentfault.com/a/1190000011652907>
* <https://blog.csdn.net/ligang2585116/article/details/70843713>promise，async，generater

### class类

通过class关键字，可以定义类。

//定义类

class Point { constructor(x, y) { this.x = x; this.y = y; } toString() { return '(' + this.x + ', ' + this.y + ')'; } }

在类中通过constructor方法名来定义构造函数，实例中的私有属性，不应该出现在原型上，所以应该在构造函数方法中创建所有私有属性

Point类除了构造方法，还定义了一个toString方法。注意，定义“类”的方法的时候，前面不需要加上function这个关键字，直接把函数定义放进去了就可以了。另外，方法之间不需要逗号分隔，加了会报错。

class Point { // ... } typeof Point // "function" Point === Point.prototype.constructor // true

上面代码表明，类的数据类型就是函数，类本身就指向构造函数。类的所有方法都定义在类的prototype属性上面。

由于类的方法都定义在prototype对象上面，所以类的新方法可以添加在prototype对象上面。Object.assign方法可以很方便地一次向类添加多个方法。

class的继承

Class 可以通过extends关键字实现继承，这比 ES5 的通过修改原型链实现继承，要清晰和方便很多。

class Point { } class ColorPoint extends Point { }

上面代码定义了一个ColorPoint类，该类通过extends关键字，继承了Point类的所有属性和方法。

super关键字

super这个关键字，既可以当作函数使用，也可以当作对象使用。在这两种情况下，它的用法完全不同。

第一种情况，super作为函数调用时，代表父类的构造函数。ES6 要求，子类的构造函数必须执行一次super函数。

class A {} class B extends A { constructor() { super(); } }

上面代码中，子类B的构造函数之中的super()，代表调用父类的构造函数。这是必须的，否则 JavaScript 引擎会报错。

注意，super虽然代表了父类A的构造函数，但是返回的是子类B的实例，即super内部的this指的是B，因此super()在这里相当于A.prototype.constructor.call(this)。

第二种情况，super作为对象时，在普通方法中，指向父类的原型对象；在静态方法中，指向父类。

class A { p() { return 2; } } class B extends A { constructor() { super(); console.log(super.p()); // 2 } } let b = new B();

上面代码中，子类B当中的super.p()，就是将super当作一个对象使用。这时，super在普通方法之中，指向A.prototype，所以super.p()就相当于A.prototype.p()。

这里需要注意，由于super指向父类的原型对象，所以定义在父类实例上的方法或属性，是无法通过super调用的。

，通过super调用父类的方法时，方法内部的this指向子类。

## 基础

### 数组

、、创建数组

var arr = new Array() 、、构造函数

var arr = [ ]; 、、数组字面量

、、检测数组

arr.instanceof Array 、、返回布尔值

Array.isArray(arr) 、、同上

、、转换方法

arr.toString(); 、、返回字符串

arr.valueOf(); 、、返回数组本身

arr.toLocaleString(); 、、返回字符串

arr.join(‘…’); 、、用指定方式连成字符串，不传为逗号

、、添加删除方法

arr.push(); 、、向后添加，返回添加后的长度

arr.pop(); 、、删除最后一项，返回移除的项

arr.unshift(); 、、向前添加，返回添加后的长度

arr.shift(); 、、删除第一项，返回移除的项

、、重排序方法

arr.reverse(); 、、倒序

arr.sort(); 、、不传参时，按比较字符串的方法从小到大排序，可接收一个比较函数做参数

、、操作方法

arr1.concat(arr2); 、、返回连接后的新数组

arr.slice(); 、、截取数组，返回一个新数组

1个参数时，从参数位置到数组末尾的所有项

2个参数时，起始和结束位置之间的项（不包括结束项）

arr.splice(); 、、删除、插入、替换，返回一个新数组，里面是被删除的项

参数：起始位置 | 要删除的个数 | 要插入的项

eg：（2,1）删除

（2,0，”red“）插入

（2,1，“red”）替换

、、位置方法

arr.indexOf() 、、返回要找的项在数组中的位置，从前往后找，找不到返回-1

arr.LastIndexOf() 、、同上，从后往前找，返回索引值

、、迭代方法 （对数组中的每一项运行给定函数）

arr.every( fn(item，index，array) ) 、、如果该函数对每一项都返回true，则返回true

arr.some( fn() ) 、、如果该函数对任一项返回true，则返回true

arr.filter( fn() ) 、、返回该函数会返回true的项组成的数组

arr.forEach( fn() ) 、、没有返回值，只是对数组中的每一项运行传入的函数

arr.map( fn() ) 、、返回每次函数调用的结果组成的数组

### 数据类型

基本类型： Undefined、Null、Boolean、Number、String，

新增:Symbol(创建后独一无二且不可变的数据类型 )

引用类型：object

对象在内存中存放在堆中，变量中保存着指向堆的指针用来引用对象。

所有对象都继承了Object的属性和方法。

引用类型包括：

Array

Date

Function

Json

**如何检测类型**

基本类型除了null，其余都可以用typeof检测，function也可以用typeof检测出，null可以用===判断

引用类型可以用instanceof检测出 ： 对象 instanceof 构造函数

数组还可以用Array.isArray()判断是否为数组

在Javascript中,想要判断某个对象值属于哪种内置类型,最靠谱的做法就是通过Object.prototype.toString方法.

var arr = [];

console.log(Object.prototype.toString.call(arr)) //”[object Array]”

### Ajax

new Promise(function(resolve,reject){

var request = new XMLHttpRequest();

request.onreadystatechange = function(){ //状态发生变化时，函数被调回

if(request.readyState === 4){ //成功完成

//判断响应结果

if(request.status === 200){ //成功，通过responseText拿到响应的文本

resolve(request.responseText);

}

}

}

request.open('GET','data.txt',true);

request.send(null);

})

.then(function(res){

console.log(res);

},function(){

console.log('erro');

})

## 难点

### 原型

1、如何理解面向对象？

面向对象的语言有一个标志，即拥有类的概念，它可以抽象实例对象的公共属性与方法，基于类可以创建任意多个实例对象，一般具有封装、继承、多态的特性！但JS中对象与纯面向对象语言中的对象是不同的，ECMA标准定义JS中对象：无序属性的集合，其属性可以包含基本值、对象或者函数。可以简单理解为JS的对象是一组无序的值，其中的属性或方法都有一个名字，根据这个名字可以访问相映射的值（值可以是基本值/对象/方法）。

在类语言中，对象基于模板来创建，首先定义一个类作为对现实世界的抽象，然后由类来实例化对象；

js（ES5）本身是没有class类型的，但是每个函数都有一个prototype属性。prototype指向一个对象，当函数作为构造函数时，prototype则起到类似class的作用。比如用原型模式创建对象，只需要在构造函数的原型上面定义属性和方法，那么所有实例都可以共享原型对象上的属性和方法，因为他们内部都有一个\_proto\_属性，指向了原型对象。

2、创建对象的几种方式？

第一，new+构造函数 var obj = new Object（）；

第二，对象字面量 var obj = { }；

第三，工厂模式

function person(name) {

var o = new Object();

o.name = name;

o.getName = function() {

return this.name;

}

return o;

}

var person1 = person('rose');

var person2 = person('jake');

这种模式在函数的内部创建了一个空对象，然后逐一添加属性和方法，最后返回，实现了对象得以复用的目的。但是存在2个很大的问题

无法识别对象的类型

console.log(person1 instanceof person); // false

每个对象调用的同名方法其实并不同一个方法

console.log(person1.getName == person2.getName); // false

其实就相当于每次声明对象都被重新创建，只不过写法上简单了一点而已。

第四、构造函数模式

通过参数控制每个实例里的属性的值

var Person = function(name) {

this.name = name;

this.getName = function() {

return this.name;

}

}

var person1 = new Person('tom'); //new的时候创建一个新对象

var person2 = new Person('tim');

console.log(person1 instanceof Person); // ture

console.log(person1.getName == person2.getName); //false

从上面代码可以看出，对象的类别可以判断了，person1就是Person的对象，可是2个同名方法任然不是同一个方法，而是重新创建，

第五，原型模式

让所有对象实例共享他所包含的属性和方法

function Person(){}

Person.prototype.name='小明';

Person.prototype.age = 23;

Person.prototype.sayName=function(){

alert(this.name)

}

var Person1 = new Person();

var Person2 = new Person();

alert(Person1.sayName == Person2.sayName) // true

第六，组合使用构造函数和原型模式

通过构造函数模式定义实例属性，通过原型模式定义方法和共享的属性

这样person1和person2就有各自的name,age,和job，而有着共同的方法sayName()

3 / 构造函数模式 定义实例属性/

4 function　Person(name,age,job){

5 this.name = name;

6 this.age = age;

7 this.job = job;

8 }

9

10

11 /原型模式 定义方法和共享的属性/

12 Person.prototype.sayName=function(){

13 alert(this.name)

14

15 var Person1 = new Person('小明',23,'前端')

16 var Person2 = new Person('小明明',33,'老师')

3、new操作符具体干了什么呢?

1、创建一个空对象，并且 this 变量引用该对象，同时还继承了该函数的原型。

2、属性和方法被加入到 this 引用的对象中。

3、新创建的对象由 this 所引用，并且最后隐式的返回 this 。

var obj = {};

obj.\_\_proto\_\_ = Base.prototype;

Base.call(obj);

4、如何理解原型和原型链？

我们创建的每个函数都有一个prototype属性，这个属性是一个指向原型对象的指针，原型对象里包含着所有实例共享的属性和方法。同时原型对象里也有一个constructor指回了其对应的构造函数。

每个用构造函数创建的对象实例都会在其内部初始化一个\_proto\_属性，它是一个指向prototype(原型)的指针，当我们访问一个对象的属性时，如果这个对象内部不存在这个属性，那么他就会通过\_proto\_去prototype里找这个属性，这个prototype又会有自己的\_proto\_指向prototype， 于是就这样一直找下去，形成了原型链的概念。

原型链实现的是对属性的查找，一直顺着原型链查找到object对象，如果没查到返回undefined，还有值得注意的一点就是，所有对象里的\_proto\_都是对原型对象的一个引用，而不是副本，所以当我们修改原型时，与之相关的对象也会继承这一改变。

### 继承

js主要有以下几种继承方式：对象冒充，call()方法，apply()方法，原型链继承以及混合方式。

第一，对象冒充

1 function Parent(username){

2 this.username = username;

3 this.hello = function(){

4 alert(this.username);

5 }

6 }

7 function Child(username,password）

12 this.method = Parent;//第一步：赋值，让内部属性指向所要继承的对象，

13 this.method(username); //第二步：调用刚才新创建的方法，也就是执行原函数

14 delete this.method; //第三步：删除该属性，这样以后就不能再调用它

15 this.password = password;

16 this.world = function(){

17 alert(this.password);

18 }

19 }

20 var parent = new Parent("zhangsan");

21 var child = new Child("lisi","123456");

22 parent.hello();

23 child.hello();

24 child.world();

第二，call()方法 （借用构造函数）

call方法是Function类中的方法

call方法的第一个参数的值赋值给类(即方法)中出现的this

call方法的第二个参数开始依次赋值给类(即方法)所接受的参数

7 function Parent(username){

8 this.username = username;

9 this.hello = function(){

10 alert(this.username);

11 }

12 }

13 function Child(username,password){

14 Parent.call(this,username); //第一个参数值this传递给了Parent类(即方法)中出现的this，而第二个参数赋值给Parent类中真正的参数

15 this.password = password;

16 this.world = function(){

17 alert(this.password);

18 }

19 }

20 var parent = new Parent("zhangsan");

21 var child = new Child("lisi","123456");

22 parent.hello();

23 child.hello();

24 child.world();

第三，apply()方法

apply方法接受2个参数，

A、第一个参数与call方法的第一个参数一样，即赋值给类(即方法)中出现的this

B、第二个参数为数组类型，这个数组中的每个元素依次赋值给类(即方法)所接受的参数

1 function Parent(username){

2 this.username = username;

3 this.hello = function(){

4 alert(this.username);

5 }

6 }

7 function Child(username,password){

8 Parent.apply(this,[username]);

9 this.password = password;

10 this.world = function(){

11 alert(this.password);

12 }

13 }

14 var parent = new Parent("zhangsan");

15 var child = new Child("lisi","123456");

16 parent.hello();

17 child.hello();

18 child.world();

第四，原型链继承

利用原型链来实现继承的原理是： 子类.prototype=new 父类的实例

function Parent(){

this.name = 'wang';

}

function Child(){

this.age = 28;

}

Child.prototype = new Parent();//继承了Parent，通过原型

var demo = new Child();

alert(demo.age);

alert(demo.name);//得到被继承的属性

缺点：如果父类的构造函数里有一个引用类型的属性，那么在其任何一个子类实例中改变这个属性，所有子类实例中的这个属性都会跟着改变，因为所有子类实例的\_proto\_属性都指向了同一个prototype原型对象，而这个原型对象正是父类的一个实例。也就是说，原型对象变成了父类的实例，所以，原先的实例属性也就变成了现在的原型属性了。

Sub.prototype = new Super; 会导致Sub.prototype的constructor指向Super;

然而constructor的定义是要指向原型属性对应的构造函数的，Sub.prototype是Sub构造函数的原型，所以应该添加一句纠正：

Sub.prototype.constructor = Sub;

第五，组合使用原型链和借用构造函数实现继承

原理是：使用原型链实现对共享属性和方法的继承，同时，通过借用构造函数（call）来实现对实例属性的继承。这样，实例既可以有自己的属性，也可以共享原型上的方法。

1 function Parent(name){

2 this.name= name;

3 this.colors=["red","blue","green"]

6 }

7 function Child(name,password){

8 Parent.apply(this,[name]); //继承属性

9 this.password = password;

13 }

14 Child.prototype=new Parent(); //继承方法

15 var child = new Child("lisi","123456");

https://blog.csdn.net/heyue\_99/article/details/54932553

ES6继承

ES6封装了class，extends关键字来实现继承，它内部的实现原理其实依然是基于es5的原型链，不过进过一层封装后，Javascript的继承得以更加简洁优雅地实现

es5中的构造函数在es6中被写成constructor,继承属性被简化成super(...)

class Father{

constructor(name){

this.name = name;

}

getName(){

console.log(this.name);

}

// 这里是父类的f方法

f(){

console.log('fffffffffffffffffffffff');

}

}

class Son extends Father{

constructor(name,age){

super(name); // HACK: 这里super()要在第一行

this.age = age;

}

getAge(){

console.log(this.age);

}

// 子类的f方法

f(){

console.log('sssssssssssssssssssssss');

}

}

### 作用域和作用域链

变量对象与活动对象

每个执行环境都有一个表示变量的对象-------变量对象，这个对象里储存着在当前环境中所有的变量和函数。

变量对象对于执行环境来说很重要，它在函数执行之前被创建。它包含着当前函数中所有的 参数， 变量， 函数。这个创建变量对象的过程实际就是函数内数据(函数参数、内部变量、内部函数)初始化的过程。

在没有执行当前环境之前，变量对象中的属性都不能访问！但是进入执行阶段之后，变量对象转变为了活动对象，里面的属性都能被访问了，然后开始进行执行阶段的操作。所以活动对象实际就是变量对象在真正执行时的另一种形式。

1. 每个执行环境有一个与之对应的变量对象。
2. 环境中定义的所有变量和函数都保存在这个对象里。
3. 对于函数，执行前的初始化阶段叫变量对象，执行中就变成了活动对象。

* 什么是作用域链？

全局函数无法查看局部函数的内部细节，但局部函数可以查看其上层的函数细节，直至全局细节。 当需要从局部函数查找某一属性或方法时，如果当前作用域没有找到，就会上溯到上层作用域查找， 直至全局函数。作用域链其实就是引用了当前执行环境的变量对象的指针列表，它只是引用，不是包含。作用域链的前端，始终都是当前执行的代码所在环境的变量对象。全局执行环境的变量对象也始终都是链的最后一个对象。

* 作用域链的形成流程。

1. 在创建函数时，作用域链已经预先包含了一个全局对象，并保存在内部属性[[ Scope ]]当中。
2. 执行函数时，创建执行环境与活动对象后，取出函数的内部属性[[Scope]]构建当前环境的作用域链(取出后，只有全局变量对象，然后此时追加了一个它自己的活动对象)。
3. 执行结束后，移出环境栈。
4. 当javscript 监听到函数没有被任何变量所引用，开始实施垃圾回收机制，清空占用内存。

### 闭包

闭包就是在一个函数内部返回一个函数，返回的这个函数里引用着外部函数的活动变量，这样就使得在外部函数调用结束以后，其执行环境和作用域链被销毁了，但是其活动对象保存在了闭包之中。因为引用的关系，外部函数的活动对象不能被垃圾回收机制回收，形成了依然可以通过引用访问其作用域链的结果。

利用闭包可以突破作用于链的限制，可以在函数外部访问该函数内部的变量和方法。还有就是让这些变量的值始终保持在内存中。

可能会引起内存泄漏

闭包的特性： 1.函数内再嵌套函数

2. 内部函数可以引用外层的参数和变量

3.参数和变量不会被垃圾回收机制回收

### This

this是js中的一个关键字，函数运行时自动生成的一个内部对象，只能在函数内部使用。我们要讨论的是 this 的指向。

this就是函数运行时自动生成的一个内部对象

this的指向不是在创建时就决定了，而是由执行环境决定的。

下面介绍一下几种情况下，this的指向

1、全局环境

全局环境下，this就代表window对象。（针对web 应用来讲）

var name = 'zhar';

function say(){

console.log(this.name);//zhar

}

say();

同样，在 setTimeout 或 setInterval 这样的延时函数中调用也属于全局对象

var name = 'zhar';

setTimeout(function(){

console.log(this.name);//zhar

},0);

2、对象环境

对象环境指向对象。

var obj = {

name : "zhar",

say : function(){

console.log(this.name);//zhar

}

}

obj.say();

下面举两个经典的例子：

var name = 'tom';

var obj = {

name : "zhar",

say : function(){

console.log(this.name);

}

}

var fun = obj.say;

fun();//输出 ?//tom-->fun定义在全局环境下，即window.fun()

//再次说明了this的指向是由运行时的执行环境来决定的

var name = 'tom';

var obj = {

name : "zhar",

say : function(){

return function(){

console.log(this.name);

}

}

}

obj.say()();//输出 ？//tom

3、构造函数环境

构造函数中的this 会指向创建出来的实例对象，使用new 调用构造函数时，会先创建出一个空对象，然后用call函数把构造函数中的this指针修改为指向这个空对象。执行完环境后，空对象也就有了相关的属性，然后将对象返回出去，所以说就不用我们自己手动返回啦~

function Person() {

this.name = 'zhar';

}

var p = new Person();

console.log(p.name);

综合以上，构造函数不需要返回值，如果我们指定一个返回值时，this的指向将发生变化

function Person() {

this.name = 'zhar';

return {};

}

var p = new Person();

console.log(p.name);//undefined

//--------------------------------------

function Person() {

this.name = 'zhar';

return {name:'tom'};

}

var p = new Person();

console.log(p.name);//tom 如果构造函数返回对象(Object,Array,Function)，那 this 将指向这个对象，其它基础类型则不受影响

//--------------------------------------

function Person() {

this.name = 'zhar';

return 1;//number string boolean 等

}

var p = new Person();

console.log(p.name);//zhar

所以，如无必要我们通常不要设置构造函数的返回值

4、事件对象

在 DOM 事件中使用 this，this 指向了触发事件的 DOM 元素本身

li.onclick = function(){

console.log(this.innerHTML);

}

总结下来就是一句话：是谁调用的，this就指向谁

下面介绍一下如何来修改this 的指向

1、可以使用局部变量来代替this指针

var name = "zhar";

var obj = {

name : "zhar",

say : function(){

var \_this = this;//使用一个变量指向 this

setTimeout(function(){

console.log(\_this.name);

},0);

}

}

obj.say();

该方法为非常常用的一个方法

2、使用call 或 apply 方法

首先说明一下，call也是函数调用的一种形式，可以通过 函数名.call（）来调用函数。但是提供了一个修改this指向的方法。

fun.call(thisObj[,arg1[,arg2[,...]]])

调用方式和传入参数如上面的形式。其中，所以call(thisObj[,arg1[,arg2[,...]]])中的第一个参数就是要更改this指向的对象，为必选参数; 之后的参数要根据调用的函数是否需要传入参数(为可选的)

下面通过代码来展示call 如何使用：

var name = 'zhar';

function say(){

console.log(this.name);

};

say();//zhar;

var obj = {

name : 'tom',

say : function(){

console.log(this.name);

}

}

say.call(obj);//tom 将 say 函数中的 this 替换为传入的对象

obj.say();//tom

obj.say.call(null);//zhar 将 obj.say 函数的 this 替换为了 null，也就意味着指向了全局环境

//前面课程的继承代码

function Person(){

this.name = "人";

}

function Student(){

Person.call(this,null);

}

var s = new Student();

console.log(s.name);

li.onclick = function(){

console.log(this.innerHTML);//此处的 this 代表着 DOM 元素

function update(){

this.innerHTML += " new ";

}

//update();//这样做的话，this 的指向将变为window

update.call(this);//通过 call 方法修改函数内 this 的指向

}

//call 的传参

function say(arg1,arg2){

console.log(this.name,arg1,arg2);

};

var obj = {

name : 'tom',

say : function(){

console.log(this.name);

}

}

say.call(obj,'one','two');//tom one two

apply

apply的作用和call一样，不同的是传参的形式。apply需要以数组的形式传递参数

//apply 的传参

function say(arg1,arg2){

console.log(this.name,arg1,arg2);

};

var obj = {

name : 'tom',

say : function(){

console.log(this.name);

}

}

say.apply(obj,['one','two']);//tom one two

如果没有提供 thisObj 参数，那么 Global 对象被用作 thisObj。

3.ES5还定义了一个方法：bind()，它会创建一个函数的实例，其this值会被绑定到传给bind()函数的值。如

indow.color='red';

var o={color:'blue'};

function sayColor(){

console.log(this.color);

}

var objectSaycolor=sayColor.bind(o);

//var objectSaycolor=sayColor.bind();

objectSaycolor();//blue

在这里sayColor()调用bind()并传入对象o,创建了objectSayColor()函数。objectSayColor()函数的this值等于o,因此即使是在全局作用域中调用这个函数，也会看到blue。

## 常见算法

### 深拷贝

首先深复制和浅复制只针对像 Object, Array 这样的复杂对象的。简单来说，浅复制只复制一层对象的属性，而深复制则递归复制了所有层级。

简单的赋值没有办法复制引用类型，这样只是复制了一次引用类型的地址而已，所以只要更改其中的任何一个，其他的也会跟着改变。本质上都是操作同一个对象。

一、浅拷贝

var color1 = ['red','green'];

var color2 = [];

for(var i = 0;i < color1.length;i++){

color2[i] = color1[i];

}

二、深拷贝

深拷贝实现的方式:对属性中所有引用类型的值，遍历到是基本类型的值为止，常用的做法是利用递归来实现深拷贝。

var deepCopy= function(source) {

var result={};

for (var key in source) {

result[key] = typeof source[key]===’object’? deepCoyp(source[key]): source[key];

}

return result;

}

JSON.parse(JSON.stringify(obj))

**递归遍历数组**

var arrs = [2 , [6,7,12,34,3] , [ 5,23,[1],9] ] ;

var arr = [];

function recursive(arrs) {

for(var i = 0; i < arrs.length; i++){

if(typeof arrs[i] === "number" ) {

arr.push( arrs[i] );

}else{

recursive( arrs[i] );

}

}

}

recursive(arrs);

console.log(arr);

# Vue

## 双向数据绑定的底层原理

## jquery和vue的本质区别

## vue的特点

# 网络协议相关

## 常见状态码

100 Continue 继续，一般在发送post请求时，已发送了http header之后服务端将返回此信息，表示确认，之后发送具体参数信息

200 OK 正常返回信息

201 Created 请求成功并且服务器创建了新的资源

202 Accepted 服务器已接受请求，但尚未处理

301 Moved Permanently 请求的网页已永久移动到新位置。

302 Found 临时性重定向。请求的网页临时移动到新位置。

303 See Other 临时性重定向，且总是使用 GET 请求新的 URI。

304 Not Modified 自从上次请求后，请求的网页未修改过。

400 Bad Request 服务器无法理解请求的格式，客户端不应当尝试再次使用相同的内容发起请求。

401 Unauthorized 请求未授权。

403 Forbidden 禁止访问。

404 Not Found 找不到如何与 URI 相匹配的资源。

500 Internal Server Error 最常见的服务器端错误。服务器端出错

503 Service Unavailable 由于临时的服务器维护或者过载，服务器暂时无法处理请求

## htttp缓存

分为强缓存和协商缓存

1.强缓存：用户发送的请求，直接从客户端缓存中获取，不发送请求到服务器，不与服务器发生交互行为。

描述强缓存的是服务器端的response header里的两个字段：expires 、Cache-Control

expires：Thu, 28 Sep 2017 06:38:37 GMT

Http1.0 中的标准，表示过期时间，它指的是服务器端的绝对时间，但是在客户端比较的是客户端的本地时间，所以可能存在偏差

Cache-Control：max-age=263563

Http1.1 中的标准，可以看成是 expires 的补充。使用的是相对时间的概念。

简单介绍下Cache-Control的属性设置。

1）max-age: 设置缓存的最大的有效时间，单位为秒（s）。max-age会覆盖掉Expires

3) public：响应会被缓存，并且在多用户间共享。默认是public。

4) private: 响应只作为私有的缓存，不能在用户间共享。如果要求HTTP认证，响应会自动设置为private。

5）no-cache:指定不缓存响应，表明资源不进行缓存。但是设置了no-cache之后并不代表浏览器不缓存，而是在缓存前要向服务器确认资源是否被更改。因此有的时 候只设置no-cache防止缓存还是不够保险，还可以加上private指令，将过期时间设为过去的时间。

6）no-store: 绝对禁止缓存。

2.协商缓存：用户发送的请求，发送到服务器后，由服务器判定是否从缓存中获取资源。

浏览器向服务器发送请求，如果上一次的缓存中有Last-modified 和 Etag 字段，

浏览器将在request header 中加入If-Modified-Since（对应于Last-modified）， 和If-None-Match（对应于Etag）。

Last-modified: 表明请求的资源上次的修改时间。

If-Modified-Since：客户端保留的资源上次的修改时间。

Etag：资源的内容标识。（不唯一，通常为文件的md5或者一段hash值，只要保证写入和验证时的方法一致即可）

If-None-Match： 客户端保留的资源内容标识。

如果资源没被修改过，就直接返回304状态码，告知浏览器使用缓存，否则返回200和请求内容。

## http和https的区别

## 网络分层

* 应用层：应用层、表示层、会话层（从上往下）（HTTP、FTP、SMTP、DNS）
* 传输层（TCP和UDP）
* 网络层（IP）
* 物理和数据链路层（以太网）
* 每一层的作用如下：
  + 物理层：通过媒介传输比特,确定机械及电气规范（比特Bit） 数据链路层：将比特组装成帧和点到点的传递（帧Frame）
  + 网络层：负责数据包从源到宿的传递和网际互连（包PackeT）
  + 传输层：提供端到端的可靠报文传递和错误恢复（段Segment）
  + 会话层：建立、管理和终止会话（会话协议数据单元SPDU）
  + 表示层：对数据进行翻译、加密和压缩（表示协议数据单元PPDU）
  + 应用层：允许访问OSI环境的手段（应用协议数据单元APDU）

## DNS解析过程

1. 浏览器先检查自身缓存中有没有被解析过的这个域名对应的ip地址，如果有，解析结束。同时域名被缓存的时间也可通过TTL属性来设置。

2. 如果浏览器缓存中没有（专业点叫还没命中），浏览器会检查操作系统缓存中有没有对应的已解析过的结果。而操作系统也有一个域名解析的过程。在windows中可通过c盘里一个叫hosts的文件来设置，如果你在这里指定了一个域名对应的ip地址，那浏览器会首先使用这个ip地址。

但是这种操作系统级别的域名解析规程也被很多黑客利用，通过修改你的hosts文件里的内容把特定的域名解析到他指定的ip地址上，造成所谓的域名劫持。所以在windows7中将hosts文件设置成了readonly，防止被恶意篡改。

3. 如果至此还没有命中域名，才会真正的请求本地域名服务器（LDNS）来解析这个域名，这台服务器一般在你的城市的某个角落，距离你不会很远，并且这台服务器的性能都很好，一般都会缓存域名解析结果，大约80%的域名解析到这里就完成了。

4. 如果LDNS仍然没有命中，就直接跳到Root Server 域名服务器请求解析

5. 根域名服务器返回给LDNS一个所查询域的主域名服务器（gTLD Server，国际顶尖域名服务器，如.com .cn .org等）地址

6. 此时LDNS再发送请求给上一步返回的gTLD

7. 接受请求的gTLD查找并返回这个域名对应的Name Server的地址，这个Name Server就是网站注册的域名服务器

8. Name Server根据映射关系表找到目标ip，返回给LDNS

9. LDNS缓存这个域名和对应的ip

10. LDNS把解析的结果返回给用户，用户根据TTL值缓存到本地系统缓存中，域名解析过程至此结束

## 三握四挥

**说说TCP传输的三次握手四次挥手策略**

* 为了准确无误地把数据送达目标处，TCP协议采用了三次握手策略。用TCP协议把数据包送出去后，TCP不会对传送 后的情况置之不理，它一定会向对方确认是否成功送达。握手过程中使用了TCP的标志：SYN和ACK
* 发送端首先发送一个带SYN标志的数据包给对方。接收端收到后，回传一个带有SYN/ACK标志的数据包以示传达确认信息。 最后，发送端再回传一个带ACK标志的数据包，代表“握手”结束。 若在握手过程中某个阶段莫名中断，TCP协议会再次以相同的顺序发送相同的数据包

**断开一个TCP连接则需要“四次握手”：**

* 第一次挥手：主动关闭方发送一个FIN，用来关闭主动方到被动关闭方的数据传送，也就是主动关闭方告诉被动关闭方：我已经不 会再给你发数据了(当然，在fin包之前发送出去的数据，如果没有收到对应的ack确认报文，主动关闭方依然会重发这些数据)，但是，此时主动关闭方还可 以接受数据
* 第二次挥手：被动关闭方收到FIN包后，发送一个ACK给对方，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号）
* 第三次挥手：被动关闭方发送一个FIN，用来关闭被动关闭方到主动关闭方的数据传送，也就是告诉主动关闭方，我的数据也发送完了，不会再给你发数据了
* 第四次挥手：主动关闭方收到FIN后，发送一个ACK给被动关闭方，确认序号为收到序号+1，至此，完成四次挥手
* 这个网上转载的例子不错：
* 三次握手：
* A:“喂，你听得到吗？”A->SYN\_SEND
* B:“我听得到呀，你听得到我吗？”应答与请求同时发出 B->SYN\_RCVD | A->ESTABLISHED
* A:“我能听到你，今天balabala……”B->ESTABLISHED
* 四次挥手：
* A:“喂，我不说了。”A->FIN\_WAIT1
* B:“我知道了。等下，上一句还没说完。Balabala…..”B->CLOSE\_WAIT | A->FIN\_WAIT2
* B:”好了，说完了，我也不说了。”B->LAST\_ACK
* A:”我知道了。”A->TIME\_WAIT | B->CLOSED
* A等待2MSL,保证B收到了消息,否则重说一次”我知道了”,A->CLOSED

# 构建工具和工程化

## Webpack

## git

# 性能优化

## 节流和防抖

# 关于找工作

### boss直聘

### 牛客网讨论区，内推信息

### 微信公众号：铅笔求职，内推君

### 公司官网

# 推荐UI库

## pc端

### ant-design-vue

### element-ui

移动端

### vux