准备

页面布局

- 一、假设高度已知，请写出三栏布局，其中左右两栏宽度各为300px，中间自适应。
- 二、语义化
- 三、在写页面结构时，我们应该注意什么呢?

盒模型（盒模型是CSS的基石，很重要）

- 一、谈谈你对CSS盒模型的认识
- 二、css如何设置两种模型
- 三、JS如何设置获取盒模型对应的宽和高
- 四、实例题（根据盒模型解释边距重叠）
  - 边距重叠解决方案（BFC）
  - BFC的原理（即BFC的渲染规则）
  - 如何创建BFC
  - 应用场景（https://www.cnblogs.com/lhb25/p/inside-block-formatting-ontext.html）

DOM事件

- DOM事件的级别（DOM定义标准的一个级别）
- DOM事件模型（冒泡、捕获）
- DOM事件流
- DOM事件捕获的具体流程
- Event对象的常见应用
- 自定义事件（或者叫模拟事件 new Event()）

HTTP协议类

- HTTP协议的主要特点
- HTTP报文的组成部分
- HTTP方法
- POST和GET的区别

get、post的区别

- HTTP状态码
- HTTP协议持久连接
- HTTP协议管线化

原型链

- 创建对象的几种方法
- 原型、构造函数、实例、原型链
- instanceof的原理
- new运算符

call 和 apply

浏览器缓存 sessionStorage、localStorage、cookie、userData

面向对象

- 类与实例
- 类与继承

说一下继承的几种方式及优缺点?

深度克隆

- 1 硬刚法（迭代法，适用于所有）
- 2 投机取巧法（Json方法，适用于部分）

JS数组去重

递归
用js递归的方式写1到100求和?
排序
    冒泡排序
    插入排序
js操作获取和设置**cookie**
写一个通用的事件侦听器函数**?**
window.onload和document.ready的区别？哪一个先执行?
null和undefined的区别
通信
    一个页面从输入 URL 到页面加载显示完成，这个过程中都发生了什么?
    性能优化
安全
算法
控制台

严肃对待，是精心设计的游戏关卡，

准备

1. 职位描述（JD）分析（判断自己的能力是否可以胜任，以及自己需要做的准备）
    - 去看他们的招聘公告
    - PC/H5的前端开发：PC端和移动端的技术栈是不一样的
    - 调试接口：mock数据是最基本的技能
    - 组件库的建立：对原生js/css的理解、之前是否有前端组件库的编写经验、是否通读过其他的UI组件库
    - 系统的优化与重构：可以回答对现在公司的系统的优化和重构的技术和步骤

    任职要求：
    - 面向对象的编程方法、组件化的编程（真正的组件化思想是离不开面向对象的）
    - 一定要准备一个项目的架构分析和模式：目录结构、复用性、模块化、自动化测试等
    - 笔试的回答要让代码体现易读、易维护、高质量、高效率
    - 体现出自己学习能力和追求新知的欲望，能说出一些即可
    - 准备一些技术博客的知识点
    - 熟悉web构建工具，Grunt、Gulp，能够自己搭建前端构建环境（重点看一下gulp，然后看一下grunt和gulp的构建区别，告诉人家你为什么要用gulp而忽略grunt）
    - 动画相关的技术：canvas、svg、transition（css3）、animation（css3）、css3哪些属性可以做cup加速的
    - ES6是js的最新标准

- 对可用性、可访问性等相关知识有实际的了解和实践，如何捕获js异常（例如js的资源加载的错误、js运行异常的捕获）

2. 业务分析或实战模拟

   有时候理论知识丰富，但是实战经验单薄，会导致

   - 如何提升网站性能：DNS预解析
   - 图标字体制作
   - jquery处理大片dom片段，要借助于模版引擎：hanldlebars、template
   - vue写页面的时候，template直接塞页面就可以了，可以直接渲染
   - jquery的：事件委托、事件代理、延迟的方法、ajax

3. 技术栈准备

   每个公司的技术栈使用不同，有的用vue，有的用react等等

   - jquery的源码：核心架构
   - vueJS、react、anuglar，MVVM框架（vue1源码还是比较简洁的，vue2涉及到了很多ssr方面的东西以及静态语法类型检查的东西，读起来不太容易）
   - nodeJS：如果不会就不要写在简历里面
   - sass、less是预编译语言，gulp、grunt是打包工具，npm，webpack（必看）

4. 自我介绍

   - 简历：姓名、年龄、手机、邮箱、籍贯，不用写自我介绍，画蛇添足，简历里面的内容一定要有回答的准备
   - 学历：
   - 工作经历：时间-公司-岗位-职责-技术栈-业绩
   - 开源项目、github和说明

   自我陈述：

   - 把握面试的沟通方向
   - 豁达、自信的适度发挥

页面布局

一、假设高度已知，请写出三栏布局，其中左右两栏宽度各为300px，中间自适应。

1. 浮动布局

   优点：兼容性较好

   缺点：浮动会脱离文档流，如果处理不好上下层的关系和位置，会带来很多问题

2. 绝对定位（但是外面盒子会无法被撑开）

   优点：快捷

   缺点：绝对定位，脱离了文档流，会导致他下面的子元素也是脱离文档流的，所以可使用性较差

3. flex（flex属性是flex-grow, flex-shrink 和 flex-basis的简写，默认值为0 1 auto。后两个属性可选。）（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。）

   优点：flex 布局，现在移动端基本都用这个，总体来说很实用

   缺点：css3，兼容，pc端版本较低

4. 表格布局（display:table）

   优点：兼容性很好

   缺点：如果当其中的某个单元格高度超出时候，剩余两个也会跟着增高

   > display:table => 相当于“table”标签；
   >
   > display:table-row => 相当于“tr”标签；
   >
   > display:table-cell => 相当于“td”标签；

5. 网格布局（display:grid）

   优点：是新的技术，代码量简化很多

   缺点：

6. 圣杯布局
7. 双飞翼布局

二、语义化

1. 不要通篇div，要让标签去掉css样式后，仍然能够结构清晰明了，为了不穿外衣，仍然能够裸奔。
2. 提高用户体验，例如：title，alt 用于解释名词和图片信息的属性
3. 利于SEO，语义化能和搜索引擎建立良好的联系，有利于爬虫抓取更多的有效信息。爬虫依赖于标签来确定上下文和各个关键字的权重。
4. 方便其他设备解析（如屏幕阅读器、盲人阅读器、移动设备）以语义的方式来渲染网页。
5. 便于团队开发和维护，语义化更具可读性，如果遵循W3C标准的团队都遵循这个标准，可以减少差异化，利于规范化。

三、在写页面结构时，我们应该注意什么呢?

1. 尽可能少的使用没有语义的div和span元素。
2. 在对语义要求不明显时，技能使用div也能使用p,那就使用p，以为p默认有上下边距，可以兼容特殊终端。
3. 不要使用纯样式标签，如：b、font、u等，改用css设置。
4. 需要强调的文本，可以包含在strong或者em标签中（浏览器预设样式，能用CSS指定就不用他们），strong默认样式是加粗（不要用b，因为没语义），em是斜体（不用i同b）；
5. 使用表格时，标题要用caption，表头用thead，主体部分用tbody包围，尾部用tfoot包围。表头和一般单元格要区分开，表头标题用th，内容单元格用td；

盒模型（盒模型是CSS的基石，很重要）

> Box 是 CSS 布局的对象和基本单位， 直观点来说，就是一个页面是由很多个 Box 组成的。
>
> 元素的类型和 display 属性，决定了这个 Box 的类型。
>
> 不同类型的 Box， 会参与不同的 Formatting Context（一个决定如何渲染文档的容器），因此Box 内的元素会以不同的方式渲染。

一、谈谈你对CSS盒模型的认识

盒模型的基本概念：标准模型 和IE模型

盒模型 包括：content、padding、border、margin

- 标准模型：标准盒模型的宽高只是内容（content）的宽高
- IE模型：IE盒模型的宽高是内容(content)+填充(padding)+边框(border)的总宽高

二、css如何设置两种模型

```
/* 标准模型 */
box-sizing:content-box;

 /*IE模型*/
box-sizing:border-box;
```

三、JS如何设置获取盒模型对应的宽和高

1. dom.style.width/height

   这种方式只能取到dom元素内联样式所设置的宽高，也就是说如果该节点的样式是在style标签中或外联的CSS文件中设置的话，通过这种方法是获取不到dom的宽高的。

2. dom.currentStyle.width/height

   这种方式获取的是在页面渲染完成后的结果，就是说不管是哪种方式设置的样式，都能获取到。

   但这种方式只有IE浏览器支持。

3. window.getComputedStyle(dom).width/height

   这种方式的原理和2是一样的，这个可以兼容更多的浏览器，通用性好一些。

4. dom.getBoundingClientRect().width/height

   这种方式是根据元素在视窗中的绝对位置来获取宽高的

5. dom.offsetWidth/offsetHeight

   这个就没什么好说的了，最常用的，也是兼容最好的。

四、实例题（根据盒模型解释边距重叠）

什么是边距重叠

父元素没有设置margin-top，而子元素设置了margin-top:20px; 可以看出，父元素也一起有了边距。

边距重叠解决方案（BFC）

- BFC ： Block Formatting Context 直译为“ 块级格式化上下文 ”
- IFC ： Inline formatting context (简称IFC)

BFC的原理（即BFC的渲染规则）

1. 内部的box会在垂直方向，一个接一个的放置
2. 每个元素的margin box的左边，与包含块border box的左边相接触（对于从做往右的格式化，否则相反）
3. 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
4. bfc的区域不会与浮动区域的box重叠
5. bfc是一个页面上的独立的容器，外面的元素不会影响bfc里的元素，反过来，里面的也不会影响外面的
6. 计算bfc高度的时候，浮动元素也会参与计算

如何创建BFC

1. float属性不为none（脱离文档流）
2. position为absolute或fixed
3. display为inline-block,table-cell,table-caption,flex,inine-flex
4. overflow不为visible
5. 根元素

应用场景（https://www.cnblogs.com/lhb25/p/inside-block-formatting-ontext.html）

1. 自适应两栏布局
2. 清除内部浮动
3. 防止垂直margin重叠

DOM事件

> 事件，就是文档或浏览器窗口中发生的一些**特定交互瞬间**。
>
> **事件的三要素**：事件目标、事件处理程序、事件对象
>
> 例如 btn.function(event){} 对该用事件来代码分析，则 btn为事件目标 ，事件处理程序为 function 函数， 事件对象 event

DOM事件的级别（DOM定义标准的一个级别）

1. DOM 0: element.onclick = function(){}
2. DOM 2: element.addEventListener('click', function(){}, false);

   DOM2中，最后的false还是ture，其实是指定了这个响应函数是冒泡还是捕获

   没有DOM1，是因为DOM1标准没有涉及事件相关的改变
3. DOM 3: element.addEventListener('keyup', function(){}, false)

DOM3中，添加了很多事件类型，例如：鼠标事件、键盘事件等等

DOM事件模型（冒泡、捕获）

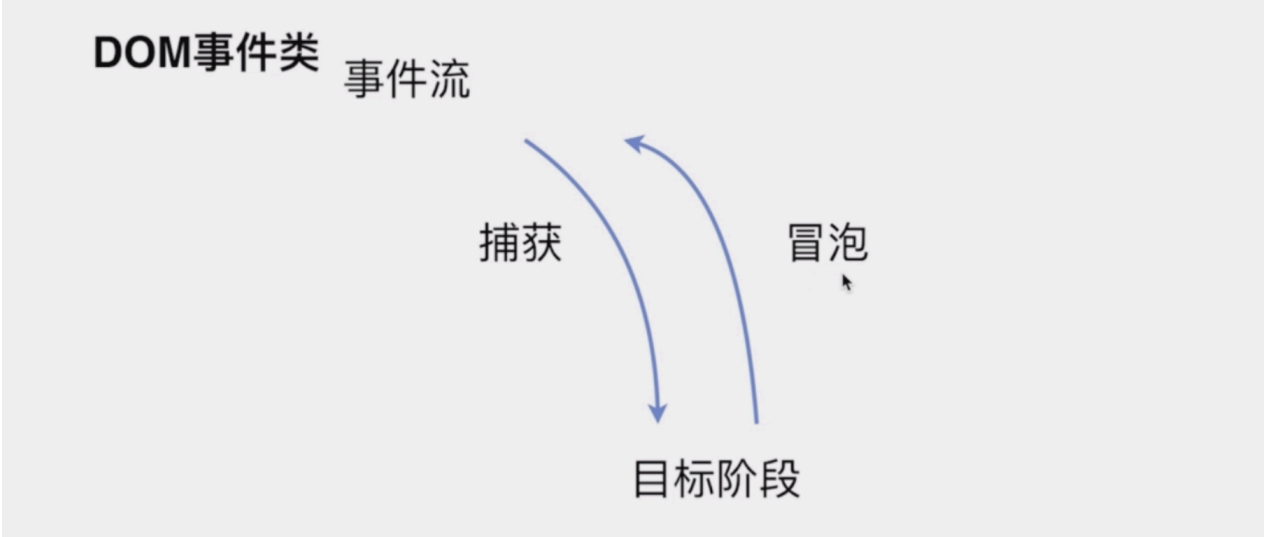
- 冒泡：从下往上，从当前元素也就是目标元素往上。element.addEventListener('click', function(){}, false);
- 捕获：从上往下。element.addEventListener('click', function(){}, true);

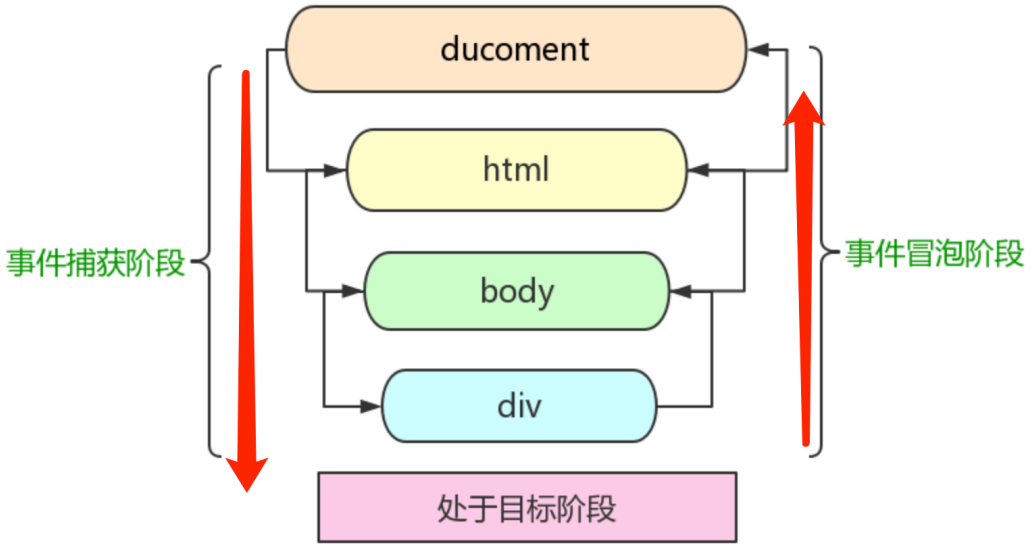
DOM事件流

> 事件流描述的是从页面中接收事件的顺序。事件流可分为冒泡事件流和捕获事件流。
>
> 一个完整的事件流分三个阶段：
>
> - 事件捕获阶段
> - 处于目标阶段
> - 事件冒泡阶段
>
> 首先从事件捕获阶段开始，然后到实际的目标接受事件，最后为事件冒泡阶段。

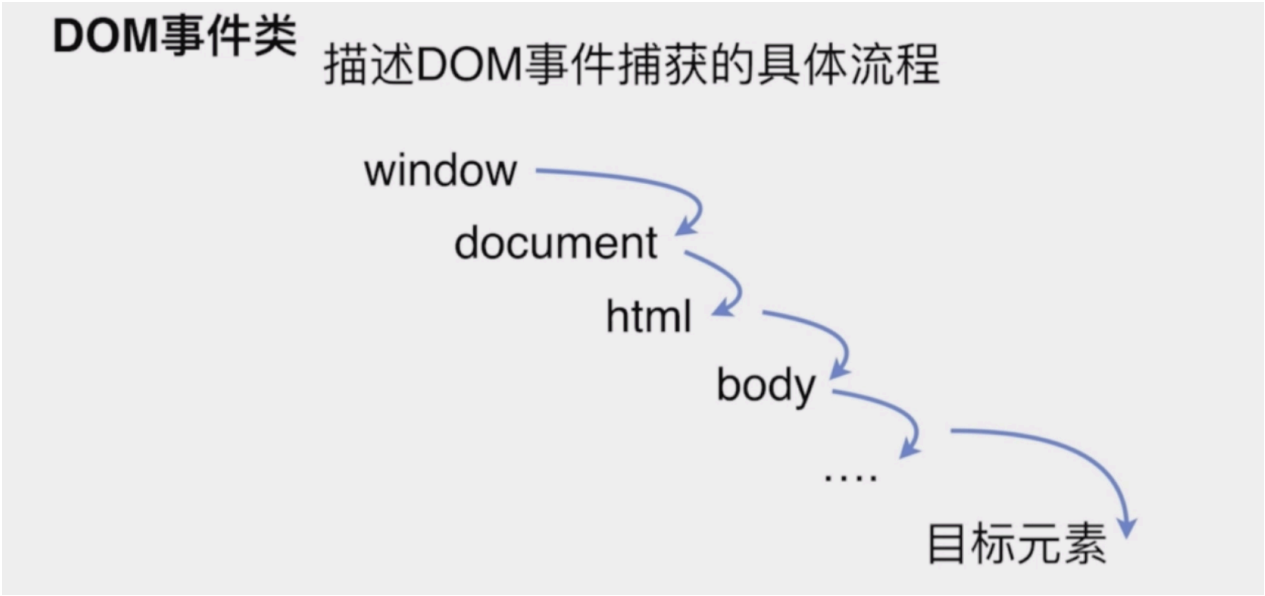
事件流：就是浏览器在为当前页面与用户做交互的过程中，比如我点击了点击鼠标左键，这个左键怎么传到页面上的

事件流的完整阶段：事件通过捕获到达目标元素，这个是捕获的过程。再从目标元素上传到window对象，也就是冒泡的过程





DOM事件捕获的具体流程



Event对象的常见应用

enevt是dom事响应中中非常重要的一个对象，例如想要拿到用户到底集中的是哪个键，想要拿到这种类似的所有东西基本都是从event对象获取到的。

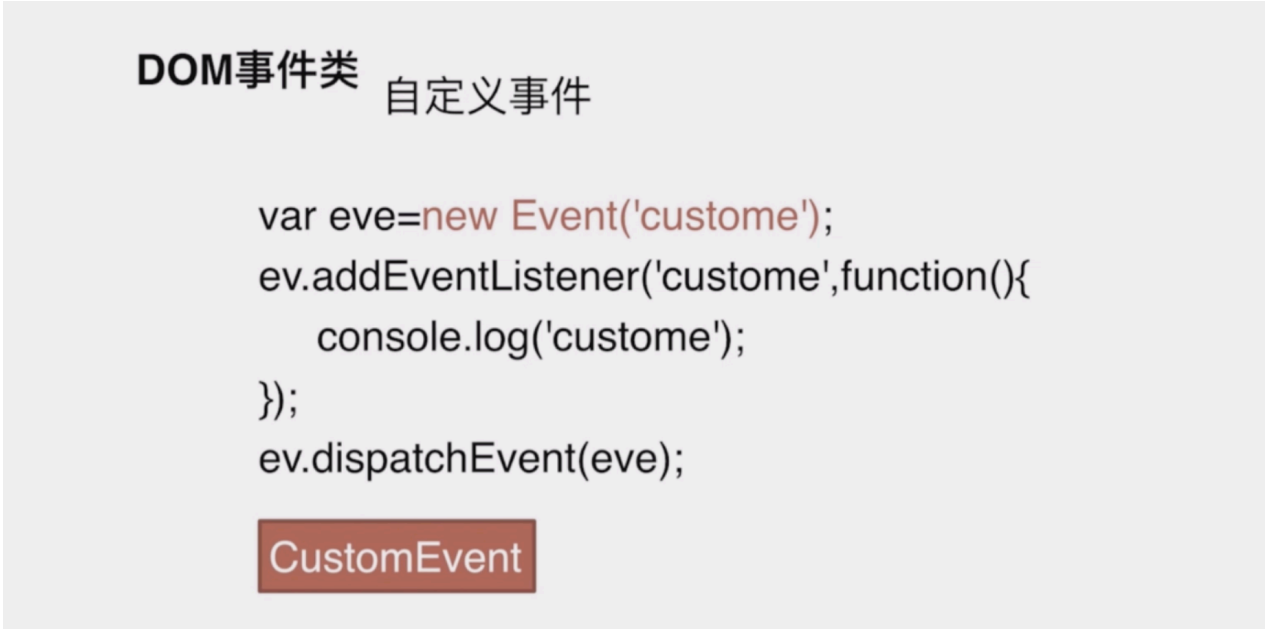
- event.preventDefault() 阻止默认事件，例如：a标签，给a标签绑定了一个click事件，那么在响应函数中，如果设置了preventDefault()，就阻止了链接一个默认跳转的行为。
- event.stopPropagation() 阻止冒泡，有时候业务不需要冒泡，例如子元素有一个点击，父元素有一个点击事件，想让子元素单击时候做一件事情，父元素单击时候做一件事情，如果不阻止冒泡，单击了子元素范围的时候，按照冒泡规则，父元素也会随之被响应。
- event.stopImmediatePropagation() 比如一个按钮绑定了两个click事件，第一个响应是a事件，第二个响应是b事件，我想通过优先级的方式，先响应a，再响应b，想在a点击的时候，不要再执行b，如果在a的响应函数中中加入这样一个方法，就会阻止b事件的执行
- event.currentTarget 面试经常考察，currentTarget表示当前所绑定的事件。
- event.target 面试经常考察，一个for循环给一个dom注册了n个事件，问要怎么优化。其实就是为了

让你用事件代理，把所有的子元素的事件代理转移到父元素上，只绑定一次click事件就可以了。然后做响应的时候，就要判断是哪个子元素做的点击，那么怎么获取这个被点击的元素呢，这时候就需要target上场了。target表示当前被点击的元素

自定义事件（或者叫模拟事件 new Event()）

自定义事件的使用场景：现在有一个按钮，不是一个常规的click按钮，想自己增加一个事件，在别的地方触发这个事件，而不是用回调的方式处理，这时候就需要自定义事件。

通过document.dispatchEvent(eve);来触发执行这个事件



HTTP协议类

HTTP协议的主要特点

1. 简单快速：访问什么输入url就可以了
2. 灵活：在每个http头部协议，都会有一个数据类型，通过一个http协议就可以完成不同数据类型的一个传输
3. 无连接：我连接一次，就会自己断开
4. 无状态：客户端与服务端是两种身份，但从http协议上是没有办法区分你的身份的

HTTP报文的组成部分

1. 请求报文：
    - 请求行：http方法（get/post），页面地址，http协议，版本
    - 请求头：key value值
    - 空行：
    - 请求体：数据部分
2. 响应报文：
    - 状态行：http方法（get/post），页面地址，http协议，版本

- 响应头：key value值
- 空行
- 响应体：数据部分

HTTP方法

1. get——获取资源
2. post——传输资源
3. put——更新资源
4. delete——删除资源
5. head——获得报文首部

POST和GET的区别

get、post的区别

> 此题比较简单，但一定要回答的全面

1. get传参方式是通过地址栏URL传递，是可以直接看到get传递的参数，post传参方式参数URL不可见，get把请求的数据在URL后通过？连接，通过&进行参数分割。psot将参数存放在HTTP的包体内
2. get传递数据是通过URL进行传递，对传递的数据长度是受到URL大小的限制，URL最大长度是2048个字符。post没有长度限制
3. get后退不会有影响，post后退会重新进行提交
4. get请求可以被缓存，post不可以被缓存
5. get请求只URL编码，post支持多种编码方式
6. get请求的记录会留在历史记录中，post请求不会留在历史记录
7. get只支持ASCII字符，post没有字符类型限制

HTTP状态码

1. 1xx：指示信息——表示请求已接收，急需处理
    - 100： **客户端** 应当继续发送请求。
    - 101：服务器已经理解了客户端的请求，并将通过Upgrade 消息头通知客户端采用不同的协议来完成这个请求。
    - 102：由WebDAV（RFC 2518）扩展的状态码，代表处理将被继续执行。
2. 2xx：成功——表示请求已经被成功接收
    - 200：请求成功
    - 206：客户发送了一个带有range头的get请求，服务器完成了它。一般情况，当你发送一个video或者redio的很大的视频或音频文件给服务器时，他会返回206
3. 3xx：重定向——要完成请求必须进行更进一步的操作

- 301：所有的请求页面已经转移到新的url，永久重定向
- 302：所有请求页面已经临时转移到了新的url，临时重定向
- 304：客户端有缓冲的文档并发出了一个条件性的请求，服务器告诉客户，原来缓冲的文档还可以继续使用。

4. 4xx：客户端错误——请求有语法错误或请求无法实现
   - 403：被请求资源禁止被访问
   - 404：请求资源不存在
5. 5xx：服务器错误——服务器未能实现合法的请求
   - 500：服务器错误
   - 503：服务器宕机，过载

HTTP协议持久连接

http协议采用“请求-应答”模式，并且http支持持久连接。

当使用普通模式时，每个请求/应答客户和服务器都要新建立一个连接，完成之后立即断开连接（http协议本身是无连接的协议）

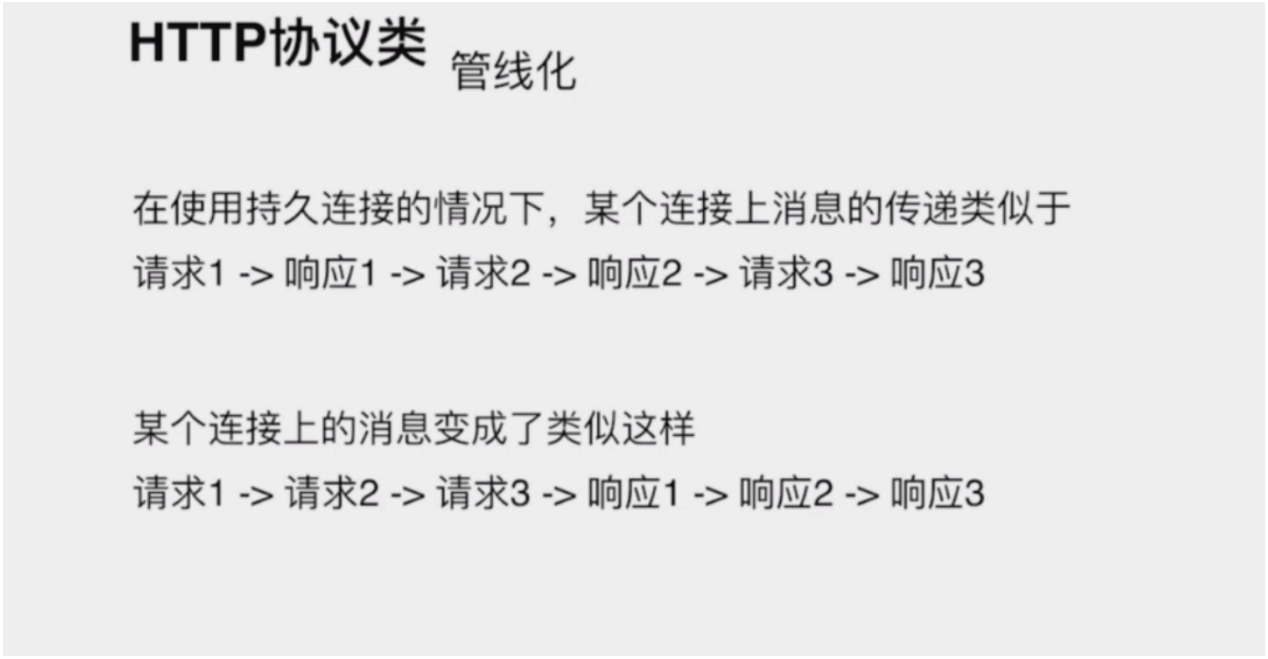
当使用Keep-Alive模式（又称持久连接、连接重用）时，Keep-Alive功能使客户端到服务器端的连接持续有效，当出现对服务器的后记请求时，Keep-Alive功能避免了建立或者重新建立连接。

HTTP协议管线化

管线化机制通过持久连接完成，仅http/1.1支持此技术

get和head请求可以进行管线化，而post则有所限制

由于服务端问题，开启管线化并不会为性能做出很大的贡献，而且很多服务端和代理程序对管线化的支持并不好，所以现在浏览器默认并未开启管线化



原型链

创建对象的几种方法

```
//字面量方法
var o1 = {name: 'o1'};   //字面量的时候，会默认这个对象的原型链指向object
var o2 = new Object({name:'o2'});   //new Object，new一个Object构造函数

//new方法，显示的构造函数来创建对象
var M = function(name){this.name=name}
var o3 = new M('o3');

//Object.create 方法创建
var P={name:"P"};
var o4 = Object.create(P);
```

原型、构造函数、实例、原型链

原型链

构造函数可以通过new 这个构造函数，生成一个实例。

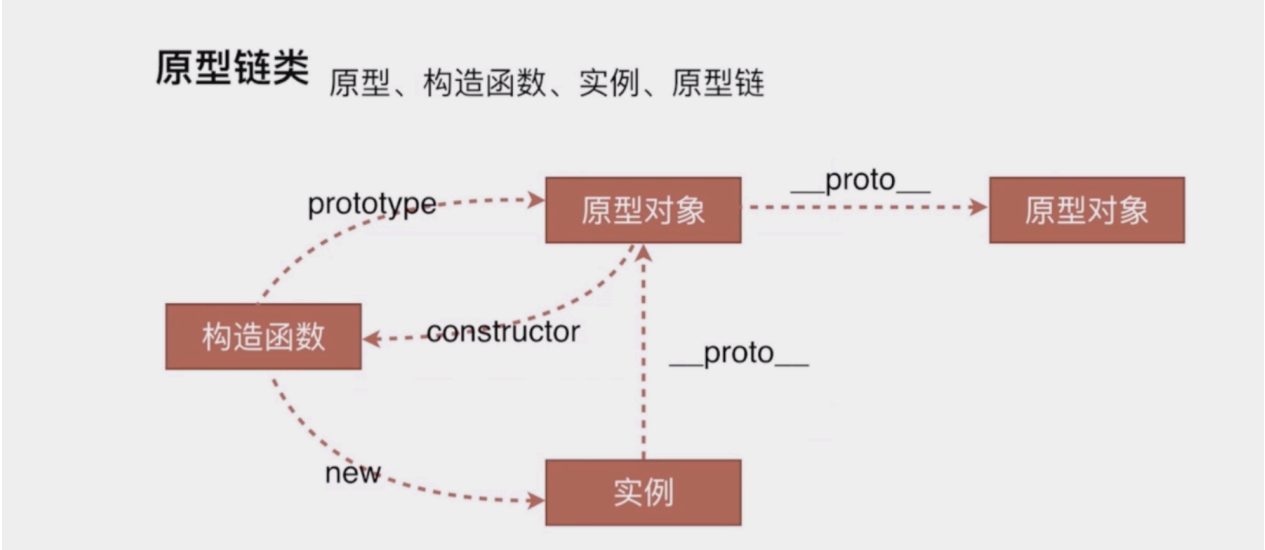
任何函数都可以当作构造函数，只要被new，就会变成构造函数。

所有函数都有prototype属性，这是声明这个函数的时候，js引擎会自动给这个函数加上的这个属性，这个属性会初始化一个空对象，也就是原型对象。

原型对象中，怎么来区分是被哪一个构造函数所引用呢。原型对象中会有一个构造器constructor，他会告诉我们谁是自己的的构造函数。

① `__proto__` 和 `constructor` 属性是 `对象所独有` 的；（原型对象、实例对象、函数对象，万物皆对象）

② `prototype` 属性是 `函数所独有` 的。



原型链：

把一个实例对象，往上找构造这个实例相关联的对象（构造函数），然后再往上找创造这个实例对象的对象的关联对象，以此类推，一直到object.prototype，那么这时候，这个链条就断了。

也就是说，object.prototype这个属性是整个原型链的顶端，到这里终止，那么原型链是通过什么实现向上找的这个过程呢？就是通过prototype和_ _proto_ _这个属性实现的。

```
//new方法，显示的构造函数来创建对象
var M = function(name){this.name=name}
var o3 = new M('o3');

M.prototype===o3.__proto__; //true | o3的__proto__和M的prototype 完全一致，说明M是o3的构造函数
Function.prototype === M.__proto__; //true | M的__proto__和Function的prototype 完全一致，说明Function是M的构造函数，M自身也是一个实例对象，可以往上追溯自己的构造函数
M.prototype.__proto__===Object.prototype; //true
```

原型链的应用场景：

原型对象和原型链之间到底是起了一个什么样的作用呢？

如果我的构造函数中多了很多属性和方法，那么是不是我的实例就可以共用这个东西，当多个实例要共同都拥有一个方法的时候，不可能所有的实例都拷贝一份构造函数，那么这个时候就要考虑将共用的方法加到一个共同的东西上，这个共同的东西就是原型对象。

```
//new方法，显示的构造函数来创建对象
var M = function(name){this.name=name}
var o3 = new M('o3');

//通过原型链增加共用的方法
M.prototype.say = function(){
  console.log("say hi")
}

var o5 = new M("o5");

//打印
console.log(o3.say,o3.name);  //say hi,o3
console.log(o5.say,o5.name);  //say hi,o5
```

instanceof的原理

instanceof的原理就是来判断，实例对象的_ _proto_ _和构造函数的prototype是不是同一个引用，如果是就反回true，不是就返回false

怎么用呢？

```
//new方法，显示的构造函数来创建对象
var M = function(name){this.name=name}
var o3 = new M('o3');

o3 instanceof M;   //true
o3 instanceof Object; //true，也就是只要是在这个原型链上的构造函数，都会被认为是o3的一个构造函数，面试中很容易错
M.prototype.__proto__===Object.prototype; //true
o3.__proto__.constructor==M;  //true
```

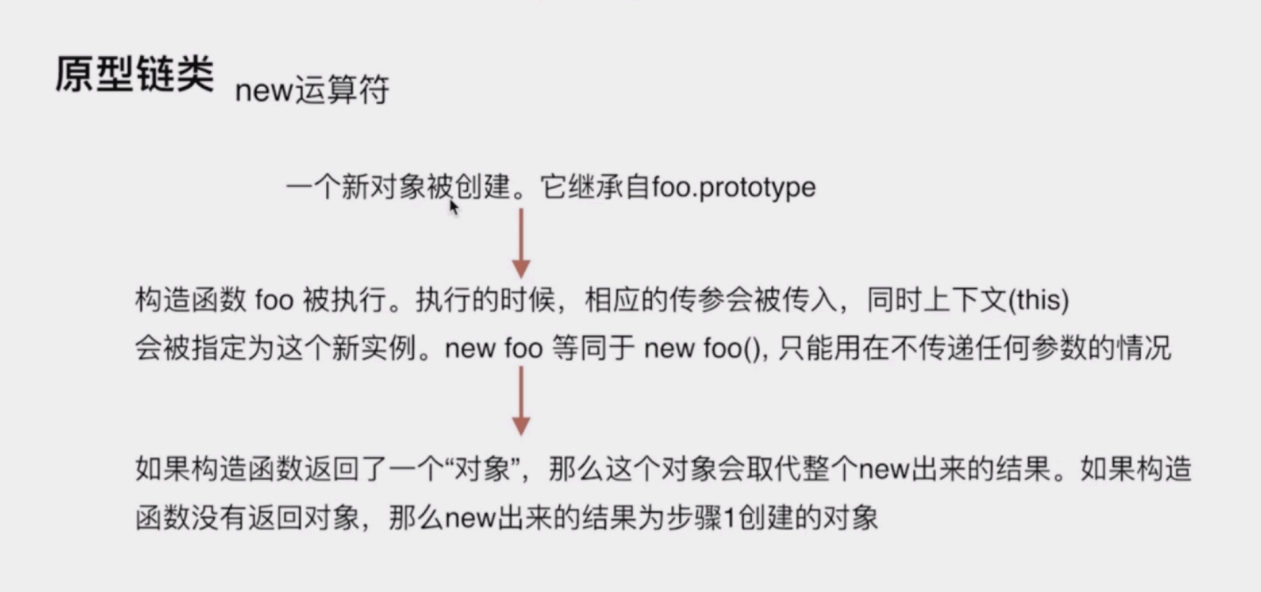
new运算符

new的工作原理（例如new一个构造函数foo，从而创建新实例）

1. new运算符 先创建了一个新的空对象，此时实例对象还没有生成，这个空对象是继承构造函数的原型对象，也就是从这个空对象，指向了原型对象。也就是此时原型链已经被关联上一部分了。
2. 第二步时候，构造函数 foo 被执行。执行时候，相应的船参会被传入，同时上下文（this）会被指定为这个新实例，也就是this 相关联。如果没有参数的时候，new foo等同于new foo()，只能用再不传递任何参数的情况。
3. 如果构造函数返回了一个“对象”，那么这个对象会取代整个new出来的结果。如果构造函数没有返回对象，那么new出来的结果为步骤1创建的对象。

```
//new工作原理，模拟new运算符
var new2 = function(func){
  //第一步，创建一个新对象，空对象要关联构造函数的执行对象func.prototype
  var o = Object.create(func.prototype);
  //第二步，执行构造函数，指针指向
  var k = func.call(o);
  //第三部，判断转移完的是不是对象类型
  if(typeof k ==='object'){
    return k;
  }else {
    return o;
  }
}
var M = function(name){this.name=name}
//使用new2
var o6 =new2(M);

o6.__proto__ === M.prototype  //true
```



call 和 apply

ECMAScript 规范给 `所有函数` 都定义了 `call` 与 `apply` 两个方法，它们的应用非常广泛，作用也是一模一样，只是 `传参形式有区别` 而已，并且他们的**目的**都是为了 `改变this的指向` 。

注意：

call()方法的作用和 apply() 方法类似

- 第一个参数都是 **在*fun*函数运行时指定的 `this` 值**

区别就是，第二个参数（或call中的第二个以及第二个以后的参数）：

- `call()` 方法接受的是**参数列表**（把参数按顺序穿进调用call的方法里）：func.call(`this` , arg1, arg2);
- `apply()` 方法接受的是**一个参数数组**（把参数放在数组里统一传到调用call的方法里）：func.apply(`this` , [arg1, arg2])

总结call、apply、bind：

1. apply 、 call 、bind 三者都是用来改变函数的this对象的指向的；
2. apply 、 call 、bind 三者第一个参数都是this要指向的对象，也就是想指定的上下文；
3. apply 、 call 、bind 三者都可以利用后续参数传参；
4. bind 返回值是对应的函数，便于稍后调用
5. apply 、call 则是立即调用

```
var obj = {
    name: 'boxZhang'
```

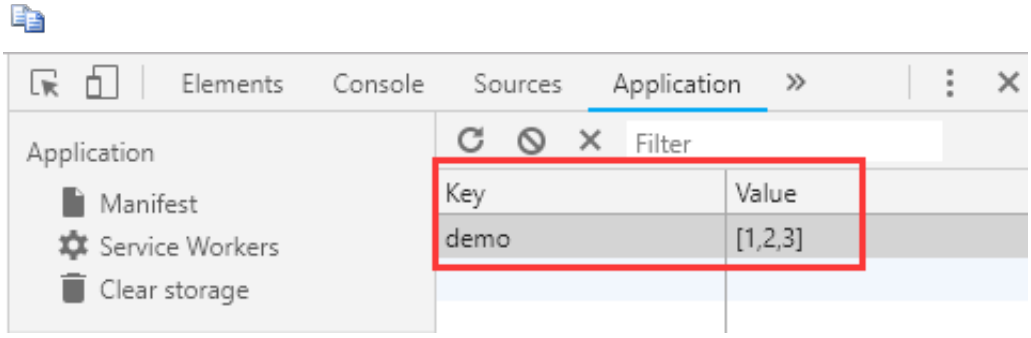
```
}
function func() {
   return this.name;
}
console.log(func.bind(obj)());  //boxZhang 多了个括号，因为bind()要返回一个原函数的拷贝
console.log(func.call(obj));    //boxZhang
console.log(func.apply(obj));   //boxZhang

// 三个输出的都是boxZhang，但是注意看使用 bind() 方法的，他后面多了对括号。
// 也就是说，区别是，bind 方法不会立即执行，而是回调执行的时候，使用 bind() 方法。bind()的返回值是一个函数。
// 而 apply和call 则会立即执行函数。
```

浏览器缓存 sessionStorage、localStorage、cookie、userData

1. sessionStorage 优点：可以临时存储，关闭页面标签自动回收，不支持跨页面交互 缺点：只能作为临时存储，不能存储持久化
2. localStorage 优点：用于长久保存整个网站的数据，保存的数据没有过期时间，直到手动去删除。缺点：存在大小限制，IE8以上的IE版本才支持这个属性；目前所有的浏览器中都会把localStorage的值类型限定为string类型，这个在对我们日常比较常见的JSON对象类型需要一些转换

```
//！！！！！重点
//localStorage/sessionStorage默认只能存储字符串，而实际开发中，我们往往需要存储的数据多为对象类型，那么这里我们就可以在存储时利用json.stringify()将对象转为字符串，而在取缓存时，只需配合json.parse()转回对象即可。
//存
function setLocalStorage(key,val){
    window.localStorage.setItem(key,JSON.stringify(val));
};
//取
function getLocalStorage(key){
    let val = JSON.parse(window.localStorage.getItem(key));
    return val;
};
//测试
setLocalStorage('demo',[1,2,3]);
let  a = getLocalStorage('demo');//[1,2,3]
```



3. cookie 优点：兼容性最好，几乎所有的浏览器都支持 缺点：大小有限制，而且每次发送请求，请求头里会带着cookie一起发过去，现在基本大多数登录的合法性验证都是用cookie验证的
4. userData 优点：出现的时间比sessionStorage要早 缺点：IE专门的存储方式，存储大小有限，单个文件的大小限制是128KB，一个域名下总共可以保存1024KB的文件，文件个数应该没有限制。在受限站点里这两个值分别是64KB和640KB

面向对象

类与实例

类的声明

```
```

生成实例

```
```

类与继承

如何实现继承

说一下继承的几种方式及优缺点?

> 说比较经典的几种继承方式并比较优缺点就可以了

1. 借用构造函数继承，使用call或apply方法，将父对象的构造函数绑定在子对象上
2. 原型继承，将子对象的prototype指向父对象的一个实例
3. 组合继承

原型链继承的缺点

- 字面量重写原型会中断关系，使用引用类型的原型，并且子类型还无法给超类型传递参数。

借用构造函数（类式继承）

- 借用构造函数虽然解决了刚才两种问题，但没有原型，则复用无从谈起。

组合式继承

- 组合式继承是比较常用的一种继承方法，其背后的思路是使用原型链实现对原型属性和方法的继承，而通过借用构造函数来实现对实例属性的继承。这样，既通过在原型上定义方法实现了函数复用，又保证每个实例都有它自己的属性。

深度克隆

JS中的深度克隆，指的是原对象改变了，克隆出来的新对象也不会改变，原对象与新对象是完全独立的关系。

实现深度克隆的原理得从对象是一种引用类型说起

众所周知，对象是一种引用类型，对象的地址指针存放于栈中，而对象实际的数据存放于堆中。

因此当我们简单地执行复制操作时，实际是把地址指针进行了复制操作，因此在对象的实际数据改变之后，新老对象都会受到影响。

那么如何让他不受到影响呢?

答案是利用基本数据类型的特点，基本类型在执行复制操作后，新老数值之间不会互相产生影响

所以，我们要对对象不断进行分解，直到其是基本数据类型之后，再进行复制操作。

1 硬刚法（迭代法，适用于所有）

```
// 方法一 利用迭代
// 思路是：判定要克隆的对象是不是引用类型，如果是引用类型，则继续迭代,如果该项是基本类型，则直接复制。
function deepClone(obj){
    let newObj=Array.isArray(obj)?[]:{} //判断数据类型是否是数组

    if(obj && typeof obj ==="object"){
        for(let key in obj){
            if(obj.hasOwnProperty(key)){  //.hasOwnProperty(propname)方法来检查对象是否有该属性。如果有返回true，反之返回 false。
                newObj[key]=(obj && typeof obj[key]==='object') ? deepClone(obj[key]) : obj[key];
            }
        }
    }
    return newObj
}

let a=[{c:1},2,3,4],
b=deepClone(a);
d=deepClone(a);
a[0]={c:2};

console.log(a,b);
console.log(b==d);  //false，说明克隆出来的每一个新对象都不一样
console.log(b===d); //false
```

2 投机取巧法（Json方法，适用于部分）

```
//方法2  利用JSON对象的方法
//主要是利用JSON.stringify和JSON.parse,这种办法只能实现纯数据的克隆，如果存在function则会直接忽略，不会进行复制操作

function deepClone2(obj){
    return JSON.parse(JSON.stringify(obj))
}

let a1={a: 1, b: function() {}}
b1=deepClone2(a1);
console.log(a1,b1);
```

JS对象深拷贝

> js深拷贝和浅拷贝的概念和区别。
>
> 1. **浅拷贝**：拷贝就是把父对像的属性，全部拷贝给子对象。此时子对象拷贝的是父对象的地址，子父对象相互影响。
> 2. **深拷贝**：就是把父对象的属性中的值拷贝给子对象此时不论父对象如何改变都不会再影响到子对象。

```
//浅拷贝
var father={familyName:"张"}; var son={};
son=father

father.familyName="xiao";
console.log(son.familyName);  //xiao，此时father的值改变时，会影响son的值，这种拷贝方式为浅拷贝，拷贝的是存储变量的地址，会相互影响。
```

深拷贝的几种方法：

1. 借助于JSON对象的两个函数：JSON.stringfy(father); 和 JSON.parse(str);

   ```
   //JSON.stringify(father)将一个json对象转为json字符串
   //JSON.parse(str)将一个json字符串转为json对象
   //它的实现原理是现将对象转为一个基本数据类型，在执行拷贝，不过这个只是适用于json格式的数据对其它情况不一定都能满足，例如对函数就不行。
   //因为基本数据类型拷贝时候只是拷贝的数据，而数组、对象等数据类型拷贝的时候，拷贝的是对象，是指针地址。测试如下：

   var father={familyName:"张"};var son={};
   son = JSON.parse(JSON.stringify(father));
   father.familyName="李";
   son.familyName; //输出：“张”
   ```

2. 第四步：借助于for 循环实现数组的深拷贝

   ```
   var father = [1,2,3,4,5]
   var son = copyArr(father)
   ```

```
function copyArr(father) {
    let res = []
    for (let i = 0; i < father.length; i++) {
     res.push(father[i])
    }
    return res
}
console.log(father);  //[1,2,3,4,5]
console.log(son); //[1,2,3,4,5]
console.log(son==father); //false
father[2] = 5
console.log(father);  //[1, 2, 5, 4, 5]
console.log(son); //[1, 2, 3, 4, 5]
```

3. 借助于slice 方法实现数组的深拷贝

```
//arr.slice(start,end)//start==>开始位置下标    end==>结束位置下标，如果没有end就说明
var father = [1,2,3,4,5];
var son = father.slice(0);
console.log(father);  //[1, 2, 3, 4, 5]
console.log(son); //[1, 2, 3, 4, 5]
father[2] = 5;
console.log(father);  //[1, 2, 5, 4, 5]
console.log(son); //[1, 2, 3, 4, 5]
```

4. 借助于concat 方法实现数组的深拷贝

```
var father = [1,2,3,4,5];
var son = father.concat();
console.log(father);  //[1, 2, 3, 4, 5]
console.log(son); //[1, 2, 3, 4, 5]
father[2] = 5;
console.log(father);  //[1, 2, 5, 4, 5]
console.log(son); //[1, 2, 3, 4, 5]
```

5. 使用ES6扩展运算符实现数组的深拷贝

```
var father = ['张三','李四','刘德华','周润发']
var [...son] = father;  //ES6扩展运算符（spread）是三个点（...）。它好比 rest 参数的逆运算，将一个数组转为用逗号分隔的参数序列。
console.log(father);  // ["张三", "李四", "刘德华", "周润发"]
console.log(son); // ["张三", "李四", "刘德华", "周润发"]
father[2] = 5;  // ["张三", "李四", "刘德华", "周润发"]
console.log(father);  // ["张三", "李四", 5, "周润发"]
console.log(son); // ["张三", "李四", "刘德华", "周润发"]
```

JS数组去重

```
//数组去重 方法一
```

```
Array.prototype.unique1 = function () {
  var n = []; //一个新的临时数组
  for (var i = 0; i < this.length; i++) //遍历当前数组
    {
      //stringObject.indexOf(searchvalue,fromindex) 方法可返回某个指定的字符串searchvalue值在字符串中首次出现的位置。如果没有就返回-1，fromindex 规定在字符串中开始检索的位置。它的合法取值是 0 到 stringObject.length - 1。
      //如果当前数组的第i已经保存进了临时数组，那么跳过，
      //否则把当前项push到临时数组里面
      console.log(n)
      if (n.indexOf(this[i]) == -1) n.push(this[i]);
    }
  return n;
}
var a1=["a","b",3,"dd","a","c",4,3,9,7,5,2]
a1.unique1()

//控制台输出
[]
VM1943:8 ["a"]
VM1943:8 (2) ["a", "b"]
VM1943:8 (3) ["a", "b", 3]
VM1943:8 (4) ["a", "b", 3, "dd"]
VM1943:8 (4) ["a", "b", 3, "dd"]
VM1943:8 (5) ["a", "b", 3, "dd", "c"]
VM1943:8 (6) ["a", "b", 3, "dd", "c", 4]
VM1943:8 (6) ["a", "b", 3, "dd", "c", 4]
VM1943:8 (7) ["a", "b", 3, "dd", "c", 4, 9]
VM1943:8 (8) ["a", "b", 3, "dd", "c", 4, 9, 7]
VM1943:8 (9) ["a", "b", 3, "dd", "c", 4, 9, 7, 5]
(10) ["a", "b", 3, "dd", "c", 4, 9, 7, 5, 2]
```

```
//数组去重 方法二
Array.prototype.unique2 = function()
{
  var n = {},r=[]; //n为hash表，r为临时数组
  for(var i = 0; i < this.length; i++) //遍历当前数组
    {
      if (!n[this[i]]) //如果hash表中没有当前项
      {
        n[this[i]] = true; //存入hash表
console.log(n)
        r.push(this[i]); //把当前数组的当前项push到临时数组里面
      }
    }
  return r;
}

var a2=["a","b",3,"dd","a","c",4,3,9,7,5,2]
a2.unique2()

//控制台输出
```

```
{a: true}
VM1523:9 {a: true, b: true}
VM1523:9 {3: true, a: true, b: true}2: true3: true4: true5: true7: true9: truea:
trueb: truec: truedd: true__proto__: Object
VM1523:9 {3: true, a: true, b: true, dd: true}
VM1523:9 {3: true, a: true, b: true, dd: true, c: true}
VM1523:9 {3: true, 4: true, a: true, b: true, dd: true, c: true}
VM1523:9 {3: true, 4: true, 9: true, a: true, b: true, dd: true, c: true}2: true3:
true4: true5: true7: true9: truea: trueb: truec: truedd: true__proto__: Object
VM1523:9 {3: true, 4: true, 7: true, 9: true, a: true, b: true, dd: true, c: true}
VM1523:9 {3: true, 4: true, 5: true, 7: true, 9: true, a: true, b: true, dd: true, c:
true}
VM1523:9 {2: true, 3: true, 4: true, 5: true, 7: true, 9: true, a: true, b: true, dd:
true, c: true}
(10) ["a", "b", 3, "dd", "c", 4, 9, 7, 5, 2]
```

```
//数组去重 方法三
Array.prototype.unique3 = function()
{
  var n = [this[0]]; //结果数组
  for(var i = 1; i < this.length; i++) //从第二项开始遍历
    {
      //如果当前数组的第i项在当前数组中第一次出现的位置不是i,
      //那么表示第i项是重复的，忽略掉。否则存入结果数组
      console.log(n)
      if (this.indexOf(this[i]) == i) n.push(this[i]);
    }
  return n;
}
var a3=["a","b",3,"dd","a","c",4,3,9,7,5,2]
a3.unique3()

//控制台输出
["a"]
VM2061:9 (2) ["a", "b"]
VM2061:9 (3) ["a", "b", 3]
VM2061:9 (4) ["a", "b", 3, "dd"]
VM2061:9 (4) ["a", "b", 3, "dd"]
VM2061:9 (5) ["a", "b", 3, "dd", "c"]
VM2061:9 (6) ["a", "b", 3, "dd", "c", 4]
VM2061:9 (6) ["a", "b", 3, "dd", "c", 4]
VM2061:9 (7) ["a", "b", 3, "dd", "c", 4, 9]
VM2061:9 (8) ["a", "b", 3, "dd", "c", 4, 9, 7]
VM2061:9 (9) ["a", "b", 3, "dd", "c", 4, 9, 7, 5]
(10) ["a", "b", 3, "dd", "c", 4, 9, 7, 5, 2]
```

递归

递归,就是在运行的过程中调用自己。

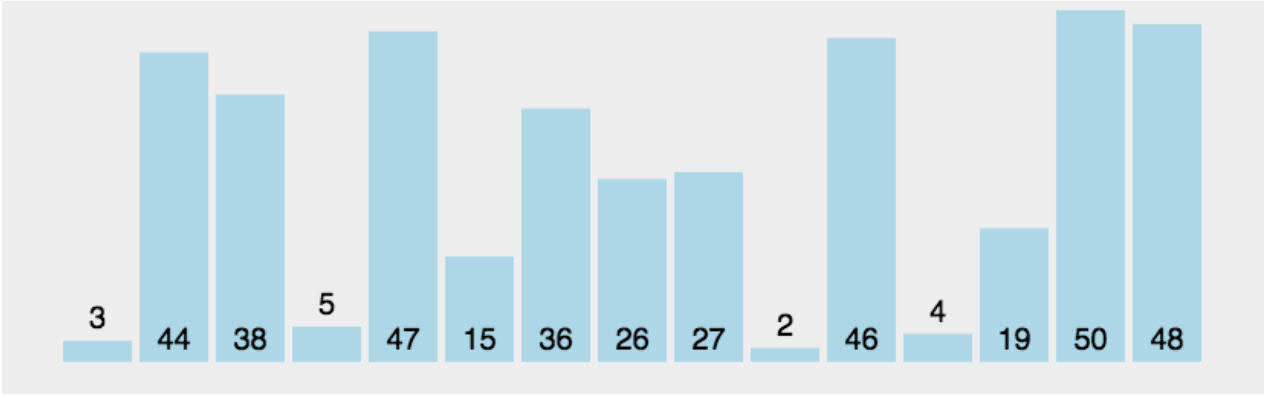
用js递归的方式写1到100求和?

> 递归我们经常用到，vue在实现双向绑定进行数据检验的时候用的也是递归，但要我们面试的时候手写一个递归，如果对递归的概念理解不透彻，可能还是会有一些问题。

```
function add(num1,num2){
  var num = num1+num2;
        if(num2+1>100){
   return num;
  }else{
    return add(num,num2+1)
        }
 }
var sum =add(1,2);
```

排序

冒泡排序




```
//冒泡排序 即比较相邻的元素，如果第一个比第二个大，就交换他们两个。
var canArr =
[0,4,2,6,7,8,2,14,57,8,99,0,45,32,12,1,1,1,1,45,0,9,8,7,6,5,4,3,2,115,67,68,56,55,43,2
1];
console.log(canArr);

function bubble(arr) {
  var s ;
  //从小到大
  // for (var i =0;i<arr.length;i++) {
  //   for (var j = 0; j < arr.length; j++) {
  // if (arr[j] > arr[j + 1]){
  // s = arr[j];
```

```
  // arr[j]=arr[j+1];
  // arr[j+1]=s;
  // }
  //    }
  // }

  //从大到小
  for (var i =0; i<arr.length; i++) {
    for (var j = 0; j < arr.length; j++) {
      if (arr[j] < arr[j + 1]){
        s = arr[j];
        arr[j]=arr[j+1];
        arr[j+1]=s;
      }
    }
  }
  return arr;
}
console.log("冒泡排序:");
console.log(bubble(canArr));
```

插入排序

```
//插入排序，每次处理就是将无序数列的第一个元素与有序数列的元素从后往前逐个进行比较,找出插入位置,将该元素插入到有序数列的合适位置中
var canArr =
[0,4,2,6,7,8,2,14,57,8,99,0,45,32,12,1,1,1,1,45,0,9,8,7,6,5,4,3,2,115,67,68,56,55,43,21];
console.log(canArr);

//插入排序
function insert(arr){
  var s;
  //升序
  for (var i = 1; i < arr.length; i++) {
    for (var j = i; j > 0; j--) {
      if (arr[j] < arr[j - 1]) {
        s=arr[j];
        arr[j]=arr[j-1];
        arr[j-1]=s ;
        //console.log(arr);//可以打印出来每一个改变的步骤
      }
    }
  }
  //降序
  for (var i = 1; i < arr.length; i++) {
    for (var j = i; j > 0; j--) {
      if (arr[j] < arr[j - 1]) {
        s=arr[j];
```

```
        arr[j]=arr[j-1];
        arr[j-1]=s ;
        //console.log(arr);//可以打印出来每一个改变的步骤
      }            }
  }
  return arr;
}
console.log("插入排序:");
console.log(insert(quchArr));
```

js操作获取和设置cookie

```
//创建cookie
function setCookie(name, value, expires, path, domain, secure) {
  var cookieText = encodeURIComponent(name) + '=' + encodeURIComponent(value);
  if (expires instanceof Date) { cookieText += '; expires=' + expires;
                                }
  if (path) {
    cookieText += '; expires=' + expires;
  }
  if (domain) {
    cookieText += '; domain=' + domain;
  }
  if (secure) {
    cookieText += '; secure';
  }
  document.cookie = cookieText;
}
//获取cookie
function getCookie(name) {
  var cookieName = encodeURIComponent(name) + '=';
  var cookieStart = document.cookie.indexOf(cookieName);
  var cookieValue = null;
  if (cookieStart > -1) {
    var cookieEnd = document.cookie.indexOf(';', cookieStart);
    if (cookieEnd == -1) {
      cookieEnd = document.cookie.length;
    }
  }
}
```

写一个通用的事件侦听器函数?

```
// event(事件)工具集，来源：github.com/markyun
markyun.Event = {
  // 页面加载完成后
```

```
readyEvent : function(fn) {
  if (fn==null) {
    fn=document;
  }
  var oldonload = window.onload;
  if (typeof window.onload != 'function') {
    window.onload = fn;
  } else {
    window.onload = function() {
      oldonload();
      fn();
    };
  }
},

// 视能力分别使用dom0||dom2||IE方式 来绑定事件
// 参数:  操作的元素,事件名称 ,事件处理程序
addEvent : function(element, type, handler) {
  if (element.addEventListener) {
    //事件类型、需要执行的函数、是否捕捉
    element.addEventListener(type, handler, false);
  } else if (element.attachEvent) {
    element.attachEvent('on' + type, function() { handler.call(element);
                                                  });
  } else {
    element['on' + type] = handler;
  }
},
// 移除事件
removeEvent : function(element, type, handler) {
  if (element.removeEnentListener) {
    element.removeEnentListener(type, handler, false);
  } else if (element.datachEvent) {
    element.detachEvent('on' + type, handler);
  } else {
    element['on' + type] = null;
  }
},
// 阻止事件 (主要是事件冒泡，因为IE不支持事件捕获)
stopPropagation : function(ev) {
  if (ev.stopPropagation) {
    ev.stopPropagation();
  } else {
    ev.cancelBubble = true;
  }
},
// 取消事件的默认行为
preventDefault : function(event) {
  if (event.preventDefault) {
    event.preventDefault();
  } else {
    event.returnValue = false;
```

```
    }
  },
  // 获取事件目标
  getTarget : function(event) {
    return event.target || event.srcElement;
  },
  // 获取event对象的引用，取到事件的所有信息，确保随时能使用event;
  getEvent : function(e) {
    var ev = e || window.event;
    if (!ev) {
      var c = this.getEvent.caller;
      while (c) {
        ev = c.arguments[0];
        if (ev && Event == ev.constructor) {
          break;
        }
        c = c.caller;
      }
    }
    return ev;
  }
};
```

window.onload和document.ready的区别？哪一个先执行?

- 一般情况一个页面响应加载的顺序是，域名解析-加载html-加载js和css-加载图片等其他信息。
- window.onload是在DOM文档树加载完和所有文件加载完之后执行一个函数，也就是在页面响应加载的顺序中的“加载图片等其他信息”之后，可以操作DOM。只能执行一次，如果有多个，那么第一次的执行会被覆盖
- document.ready是在DOM加载完成后就可以可以对DOM进行操作，也就是在在“加载js和css”和“加载图片等其他信息”之间，就可以操作DOM了。可以执行多次

  所以，document.ready函数只需对 DOM 树的等待，而无需对图像或外部资源加载的等待，从而执行起来更快

null和undefined的区别

`null` **表示没有对象，即该处不应该有值**。代表“空值”，代表一个空对象指针，使用typeof运算得到“object”，所以认为它是一个特殊的对象值。

1） 作为函数的参数，表示该函数的参数不是对象

2） 作为对象原型链的终点

`undefined` **表示缺少值，即此处应该有值，但没有定义**。当一个声明了一个变量未初始化时，得到的就是undefined

1）定义了形参，没有传实参，显示undefined

2）对象属性名不存在时，显示undefined

3）函数没有写返回值，即没有写return，拿到的是undefined

4）写了return，但没有赋值，拿到的是undefined

null和undefined转换成number数据类型

`null` 默认转成 `0`

`undefined` 默认转成 `NaN`

通信

一个页面从输入 URL 到页面加载显示完成，这个过程中都发生了什么？

1. 浏览器根据请求的URL交给DNS域名解析，找到真实IP；
2. 浏览器根据 IP 地址向服务器发起 TCP 连接，与浏览器建立 TCP 三次握手 a.客户端向服务器发送一个建立连接的请求 b.服务器接到请求后发送同意连接的信号 c.客户端接到同意连接的信号后，再次向服务器发送了确认信号，然后客户端与服务器的连接建立成功
3. 浏览器发送HTTP请求 浏览器根据 URL 内容生成 HTTP 请求，请求中包含请求文件的位置、请求文件的方式等等；
4. 服务器处理请求并返回HTTP报文（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。）： a服务器接到请求后，会根据 HTTP 请求中的内容来决定如何获取相应的 HTML 文件； b.服务器将得到的 HTML 文件发送给浏览器； c.在浏览器还没有完全接收 HTML 文件时便开始渲染、显示网页； d在执行 HTML 中代码时，根据需要，浏览器会继续请求图片、CSS、JavsScript等文件，过程同请求 HTML 。
5. 断开连接

性能优化

你都用过那种性能优化的方法?

1. 尽可能的减少http请求 使用CSS Sprites；JS、CSS源码压缩；图片大小控制合适；启用Gzip压缩，CDN托管，data缓存；图片服务器
2. 避免在CSS中使用 Expression Expression（css表达式)又称Dynamic properties(动态属性)
3. 添加expire/Cache-Control头
4. 少用全局变量、缓存DOM节点查找的结果。减少IO读取操作。
5. 图片预加载，将样式表放在顶部，将脚本放在底部，加上时间戳。
6. 当需要设置的样式很多时设置className而不是直接操作style。
7. 用innerHTML代替DOM操作，减少DOM操作次数，优化javascript性能。
8. 前端模板 JS+数据，减少由于HTML标签导致的带宽浪费，前端用变量保存AJAX请求结果，每次操作本地变量，不用请求，减少请求次数

安全

算法

控制台

```
ls   |查看当前目录
cd   |进入目录
touch index.html   |创建index.html文件
open -a atom index.html |打开index.html文件, open 指定编辑器
```

1、面试题基础

2、老内容面试

3、新内容vue

4、问题集成