跨域传值

浏览器同源(端口 域名 协议)策略：不同源的访问，浏览器会默认禁止
解决跨域的方法:

1.前端跨域: jsonp

1.jsonp(json with padding)（兼容性好，但只支持 get 方法，通过 callback 实现）
用 json 填充函数调用的实参
函数名(json);
JSONP 原理: JSONP 原理
ajax 请求受同源策略影响，不允许进行跨域请求，而 script 标签的 src 属性中的链接却可以访问跨域的 js 脚本，利用这个特性，服务端不再返回 JSON 格式的数据，而是返回一段调用某个函数的 js 代码，用 json 填充函数的实参，在 src 中进行了调用，这样实现了跨域。
jsonp 兼容性好但是只支持 get 请求 不支持 post 请求
2.webpack 中代理服务器实现跨域

```
proxy: {
  "/data": { //地址
  "target": " http://www.bjlink32.com/data.php", //接口地址,跨域访问
  // secure: false,// 如果是 https 接口，需要配置这个参数
  "changeOrigin": true,//开启跨域
  "pathRewrite": { "^/data" : "" }//如果接口本身没有/data 需要通过 pathRewrite 来重写了地址
  }
}
```

2.设置 document.domain 解决无法读取非同源网页的 Cookie 问题

因为浏览器是通过 document.domain 属性来检查两个页面是否同源，因此只要通过设置相同的 document.domain，两个页面就可以共享 Cookie（此方案仅限主域相同，子域不同的跨域应用场景。）

3.跨文档通信 API：window.postMessage()

调用 postMessage 方法实现父窗口 http://test1.com 向子窗口 http://test2.com 发消息（子窗口同样可以通过该方法发送消息给父窗口）
它可用于解决以下方面的问题：

- 页面和其打开的新窗口的数据传递
- 多窗口之间消息传递
- 页面与嵌套的 iframe 消息传递
- 上面三个场景的跨域数据传递

4.后端跨域

-- JSONP 原理

利用 html 里面 script 标签可以加载其他域下的 js 这一特性，使用 script src 的形式来获取其他域下的数据，但是因为是通过标签引入的，所以会将请求到的 JSON 格式的数据作为 js 去运行处理，显然这样运行是不行的，所以就需要提前将返回的数据包装一下，封装成函数进行运行处理，函数名通过接口传参的方式传给后台，后台解析到函数名后在原始数据上「包裹」这个函数名，发送给前端。（JSONP 需要对应接口的后端的配合才能实现）

深拷贝浅拷贝以及实现方法

浅拷贝：浅拷贝只是复制了对象的引用地址，两个对象指向同一个内存地址，所以修改其中任意的值，另一个值都会随之变化，这就是浅拷贝（object.assign 方法可以实现浅拷贝）
深拷贝：深拷贝是将对象及值复制过来，两个对象修改其中任意的值另一个值不会改变，这就是深拷贝
实现深拷贝的方法
用 for…in 实现遍历和复制
利用数组的 Array.prototype.forEach 进行复制
利用 JSON 实现： JSON.parse(JSON.stringify(obj))

Storage

SessionStorage（保存在 session 中，他是临时保存，关闭浏览器即刻销毁）
保存数据：sessionStorage.setItem(key,value);
读取数据：variable = sessionStorage.getItem(key);
localStorage（保存在本地磁盘）
保存数据：localStorage.setItem(key,value);
读取数据：variable = localStorage.getItem(key);

sessionStore 和 localStorage 区别

localStorage 生命周期是永久的，除非用户在浏览器上清除 localStorage 信息，否则这些信息将永远存在。存放数据大小一般为 5MB,而且它仅在客户端（即浏览器）中保存，不参与和服务器的通信
sessionStorage 仅在当前会话下有效，关闭页面或浏览器后被清除。存放数据大小一般为 5MB,

而且它仅在客户端（即浏览器）中保存，不参与和服务器的通信。

cookie 和 session 区别

cookie 和 session 的区别是：cookie 数据保存在客户端，session 数据保存在服务器端
cookie 原理是 web 服务器向客户端浏览器中存储 cookie 数据，下次进入这个网站的时候，cookie 会和 request headers 一起自动发送给 web 服务器
session 其实指的就是访问者从到达某个特定主页到离开为止的那段时间

设置 cookie

```
/**
 2      * 设置 cookie
 3      * @param name cookie 的名称
 4      * @param value cookie 的值
 5      * @param day cookie 的过期时间
 6      */
 7     var setCookie = function (name, value, day) {
 8       if(day !== 0){     //当设置的时间等于 0 时，不设置 expires 属性，cookie 在浏览器关闭后
删除
 9         var expires = day * 24 * 60 * 60 * 1000;
10         var date = new Date(+new Date()+expires);
11         document.cookie = name + "=" + escape(value) + ";expires=" + date.toUTCString();
12       }else{
13         document.cookie = name + "=" + escape(value);
14       }
15     };
```

节流和防抖

防抖：指触发事件后在 n 秒内函数只能执行一次，如果在 n 秒内又触发了事件，则会重新计算函数执行时间 （应用：实时搜索

（ 原理是设置一个计时器，规定在 delay 时间后触发函数，但是在 delay 时间内再次触发的话，就会取消之前的计时器而重新设置。这样一来，只有最后一次操作能被触发。）

节流：指连续触发事件但是在 n 秒中只执行一次函数。（应用：窗口调整（resize），页面滚动（scroll），抢购疯狂点击。）

（原理是通过判断是否到达一定时间来触发函数。根据 时间戳和定时器）

函数节流不管事件触发有多频繁，都会保证在规定时间内一定会执行一次真正的事件处理函数，而函数防抖只是在最后一次事件后才触发一次函数。 比如在页面的无限加载场景下，

我们需要用户在滚动页面时，

每隔一段时间发一次 Ajax 请求，而不是在用户停下滚动页面操作时才去请求数据。这样的场景，就适合用节流技术来实现。

防抖和节流怎么实现？

防抖就是一定时间内多次触发只执行最后一次：

```
function debounce(func, wait) {
  let timeout = null
  return function() {
      clearTimeout(timeout)
    timeout = setTimeout(() => {
       func.apply(this, arguments)
    }, wait)
  }}
```

防抖应用在防止重复提交

节流就是单位时间内只能运行一次：

```
function throttle(func, wait) {
    let previous = 0
  return function() {
  let now = +new Date()                    //触发事件的时间戳
  let remain = wait - (now - previous)         //等待时间大于当前时间减去上一次出发时间，大于 0 就可以再次调用
    if (remain < 0) {
    previous = now                    //当前时间
    func.call(this, arguments)
  }
  }}
```

闭包

闭包是指有权访问一个函数的私有变量的函数。

优点：实现变量复用；避免全局变量的污染。

缺点：比较消耗内存；使用不当容易造成内存泄露。

应用场景：

根据闭包的特征，我们可以做到在函数外部取到内部的变量值，因此我们可以将某些不想要定义到全局的变量，用闭包来实现，类似于“私有变量”。

闭包的应用回答 4 个

1）迭代器的使用
2）Getter 与 setter （Getter 可以获得属性值，setter 可以修改属性值）
3）页面功能 js 插件的封装
4）定时器，settimeout/setinterval

为什么用闭包封装

其实按照闭包的一般写法形式，简单的来说就是 函数里面又嵌套了函数。在团队开发中，为了防止命名冲突，我们一般会把相应的代码用闭包的形式包裹起来，以避免暴露在全局作用域下面。但是有个不好的地方是其内部变量不会被立马回收，有内存溢出的风险。

17.垃圾回收机制

Javascript 具有自动垃圾回收机制，会定期对那些我们不再使用的变量、对象所占用的内存进行释放

Javascript 的垃圾回收机制：变量生命周期结束，内存会被回收

全部变量：生命周期会一直持续，直到页面卸载

局部变量：函数调用结束，局部变量也不再被使用，它们所占用的空间也就被释放

闭包:由于闭包的原因，局部变量依然在被使用，所以也就不能够被回收
无法回收函数调用完应该释放的变量，所以闭包不能大量使用

Javascript 自动垃圾回收机制可以做到：
标记清除 :调用函数标记入栈，函数执行完出栈，内存被回收，标记清除
引用计数 :统计引用类型变量声明后被引用的次数，当次数为 0 时，该变量将被回收

18.常见内存泄漏的场景:

1.全局变量造成内存泄漏
2.未销毁的定时器和回调函数照成内存泄露
3. 闭包造成内存泄露
4.DOM 引用造成内存泄漏：Dom 的操作，会把 Dom 的引用保存在一个数组或者 Map 中，标记已经移除，DOM 对象还在内存中

19. XSS 攻击浏览器安全性问题

XSS 全称是 Cross Site Scripting 跨站脚本伪造/跨站脚本攻击

XSS 攻击:是值黑客向 HTML 文件中或 DOM 中注入恶意脚本。发展到现在，向 HTML 文件中注入恶意脚本的方式越来越多，所以是否跨域注入脚本已经不是唯一的注入手段，XSS 这个名字却一直保留。

19-1XSS 攻击主要手段:
1.窃取 cookie: document.cookie
2.监听用户行为
3.修改 DOM，伪造登录窗口
4.在页面内生成浮动广告、弹窗，影响用户体验

19-2 阻止 XSS 工具的主要手段:
1.服务器对象输入脚本进行过滤或转码
比如:str:<script>alert('xss 攻击')</script>
过滤:str:
替换:<script>alert('xss 攻击')</script>
2.充分利用 CSP 策略
禁止向第三方域提交数据，这样用户数据不会外泄
禁止执行内联脚本和未授权的脚本
提供上报跟踪机制，可以尽快发下有哪些 XSS 工具，尽快进行修复
限制加载其他域下的资源文件，这样黑客即使插入一个 js 文件，这个文件也无法被加载
3.使用 HttpOnly 属性
很多 XSS 攻击都来自于盗用 cookie,可以使用 HttpOnly 属性保护 cookie 安全，不允许 js 读取 cookie
php5.2 以上支持
<?php setcookie("abc", "test", NULL, NULL, NULL, NULL, TRUE); //设置客户端浏览器 js，无法读取 cookie ?>

20.CSRF 全称是 Cross-site request forgery 跨站请求伪造

CSRF 攻击:是指黑客引诱用户打开黑客的网站，在黑客的网站中，利用用户的登录状态发起的跨站请求。

CSRF 攻击主要手段:

1.自动发起 GET 请求

2.自动发起 POST 请求

3.引诱用户点击链接

4.在页面内生成浮动广告、弹窗，影响用户体验

如何阻止 CSRF 攻击:

1.充分利用好 cookie 的 SameSite 属性

php7.1 以上支持

```
<?php setcookie('samesite-test', '1', 0, '/; samesite=strict'); ?>
```

2.验证请求的来源站点

3.CSRF Token　进行登陆验证

21.XSS 攻击和 CSRF 攻击区别:

XSS 攻击主要是将脚本注入用户的页面，而 CSRF 主要是利用服务器的漏洞和用户登录状态进行攻击

http 状态码:

服务器和客户端浏览器约定好的请求服务器状态的编码

1XX：表示可继续发请求

2XX：表示成功

200 成功

204 成功无内容。

206 成功部分内容

3XX：表示重定向

301 永久重定向

302 临时重定向 禁止 POST 变为 GET，与 301 类似。但资源只是临时被移动。客户端应继续使用原有 URI

303 另外一个 URI，查看其它地址。与 301 类似。使用 GET 和 POST 请求查看

304 判断是否要更新缓存 请求头部携带 if-modified-since 自从上次更新距这次多久

307 临时重定向。与 302 类似。使用 GET 请求重定向

4XX：表示客户端错误

400 客户端语法错误

401 请求未经授权

403 服务器拒绝服务

404 请求资源不存在

5XX：服务端错误

500 不可预期的错误

503 此时不能提供服务 稍后恢复正常

http 和 https

HTTP（HyperText Transfer Protocol：超文本传输协议）是一种用于分布式、协作式和超媒体信息系统的应用层协议

HTTPS（Hypertext Transfer Protocol Secure：超文本传输安全协议）是一种透过计算机网络进行安全通信的传输协议

HTTP 与 HTTPS 区别

HTTP 明文传输，数据都是未加密的，安全性较差，HTTPS（SSL+HTTP） 数据传输过程是加密的，安全性较好。

使用 HTTPS 协议需要到 CA（Certificate Authority，数字证书认证机构） 申请证书，一般免费证书较少，因而需要一定费用。证书颁发机构如：Symantec、Comodo、GoDaddy 和 GlobalSign 等。

HTTP 页面响应速度比 HTTPS 快，主要是因为 HTTP 使用 TCP 三次握手建立连接，客户端和服务器需要交换 3 个包，而 HTTPS 除了 TCP 的三个包，还要加上 ssl 握手需要的 9 个包，所以一共是 12 个包。

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。

HTTPS 其实就是建构在 SSL/TLS 之上的 HTTP 协议，所以，要比较 HTTPS 比 HTTP 要更耗费服务器资源。

浏览器

0-1 浏览器内核

1、IE 浏览器内核：Trident 内核

2、Chrome 浏览器内核：2008 年以前是 Webkit 内核，2008 年以后统称为 Chromium 内核，2013 年后是 Blink 内核

3、Firefox 浏览器内核：Gecko 内核

4、Safari 浏览器内核：Webkit 内核（webkit 是 linux 的 KDE 的 KHTML 引擎的一个开源分支）

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

0.浏览器架构：

用户界面--》浏览器引擎--》数据存储（cookie 和 localstroge）和浏览器内核--》ui 后端（html 和 css 部分）、js 解释器、网络部分

0-2 进程和线程：

1.进程：就是一个程序运行的实例，启动一个程序的时候，操作系统会为该程序创建一块内存，用来存放代码、运行中的数据和一个执行任务的主线程，这样的一个独立运行环境称为一个进程
2.线程:每个进程可以分成多个线程协同工作

进程和线程的关系及联系:
1.进程之间是相互独立的，互相不干扰，如果需要进程间通信，可以使用 IPC 机制
2.进程中任意线程出错，都会导致整个进程崩溃
3.一个进程的多个线程之间，可以数据共享
4.当一个进程关闭，操作系统会回收进程所占资源
5.进程是操作系统分配资源的最小单位，线程是程序执行的最小单位。

0-3Chrome 多进程种类构成：　　五类

1. 浏览器（Browser 进程）主进程:
2. 网络进程：主要负责页面网络资源的加载等
3. 插件进程：每当插件运行时，就会创建一个插件进程
4. 渲染进程：负责页面渲染，将 HTML、CSS、JavaScript 翻译成用户能读懂的页面，浏览器会为每个页面创建一个浏览器渲染进程（包括浏览器内核[chrome JavaScript 的 v8 引擎、Blink 排版引擎]）
5. GPU 进程：用于 3D 绘制等

0-4Chrome 为什么要使用多进程浏览器架构：

1.稳定性：程序比较稳定，不容易崩溃，HTML、JS、CSS 是比较复杂的语言也会有各种 BUG 如果采用多线程可能会出现一个线程导致其他所有线程等崩溃的情况。放到不同的进程中就可以保证一个地方出现严重 BUG 的时候，不至于导致整个程序崩溃。
2.高效率：多进程可以利用多核 CPU 的并行处理优势
3.安全性：提高对恶意代码的防御能力。如果采用多线程架构里面，由于线程共享内存地址空间，有的恶意网页脚本，可能会通过一些技术对其他线程形成威胁。

1. 从输入 URL 到页面加载流程

常用概念简介:

DNS 解析时间:用户输入网址，域名解析成 IP 的时间，可以使用 ping 命令测试

TCP 连接时间:DNS 解析后，与服务器建立连接所花的时间
头部资源下载时间: 下载指定资源的时间
4. First Paint 时间（简称 FP）: 浏览器第一次渲染(paint)，用户从输入 URL 回车开始到页面开始有东西呈现为止（First Paint 包括 DNS 解析时间）

First Paint 时间=DNS 解析时间+TCP 连接时间+资源下载时间+DOM 创建时间+第一个 DOM 元素渲染时间

5.首屏时间: 用户浏览器首屏内所有内容都呈现出来所花费的时间

首屏时间= DNS 解析时间+TCP 连接时间+资源下载时间+DOM 创建时间+DOM 元素渲染时间

2.从输入 URL 到页面加载流程

1.浏览器输入 url，浏览器发送请求到服务器，服务器将请求的 HTML 返回给浏览器。

--可以通过 network 监控，查看

2. 浏览器下载完成 HTML(Finish Loading HTML)之后，便开始从上到下解析。
3. 解析的过程中碰到 css 和 js 外链都会请求资源（其实 HTML 的下载也是这个流程）都会执行以下过程：

Send Request:表示给这个外链对应的服务器发送请求

Receive Response: 表示接收响应，这里是表示告诉浏览器可以开始从网络接收数据了

Receive Data:表示开始接收数据

Finish Loading: 表示已经完成下载数据。

Parse Stylesheet/Evaluate（默认情况下 js 下载完成之后执行 Evaluate，css 下载完成后会进行 Parse Stylesheet）

4. 所有的 css 下载完成后 Parse Stylesheet 然后开始构建 CSSOM
5. DOM（文档对象模型）和 CSSOM（CSS 对象模型）会合并生成一个渲染树(Render Tree)
6. 根据渲染树的内容计算处各个节点在网页中的大小和位置（Layout，可以理解为“刻章”）
7. 根据 Layout 绘制内容在浏览器上（Paint，可以理解为“盖章”）。

HTML 会从上到下依次创建 DOM，css 会创建 CSSOM
然后通过 render 合并成一棵渲染树
根据渲染树内容计算各个节点在网页的大小和位置 Layout 过程，相当于刻章
根据 layout 再浏览器上绘制内容

http 请求的全过程

输入域名(url)-->DNS 映射为 IP-->TCP 三次握手-->HTTP 请求-->HTTP 响应-->(浏览器 跟踪重定向地址)-->服务器处理请求-->服务器返回一个 html 响应-->(视情况决定释放 TCP 连接)-->客户端解析 HTML-->获取嵌入在 HTML 中的对象重新发起 http 请求

TCP 三次握手

首先是客户端浏览器向服务器发起 TCP 连接，发送一个 SYN 同步报文
然后服务器会响应客户端浏览器发送一个 ACK 应答报文--表示已创建连接
然后客户端浏览器给服务器发送 ACK 应答报文，表示已经收到连接
说明:SYN:表示同步报文
ACK:表示响应
FIN 表示关闭连接

四次挥手

首先客户端浏览器会向服务器发送 FIN（关闭连接）
服务器向浏览器发送应答报文
然后再向客户端浏览器发送 FIN 关闭连接的报文
客户端浏览器收到关闭报文后，发送 ACK 报文确认关闭

event loop---事件循环机制

为什么要用 event LOOP:因为 js 是单线程，如果没有 event loop 遇到异步请求时就会等到异步请求结束后再向下执行

Event Loop 即事件循环机制，是指浏览器或 Node 的一种解决 javaScript 单线程运行时不会阻塞的一种机制，也就是我们经常使用异步的原理。

事件队列包括宏任务和微任务
宏任务包括：整体代码的 script 标签，settimeout，setiniterval
微任务包括：promise，process.nextTick
script 标签就是第一个红任务，执行顺序是先执行红任务中的同步代码，遇到微任务放到一侧进入支线排队，同步代码执行完毕执行微任务，如果遇到宏任务，就将红任务进行排队，然后继续执行，判断一个红任务是否执行完毕的方法就是看是否还有未执行完的微任务，如果宏任务执行完，且没有微任务未执行，那么就是结束 js 脚本的执行

settimeout 如果里面有参数，那么他就是多久时间后在去排队

Settime

将设置的日期和时间距离 1970 年 1 月 1 日午夜之间的毫秒数转换成年月日这种日期格式。(这种类型的毫秒值可以传递给 Date() 构造函数，可以通过调用 Date.UTC() 和 Date.parse() 方法获得该值。以毫秒形式表示日期可以使它独立于时区。)

懒加载和预加载

懒加载:懒加载(延迟加载),当数据在需要用到的时候采取加载。

懒加载的作用：

1>增强用户体验

2>优化代码

3>减少页面的首屏加载时间

4>减少 http 的请求

5>按需加载,减少服务器压力

实现方法:设置一个函数当元素到浏览器的可视范围的距离小于浏览器窗口的尺寸时让图片显示出来,否则图片不加载。

预加载:

1>图片体积小,js 实现与加载

2>图片转换成 base64 编码,插入到 css 中，随 css 一起加载,css 文件一般都放到<head></head>中

通过 js 实现图片预加载/插入到 css 中，随 css 一起加载

```
var img = new Image();  img.src="...."
  document.createElment("img"); div1.appendChild(img1);
```

get 和 post 的区别

请求参数：get 参数附在 URL 后面?隔开，POST 参数放在包体中

大小限制：GET 限制为 2048 字符，post 无限制

安全问题：GET 参数暴露在 URL 中，不如 POST 安全

浏览器历史记录：GET 可以记录，POST 无记录

缓存：GET 可被缓存，post 无

书签：GET 可被收藏为书签，post 不可

数据类型：GET 只能 ASCII 码，post 无限制

提供公共服务的用 get 方法，传输少量数据用 get 方法

全捡信息用 post 方法，还可以传输大量数据

浏览器缓存机制

--两者的共同点是，都是从客户端缓存中读取资源；区别是强缓存不会发请求，协商缓存会发请求。

--强制缓存

不会向服务器发送请求，直接从缓存中读取资源，在 chrome 控制台的 Network 选项中可以看到该请求返回 200 的状态码，

并且 Size 显示 from disk cache 或 from memory cache。强缓存可以通过设置两种 HTTP Header 实现：Expires 和 Cache-Control。

--协商缓存

协商缓存就是强制缓存失效后，浏览器携带缓存标识向服务器发起请求，由服务器根据缓存标识决定是否使用缓存的过程，主要有以下两种情况：

协商缓存生效，返回 304 和 Not Modified

协商缓存失效，返回 200 和请求结果

协商缓存可以通过设置两种 HTTP Header 实现：Last-Modified 和 ETag。

DOM 的回流和重绘：

DOM 回流：页面中元素的位置，大小或结构、定位发生改变，会引发浏览器对当前页面的结构进行重新计算，非常耗性能；

DOM 重绘：当元素的背景、透明度、颜色发生变化，那么浏览器会对元素进行重新描绘；这个过程就是浏览器的重绘

区别: .

他们的区别很大:

回流必将引起重绘，而重绘不一定会引 起回流。比如:只有颜色改变的时候就只会发生重绘而不会引起回流

当页面布局和几何属性改变时就需要回流

比如:添加或者删除可见的 DOM 元素,元素位置改变,元素尺寸改变一边距、 填充、边框、宽度和高度，内容改变

网站性能优化

web 性能时间:从用户输入网址到页面呈现到用户面前，所花的时间。

--1.域名解析事件 域名-->DNS-->IP

--2.网页下载速度

---网页资源体积尽可能小

----js 压缩

----css 压缩

----图片优化

----代码纯手写

---减少 http 请求次数

---jss 合并--模块化开发

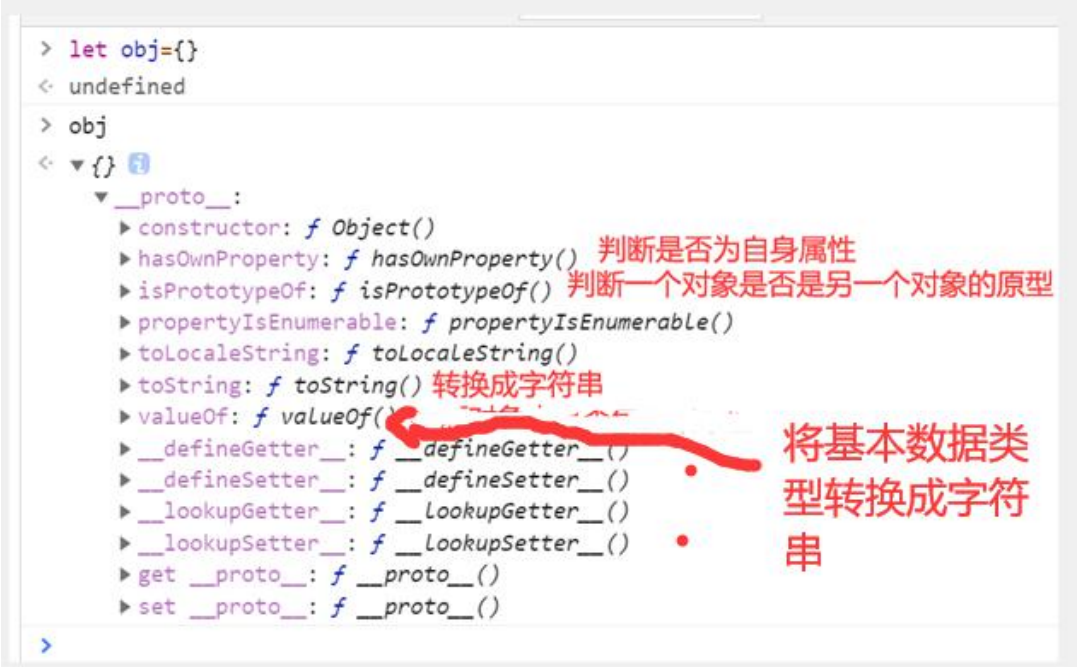
---css sprites

---webpack 打包
---js 采用异步加载 defer/async
--3.网页渲染时间
--CSS 进行优化
---CSS 书写顺序
---CSS 后代选择器不要超过三层
---颜色不要缩写
---代码要精炼
---CSS 样式初始化
---不使用通用选择器
--js 进行优化
---尽量减少 DOM 刷新次数
---react vue 引入虚拟 DOM diff 算法
---原生 js,引入文档碎片
---if-else 嵌套代替 if-else-if
---避免死循环
---少用闭包

原型和原型链

答：简单来说，每个实例化对象（有例外）都有预定义属性 prototype 和__proto__，实例化对象的__proto__属性指向原型对象，该原型对象也有__proto__属性，它指向创建它的函数 Object 的 prototype，Object.prototype 的__proto__属性为 null，我们把这条链称为原型链。

__proto__都有哪些属性:



补充 1：
当我们调用一个对象的属性时，如果对象没有该属性，JavaScript 解释器就会从对象的原型对象上去找该属性，如果原型上也没有该属性，那就去找原型的原型，直到最后返回 null 为止。这种属性查找的方式被称为原型链（prototype chain）。
补充 2：
prototype 属性属性包含一个对象（以下简称"prototype 对象"），所有实例对象需要共享的属性和方法，都放在这个对象里面；那些不需要共享的属性和方法，就放在构造函数里面。实例对象一旦创建，将自动引用 prototype 对象的属性和方法。也就是说，实例对象的属性和方法，分成两种，一种是本地的，另一种是引用的。

作用:

1.实现同一个构造器或类 创建的对象方法共享

2.实现对象的继承

5.如果自身属性和原型属性发生冲突时，像调用原型属性上的方法怎么调用

对象.__proto__.方法名调用

因为 ie6--10 不能直接调用__proto__，所以可以使用 constrouctor 调用原型上的方法

对象名.constructor.prototype.方法名调用

附加：如果对象的中和原型中有同名属性或方法访问方式

对象名. constructor.prototype.属性/方法名

说明:每个对象都有一个构造器，而原型 prototype 本身就是一个对象，它也有构造器，这个构造器又会有自己的原型

li. constructor

返回 li 的构造器 Stu

li.constructor.prototype

返回构造器的原型

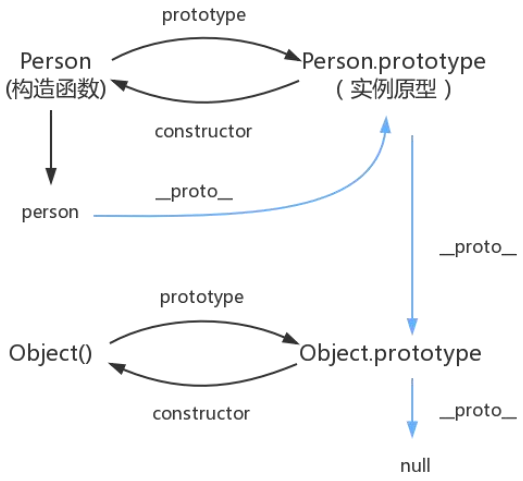
li.constructor.prototype. constructor

返回 构造器 Stu

li.constructor.prototype. constructor.prototype

返回 Object{}

说明:Object 是最高级的父对象



IE 和 DOM 事件流的区别：

答案：当一个 HTML 元素产生一个事件时，该事件会在元素结点与根结点之间的路径传播，路径所经过的结点都会收到该事件，这个传播过程可称为 DOM 事件流。

IE 采用冒泡型事件
Netscape 使用捕获型事件
DOM 使用先捕获后冒泡型事件
冒泡型事件模型： button->div->body (IE 事件流)
捕获型事件模型： body->div->button (Netscape 事件流)
DOM 事件模型： body->div->button->button->div->body (先捕获后冒泡)

addeventListener 方法中有三个参数，第一个是绑定的事件，第二个是事件处理函数，第三个就是参数为 true 时是捕获型事件，为 false 是冒泡性事件

阻止冒泡事件

1.event.stopPropagation(); // 阻止了事件冒泡，但不会阻击默认行为
2.return false; //阻止全部事件
3.event.preventDefault(); //阻止默认事件
IE6：event. cancelBubble 读音：看搜把 bou

事件委托（事件代理）绑定

事件委托：

对“事件处理程序过多”问题的解决方案就是事件委托。事件委托利用了事件冒泡，只制定一个事件处理程序，就可以管理某一类型的所有事件。
（1）例如 click 事件一直会冒泡到 document 层。也就是我们可以只指定 onclick 事件处理程序，而不必给每个事件分别添加处理程序
（2）如果是给 ul 中多个 li 添加事件，就可以把所有的 li 事件都写在 ul 中，通过节点去判断是否为 li 节点

事件绑定：
嵌入 dom
直接绑定
事件监听

原生 js 常用方法 DOM 操作（如何获取 dom 对象）

document.getElementById ： 根据 ID 查找元素，大小写敏感，如果有多个结果，只返回第一个；
document.getElementsByClassName ： 根据类名查找元素，多个类名用空格分隔，返回一个
document.getElementsByTagName ： 根据标签查找元素， * 表示查询所有标签，返回一个 HTMLCollection 。
document.getElementsByName ： 根据元素的 name 属性查找，返回一个 NodeList 。
document.querySelector ： 优先匹配对应的第一个标记。
document.querySelectorAll ： 所有匹配对应标记所集成的数组。
document.createElement("div") 创建一个 dom 对象
appendChild 它会将 child 追加到 parent 的子节点的最后面。

onload DOMContentLoaded 区别

当 onload 事件触发的时候，页面上所有的 DOM，样式，js，图片，都已经加载完成了。
当 DOMContentLoaded 事件触发的时候，仅 DOM 加载完成，不包括样式，js，图片。

如何将 json 转换为 JavaScript 对象格式

JSON。prase（js 对象） / JS 自带的 eval()函数;
17.如何将 JavaScript 对象转换为 json 格式

JSON.stringify（[要转换的对象]，[转换函数]，[空白处理方法]）

json 和 js 之间转换

利用 json 解析器 JSON.parse(str)的方式,将 json 字符串转换成 object 对象
eval ("(" + txt + ")")　　将 json 字符串转换成 object 对象

typeof 和 instanceof 区别

1.typeof 6 个值　类型检测运算符
　　Number　string　Boolean　undefined　object　function
typeof 判断基本类型/instanceof 判断对象拓展类型
2.instanceof 判断对象的类型
　　String(字符串对象)
　　Array（数组）
　　Math（数学）
　　Date（时间）
　　Number （数字）
　　Function （函数/方法）
　　RegExp（正则表达式）
　　Error （异常）
instanceof 判断对象的类型--->判断某个对象是否是某个类的实例-->判断某个对象的是否是某个构造器函数创建的

对象的分类:--普通对象和函数对象

1.函数对象 function
　　有两个内置属性:
　　　　prototype:用来共享对象的属性或方法
　　　　__proto__:指向构造器的原型 prototype 属性

2.普通对象 object
　　只有__proto__属性，始终指向构造器的 prototype 属性
　　普通对象.__proto__===构造器.prototype

如何实现链式操作

jQuery 有一个十分强大的特点：链式操作。其实原理很简单，**就是每次方法执行完后返回 this 对象，**

这样后面的方法就可以继续在 this 环境下执行。先来实现一下：

链式操作的优缺点

首先来说说缺点。

如果请求方法中里因为有值需要返回，就不能返回 this，。这就是链式操作的一个局限：

只能应用在不需要返回值的情况下，或者只能最后一步需要返回值的情况下使用链式操作。 jQuery 主要是对 DOM 元素的操作，只需要改变 DOM 元素的表现而不需要返回值，所以适合链式操作。

优点：

js 的执行环境为单线程，为了避免阻塞，可以使用异步编程方式来完成一些可能产生阻塞的操作。

常见的异步编程方式包括回调函数、事件监听、ES6 中的 Promise 对象。

链式操作可以使异步编程的流程更加清晰，不会像回调函数一样相互耦合，难以分辨函数的执行顺序且维护困难。

ES6 中的 Promise 也正是沿用了这一思想，每一个异步任务返回一个 Promise 对象，通过 then 方法指定回调函数。

```
//创建一个类
function Person(){};
//在原型上定义相关方法
Person.prototype ={
    setName:function(){
        this.name = name;
        return this;
    },
    setAge:function(){
        this.age = age;
        return this;
    }
}
//实例化
var person= new Person();
person.setName("Mary").setAge(20);
```

以上过程就实现了链式操作，而原理只是在每个方法调用后返回了 this。jQuery 中不用 new 一个实例化对象，

所以可以将实例化过程封装如下：

```
    //将实例化过程封装为一个函数
function newObj(){
    return new Person();
```

```
}
newObj().setName("Mary").setAge(20);
```
此时再想想上面的那道题，就很简单了：

```
function Person(){};
//实现部分
Person.prototype ={
    set:function(value){
        this.value = value;
        return this;
    },
    get:function(){
        return this.value*2;//由于要返回值，且不需要继续调用方法，所以不返回 this
    }
}
var person = new Person();
console.log(person.set(10).get());//20
```

文档碎片

DocumentFragments 是 DOM 节点。它们不是主 DOM 树的一部分。通常的用例是创建文档片段，将元素附加到文档片段，然后将文档片段附加到 DOM 树。在 DOM 树中，文档片段被其所有的孩子所代替。因为文档碎片存在于内存中,并不在 DOM 树中，所以将子元素插入到文档片段时不会引起页面回流。因此，使用文档片段 document fragments 通常会起到优化性能的作用。

DocumentFragment 节点不属于文档树，继承的 parentNode 属性总是 null。

可以用 Document.createDocumentFragment()方法创建新的空 DocumentFragment 节点

也可以用 Range.extractContents()方法或 Range.cloneContents()方法获取现有文档的片段的 DocumentFragment 节点

事件对象及事件对象常用属性和方法

e.target 获取真正触发事件的对象

e.preventDefault() 取消默认行为

e.pageX/e.pageY 页面位置， IE8 以前不支持，

event.stopPropagation() 该方法将停止事件的传播，阻止被分派到其他 Document 节点。在事件传播的任何阶段都可以调用它。

获取 url 后面的属性值

https://www.baidu.com/?a=1&b=2,如何截取对应的 a 和 b 的值

```
function array(url) {
        var getValue={}
        url = url.replace(/.*\?/, '')          //将 url？前面的部分替换成空字符串
        console.log(url)
        var arr = url.split('&');              //将字符串按照&分割成数组
        var temp;
        for (var i = 0; i < arr.length; i++) {
            temp = arr[i].split('=');          //将数组元素按照等号分割成数组
            console.log("temp",temp)                  //temp (2) ["a", "1"]
            getValue[temp[0]] = temp[1];
            console.log(temp[1])
        };
        console.log(getValue);

    }
    array("https://www.baidu.com/?a=1&b=2")
```

什么是枚举属性

可以通过 for in 遍历出来的是枚举属性，对象自带的是不可枚举属性

如何判断对象的属性是否是枚举属性？

--通过对象的 propertyIsEnumerable()方法判断是否是枚举属性
--对于所有的对象自己带属性 propertyIsEnumerable()都是 false

```
s1.propertyEnumerable("属性名")
s1.constructor.prototype.propertyIsEnumerable("属性名")
```

如何判断对象属性是自身属性还是原型属性

对象属性是自身属性还是原型属性可以通过对象的方法 hasOwnProperty()判断

```
s1.hasOwnProperty("属性名")
```

isPrototypeOf()方法判断对象是否是另一个对象的原型

每个对象都有一个 isPrototypeOf 方法，判断对象是否是另一个对象的原型

delete 操作符:

可以删除对象的属性和方法，不能直接删除对象构造器原型上的属性和方法。

说明:但是可以删除类原型上的属性和方法
可以使用 delete Stu.prototype.方法名

JavaScript 脚本异步加载—defer，async

1.defer --和 DOM 操作相关的脚本使用 defer 进行加载
脚本加载时间和 DOM 创建时间，并行进行
脚本等所有 DOM 对象创建完毕才执行
2.async --和 DOM 操作无关的脚本
脚本加载时间和 DOM 创建时间，并行进行
脚本加载完成不管所有 DOM 对象是否创建完毕立即执行脚本

js 兼容性问题

--主要原因:IE6/7/8 不完全遵守 ECMASCRIPT 标准，加入自己的私有写法
w3c/IE9+ addEventListener
IE6/7/8 attachEvent
IE9/10 既支持私有写法，也支持 w3c 标准
IE11 开始，只支持 w3c 标准

获取元素高宽

网页可见区域宽： document.body.clientWidth;
网页可见区域高： document.body.clientHeight;
网页可见区域宽： document.body.offsetWidth (包括边线和滚动条的宽);
网页可见区域高： document.body.offsetHeight (包括边线的宽);
网页正文全文宽： document.body.scrollWidth;
网页正文全文高： document.body.scrollHeight;
网页被卷去的高(ff)：document.body.scrollTop;
网页被卷去的高(ie): document.documentElement.scrollTop;
网页被卷去的左：document.body.scrollLeft;
网页正文部分上：window.screenTop;
网页正文部分左：window.screenLeft;
某个元素的宽度： obj.offsetWidth;
某个元素的高度：obj.offsetHeight;

某个元素的上边界到 body 最顶部的距离：obj.offsetTop;（在元素的包含元素不含滚动条的情况下）
某个元素的左边界到 body 最左边的距离：obj. offsetLeft;（在元素的包含元素不含滚动条的情况下）
返回当前元素的上边界到它的包含元素的上边界的偏移量：obj. offsetTop（在元素的包含元素含滚动条的情况下）
返回当前元素的左边界到它的包含元素的左边界的偏移量：obj. offsetLeft（在元素的包含元素含滚动条的情况下）

scrollTop, scrollLeft 设置或返回已经滚动到元素的左边界或上边界的像素数。只有在元素有滚动条的时候，例如，元素的 CSS overflow 属性设置为 auto 的时候，这些像素才有用。这些属性也只在文档的 <body> 或 <html> 标记上定义（这和浏览器有关），并且一起来制定滚动文档的位置。注意，这些属性并不会指定一个 <iframe> 标记的滚动量。这是非标准的但却得到很好支持的属性

-- Web Worker

JavaScript 语言采用的是单线程模型（同一时间只能做一件事），也就是说，所有任务只能在一个线程上完成，一次只能做一件事。
web worker 对象的出现 ，就是为了 javascript 创造多线程环境（同一时间能做多件事），语序主线程创建 worker 线程，将一些任务分配给后者运行。在主线程运行的同时，worker 线程（外部的 j 其他 s 文件）在后台运行，两者互不干扰。

ajax 执行流程

--ajax 执行流程--封装
代码：

```
function ajax(param){
    try{
      if(param!=null&&typeof param=="object"){
        //创建 ajax 对象
        if(window.XMLHttpRequest){
          var xmlHttp=new XMLHttpRequest();
        }else{
          var xmlHttp=new ActiveXOject("Microsoft.XMLHTTP");
        }
        xmlHttp.onload=function(){
          param.completed();
        }
        //设置接口地址和请求方式
        if(param.type=="GET"&&param.data!=undefined){
```

```
                xmlHttp.open(param.type,param.url+"?"+param.data);
            }else{
                xmlHttp.open(param.type,param.url);
            }
            //设置数据的编码格式
            if(param.contentType!=undefined&&param.contentType!="formdata"){
              switch(param.contentType){
                    case "urlencoded":

xmlHttp.setRequestHeader("content-type","application/x-www-form-urlencoded");
                    break;
                    case "json":
                    xmlHttp.setRequestHeader("content-type","application/json");
                    break;
              }
            }
            //绑定 ajax 时间，监控请求 ajax
            xmlHttp.onreadystatechange=function(){
             if(xmlHttp.readyState==4&&xmlHttp.status==200){
                var data=xmlHttp.responseText;
                switch(param.dataType){
                        case "json":
                       data=JSON.parse(data);
                        break;
                }
                param.success(data);
              }
            }
            //发送请求
            if(param.type=="POST"&&param.data!=undefined){
              xmlHttp.send(param.data);
            }else{
                xmlHttp.send();
            }
         }else{
            throw new Error("参数不正确");
          }
        }catch(e){
          alert(e.message);
        }
    }
```

module.exports 和 exports 区别

exports 方式使用方法是：exports.[function name] = [function name]
moudle.exports 方式使用方法是：moudle.exports= [function name]
exports 返回的是模块函数
module.exports 返回的是模块对象本身，返回的是一个类

js 执行流程

JavaScript 执行流程分为两个阶段:
第一个阶段（预）编译阶段：进行变量提升
1.创建所有带 var 的变量，并且赋值为 undefined
2.创建所有有名的函数
说明:匿名函数不参加预编译
赋值不在预编译阶段执行

第二个阶段 执行阶段:从上到依次执行代码

JavaScript 执行上下文三种环境

```
var x=100;
let y=200;
```

全局上下文环境 ：
JavaScript 在执行全局代码的时候，会编译全局代码并创建全局执行上下文，在整个页面生命周期中，全局执行上下文只有一个份

函数上下文:

```
function test(){
    var x=3000;       //test 函数上下文环境
}
```

当调用一个函数的时候，函数内的代码会被编译，并创建函数执行上下文，一般情况下，函数执行结束后，创建的函数执行上下文会被销毁。

eval 上下文环境：

```
eval("var x=9000")//eval 上下文环境
```

9-0　如何管理上下文环境---结构---栈结构什么是调用栈

JavaScript 引擎会将执行上下文压入栈中，用来管理执行上下文的栈，称为执行上下文栈，又称为调用栈。

9.js 调用栈的的执行规则，

上下文环境先入栈，然后再执行这个栈中的 js 代码

分为两个阶段:预编译阶段　执行阶段

10.什么是作用域栈

用来管理上下文环境的栈结构

11.栈是有大小限制的，如果过大，会显示栈溢出

12.Javascript 作用域:

1.全局作用域
2.局部作用域
3.块作用域　– ES6 新增
4.严格模式下有个 eval 作用域　（可不说）

13.Es6 中的块作用域:

1. if 块

if(){}

2. while 块

while(){}

3. 函数块

function foo(){}

4. for 循环块

for(let i = 1; i<100; i++){}

5. 单独一个块

{}

块作用域使用:let 或 const

为什么使用块作用域：为了将变量命名冲突，减少到最小的范围

14,词法环境优先级高于变量环境，词法环境就是 let，和 const 块的运行环境

15.什么是作用域链

当前函数中没有要查找的变量，JavaScript 引擎就去全局上下文查找，这个查找的链条称为作用域链

16.JavaScript 中数据类型分为:

1.基本数据类型—存储到栈结构中（存储空间小）

--基本类型赋值的、拷贝的、比较的、当做参数传递的都是值

2.对象类型（引用类型）—存储到堆结构中（存储空间大）

--对象类型赋值的、拷贝的、比较的、当做参数传递的都是地址

git rebase 和 git merge

merge:将在子分支的所有提交记录成一次 commit，保留在记录中。（下图的 E 即为该记录）

rebase:不会保留 commit 记录，直接将分支中的内容排到 master 的记录之后

git 解决冲突

解决方法：

1、原始方法：

a、手动修改文件： 一是选择使用分支 1 的代码，二是选择使用分支的代码，三是手动修改冲突部分的代码

b、git add: 解决好冲突后，输入 git status，会提示 Unmerged paths。使用 git add 表示修改冲突成功（或是在 sourceTree 中标记冲突解决，在 sourceTree 中解决冲突时请始终选择暂存区内的冲突文件进行操作），此时会加入暂存区。

c、git commit 和 git push

冲突的类型：

逻辑冲突：

a、比如另外一个人修改了文件名，但我还使用老的文件名，这种情况下自动处理是能成功的，但实际上是有问题的

b、函数返回值含义变化，但我还使用老的含义，这种情况自动处理成功，但可能隐藏着重大 BUG

内容冲突：

两个用户修改了同一个文件的同一块区域，git 会报告内容冲突

树冲突：

文件名修改造成的冲突，称为树冲突。

如：a 用户把文件改名为 a.c，b 用户把同一个文件改名为 b.c

做过登录的页面没有 后台给一个 token 怎么处理

前端登录

后台获取登录信息校验

后台校验成功后返回 token

前台拿到 token 存储到 cookie 或者 localstage 中

以后每次请求都会携带 token

你在你们项目组的时候，怎么解决跨域联调的

找到四种方法：

1、webpack-dev-server 配置跨域方案

2、http-proxy-middleware 模块设置代理服务
3、让新版 Chrome 支持本地跨域请求调试
4、nginx 反向代理解决跨域设置