前端面试宝典

HTML

目录

—

HTML..... 5

1.H5的新特性有哪些..... 5

2.两种实现前端路由的方式..... 6

3.websocket..... 7

4.webSocket如何兼容低浏览器？(阿里)..... 8

1.Adobe Flash Socket 、..... 8

2.ActiveX HTMLFile (IE) 、..... 8

3. 基于 multipart 编码发送 XHR 、..... 8

4.基于长轮询的XHR..... 8

1. WebSocket、SharedWorker；..... 10

2. 也可以调用localstorge、cookies等本地存储方式；..... 10

1. 结构与样式分离..... 12

2. 代码语义性好..... 12

3. 更符合HTML标准规范..... 12

4. SEO友好..... 12

二CSS..... 18

1.flex..... 18

2.tabel..... 19

3.absolute+transform 水平垂直居中..... 19

4.webkit-box..... 20

1.table(父级元素)与tabel-cell（两个子集元素）..... 21

2.flex(父级元素)+flex :1（右边子元素）..... 21

3.左边定宽，并且左浮动；右边设置距离左边的宽度..... 21

4.左边定宽，左边设置position:absolute；右边设置距离左边的宽度..... 21

1. 采用浮动布局（左边左浮动，右边右浮动，中间margin：0 宽度值）..... 21

2. 绝对定位方式（左右绝对定位，左边left0右边right0，中间上同）..... 21

1.在浮动元素末尾添加空标签清除浮动 clear:both （缺点：增加无意义标签）..... 22

2.给父元素设置 overflow:auto属性..... 22

3.after伪元素..... 23

方案一：CSS3 vw 单位..... 23

方案二：设置垂直方向的padding撑开容器..... 23

方案三：利用伪元素的 margin(padding)-top 撑开容器..... 23

1、map+area或者svg..... 23

2、border-radius..... 23

3、纯js实现 需要求一个点在不在圆上简单算法、获取鼠标坐标等等..... 23

二、多元素的组合选择器..... 24
三、CSS 2.1 属性选择器..... 24
四、CSS 2.1中的伪类..... 25
五、 CSS 2.1中的伪元素..... 25
六、CSS 3的同级元素通用选择器..... 25
七、CSS 3 属性选择器..... 26
八、CSS 3中与用户界面有关的伪类..... 26
九、CSS 3中的结构性伪类..... 26
十、CSS 3的反选伪类..... 27
十一、CSS 3中的 :target 伪类..... 27
三 JavaScript..... 49
1.js基础..... 49
第一种（函数声明）：..... 69
第二种（函数表达式）：..... 69
第三种（函数对象方式）：..... 69
方法一:..... 76

方法二：..... 77
1 undefined（因为是window对象执行了func函数）;..... 88
2.DOM相关..... 108
3.BOM相关..... 112
4.JS高级..... 112
1.函数高级..... 112
2.对象高级..... 117
3.线程机制..... 122
5.ES6相关..... 123
1.谈一谈let与var和const的区别？..... 123
2.箭头函数..... 123
6.跨域..... 128
1.JSONP..... 128
2.document.domain 跨子域..... 128
3. iframe..... 128
4.postMessage()..... 128

5.CORS..... 128
6.服务端代理..... 129
7.Ajax..... 129
1.ajax请求和原理..... 129
2.fetch和Ajax有什么不同..... 130
3.GET和POST的区别..... 131
4.GET,POST,PUT,Delete..... 131
1.node核心..... 134
2.单线程优点..... 134
3.Node.js是如何做到I/O的异步和非阻塞的呢..... 134
4.并行与并发，进程与线程..... 135
5.谈谈Nodejs优缺点..... 135
m1 : m1,..... 136
m2 : m2..... 136
四 构建工具..... 138

1.webpack相关问题..... 138
1.loader和plugin区别..... 138
2.什么是 chunk..... 138
3.如何开发一个loader，原理是啥..... 139
4.打包原理..... 139
5.webpack和gulp的区别..... 141
6.如何写一个 plugin..... 141
7.webpack打包后文件体积过大怎么办？..... 142
五 前端框架..... 142
1.jQuery..... 142
2.angular..... 148
1.使用过angular吗？angular中的过滤器是干什么用的..... 148
3.react..... 148
1.react和vue的区别..... 148
2.react的优缺点..... 149
4.vue..... 158

1.vue 双向绑定底层实现原理..... 158
2.vue 虚拟DOM和react 虚拟DOM的区别..... 158
3.v-show和v-if区别..... 159
4.vue组件通信..... 159
5.你如何评价vue..... 159
6.说说你对MVVM的理解..... 160
7.为什么选择vue..... 160
8.vue中mixin与extend区别..... 161
9.双向绑定和单向数据绑定的优缺点..... 161
10.VueJS 开发常见问题集锦..... 162
11.谈谈你对组件的理解..... 162
六 移动端问题..... 163
1.说说你知道的移动端web的兼容性bug..... 163
2.有哪些多屏适配方案..... 164
3.移动端的布局用过媒体查询吗？..... 164
4.移动端（Android IOS）怎么做好用户体验?..... 165
5.移动端最小触控区域是多大？..... 165

七 大厂面试题..... 165

1.美团面试题..... 165

2.饿了么面试题..... 168

3.糯米面试题..... 171

八 扩展性问题..... 172

1.首先应该是一个优秀的程序员..... 172

2.其次是努力使自己成为某一领域的技术专家..... 172

3.通过技术更好的服务于团队和业务..... 172

4.提高沟通能力，团队协作，发现问题，解决问题，总结问题能力..... 172

5.写写博客，输出就是最好的学习..... 172

6.提升个人前端的工作效率和工作质量..... 172

7.关注前端前言技术和发展方向，通过新技术服务团队和业务..... 172

8.一专多长..... 172

九 网络知识相关..... 182

1.http状态码..... 182

2.http报头有哪些..... 187

3.代理和反向代理..... 188
4.网络优化/性能优化..... 188
5.首屏优化..... 189
6.网页验证码是干嘛的，是为了解决什么安全问题。..... 190
7.为什么利用多个域名来存储网站资源会更有效？..... 190
8.请谈一下你对网页标准和标准制定机构重要性的理解。..... 190
9.前端开发的优化问题。..... 190
十 地址收藏..... 191
1. 前端面试经典题目合集_牛客网..... 191
2.鹅厂(腾讯)web前端面试居然还有套路？..... 191
3.面试那些事儿..... 191
4.前沿开发团队的面试过程..... 191
5.向那些年我们经历的一面N面致敬..... 192
6.五个典型的JavaScript面试题..... 192
7.征服前端面试..... 192
8.阿里面经（附回答）面试必看..... 192

一 HTML

1.H5新特性

1.H5的新特性有哪些

1.画布(Canvas) API

2.地理(Geolocation) API

3.音频、视频API(audio,video)

4.localStorage和sessionStorage

5.webworker, websocket

6.header,nav,footer,aside,article,section

7.web worker是运行在浏览器后台的js程序，他不影响主程序的运行，是另开的一个js线程，可以用这个线程执行复杂的数据操作，然后把操作结果通过postMessage传递给主线程，这样在进行复杂且耗时的操作时就不会阻塞主线程了。

2.两种实现前端路由的方式

HTML5 History两个新增的API：history.pushState 和 history.replaceState，两个 API 都会操作浏览器的历史记录，而不会引起页面的刷新。

Hash就是url 中看到 # ,我们需要一个根据监听哈希变化触发的事件(hashchange) 事件。我们用 window.location 处理哈希的改变时不会重新渲染页面，而是当作新页面加到历史记录中，这样我们跳转页面就可以在 hashchange 事件中注册 ajax 从而改变页面内容。 可以为hash的改变添加监听事件：

window.addEventListener("hashchange", funcRef, false)

优点

从性能和用户体验的层面来比较的话，后端路由每次访问一个新页面的时候都要向服务器发送请求，然后服务器再响应请求，这个过程肯定会有延迟。而前端路由在访问一个新页面的时候仅仅是变换了一下路径而已，没有了网络延迟，对于用户体验来说会有相当大的提升。

前端路由的优点有很多，比如页面持久性，像大部分音乐网站，你都可以在播放歌曲的同时，跳转到别的页面而音乐没有中断，再比如前后端彻底分离。 开发一个前端路由，主要考虑到页面的可插拔、页面的生命周期、内存管理等。

缺点

使用浏览器的前进，后退键的时候会重新发送请求，没有合理地利用缓存。

History interface提供了两个新的方法：pushState(), replaceState()使得我们可以对浏览器历史记录栈进行修改：

window.history.pushState(stateObject, title, URL)

window.history.replaceState(stateObject, title, URL)

3.websocket

WebSocket 使用ws或wss协议，Websocket是一个持久化的协议，相对于HTTP这种非持久的协议来说。WebSocket API最伟大之处在于服务器和客户端可以在给定的时间范围内的任意时刻，相互推送信息。WebSocket并不限于以Ajax(或XHR)方式通信，因为Ajax技术需要客户端发起请求，而WebSocket服务器和客户端可以彼此相互推送信息；XHR受到域的限制，而WebSocket允许跨域通信。

```
// 创建一个Socket实例
var socket = new WebSocket('ws://localhost:8080');
// 打开Socket
socket.onopen = function(event) {
 // 发送一个初始化消息
 socket.send('I am the client and I\'m listening!');
 // 监听消息
 socket.onmessage = function(event) {
  console.log('Client received a message',event);
 };
 // 监听Socket的关闭
 socket.onclose = function(event) {
  console.log('Client notified socket has closed',event);
 };
 // 关闭Socket....
 //socket.close()
```

};

4.webSocket如何兼容低浏览器？(阿里)

1.Adobe Flash Socket 、

2.ActiveX HTMLFile (IE) 、

3. 基于 multipart 编码发送 XHR 、

4.基于长轮询的 XHR

2.浏览器渲染原理解析

1、首先渲染引擎下载HTML，解析生成DOM Tree

2、遇到css标签或JS脚本标签就新起线程去下载他们，并继续构建DOM。（其中css是异步下载同步执行）浏览器引擎通过 DOM Tree 和 CSS Rule Tree 构建 Rendering Tree

3、 通过 CSS Rule Tree 匹配 DOM Tree 进行定位坐标和大小，这个过程称为 Flow 或 Layout 。

4、最终通过调用Native GUI 的 API 绘制网页画面的过程称为 Paint 。

当用户在浏览网页时进行交互或通过 js 脚本改变页面结构时，以上的部分操作有可能重复运行，此过程称为 Repaint 或 Reflow。 重排是指dom树发生结构变化后，需要重新构建dom结构。 重绘是指dom节点样式改变，重新绘制。 重排一定会带来重绘，重绘不一定有重排。

如何减少浏览器重排：将需要多次重排的元素，position属性设为absolute或fixed，这样此元素就脱离了文档流，它的变化不会影响到其他元素。

3.iframe有那些缺点?

1. iframe会阻塞主页面的Onload事件;

2. 搜索引擎的检索程序无法解读这种页面，不利于SEO;

3. iframe和主页面共享连接池，而浏览器对相同域的连接有限制，所以会影响页面的并行加载。

4. 使用iframe之前需要考虑这两个缺点。如果需要使用iframe，最好是通过javascript

5. 动态给iframe添加src属性值，这样可以绕开以上两个问题。

4.Label的作用是什么？是怎么用的?

label标签来定义表单控制间的关系,当用户选择该标签时，浏览器会自动将焦点转到和标签相关的表单控件上。

```
<label for="Name">Number:</label>
<input type=“text“name="Name" id="Name"/>
```

<label>Date:<input type="text" name="B"/></label>

5.HTML5的form如何关闭自动完成功能?

给不想要提示的 form 或某个 input 设置为 autocomplete=off。

6.如何实现浏览器内多个标签页之间的通信? (阿里)

1. WebSocket、SharedWorker;

2. 也可以调用localstorge、cookies等本地存储方式;

localstorge另一个浏览上下文里被添加、修改或删除时，它都会触发一个事件，

我们通过监听事件，控制它的值来进行页面信息通信;

注意quirks：Safari 在无痕模式下设置localstorge值时会抛出 QuotaExceededError 的异常;

7.页面可见性（Page Visibility API） 可以有哪些用途?

1 .通过 visibilityState 的值检测页面当前是否可见，以及打开网页的时间等;

2. 在页面被切换到其他后台进程的时候，自动暂停音乐或视频的播放;

8.实现不使用 border 画出1px高的线，在不同浏览器的标准模式与怪异模式下都能保持一致的效果。

9.title与h1的区别、b与strong的区别、i与em的区别?

title属性没有明确意义只表示是个标题，H1则表示层次明确的标题，对页面信息的抓取也有很大的影响;

strong是标明重点内容，有语气加强的含义，使用阅读设备阅读网络时：会重读，而<B>是展示强调内容。

i内容展示为斜体，me表示强调的文本;

Physical Style Elements -- 自然样式标签

b, i, u, s, pre

Semantic Style Elements -- 语义样式标签

strong, em, ins, del, code

应该准确使用语义样式标签, 但不能滥用, 如果不能确定时首选使用自然样式标签。

10.你做的页面在哪些流览器测试过？这些浏览器的内核分别是什么?

Trident内核:IE系列

Gecko内核:Firefox

Webkit内核:Safari

Blink内核：是基于Webkit内核的子项目,使用的浏览器有：

Chrome/opera等除IE、Firefox、Safari之外的几乎所有浏览器

几乎所有国产双内核浏览器（Trident/Blink）如360、猎豹、qq、百度等

11.每个HTML文件里开头都有个很重要的东西，Doctype，知道这是干什么的吗?

文档声明。

<!DOCTYPE> 声明位于文档中的最前面的位置，处于 <html> 标签之前。此标签可告知浏览器文档使用哪种 HTML 或 XHTML 规范。（重点：告诉浏览器按照何种规范解析页面）

IE下如不书写文档声明会使用怪异模式解析网页导致一系列CSS兼容性问题。

12.div+css的布局较table布局有什么优点?

正常场景一般都适用div+CSS布局，

优点：

1. 结构与样式分离
2. 代码语义性好
3. 更符合HTML标准规范
4. SEO友好

Table布局的适用场景：

某种原因不方便加载外部CSS的场景，例如邮件正文，此时用table布局可以在无css情况下保持页面布局正常。

13.img的alt与title有何异同？ strong与em的异同?

a:alt(alt text):为不能显示图像、窗体或applets的用户代理（UA），alt属性用来指定替换文字。替换文字的语言由lang属性指定。(在IE浏览器下会在没有title时把alt当成 tool tip显示)

title(tool tip):该属性为设置该属性的元素提供建议性的信息。

em:表现为斜体，语义表示强调

strong:表现为粗体，语义为强烈语气，强调程度超过em

14.简述一下src与href的区别

src用于替换当前元素，href用于在当前文档和引用资源之间确立联系。

src是source的缩写，指向外部资源的位置，指向的内容将会嵌入到文档中当前标签所在位置；在请求src资源时会将其指向的资源下载并应用到文档内，例如js脚本，img图片和frame等元素。

<script src =”js.js”></script>

当浏览器解析到该元素时，会暂停其他资源的下载和处理，直到将该资源加载、编译、执行完毕，图片和框架等元素也如此，类似于将所指向资源嵌入当前标签内。这也是为什么将js脚本放在底部而不是头部。

href是Hypertext Reference的缩写，指向网络资源所在位置，建立和当前元素（锚点）或当前文档（链接）之间的链接，如果我们在文档中添加

<link href=”common.css” rel=”stylesheet”/>

那么浏览器会识别该文档为css文件，就会并行下载资源并且不会停止对当前文档的处理。这也是为什么建议使用link方式来加载css，而不是使用@import方式。

15.知道的网页制作会用到的图片格式有哪些?

png-8，png-24，jpeg，gif，svg。

但是上面的那些都不是面试官想要的最后答案。面试官希望听到是Webp。（是否有关注新技术，新鲜事物）

科普一下Webp：WebP格式，谷歌（google）开发的一种旨在加快图片加载速度的图片格式。图片压缩体积大约只有JPEG的2/3，并能节省大量的服务器带宽资源和数据空间。Facebook Ebay等知名网站已经开始测试并使用WebP格式。

在质量相同的情况下，WebP格式图像的体积要比JPEG格式图像小40%

16.在css/js代码上线之后开发人员经常会优化性能，从用户刷新网页开始，一次js请求一般情况下有哪些地方会有缓存处理?

dns缓存，cdn缓存，浏览器缓存，服务器缓存。

17.一个页面上有大量的图片（大型电商网站），加载很慢，你有哪些方法优化这些图片的加载，给用户更好的体验。

图片懒加载，在页面上的未可视区域可以添加一个滚动条事件，判断图片位置与浏览器顶端的距离与页面的距离，如果前者小于后者，优先加载。

如果为幻灯片、相册等，可以使用图片预加载技术，将当前展示图片的前一张和后一张优先下载。

如果图片为css图片，可以使用CSSsprite，SVGsprite，Iconfont、Base64等技术。

如果图片过大，可以使用特殊编码的图片，加载时会先加载一张压缩的特别厉害的缩略图，以提高用户体验。

如果图片展示区域小于图片的真实大小，则因在服务器端根据业务需要先行进行图片压缩，图片压缩后大小与展示一致。

18.你如何理解HTML结构的语义化?

HTML结构语义化：

更符合W3C统一的规范标准，是技术趋势。

没有样式时浏览器的默认样式也能让页面结构很清晰。

对功能障碍用户友好。屏幕阅读器（如果访客有视障）会完全根据你的标记来“读”你的网页。

对其他非主流终端设备友好。例如机顶盒、PDA、各种移动终端。

对SEO友好。

19.谈谈以前端角度出发做好SEO需要考虑什么?

搜索引擎主要以:

外链数量和质量

网页内容和结构

来决定某关键字下的网页搜索排名。

前端应该注意网页结构和内容方面的情况：

Meta标签优化

主要包括主题（Title)，网站描述(Description)。还有一些其它的隐藏文字比如Author（作者），Category（目录），Language（编码语种）等。

符合W3C规范的语义性标签的使用。

如何选取关键词并在网页中放置关键词

搜索就得用关键词。关键词分析和选择是SEO最重要的工作之一。首先要给网站确定主关键词（一般在5个上下），然后针对这些关键词进行优化，包括关键词密度（Density），相关度（Relavancy），突出性（Prominency）等等。

20.html5有哪些新特性、移除了那些元素？如何处理HTML5新标签的浏览器兼容问题？如何区分 HTML 和 HTML5?

新特性：

1. 拖拽释放(Drag and drop) API

2. 语义化更好的内容标签（header,nav,footer,aside,article,section）

3. 音频、视频API(audio,video)

4. 画布(Canvas) API

5. 地理(Geolocation) API

6. 本地离线存储 localStorage 长期存储数据，浏览器关闭后数据不丢失；

7. sessionStorage 的数据在浏览器关闭后自动删除

8. 表单控件，calendar、date、time、email、url、search

9. 新的技术webworker, websocket, Geolocation

移除的元素：

1. 纯表现的元素：basefont，big，center，font, s，strike，tt，u；

2. 对可用性产生负面影响的元素：frame，frameset，noframes；

支持HTML5新标签：

1. IE8/IE7/IE6支持通过 document.createElement 方法产生的标签，可以利用这一特性让这些浏览器支持 HTML5 新标签，浏览器支持新标签后，还需要添加标签默认的样式（当然最好的方式是直接使用成熟的框架、使用最多的是html5shim框架）：

<!--[if lt IE 9]>

<script> src="http://html5shim.googlecode.com/svn/trunk/html5.js"</script>

<![endif]-->

如何区分：

DOCTYPE声明新增的结构元素、功能元素

21.HTML5 Canvas 元素有什么用?

Canvas 元素用于在网页上绘制图形，该元素标签强大之处在于可以直接在 HTML 上进行图形操作。

22.如何在 HTML5 页面中嵌入音频?

HTML 5 包含嵌入音频文件的标准方式，支持的格式包括 MP3、Wav 和 Ogg：

```
<audio controls>
  <source src="jamshed.mp3" type="audio/mpeg">
   Your browser does'nt support audio embedding feature.
</audio>
```

23.如何在 HTML5 页面中嵌入视频?

和音频一样，HTML5 定义了嵌入视频的标准方法，支持的格式包括：MP4、WebM 和 Ogg：

```
<video width="450" height="340" controls>
  <source src="jamshed.mp4" type="video/mp4">
```

Your browser does'nt support video embedding feature.

</video>

24.HTML5 引入什么新的表单属性?

Datalist datetime output keygen date month week time number range emailurl

二 CSS

1.box-sizing

box-sizing属性主要用来控制元素的盒模型的解析模式。默认值是content-box。

content-box：让元素维持W3C的标准盒模型。元素的宽度/高度由border + padding + content的宽度/高度决定，设置width/height属性指的是content部分的宽/高

border-box：让元素维持IE传统盒模型（IE6以下版本和IE6~7的怪异模式）。设置width/height属性指的是border + padding + content

应用场景：统一风格的表单元素 表单中有一些input元素其实还是展现的是传统IE盒模型，带有一些默认的样式，而且在不同平台或者浏览器下的表现不一，造成了表单展现的差异。此时我们可以通过box-sizing属性来构建一个风格统一的表单元素。

2.水平垂直居中的方法

行内布局

line-height + text-align vertical-align + text-align

块布局

position absolute + margin auto position absolute + negative margin position absolute + translate(-50%, -50%)

父容器子容器不确定宽高的块级元素，做上下居中

1.flex

```
#wrap{
  display:flex;
  justify-content:center;
```

```
    align-items:center;
}
```

2.tabel

```
.parent {
  text-align: center;//水平居中
  display: table-cell;
  vertical-align: middle;//垂直居中
}
.child {
    display: inline-block;//防止块级元素宽度独占一行，内联元素可不设置
}
```

3.absolute+transform 水平垂直居中

```
<div class="parent">
 <div class="child">Demo</div>
</div>

<style>
 .parent {
   position: relative;
 }
 .child {
   position: absolute;
   left: 50%;
   top: 50%;
   transform: translate(-50%, -50%);
 }
</style>
```

4.webkit-box

//对父级元素设置

position: relative;

display: -webkit-box;

-webkit-box-align: center;

-webkit-box-pack: center;

for detail: https://github.com/hawx1993/tech-blog/issues/12

3.实现左边定宽右边自适应效果

1.table(父级元素)与tabel-cell（两个子集元素）

2.flex(父级元素)+flex :1（右边子元素）

3.左边定宽，并且左浮动；右边设置距离左边的宽度

4.左边定宽，左边设置position:absolute；右边设置距离左边的宽度

4.三列布局（中间固定两边自适应宽度）

1. 采用浮动布局（左边左浮动，右边右浮动，中间margin：0 宽度值）

2. 绝对定位方式（左右绝对定位，左边left0右边right0，中间上同）

5.BFC（Block Formatting Contexts）块级格式化上下文

块格式化上下文（block formatting context） 是页面上的一个独立的渲染区域，容器里面的子元素不会在布局上影响到外面的元素。它是决定块盒子的布局及浮动元素相互影响的一个因素。

下列情况将创建一个块格式化上下文：

① float

② overflow

③ display（display为inline-block、table-cell）

④ position（absolute 或 fixed）

BFC的作用

1.清除内部浮动：对子元素设置浮动后，父元素会发生高度塌陷，也就是父元素的高度变为0。解决这个问题，只需要把把父元素变成一个BFC就行了。常用的办法是给父元素设置overflow:hidden。

2.上下margin重合问题，可以通过触发BFC来解决

6.清除浮动元素的方法和各自的优缺点

清除浮动，实际上是清除父元素的高度塌陷。因为子元素脱离了父元素的文档流，所以，父元素失去了高度，导致了塌陷。要解决这个问题，就是让父元素具有高度。

浮动元素的特性： 在正常布局中位于该浮动元素之下的内容，此时会围绕着浮动元素，填满其右侧的空间。浮动到右侧的元素，其他内容将从左侧环绕它（浮动元素影响的不仅是自己，它会影响周围的元素对其进行环绕。float仍会占据其位置，position:absolute不占用页面空间 会有重叠问题 ）

1.在浮动元素末尾添加空标签清除浮动 clear:both （缺点：增加无意义标签）

2.给父元素设置 overflow:auto属性

3.after伪元素

7.css实现自适应正方形

方案一：CSS3 vw 单位

方案二：设置垂直方向的padding撑开容器

方案三：利用伪元素的 margin(padding)-top 撑开容器

8.position的值

1. absolute :生成绝对定位的元素， 相对于最近一级的 定位不是 static 的父元素来进行定位。

2. fixed （老IE不支持）生成绝对定位的元素，通常相对于浏览器窗口或 frame 进行定位。

3. relative 生成相对定位的元素，相对于其在普通流中的位置进行定位。

4. static 默认值。没有定位，元素出现在正常的流中

5. sticky 生成粘性定位的元素，容器的位置根据正常文档流计算得出

9.如何在页面上实现一个圆形的可点击区域?

1、map+area或者svg

2、border-radius

3、纯js实现 需要求一个点在不在圆上简单算法、获取鼠标坐标等等

10.介绍一下标准的CSS的盒子模型？低版本IE的盒子模型有什么不同的?

（1）有两种， IE 盒子模型、W3C 盒子模型；

（2）盒模型： 内容(content)、填充(padding)、边界(margin)、 边框(border)；

（3）区 别： IE的content部分把 border 和 padding计算了进去;

11.CSS选择符有哪些？哪些属性可以继承?

* 一、基本选择器

1. * 通用元素选择器，匹配任何元素

2. E 标签选择器，匹配所有使用E标签的元素

3. .info class选择器，匹配所有class属性中包含info的元素

4. #footer id选择器，匹配所有id属性等于footer的元素

二、多元素的组合选择器

5. E,F 多元素选择器，同时匹配所有E元素或F元素，E和F之间用逗号分隔

6. E F 后代元素选择器，匹配所有属于E元素后代的F元素，E和F之间用空格分隔

7. E > F 子元素选择器，匹配所有E元素的子元素F

8. E + F 毗邻元素选择器，匹配所有紧随E元素之后的同级元素F

三、CSS 2.1 属性选择器

9. E[att] 匹配所有具有att属性的E元素，不考虑它的值。（注意：E在此处可以省略，比如"[cheacked]"。以下同。）

10. E[att=val] 匹配所有att属性等于"val"的E元素

11. E[att~=val] 匹配所有att属性具有多个空格分隔的值、其中一个值等于"val"的E元素

12. E[att|=val] 匹配所有att属性具有多个连字号分隔（hyphen-separated）的值、其中一个值以"val"开头的E元素，主要用于lang属性，比如"en"、"en-us"、"en-gb"等等

四、CSS 2.1中的伪类

13. E:first-child 匹配父元素的第一个子元素
14. E:link匹配所有未被点击的链接
15. E:visited 匹配所有已被点击的链接
16. E:active 匹配鼠标已经其上按下、还没有释放的E元素
17. E:hover 匹配鼠标悬停其上的E元素
18. E:focus 匹配获得当前焦点的E元素
19. E:lang(c) 匹配lang属性等于c的E元素

五、 CSS 2.1中的伪元素

20. E:first-line 匹配E元素的第一行
21. E:first-letter 匹配E元素的第一个字母
22. E:before 在E元素之前插入生成的内容
23. E:after 在E元素之后插入生成的内容

六、CSS 3的同级元素通用选择器

24. E ~ F 匹配任何在E元素之后的同级F元素

七、CSS 3 属性选择器

25. E[att^="val"] 属性att的值以"val"开头的元素
26. E[att$="val"] 属性att的值以"val"结尾的元素
27. E[att*="val"] 属性att的值包含"val"字符串的元素

八、CSS 3中与用户界面有关的伪类

28. E:enabled 匹配表单中激活的元素
29. E:disabled 匹配表单中禁用的元素
30. E:checked 匹配表单中被选中的radio（单选框）或checkbox（复选框）元素
31. E::selection 匹配用户当前选中的元素

九、CSS 3中的结构性伪类

32. E:root 匹配文档的根元素，对于HTML文档，就是HTML元素

33. E:nth-child(n) 匹配其父元素的第n个子元素，第一个编号为1

34. E:nth-last-child(n) 匹配其父元素的倒数第n个子元素，第一个编号为1

35. E:nth-of-type(n) 与:nth-child()作用类似，但是仅匹配使用同种标签的元素

36. E:nth-last-of-type(n) 与:nth-last-child() 作用类似，但是仅匹配使用同种标签的元素

37. E:last-child 匹配父元素的最后一个子元素，等同于:nth-last-child(1)

38. E:first-of-type匹配父元素下使用同种标签的第一个子元素，等同于:nth-of-type(1)

39. E:last-of-type 匹配父元素下使用同种标签的最后一个子元素，等同于:nth-last-of-type(1)

40. E:only-child 匹配父元素下仅有的一个子元素，等同于:first-child:last-child或 :nth-child(1):nth-last-child(1)

41. E:only-of-type 匹配父元素下使用同种标签的唯一一个子元素，等同于:first-of-type:last-of-type或 :nth-of-type(1):nth-last-of-type(1)

42. E:empty 匹配一个不包含任何子元素的元素，注意，文本节点也被看作子元素

十、CSS 3的反选伪类

43. E:not(s) 匹配不符合当前选择器的任何元素

十一、CSS 3中的 :target 伪类

44. E:target 匹配文档中特定"id"点击后的效果

* 可继承的样式： font-size font-family color, UL LI DL DD DT;

* 不可继承的样式：border padding margin width height ;

12.CSS优先级算法如何计算?

* 优先级就近原则，同权重情况下样式定义最近者为准;

* 载入样式以最后载入的定位为准;

优先级为:

同权重: 内联样式表（标签内部）> 嵌入样式表（当前文件中）> 外部样式表（外部文件中）。

!important > id > class > tag

important 比 内联优先级高

13.CSS3新增伪类有那些?

举例:

p:first-of-type 选择属于其父元素的首个 <p> 元素的每个 <p> 元素。

p:last-of-type 选择属于其父元素的最后 <p> 元素的每个 <p> 元素。

p:only-of-type 选择属于其父元素唯一的 <p> 元素的每个 <p> 元素。

p:only-child 选择属于其父元素的唯一子元素的每个 <p> 元素。

p:nth-child(2) 选择属于其父元素的第二个子元素的每个 <p> 元素。

::after 在元素之前添加内容,也可以用来做清除浮动。

::before 在元素之后添加内容

:enabled

:disabled 控制表单控件的禁用状态。

:checked 单选框或复选框被选中。

14.display有哪些值？说明他们的作用。

block 块类型。默认宽度为父元素宽度，可设置宽高，换行显示。

none 缺省值。象行内元素类型一样显示。

inline 行内元素类型。默认宽度为内容宽度，不可设置宽高，同行显示。

inline-block 默认宽度为内容宽度，可以设置宽高，同行显示。

list-item 象块类型元素一样显示，并添加样式列表标记。

table 此元素会作为块级表格来显示。

inherit 规定应该从父元素继承 display 属性的值。

15.CSS3有哪些新特性?

新增各种CSS选择器 （: not(.input)：所有 class 不是“input”的节点）

圆角 （border-radius:8px）

多列布局 （multi-column layout）

阴影和反射 （Shadow\Reflect）

文字特效 （text-shadow、）

文字渲染 （Text-decoration）

线性渐变 （gradient）

旋转 （transform）

缩放,定位,倾斜,动画,多背景

例如:transform:\scale(0.85,0.90)\ translate(0px,-30px)\ skew(-9deg,0deg)\Animation:

16.请解释一下CSS3的Flexbox（弹性盒布局模型）,以及适用场景?

一个用于页面布局的全新CSS3功能，Flexbox可以把列表放在同一个方向（从上到下排列，从左到右），并让列表能延伸到占用可用的空间。

较为复杂的布局还可以通过嵌套一个伸缩容器（flex container）来实现。

采用Flex布局的元素，称为Flex容器（flex container），简称"容器"。

它的所有子元素自动成为容器成员，称为Flex项目（flex item），简称"项目"。

常规布局是基于块和内联流方向，而Flex布局是基于flex-flow流可以很方便的用来做局中，能对不同屏幕大小自适应。

在布局上有了比以前更加灵活的空间。

具体：http://www.w3cplus.com/css3/flexbox-basics.html

17.用纯CSS创建一个三角形的原理是什么?

把上、左、右三条边隐藏掉（颜色设为 transparent）

```
#demo {
  width: 0;
  height: 0;
  border-width: 20px;
  border-style: solid;
  border-color: transparent transparent red transparent;
}
```

18.一个满屏品字布局如何设计?

简单的方式：

上面的div宽100%，

下面的两个div分别宽50%，

然后用float或者inline使其不换行即可

19.css多列等高如何实现？

利用padding-bottom|margin-bottom正负值相抵；

设置父容器设置超出隐藏（overflow:hidden），这样子父容器的高度就还是它里面的列没有设定padding-bottom时的高度，

当它里面的任 一列高度增加了，则父容器的高度被撑到里面最高那列的高度，

其他比这列矮的列会用它们的padding-bottom补偿这部分高度差。

20.经常遇到的浏览器的兼容性有哪些？原因，解决方法是什么，常用hack的技巧？

* png24位的图片在iE6浏览器上出现背景，解决方案是做成PNG8.

* 浏览器默认的margin和padding不同。解决方案是加一个全局的*{margin:0;padding:0;}来统一。

* IE6双边距bug:块属性标签float后，又有横行的margin情况下，在ie6显示margin比设置的大。

浮动ie产生的双倍距离 #box{ float:left; width:10px; margin:0 0 0 100px;}

这种情况之下IE会产生20px的距离，解决方案是在float的标签样式控制中加入 ——_display:inline; 将其转化为行内属性。(_这个符号只有ie6会识别)

渐进识别的方式，从总体中逐渐排除局部。

首先，巧妙的使用“\9”这一标记，将IE游览器从所有情况中分离出来。

接着，再次使用“+”将IE8和IE7、IE6分离开来，这样IE8已经独立识别。

css

.bb{

```
  background-color:red;/*所有识别*/
 background-color:#00deff\9; /*IE6、7、8识别*/
 +background-color:#a200ff;/*IE6、7识别*/
 _background-color:#1e0bd1;/*IE6识别*/
}
```

* IE下,可以使用获取常规属性的方法来获取自定义属性,
  也可以使用getAttribute()获取自定义属性;
  Firefox下,只能使用getAttribute()获取自定义属性。
  解决方法:统一通过getAttribute()获取自定义属性。

* IE下,even对象有x,y属性,但是没有pageX,pageY属性;
  Firefox下,event对象有pageX,pageY属性,但是没有x,y属性。

* 解决方法：（条件注释）缺点是在IE浏览器下可能会增加额外的HTTP请求数。

* Chrome 中文界面下默认会将小于 12px 的文本强制按照 12px 显示,
  可通过加入 CSS 属性 -webkit-text-size-adjust: none; 解决。

超链接访问过后hover样式就不出现了 被点击访问过的超链接样式不在具有hover和active了解决方法是改变CSS属性的排列顺序:

L-V-H-A : a:link {} a:visited {} a:hover {} a:active {}

21.li与li之间有看不见的空白间隔是什么原因引起的？有什么解决办法？

行框的排列会受到中间空白（回车\空格）等的影响，因为空格也属于字符,这些空白也会被应用样式，占据空间，所以会有间隔，把字符大小设为0，就没有空格了。

22.为什么要初始化CSS样式。

- 因为浏览器的兼容问题，不同浏览器对有些标签的默认值是不同的，如果没对CSS初始化往往会出现浏览器之间的页面显示差异。

- 当然，初始化样式会对SEO有一定的影响，但鱼和熊掌不可兼得，但力求影响最小的情况下初始化。

最简单的初始化方法： * {padding: 0; margin: 0;} （强烈不建议）

淘宝的样式初始化代码：

body, h1, h2, h3, h4, h5, h6, hr, p, blockquote, dl, dt, dd, ul, ol, li, pre, form, fieldset, legend, button, input, textarea, th, td { margin:0; padding:0; }

body, button, input, select, textarea { font:12px/1.5tahoma, arial, \5b8b\4f53; }

h1, h2, h3, h4, h5, h6{ font-size:100%; }

address, cite, dfn, em, var { font-style:normal; }

code, kbd, pre, samp { font-family:couriernew, courier, monospace; }

small{ font-size:12px; }

ul, ol { list-style:none; }

a { text-decoration:none; }

a:hover { text-decoration:underline; }

sup { vertical-align:text-top; }

sub{ vertical-align:text-bottom; }

legend { color:#000; }

fieldset, img { border:0; }

button, input, select, textarea { font-size:100%; }

table { border-collapse:collapse; border-spacing:0; }

23.absolute的containing block(容器块)计算方式跟正常流有什么不同?

无论属于哪种，都要先找到其祖先元素中最近的 position 值不为 static 的元素，然后再判断：

1、若此元素为 inline 元素，则 containing block 为能够包含这个元素生成的第一个和最后一个 inline box 的 padding box (除 margin, border 外的区域) 的最小矩形；

2、否则,则由这个祖先元素的 padding box 构成。

如果都找不到，则为 initial containing block。

补充：

1. static(默认的)/relative：简单说就是它的父元素的内容框（即去掉padding的部分）
2. absolute: 向上找最近的定位为absolute/relative的元素
3. fixed: 它的containing block一律为根元素(html/body)，根元素也是initial containing block

24.CSS里的visibility属性有个collapse属性值是干嘛用的？在不同浏览器下以后什么区别？

对于普通元素visibility:collapse;会将元素完全隐藏,不占据页面布局空间,与display:none;表现相同. 如果目标元素为table,visibility:collapse;将table隐藏,但是会占据页面布局空间. 仅在Firefox下起作用,IE会显示元素,Chrome会将元素隐藏,但是占据空间.

25.position跟display、margin collapse、overflow、float这些特性相互叠加后会怎么样？

如果元素的display为none,那么元素不被渲染,position,float不起作用,如果元素拥有position:absolute;或者position:fixed;属性那么元素将为绝对定位,float不起作用.如果元素float属性不是none,元素会脱离文档流,根据float属性值来显示.有浮动,绝对定位,inline-block属性的元素,margin不会和垂直方向上的其他元素margin折叠.

26.对BFC规范(块级格式化上下文：block formatting context)的理解？

（W3C CSS 2.1 规范中的一个概念,它是一个独立容器，决定了元素如何对其内容进行定位,以及与其他元素的关系和相互作用。）

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（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,也就是说BFC内部的元素和外部的元素不会互相影响。

27. 请解释一下为什么需要清除浮动？清除浮动的方式

清除浮动是为了清除使用浮动元素产生的影响。浮动的元素，高度会塌陷，而高度的塌陷使我们页面后面的布局不能正常显示。

1、父级div定义height；
2、父级div 也一起浮动；
3、常规的使用一个class；

```
.clearfix::before, .clearfix::after {
    content: " ";
    display: table;
}
.clearfix::after {
    clear: both;
}
.clearfix {
    *zoom: 1;
}
```

4、SASS编译的时候，浮动元素的父级div定义伪类:after

```
&::after,&::before{
    content: " ";
     visibility: hidden;
     display: block;
     height: 0;
     clear: both;
}
```

解析原理：

1) display:block 使生成的元素以块级元素显示,占满剩余空间;

2) height:0 避免生成内容破坏原有布局的高度。

3) visibility:hidden 使生成的内容不可见，并允许可能被生成内容盖住的内容可以进行点击和交互;

4）通过 content:"."生成内容作为最后一个元素，至于content里面是点还是其他都是可以的，例如oocss里面就有经典的 content:".",有些版本可能content 里面内容为空,一丝冰凉是不推荐这样做的,firefox直到7.0 content:” " 仍然会产生额外的空隙；

5）zoom：1 触发IE hasLayout。

通过分析发现，除了clear：both用来闭合浮动的，其他代码无非都是为了隐藏掉content生成的内容，这也就是其他版本的闭合浮动为什么会有font-size：0，line-height：0。

28.什么是外边距合并?

外边距合并指的是，当两个垂直外边距相遇时，它们将形成一个外边距。

合并后的外边距的高度等于两个发生合并的外边距的高度中的较大者。

w3school介绍网址： http://www.w3school.com.cn/css/css_margin_collapsing.asp

29.zoom:1的清除浮动原理?

清除浮动，触发hasLayout；

Zoom属性是IE浏览器的专有属性，它可以设置或检索对象的缩放比例。解决ie下比较奇葩的bug。

譬如外边距（margin）的重叠，浮动清除，触发ie的haslayout属性等。

来龙去脉大概如下：

当设置了zoom的值之后，所设置的元素就会就会扩大或者缩小，高度宽度就会重新计算了，这里一旦改变zoom值时其实也会发生重新渲染，运用这个原理，也就解决了ie下子元素浮动时候父元素不随着自动扩大的问题。

Zoom属是IE浏览器的专有属性，火狐和老版本的webkit核心的浏览器都不支持这个属性。然而，zoom现在已经被逐步标准化，出现在 CSS 3.0 规范草案中。

目前非ie由于不支持这个属性，它们又是通过什么属性来实现元素的缩放呢?

可以通过css3里面的动画属性scale进行缩放。

30.CSS优化、提高性能的方法有哪些?

关键选择器（key selector）。选择器的最后面的部分为关键选择器（即用来匹配目标元素的部分）；

如果规则拥有 ID 选择器作为其关键选择器，则不要为规则增加标签。过滤掉无关的规则（这样样式系统就不会浪费时间去匹配它们了）；

提取项目的通用公有样式，增强可复用性，按模块编写组件；增强项目的协同开发性、可维护性和可扩展性;

使用预处理工具或构建工具（gulp对css进行语法检查、自动补前缀、打包压缩、自动优雅降级）；

31.怎么让Chrome支持小于12px 的文字?

1、用图片：如果是内容固定不变情况下，使用将小于12px文字内容切出做图片，这样不影响兼容也不影响美观。

2、使用12px及12px以上字体大小：为了兼容各大主流浏览器，建议设计美工图时候设置大于或等于12px的字体大小，如果是接单的这个时候就需要给客户讲解小于12px浏览器不兼容等事宜。

3、继续使用小于12px字体大小样式设置：如果不考虑chrome可以不用考虑兼容，同时在设置小于12px对象设置-webkit-text-size-adjust:none，做到最大兼容考虑。

4、使用12px以上字体：为了兼容、为了代码更简单 从新考虑权重下兼容性。

32.position:fixed;在android下无效怎么处理?

fixed的元素是相对整个页面固定位置的，你在屏幕上滑动只是在移动这个所谓的viewport，

原来的网页还好好的在那，fixed的内容也没有变过位置，

所以说并不是iOS不支持fixed，只是fixed的元素不是相对手机屏幕固定的。

<meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0, minimum-scale=1.0, user-scalable=no"/>

33.如果需要手动写动画，你认为最小时间间隔是多久，为什么？（阿里）

多数显示器默认频率是60Hz，即1秒刷新60次，所以理论上最小间隔为1/60＊1000ms ＝ 16.7ms

34.display:inline-block 什么时候会显示间隙？(携程)

移除空格、使用margin负值、使用font-size:0、letter-spacing、word-spacing

35.有哪项方式可以对一个DOM设置它的CSS样式?

外部样式表，引入一个外部css文件

内部样式表，将css代码放在 <head> 标签内部

内联样式，将css样式直接定义在 HTML 元素内部

36.超链接访问过后hover样式就不出现的问题是什么？如何解决?

被点击访问过的超链接样式不再具有hover和active了,解决方法是改变CSS属性的排列顺序: L-V-H-A（link,visited,hover,active）

37.什么是Css Hack？ie6,7,8的hack分别是什么?

针对不同的浏览器写不同的CSS code的过程，就是CSS hack。

示例如下：

```
  #test{
     background-color:yellow;   /*ie8*/
```

```
    +background-color:pink;      /*ie7*/
    _background-color:orange;     /*ie6*/   }
```

更好的方式是使用IE条件判断语句：

<!–[if lte IE 6]>

内容

<![endif]–>

等

38.什么是外边距重叠？重叠的结果是什么？

外边距重叠就是margin-collapse。

在CSS当中，相邻的两个盒子（可能是兄弟关系也可能是祖先关系）的外边距可以结合成一个单独的外边距。这种合并外边距的方式被称为折叠，并且因而所结合成的外边距称为折叠外边距。

折叠结果遵循下列计算规则：

两个相邻的外边距都是正数时，折叠结果是它们两者之间较大的值。

两个相邻的外边距都是负数时，折叠结果是两者绝对值的较大值。

两个外边距一正一负时，折叠结果是两者的相加的和。

39.rgba()和opacity的透明效果有什么不同？

rgba()和opacity都能实现透明效果，但最大的不同是opacity作用于元素，以及元素内的所有内容的透明度，

而rgba()只作用于元素的颜色或其背景色。（设置rgba透明的元素的子元素不会继承透明效果！）

40.如何垂直居中一个浮动元素？

```
  // 方法一：已知元素的高宽
#div1{
  background-color:#6699FF;
  width:200px;
  height:200px;
  position: absolute;      /*父元素需要相对定位*/
  top: 50%;
```

```
  left: 50%;
  margin-top:-100px ;  /*二分之一的height，width*/
  margin-left: -100px;
  }

//方法二:未知元素的高宽

 #div1{
  width: 200px;
  height: 200px;
  background-color: #6699FF;

  margin:auto;
  position: absolute;      /*父元素需要相对定位*/
  left: 0;
  top: 0;
  right: 0;
  bottom: 0;
  }
/*如何垂直居中一个<img>?（用更简便的方法。）*/

#container    /*<img>的容器设置如下*/
{
  display:table-cell;
  text-align:center;
  vertical-align:middle;
}
```

41.px和em的区别

px和em都是长度单位，区别是：

px值固定，容易计算。

em值不固定，是相对单位，其相对应父级元素的字体大小会调整

42.描述一个”reset”的CSS文件并如何使用它。知道normalize.css吗？你了解他们的不同之处？

Reset样式的目的是清除浏览器某些默认样式，方便css书写，例如：

* {margin:0;padding:0;list-style:none;}

Normalize的理念与reset不同，它并不清除浏览器默认样式，而是尽量将所有浏览器的默认样式统一。

43.Sass、LESS是什么？大家为什么要使用他们？

他们是CSS预处理器。他是CSS上的一种抽象层。他们是一种特殊的语法/语言编译成CSS。

例如Less是一种动态样式语言. 将CSS赋予了动态语言的特性，如变量，继承，运算， 函数. LESS 既可以在客户端上运行 (支持IE 6+, Webkit, Firefox)，也可一在服务端运行 (借助 Node.js)。

为什么要使用它们？

结构清晰，便于扩展。

可以方便地屏蔽浏览器私有语法差异。这个不用多说，封装对浏览器语法差异的重复处理，减少无意义的机械劳动。

可以轻松实现多重继承。

完全兼容 CSS 代码，可以方便地应用到老项目中。LESS 只是在 CSS 语法上做了扩展，所以老的 CSS 代码也可以与 LESS 代码一同编译。

44.为什么要初始化样式？

用于浏览器默认css样式的存在并且不同浏览器对相同HTML标签的默认样式不同，若不初始化会造成不同浏览器之间的显示差异。

45.IE的双边距BUG：块级元素float后设置横向margin，ie6显示的margin比设置的较大。

解决：加入_display：inline

46.html常见兼容性问题？

1.双边距BUG float引起的 使用display

2.3像素问题 使用float引起的 使用dislpay:inline -3px

3.超链接hover 点击后失效 使用正确的书写顺序 link visited hover active

4.IE z-index问题 给父级添加position:relative

5.Png 透明 使用js代码 改

6.Min-height 最小高度！ Important 解决’

7.select 在ie6下遮盖 使用iframe嵌套

8.为什么没有办法定义1px左右的宽度容器（IE6默认的行高造成的，使用over:hidden,zoom:0.08 line-height:1px）

9.IE5-8不支持opacity，解决办法：

```
.opacity {
   opacity: 0.4
   filter: alpha(opacity=60); /* for IE5-7 */
   -ms-filter: "progid:DXImageTransform.Microsoft.Alpha(Opacity=60)"; /* for IE 8*/
}
```

10. IE6不支持PNG透明背景，解决办法: IE6下使用gif图片

47.对WEB标准以及W3C的理解与认识

标签闭合、标签小写、不乱嵌套、提高搜索机器人搜索几率、使用外 链css和js脚本、结构行为表现的分离、文件下载与页面速度更快、内容能被更多的用户所访问、内容能被更广泛的设备所访问、更少的代码和组件，容易维 护、改版方便，不需要变动页面内容、提供打印版本而不需要复制内容、提高网站易用性。

48.行内元素和块级元素的具体区别是什么？行内元素的padding和margin可设置吗？

块级元素(block)特性：

总是独占一行，表现为另起一行开始，而且其后的元素也必须另起一行显示;

宽度(width)、高度(height)、内边距(padding)和外边距(margin)都可控制;

内联元素(inline)特性：

和相邻的内联元素在同一行;

宽度(width)、高度(height)、内边距的top/bottom(padding-top/padding-bottom)和外边距的top/bottom(margin-top/margin-bottom)都不可改变（也就是padding和margin的left和right是可以设置的），就是里面文字或图片的大小。

那么问题来了，浏览器还有默认的天生inline-block元素（拥有内在尺寸，可设置高宽，但不会自动换行），有哪些？

答案：<input>、<img>、<button>、<textarea>、<label>

49.什么是外边距重叠？重叠的结果是什么？

答案：

外边距重叠就是margin-collapse。

在CSS当中，相邻的两个盒子（可能是兄弟关系也可能是祖先关系）的外边距可以结合成一个单独的外边距。这种合并外边距的方式被称为折叠，并且因而所结合成的外边距称为折叠外边距。

折叠结果遵循下列计算规则：

两个相邻的外边距都是正数时，折叠结果是它们两者之间较大的值。

两个相邻的外边距都是负数时，折叠结果是两者绝对值的较大值。

两个外边距一正一负时，折叠结果是两者的相加的和。

50.知道css有个content属性吗？有什么作用？有什么应用？

知道。css的content属性专门应用在 before/after 伪元素上，用来插入生成内容。最常见的应用是利用伪类清除浮动。

```
//一种常见利用伪类清除浮动的代码
.clearfix:after {
  content:"."; //这里利用到了content属性
  display:block;
  height:0;
  visibility:hidden;
  clear:both; }
.clearfix {
  *zoom:1;
}
```

after伪元素通过 content 在元素的后面生成了内容为一个点的块级素，再利用clear:both清除浮动。

那么问题继续还有，知道css计数器（序列数字字符自动递增）吗？如何通过css content属性实现css计数器？

答案：css计数器是通过设置counter-reset 、counter-increment 两个属性 、及counter()/counters()一个方法配合after / before 伪类实现。

51.介绍下 BFC 及其应用

1. 内部box会在垂直方向，一个接一个地放置。

2. Box垂直方向的距离由margin决定，在一个BFC中，两个相邻的块级盒子的垂直外边距会产生折叠。
3. 在BFC中，每一个盒子的左外边缘（margin-left）会触碰到容器的左边缘(border-left)（对于从右到左的格式来说，则触碰到右边缘）
4. 形成了BFC的区域不会与float box重叠
5. 计算BFC高度时，浮动元素也参与计算

生成BFC除了 @webproblem 童鞋所说的还有：行内块元素、网格布局、contain值为layout、content或 strict的元素等。

BFC作用：

1. 利用特性4可实现左图右文之类的效果：

HTML

```
<img src='image.png'>
<p>我是超长的文字<p>

img {
    float:left
}
p {
    overflow:hidden
}
```

2. 利用特性5可以解决浮动元素造成的父元素高度塌陷问题：

C#

```
<div class='parent'>
    <div class='float'>浮动元素</div>
</div>

.parent {
    overflow:hidden;
}
.float {
    float:left;
}
```

51.怎么让一个 div 水平垂直居中

1. 定宽高 使用定位 + margin

SCSS

```scss
element.style {
    position: absolute;
    left: 50%;
    top: 50%;
    margin-left: -250px;
    margin-top: -250px;
    width: 500px;
    height: 500px;
    background: yellow;
    z-index: 1;
}
```

2. 使用定位 + trnsfrom

SCSS

```scss
element.style {
    position: absolute;
    left: 50%;
    top: 50%;
    width: 500px;
    height: 500px;
    background: yellow;
    z-index: 1;
    transform: translate3d(-50%, -50%, 0)
```

52.分析比较 opacity: 0、visibility: hidden、display: none 优劣和适用场景。

结构： display:none: 会让元素完全从渲染树中消失，渲染的时候不占据任何空间, 不能点击，visibility: hidden:不会让元素从渲染树消失，渲染元素继续占据空间，只是内容不可见，不能点击 opacity: 0: 不会让元素从渲染树消失，渲染元素继续占据空间，只是内容不可见，可以点击

继承： display: none和opacity: 0：是非继承属性，子孙节点消失由于元素从渲染树消失造成，通过修改子孙节点属性无法显示。 visibility: hidden：是继承属性，子孙节点消失由于继承了hidden，通过设置visibility: visible;可以让子孙节点显式。

性能： displaynone : 修改元素会造成文档回流,读屏器不会读取display: none元素内容，性能消耗较大 visibility:hidden: 修改元素只会造成本元素的重绘,性能消耗较少读屏器读取visibility: hidden元素内容 opacity: 0 ： 修改元素会造成重绘，性能消耗较少

联系：它们都能让元素不可见

53.已知如下代码，如何修改才能让图片宽度为 300px？注意下面代码不可修改。

JavaScript

```
<img src="1.jpg" style="width:480px!important;”>
```

SCSS

```
box-sizing: border-box;
padding: 0 90px;
```

1.css方法 max-width:300px;覆盖其样式； transform: scale(0.625)；按比例缩放图片；

2.js方法 document.getElementsByTagName("img")
[0].setAttribute("style","width:300px!important;")

54.如何解决移动端 Retina 屏 1px 像素问题

伪元素 + transform scaleY(.5) 2 border-image 3 background-image 4 box-shadow

55.介绍下 BFC、IFC、GFC 和 FFC

BFC（Block formatting contexts）：块级格式上下文页面上的一个隔离的渲染区域，那么他是如何产生的呢？可以触发BFC的元素有float、position、overflow、display：table-cell/ inline-block/table-caption ； BFC有什么作用呢？比如说实现多栏布局’

IFC（Inline formatting contexts）：内联格式上下文IFC的line box（线框）高度由其包含行内元素中最高的实际高度计算而来（不受到竖直方向的padding/margin影响)IFC中的line box一般左右都贴紧整个IFC，但是会因为float元素而扰乱。float元素会位于IFC与与line box之间，使得line box宽度缩短。 同个ifc下的多个line box高度会不同 IFC中时不可能有块级元素的，当插入块级元素时（如p中插入div）会产生两个匿名块与div分隔开，即产生两个IFC，每个IFC对外表现为块级元素，与div垂直排列。 那么IFC一般有什么用呢？ 水平居中：当一个块要在环境中水平居中时，设置其为inline-block则会在外层产生IFC，通过text-align则可以使其水平居中。 垂直居中：创建一个IFC，用其中一个元素撑开父元素的高度，然后设置其vertical-align:middle，其他行内元素则可以在此父元素下垂直居中。

GFC（GrideLayout formatting contexts）：网格布局格式化上下文当为一个元素设置display值为grid的时候，此元素将会获得一个独立的渲染区域，我们可以通过在网格容器（grid container）上定义网格定义行（grid definition rows）和网格定义列（grid definition columns）属性各在网格项目（grid item）上定义网格行（grid row）和网格列（grid columns）为每一个网格项目（grid item）定义位置和空间。那么GFC有什么用呢，和table又有什么区别呢？首先同样是一个二维的表格，但GridLayout会有更加丰富的属性来控制行列，控制对齐以及更为精细的渲染语义和控制。

FFC（Flex formatting contexts）:自适应格式上下文display值为flex或者inline-flex的元素将会生成自适应容器（flex container），可惜这个牛逼的属性只有谷歌和火狐支持，不过在移动端也足够

了，至少safari和chrome还是OK的，毕竟这俩在移动端才是王道。Flex Box 由伸缩容器和伸缩项目组成。通过设置元素的 display 属性为 flex 或 inline-flex 可以得到一个伸缩容器。设置为 flex 的容器被渲染为一个块级元素，而设置为 inline-flex 的容器则渲染为一个行内元素。伸缩容器中的每一个子元素都是一个伸缩项目。伸缩项目可以是任意数量的。伸缩容器外和伸缩项目内的一切元素都不受影响。简单地说，Flexbox 定义了伸缩容器内伸缩项目该如何布局。

56.如何用 css 或 js 实现多行文本溢出省略效果，考虑兼容性

单行

SCSS

```scss
width: 300px;
overflow: hidden;
text-overflow: ellip
white-space: nowrap;
```

多行（文本一定会溢出的情况下）

SCSS

```scss
div{
  width: 300px;
  position: relative;
  line-height: 1.4em;
  height: 4.2em;
  overflow: hidden;
}
div::after{
  content: "...";
  position: absolute;
  right: 0;
  bottom: 0;
}
```

三 JavaScript

1.js基础

1.eval是做什么的?

它的功能是把对应的字符串解析成JS代码并运行;

应该避免使用eval，不安全，非常耗性能（2次，一次解析成js语句，一次执行）。

由JSON字符串转换为JSON对象的时候可以用eval，var obj =eval('('+ str +')');

2.什么是window对象? 什么是document对象?

window对象是指浏览器打开的窗口。

document对象是Documentd对象（HTML 文档对象）的一个只读引用，window对象的一个属性。

3.null，undefined 的区别?

null　表示一个对象是“没有值”的值，也就是值为“空”；

undefined　表示一个变量声明了没有初始化(赋值)；

undefined不是一个有效的JSON，而null是；

undefined的类型(typeof)是undefined；

null的类型(typeof)是object；

Javascript将未赋值的变量默认值设为undefined；

Javascript从来不会将变量设为null。它是用来让程序员表明某个用var声明的变量时没有值的。

typeof undefined

//"undefined"

undefined :是一个表示"无"的原始值或者说表示"缺少值"，就是此处应该有一个值，但是还没有定义。当尝试读取时会返回 undefined；

例如变量被声明了，但没有赋值时，就等于undefined

typeof null

//"object"

null : 是一个对象(空对象, 没有任何属性和方法)；

例如作为函数的参数，表示该函数的参数不是对象；

注意：

在验证null时，一定要使用　=== ，因为 == 无法分别 null 和　undefined

null == undefined // true

null === undefined // false

再来一个例子：

null

Q：有张三这个人么?

A：有！

Q：张三有房子么？

A：没有！

undefined

Q：有张三这个人么？

A：有！

Q: 张三有多少岁？

A: 不知道（没有被告诉）

4.["1", "2", "3"].map(parseInt) 答案是多少？

parseInt() 函数能解析一个字符串，并返回一个整数，需要两个参数 (val, radix)，

其中 radix 表示要解析的数字的基数。【该值介于 2 ~ 36 之间，并且字符串中的数字不能大于radix才能正确返回数字结果值】；

但此处 map 传了 3 个 (element, index, array),我们重写parseInt函数测试一下是否符合上面的规则。

```
function parseInt(str, radix) {
   return str+'-'+radix;
};
var a=["1", "2", "3"];
a.map(parseInt);  // ["1-0", "2-1", "3-2"] 不能大于radix
```

因为二进制里面，没有数字3,导致出现超范围的radix赋值和不合法的进制解析，才会返回NaN

所以["1", "2", "3"].map(parseInt) 答案也就是：[1, NaN, NaN]

5.事件是？IE与火狐的事件机制有什么区别？如何阻止冒泡？

1. 我们在网页中的某个操作（有的操作对应多个事件）。例如：当我们点击一个按钮就会产生一个事件。是可以被 JavaScript 侦测到的行为。

2. 事件处理机制：I.E.是事件冒泡、Firefox同时支持两种事件模型，也就是：捕获型事件和冒泡型事件；

3. ev.stopPropagation();（旧ie的方法 ev.cancelBubble = true;）

6.javascript 代码中的"use strict";是什么意思？使用它区别是什么？

use strict是一种ECMAscript 5 添加的（严格）运行模式,这种模式使得 Javascript 在更严格的条件下运行,

使JS编码更加规范化的模式,消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为。

默认支持的糟糕特性都会被禁用，比如不能用with，也不能在意外的情况下给全局变量赋值;

全局变量的显示声明,函数必须声明在顶层，不允许在非函数代码块内声明函数,arguments.callee也不允许使用；

消除代码运行的一些不安全之处，保证代码运行的安全,限制函数中的arguments修改，严格模式下的eval函数的行为和非严格模式的也不相同;

提高编译器效率，增加运行速度；

为未来新版本的Javascript标准化做铺垫。

7.Javascript中，有一个函数，执行时对象查找时，永远不会去查找原型，这个函数是？

hasOwnProperty

javaScript中hasOwnProperty函数方法是返回一个布尔值，指出一个对象是否具有指定名称的属性。此方法无法检查该对象的原型链中是否具有该属性；该属性必须是对象本身的一个成员。

使用方法：

object.hasOwnProperty(proName)

其中参数object是必选项。一个对象的实例。

proName是必选项。一个属性名称的字符串值。

如果 object 具有指定名称的属性，那么JavaScript中hasOwnProperty函数方法返回 true，反之则返回 false。

8.JSON 的了解？

JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。

它是基于JavaScript的一个子集。数据格式简单, 易于读写, 占用带宽小

如：{"age":"12", "name":"back"}

JSON字符串转换为JSON对象:

```
var obj =eval('('+ str +')');
var obj = str.parseJSON();
var obj = JSON.parse(str);
```

JSON对象转换为JSON字符串：

```
var last=obj.toJSONString();
var last=JSON.stringify(obj);
```

9.js延迟加载的方式有哪些?

defer和async、动态创建DOM方式（用得最多）、按需异步载入js

10.如何判断当前脚本运行在浏览器还是node环境中？（阿里）

this === window ? 'browser' : 'node';

通过判断Global对象是否为window，如果不为window，当前脚本没有运行在浏览器中

11.javascript的typeof返回哪些数据类型

```
alert(typeof [1, 2]); //object
  alert(typeof 'leipeng'); //string
  var i = true;
  alert(typeof i); //boolean
  alert(typeof 1); //number


  var a;
  alert(typeof a); //undefined
  function a(){;};
  alert(typeof a) //function
```

12.例举3种强制类型转换和2种隐式类型转换?

强制（parseInt(),parseFloat(),Number()）

隐式（== ,!!）

13.split()、join() 的区别

前者是切割成数组的形式，后者是将数组转换成字符串

14.数组方法pop() push() unshift() shift()

push()尾部添加 pop()尾部删除

unshift()头部添加 shift()头部删除

map() : 遍历数组中的元素, 返回一个新数组(包含回调函数返回的数据)

filter():遍历数组中的元素, 返回一个新数组(包含回调函数返回true的元素)

15.事件绑定和普通事件有什么区别

普通添加事件的方法：

```
var btn = document.getElementById("hello");
btn.onclick = function(){
   alert(1);
}
btn.onclick = function(){
   alert(2);
}
```

执行上面的代码只会alert 2

事件绑定方式添加事件：

```
var btn = document.getElementById("hello");
btn.addEventListener("click",function(){
   alert(1);
},false);
btn.addEventListener("click",function(){
   alert(2);
},false);
```

执行上面的代码会先alert 1 再 alert 2

普通添加事件的方法不支持添加多个事件，最下面的事件会覆盖上面的，而事件绑定（addEventListener）方式添加事件可以添加多个。

addEventListener不兼容低版本IE

普通事件无法取消

addEventLisntener还支持事件冒泡+事件捕获

16.IE和DOM事件流的区别

1.执行顺序不一样、

2.参数不一样

3.事件加不加on

4.this指向问题

17.IE和标准下有哪些兼容性的写法

var ev = ev || window.event

document.documentElement.clientWidth || document.body.clientWidth

var target = ev.srcElement||ev.target

18.如何阻止事件冒泡和事件默认行为

```
//阻止事件冒泡
if(typeof ev.stopPropagation=='function') {  //标准的
   ev.stopPropagation();
} else { //非标准IE
   window.event.cancelBubble = true;
}
//阻止事件默认行为
return false
```

19.window.onload 和document ready的区别

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document.ready原生中没有这个方法，jquery中有 $().ready(function),在dom文档树加载完之后执行一个函数（注意，这里面的文档树加载完不代表全部文件加载完）。

$(document).ready要比window.onload先执行

window.onload只能出来一次，$(document).ready可以出现多次

20.”==”和“===”的不同

前者会自动转换类型

后者不会

21.JavaScript是一门什么样的语言，它有哪些特点?

javaScript一种直译式脚本语言，是一种动态类型、弱类型、基于原型的语言，内置支持类型。它的解释器被称为JavaScript引擎，为浏览器的一部分，广泛用于客户端的脚本语言，最早是在HTML网页上使用，用来给HTML网页增加动态功能。JavaScript兼容于ECMA标准，因此也称为ECMAScript。

基本特点

1. 是一种解释性脚本语言（代码不进行预编译）。

2. 主要用来向HTML（标准通用标记语言下的一个应用）页面添加交互行为。

3. 可以直接嵌入HTML页面，但写成单独的js文件有利于结构和行为的分离。

跨平台特性，在绝大多数浏览器的支持下，可以在多种平台下运行（如Windows、Linux、Mac、Android、iOS等）。

22.JavaScript的数据类型都有什么?

基本数据类型：String,boolean,Number,Undefined, Null

引用数据类型：Object, Array, Function

那么问题来了，如何判断某变量是否为数组数据类型?

方法一.判断其是否具有“数组性质”，如slice()方法。可自己给该变量定义slice方法，故有时会失效

方法二.obj instanceof Array 在某些IE版本中不正确

方法三.方法一二皆有漏洞，在ECMA Script5中定义了新方法Array.isArray(), 保证其兼容性，最好的方法如下：

```
   if(typeof Array.isArray==="undefined")
{
 Array.isArray = function(arg){
      return Object.prototype.toString.call(arg)==="[object Array]"
   };
}
```

23.当一个DOM节点被点击时候，我们希望能够执行一个函数，应该怎么做?

直接在DOM里绑定事件：

在JS里通过onclick绑定：xxx.onclick = test

通过事件添加进行绑定：addEventListener(xxx, ‘click’, test)

那么问题来了，Javascript的事件流模型都有什么？

“事件冒泡”：事件开始由最具体的元素接受，然后逐级向上传播

“事件捕捉”：事件由最不具体的节点先接收，然后逐级向下，一直到最具体的

“DOM事件流”：三个阶段：事件捕捉，目标阶段，事件冒泡

24.看下列代码输出为何？解释原因。

```
var a;
alert(typeof a); // undefined
alert(b); // 报错
```

解释：Undefined是一个只有一个值的数据类型，这个值就是“undefined”，在使用var声明变量但并未对其赋值进行初始化时，这个变量的值就是undefined。而b由于未声明将报错。注意未申明的变量和声明了未赋值的是不一样的。

25.看下列代码,输出什么？解释原因。

```
var undefined;
undefined == null; // true
1 == true;  // true
2 == true;  // false
0 == false;  // true
0 == '';    // true
NaN == NaN;  // false
[] == false; // true
[] == ![];  // true
```

undefined与null相等，但不恒等（===）

一个是number一个是string时，会尝试将string转换为number

尝试将boolean转换为number，0或1

尝试将Object转换成number或string，取决于另外一个对比量的类型

所以，对于0、空字符串的判断，建议使用 “===” 。“===”会先判断两边的值类型，类型不匹配时为false。

那么问题来了，看下面的代码，输出什么，foo的值为什么？

```
var foo = "11"+2-"1";
console.log(foo);
console.log(typeof foo);
```

执行完后foo的值为111，foo的类型为String。

26.已知数组var stringArray = [“This”，“is”，“Baidu”，“Campus”]，Alert出”This is Baidu Campus”。

```
alert(stringArray.join(“”))
```

27.已知有字符串foo=”get-element-by-id”,写一个function将其转化成驼峰表示法”getElementById”

```
function combo(msg){
   var arr=msg.split("-");
   for(var i=1;i<arr.length;i++){
    arr[i]=arr[i].charAt(0).toUpperCase()+arr[i]
.substr(1,arr[i].length-1);
   }
   msg=arr.join("");
   return msg;
}
```

28.输出今天的日期，以YYYY-MM-DD的方式，比如今天是2014年9月26日，则输出2014-09-26

```
var d = new Date();
// 获取年，getFullYear()返回4位的数字
var year = d.getFullYear();
// 获取月，月份比较特殊，0是1月，11是12月
var month = d.getMonth() + 1;
var d = new Date();
// 获取年，getFullYear()返回4位的数字
```

```
var year = d.getFullYear();
// 获取月，月份比较特殊，0是1月，11是12月
var month = d.getMonth() + 1;
```

29.将字符串”<tr><td>{$id}</td><td>{$name}</td></tr>”中的{$id}替换成10，{$name}替换成Tony（使用正则表达式）

```
"<tr><td>{$id}</td><td>{$id}_{$name}</td></tr>".replace(/{\$id}/g, '10').replace(/{\$name}/g, 'Tony');
```

30.为了保证页面输出安全，我们经常需要对一些特殊的字符进行转义，请写一个函数escapeHtml，将<, >, &, “进行转义

```
function escapeHtml(str) {
return str.replace(/[<>”&]/g, function(match) {
  switch (match) {
        case “<”:
          return “&lt;”;
        case “>”:
          return “&gt;”;
        case “&”:
          return “&amp;”;
        case “\””:
          return “&quot;”;
   }
 });
}
```

31.foo = foo||bar ，这行代码是什么意思？为什么要这样写？

如果foo存在，值不变，否则把bar的值赋给foo。

短路表达式：作为”&&”和”||”操作符的操作数表达式，这些表达式在进行求值时，只要最终的结果已经可以确定是真或假，求值过程便告终止，这称之为短路求值。

32.看下列代码，将会输出什么?(变量声明提升)

```
var foo = 1;
```

```
(function(){
   console.log(foo);
   var foo = 2;
   console.log(foo);
})()
```

答案：输出undefined 和 2。上面代码相当于：

```
var foo = 1;
(function(){
   var foo;
   console.log(foo); //undefined
   foo = 2;
   console.log(foo); // 2;
})()
```

函数声明与变量声明会被JavaScript引擎隐式地提升到当前作用域的顶部，但是只提升名称不会提升赋值部分。

33.用js实现随机选取10–100之间的10个数字，存入一个数组，并排序。

```
function randomNub(aArray, len, min, max) {
        if (len >= (max - min)) {
           return '超过' + min + '-' + max + '之间的个数范围' + (max - min - 1) + '个的总数';
        }
        if (aArray.length >= len) {
           aArray.sort(function(a, b) {
              return a - b
           });
           return aArray;
        }
        var nowNub = parseInt(Math.random() * (max - min - 1)) + (min + 1);
        for (var j = 0; j < aArray.length; j++) {
           if (nowNub == aArray[j]) {
```

```
            randomNub(aArray, len, min, max);
            return;
          }
        }
        aArray.push(nowNub);
        randomNub(aArray, len, min, max);
        return aArray;

         var arr=[];
         randomNub(arr,10,10,100);
```

34.把两个数组合并，并删除第二个元素。

```
var array1 = ['a','b','c'];
var bArray = ['d','e','f'];
var cArray = array1.concat(bArray);
cArray.splice(1,1);
```

35.有这样一个URL：http://item.taobao.com/item.htm?a=1&b=2&c=&d=xxx&e，请写一段JS程序提取URL中的各个GET参数(参数名和参数个数不确定)，将其按key-value形式返回到一个json结构中，如{a:’1′, b:’2′, c:”, d:’xxx’, e:undefined}。

```
function serilizeUrl(url) {
  var urlObject = {};
  if (/\?/.test(url)) {
    var urlString = url.substring(url.indexOf("?") + 1);
    var urlArray = urlString.split("&");
    for (var i = 0, len = urlArray.length; i < len; i++) {
      var urlItem = urlArray[i];
      var item = urlItem.split("=");
      urlObject[item[0]] = item[1];
    }
    return urlObject;
```

```
    }
    return null;
}
```

36.正则表达式构造函数var reg=new RegExp(“xxx”)与正则表达字面量var reg=//有什么不同？匹配邮箱的正则表达式？

当使用RegExp()构造函数的时候，不仅需要转义引号（即\”表示”），并且还需要双反斜杠（即\\表示一个\）。使用正则表达字面量的效率更高。

邮箱的正则匹配：

```
var regMail = /^([a-zA-Z0-9_-])+@([a-zA-Z0-9_-])+((.[a-zA-Z0-9_-]{2,3}){1,2})$/;
```

37.写一个function，清除字符串前后的空格。

```
if (!String.prototype.trim) {
    String.prototype.trim = function() {
    return this.replace(/^\s+/, "").replace(/\s+$/,"");
 }
}
//测试
var str = " \t\n test string ".trim();
alert(str == "test string"); // alerts "true"
```

38.以下两个变量a和b，a+b的哪个结果是NaN？

A、var a=undefined; b=NaN

B、var a= ‘123’; b=NaN

C、var a =undefined , b =NaN

D、var a=NaN , b='undefined'

39.下面的JavaScript语句中，（）实现检索当前页面中的表单元素中的所有文本框，并将它们全部清空

```
A. for(vari=0;i< form1.elements.length;i++) {
if(form1.elements.type==”text”)
form1.elements.value=”";}
B. for(vari=0;i<document.forms.length;i++) {
if(forms[0].elements.type==”text”)
```

forms[0].elements.value=”";

}

C. if(document.form.elements.type==”text”)

form.elements.value=”";

D. for(vari=0;i<document.forms.length; i++){

for(var j=0;j<document.forms.elements.length; j++){

if(document.forms.elements[j].type==”text”)

document.forms.elements[j].value=”";

}

}

40.typeof运算符返回值中有一个跟javascript数据类型不一致，它是

Array

41.写出函数DateDemo的返回结果，系统时间假定为今天

function DateDemo(){

var d, s="今天日期是：";

d = new Date();

s += d.getMonth() +1+ "/";

s += d.getDate() + "/";

s += d.getFullYear();

return s;

}

结果：今天日期是：7/21/2016

42.写出简单描述html标签（不带属性的开始标签和结束标签）的正则表达式，并将以下字符串中的html标签去除掉

var str = “ 这里是div 里面的段落 ”;

<scripttype=”text/javascript”>

var reg = /<\/?\w+\/?>/gi;

var str = “ 这里是div 里面的段落 ”;

alert(str.replace(reg,”));

</script>

43.截取字符串abcdefg的efg

```
alert('abcdefg'.substring(4));
```

44.简述创建函数的几种方式

第一种（函数声明）：

```
function sum1(num1,num2){
  return num1+num2;
}
```

第二种（函数表达式）：

```
var sum2 = function(num1,num2){
  return num1+num2;
}
```

第三种（函数对象方式）：

```
var sum3 = new Function("num1","num2","return num1+num2");
```

45.Javascript如何实现继承?

1.构造继承法

2.原型继承法

3.实例继承法

46.Javascript创建对象的几种方式?

1、var obj = {};（使用json创建对象）

如：obj.name = '张三';

```
obj.action = function ()
{
alert('吃饭');
};
```

2、var obj = new Object();（使用Object创建对象）

如：obj.name = '张三';

```
obj.action = function ()
```

{

alert('吃饭');

};

3、通过构造函数创建对象。

(1)、使用this关键字

如：var obj = function (){

this.name ='张三';

this.age = 19;

this.action = function ()

{

alert('吃饭');

};

}

(2)、使用prototype关键字

如：function obj (){}

obj.prototype.name ='张三';

obj.prototype.action=function ()

{

alert('吃饭');

};

4、使用内置对象创建对象。

如：var str = new String("实例初始化String");

var str1 = "直接赋值的String";

var func = new Function("x","alert(x)");//示例初始化func

var obj = new Object();//示例初始化一个Object

47.js延迟加载的方式有哪些?

1. defer和async

2. 动态创建DOM方式（创建script，插入到DOM中，加载完毕后callBack）

3. 按需异步载入js

48.哪些操作会造成内存泄漏?

内存泄漏指任何对象在您不再拥有或需要它之后仍然存在。

垃圾回收器定期扫描对象，并计算引用了每个对象的其他对象的数量。如果一个对象的引用数量为 0（没有其他对象引用过该对象），或对该对象的惟一引用是循环的，那么该对象的内存即可回收。

1. setTimeout 的第一个参数使用字符串而非函数的话，会引发内存泄漏。

2. 闭包

3. 控制台日志

4. 循环（在两个对象彼此引用且彼此保留时，就会产生一个循环）

49.判断一个字符串中出现次数最多的字符，统计这个次数

```
var str = 'asdfssaaasasasasaa';
var json = {};
for (var i = 0; i < str.length; i++) {
    if(!json[str.charAt(i)]){
        json[str.charAt(i)] = 1;
    }else{
        json[str.charAt(i)]++;
    }
};
var iMax = 0;
var iIndex = '';
for(var i in json){
    if(json[i]>iMax){
        iMax = json[i];
        iIndex = i;
    }
}
alert('出现次数最多的是:'+iIndex+'出现'+iMax+'次');
```

50.写一个获取非行间样式的函数

```
function getStyle(obj,attr,value)
{
  if(!value)
  {
    if(obj.currentStyle)
    {
      return obj.currentStyle(attr);
    }
    else{
      obj.getComputedStyle(attr,false);
    }
  }
  else
  {
    obj.style[attr] = value;
  }
}
```

51.字符串反转，如将 '12345678' 变成 '87654321'

```
//思路：先将字符串转换为数组 split()，利用数组的反序函数 reverse()颠倒数组，再利用 jion() 转换为字符串
var str = '12345678';
str = str.split('').reverse().join('');
```

52.将数字 12345678 转化成 RMB形式 如： 12,345,678

```
//思路：先将数字转为字符， str= str + '' ;
//利用反转函数，每三位字符加一个 ','最后一位不加； re()是自定义的反转函数，最后再反转回去！
function re(str) {
  str += '';
  return str.split("").reverse().join("");
```

```
}


function toRMB(num) {
  var tmp='';
  for (var i = 1; i <= re(num).length; i++) {
    tmp += re(num)[i - 1];
    if (i % 3 == 0 && i != re(num).length) {
      tmp += ',';
    }
  }
  return re(tmp);
}
```

53.生成5个不同的随机数；

```
//思路：5个不同的数，每生成一次就和前面的所有数字相比较，如果有相同的，则放弃当前生成的数字！
var num1 = [];
for(var i = 0; i < 5; i++){
  num1[i] = Math.floor(Math.random()*10) + 1; //范围是 [1, 10]
  for(var j = 0; j < i; j++){
    if(num1[i] == num1[j]){
      i--;
    }
  }
}
```

54.去掉数组中重复的数字

方法一:

```
//思路：每遍历一次就和之前的所有做比较，不相等则放入新的数组中！
//这里用的原型 个人做法；
Array.prototype.unique = function(){
```

```
    var len = this.length,
        newArr = [],
        flag = 1;
    for(var i = 0; i < len; i++, flag = 1){
        for(var j = 0; j < i; j++){
            if(this[i] == this[j]){
                flag = 0;       //找到相同的数字后，不执行添加数据
            }
        }
        flag ? newArr.push(this[i]) : '';
    }
    return newArr;
}
```

方法二：

```
var arr=[1,2,3,3,4,4,5,5,6,1,9,3,25,4];
Array.prototype.unique2 = 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
    {
        //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
        if (n.indexOf(this[i]) == -1) n.push(this[i]);
    }
    return n;
}
var newArr2=arr.unique2(arr);
alert(newArr2); //输出1,2,3,4,5,6,9,25
```

55.阶乘函数；

```
//原型方法
Number.prototype.N = function(){
   var re = 1;
   for(var i = 1; i <= this; i++){
      re *= i;
   }
   return re;
}
var num = 5;
alert(num.N());
```

56.window.location.search() 返回的是什么?

http://localhost:8080/xxx?ver=1.0&id=123

返回值：?ver=1.0&id=timlq 也就是问号后面的部分

57.window.location.reload() 作用?

新当前页面。

58.javascript 中的垃圾回收机制?

在Javascript中，如果一个对象不再被引用，那么这个对象就会被GC回收。如果两个对象互相引用，而不再 被第3者所引用，那么这两个互相引用的对象也会被回收。因为函数a被b引用，b又被a外的c引用，这就是为什么 函数a执行后不会被回收的原因。

59.精度问题: JS 精度不能精确到 0.1 所以 。。。。同时存在于值和差值中

```
var n = 0.3,m = 0.2, i = 0.2, j = 0.1;
alert((n - m) == (i - j)); //false
alert((n-m) == 0.1); //false
alert((i-j)==0.1); //true
```

60.计算字符串字节数：

```
new function(s){
   if(!arguments.length||!s) return null;
   if(""==s) return 0;
```

```
    var l=0;
    for(var i=0;i<s.length;i++){
        if(s.charCodeAt(i)>255) l+=2; else l+=1;  //charCodeAt()得到的是unCode码
    }    //汉字的unCode码大于 255bit 就是两个字节
    alert(l);
}("hello world!");
```

61.匹配输入的字符：第一个必须是字母或下划线开头，长度5-20

```
var reg = /^[a-zA-Z_][a-zA-Z0-9_]{5,20}/,
        name1 = 'leipeng',
        name2 = '0leipeng',
        name3 = '你好leipeng',
        name4 = 'hi';


    alert(reg.test(name1));
    alert(reg.test(name2));
    alert(reg.test(name3));
    alert(reg.test(name4));
```

62.如何在HTML中添加事件，几种方法?

1、标签之中直接添加 onclick="fun()";

2、JS添加 Eobj.onclick = method;

3、绑定事件 IE： obj.attachEvent('onclick', method)；

FF: obj.addEventListener('click', method, false);

63.你如何优化自己的代码?

代码重用

避免全局变量（命名空间，封闭空间，模块化mvc..）

拆分函数避免函数过于臃肿

注释

64.使用js实现这样的效果：在文本域里输入文字时，当按下enter键时不换行，而是替换成“{{enter}}”,(只需要考虑在行尾按下enter键的情况).

```
<html>
<head>
  <script>
    function back(ele,event){
      event = event || window.event;
      if(event.keyCode==13){
        event.returnValue = false;
        ele.value+="{{enter}}"
        return false;
      }
    }
  </script>
</head>
<body>
<textarea rows="3" cols="40" id="te" onkeypress="back(this,event);"></textarea>
</body>
</html>
```

65.简述readyonly与disabled的区别

ReadOnly和Disabled的作用是使用户不能够更改表单域中的内容.

但是二者还是有着一些区别的：

1、Readonly只针对input(text/password)和textarea有效，而disabled对于所有的表单元素有效，包括select,radio,checkbox,button等。

2、在表单元素使用了disabled后，我们将表单以POST或者GET的方式提交的话，这个元素的值不会被传递出去，而readonly会将该值传递出去

66.为什么扩展javascript内置对象不是好的做法?

因为你不知道哪一天浏览器或javascript本身就会实现这个方法，而且和你扩展的实现有不一致的表现。到时候你的javascript代码可能已经在无数个页面中执行了数年，而浏览器的实现导致所有使用扩展原型的代码都崩溃了。。。

67.什么是三元表达式？“三元”表示什么意思？

三元运算符:

三元如名字表示的三元运算符需要三个操作数。

语法是 条件 ? 结果1 : 结果2;. 这里你把条件写在问号(?)的前面后面跟着用冒号(:)分隔的结果1和结果2。满足条件时结果1否则结果2。

68.变量的命名规范以及命名推荐

变量，函数，方法：小写开头，以后的每个单词首字母大写 （驼峰）

构造函数，class：每个单词大写开头

基于实际情况，以动词，名词，谓词来命名。尽量言简意骇，以命名代替注释

69.三种弹窗的单词以及三种弹窗的功能

1.alert

```
//弹出对话框并输出一段提示信息
  function ale() {
    //弹出一个对话框
    alert("提示信息！");

  }
```

2.confirm

```
  //弹出一个询问框，有确定和取消按钮
  function firm() {
    //利用对话框返回的值 （true 或者 false）
    if (confirm("你确定提交吗？")) {
      alert("点击了确定");
    }
    else {
      alert("点击了取消");
    }
```

```
}
```

3.prompt

```
//弹出一个输入框，输入一段文字，可以提交
function prom() {
    var name = prompt("请输入您的名字", ""); //将输入的内容赋给变量 name ，

    //这里需要注意的是，prompt有两个参数，前面是提示的话，后面是当对话框出来后，在对话框里的默认值
    if (name)//如果返回的有内容
    {
        alert("欢迎您：" + name)
    }
}
```

70.主流浏览器内核

IE trident　　火狐gecko　谷歌苹果webkit　Opera：Presto

71.JavaScript的循环语句有哪些?

for,for..in,while,do...while

72.闭包：下面这个ul，如何点击每一列的时候alert其index?

```
<ul id="test">
    <li>这是第一条</li>
    <li>这是第二条</li>
    <li>这是第三条</li>
</ul>
//js
 window.onload = function() {
    var lis = document.getElementById('test').children;
    for (var i = 0; i < lis.length; i++) {
```

```
        lis[i].onclick = (function(i) {
            return function() {
                alert(i)
            };
        })(i);
    };
}
```

73.列出3条以上ff和IE的脚本兼容问题

(1) window.event：

表示当前的事件对象，IE有这个对象，FF没有，FF通过给事件处理函数传递事件对象

(2) 获取事件源

IE用srcElement获取事件源，而FF用target获取事件源

(3) 添加，去除事件

IE：element.attachEvent(“onclick”, function) element.detachEvent(“onclick”, function)

FF：element.addEventListener(“click”, function, true)
element.removeEventListener(“click”, function, true)

(4) 获取标签的自定义属性

IE：div1.value或div1[“value”]

FF：可用div1.getAttribute(“value”)

74.在Javascript中什么是伪数组？如何将伪数组转化为标准数组？

伪数组（类数组）：无法直接调用数组方法或期望length属性有什么特殊的行为，但仍可以对真正数组遍历方法来遍历它们。典型的是函数的argument参数，还有像调用getElementsByTagName,document.childNodes之类的,它们都返回NodeList对象都属于伪数组。可以使用Array.prototype.slice.call(fakeArray)将数组转化为真正的Array对象。

75.请写出一个程序，在页面加载完成后动态创建一个form表单，并在里面添加一个input对象并给它任意赋值后义post方式提交到：http://127.0.0.1/save.php

```
window.onload=function(){
```

```
    var form=document.createElement("form");
    form.setAttribute("method", "post");
    form.setAttribute("action", "http://127.0.0.1/save.php");

    var input=document.createElement("input");
    form.appendChild(input);
    document.body.appendChild(form);
    input.value="cxc";
    form.submit();//提交表单
}
```

76.用JavaScript实现冒泡排序。数据为23、45、18、37、92、13、24

```
//升序算法
function sort(arr){
    for (var i = 0; i <arr.length; i++) {
        for (var j = 0; j <arr.length-i; j++) {
            if(arr[j]>arr[j+1]){
                var c=arr[j];//交换两个变量的位置
                arr[j]=arr[j+1];
                arr[j+1]=c;
            }
        };
    };
    return arr.toString();
}
console.log(sort([23,45,18,37,92,13,24]));
```

77.前端代码优化的方法

```
var User = {
    count = 1，
```

```
  getCount：function（）{
      return this.count;
  }
}
console.log(User.getCount());
var func = User.getCount;
console.log(func());
```

1 undefined（因为是window对象执行了func函数）；

78.用程序实现找到html中id名相同的元素?

```
<body>
  <form id='form1'>
    <div id='div1'></div>
    <div id='div2'></div>
    <div id='div3'></div>
    <div id='div4'></div>
    <div id='div5'></div>
    <div id='div3'>id名重复的元素</div>
  </form>
</body>
var nodes=document.querySelectorAll("#form1>*");
for(var i=0,len=nodes.length;i<len;i++){
  var attr=nodes[i].getAttribute("id");
  var s=1;
  for(var j=i+1;j<len;j++){
    if(nodes[j].getAttribute("id")==attr){
      s++;
      alert("id为："+attr+"的元素出现"+s+"次");
    }
```

```
  }
}
```

79.程序中捕获异常的方法?

window.error

try{}catch(){}finally{}

80.下列控制台都输出什么

第1题:

```
function setName(){
  name="张三";
}
setName();
console.log(name);
```

答案："张三"

第2题:

```
//考点：1、变量声明提升 2、变量搜索机制
var a=1;
function test(){
  console.log(a);
  var a=1;
}
test();
```

答案：undefined

第3题:

```
var b=2;
function test2(){
  window.b=3;
  console.log(b);
}
```

```
test2();
```

答案：3

第4题：

```
c=5;//声明一个全局变量c
function test3(){
   window.c=3;
   console.log(c);      //答案：undefined，原因：由于此时的c是一个局部变量c，并且没有被赋值
   var c;
   console.log(window.c);//答案：3，原因：这里的c就是一个全局变量c
}
test3();
```

第5题：

```
var arr = [];
arr[0]  = 'a';
arr[1]  = 'b';
arr[10] = 'c';
alert(arr.length); //答案：11
console.log(arr[5]); //答案：undefined
```

第6题：

```
var a=1;
console.log(a++);      //答案：1
console.log(++a);      //答案：3
```

第7题：

```
console.log(null==undefined); //答案：true
console.log("1"==1);     //答案：true，因为会将数字1先转换为字符串1
console.log("1"===1);    //答案：false，因为数据类型不一致
```

第8题：

```
typeof 1;      "number"
typeof "hello";      "string"
```

typeof /[0-9]/;　　"object"

typeof {};　　"object"

typeof null;　　"object"

typeof undefined;　"undefined"

typeof [1,2,3];　"object"

typeof function(){}; //"function"

第9题：

parseInt(3.14);　　//3

parseFloat("3asdf");　　//3

parseInt("1.23abc456");

parseInt(true);//"true" NaN

第10题：

//考点：函数声明提前

```
function bar() {
  return foo;
  foo = 10;
  function foo() {}
  //var foo = 11;
}
```

alert(typeof bar());//"function"

第11题：

//考点：函数声明提前

```
var foo = 1;
function bar() {
  foo = 10;
  return;
  function foo() {}
}
```

bar();

```
alert(foo);//答案：1
第12题：
console.log(a);//是一个函数
var a = 3;
function a(){}
console.log(a);////3
第13题：
//考点：对arguments的操作
function foo(a) {
  arguments[0] = 2;
  alert(a);//答案：2，因为：a、arguments是对实参的访问，b、通过arguments[i]可以修改指定实参的值
}
foo(1);
第14题：
function foo(a) {
  alert(arguments.length);//答案：3，因为arguments是对实参的访问
}
foo(1, 2, 3);
第15题
bar();//报错
var foo = function bar(name) {
  console.log("hello"+name);
  console.log(bar);
};
//alert(typeof bar);
foo("world");//"hello"
console.log(bar);//undefined
console.log(foo.toString());
```

```
bar();//报错
```

第16题：

```
function test(){
   console.log("test函数");
}
setTimeout(function(){
   console.log("定时器回调函数");
}, 0)
test();
```

结果：

```
test函数
定时器回调函数
```

81.请说出三种减低页面加载时间的方法

1、压缩css、js文件

2、合并js、css文件，减少http请求

3、外部js、css文件放在最底下

4、减少dom操作，尽可能用变量替代不必要的dom操作

82.实现一个函数clone，可以对JavaScript中的5种主要的数据类型（包括Number、String、Object、Array、Boolean）进行值复制

```
  // 方法一：
Object.prototype.clone = function(){
  var o = this.constructor === Array ? [] : {};
  for(var e in this){
    o[e] = typeof this[e] === "object" ? this[e].clone() : this[e];
  }
  return o;
}
//方法二：
  /**
```

```
 * 克隆一个对象
 * @param Obj
 * @returns
 */
function clone(Obj) {
  var buf;
  if (Obj instanceof Array) {
    buf = [];//创建一个空的数组
    var i = Obj.length;
    while (i--) {
      buf[i] = clone(Obj[i]);
    }
    return buf;
  }else if (Obj instanceof Object){
    buf = {};//创建一个空对象
    for (var k in Obj) { //为这个对象添加新的属性
      buf[k] = clone(Obj[k]);
    }
    return buf;
  }else{ //普通变量直接赋值
    return Obj;
  }
}
```

83.如何消除一个数组里面重复的元素?

```
var arr=[1,2,3,3,4,4,5,5,6,1,9,3,25,4];
  function deRepeat(){
    var newArr=[];
    var obj={};
```

```
        var index=0;
        var l=arr.length;
        for(var i=0;i<l;i++){
          if(obj[arr[i]]==undefined)
           {
            obj[arr[i]]=1;
             newArr[index++]=arr[i];
           }
          else if(obj[arr[i]]==1)
           continue;
        }
        return newArr;
      }
      var newArr2=deRepeat(arr);
      alert(newArr2); //输出1,2,3,4,5,6,9,25
```

84.定义一个log方法，让它可以代理console.log的方法。

```
function log(msg)  {
  console.log(msg);
}
log("hello world!") // hello world!
```

如果要传入多个参数呢？显然上面的方法不能满足要求，所以更好的方法是：

```
function log(){
  console.log.apply(console, arguments);
};
```

到此，追问apply和call方法的异同。

对于apply和call两者在作用上是相同的，即是调用一个对象的一个方法，以另一个对象替换当前对象。将一个函数的对象上下文从初始的上下文改变为由 thisObj 指定的新对象。

但两者在参数上有区别的。对于第一个参数意义都一样，但对第二个参数： apply传入的是一个参数数组，也就是将多个参数组合成为一个数组传入，而call则作为call的参数传入（从第二个参数开

始）。 如 func.call(func1,var1,var2,var3)对应的apply写法为：func.apply(func1,[var1,var2,var3])。

85.原生JS的window.onload与Jquery的$(document).ready(function(){})有什么不同？如何用原生JS实现Jq的ready方法？

window.onload()方法是必须等到页面内包括图片的所有元素加载完毕后才能执行。

$(document).ready()是DOM结构绘制完毕后就执行，不必等到加载完毕。

```
/*
 * 传递函数给whenReady()
 * 当文档解析完毕且为操作准备就绪时，函数作为document的方法调用
 */
var whenReady = (function() {          //这个函数返回whenReady()函数
    var funcs = [];          //当获得事件时，要运行的函数
    var ready = false;        //当触发事件处理程序时,切换为true
    //当文档就绪时,调用事件处理程序
    function handler(e) {
        if(ready) return;     //确保事件处理程序只完整运行一次
        //如果发生onreadystatechange事件，但其状态不是complete的话,那么文档尚未准备好
        if(e.type === 'onreadystatechange' && document.readyState !== 'complete') {
            return;
        }
        //运行所有注册函数
        //注意每次都要计算funcs.length
        //以防这些函数的调用可能会导致注册更多的函数
        for(var i=0; i<funcs.length; i++) {
            funcs[i].call(document);
        }
        //事件处理函数完整执行,切换ready状态, 并移除所有函数
        ready = true;
        funcs = null;
```

```
    }
    //为接收到的任何事件注册处理程序
    if(document.addEventListener) {
        document.addEventListener('DOMContentLoaded', handler, false);
        document.addEventListener('readystatechange', handler, false);          //IE9+
        window.addEventListener('load', handler, false);
    }else if(document.attachEvent) {
        document.attachEvent('onreadystatechange', handler);
        window.attachEvent('onload', handler);
    }
    //返回whenReady()函数
    return function whenReady(fn) {
        if(ready) { fn.call(document); }
        else { funcs.push(fn); }
    }
})();
```

如果上述代码十分难懂，下面这个简化版：

```
function ready(fn){
    if(document.addEventListener) {//标准浏览器
        document.addEventListener('DOMContentLoaded', function() {
            //注销事件, 避免反复触发
            document.removeEventListener('DOMContentLoaded',arguments.callee, false);
            fn();//执行函数
        }, false);
    }else if(document.attachEvent) {//IE
        document.attachEvent('onreadystatechange', function() {
            if(document.readyState == 'complete') {
                document.detachEvent('onreadystatechange', arguments.callee);
                fn();//函数执行
```

```
        }
    });
  }
};
```

86.（设计题）想实现一个对页面某个节点的拖曳？如何做？（使用原生JS）

回答出概念即可，下面是几个要点

给需要拖拽的节点绑定mousedown, mousemove, mouseup事件

mousedown事件触发后，开始拖拽

mousemove时，需要通过event.clientX和clientY获取拖拽位置，并实时更新位置

mouseup时，拖拽结束

需要注意浏览器边界的情况

87.数组和对象有哪些原生方法，列举一下？

Array.concat( ) 连接数组

Array.join( ) 将数组元素连接起来以构建一个字符串

Array.length 数组的大小

Array.pop( ) 删除并返回数组的最后一个元素

Array.push( ) 给数组添加元素

Array.reverse( ) 颠倒数组中元素的顺序

Array.shift( ) 将元素移出数组

Array.slice( ) 返回数组的一部分

Array.sort( ) 对数组元素进行排序

Array.splice( ) 插入、删除或替换数组的元素

Array.toLocaleString( ) 把数组转换成局部字符串

Array.toString( ) 将数组转换成一个字符串

Array.unshift( ) 在数组头部插入一个元素

Object.hasOwnProperty( ) 检查属性是否被继承

Object.isPrototypeOf( ) 一个对象是否是另一个对象的原型

Object.propertyIsEnumerable( ) 是否可以通过for/in循环看到属性

Object.toLocaleString( ) 返回对象的本地字符串表示

Object.toString( ) 定义一个对象的字符串表示

Object.valueOf( ) 指定对象的原始值

88.JS 怎么实现一个类。怎么实例化这个类

严格来讲js中并没有类的概念，不过js中的函数可以作为构造函数来使用，通过new来实例化，其实函数本身也是一个对象。

89.在JS中有哪些会被隐式转换为false

Undefined、null、关键字false、NaN、零、空字符串

90.外部JS文件出现中文字符，会出现什么问题，怎么解决?

会出现乱码，加charset=”utf-8”；

91.写一个通用的事件侦听器函数

```
// event(事件)工具集，来源：https://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;
  }
};
```

92. JSON 的了解

JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。它是基于JavaScript的一个子集。数据格式简单, 易于读写, 占用带宽小

{'age':'12', 'name':'back'}

93. 说说 typeof 与 instanceof 区别?

Plain Text

```
一、typeof
typeof操作符返回一个字符串，表示未经计算的操作数的类型.
虽然typeof null为object，但这只是JavaScript存在的一个悠久Bug，不代表null就是引用数据类型，并且null本身也不是对象
引用类型数据，用typeof来判断的话，除了function会被识别出来之外，其余的都输出object
如果我们想要判断一个变量是否存在，可以使用typeof
二、instanceof
instanceof运算符用于检测构造函数的prototype属性是否出现在某个实例对象的原型链上
三、区别
typeof与instanceof都是判断数据类型的方法，区别如下：
typeof会返回一个变量的基本类型，instanceof返回的是一个布尔值
instanceof可以准确地判断复杂引用数据类型，但是不能正确判断基础数据类型
而typeof也存在弊端，它虽然可以判断基础数据类型（null除外），但是引用数据类型中，除了function类型以外，其他的也无法判断
可以看到，上述两种方法都有弊端，并不能满足所有场景的需求
如果需要通用检测数据类型，可以采用Object.prototype.toString，调用该方法，统一返回格式“[object Xxx]”的字符串
```

94. 说说Javascript中的继承?

Plain Text

```
继承（inheritance）是面向对象软件技术当中的一个概念
如果一个类别B“继承自”另一个类别A，就把这个B称为“A的子类”，而把A称为“B的父类别”也可以称“A是B的超类”
继承的优点
继承可以使得子类具有父类别的各种属性和方法，而不需要再次编写相同的代码
在子类别继承父类别的同时，可以重新定义某些属性，并重写某些方法，即覆盖父类别的原有属性和方法，使其获得与父类别不同的功能
虽然JavaScript并不是真正的面向对象语言，但它天生的灵活性，使应用场景更加丰富
```

94. Javascript继承的实现方式



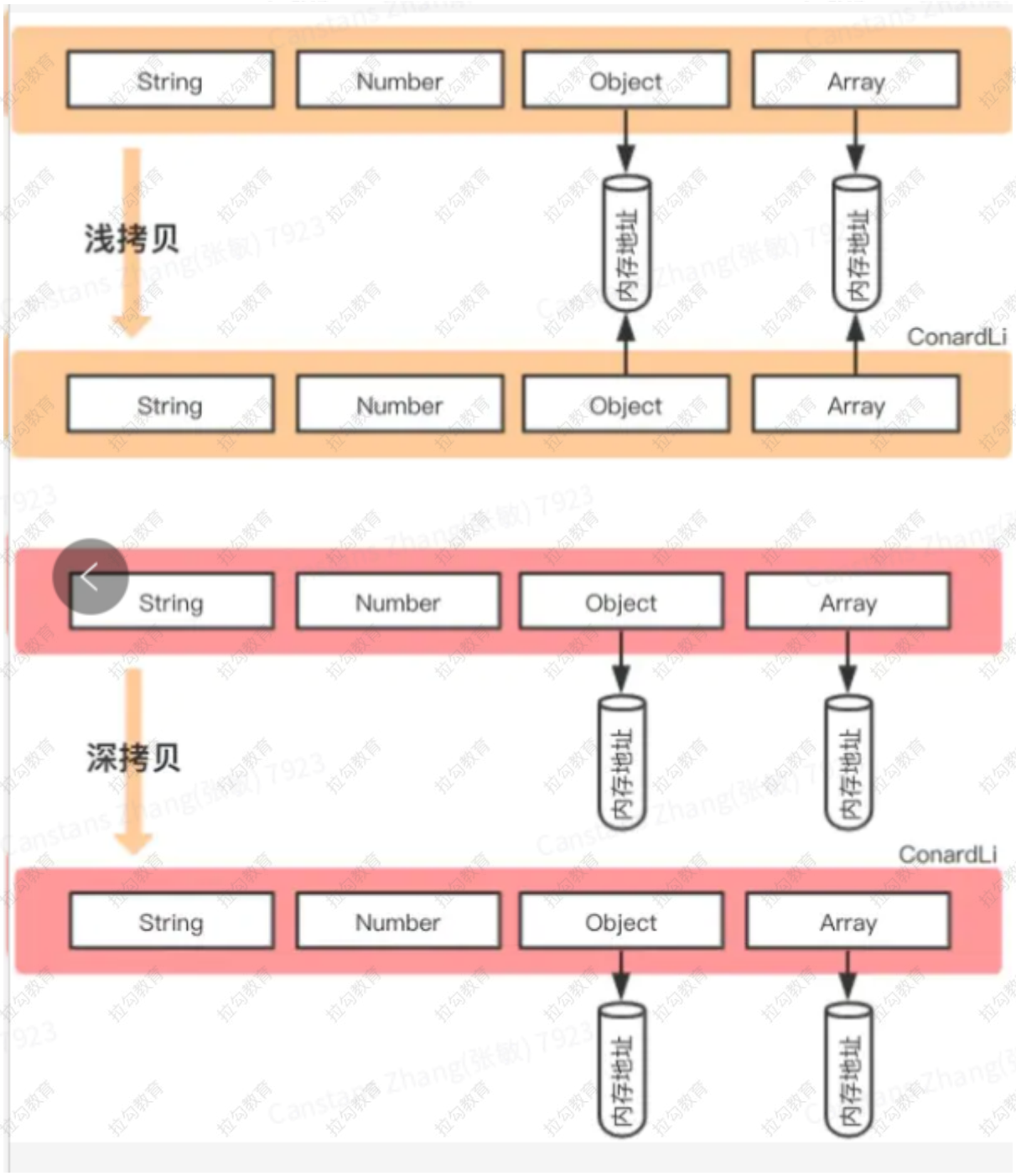
通过`Object.create`来划分不同的继承方式，最后的寄生式组合继承方式是通过组合继承改造之后的最优继承方式，而`extends`的语法糖和寄生组合继承的方式基本类似

Markdown

```
原型链继承
构造函数继承（借助 call）
组合继承
原型式继承
寄生式继承
寄生组合式继承
原型链继承
原型链继承是比较常见的继承方式之一，其中涉及的构造函数、原型和实例，三者之间存在着一定的关系，即每一个构造函数都有一个原型对象，原型对象又包含一个指向构造函数的指针，而实例则包含一个原型对象的指针
构造函数继承
借助call调用Parent函数
组合继承
前面我们讲到两种继承方式，各有优缺点。组合继承则将前两种方式继承起来
原型式继承
这里主要借助Object.create方法实现普通对象的继承
寄生式继承
寄生式继承在上面继承基础上进行优化，利用这个浅拷贝的能力再进行增强，添加一些方法
寄生组合式继承
寄生组合式继承，借助解决普通对象的继承问题的Object.create方法，在几种继承方式的优缺点基础上进行改造，这也是所有继承方式里面相对最优的继承方式

```

95. 深拷贝浅拷贝的区别?


String
Number
Object
Array
浅拷贝
内存地址
内存地址
ConardLi
String
Number
Object
Array
String
Number
Object
Array
深拷贝
内存地址
内存地址
ConardLi
String
Number
Object
Array
内存地址
内存地址

址

JavaScript

```
一、数据类型存储
JavaScript中存在两大数据类型：
基本类型
引用类型
基本类型数据保存在在栈内存中
引用类型数据保存在堆内存中，引用数据类型的变量是一个指向堆内存中实际对象的引用，存在栈中
二、浅拷贝
浅拷贝，指的是创建新的数据，这个数据有着原始数据属性值的一份精确拷贝
如果属性是基本类型，拷贝的就是基本类型的值。如果属性是引用类型，拷贝的就是内存地址
即浅拷贝是拷贝一层，深层次的引用类型则共享内存地址
在JavaScript中，存在浅拷贝的现象有：
Object.assign
Array.prototype.slice(),Array.prototype.concat()
使用拓展运算符实现的复制
三、深拷贝
深拷贝开辟一个新的栈，两个对象属性完成相同，但是对应两个不同的地址，修改一个对象的属性，不会改变另一个对象的属性
常见的深拷贝方式有：
_.cloneDeep()
jQuery.extend()
JSON.stringify()
手写循环递归
四 前提为拷贝类型为引用类型的情况下：
浅拷贝是拷贝一层，属性为对象时，浅拷贝是复制，两个对象指向同一个地址
深拷贝是递归拷贝深层次，属性为对象时，深拷贝是新开栈，两个对象指向不同的地
```

96. 谈谈 JavaScript 中的类型转换机制

JavaScript

```
JS中有六种简单数据类型：undefined、null、boolean、string、number、symbol，以及引用类型：object
常见的类型转换有：
强制转换（显示转换）
自动转换（隐式转换）
二、显示转换
显示转换，即我们很清楚可以看到这里发生了类型的转变，常见的方法有：
Number()
parseInt()
String()
Boolean()
三、隐式转换
在隐式转换中，我们可能最大的疑惑是 ：何时发生隐式转换?
我们这里可以归纳为两种情况发生隐式转换的场景：
比较运算（==、!=、>、<）、if、while需要布尔值地方
算术运算（+、-、*、/、%）
除了上面的场景，还要求运算符两边的操作数不是同一类型
```

97. 如何实现骨架屏，说说你的思路

Apache

```
1.如果是首屏可以在index.html中手写骨架屏样式
2.如果是其他页面，可以让UI做一个小的SVG图
3.可以使用组件库中的骨架屏组件 4.可以使用饿了么团队开源的根据页面样式生成骨架屏的工具[还可配置生效路由]
```

98. 介绍下前端加密的常见场景和方法

HTML

```
先，加密的目的，简而言之就是将明文转换为密文、甚至转换为其他的东西，用来隐藏明文内容本身，防止其他人直接获取到敏感明文信息、或者提高其他人获取到明文信息的难度。<br />通常我们提到加密会想到密码加密、HTTPS 等关键词，这里从场景和方法分别提一些我的个人见解。
<a name="867e6c89"></a>
场景-密码传输
前端密码传输过程中如果不加密，在日志中就可以拿到用户的明文密码，对用户安全不太负责。<br />这种加密其实相对比较简单，可以使用 PlanA-前端加密、后端解密后计算密码字符串的MD5/MD6存入数据库；也可以 PlanB-直接前端使用一种稳定算法加密成唯一值、后端直接将加密结果进行MD5/MD6，全程密码明文不出现在程序中。
PlanA<br />使用 Base64 / Unicode+1 等方式加密成非明文，后端解开之后再存它的 MD5/MD6 。
PlanB<br />直接使用 MD5/MD6 之类的方式取 Hash ，让后端存 Hash 的 Hash 。
<a name="4687d461"></a>
场景-数据包加密
应该大家有遇到过：打开一个正经网站，网站底下蹦出个不正经广告——比如X通的流量浮层，X信的插入式广告……（我没有针对谁）<br />但是这几年，我们会发现这种广告逐渐变少了，其原因就是大家都开始采用 HTTPS 了。<br />被人插入这种广告的方法其实很好理解：你的网页数据包被抓取->在数据包到达你手机之前被篡改->你得到了带网页广告的数据包->渲染到你手机屏幕。<br />而 HTTPS 进行了包加密，就解决了这个问题。严格来说我认为从手段上来看，它不算是一种前端加密场景；但是从解决问题的角度来看，这确实是前端需要知道的事情。
Plan<br />全面采用 HTTPS
<a name="H4e5B"></a>
```

99. input 搜索如何防抖，如何处理中文输入

Markdown

```

```

2.DOM相关

1.原生DOM操作和事件相关

1. 如需替换 HTML DOM 中的元素，请使用replaceChild(newnode,oldnode)方法

2. 从父元素中删除子元素 parent.removeChild(child);

3. insertBefore(newItem,existingItem) 在指定的已有子节点之前插入新的子节点

4. appendChild(newListItem向元素添加新的子节点，作为最后一个子节点

5. document.documentElement - 全部文档 document.body - 文档的主体

http://www.w3school.com.cn/jsref/dom_obj_all.asp

JS事件：target与currentTarget区别

target在事件流的目标阶段；currentTarget在事件流的捕获，目标及冒泡阶段。只有当事件流处在目标阶段的时候，两个的指向才是一样的， 而当处于捕获和冒泡阶段的时候，target指向被单击的对象而currentTarget指向当前事件活动的对象（一般为父级）。

事件模型

事件捕捉阶段：事件开始由顶层对象触发，然后逐级向下传播，直到目标的元素； 处于目标阶段：处在绑定事件的元素上； 事件冒泡阶段：事件由具体的元素先接收，然后逐级向上传播，直到不具体的元素；

阻止 冒泡／捕获 event.stopPropagation()和IE的event.cancelBubble=true

DOM事件绑定 1.绑定事件监听函数：addEventListener和attchEvent 2.在JavaScript代码中绑定：获取DOM元素 dom.onlick = fn 3.在DOM元素中直接绑定：

DOM事件流包括三个阶段：事件捕获阶段、处于目标阶段、事件冒泡阶段。首先发生的事件捕获，为截获事件提供机会。然后是实际的目标接受事件。最后一个阶段是时间冒泡阶段，可以在这个阶段对事件做出响应。

事件委托

因为事件具有冒泡机制，因此我们可以利用冒泡的原理，把事件加到父级上，触发执行效果。这样做的好处当然就是提高性能了

最重要的是通过event.target.nodeName判断子元素

```
<div>
  <ul id = "bubble">
```

```
      <li>1</li>
      <li>2</li>
      <li>3</li>
      <li>4</li>
    </ul>
</div>

  window.onload = function () {
      var aUl = document.getElementsById("bubble");
      var aLi = aUl.getElementsByTagName("li");

      //不管在哪个事件中，只要你操作的那个元素就是事件源。
      // ie：window.event.srcElement
      // 标准下:event.target
      aUl.onmouseover = function (ev) {
          var ev = ev || window.event;
          var target = ev.target || ev.srcElement;

          if(target.nodeName.toLowerCase() == "li"){
              target.style.background = "blue";
          }
      };
  };
```

2.documen.write和 innerHTML的区别

document.write只能重绘整个页面

innerHTML可以重绘页面的一部分

3.DOM操作——怎样添加、移除、移动、复制、创建和查找节点?

（1）创建新节点

createDocumentFragment()　//创建一个DOM片段

createElement()　//创建一个具体的元素

createTextNode()　//创建一个文本节点

（2）添加、移除、替换、插入

appendChild()

removeChild()

replaceChild()

insertBefore() //在已有的子节点前插入一个新的子节点

（3）查找

getElementsByTagName()　//通过标签名称

getElementsByName()　//通过元素的Name属性的值(IE容错能力较强，会得到一个数组，其中包括id等于name值的)

getElementById()　//通过元素Id，唯一性

3.BOM相关

4. JS高级

- **1. 函数高级**

1. 闭包

特性： 1.函数嵌套函数 2.函数内部可以引用外部的参数和变量 3.参数和变量不会被垃圾回收机制回收

闭包的缺点就是常驻内存，会增大内存使用量，使用不当很容易造成内存泄露。

为什么要使用闭包：

为了设计私有方法和变量，避免全局变量污染 希望一个变量长期驻扎在内存中

view detail: https://segmentfault.com/a/1190000000652891

2. 原型与原型链

当从一个对象那里调取属性或方法时，如果该对象自身不存在这样的属性或方法，就会去自己关联的prototype对象那里寻找，如果prototype没有，就会去prototype关联的前辈prototype那里寻找，如果再没有则继续查找Prototype.Prototype引用的对象，依次类推，直到Prototype.....Prototype为undefined（Object的Prototype就是undefined）从而形成了所谓的“原型链”。

其中foo是Function对象的实例。而Function的原型对象同时又是Object的实例。这样就构成了一条原型链。

instanceof 确定原型和实例之间的关系

用来判断某个构造函数的prototype属性是否存在另外一个要检测对象的原型链上

对象的__proto__指向自己构造函数的prototype。obj.__proto__.__proto__...的原型链由此产生，包括我们的操作符instanceof正是通过探测obj.__proto__.__proto__... === Constructor.prototype来验证obj是否是Constructor的实例。

function C(){}

var o = new C(){}

//true 因为Object.getPrototypeOf(o) === C.prototype

o instanceof C

instanceof只能用来判断对象和函数，不能用来判断字符串和数字

isPrototypeOf

用于测试一个对象是否存在于另一个对象的原型链上。

判断父级对象 可检查整个原型链

3. 作用域与作用域链

作用域链的作用是保证执行环境里有权访问的变量和函数是有序的，作用域链的变量只能向上访问，变量访问到window对象即被终止，作用域链向下访问变量是不被允许的。

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，这种组织形式就是作用域链。

4.apply, call和bind有什么区别?

参考答案：三者都可以把一个函数应用到其他对象上，call、apply是修改函数的作用域（修改this指向），并且立即执行，而bind是返回了一个新的函数，不是立即执行．apply和call的区别是apply接受数组作为参数，而call是接受逗号分隔的无限多个参数列表，

```
Array.prototype.slice.call(null, args)
```

```
function getMax(arr){
  return Math.max.apply(null, arr);
}
//call
function foo() {
  console.log(this);//{id: 42}
}

foo.call({ id: 42 });
```

如果该方法是非严格模式代码中的函数，则null和undefined将替换为全局对象，并且原始值将被包装。 当你调用apply传递给它null时，就像是调用函数而不提供任何对象

5.谈谈对this的理解

this总是指向函数的直接调用者（而非间接调用者）；

如果有new关键字，this指向new出来的那个对象；

在事件中，this指向触发这个事件的对象，特殊的是，IE中的attachEvent中的this总是指向全局对象Window；

6.那些操作会造成内存泄漏?

内存泄漏指任何对象在您不再拥有或需要它之后仍然存在。

垃圾回收器定期扫描对象，并计算引用了每个对象的其他对象的数量。如果一个对象的引用数量为 0（没有其他对象引用过该对象），或对该对象的惟一引用是循环的，那么该对象的内存即可回收。

setTimeout 的第一个参数使用字符串而非函数的话，会引发内存泄漏。

闭包、控制台日志、循环（在两个对象彼此引用且彼此保留时，就会产生一个循环）

7.深入贯彻闭包思想，全面理解JS闭包形成过程

https://segmentfault.com/a/1190000009886713

8.下面这个ul，如何点击每一列的时候alert其index?（闭包）

```
<ul id=”test”>
<li>这是第一条</li>
<li>这是第二条</li>
<li>这是第三条</li>
</ul>
// 方法一:
var lis=document.getElementById('2223').getElementsByTagName('li');
for(var i=0;i<3;i++)
{
  lis[i].index=i;
  lis[i].onclick=function(){
    alert(this.index);
  };
}
//方法二:
var lis=document.getElementById('2223').getElementsByTagName('li');
for(var i=0;i<3;i++){
  lis[i].index=i;
  lis[i].onclick=(function(a){
```

```
    return function() {
      alert(a);
    }
  })(i);
}
```

4. 对象高级

1.js继承方式及其优缺点

原型链继承的缺点

一 是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。所以我们需要原型链+借用构造函数的模式，这种模式称为组合继承

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。

2.上下文环境对象

3.new操作符具体做了什么

1、创建一个空对象，并且this变量引用该对象，同时继承了该函数的原型（实例对象通过__proto__属性指向原型对象；obj.__proto__ = Base.prototype;） 2、属性和方法被加入到 this 引用的对象中。

```
function Animal(name) {
  this.name = name;
}

Animal.prototype.run = function() {
  console.log(this.name + 'can run...');
}
```

var cat = new Animal('cat');

//模拟过程

new Animal('cat')=function(){

let obj={}; //创建一个空对象

obj.__proto__=Animal.prototype;

//把该对象的原型指向构造函数的原型对象，就建立起原型了：obj->Animal.prototype->Object.prototype->null

return Animal.call(obj,'cat');// 绑定this到实例化的对象上

}

4.创建对象的几种方式

javascript创建对象简单的说,无非就是使用内置对象或各种自定义对象，当然还可以用JSON；但写法有很多种，也能混合使用。

1、对象字面量的方式

person={firstname:"Mark",lastname:"Yun",age:25,eyecolor:"black"};

2、用function来模拟无参的构造函数

function Person(){}

var person=new Person();//定义一个function，如果使用new"实例化",该function可以看作是一个Class

person.name="Mark";

person.age="25";

person.work=function(){

alert(person.name+" hello...");

}

```
person.work();
```

3、用function来模拟参构造函数来实现（用this关键字定义构造的上下文属性）

```
function Pet(name,age,hobby){
  this.name=name;//this作用域：当前对象
  this.age=age;
  this.hobby=hobby;
  this.eat=function(){
    alert("我叫"+this.name+",我喜欢"+this.hobby+",是个程序员");
  }
}
var maidou =new Pet("麦兜",25,"coding");//实例化、创建对象
maidou.eat();//调用eat方法
```

4、用工厂方式来创建（内置对象）

```
var wcDog =new Object();
wcDog.name="旺财";
wcDog.age=3;
wcDog.work=function(){
  alert("我是"+wcDog.name+",汪汪汪.....");
}
wcDog.work();
```

5、用原型方式来创建

```
function Dog(){

}
Dog.prototype.name="旺财";
Dog.prototype.eat=function(){
alert(this.name+"是个吃货");
}
var wangcai =new Dog();
wangcai.eat();
```

5、用混合方式来创建

```
function Car(name,price){
 this.name=name;
 this.price=price;
}
 Car.prototype.sell=function(){
  alert("我是"+this.name+"，我现在卖"+this.price+"万元");
  }
var camry =new Car("凯美瑞",27);
camry.sell();
```

5. 线程机制

1. 同步和异步的区别?

同步：阻塞的

-张三叫李四去吃饭，李四一直忙得不停，张三一直等着，直到李四忙完两个人一块去吃饭

=浏览器向服务器请求数据，服务器比较忙，浏览器一直等着（页面白屏），直到服务器返回数据，浏览器才能显示页面

异步：非阻塞的

-张三叫李四去吃饭，李四在忙，张三说了一声然后自己就去吃饭了，李四忙完后自己去吃

=浏览器向服务器请求数据，服务器比较忙，浏览器可以自如的干原来的事情（显示页面），服务器返回数据的时候通知浏览器一声，浏览器把返回的数据再渲染到页面，局部更新

2. 说说你对JavaScript中事件循环的理解

Plain Text

```
JavaScript在设计之初便是单线程，即指程序运行时，只有一个线程存在，同一时间只能做一件事
为什么要这么设计，跟JavaScript的应用场景有关
JavaScript初期作为一门浏览器脚本语言，通常用于操作DOM，如果是多线程，一个线程进行了删除DOM，另一个添加DOM，此时浏览器该如何处理?
为了解决单线程运行阻塞问题，JavaScript用到了计算机系统的一种运行机制，这种机制就叫做事件循环（Event Loop）
事件循环（Event Loop）
在JavaScript中，所有的任务都可以分为
同步任务：立即执行的任务，同步任务一般会直接进入到主线程中执行
异步任务：异步执行的任务，比如ajax网络请求，setTimeout定时函数等.
同步任务进入主线程，即主执行栈，异步任务进入任务队列，主线程内的任务执行完毕为空，会去任务队列读取对应的任务，推入主线程执行。上述过程的不断重复就是事件循环.
```

3. 什么是宏任务微任务

Plain Text

```
微任务
一个需要异步执行的函数，执行时机是在主函数执行结束之后、当前宏任务结束之前
常见的微任务有:
Promise.then
MutaionObserver
Object.observe（已废弃；Proxy 对象替代）
process.nextTick（Node.js）
宏任务
宏任务的时间粒度比较大，执行的时间间隔是不能精确控制的，对一些高实时性的需求就不太符合
常见的宏任务有:
script （可以理解为外层同步代码）
setTimeout/setInterval
UI rendering/UI事件
postMessage、MessageChannel
setImmediate、I/O（Node.js）
```

4. JavaScript执行上下文是什么?

JavaScript

简单的来说，执行上下文是对Javascript代码执行环境的一种抽象概念，只要有Javascript代码运行，那么它就一定是运行在执行上下文中
执行上下文的类型分为三种：
全局执行上下文：只有一个，浏览器中的全局对象就是window对象，this指向这个全局对象
函数执行上下文：存在无数个，只有在函数被调用的时候才会被创建，每次调用函数都会创建一个新的执行上下文
Eval 函数执行上下文：指的是运行在eval函数中的代码，很少用而且不建议使用

5. JS中执行上下文的生命周期是什么

Plain Text

执行上下文的生命周期包括三个阶段：创建阶段 → 执行阶段 → 回收阶
创建阶段
创建阶段即当函数被调用，但未执行任何其内部代码之前
创建阶段做了三件事：
确定 this 的值，也被称为This Binding
LexicalEnvironment（词法环境） 组件被创建
VariableEnvironment（变量环境） 组件被创建
执行阶段
在这阶段，执行变量赋值、代码执行
如果Javascript引擎在源代码中声明的实际位置找不到变量的值，那么将为其分配undefined值
回收阶段
执行上下文出栈等待虚拟机回收执行上下文

6. JS中执行栈是什么

Plain Text

执行栈，也叫调用栈，具有 LIFO（后进先出）结构，用于存储在代码执行期间创建的所有执行上下文.
当Javascript引擎开始执行你第一行脚本代码的时候，它就会创建一个全局执行上下文然后将它压到执行栈中
每当引擎碰到一个函数的时候，它就会创建一个函数执行上下文，然后将这个执行上下文压到执行栈中
引擎会执行位于执行栈栈顶的执行上下文(一般是函数执行上下文)，当该函数执行结束后，对应的执行上下文就会被弹出，然后控制流程到达执行栈的下一个执行上下文

7. JavaScript中的事件流如何理解?

Plain Text

```
javascript中的事件，可以理解就是在HTML文档或者浏览器中发生的一种交互操作，使得网页具备互动性， 常见的有加载事件、鼠标事件、自定义事件等
由于DOM是一个树结构，如果在父子节点绑定事件时候，当触发子节点的时候，就存在一个顺序问题，这就涉及到了事件流的概念
事件流都会经历三个阶段：
    事件捕获阶段(capture phase)
    处于目标阶段(target phase)
    事件冒泡阶段(bubbling phase)
    事件冒泡是一种从下往上的传播方式，由最具体的元素（触发节点）然后逐渐向上传播到最不具体的那个节点，也就是DOM中最高层的父节点
    事件捕获与事件冒泡相反，事件最开始由不太具体的节点最早接收事件，而最具体的节点（触发节点）最后接收事件
```

8. 对于Js中的事件模型如何理解

Plain Text

```
事件模型可以分为三种：
原始事件模型（DOM0级）
标准事件模型（DOM2级）
IE事件模型（基本不用）
原始事件模型
    HTML代码中直接绑定
    通过JS代码绑定
    特性：
    绑定速度快
DOM0级事件具有很好的跨浏览器优势，会以最快的速度绑定，但由于绑定速度太快，可能页面还未完全加载出来，以至于事件可能无法正常运行
只支持冒泡，不支持捕获
同一个类型的事件只能绑定一次
标准事件模型
    在该事件模型中，一次事件共有三个过程：
    事件捕获阶段：事件从document一直向下传播到目标元素，依次检查经过的节点是否绑定了事件监听函数，如果有则执行
    事件处理阶段：事件到达目标元素，触发目标元素的监听函数
    事件冒泡阶段：事件从目标元素冒泡到document，依次检查经过的节点是否绑定了事件监听函数，如果有则执行
    事件绑定监听函数以及移除的方式如下：
        addEventListener(eventType, handler, useCapture)
        removeEventListener(eventType, handler, useCapture)
        特性：
        可以在一个DOM元素上绑定多个事件处理器，各自并不会冲突
        当第三个参数(useCapture)设置为true就在捕获过程中执行，反之在冒泡过程中执行处理函数
IE事件模型
    IE事件模型共有两个过程：
        事件处理阶段：事件到达目标元素，触发目标元素的监听函数。
        事件冒泡阶段：事件从目标元素冒泡到document，依次检查经过的节点是否绑定了事件监听函数，如果有则执行
        事件绑定监听函数的方式如下：
            attachEvent(eventType, handler)
            detachEvent(eventType, handler)
```

5.ES6相关

1.谈一谈let与var和const的区别?

let为ES6新添加申明变量的命令，它类似于var，但是有以下不同：

let命令不存在变量提升，如果在let前使用，会导致报错

暂时性死区的本质，其实还是块级作用域必须“先声明后使用”的性质。

let，const和class声明的全局变量不是全局对象的属性。

const声明的变量与let声明的变量类似，它们的不同之处在于，const声明的变量只可以在声明时赋值，不可随意修改，否则会导致SyntaxError（语法错误）。

const只是保证变量名指向的地址不变，并不保证该地址的数据不变。const可以在多个模块间共享 let暂时性死区的原因：var 会变量提升，let 不会。

2.箭头函数

箭头函数不属于普通的 function，所以没有独立的上下文。箭头函数体内的this对象，就是定义时所在的对象，而不是使用时所在的对象。 由于箭头函数没有自己的this，函数对象中的call、apply、bind三个方法，无法"覆盖"箭头函数中的this值。 箭头函数没有原本(传统)的函数有的隐藏arguments对象。 箭头函数不能当作generators使用，使用yield会产生错误。

在以下场景中不要使用箭头函数去定义：

1. 定义对象方法、定义原型方法、定义构造函数、定义事件回调函数。
2. 箭头函数里不但没有 this，也没有 arguments, super ……

3. Map和Set

Symbol，Map和Set

Map 对象保存键值对。一个对象的键只能是字符串或者 Symbols，但一个 Map 的键可以是任意值。Set 对象允许你存储任何类型的唯一值，Set对象是值的集合，Set中的元素只会出现一次 Symbol 是

一种特殊的、不可变的数据类型，可以作为对象属性的标识符使用(Symbol([description]))

```
let mySet = new Set()
mySet.add(1)
mySet.add('hello')
mySet.add('hello')
console.log(mySet.size);//2
console.log(mySet);//Set {1,'hello'}

//Map保存键值对也不能有重复的
let myMap = new Map();
let key1 = 'China',key2 = 'America';
myMap.set(key1,'welcome')
myMap.set(key2,'gold bless you')
console.log(myMap);//Map { 'China' => 'welcome', 'America' => 'gold bless you' }
console.log(myMap.get(key1));//welcome
console.log(myMap.get(key2));//gold bless you

let mySymbol = Symbol('symbol1');
let mySymbol2 = Symbol('symbol1');
console.log(mySymbol == mySymbol2);//false
//Symbols 在 for...in 迭代中不可枚举。
let obj = {}
obj['c'] = 'c'
obj.d ='d'
obj[Symbol('a')] = 'a'
obj[Symbol.for('b')] = 'b'
for(let k in obj){
   console.log(k);//logs 'c' and 'd'
```

}

for...of可以用来遍历数组，类数组对象，argument，字符串，Map和Set，for...in用来遍历对象

4.Promise实现原理

现在回顾下Promise的实现过程，其主要使用了设计模式中的观察者模式：

通过Promise.prototype.then和Promise.prototype.catch方法将观察者方法注册到被观察者Promise对象中，同时返回一个新的Promise对象，以便可以链式调用。

被观察者管理内部pending、fulfilled和rejected的状态转变，同时通过构造函数中传递的resolve和reject方法以主动触发状态转变和通知观察者。

Promise.then()是异步调用的，这也是Promise设计上规定的，其原因在于同步调用和异步调用同时存在会导致混乱。

为了暂停当前的 promise，或者要它等待另一个 promise 完成，只需要简单地在 then() 函数中返回另一个 promise。

Promise 也有一些缺点。首先，无法取消 Promise，一旦新建它就会立即执行，无法中途取消。其次，如果不设置回调函数，Promise 内部抛出的错误，不会反应到外部。第三，当处于 Pending 状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

一般来说，不要在then方法里面定义Reject状态的回调函数（即then的第二个参数），总是使用catch方法，理由是更接近同步的写法。 then的第二个函数参数和catch等价

Promise.all和Promise.race的区别?

Promise.all 把多个promise实例当成一个promise实例,当这些实例的状态都发生改变时才会返回一个新的promise实例，才会执行then方法。 Promise.race 只要该数组中的 Promise 对象的状态发生变化（无论是resolve还是reject）该方法都会返回。

5.Object.is() 与原来的比较操作符“===”、“==”的区别?

两等号判等，会在比较时进行类型转换；

三等号判等(判断严格)，比较时不进行隐式类型转换，（类型不同则会返回false）；

Object.is 在三等号判等的基础上特别处理了 NaN 、-0 和 +0 ，保证 -0 和 +0 不再相同，

但 Object.is(NaN, NaN) 会返回 true.

Object.is 应被认为有其特殊的用途，而不能用它认为它比其它的相等对比更宽松或严格。

6.前端面试之ES6篇

https://juejin.im/post/59c8aec0f265da065c5e965e

6.跨域

1.JSONP

回调函数+数据就是 JSON With Padding，简单、易部署。（做法：动态插入script标签，设置其src属性指向提供JSONP服务的URL地址，查询字符串中加入 callback 指定回调函数，返回的 JSON 被包裹在回调函数中以字符串的形式被返回，需将script标签插入body底部）。缺点是只支持GET，不支持POST（原因是通过地址栏传参所以只能使用GET）

2.document.domain 跨子域

document.domain 跨子域 （ 例如a.qq.com嵌套一个b.qq.com的iframe ，如果a.qq.com设置document.domain为qq.com 。b.qq.com设置document.domain为qq.com， 那么他俩就能互相通信了，不受跨域限制了。 注意：只能跨子域）

3. iframe

window.name + iframe ==> http://www.tuicool.com/articles/viMFbqV，支持跨主域。不支持POST

4.postMessage()

HTML5的postMessage()方法允许来自不同源的脚本采用异步方式进行有限的通信，可以实现跨文本档、多窗口、跨域消息传递。适用于不同窗口iframe之间的跨域

5.CORS

CORS（Cross Origin Resource Share）对方服务端设置响应头

设置相应头：”Access-Control-Allow-Origin“

CORS请求默认不发送Cookie和HTTP认证信息。如果要把Cookie发到服务器，一方面要服务器同意，指定Access-Control-Allow-Credentials字段。

6.服务端代理

服务端代理 在浏览器客户端不能跨域访问，而服务器端调用HTTP接口只是使用HTTP协议，不会执行JS脚本，不需要同源策略，也就没有跨越问题。简单地说，就是浏览器不能跨域，后台服务器可以跨域。（一种是通过http-proxy-middleware插件设置后端代理；另一种是通过使用http模块发出请求）

7.Ajax

1.ajax请求和原理

```
var xhr = new XMLHTTPRequest();
// 请求 method 和 URI
xhr.open('GET', url);
// 请求内容
xhr.send();
// 响应状态
xhr.status
// xhr 对象的事件响应
xhr.onreadystatechange = function() {}
xhr.readyState
// 响应内容
xhr.responseText
```

AJAX的工作原理

Ajax的工作原理相当于在用户和服务器之间加了—个中间层(AJAX引擎),使用户操作与服务器响应异步化。　Ajax的原理简单来说通过XmlHttpRequest对象来向服务器发异步请求，从服务器获得数据，然后用javascript来操作DOM而更新页面。

ajax优缺点

优点：无刷新更新数据 异步与服务器通信 前后端负载均衡

缺点：

1）ajax干掉了Back和history功能，对浏览器机制的破坏 2）对搜索引擎支持较弱 3）违背了URI和资源定位的初衷

2.fetch和Ajax有什么不同

XMLHttpRequest 是一个设计粗糙的 API，不符合关注分离（Separation of Concerns）的原则，配置和调用方式非常混乱，而且基于事件的异步模型写起来也没有现代的 Promise，generator/yield，async/await 友好。

fetch 是浏览器提供的一个新的 web API，它用来代替 Ajax（XMLHttpRequest），其提供了更优雅的接口，更灵活强大的功能。 Fetch 优点主要有：

- 语法简洁，更加语义化
- 基于标准 Promise 实现，支持 async/await

```
fetch(url).then(response => response.json())
  .then(data => console.log(data))
  .catch(e => console.log("Oops, error", e))
```

3.GET和POST的区别

GET和POST的区别

GET使用URL或Cookie传参，而POST将数据放在BODY中，这个是因为HTTP协议用法的约定。并非它们的本身区别。

GET方式提交的数据有长度限制，则POST的数据则可以非常大，这个是因为它们使用的操作系统和浏览器设置的不同引起的区别。也不是GET和POST本身的区别。

POST比GET安全，因为数据在地址栏上不可见，这个说法没毛病，但依然不是GET和POST本身的区别。

GET和POST最大的区别主要是GET请求是幂等性的，POST请求不是。（幂等性：对同一URL的多个请求应该返回同样的结果。）因为get请求是幂等的，在网络不好的隧道中会尝试重试。如果用get请求增数据，会有重复操作的风险，而这种重复操作可能会导致副作用

4.GET,POST,PUT,Delete

1. GET请求会向数据库获取信息，只是用来查询数据，不会修改，增加数据。使用URL传递参数，对所发送的数量有限制，一般在2000字符

2. POST向服务器发送数据，会改变数据的种类等资源，就像insert操作一样，会创建新的内容，大小一般没有限制，POST安全性高，POST不会被缓存

3. PUT请求就像数据库的update操作一样，用来修改数据内容，不会增加数据种类

4. Delete用来删除操作

5.缓存，存储相关（cookie，web storage和session）

cookie优点： 1.可以解决HTTP无状态的问题，与服务器进行交互 缺点： 1.数量和长度限制，每个域名最多20条，每个cookie长度不能超过4kb 2.安全性问题。容易被人拦截 3.浪费带宽，每次请求新页面，cookie都会被发送过去

cookie和session区别

1.cookie数据存放在客户的浏览器上，session数据放在服务器上。 2.session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能。考虑到减轻服务器性能方面，应当使用COOKIE。

sessionStorage是当前对话的缓存，浏览器窗口关闭即消失，localStorage持久存在，除非清除浏览器缓存。

页面缓存原理

页面缓存状态是由http header决定的，一个浏览器请求信息，一个是服务器响应信息。主要包括Pragma: no-cache、Cache-Control、 Expires、 Last-Modified、If-Modified-Since。

6.什么是Cookie 隔离？（或者说：请求资源的时候不要让它带cookie怎么做）

如果静态文件都放在主域名下，那静态文件请求的时候都带有的cookie的数据提交给server的，非常浪费流量，

所以不如隔离开。

因为cookie有域的限制，因此不能跨域提交请求，故使用非主要域名的时候，请求头中就不会带有cookie数据，

这样可以降低请求头的大小，降低请求时间，从而达到降低整体请求延时的目的。

同时这种方式不会将cookie传入Web Server，也减少了Web Server对cookie的处理分析环节，

提高了webserver的http请求的解析速度。

7. Ajax 解决浏览器缓存问题?

1、在ajax发送请求前加上 anyAjaxObj.setRequestHeader("If-Modified-Since","0")。

2、在ajax发送请求前加上 anyAjaxObj.setRequestHeader("Cache-Control","no-cache")。

3、在URL后面加上一个随机数： "fresh=" + Math.random();。

4、在URL后面加上时间戳："nowtime=" + new Date().getTime();。

5、如果是使用jQuery，直接这样就可以了 $.ajaxSetup({cache:false})。这样页面的所有ajax都会执行这条语句就是不需要保存缓存记录。

8.Node

1.node核心

核心模块：EventEmitter, Stream, FS, Net和全局对象

全局对象：process, console, Buffer和exports

exports和module.exports区别

exports 是 module.exports 的一个引用 module.exports 初始值为一个空对象 {}，所以 exports 初始值也是 {} require 引用模块后，返回的是 module.exports 而不是 exports

2.单线程优点

Node.js依托于v8引擎，都是以单线程为基础的。单线程资源占用小。单线程避免了传统PHP那样频繁创建、切换线程的开销，使执行速度更加迅速

3.Node.js是如何做到I/O的异步和非阻塞的呢

其实Node在底层访问I/O还是多线程的。Node可以借助livuv来来实现多线程。

如果我们非要让Node.js支持多线程，还是提倡使用官方的做法，利用libuv库来实现。

cluster可以用来让Node.js充分利用多核cpu的性能

4.并行与并发，进程与线程

并发 (Concurrent) = 2 队列对应 1 咖啡机.

并行 (Parallel) = 2 队列对应 2 咖啡机.

线程是进程下的执行者，一个进程至少会开启一个线程（主线程），也可以开启多个线程。

5.谈谈Nodejs优缺点

优点：

1. 事件驱动，异步编程，占用内存少

2. npm设计得好

缺点：

1. Debug 很困难。没有 stack trace，出了问题很难查找问题的原因；

2 .如果设计不好，很容易让代码充满 callback，代码不优雅；

3. 可靠性低；

4. 单进程，单线程，只支持单核CPU，不能充分的利用多核CPU服务器。

9.模块化

1.模块化开发怎么做?

立即执行函数,不暴露私有成员

```
var module1 = (function(){
```

```
        var _count = 0;
        var m1 = function(){
            //...
        };
        var m2 = function(){
            //...
        };
        return {
            m1 : m1,
            m2 : m2
        };
    })();
```

（待完善）

2.AMD（Modules/Asynchronous-Definition）、CMD（Common Module Definition）规范区别?

AMD 规范在这里：https://github.com/amdjs/amdjs-api/wiki/AMD

CMD 规范在这里：https://github.com/seajs/seajs/issues/242

Asynchronous Module Definition，异步模块定义，所有的模块将被异步加载，模块加载不影响后面语句运行。所有依赖某些模块的语句均放置在回调函数中。

区别：

1. 对于依赖的模块，AMD 是提前执行，CMD 是延迟执行。不过 RequireJS 从 2.0 开始，也改成可以延迟执行（根据写法不同，处理方式不同）。CMD 推崇 as lazy as possible.

2. CMD 推崇依赖就近，AMD 推崇依赖前置。看代码：

```
// CMD
define(function(require, exports, module) {
  var a = require('./a')
  a.doSomething()
```

```
    // 此处略去 100 行
    var b = require('./b') // 依赖可以就近书写
    b.doSomething()
    // ...
})

// AMD 默认推荐
define(['./a', './b'], function(a, b) { // 依赖必须一开始就写好
    a.doSomething()
    // 此处略去 100 行
    b.doSomething()
    // ...
})
```

3.requireJS的核心原理是什么？（如何动态加载的？如何避免多次加载的？如何 缓存的？）

参考：http://annn.me/how-to-realize-cmd-loader/

四 异步

1.setTimeout、Promise、Async/Await 的区别

1. setTimeout

console.log('script start')//1. 打印 script startsetTimeout(function(){console.log('settimeout')// 4. 打印 settimeout})// 2. 调用 setTimeout 函数，并定义其完成后执行的回调函数console.log('script end')//3. 打印 script start// 输出顺序：script start->script end->settimeout

2. Promise

Promise本身是**同步的立即执行函数**， 当在executor中执行resolve或者reject的时候, 此时是异步操作， 会先执行then/catch等，当主栈完成后，才会去调用resolve/reject中存放的方法执行，打印p的时候，是打印的返回结果，一个Promise实例。

console.log('script start')letpromise1=newPromise(function(resolve) {console.log('promise1')resolve()console.log('promise1 end')}).then(function() {console.log('promise2')})setTimeout(function(){console.log('settimeout')})console.log('script end')// 输出顺序: script start->promise1->promise1 end->script end->promise2->settimeout

当JS主线程执行到Promise对象时，

- promise1.then() 的回调就是一个 task
- promise1 是 resolved或rejected: 那这个 task 就会放入当前事件循环回合的 microtask queue
- promise1 是 pending: 这个 task 就会放入 事件循环的未来的某个(可能下一个)回合的 microtask queue 中
- setTimeout 的回调也是个 task ，它会被放入 macrotask queue 即使是 0ms 的情况

3. async/await

asyncfunctionasync1(){console.log('async1 start');awaitasync2();console.log('async1 end')}asyncfunctionasync2(){console.log('async2')}console.log('script start');async1();console.log('script end')// 输出顺序：script start->async1 start->async2->script end->async1 end

async 函数返回一个 Promise 对象，当函数执行的时候，一旦遇到 await 就会先返回，等到触发的异步操作完成，再执行函数体内后面的语句。可以理解为，是让出了线程，跳出了 async 函数体。

举个例子：

asyncfunctionfunc1(){return1}console.log(func1())

```
▼ Promise {<resolved>: 1}
  ▶ __proto__: Promise
    [[PromiseStatus]]: "resolved"
    [[PromiseValue]]: 1
```

很显然，func1的运行结果其实就是一个Promise对象。因此我们也可以使用then来处理后续逻辑。

JavaScript

```
func1().then(res => {
    console.log(res);  // 30
})
```

await的含义为等待，也就是 async 函数需要等待await后的函数执行完成并且有了返回结果（Promise对象）之后，才能继续执行下面的代码。await通过返回一个Promise对象来实现同步的效果。

2.Async/Await 如何通过同步的方式实现异步

作为前端人员要回答这个问题，需要了解这三个知识点：

- 同步
- 异步
- Async/Await

首先，**js 是单线程的（重复三遍）**，所谓单线程，

通俗的讲就是，一根筋（比喻有点过分，哈哈）执行代码是一行一行的往下走（即所谓的**同步**），

如果上面的没执行完，就痴痴的等着（是不是很像恋爱中在路边等她/他的你，假装 new 了个对象，啊哈哈哈，调皮一下很开心），

还是举个吧：

// chrome 75functiontest(){letd=Date.now();for(leti=0;i<1e8;i++){}console.log(Date.now()-d);// 62ms左右}functiontest1(){letd=Date.now();console.log(Date.now()-d);// 0}test();test1();

上面仅仅是一个 for 循环，而在实际应用中，会有大量的网络请求，它的响应时间是不确定的，这种情况下也要痴痴的等么？显然是不行的，因而 js 设计了异步，即 发起网络请求（诸如 IO 操作，定时器），由于需要等服务器响应，就先不理会，而是去做其他的事儿，等请求返回了结果的时候再说（即**异步**）。

那么如何实现异步呢？其实我们平时已经在大量使用了，那就是callback，例如：

$.ajax({url: 'http://xxx',success: function(res){console.log(res);},});

success 作为函数传递过去并不会立即执行，而是等请求成功了才执行，即**回调函数**（callback）

constfs=require('fs');fs.rename('旧文件.txt','新文件.txt',err=>{if(err)throwerr;console.log('重命名完成');});

和网络请求类似，等到 IO 操作有了结果（无论成功与否）才会执行第三个参数：(err)=>{}

从上面我们就可以看出，实现异步的核心就是回调钩子，将 cb 作为参数传递给异步执行函数，当有了结果后在触发 cb。想了解更多，去看看event-loop机制吧。

至于 async/await 是如何出现的呢，在 es6 之前，大多 js 数项目中会有类似这样的代码：

```
ajax1(url,()=>{ajax2(url,()=>{ajax3(url,()=>{// do something});});});
```

这种函数嵌套，大量的回调函数，使代码阅读起来晦涩难懂，不直观，形象的称之为**回调地狱（callback hell）**，所以为了在写法上能更通俗一点，es6+陆续出现了Promise、Generator、Async/await，力求在写法上简洁明了，可读性强。

==========我是分割线==========

以上只是铺垫，下面在进入正题，开始说道说道主角：async/await

==========我是分割线==========

async/await是参照Generator封装的一套异步处理方案，可以理解为Generator的语法糖，

所以了解async/await就不得不讲一讲Generator,

而Generator又依赖于迭代器Iterator，

所以就得先讲一讲Iterator,

而Iterator的思想呢又来源于单向链表，

终于找到源头了：单向链表

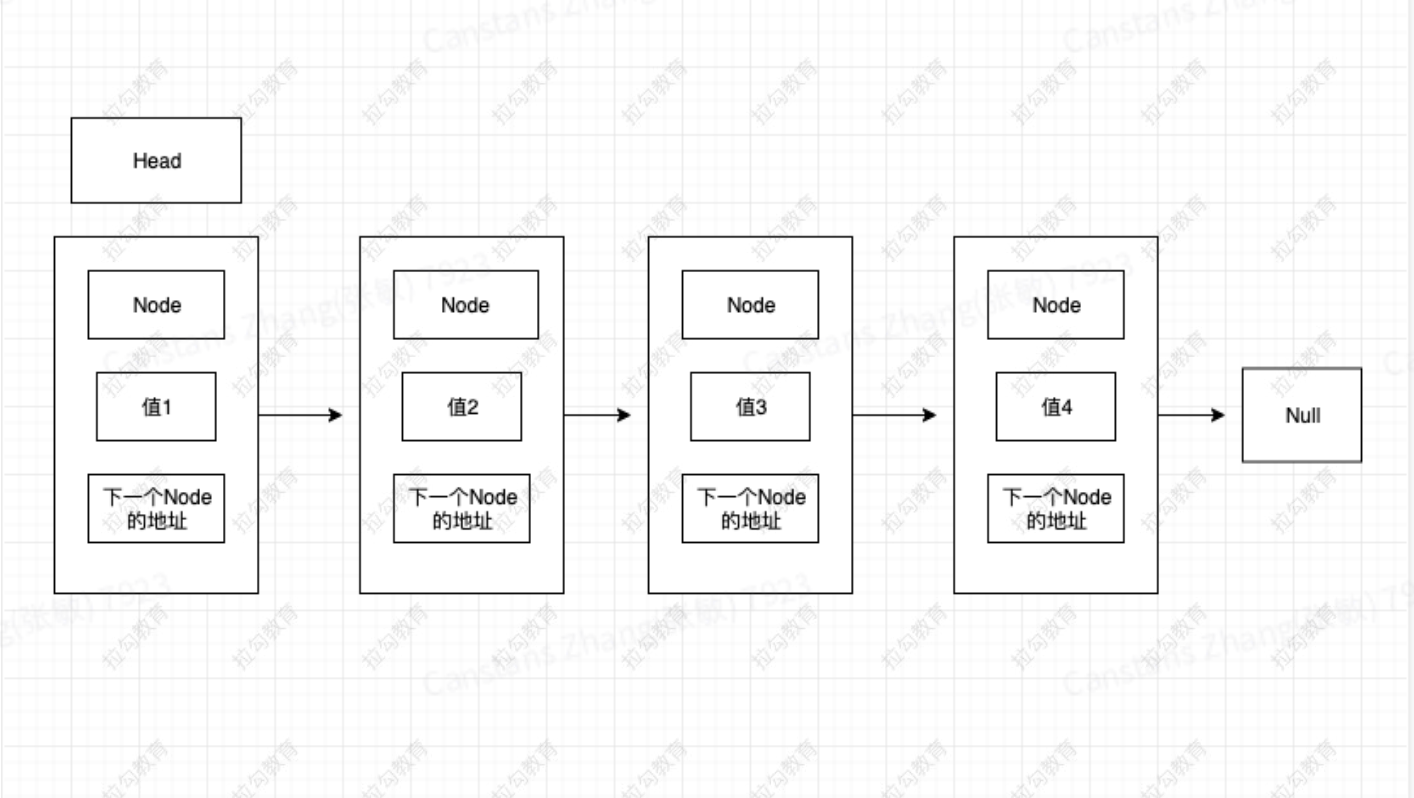
1. 单向链表

> wiki：链表（Linked list）是一种常见的基础数据结构，是一种线性表，但是并不会按线性的顺序储存数据，而是在每一个节点里存到下一个节点的指针（Pointer）。由于不必须按顺序储存，链表在插入的时候可以达到 o(1)的复杂度，比另一种线性表顺序表快得多，但是查找一个节点或者访问特定编号的节点则需要 o(n)的时间，而顺序表响应的时间复杂度分别是 o(logn)和 o(1)。

总结一下链表优点：

- 无需预先分配内存
- 插入/删除节点不影响其他节点，效率高（典型的例子：git commit、~~dom 操作~~）

单向链表：是链表中最简单的一种，它包含两个域，一个信息域和一个指针域。这个链接指向列表中的下一个节点，而最后一个节点则指向一个空值。



一个单向链表包含两个值: 当前节点的值和一个指向下一个节点的链接

单链特点：节点的链接方向是单向的；相对于数组来说，单链表的的随机访问速度较慢，但是单链表删除/添加数据的效率很高。

理解 js 原型链/作用域链的话，理解这个很容易，他们是相通的。~~编程语言中，数组的长度时固定的，所以数组中的增加和删除比较麻烦，需要频繁的移动数组中的其他元素，而 js 作为一门动态语言，数组本质是一个类似数组的对象，是动态的，不需要预先分配内存~~

那么如何设计一个单向链表呢？这个取决于我们需要哪些操作，通常有：

- append(element)：追加节点
- insert(element,index)：在索引位置插入节点
- remove(element)：删除第一个匹配到的节点
- removeAt(index)：删除指定索引节点
- removeAll(element)：删除所有匹配的节点
- get(index)：获取指定索引的节点信息
- set(element,index)：修改指定索引的节点值
- indexOf(element)：获取某节点的索引位置
- clear()：清除所有节点
- length()：返回节点长度
- printf()：打印节点信息

看到这些方法是不是有些许熟悉，当你用原生 js 或 jq 时常会用上面类似的方法，现在根据上面列出的方法进行实现一个单向链：

```
// 节点模型classLinkNode{constructor(element,next)
{this.element=element;this.next=next;}}classLinkedList{constructor()
{this._head=null;this._size=0;this._errorBoundary=this._errorBoundary.bind(this);this._getNodeByIndex=this._getNodeByIndex.bind(this);this.append=this.append.bind(this);this.insert=this.insert.bind(this);this.remove=this.remove.bind(this);this.removeAt=this.removeAt.bind(this);this.removeAll=this.removeAll.bind(this);this.getElement=this.getElement.bind(this);this.setIndex=this.setIndex.bind(this);this.indexOf=this.indexOf.bind(this);this.clear=this.clear.bind(this);this.length=this.length.bind(this);this.printf=this.printf.bind(this);}// 边界检验_errorBoundary(index)
{if(index<0||index >= this._size){throw`超出边界(${0}~${this._size})，目标位置${index}不存在！`;}}// 根据索引获取目标对象_getNodeByIndex(index)
{this._errorBoundary(index);letobj=this._head;for(leti=0;i<index;i++){obj=obj.next;}returnobj;}// 追加节点append(element){if(this._size===0)
{this._head=newLinkNode(element,null);}else{letobj=this._getNodeByIndex(this._size-1);obj.next=newLinkNode(element,null);}this._size++;}// 在索引位置插入节点
insert(element,index){if(index===0)
{this._head=newLinkNode(element,this._head);}else{letobj=this._getNodeByIndex(index-1);obj.next=newLinkNode(element,obj.next);}this._size++;}// 删除第一个匹配到的节点
remove(element){if(this._size<1)returnnull;if(this._head.element==element)
{this._head.element=this._head.next;this._size--;returnelement;}else{lettemp=this._head;while(temp.next){if(temp.next.element==element)
{temp.next=temp.next.next;this._size--;returnelement;}else{temp=temp.next;}}}returnnull;}// 删除指定索引节点removeAt(index){this._errorBoundary(index);letelement=null;if(index===0)
{element=this._head.element;this._head=this._head.next;}else{letprev=this._getNodeByIndex(index-1);element=prev.next.element;prev.next=prev.next.next;}this._size--;returnelement;}// 删除所有匹配的节点removeAll(element){// 创建虚拟头节点，
letv_head=newLinkNode(null,this._head);lettempNode=v_head;// let tempEle = null;while(tempNode.next){if(tempNode.next.element==element)
{tempNode.next=tempNode.next.next;this._size--;// tempEle = element;}else{tempNode=tempNode.next;}}this._head=v_head.next;}// 获取指定索引的节点信息
getElement(index){returnthis._getNodeByIndex(index).element;}// 修改指定索引的节点值
setIndex(element,index)
{this._errorBoundary(index);letobj=this._getNodeByIndex(index);obj.element=element;}// 获取某节点的索引位置indexOf(element){letobj=this._head;letindex=-1;for(leti=0;i<this._size;i++)
{if(obj.element==element){index=i;break;}obj=obj.next;}returnindex;}// 清除所有节点clear()
{this._head=null;this._size=0;}// 返回节点长度length(){returnthis._size;}// 打印节点信息printf()
```

{letobj=this._head;constarr=[];while(obj != null)
{arr.push(obj.element);obj=obj.next;}conststr=arr.join('->');returnstr||null;}}constobj=newLinkedList();obj.append(0);obj.append(1);obj.append(2);obj.printf();// "0->1->2"obj.insert(3,3);obj.printf();// "0->1->2->3"obj.remove(3);obj.printf();// "0->1->2"obj.removeAt(0);obj.printf();// "1->2"obj.setIndex(0,0);obj.printf();// "0->2"obj.indexOf(2);// 1obj.length();// 2obj.clear();obj.printf();// null

查看源码

通过以上，我假装你明白什么是单向链表，并且能够用代码实现一个单向链表了，下一步开始说一说**迭代器**Iterator

2. Iterator

Iterator翻译过来就是**迭代器（遍历器）**让我们先来看看它的遍历过程(类似于单向链表)：

- 创建一个**指针对象**，指向当前数据结构的起始位置
- 第一次调用指针对象的next方法，将指针指向数据结构的第一个成员
- 第二次调用指针对象的next方法，将指针指向数据结构的第二个成员
- 不断的调用指针对象的next方法，直到它指向数据结构的结束位置

一个对象要变成可迭代的，必须实现@@iterator方法，即对象（或它原型链上的某个对象）必须有一个名字是Symbol.iterator的属性（原生具有该属性的有：字符串、数组、类数组的对象、Set 和 Map）：

	A	B
1	**属性**	**值**
2	[Symbol.iterator]:	返回一个对象的无参函数，被返回对象符合迭代器协议

当一个对象需要被迭代的时候（比如开始用于一个for..of循环中），它的@@iterator方法被调用并且无参数，然后返回一个用于在迭代中获得值的迭代器

迭代器协议：产生一个有限或无限序列的值，并且当所有的值都已经被迭代后，就会有一个默认的返回值

当一个对象只有满足下述条件才会被认为是一个迭代器：

它实现了一个next()的方法，该方法**必须返回一个对象**,对象有两个必要的属性：

- done（bool）
  - true：迭代器已经超过了可迭代次数。这种情况下,value 的值可以被省略

- 如果迭代器可以产生序列中的下一个值，则为 false。这等效于没有指定 done 这个属性
- value迭代器返回的任何 JavaScript 值。done 为 true 时可省略

根据上面的规则，咱们来自定义一个简单的迭代器：

constmakeIterator=arr=>{letnextIndex=0;return{next: ()=>nextIndex<arr.length ? {value: arr[nextIndex++],done: false} : {value: undefined,done: true},};};constit=makeIterator(['人月','神话']);console.log(it.next());// { value: "人月", done: false }console.log(it.next());// { value: "神话", done: false }console.log(it.next());// {value: undefined, done: true }

我们还可以自定义一个可迭代对象：

constmyIterable={};myIterable[Symbol.iterator]=function*() {yield1;yield2;yield3;};for(letvalueofmyIterable){console.log(value);}// 1// 2// 3//orconsole.log([...myIterable]);// [1, 2, 3]

了解了迭代器，下面可以进一步了解生成器了

3. Generator

Generator：生成器对象是生成器函数（GeneratorFunction）返回的，它符合**可迭代协议**和**迭代器协议**，既是迭代器也是可迭代对象，可以调用next方法，但它不是函数，更不是构造函数

生成器函数（GeneratorFunction）：

> function* name([param[, param[, ... param]]]) { statements }
>
> - name：函数名
> - param：参数
> - statements：js 语句

调用一个生成器函数并不会马上执行它里面的语句，而是返回一个这个生成器的迭代器对象，当这个迭代器的next()方法被首次（后续）调用时，其内的语句会执行到第一个（后续）出现yield的位置为止（让执行处于**暂停状**），yield后紧跟迭代器要返回的值。或者如果用的是yield*（多了个星号），则表示将执行权移交给另一个生成器函数（当前生成器**暂停执行**），调用next()（再启动）方法时，如果传入了参数，那么这个参数会作为**上一条执行的yield语句的返回值**，例如：

function* another(){yield'人月神话';}function* gen(){yield* another();// 移交执行权consta=yield'hello';constb=yielda;// a='world' 是 next('world') 传参赋值给了上一个 yidle 'hello' 的左值yieldb;// b=！ 是 next('！') 传参赋值给了上一个 yidle a 的左值}constg=gen();g.next();// {value: "人月神话", done: false}g.next();// {value: "hello", done: false}g.next('world');// {value: "world", done: false} 将 'world' 赋给上一条 yield 'hello' 的左值，即执行 a='world'，g.next('!');// {value: "!", done: false} 将 '!' 赋给上一条 yield a 的左值，即执行 b='!'，返回 bg.next();// {value: undefined, done: false}

看到这里，你可能会问，Generator和callback有啥关系，如何处理异步呢？其实二者没有任何关系，我们只是通过一些方式强行的它们产生了关系，才会有Generator处理异步

我们来总结一下Generator的本质，暂停，它会让程序执行到指定位置先暂停（yield），然后再启动（next），再暂停（yield），再启动（next），而这个暂停就很容易让它和异步操作产生联系，因为我们在处理异步时：开始异步处理（网络求情、IO 操作），然后暂停一下，等处理完了，再该干嘛干嘛。不过值得注意的是，**js 是单线程的（又重复了三遍）**，异步还是异步，callback 还是 callback，不会因为Generator而有任何改变

下面来看看，用Generator实现异步：

```
constpromisify=require('util').promisify;constpath=require('path');constfs=require('fs');constreadFile=promisify(fs.readFile);constgen=function*()
{constres1=yieldreadFile(path.resolve(__dirname,'../data/a.json'),{encoding:
'utf8'});console.log(res1);constres2=yieldreadFile(path.resolve(__dirname,'../data/b.json'),
{encoding:
'utf8'});console.log(res2);};constg=gen();constg1=g.next();console.log('g1:',g1);g1.value.then(res1=>{console.log('res1:',res1);constg2=g.next(res1);console.log('g2:',g2);g2.value.then(res2=>
{console.log('res2:',res2);g.next(res2);}).catch(err2=>{console.log(err2);});}).catch(err1=>
{console.log(err1);});// g1: { value: Promise { <pending> }, done: false }// res1: {//   "a": 1// }// {//
"a": 1// }// g2: { value: Promise { <pending> }, done: false }// res2: {//   "b": 2// }// {//   "b": 2// }
```

以上代码是Generator和callback结合实现的异步，可以看到，仍然需要手动执行.then层层添加回调，但由于next()方法返回对象{value: xxx,done: true/false}所以我们可以简化它，写一个自动执行器：

```
constpromisify=require('util').promisify;constpath=require('path');constfs=require('fs');constreadFile=promisify(fs.readFile);functionrun(gen){constg=gen();functionnext(data)
{constres=g.next(data);// 深度递归，只要 `Generator` 函数还没执行到最后一步，`next` 函数就调用自身if(res.done)returnres.value;res.value.then(function(data)
{next(data);});}next();}run(function*()
{constres1=yieldreadFile(path.resolve(__dirname,'../data/a.json'),{encoding:
'utf8'});console.log(res1);// {//   "a": 1//
}constres2=yieldreadFile(path.resolve(__dirname,'../data/b.json'),{encoding:
'utf8'});console.log(res2);// {//   "b": 2// }});
```

说了这么多，怎么还没有到async/await，客官别急，马上来了（其实我已经漏了一些内容没说：Promise 和 callback 的关系，thunk 函数，co 库，感兴趣的可以去 google 一下，ruanyifeng 老师讲的es6 入门非常棒，我时不时的都会去看一看）

4. Async/Await

首先，async/await是Generator的语法糖，上面*我是分割线*下的第一句已经讲过，先来看一下二者的对比：

// Generatorrun(function*(){constres1=yieldreadFile(path.resolve(__dirname,'../data/a.json'), {encoding: 'utf8'});console.log(res1);constres2=yieldreadFile(path.resolve(__dirname,'../data/b.json'), {encoding: 'utf8'});console.log(res2);});// async/awaitconstreadFile=async()=> {constres1=awaitreadFile(path.resolve(__dirname,'../data/a.json'),{encoding: 'utf8'});console.log(res1);constres2=awaitreadFile(path.resolve(__dirname,'../data/b.json'), {encoding: 'utf8'});console.log(res2);return'done'；}constres=readFile();

可以看到，async function代替了function*，await代替了yield，同时也无需自己手写一个自动执行器run了

现在再来看看async/await的特点：

- 当await后面跟的是 Promise 对象时，才会异步执行，其它类型的数据会同步执行
- 执行const res = readFile();返回的仍然是个 Promise 对象，上面代码中的return 'done';会直接被下面then函数接收到

res.then(data=>{console.log(data);// done});

啊，终于完了，一个async-await连带出来这么多知识点，以后面试被问到它的原理时，希望能够帮助到你

【参考】：

1. https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Iteration_protocols#%E5%8F%AF%E8%BF%AD%E4%BB%A3%E5%8D%8F%E8%AE%AE
2. http://es6.ruanyifeng.com/#docs/iterator
3. http://es6.ruanyifeng.com/#docs/async

3. 异步笔试题，请写出下面代码的运行结果

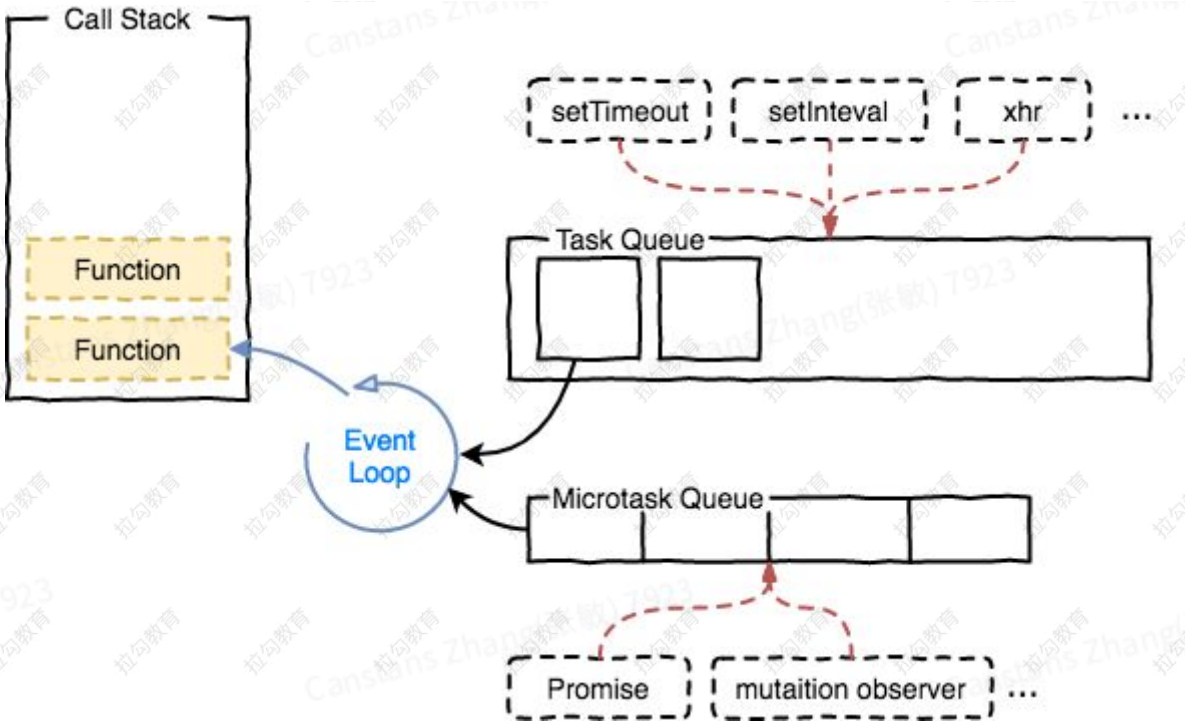
JavaScript

```javascript
async function async1() {
    console.log('async1 start');
    await async2();
    console.log('async1 end');
}
async function async2() {
    console.log('async2');
}
console.log('script start');
setTimeout(function() {
    console.log('setTimeout');
}, 0)
async1();
new Promise(function(resolve) {
    console.log('promise1');
    resolve();
}).then(function() {
    console.log('promise2');
});
console.log('script end');
```

首先我们需要明白以下几件事情：

- JS分为同步任务和异步任务
- 同步任务都在主线程上执行，形成一个执行栈
- 主线程之外，事件触发线程管理着一个任务队列，只要异步任务有了运行结果，就在任务队列之中放置一个事件。
- 一旦执行栈中的所有同步任务执行完毕（此时JS引擎空闲），系统就会读取任务队列，将可运行的异步任务添加到可执行栈中，开始执行。

根据规范，事件循环是通过任务队列的机制来进行协调的。一个 Event Loop 中，可以有一个或者多个任务队列(task queue)，一个任务队列便是一系列有序任务(task)的集合；**每个任务都有一个任务源(task source)，源自同一个任务源的 task 必须放到同一个任务队列，从不同源来的则被添加到不同队列。**setTimeout/Promise 等API便是任务源，而进入任务队列的是他们指定的具体执行任务。



宏任务

(macro)task（又称之为宏任务），可以理解是每次执行栈执行的代码就是一个宏任务（包括每次从事件队列中获取一个事件回调并放到执行栈中执行）。

浏览器为了能够使得JS内部(macro)task与DOM任务能够有序的执行，**会在一个(macro)task执行结束后，在下一个(macro)task 执行开始前，对页面进行重新渲染**，流程如下：

```Shell
(macro)task->渲染->(macro)task->...
```

(macro)task主要包含：script(整体代码)、setTimeout、setInterval、I/O、UI交互事件、postMessage、MessageChannel、setImmediate(Node.js 环境)

微任务

microtask（又称为微任务），**可以理解是在当前 task 执行结束后立即执行的任务**。也就是说，在当前task任务后，下一个task之前，在渲染之前。

所以它的响应速度相比setTimeout（setTimeout是task）会更快，因为无需等渲染。也就是说，在某一个macrotask执行完后，就会将在它执行期间产生的所有microtask都执行完毕（在渲染前）。

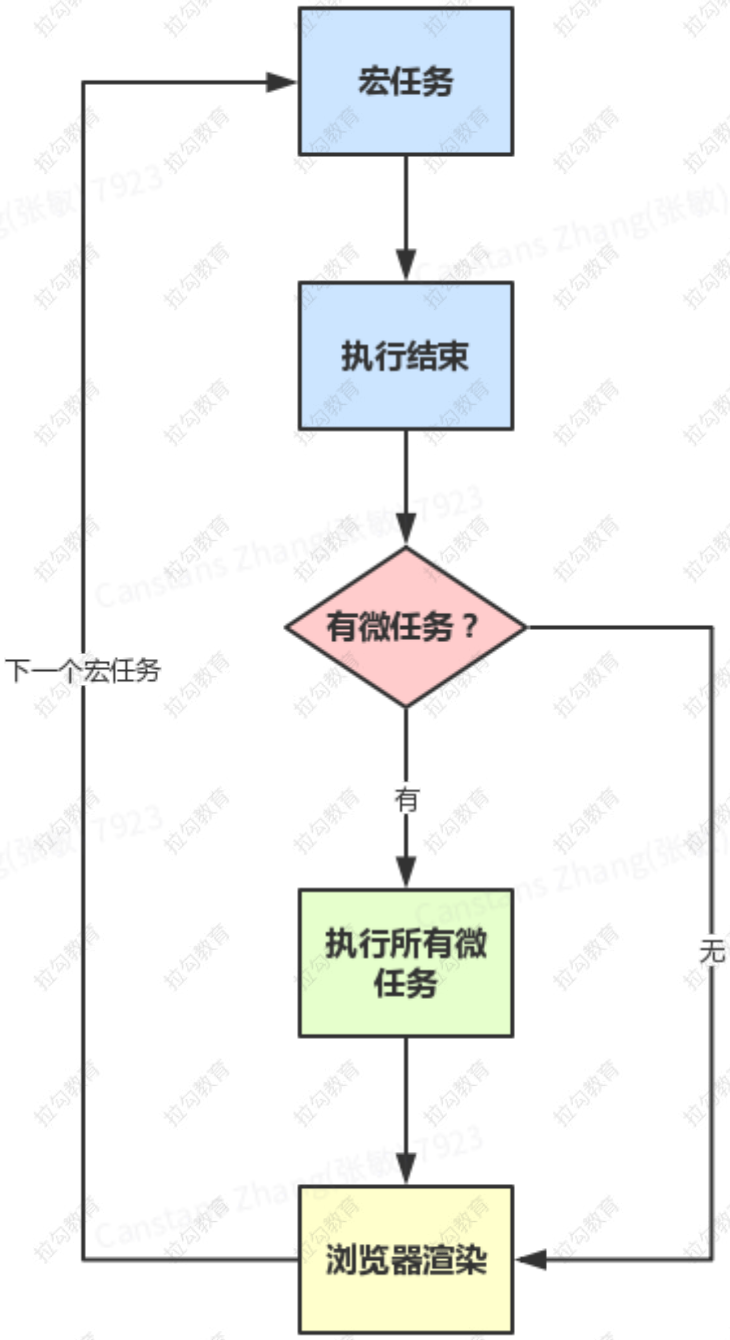
microtask主要包含：Promise.then、MutaionObserver、process.nextTick(Node.js 环境)

运行机制

在事件循环中，每进行一次循环操作称为 tick，每一次 tick 的任务处理模型是比较复杂的，但关键步骤如下：

- 执行一个宏任务（栈中没有就从事件队列中获取）
- 执行过程中如果遇到微任务，就将它添加到微任务的任务队列中
- 宏任务执行完毕后，立即执行当前微任务队列中的所有微任务（依次执行）
- 当前宏任务执行完毕，开始检查渲染，然后GUI线程接管渲染
- 渲染完毕后，JS线程继续接管，开始下一个宏任务（从事件队列中获取）

流程图如下：



Promise和async中的立即执行

我们知道Promise中的异步体现在then和catch中，所以写在Promise中的代码是被当做同步任务立即执行的。而在async/await中，在出现await出现之前，其中的代码也是立即执行的。那么出现了await时候发生了什么呢?

await做了什么

从字面意思上看await就是等待，await 等待的是一个表达式，这个表达式的返回值可以是一个promise对象也可以是其他值。

很多人以为await会一直等待之后的表达式执行完之后才会继续执行后面的代码，**实际上await是一个让出线程的标志。await后面的表达式会先执行一遍，将await后面的代码加入到microtask中，然后就会跳出整个async函数来执行后面的代码。**

这里感谢@chenjigeng的纠正：

由于因为async await 本身就是promise+generator的语法糖。所以await后面的代码是microtask。所以对于本题中的

asyncfunctionasync1(){console.log('async1 start');awaitasync2();console.log('async1 end');}

等价于

asyncfunctionasync1(){console.log('async1 start');Promise.resolve(async2()).then(()=>{console.log('async1 end');})}

回到本题

以上就本道题涉及到的所有相关知识点了，下面我们再回到这道题来一步一步看看怎么回事儿。

1. 首先，事件循环从宏任务(macrotask)队列开始，这个时候，宏任务队列中，只有一个script(整体代码)任务；当遇到任务源(task source)时，则会先分发任务到对应的任务队列中去。所以，上面例子的第一步执行如下图所示：



2. 然后我们看到首先定义了两个async函数，接着往下看，然后遇到了console语句，直接输出script start。输出之后，script 任务继续往下执行，遇到setTimeout，其作为一个宏任务源，则会先将其任务分发到对应的队列中：



3. script 任务继续往下执行，执行了async1()函数，前面讲过async函数中在await之前的代码是立即执行的，所以会立即输出async1 start。

遇到了await时，会将await后面的表达式执行一遍，所以就紧接着输出async2，然后将await后面的代码也就是console.log('async1 end')加入到microtask中的Promise队列中，接着跳出async1函数来执行后面的代码。



4. script任务继续往下执行，遇到Promise实例。由于Promise中的函数是立即执行的，而后续的.then则会被分发到 microtask 的Promise队列中去。所以会先输出promise1，然后执行resolve，将promise2分配到对应队列。



5. script任务继续往下执行，最后只有一句输出了script end，至此，全局任务就执行完毕了。

根据上述，每次执行完一个宏任务之后，会去检查是否存在 Microtasks；如果有，则执行 Microtasks 直至清空 Microtask Queue。

因而在script任务执行完毕之后，开始查找清空微任务队列。此时，微任务中，Promise队列有的两个任务async1 end和promise2，因此按先后顺序输出async1 end，promise2。当所有的 Microtasks 执行完毕之后，表示第一轮的循环就结束了。

6. ~~第二轮循环开始，这个时候就会跳回async1函数中执行后面的代码，然后遇到了同步任务console语句，直接输出async1 end。这样第二轮的循环就结束了。（也可以理解为被加入到script任务队列中，所以会先与setTimeout队列执行）~~
7. 第二轮循环依旧从宏任务队列开始。此时宏任务中只有一个setTimeout，取出直接输出即可，至此整个流程结束。

4.JS 异步解决方案的发展历程以及优缺点。

1. 回调函数（callback）

JavaScript

```
setTimeout(() => {
    // callback 函数体
}, 1000)
```

缺点：回调地狱，不能用 try catch 捕获错误，不能 return

回调地狱的根本问题在于：

- 缺乏顺序性： 回调地狱导致的调试困难，和大脑的思维方式不符
- 嵌套函数存在耦合性，一旦有所改动，就会牵一发而动全身，即（**控制反转**）
- 嵌套函数过多的多话，很难处理错误

JavaScript

```
ajax('XXX1', () => {
    // callback 函数体
    ajax('XXX2', () => {
        // callback 函数体
        ajax('XXX3', () => {
            // callback 函数体
        })
    })
})
```

优点：解决了同步的问题（只要有一个任务耗时很长，后面的任务都必须排队等着，会拖延整个程序的执行。）

2. Promise

Promise就是为了解决callback的问题而产生的。

Promise 实现了链式调用，也就是说每次 then 后返回的都是一个全新 Promise，如果我们在 then 中 return ，return 的结果会被 Promise.resolve() 包装

优点：解决了回调地狱的问题

JavaScript

```
ajax('XXX1')
  .then(res => {
      // 操作逻辑
      return ajax('XXX2')
  }).then(res => {
      // 操作逻辑
      return ajax('XXX3')
  }).then(res => {
      // 操作逻辑
  })
```

缺点：无法取消 Promise ，错误需要通过回调函数来捕获

3. Generator

特点：可以控制函数的执行，可以配合 co 函数库使用

JavaScript

```
function *fetch() {
    yield ajax('XXX1', () => {})
    yield ajax('XXX2', () => {})
    yield ajax('XXX3', () => {})
}
let it = fetch()
let result1 = it.next()
let result2 = it.next()
let result3 = it.next()
```

4. Async/await

async、await 是异步的终极解决方案

优点是：代码清晰，不用像 Promise 写一大堆 then 链，处理了回调地狱的问题

缺点：await 将异步代码改造成同步代码，如果多个异步操作没有依赖性而使用 await 会导致性能上的降低。

JavaScript

```
async function test() {
  // 以下代码没有依赖性的话，完全可以使用 Promise.all 的方式
  // 如果有依赖性的话，其实就是解决回调地狱的例子了
  await fetch('XXX1')
  await fetch('XXX2')
  await fetch('XXX3')
}
```

下面来看一个使用await的例子：

JavaScript

```
let a = 0
let b = async () => {
  a = a + await 10
  console.log('2', a) // -> '2' 10
}
b()
a++
console.log('1', a) // -> '1' 1
```

对于以上代码你可能会有疑惑，让我来解释下原因

- 首先函数b先执行，在执行到await 10之前变量a还是 0，因为await内部实现了generator，**generator会保留堆栈中东西，所以这时候a = 0被保存了下来**
- 因为await是异步操作，后来的表达式不返回Promise的话，就会包装成Promise.reslove(返回值)，然后会去执行函数外的同步代码
- 同步代码执行完毕后开始执行异步代码，将保存下来的值拿出来使用，这时候a = 0 + 10

上述解释中提到了await内部实现了generator，其实await就是generator加上Promise的语法糖，且内部实现了自动执行generator。如果你熟悉 co 的话，其实自己就可以实现这样的语法糖。

5.Promise 构造函数是同步执行还是异步执行，那么 then 方法呢?

CoffeeScript

```coffeescript
const promise = new Promise((resolve, reject) => {
  console.log(1)
  resolve()
  console.log(2)
})

promise.then(() => {
  console.log(3)
})

console.log(4)
```

执行结果是：1243

promise构造函数是同步执行的，then方法是异步执行的

6.模拟实现一个 Promise.finally

JavaScript

```javascript
window.Promise && !('finally' in Promise) && !function() {
  Promise.prototype.finally = function(cb) {
    cb = typeof cb === 'function' ? cb : function() {};

    var Fn = this.constructor;  // 获取当前实例构造函数的引用

    // 接受状态：返回数据
    var onFulfilled = function(data) {
      return Fn.resolve(cb()).then(function() {
        return data
      })
    };

    // 拒绝状态：抛出错误
    var onRejected = function(err) {
      return Fn.resolve(cb()).then(function() {
        throw err
      })
    };

    return this.then(onFulfilled, onRejected);
  }
}();
```

```
})();

/********************** 测试 **********************/
const p = new Promise((resolve, reject) => {
  console.info('starting...');

  setTimeout(() => {
    Math.random() > 0.5 ? resolve('success') : reject('fail');
  }, 1000);
});

// 正常顺序测试
p.then((data) => {
    console.log(`%c resolve: ${data}`, 'color: green')
  })
  .catch((err) => {
    console.log(`%c catch: ${err}`, 'color: red')
  })
  .finally(() => {
    console.info('finally: completed')
  });

// finally 前置测试
p.finally(() => {
    console.info('finally: completed')
  })
  .then((data) => {
    console.log(`%c resolve: ${data}`, 'color: green')
  })
  .catch((err) => {
    console.log(`%c catch: ${err}`, 'color: red')
  });
```

7.介绍下 Promise.all 使用、原理实现及错误处理

一、Promise概念

Promise是JS异步编程中的重要概念，异步抽象处理对象，是目前比较流行Javascript异步编程解决方案之一。Promise.all()接受一个由promise任务组成的数组，可以同时处理多个promise任务，当所有的任务都执行完成时，Promise.all()返回resolve，但当有一个失败(reject)，则返回失败的信息，即使其他promise执行成功，也会返回失败。和后台的事务类似。和rxjs中的forkJoin方法类似，合并多个 Observable 对象 ，等到所有的 Observable 都完成后，才一次性返回值。

二、Promise.all如何使用

对于 Promise.all(arr) 来说，在参数数组中所有元素都变为决定态后，然后才返回新的 promise。

JavaScript

```
// 以下 demo，请求两个 url，当两个异步请求返还结果后，再请求第三个 url
const p1 = request(`http://some.url.1`)
const p2 = request(`http://some.url.2`)
Promise.all([p1, p2])
  .then((datas) => { // 此处 datas 为调用 p1, p2 后的结果的数组
    return request(`http://some.url.3?a=${datas[0]}&b=${datas[1]}`)
  })
  .then((data) => {
    console.log(msg)
  })
```

三、Promise.all原理实现

JavaScript

```
function promiseAll(promises){
     return new Promise(function(resolve,reject){
           if(!Array.isArray(promises)){
            return reject(new TypeError("argument must be anarray"))
          }
    var countNum=0;
    var promiseNum=promises.length;
    var resolvedvalue=new Array(promiseNum);
    for(var i=0;i<promiseNum;i++){
      (function(i){
         Promise.resolve(promises[i]).then(function(value){
            countNum++;
           resolvedvalue[i]=value;
          if(countNum===promiseNum){
              return resolve(resolvedvalue)
          }
       },function(reason){
        return reject(reason)
      )
     })(i)
    }
})
}
var p1=Promise.resolve(1),
p2=Promise.resolve(2),
p3=Promise.resolve(3);
promiseAll([p1,p2,p3]).then(function(value){
console.log(value)
})
```

四、Promise.all错误处理

有时候我们使用Promise.all()执行很多个网络请求，可能有一个请求出错，但我们并不希望其他的网络请求也返回reject，要错都错，这样显然是不合理的。如何做才能做到promise.all中即使一个promise程序reject，promise.all依然能把其他数据正确返回呢?

1、全部改为串行调用（失去了node 并发优势）

2、当promise捕获到error 的时候，代码吃掉这个异常，返回resolve，约定特殊格式表示这个调用成功了

JavaScript

```
var p1 =new Promise(function(resolve,reject){
    setTimeout(function(){
        resolve(1);
    },0)
});
var p2 = new Promise(function(resolve,reject){
        setTimeout(function(){
            resolve(2);
        },200)
 });
 var p3 = new Promise(function(resolve,reject){
        setTimeout(function(){
            try{
            console.log(XX.BBB);
            }
            catch(exp){
                resolve("error");
            }
        },100)
});
Promise.all([p1, p2, p3]).then(function (results) {
    console.log("success")
     console.log(results);
}).catch(function(r){
    console.log("err");
    console.log(r);
});
```

8.设计并实现 Promise.race()

JavaScript

```
Promise.myrace = function(iterator) {
    return new Promise ((resolve,reject) => {
        try {
            let it = iterator[Symbol.iterator]();
            while(true) {
                let res = it.next();
                console.log(res);
                if(res.done) break;
                if(res.value instanceof Promise) {
                    res.value.then(resolve,reject);
                } else {
                    resolve(res.value)
                }

            }
        } catch (error) {
            reject(error)
        }
    })
}
```

五 构建工具

1.webpack相关问题

1.loader和plugin区别

loader用于加载某些资源文件，因为webpack本身只能打包CommonJS规范的js文件，对于其他资源，例如css，图片等，是没有办法加载的，这就需要对应的loader将资源转换 plugin用于扩展webpack的功能，直接作用于webpack，loader只专注于转换文件，而plugin不仅局限于资源加载

Loader只能处理单一文件的输入输出，而Plugin则可以对整个打包过程获得更多的灵活性，譬如ExtractTextPlugin，它可以将所有文件中的css剥离到一个独立的文件中，这样样式就不会随着组件加载而加载了。

2.什么是chunk

Webpack提供一个功能可以拆分模块，每一个模块称为chunk，这个功能叫做Code Splitting。你可以在你的代码库中定义分割点，调用require.ensure，实现按需加载

3.如何开发一个loader，原理是啥

A loader is a node module exporting a function.

缓存： Webpack Loader 同样可以利用缓存来提高效率，并且只需在一个可缓存的 Loader 上加一句 this.cacheable() 异步：在一个异步的模块中，回传时需要调用 Loader API 提供的回调方法 this.async()

4.打包原理

webpack打包，最基本的实现方式，是将所有的模块代码放到一个数组里，通过数组ID来引用不同的模块

```
/************************************************************************/
/******/ ([
/* 0 */
/***/ function(module, exports, __webpack_require__) {

  __webpack_require__(1);
  __webpack_require__(2);
  console.log('Hello, world!');

/***/ },
/* 1 */
/***/ function(module, exports) {

  var a = 'a.js';
  console.log("I'm a.js");

/***/ },
```

```
/* 2 */
/***/ function(module, exports) {

  var b = 'b.js';
  console.log("I'm b.js");

/***/ }
/******/ ]);
```

可以发现入口entry.js的代码是放在数组索引0的位置，其它a.js和b.js的代码分别放在了数组索引1和2的位置，而webpack引用的时候，主要通过__webpack_require__的方法引用不同索引的模块。

5.webpack和gulp的区别

webpack是一种模块化打包工具，主要用于模块化方案，预编译模块的方案；gulp是工具链、构建工具，可以配合各种插件做js压缩，css压缩，less编译 替代手工实现自动化工作。

Grunt/Gulp更多的是一种工作流；提供集成所有服务的一站式平台； gulp可以用来优化前端工作流程。

6.如何写一个plugin

Compiler在开始打包时就进行实例化，实例对象里面装着与打包相关的环境和参数，包括options、plugins和loaders等。

compilation对象，它继承于compiler，所以能拿到一切compiler的内容。Compilation表示有关模块资源，已编译资源，Compilation在每次文件变化重新打包时都进行一次实例化

apply方法：当安装这个插件的时候，这个apply方法就会被webpack compiler调用。

```
function HelloWorldPlugin(options) {
  // Setup the plugin instance with options...
}

HelloWorldPlugin.prototype.apply = function(compiler) {
```

```
  compiler.plugin('done', function() {
    console.log('Hello World!');
  });
};

module.exports = HelloWorldPlugin;
```

7.webpack打包后文件体积过大怎么办?

很多方法：异步加载模块（代码分割）；提取第三方库（使用cdn或者vender）；代码压缩；去除不必要的插件；去除devtool选项，dllplugin等等。

六 前端框架

1.jQuery

1.jQuery.fn的init方法返回的this指的是什么对象？为什么要返回this?

this执行init构造函数自身，其实就是jQuery实例对象，返回this是为了实现jQuery的链式操作

2.jquery中如何将数组转化为json字符串，然后再转化回来?

jQuery中没有提供这个功能，所以你需要先编写两个jQuery的扩展：

```
$.fn.stringifyArray = function(array) {
    return JSON.stringify(array)
}

$.fn.parseArray = function(array) {
    return JSON.parse(array)
}
```

然后调用：

```
$("").stringifyArray(array)
```

3.jQuery 的属性拷贝(extend)的实现原理是什么，如何实现深拷贝?

递归赋值

4.jquery.extend 与 jquery.fn.extend的区别?

* jquery.extend 为jquery类添加类方法，可以理解为添加静态方法

* jquery.fn.extend:

源码中jquery.fn = jquery.prototype，所以对jquery.fn的扩展，就是为jquery类添加成员函数

使用：

jquery.extend扩展，需要通过jquery类来调用，而jquery.fn.extend扩展，所有jquery实例都可以直接调用。

5.针对 jQuery 的优化方法?

*基于Class的选择性的性能相对于Id选择器开销很大，因为需遍历所有DOM元素。

*频繁操作的DOM，先缓存起来再操作。用Jquery的链式调用更好。

比如：var str=$("a").attr("href");

*for (var i = size; i < arr.length; i++) {}

for 循环每一次循环都查找了数组 (arr) 的.length 属性，在开始循环的时候设置一个变量来存储这个数字，可以让循环跑得更快：

for (var i = size, length = arr.length; i < length; i++) {}

6.JQuery一个对象可以同时绑定多个事件，这是如何实现的?

* 多个事件同一个函数：

```
$("div").on("click mouseover", function(){});
```

* 多个事件不同函数

```
$("div").on({
    click: function(){},
    mouseover: function(){}
});
```

7.bind(), live(), delegate()的区别

bind： 绑定事件，对新添加的事件不起作用，方法用于将一个处理程序附加到每个匹配元素的事件上并返回jQuery对象。

live： 方法将一个事件处理程序附加到与当前选择器匹配的所有元素（包含现有的或将来添加的）的指定事件上并返回jQuery对象。

delegate： 方法基于一组特定的根元素将处理程序附加到匹配选择器的所有元素（现有的或将来的）的一个或多个事件上。

8.jQuery框架中$.ajax()的常用参数有哪些？写一个post请求并带有发送数据和返回数据的样例

async是否异步

url请求地址

contentType发送信息至服务器时内容编码类型

data发送到服务器的数据

dataType预期服务器返回的数据类型

type请求类型

success请求成功回调函数

error请求失败回调函数

```
$.ajax({
        url: "/jquery/test1.txt",
        type: 'post',
        data: {
          id: 1
        },
        success: function(data) {
          alert(data);
        }
    }
```

9.jQuery一个对象可以同时绑定多个事件，这是如何实现的?

jQuery可以给一个对象同时绑定多个事件，低层实现方式是使用addEventListner或attachEvent兼容不同的浏览器实现事件的绑定，这样可以给同一个对象注册多个事件。

10.jquery 中如何将数组转化为json字符串，然后再转化回来?

jQuery中没有提供这个功能，所以你需要先编写两个jQuery的扩展：

```
$.fn.stringifyArray = function(array) {
    return JSON.stringify(array)
}
$.fn.parseArray = function(array) {
    return JSON.parse(array)
}
```

然后调用：

```
$("").stringifyArray(array)
```

11.Jquery与jQuery UI有啥区别?

jQuery是操作dom的框架，jQueryUI是基于jQuery做的一个UI组件库

12.jQuery和Zepto的区别？各自的使用场景?

jQuery主要用于pc端，当然有对应的jQuerymobile用于移动端，zepto比jQuery更加小巧，主要用于移动端

13.针对 jQuery 的优化方法?

优先使用ID选择器

在class前使用tag(标签名)

给选择器一个上下文

慎用 .live()方法（应该说尽量不要使用）

使用data()方法存储临时变量

14.Zepto的点透问题如何解决?

点透主要是由于两个div重合，例如：一个div调用show()，一个div调用hide()；这个时候当点击上面的div的时候就会影响到下面的那个div；

解决办法主要有2种：

1.github上有一个叫做fastclick的库，它也能规避移动设备上click事件的延迟响应，https://github.com/ftlabs/fastclick

将它用script标签引入页面（该库支持AMD，于是你也可以按照AMD规范，用诸如require.js的模块加载器引入），并且在dom ready时初始化在body上，

2.根据分析，如果不引入其它类库，也不想自己按照上述fastclcik的思路再开发一套东西，需要1.一个优先于下面的“divClickUnder”捕获的事件；2.并且通过这个事件阻止掉默认行为（下面

的“divClickUnder”对click事件的捕获，在ios的safari，click的捕获被认为和滚屏、点击输入框弹起键盘等一样，是一种浏览器默认行为，即可以被event.preventDefault()阻止的行为）。

2.angular

1.使用过angular吗？angular中的过滤器是干什么用的

在表达式中转换数据 姓名为 {{ lastName | uppercase }}

currency，是什么过滤器——格式化数字为货币格式，单位是$符。

3.react

1.react和vue的区别

相同点：

都支持服务端渲染

都有Virtual DOM，组件化开发，通过props参数进行父子组件数据的传递，都实现webComponents规范

数据驱动视图

都有支持native的方案，React的React native，Vue的weex

不同点：

React严格上只针对MVC的view层，Vue则是MVVM模式

virtual DOM 不一样 vue会跟踪每一个组件的依赖关系，不需要重新渲染整个组件树。而对于React而言，每当应用的状态被改变时，全部子组件都会重新渲染。当然，这可以通过shouldComponentUpdate这个生命周期方法来进行控制，

组件写法不一样 React 推荐的做法是 JSX + inline style，也就是把 HTML 和 CSS 全都写进 JavaScript 了，即”all in js” Vue 推荐的是使用 webpack + vue-loader 的单文件组件格式，即html,css,js写在同一个文件；

数据绑定：Vue有实现了双向数据绑定，React数据流动是单向的

state对象在react应用中是不可变的，需要使用setState方法更新状态；在Vue中，state对象并不是必须的，数据由data属性在Vue对象中进行管理。

2.react的优缺点

优点：

1. 可以通过函数式方法描述视图组件（好处：相同的输入会得到同样的渲染结果，不会有副 作 用；组件不会被实例化，整体渲染性能得到提升）

2. 集成虚拟DOM（性能好）

3. 单向数据流（好处是更容易追踪数据变化排查问题

4. 一切都是component：代码更加模块化，重用代码更容易，可维护性高

5. 大量拥抱 es6 新特性

6: jsx

缺点：

1. jsx的一个问题是，渲染函数常常包含大量逻辑，最终看着更像是程序片段，而不是视觉呈现。后期如果发生需求更改，维护起来工作量将是巨大的

2. 大而全，上手有难度

3.jsx的优缺点

允许使用熟悉的语法来定义HTML元素树 JSX 让小组件更加简单、明了、直观。 更加语义化且易懂的标签 JSX 本质是对JavaScript语法的一个扩展，看起来像是某种模板语言，但其实不是。但正因为形似HTML，描述UI就更直观了，也极大地方便了开发； 在React中babel会将JSX转换为React.createElement函数调用，然后将JSX转换为正确的JSON对象（VDOM 也是一个“树”形的结构） React/JSX乍看之下，觉得非常啰嗦，但使用JavaScript而不是模板语法来开发（模板语法比较有局限性），赋予了开发者许多编程能力。

4.dom diff算法和虚拟DOM

React中的render方法，返回一个DOM描述，结果仅仅是轻量级的js对象。Reactjs只在调用setState的时候会更新dom，而且还是先更新Virtual Dom，然后和实际DOM比较，最后再更新实际DOM。

React.js 厉害的地方并不是说它比 DOM 快（这句话本来就是错的），而是说不管你数据怎么变化，我都可以以最小的代价来更新 DOM。方法就是我在内存里面用新的数据刷新一个虚拟的 DOM 树，然后新旧 DOM 树进行比较，找出差异，再更新到真正的 DOM 树上。

当我们修改了DOM树上一些节点对应绑定的state，React会立即将它标记为“脏状态”。在事件循环的最后才重新渲染所有的脏节点。在实际的代码中，会对新旧两棵树进行一个深度优先的遍历，这样每个节点都会有一个唯一的标记，每遍历到一个节点就把该节点和新的的树进行对比。如果有差异的话就记录到一个对象里面，最后把差异应用到真正的DOM树上。 算法实现 1 步骤一：用JS对象模拟DOM树 2 步骤二：比较两棵虚拟DOM树的差异 3 步骤三：把差异应用到真正的DOM树上 这就是所谓的 diff 算法

dom diff采用的是增量更新的方式，类似于打补丁。React 需要为数据添加 key 来保证虚拟 DOM diff 算法的效率。key属性可以帮助React定位到正确的节点进行比较，从而大幅减少DOM操作次数，提高了性能。

virtual dom，也就是虚拟节点。它通过JS的Object对象模拟DOM中的节点，然后再通过特定的render方法将其渲染成真实的DOM节点。 http://react-china.org/t/dom/638

为什么js对象模拟DOM会比js操作DOM来得快

为了解决频繁操作DOM导致Web应用效率下降的问题，React提出了“虚拟DOM”（virtual DOM）的概念。Virtual DOM是使用JavaScript对象模拟DOM的一种对象结构。DOM树中所有的信息都可以用JavaScript表述出来，例如：

```
<ul>
 <li>Item 1</li>
 <li>Item 2</li>
 <li>Item 3</li>
</ul>
```

可以用以下JavaScript对象来表示：

```
{
 tag: 'ul',
 children: [{
  tag: 'li', children: ['Item 1'],
  tag: 'li', children: ['Item 2'],
  tag: 'li', children: ['Item 3']
 }]
}
```

这样可以避免直接频繁地操作DOM，只需要在js对象模拟的虚拟DOM进行比对，再将更改的部分应用到真实的DOM树上

react组件性能优化

使用PureRenderMixin、shouldComponentUpdate来避免不必要的虚拟DOM diff，在render内部优化虚拟DOM的diff速度，以及让diff结果最小化

5.react组件间的数据传递

1.兄弟组件不能直接相互传送数据，此时可以将数据挂载在父组件中，由两个组件共享

2.子组件向父组件通讯，可以通过父组件定义事件（回调函数），子组件调用该函数，通过实参的形式来改变父组件的数据来通信

```
//子组件
this.props.onCommentSubmit({author, content, date:new Date().getTime()});
//父组件
render(){
  return(
   <div className="m-index">
    <div>
     <h1>评论</h1>
    </div>
    <CommentList data={this.state.data} />
    <CommentForm onCommentSubmit={this.handleCommentSubmit.bind(this)} />
   </div>
  )
}
```

3.非父子组件间的通信：可以使用全局事件来实现组件间的沟通，React中可以引入eventProxy模块，利用eventProxy.trigger()方法发布消息，eventProxy.on()方法监听并接收消息。

4.组件间层级太深，可以使用上下文方式，让子组件直接访问祖先的数据或函数，通过this.context.xx

6.无状态组件

无状态组件其实本质上就是一个函数，传入props即可，没有state，也没有生命周期方法。组件本身对应的就是render方法。例子如下：

```
function Title({color = 'red', text = '标题'}) {
 let style = {
  'color': color
 }
 return (
  <div style = {style}>{text}</div>
 )
}
```

无状态组件不会创建对象，故比较省内存。没有复杂的生命周期方法调用，故流程比较简单。没有state，也不会重复渲染。它本质上就是一个函数而已。

对于没有状态变化的组件，React建议我们使用无状态组件。总之，能用无状态组件的地方，就用无状态组件。

7.高阶组件

高阶组件（HOC）是函数接受一个组件，返回一个新组件。其前身其实是用ES5创建组件时可用的mixin方法，但是在react版本升级过程中，使用ES6语法创建组件时，认为mixin是反模式，影响了react架构组件的封装稳定性，增加了不可控的复杂度，逐渐被HOC所替代。 实现高阶组件的方式有：

属性代理

```
import React, { Component } from 'React';
//高阶组件定义
const HOC = (WrappedComponent) =>
 class WrapperComponent extends Component {
  render() {
   return <WrappedComponent {...this.props} />;
  }
```

```
}
//普通的组件
class WrappedComponent extends Component{
   render(){
      //....
   }
}

//高阶组件使用
export default HOC(WrappedComponent)
```

反向继承

反向继承是指返回的组件去继承之前的组件(这里都用WrappedComponent代指)

```
const HOC = (WrappedComponent) =>
 class extends WrappedComponent {
   render() {
    return super.render();
   }
 }
```

我们可以看见返回的组件确实都继承自WrappedComponent,那么所有的调用将是反向调用的(例如:super.render())，这也就是为什么叫做反向继承。

8.react事件和传统事件有什么区别吗

React 实现了一个“合成事件”层（synthetic event system），这个事件模型保证了和 W3C 标准保持一致，所以不用担心有什么诡异的用法，并且这个事件层消除了 IE 与 W3C 标准实现之间的兼容问题。

“合成事件”还提供了额外的好处：

事件委托

“合成事件”会以事件委托（event delegation）的方式绑定到组件最上层，并且在组件卸载（unmount）的时候自动销毁绑定的事件。

9.react的生命周期图

1.组件更新过程

react组件更新过程：

props/state change：

1.componentWillReceiveProps(nextProps)

只要是父组件的render被调用，在render中被渲染的子组件就会经历更新的过程。不管父组件传给子组件的props有没有改变，都会触发子组件的此函数被调用。注意：通过setState方法触发的更新不会调用此函数

2.shouldComponentUpdate(nextProps,nextState) 3.componentWillUpdate 4.render 5.componentDidUpdate

10.写 React / Vue 项目时为什么要在列表组件中写 key，其作用是什么?

key是给每一个vnode的唯一id,可以 `依靠key` ,更 `准确` , 更 `快` 的拿到oldVnode中对应的vnode节点。

vue和react都是采用diff算法来对比新旧虚拟节点，从而更新节点。在vue的diff函数中（建议先了解一下diff算法过程）。

在交叉对比中，当新节点跟旧节点 `头尾交叉对比` 没有结果时，会根据新节点的key去对比旧节点数组中的key，从而找到相应旧节点（这里对应的是一个key => index 的map映射）。如果没找到就认为是一个新增节点。而如果没有key，那么就会采用遍历查找的方式去找到对应的旧节点。一种一个map映射，另一种是遍历查找。相比而言。map映射的速度更快。

11. React 中 setState 什么时候是同步的，什么时候是异步的?

在React中，**如果是由React引发的事件处理（比如通过onClick引发的事件处理），调用setState不会同步更新this.state，除此之外的setState调用会同步执行this.state**。所谓“除此之外”，指的是绕过React通过addEventListener直接添加的事件处理函数，还有通过setTimeout/setInterval产生的异步调用。

原因：在React的setState函数实现中，会根据一个变量isBatchingUpdates判断是直接更新this.state还是放到队列中回头再说，而isBatchingUpdates默认是false，也就表示setState会同步更新this.state，但是，**有一个函数batchedUpdates，这个函数会把isBatchingUpdates修改为true，而当React在调用事件处理函数之前就会调用这个batchedUpdates，造成的后果，就是由React控制的事件处理过程setState不会同步更新this.state。**

注意：setState的“异步”并不是说内部由异步代码实现，其实本身执行的过程和代码都是同步的，只是合成事件和钩子函数的调用顺序在更新之前，导致在合成事件和钩子函数中没法立马拿到更新后的值，形式了所谓的“异步”，当然可以通过第二个参数 setState(partialState, callback) 中的callback拿到更新后的结果。

12.React setState 笔试题，下面的代码输出什么？

JavaScript

```
class Example extends React.Component {
  constructor() {
    super();
    this.state = {
      val: 0
    };
  }

  componentDidMount() {
    this.setState({val: this.state.val + 1});
    console.log(this.state.val);    // 第 1 次 log

    this.setState({val: this.state.val + 1});
    console.log(this.state.val);    // 第 2 次 log

    setTimeout(() => {
      this.setState({val: this.state.val + 1});
      console.log(this.state.val);  // 第 3 次 log

      this.setState({val: this.state.val + 1});
      console.log(this.state.val);  // 第 4 次 log
    }, 0);
  }

  render() {
    return null;
  }
};
```

0023

13.聊聊 Redux 和 Vuex 的设计思想

首先两者都是处理全局状态的工具库，大致实现思想都是：全局state保存状态---->dispatch(action)

------>reducer(vuex里的mutation)----> 生成newState; 整个状态为同步操作；

最大的区别在于处理异步的不同，vuex里面多了一步commit操作，在action之后commit(mutation)之前处理异步，而redux里面则是通过中间件处理

14.redux 为什么要把 reducer 设计成纯函数

redux三大原则

1. 单一数据流

整个应用state都被储存在一个store里面 构成一个Object tree

2. State是只读的

唯一改变state的方法就是触发action, action是一个用于描述已发生事件的普通对象

3. 使用纯函数来执行修改

为了描述action如何改变state tree， 你需要编写reducers

把reducer设计成纯函数，可以实现时间旅行，记录/回放或者热加载

15.react-router 里的 `<Link>` 标签和 `<a>` 标签有什么区别

从最终渲染的 DOM 来看，这两者都是链接，都是 `<a>` 标签，区别是：

`<Link>` 是 react-router 里实现路由跳转的链接，一般配合 `<Route>` 使用，react-router 接管了其默认的链接跳转行为，区别于传统的页面跳转， `<Link>` 的“跳转”行为只会触发相匹配的 `<Route>` 对应的页面内容更新，而不会刷新整个页面。

而 `<a>` 标签就是普通的超链接了，用于从当前页面跳转到 href 指向的另一个页面（非锚点情况）。

16.React 和 Vue 的 diff 时间复杂度从 O(n^3) 优化到 O(n) ，那么 O(n^3) 和 O(n) 是如何计算出来的?

我认为react做了三种优化来降低复杂度：1：如果父节点不同，放弃对子节点的比较，直接删除旧节点然后添加新的节点重新渲染；2：如果子节点有变化，Virtual DOM不会计算变化的是什么，而是重新渲染，3：通过唯一的key策略

10.写 React / Vue 项目时为什么要在列表组件中写 key，其作用是什么?

10.写 React / Vue 项目时为什么要在列表组件中写 key，其作用是什么?

10.写 React / Vue 项目时为什么要在列表组件中写 key，其作用是什么?

10.写 React / Vue 项目时为什么要在列表组件中写 key，其作用是什么?

4.vue

1.vue 双向绑定底层实现原理

vue.js 采用数据劫持的方式，结合发布者-订阅者模式，通过Object.defineProperty()来劫持各个属性的setter，getter以监听属性的变动，在数据变动时发布消息给订阅者，触发相应的监听回调：

https://github.com/hawx1993/tech-blog/issues/11

2.vue 虚拟DOM和react 虚拟DOM的区别

在渲染过程中，会跟踪每一个组件的依赖关系，不需要重新渲染整个组件树。而对于React而言，每当应用的状态被改变时，全部子组件都会重新渲染。 在 React 应用中，当某个组件的状态发生变化时，它会以该组件为根，重新渲染整个组件子树。 如要避免不必要的子组件的重新渲染，你需要在所有可能的地方使用 PureComponent，或是手动实现shouldComponentUpdate 方法

在React中，数据流是自上而下单向的从父节点传递到子节点，所以组件是简单且容易把握的，子组件只需要从父节点提供的props中获取数据并渲染即可。如果顶层组件的某个prop改变了，React会递归地向下遍历整棵组件树，重新渲染所有使用这个属性的组件。

3.v-show和v-if区别

与v-if不同的是，无论v-show的值为true或false，元素都会存在于HTML代码中；而只有当v-if的值为true，元素才会存在于HTML代码中

4.vue组件通信

非父子组件间通信，Vue 有提供 Vuex，以状态共享方式来实现同信，对于这一点，应该注意考虑平衡，从整体设计角度去考量，确保引入她的必要。

父传子: this.$refs.xxx 子传父: this.$parent.xxx

还可以通过$emit方法出发一个消息，然后$on接收这个消息

5.你如何评价vue

框架能够让我们跑的更快，但只有了解原生的JS才能让我们走的更远。

vue专注于MVVM中的viewModel层，通过双向数据绑定，把view层和Model层连接了起来。核心是用数据来驱动DOM。这种把directive和component混在一起的设计有一个非常大的问题，它导致了很多开发者滥用Directive（指令），出现了到处都是指令的情况。

优点： 1.不需要setState，直接修改数据就能刷新页面，而且不需要react的shouldComponentUpdate就能实现最高效的渲染路径。 2.渐进式的开发模式，模版方式->组件方式->路由整合->数据流整合->服务器渲染。上手的曲线更加平滑简单，而且不像react一上来就是组件全家桶 3.v-model给开发后台管理系统带来极大的便利，反观用react开发后台就是个杯具 4.html，css与js比react更优雅地结合在一个文件上。

缺点：指令太多，自带模板扩展不方便； 组件的属性传递没有react的直观和明显

6.说说你对MVVM的理解

Model层代表数据模型，可以在Model中定义数据修改和操作业务逻辑； view 代表UI组件。负责将数据模型转换成UI展现出来 ViewModel 是一个同步View和Model的对象

用户操作view层，view数据变化会同步到Model，Model数据变化会立即反应到view中。viewModel通过双向数据绑定把view层和Model层连接了起来

7.为什么选择vue

reactjs 的全家桶方式，实在太过强势，而自己定义的 JSX 规范，揉和在 JS 的组件框架里，导致如果后期发生页面改版工作，工作量将会巨大。

vue的核心：数据绑定 和 视图组件。

Vue的数据驱动：数据改变驱动了视图的自动更新，传统的做法你得手动改变DOM来改变视图，vuejs只需要改变数据，就会自动改变视图，一个字：爽。再也不用你去操心DOM的更新了，这就是MVVM思想的实现。

视图组件化：把整一个网页的拆分成一个个区块，每个区块我们可以看作成一个组件。网页由多个组件拼接或者嵌套组成

8.vue中mixin与extend区别

全局注册混合对象，会影响到所有之后创建的vue实例，而Vue.extend是对单个实例进行扩展。

mixin 混合对象（组件复用）

同名钩子函数（bind，inserted，update，componentUpdate，unbind）将混合为一个数组，因此都将被调用，混合对象的钩子将在组件自身钩子之前调用

methods，components，directives将被混为同一个对象。两个对象的键名（方法名，属性名）冲突时，取组件（而非mixin）对象的键值对

9.双向绑定和单向数据绑定的优缺点

只有 UI控件 才存在双向，非 UI控件 只有单向。 单向绑定的优点是可以带来单向数据流，这样的好处是流动方向可以跟踪，流动单一，没有状态, 这使得单向绑定能够避免状态管理在复杂度上升时产生的各种问题, 程序的调试会变得相对容易。单向数据流更利于状态的维护及优化，更利于组件之间的通信，更利于组件的复用

双向数据流的优点：

无需进行和单向数据绑定的那些CRUD（Create，Retrieve，Update，Delete）操作； 双向绑定在一些需要实时反应用户输入的场合会非常方便 用户在视图上的修改会自动同步到数据模型中去，数据模型中值的变化也会立刻同步到视图中去；

缺点：

双向数据流是自动管理状态的, 但是在实际应用中会有很多不得不手动处理状态变化的逻辑, 使得程序复杂度上升 无法追踪局部状态的变化 双向数据流，值和UI绑定，但由于各种数据相互依赖相互绑定，导致数据问题的源头难以被跟踪到

Vue 虽然通过 v-model 支持双向绑定，但是如果引入了类似redux的vuex，就无法同时使用 v-model。

双绑跟单向绑定之间的差异只在于，双向绑定把数据变更的操作隐藏在框架内部，调用者并不会直接感知。

<input v-model="something">

<!-- 等价于以下内容 -->

<input :value="something" @input="something = $event.target.value">

也就是说，你只需要在组件中声明一个name为value的props，并且通过触发input事件传入一个值，就能修改这个value。

10.VueJS 开发常见问题集锦

https://segmentfault.com/a/1190000010230843

11.谈谈你对组件的理解

一个组件应该有以下特征：

1. 可组合（Composeable）：一个组件易于和其它组件一起使用，或者嵌套在另一个组件内部。如果一个组件内部创建了另一个组件，那么说父组件拥有（own）它创建的子组件，通过这个特性，一个复杂的 2. UI 可以拆分成多个简单的 UI 组件；

3. 可重用（Reusable）：每个组件都是具有独立功能的，它可以被使用在多个 UI 场景；

4. 可维护（Maintainable）：每个小的组件仅仅包含自身的逻辑，更容易被理解和维护；

5. 可测试（Testable）：因为每个组件都是独立的，那么对于各个组件分别测试显然要比对于整个 UI 进行测试容易的多。

12.聊聊 Vue 的双向数据绑定，Model 如何改变 View，View 又是如何改变 Model 的

vue通过Object.defineProperty 劫持传进来的数据， 然后在数据getter的时候订阅重新编译模板的消息，然后通过js监听元素的事件，这里input事件也好，keyup事件也好，总是监听输入框值变化，将新的值重新赋值给被劫持的data,这样就会触发setter函数，再setter函数中就会去发布重新编译模板的消息；

13.在 Vue 中，子组件为何不可以修改父组件传递的 Prop

为何不能修改：为了保证数据的单向流动，便于对数据进行追踪，避免数据混乱。官网有详细的信息 prop

> 所有的 prop 都使得其父子 prop 之间形成了一个**单向下行绑定**：父级 prop 的更新会向下流动到子组件中，但是反过来则不行。这样会防止从子组件意外改变父级组件的状态，从而导致你的应用的数据流向难以理解。

vue如何知道修改了父组件传来的props?

下面的代码就是实现Vue提示修改props的操作，在组件 `initProps` 方法的时候，会对props进行defineReactive操作，传入的第四个参数是自定义的set函数，该函数会在触发props的set方法时执

行，当props修改了，就会运行这里传入的第四个参数，然后进行判断，如果不是root根组件，并且不是更新子组件，那么说明更新的是props，所以会警告

14.双向绑定和 vuex 是否冲突

在严格模式中使用Vuex，当用户输入时，v-model会试图直接修改属性值，但这个修改不是在mutation中修改的，所以会抛出一个错误。当需要在组件中使用vuex中的state时，有2种解决方案：1、在input中绑定value(vuex中的state)，然后监听input的change或者input事件，在事件回调中调用mutation修改state的值 2、使用带有setter的双向绑定计算属性

15.Vue 的响应式原理中 Object.defineProperty 有什么缺陷?

Object.defineProperty无法监控到数组下标的变化，导致直接通过数组的下标给数组设置值，不能实时响应。

16.Vue 的父组件和子组件生命周期钩子执行顺序是什么

1. 加载渲染过程 `父beforeCreate->父created->父beforeMount->子beforeCreate->子created->子beforeMount->子mounted->父mounted`
2. 子组件更新过程 `父beforeUpdate->子beforeUpdate->子updated->父updated`
3. 父组件更新过程 `父beforeUpdate->父updated`
4. 销毁过程 `父beforeDestroy->子beforeDestroy->子destroyed->父destroyed`

17.vue 在 v-for 时给每项元素绑定事件需要用事件代理吗？为什么?

18.vue 渲染大量数据时应该怎么优化?

如果一次性传入大量数据 建议

1.先打死提供数据的人 ^ v ^~ （推荐）

2.如果打不过自己架设中间层

3.SPA + SSR处理 bigpipe （服务器渲染组件总比浏览器强，毕竟那么多核的CPU也不是花瓶）

4.可以考虑stream（我还没看用过...）

4.增加加载动画提升用户体验

5.同时避免浏览器处理大量的dom（具体看下面）

6.尽量不要再用vue的双向数据绑定了 或者只用部分页面中处理的数据

如果并非一次性传入大量数据 而只是分段加载 但次数特别多 建议

1.异步渲染组件

2.使用vue的v-if 最多显示上中（可视区域）下三屏 避免出现大量的dom节点

3.或者使用分页（体验可能不如上面那么好“无限加载的也要考虑dom过多的情况”）

19.vue 如何优化首页的加载速度？vue 首页白屏是什么问题引起的？如何解决呢

- 1.使用首屏SSR + 跳转SPA方式来优化
- 2.改单页应用为多页应用，需要修改webpack的entry
- 3.改成多页以后使用应该使用prefetch的就使用
- 4.处理加载的时间片，合理安排加载顺序，尽量不要有大面积空隙
- 5.CDN资源还是很重要的，最好分开，也能减少一些不必要的资源损耗
- 6.使用Quicklink，在网速好的时候 可以帮助你预加载页面资源
- 7.骨架屏这种的用户体验的东西一定要上，最好借助stream先将这部分输出给浏览器解析
- 8.合理使用web worker优化一些计算
- 9.缓存一定要使用，但是请注意合理使用
- 10.大概就这么多，最后可以借助一些工具进行性能评测，重点调优，例如使用performance自己实现下等

20.vue 是如何对数组方法进行变异的？例如 push、pop、splice 等方法

Plain Text

```
methodsToPatch.forEach(function(method) {
    // cache original method
    // 获取原方法
    var original = arrayProto[method];
    // def方法重新定义arrayMethods的method方法，然后将新的取值方法赋值
    def(arrayMethods, method, function mutator() {
      var args = [],
        len = arguments.length;
      while (len--) args[len] = arguments[len];

      var result = original.apply(this, args);
      var ob = this.__ob__;
      var inserted;
      switch (method) {
        case 'push':
        case 'unshift':
          // [].push(1),[].unshift(1)
          // arg = [1]
          inserted = args;
          break
        case 'splice':
          // [1,2,3].splice(0,1,1)
          // 第三个参数为插入的值
          inserted = args.slice(2);
          break
      }
      if (inserted) { ob.observeArray(inserted); }
      // 监听变化，如果不是插入操作直接循环响应
      // 如果是去除数组参数方法，触发一次notify将会重新计算
      // 如果仅仅是数字数据，任何操作只需要再次执行一次notify则可以
      // 但是如果新增的是一个对象类型，就需要重新监听
      // 为什么用角标和length属性不能监听的原因是因为无法触发obj的get方法，所以没法动态
监听
      // notify change
      ob.dep.notify();
      return result
    });
  });
```

21.谈一谈 nextTick 的原理

Vue 实现响应式并**不是数据发生变化之后 DOM 立即变化**，而是按一定的策略进行 DOM 的更新。

> Vue 异步执行 DOM 更新。只要观察到数据变化，Vue 将开启一个队列，并缓冲在同一事件循环中发生的所有数据改变。如果同一个 watcher 被多次触发，只会被推入到队列中一次。这种在缓冲时去除重复数据对于避免不必要的计算和 DOM 操作上非常重要。然后，在下一个的事件循环“tick”中，Vue 刷新队列并执行实际 (已去重的) 工作。Vue 在内部尝试对异步队列使用原生的 Promise.then和MessageChannel，如果执行环境不支持，会采用setTimeout(fn, 0)代替。
>
> 例如，当你设置vm.someData = 'new value'，该组件不会立即重新渲染。当刷新队列时，组件会在事件循环队列清空时的下一个“tick”更新。多数情况我们不需要关心这个过程，但是如果你想在 DOM 状态更新后做点什么，这就可能会有些棘手。虽然 Vue.js 通常鼓励开发人员沿着“数据驱动”的方式思考，避免直接接触 DOM，但是有时我们确实要这么做。为了在数据变化之后等待 Vue 完成更新 DOM ，可以在数据变化之后立即使用Vue.nextTick(callback)。这样回调函数在 DOM 更新完成后就会调用。

22.Vue 中的 computed 是如何实现的（腾讯、平安）

实质是一个惰性的watcher，在取值操作时根据自身标记 dirty属性返回上一次计算结果/重新计算值 在创建时就进行一次取值操作，收集依赖变动的对象/属性(将自身压入dep中) 在依赖的对象/属性变动时，仅将自身标记dirty致为true

23.Vue 中的 computed 和 watch 的区别在哪里（虾皮）

computed：计算属性

计算属性是由data中的已知值，得到的一个新值。 这个新值只会根据已知值的变化而变化，其他不相关的数据的变化不会影响该新值。 计算属性不在data中，计算属性新值的相关已知值在data中。 别人变化影响我自己。 watch：监听数据的变化

监听data中数据的变化 监听的数据就是data中的已知值 我的变化影响别人

1.watch擅长处理的场景：一个数据影响多个数据

2.computed擅长处理的场景：一个数据受多个数据影响

24.v-if、v-show、v-html 的原理是什么，它是如何封装的?

v-if会调用addIfCondition方法，生成vnode的时候会忽略对应节点，render的时候就不会渲染； v-show会生成vnode，render的时候也会渲染成真实节点，只是在render过程中会在节点的属性中修改show属性值，也就是常说的display； v-html会先移除节点下的所有节点，调用html方法，通过addProp添加innerHTML属性，归根结底还是设置innerHTML为v-html的值

20.vue中mixin与extend区别

20.vue中mixin与extend区别

20.vue中mixin与extend区别

七 移动端问题

1.说说你知道的移动端web的兼容性bug

1、一些情况下对非可点击元素如(label,span)监听click事件，ios下不会触发，css增加cursor:pointer就搞定了。

2.position 在Safari下的两个定位需要都写，只写一个容易发生错乱

3.Input 的placeholder会出现文本位置偏上的情况

input 的placeholder会出现文本位置偏上的情况：PC端设置line-height等于height能够对齐，而移动端仍然是偏上，解决是设置line-height：normal

4.zepto点击穿透问题

引入fastclick解决；event.preventDefault

5.当输入框在最底部的时候，弹起的虚拟键盘会把输入框挡住。

Element.scrollIntoViewIfNeeded(opt_center)

2.有哪些多屏适配方案

1. media query + rem
2. flex
3. 弹性布局
4. flexiable 整体缩放（动态设置缩放系数的方式， 让layout viewport与设计图对应，极大地 方便了重构，同时也避免了1px的问题）

3.移动端的布局用过媒体查询吗?

假设你现在正用一台显示设备来阅读这篇文章，同时你也想把它投影到屏幕上，或者打印出来， 而显示设备、屏幕投影和打印等这些媒介都有自己的特点，CSS就是为文档提供在不同媒介上展示的适配方法

当媒体查询为真时，相关的样式表或样式规则会按照正常的级联规被应用。 当媒体查询返回假， 标签上带有媒体查询的样式表 仍将被下载 （只不过不会被应用）。

包含了一个媒体类型和至少一个使用 宽度、高度和颜色等媒体属性来限制样式表范围的表达式。
CSS3加入的媒体查询使得无需修改内容便可以使样式应用于某些特定的设备范围。

<style> @media (min-width: 700px) and (orientation: landscape){ .sidebar { display: none; } }
</style>

4.移动端（Android IOS）怎么做好用户体验?

清晰的视觉纵线、

信息的分组、极致的减法、

利用选择代替输入、

标签及文字的排布方式、

依靠明文确认密码、

合理的键盘利用、

5.移动端最小触控区域是多大?

移动端的点击事件的有延迟，时间是多久，为什么会有？ 怎么解决这个延时？（click 有 300ms 延迟, 为了实现safari的双击事件的设计，浏览器要知道你是不是要双击操作。）

八 大厂面试题

1.美团面试题

事件循环

浏览器中, js引擎线程会循环从 任务队列 中读取事件并且执行, 这种运行机制称作 Event Loop (事件循环).

每个浏览器环境，至多有一个event loop。 一个event loop可以有1个或多个task queue(任务队列)

先执行同步的代码，然后js会跑去消息队列中执行异步的代码，异步完成后，再轮到回调函数，然后是去下个事件循环中执行setTimeout

它从script(整体代码)开始第一次循环。之后全局上下文进入函数调用栈。直到调用栈清空(只剩全局)，然后执行所有的micro-task。当所有可执行的micro-task执行完毕之后。循环再次从macro-task开始，找到其中一个任务队列执行完毕，然后再执行所有的micro-task，这样一直循环下去。

从规范上来讲，setTimeout有一个4ms的最短时间，也就是说不管你设定多少，反正最少都要间隔4ms才运行里面的回调。而Promise的异步没有这个问题。Promise所在的那个异步队列优先级要高一些 Promise是异步的，是指他的then()和catch()方法，Promise本身还是同步的 Promise的任务会在当前事件循环末尾中执行，而setTimeout中的任务是在下一次事件循环执行

```
//依次输出 12354
setTimeout(function(){
  console.log(4)
 },0);
new Promise(function(resolve){
  console.log(1)
  for( var i=0 ; i<10000 ; i++ ){
    i===9999 && resolve()
  }
  console.log(2)
}).then(function(){
  console.log(5)
});
console.log(3);
```

xss和csrf

事件捕获的应用

jsx的优点

webpack loader和plugin区别

性能优化

react和vue的区别

vue component和指令的区别

vue组件通信

box-sizing

jsonp缺点，为什么不能用POST

vue-router的实现原理

es6用了哪些新特性

cookie和localStorage区别

git fetch是干嘛的

事件代理和冒泡，捕获

304是干嘛的 具体，405 504又是干嘛的

BFC

其他（自我介绍，为啥离职，为啥从美团离职，git工作流，code review，单元测试）

react组件生命周期

伪类和伪元素的区别 CSS 伪类：逻辑上存在但在文档树中却无须标识的“幽灵”分类 CSS 伪元素（:first-letter，:first-line,:after,:before）代表了某个元素的子元素，这个子元素虽然在逻辑上存在，但却并不实际存在于文档树中。 CSS3标准要求伪元素使用双冒号

em和rem

2.饿了么面试题

1.什么是类数组对象，如何将类数组对象转为真正的数组

拥有length属性和若干索引属性的对象, 类数组只有索引值和长度，没有数组的各种方法，所以如果要类数组调用数组的方法，就需要使用 Array.prototype.method.call 来实现。

```
var arrayLike = {0: 'name', 1: 'age', 2: 'sex', length: 3 }

// 1. slice

Array.prototype.slice.call(arrayLike); // ["name", "age", "sex"]

// 2. splice

Array.prototype.splice.call(arrayLike, 0); // ["name", "age", "sex"]

// 3. ES6 Array.from

Array.from(arrayLike); // ["name", "age", "sex"]
```

// 4. apply

Array.prototype.concat.apply([], arrayLike)

2.跨域

3.伪元素和伪类

伪类用于当已有元素处于的某个状态时，为其添加对应的样式，这个状态是根据用户行为而动态变化的。

a:link

:first-child

:nth-child

:focus

:visited

伪元素代表了某个元素的子元素，这个子元素虽然在逻辑上存在，但却并不实际存在于文档树中。

4.bind返回什么

bind() 方法会返回一个新函数, 又叫绑定函数, 当调用这个绑定函数时, 绑定函数会以创建它时传入 bind() 方法的第一个参数作为当前的上下文, 即this, 传入 bind() 方法的第二个及之后的参数加上绑定函数运行时自身的参数按照顺序作为原函数的参数来调用原函数.

```
var x = 8;
var o = {
 x: 10,
 getX: function(){
  console.log(this.x);
 }
};
var f = o.getX;
f();//8, 由于没有绑定执行时的上下文, this默认指向window, 打印了全局变量x的值
var g = f.bind(o);
```

g();//10, 绑定this后, 成功的打印了o对象的x属性的值.

5.git rebase和git merge的区别

merge操作会生成一个新的节点，之前的提交分开显示。而rebase操作不会生成新的节点，是将两个分支融合成一个线性的提交。

6.箭头函数 箭头函数没有它自己的this值，箭头函数内的this值继承自外围作用域

箭头函数不能用作构造器，不能和new一起使用 箭头函数没有原型属性 yield关键字不能在箭头函数使用 在以下场景中不要使用箭头函数去定义：

定义对象方法、定义原型方法、定义构造函数、定义事件回调函数。

7.== 和===的区别

相等运算符在比较相同类型的数据时，与严格相等运算符完全一样。

在比较不同类型的数据时，相等运算符会先将数据进行类型转换，然后再用严格相等运算符比较。

3.糯米面试题

把面试过程中面试官的问题分享一下：

1. js有几种数据类型。

2. 将一个字符串转化为数字，将数字转化为字符串。

3. 将数字转化为布尔型。

4. 对数组的几种操作方式。

5. 获取当前网址url并从中截取信息。

6. postison的几种属性

7. rem移动端

8. 事件委托原理

9. 闭包原理并应用

10. 原型继承的几种方式

11. jsonp跨域问题，并手写代码

12. 实现斐波那契数列，并手写代码

13. 数组去重，并手写代码

4.2016百度前端面试题

http://www.cnblogs.com/bhaltair/p/6240418.html

九 扩展性问题

1.职业规划&&未来期望

职业规划：

1.首先应该是一个优秀的程序员

2.其次是努力使自己成为某一领域的技术专家

3.通过技术更好的服务于团队和业务

4.提高沟通能力，团队协作，发现问题，解决问题，总结问题能力

5.写写博客，输出就是最好的学习

6.提升个人前端的工作效率和工作质量

7.关注前端前言技术和发展方向，通过新技术服务团队和业务

8.一专多长

想成为优秀的前端工程师，首先在专业技能领域必不可少，其次在团队贡献、业务思索、价值判断上也有要求。这三方面能决定你的专业技能能够为公司产出多大的价值。

我觉得程序员最核心的竞争力是学习力和责任。 学习能力的源泉就是好奇心，也就是对新知识的渴求，以及对探索未知的冲动。

你希望加入一个什么样的团队

1.对前端开发有激情

2.能够持之以恒的学习

3.团队做事方式是否规范（代码规范，安全规范，流程规范）

4.团队有足够的成长空间，对自己有个清晰的定位。

5.团队认可我的价值

最后你有什么要问我的吗

1.可以问一下公司具体的情况，比如我即将加入的部门的主要业务

2.问一下具体工作情况，比如需要做哪些内容

3.公司的氛围和公司的文化

4.贵司对这项职务的工作内容和期望目标

2.原来公司工作流程是怎么样的，如何与其他人协作的？如何跨部门合作的？

3.你遇到过比较难的技术问题是？你是如何解决的？

4.常使用的库有哪些？常用的前端开发工具？开发过什么应用或组件？

5.页面重构怎么操作？

网站重构：在不改变外部行为的前提下，简化结构、添加可读性，而在网站前端保持一致的行为。

也就是说是在不改变UI的情况下，对网站进行优化，在扩展的同时保持一致的UI。

对于传统的网站来说重构通常是：

表格(table)布局改为DIV+CSS

使网站前端兼容于现代浏览器(针对于不合规范的CSS、如对IE6有效的)

对于移动平台的优化

针对于SEO进行优化

深层次的网站重构应该考虑的方面

减少代码间的耦合

让代码保持弹性

严格按规范编写代码

设计可扩展的API

代替旧有的框架、语言(如VB)

增强用户体验

通常来说对于速度的优化也包含在重构中

压缩JS、CSS、image等前端资源(通常是由服务器来解决)

程序的性能优化(如数据读写)

采用CDN来加速资源加载

对于JS DOM的优化

HTTP服务器的文件缓存

6.列举IE与其他浏览器不一样的特性?

1、事件不同之处：

触发事件的元素被认为是目标（target）。而在 IE 中，目标包含在 event 对象的 srcElement 属性；

获取字符代码、如果按键代表一个字符（shift、ctrl、alt除外），IE 的 keyCode 会返回字符代码（Unicode），DOM 中 按键的代码和字符是分离的，要获取字符代码，需要使用 charCode 属性；

阻止某个事件的默认行为，IE 中阻止某个事件的默认行为，必须将 returnValue 属性设置为 false，Mozilla 中，需要调用 preventDefault() 方法；

停止事件冒泡，IE 中阻止事件进一步冒泡，需要设置 cancelBubble 为 true，Mozzilla 中，需要调用 stopPropagation()；

7.什么叫优雅降级和渐进增强?

渐进增强 progressive enhancement：针对低版本浏览器进行构建页面，保证最基本的功能，然后再针对高级浏览器进行效果、交互等改进和追加功能达到更好的用户体验。

优雅降级 graceful degradation：一开始就构建完整的功能，然后再针对低版本浏览器进行兼容。

区别：优雅降级是从复杂的现状开始，并试图减少用户体验的供给，而渐进增强则是从一个非常基础的，能够起作用的版本开始，并不断扩充，以适应未来环境的需要。降级（功能衰减）意味着往回看；而渐进增强则意味着朝前看，同时保证其根基处于安全地带。

“优雅降级”观点

“优雅降级”观点认为应该针对那些最高级、最完善的浏览器来设计网站。而将那些被认为“过时”或有功能缺失的浏览器下的测试工作安排在开发周期的最后阶段，并把测试对象限定为主流浏览器（如 IE、Mozilla 等）的前一个版本。

在这种设计范例下，旧版的浏览器被认为仅能提供“简陋却无妨 (poor, but passable)” 的浏览体验。你可以做一些小的调整来适应某个特定的浏览器。但由于它们并非我们所关注的焦点，因此除了修复较大的错误之外，其它的差异将被直接忽略。

“渐进增强”观点

“渐进增强”观点则认为应关注于内容本身。

内容是我们建立网站的诱因。有的网站展示它，有的则收集它，有的寻求，有的操作，还有的网站甚至会包含以上的种种，但相同点是它们全都涉及到内容。这使得“渐进增强”成为一种更为合理的设计范例。这也是它立即被 Yahoo! 所采纳并用以构建其“分级式浏览器支持 (Graded Browser Support)”策略的原因所在。

8.是否了解公钥加密和私钥加密。

一般情况下是指私钥用于对数据进行签名，公钥用于对签名进行验证;

HTTP网站在浏览器端用公钥加密敏感数据，然后在服务器端再用私钥解密。

9.WEB应用从服务器主动推送Data到客户端有那些方式?

html5提供的Websocket

不可见的iframe

WebSocket通过Flash

XHR长时间连接

XHR Multipart Streaming

<script>标签的长时间连接(可跨域)

10.你有用过哪些前端性能优化的方法?

（1） 减少http请求次数：CSS Sprites, JS、CSS源码压缩、图片大小控制合适；网页Gzip，CDN托管，data缓存 ，图片服务器。

（2） 前端模板 JS+数据，减少由于HTML标签导致的带宽浪费，前端用变量保存AJAX请求结果，每次操作本地变量，不用请求，减少请求次数

（3） 用innerHTML代替DOM操作，减少DOM操作次数，优化javascript性能。

（4） 当需要设置的样式很多时设置className而不是直接操作style。

（5） 少用全局变量、缓存DOM节点查找的结果。减少IO读取操作。

（6） 避免使用CSS Expression（css表达式)又称Dynamic properties(动态属性)。

（7） 图片预加载，将样式表放在顶部，将脚本放在底部 加上时间戳。

（8） 避免在页面的主体布局中使用table，table要等其中的内容完全下载之后才会显示出来，显示比div+css布局慢。

对普通的网站有一个统一的思路，就是尽量向前端优化、减少数据库操作、减少磁盘IO。向前端优化指的是，在不影响功能和体验的情况下，能在浏览器执行的不要在服务端执行，能在缓存服务器上直接返回的不要到应用服务器，程序能直接取得的结果不要到外部取得，本机内能取得的数据不要到远程取，内存能取到的不要到磁盘取，缓存中有的不要去数据库查询。减少数据库操作指减少更新次数、缓存结果减少查询次数、将数据库执行的操作尽可能的让你的程序完成（例如join查询），减少磁盘IO指尽量不使用文件系统作为缓存、减少读写文件次数等。程序优化永远要优化慢的部分，换语言是无法“优化”的。

11.一个页面从输入 URL 到页面加载显示完成，这个过程中都发生了什么？（流程说的越详细越好）

注：这题胜在区分度高，知识点覆盖广，再不懂的人，也能答出几句，

而高手可以根据自己擅长的领域自由发挥，从URL规范、HTTP协议、DNS、CDN、数据库查询、

到浏览器流式解析、CSS规则构建、layout、paint、onload/domready、JS执行、JS API绑定等等；

详细版：

1、浏览器会开启一个线程来处理这个请求，对 URL 分析判断如果是 http 协议就按照 Web 方式来处理;

2、调用浏览器内核中的对应方法，比如 WebView 中的 loadUrl 方法;

3、通过DNS解析获取网址的IP地址，设置 UA 等信息发出第二个GET请求;

4、进行HTTP协议会话，客户端发送报头(请求报头);

5、进入到web服务器上的 Web Server，如 Apache、Tomcat、Node.JS 等服务器;

6、进入部署好的后端应用，如 PHP、Java、JavaScript、Python 等，找到对应的请求处理;

7、处理结束回馈报头，此处如果浏览器访问过，缓存上有对应资源，会与服务器最后修改时间对比，一致则返回304;

8、浏览器开始下载html文档(响应报头，状态码200)，同时使用缓存;

9、文档树建立，根据标记请求所需指定MIME类型的文件（比如css、js）,同时设置了cookie;

10、页面开始渲染DOM，JS根据DOM API操作DOM,执行事件绑定等，页面显示完成。

简洁版：

浏览器根据请求的URL交给DNS域名解析，找到真实IP，向服务器发起请求；

服务器交给后台处理完成后返回数据，浏览器接收文件（HTML、JS、CSS、图象等）；

浏览器对加载到的资源（HTML、JS、CSS等）进行语法解析，建立相应的内部数据结构（如HTML的DOM）；

载入解析到的资源文件，渲染页面，完成。

12.对前端工程师这个职位是怎么样理解的？它的前景会怎么样？

前端是最贴近用户的程序员，比后端、数据库、产品经理、运营、安全都近。

1、实现界面交互

2、提升用户体验

3、有了Node.js，前端可以实现服务端的一些事情

前端是最贴近用户的程序员，前端的能力就是能让产品从 90分进化到 100 分，甚至更好，

参与项目，快速高质量完成实现效果图，精确到1px；

与团队成员，UI设计，产品经理的沟通；

做好的页面结构，页面重构和用户体验；

处理hack，兼容、写出优美的代码格式；

针对服务器的优化、拥抱最新前端技术。

13.你对加班的看法?

加班就像借钱，原则应当是------救急不救穷

14.平时如何管理你的项目?

先期团队必须确定好全局样式（globe.css），编码模式(utf-8) 等；

编写习惯必须一致（例如都是采用继承式的写法，单样式都写成一行）；

标注样式编写人，各模块都及时标注（标注关键样式调用的地方）；

页面进行标注（例如 页面 模块 开始和结束）；

CSS跟HTML 分文件夹并行存放，命名都得统一（例如style.css）；

JS 分文件夹存放 命名以该JS功能为准的英文翻译。

图片采用整合的 images.png png8 格式文件使用 尽量整合在一起使用方便将来的管理

15.你在现在的团队处于什么样的角色，起到了什么明显的作用?

16.前端学习网站推荐

1. 极客标签： http://www.gbtags.com/
2. 码农周刊： http://weekly.manong.io/issues/

3. 前端周刊： http://www.feweekly.com/issues

4. 慕课网： http://www.imooc.com/

5. div.io： http://div.io

6. Hacker News： https://news.ycombinator.com/news

7. InfoQ： http://www.infoq.com/

8. w3cplus： http://www.w3cplus.com/

9. Stack Overflow： http://stackoverflow.com/

10.w3school： http://www.w3school.com.cn/

11.mozilla： https://developer.mozilla.org/zh-CN/docs/Web/JavaScript

17.常使用的库有哪些？常用的前端开发工具？开发过什么应用或组件？

使用率较高的框架有jQuery、AngularJs, ReactJs, YUI、Prototype、Dojo、Ext.js、Mootools等。尤其是jQuery，超过91%。

轻量级框架有Modernizr、underscore.js、backbone.js、Raphael.js等。（理解这些框架的功能、性能、设计原理）

前端开发工具：WebStorm, Hbuilder, Sublime Text 、Eclipse、Notepad、Firebug、HttpWatch、Yslow。

开发过的插件：城市选择插件，汽车型号选择插件、幻灯片插件。弹出层。（写过开源程序，加载器，js引擎更好）

18.平时如何管理你的项目，如何设计突发大规模并发架构？

先期团队必须确定好全局样式（globe.css），编码模式(utf-8) 等

编写习惯必须一致（例如都是采用继承式的写法，单样式都写成一行）；

标注样式编写人，各模块都及时标注（标注关键样式调用的地方）；

页面进行标注（例如 页面 模块 开始和结束）；

CSS跟HTML 分文件夹并行存放，命名都得统一（例如style.css）

JS 分文件夹存放 命民以该JS 功能为准英文翻译；

图片采用整合的 images.png png8 格式文件使用 尽量整合在一起使用方便将来的管理

19.那些操作会造成内存泄漏?

先期团队必须确定好全局样式（globe.css），编码模式(utf-8) 等

编写习惯必须一致（例如都是采用继承式的写法，单样式都写成一行）；

标注样式编写人，各模块都及时标注（标注关键样式调用的地方）；

页面进行标注（例如 页面 模块 开始和结束）；

CSS跟HTML 分文件夹并行存放，命名都得统一（例如style.css）

JS 分文件夹存放 命民以该JS 功能为准英文翻译;

图片采用整合的 images.png png8 格式文件使用 尽量整合在一起使用方便将来的管理

20.你说你热爱前端，那么应该WEB行业的发展很关注吧？ 说说最近最流行的一些东西吧?

Node.js、Mongodb、npm、react、angularjs、MVVM、MEAN

十 网络知识相关

1.http状态码

简单版

[

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。

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。

503 Service Unavailable 服务器端暂时无法处理请求（可能是过载或维护）。

]

完整版

1**(信息类)：表示接收到请求并且继续处理

100——客户必须继续发出请求

101——客户要求服务器根据请求转换HTTP协议版本

2(响应成功)：表示动作被成功接收、理解和接受**

200——表明该请求被成功地完成，所请求的资源发送回客户端

201——提示知道新文件的URL

202——接受和处理、但处理未完成

203——返回信息不确定或不完整

204——请求收到，但返回信息为空

205——服务器完成了请求，用户代理必须复位当前已经浏览过的文件

206——服务器已经完成了部分用户的GET请求

3(重定向类)：为了完成指定的动作，必须接受进一步处理**

300——请求的资源可在多处得到

301——本网页被永久性转移到另一个URL

302——请求的网页被转移到一个新的地址，但客户访问仍继续通过原始URL地址，重定向，新的URL会在response中的Location中返回，浏览器将会使用新的URL发出新的Request。

303——建议客户访问其他URL或访问方式

304——自从上次请求后，请求的网页未修改过，服务器返回此响应时，不会返回网页内容，代表上次的文档已经被缓存了，还可以继续使用

305——请求的资源必须从服务器指定的地址得到

306——前一版本HTTP中使用的代码，现行版本中不再使用

307——申明请求的资源临时性删除

4(客户端错误类)：请求包含错误语法或不能正确执行**

400——客户端请求有语法错误，不能被服务器所理解

401——请求未经授权，这个状态代码必须和WWW-Authenticate报头域一起使用

HTTP 401.1 - 未授权：登录失败

HTTP 401.2 - 未授权：服务器配置问题导致登录失败

HTTP 401.3 - ACL 禁止访问资源

HTTP 401.4 - 未授权：授权被筛选器拒绝

HTTP 401.5 - 未授权：ISAPI 或 CGI 授权失败

402——保留有效ChargeTo头响应

403——禁止访问，服务器收到请求，但是拒绝提供服务

HTTP 403.1 禁止访问：禁止可执行访问

HTTP 403.2 - 禁止访问：禁止读访问

HTTP 403.3 - 禁止访问：禁止写访问

HTTP 403.4 - 禁止访问：要求 SSL

HTTP 403.5 - 禁止访问：要求 SSL 128

HTTP 403.6 - 禁止访问：IP 地址被拒绝

HTTP 403.7 - 禁止访问：要求客户证书

HTTP 403.8 - 禁止访问：禁止站点访问

HTTP 403.9 - 禁止访问：连接的用户过多

HTTP 403.10 - 禁止访问：配置无效

HTTP 403.11 - 禁止访问：密码更改

HTTP 403.12 - 禁止访问：映射器拒绝访问

HTTP 403.13 - 禁止访问：客户证书已被吊销

HTTP 403.15 - 禁止访问：客户访问许可过多

HTTP 403.16 - 禁止访问：客户证书不可信或者无效

HTTP 403.17 - 禁止访问：客户证书已经到期或者尚未生效

404——一个404错误表明可连接服务器，但服务器无法取得所请求的网页，请求资源不存在。eg：输入了错误的URL

405——用户在Request-Line字段定义的方法不允许

406——根据用户发送的Accept拖，请求资源不可访问

407——类似401，用户必须首先在代理服务器上得到授权

408——客户端没有在用户指定的饿时间内完成请求

409——对当前资源状态，请求不能完成

410——服务器上不再有此资源且无进一步的参考地址

411——服务器拒绝用户定义的Content-Length属性请求

412——一个或多个请求头字段在当前请求中错误

413——请求的资源大于服务器允许的大小

414——请求的资源URL长于服务器允许的长度

415——请求资源不支持请求项目格式

416——请求中包含Range请求头字段，在当前请求资源范围内没有range指示值，请求也不包含If-Range请求头字段

417——服务器不满足请求Expect头字段指定的期望值，如果是代理服务器，可能是下一级服务器不能满足请求长。

5(服务端错误类)：服务器不能正确执行一个正确的请求**

HTTP 500 - 服务器遇到错误，无法完成请求

HTTP 500.100 - 内部服务器错误 - ASP 错误

HTTP 500-11 服务器关闭

HTTP 500-12 应用程序重新启动

HTTP 500-13 - 服务器太忙

HTTP 500-14 - 应用程序无效

HTTP 500-15 - 不允许请求 global.asa

Error 501 - 未实现

HTTP 502 - 网关错误

HTTP 503：由于超载或停机维护，服务器目前无法使用，一段时间后可能恢复正常

2.http报头有哪些

请求头：

Accept

Cache-control

Host

User-agent

Accenp-Language

响应头：

Cache-Control:max-age 避免了服务端和客户端时间不一致的问题。

content-type

Date

Expires

Last-Modified 标记此文件在服务期端最后被修改的时间

httpOnly 设置cookie是否能通过 js 去访问。在客户端是不能通过js代码去设置一个httpOnly类型的cookie的，这种类型的cookie只能通过服务端来设置。在发生跨域时，cookie 作为一种 credential 信息是不会被传送到服务端的。必须要进行额外设置才可以。

3.代理和反向代理

正向代理： 用浏览器访问时，被残忍的block，于是你可以在国外搭建一台代理服务器，让代理帮我去请求google.com，代理把请求返回的相应结构再返回给我。

反向代理：反向代理服务器会帮我们把请求转发到真实的服务器那里去。Nginx就是性能非常好的反向代理服务器，用来做负载均衡。 正向代理的对象是客户端，反向代理的对象是服务端

4.网络优化/性能优化

使用CDN，让用户访问最近的资源，减少来回传输时间 合并压缩CSS、js、图片、静态资源，服务器开启GZIP css放顶部，js放底部（css可以并行下载，而js加载之后会造成阻塞） 图片预加载和首屏图片之外的做懒加载 做HTTP缓存（添加Expires头和配置Etag）用户可以重复使用本地缓存，减少对服务器压力 大小超过 10KB 的 css/img 建议外联引用，以细化缓存粒度 小于 10k 的图片 base64 DNS 预解析 DNS-Prefetch 预连接 Preconnect

1.代码层面优化

少用全局变量，减少作用域链查找，缓存DOM查找结果，避免使用with（with会创建自己的作用域，会增加作用域链长度）；多个变量声明合并；减少DOM操作次数；尽量避免在HTML标签中写style属性

避免使用css3渐变阴影效果，尽量使用css3动画，开启硬件加速，不滥用float；避免使用CSS表达式；使用<link>来代替@import

2.使用图片预加载技术减少资源请求

5.首屏优化

总结以下可以通过以下三步解决

分拆打包 现在流行的路由库如 react-router 对分拆打包都有很好的支持。可以按照页面对包进行分拆，并在页面切换时加上一些 loading 和 transition 效果。

1.首屏内容最好做到静态缓存 2.首屏内联css渲染 3.图片懒加载 4.服务端渲染，首屏渲染速度更快（重点），无需等待js文件下载执行的过程 5.交互优化（使用加载占位器，在白屏无法避免的时候，为了解决等待加载过程中白屏或者界面闪烁） 6.图片尺寸大小控制

前端渲染的优势

1. 局部刷新。无需每次都进行完整页面请求

2. 懒加载。如在页面初始时只加载可视区域内的数据，滚动后rp加载其它数据，可以通过

3. react-lazyload 实现

4. 富交互。使用 JS 实现各种酷炫效果

5. 节约服务器成本。省电省钱，JS 支持 CDN 部署，且部署极其简单，只需要服务器支持静态文件即可

6. 天生的关注分离设计。服务器来访问数据库提供接口，JS 只关注数据获取和展现

7. JS 一次学习，到处使用。可以用来开发 Web、Serve、Mobile、Desktop 类型的应用

服务端渲染的优势

1. 更好的 SEO，由于搜索引擎爬虫抓取工具可以直接查看完全渲染的页面。

2. 服务端渲染不需要先下载一堆 js 和 css 后才能看到页面（首屏性能）

3. 服务端渲染不用关心浏览器兼容性问题（随意浏览器发展，这个优点逐渐消失）

4. 对于电量不给力的手机或平板，减少在客户端的电量消耗很重要

6.网页验证码是干嘛的，是为了解决什么安全问题。

区分用户是计算机还是人的公共全自动程序。可以防止恶意破解密码、刷票、论坛灌水；

有效防止黑客对某一个特定注册用户用特定程序暴力破解方式进行不断的登陆尝试。

7.为什么利用多个域名来存储网站资源会更有效?

CDN缓存更方便

突破浏览器并发限制

节约cookie带宽

节约主域名的连接数，优化页面响应速度

防止不必要的安全问题

8.请谈一下你对网页标准和标准制定机构重要性的理解。

网页标准和标准制定机构都是为了能让web发展的更‘健康’，开发者遵循统一的标准，降低开发难度，开发成本，SEO也会更好做，也不会因为滥用代码导致各种BUG、安全问题，最终提高网站易用性。

9.前端开发的优化问题。

（1） 减少http请求次数：CSS Sprites, JS、CSS源码压缩、图片大小控制合适；网页Gzip，CDN托管，data缓存 ，图片服务器。

（2） 前端模板 JS+数据，减少由于HTML标签导致的带宽浪费，前端用变量保存AJAX请求结果，每次操作本地变量，不用请求，减少请求次数

（3） 用innerHTML代替DOM操作，减少DOM操作次数，优化javascript性能。

（4） 当需要设置的样式很多时设置className而不是直接操作style。

（5） 少用全局变量、缓存DOM节点查找的结果。减少IO读取操作。

（6） 避免使用CSS Expression（css表达式)又称Dynamic properties(动态属性)。

（7） 图片预加载，将样式表放在顶部，将脚本放在底部 加上时间戳。

（8） 避免在页面的主体布局中使用table，table要等其中的内容完全下载之后才会显示出来，显示比div+css布局慢。

10.简单讲解一下http2的多路复用

在 HTTP/1 中，每次请求都会建立一次HTTP连接，也就是我们常说的3次握手4次挥手，这个过程在一次请求过程中占用了相当长的时间，即使开启了 Keep-Alive ，解决了多次连接的问题，但是依然有两个效率上的问题：

- 第一个：串行的文件传输。当请求a文件时，b文件只能等待，等待a连接到服务器、服务器处理文件、服务器返回文件，这三个步骤。我们假设这三步用时都是1秒，那么a文件用时为3秒，b文件传输完成用时为6秒，依此类推。（注：此项计算有一个前提条件，就是浏览器和服务器是单通道传输）
- 第二个：连接数过多。我们假设Apache设置了最大并发数为300，因为浏览器限制，浏览器发起的最大请求数为6，也就是服务器能承载的最高并发为50，当第51个人访问时，就需要等待前面某个

请求处理完成。

HTTP/2的多路复用就是为了解决上述的两个性能问题。 在 HTTP/2 中，有两个非常重要的概念，分别是帧（frame）和流（stream）。 帧代表着最小的数据单位，每个帧会标识出该帧属于哪个流，流也就是多个帧组成的数据流。 多路复用，就是在一个 TCP 连接中可以存在多条流。换句话说，也就是可以发送多个请求，对端可以通过帧中的标识知道属于哪个请求。通过这个技术，可以避免 HTTP 旧版本中的队头阻塞问题，极大的提高传输性能

简单来说， 就是在同一个TCP连接，同一时刻可以传输多个HTTP请求。

简单版回答：

HTTP/2 复用 TCP 连接，在一个连接里，客户端和浏览器都可以同时发送多个请求或回应，而且不用按照顺序一一对应。

举例来说，在一个TCP连接里面，服务器同时收到了A请求和B请求，于是先回应A请求，结果发现处理过程非常耗时，于是就发送A请求已经处理好的部分， 接着回应B请求，完成后，再发送A请求剩下的部分。

历史原因解释：

1、HTTP/1.0 版本

该版本主要缺点是，每个TCP连接只能发送一个请求。发送数据完毕，连接就关闭，如果还要请求其他资源，就必须再新建一个连接。为了解决这个问题，需要使用`Connection: keep-alive`这个字段。

2、HTTP/1.1 版本

该版本引入了持久连接（persistent connection），即 TCP 连接默认不关闭，可以被多个请求复用，不用声明 Connection: keep-alive。还引入了管道机制（pipelining），即在同一个TCP连接里面，客户端可以同时发送多个请求。这样就进一步改进了HTTP协议的效率。

虽然1.1版允许复用TCP连接，但是同一个TCP连接里面，所有的数据通信是按次序进行的。服务器只有处理完一个回应，才会进行下一个回应。要是前面的回应特别慢，后面就会有许多请求排队等着。这称为"队头堵塞"（Head-of-line blocking）。

http/1中的每个请求都会建立一个单独的连接，除了在每次建立连接过程中的三次握手之外，还存在TCP的慢启动导致的传输速度低。其实大部分的http请求传送的数据都很小，就导致每一次请求基本上都没有达到正常的传输速度。

在http1.1中默认开启keep-alive，解决了上面说到的问题，但是http的传输形式是一问一答的形式，一个请求对应一个响应（http2中已经不成立，一个请求可以有多个响应，server push），在keep-alive中，必须等下上一个请求接受才能发起下一个请求，所以会收到前面请求的阻塞。

使用pipe-line可以连续发送一组没有相互依赖的请求而不比等到上一个请求先结束，看似pipe-line是个好东西，但是到目前为止我还没见过这种类型的连接，也间接说明这东西比较鸡肋。pipe-line依然

没有解决阻塞的问题，因为请求响应的顺序必须和请求发送的顺序一致，如果中间有某个响应花了很长的时间，后面的响应就算已经完成了也要排队等阻塞的请求返回，这就是线头阻塞。

http2的多路复用就很好的解决了上面所提出的问题。http2的传输是基于二进制帧的。每一个TCP连接中承载了多个双向流通的流，每一个流都有一个独一无二的标识和优先级，而流就是由二进制帧组成的。二进制帧的头部信息会标识自己属于哪一个流，所以这些帧是可以交错传输，然后在接收端通过帧头的信息组装成完整的数据。这样就解决了线头阻塞的问题，同时也提高了网络速度的利用率。

http/1.0：如需要发送多个请求必须创建多个 TCP 连接，并且浏览器对于单域名请求有数量限制（一般6个），其连接无法被复用

http/1.1：引入流水线（Pipelining）技术，但先天 FIFO（先进先出）机制导致当前请求的执行依赖于上一个请求执行的完成，容易引起报头阻塞，并没有从根本上解决问题

http/2：重新定义底层 http 语义映射，允许同一个连接上使用请求和响应双向数据流。同一域名只需占用一个 TCP 连接，通过数据流（Stream）以帧为基本协议单位，从根本上解决了问题，避免了因频繁创建连接产生的延迟，减少了内存消耗，提升了使用性能

11.谈谈你对TCP三次握手和四次挥手的理解

三次握手之所以是三次是保证client和server均让对方知道自己的接收和发送能力没问题而保证的最小次数。

第一次client => server 只能server判断出client具备发送能力

第二次 server => client client就可以判断出server具备发送和接受能力。此时client还需让server知道自己接收能力没问题于是就有了第三次 第三次 client => server 双方均保证了自己的接收和发送能力没有问题

其中，为了保证后续的握手是为了应答上一个握手，每次握手都会带一个标识 seq，后续的ACK都会对这个seq进行加一来进行确认

一、三次握手讲解

1. 客户端发送位码为syn＝1,随机产生seq number=1234567的数据包到服务器，服务器由SYN=1知道客户端要求建立联机（客户端：我要连接你）
2. 服务器收到请求后要确认联机信息，向A发送ack number=(客户端的seq+1),syn=1,ack=1,随机产生seq=7654321的包（服务器：好的，你来连吧）
3. 客户端收到后检查ack number是否正确，即第一次发送的seq number+1,以及位码ack是否为1，若正确，客户端会再发送ack number=(服务器的seq+1),ack=1，服务器收到后确认seq值与ack=1则连接建立成功。（客户端：好的，我来了）

二、为什么http建立连接需要三次握手，不是两次或四次?

答：三次是最少的安全次数，两次不安全，四次浪费资源；

三、TCP关闭连接过程

4. Client向Server发送FIN包，表示Client主动要关闭连接，然后进入FIN_WAIT_1状态，等待Server返回ACK包。此后Client不能再向Server发送数据，但能读取数据。
5. Server收到FIN包后向Client发送ACK包，然后进入CLOSE_WAIT状态，此后Server不能再读取数据，但可以继续向Client发送数据。
6. Client收到Server返回的ACK包后进入FIN_WAIT_2状态，等待Server发送FIN包。
7. Server完成数据的发送后，将FIN包发送给Client，然后进入LAST_ACK状态，等待Client返回ACK包，此后Server既不能读取数据，也不能发送数据。
8. Client收到FIN包后向Server发送ACK包，然后进入TIME_WAIT状态，接着等待足够长的时间（2MSL）以确保Server接收到ACK包，最后回到CLOSED状态，释放网络资源。
9. Server收到Client返回的ACK包后便回到CLOSED状态，释放网络资源。

四、为什么要四次挥手?

TCP是全双工信道，何为全双工就是客户端与服务端建立两条通道，通道1:客户端的输出连接服务端的输入；通道2:客户端的输入连接服务端的输出。两个通道可以同时工作：客户端向服务端发送信号的同时服务端也可以向客户端发送信号。所以关闭双通道的时候就是这样：

客户端：我要关闭输入通道了。 服务端：好的，你关闭吧，我这边也关闭这个通道。

服务端：我也要关闭输入通道了。 客户端：好的你关闭吧，我也把这个通道关闭。

12.A、B 机器正常连接后，B 机器突然重启，问 A 此时处于 TCP 什么状态

如果A 与 B 建立了正常连接后，从未相互发过数据，这个时候 B 突然机器重启，问 A 此时处于 TCP 什么状态？如何消除服务器程序中的这个状态?

结论

A侧的TCP链路状态在未发送任何数据的情况下与等待的时间相关，如果在多个超时值范围以内那么状态为<**established**>;如果触发了某一个超时的情况那么视情况的不同会有不同的改变。

一般情况下不管是KeepAlive超时还是内核超时，只要出现超时，那么必然会抛出异常，只是这个异常截获的时机会因编码方式的差异而有所不同。（同步异步IO，以及有无使用select、poll、epoll等IO多路复用机制）

原因与相关细节

<大前提>

基于IP网络的无状态特征，A侧系统不会在无动作情况下收到任何通知获知到B侧掉线的情况(除非AB是直连状态，那么A可以获知到自己网卡掉线的异常)

在此大前提的基础上，会因为链路环境、SOCKET设定、以及内核相关配置的不同，A侧会在不同的时机获知到B侧无响应的结果，但总归是以异常的形式获得这个结果。

<关于内核对待无数据传递SOCKET的方式>

操作系统有一堆时间超级长的兜底用timeout参数，用于在不同的时候给TCP栈一个异常退出的机会，避免无效连接过多而耗尽系统资源

其中，**<TCP KeepAive>**特性能让应用层配置一个远小于内核timeout参数的值，用于在这一堆时间超长的兜底参数生效之前，判断链路是否为有效状态。

<关于超时的各个节点>

以下仅讨论三次握手成功之后的兜底情况

TCP链路在建立之后，内核会初始化一个由**<nf_conntrack_tcp_timeout_established>**参数控制的计时器（这个计时器在Ubuntu 18.04里面长达5天），以防止在未开启TCP KeepAlive的情况下连接因各种原因导致的长时间无动作而过度消耗系统资源，这个计时器会在每次TCP链路活动后重置

TCP正常传输过程中，每一次数据发送之后，必然伴随对端的ACK确认信息。如果对端因为各种原因失去反应（网络链路中断、意外掉电等）这个ACK将永远不会到来，内核在每次发送之后都会重置一个由**<nf_conntrack_tcp_timeout_unacknowledged>**参数控制的计时器，以防止对端以外断网导致的资源过度消耗。（这个计时器在Ubuntu 18.04里面是300秒/5分钟）

以上两个计时器作为keepalive参数未指定情况下的兜底参数，为内核自保特性，所以事件都很长，建议实际开发与运维中用更为合理的参数覆盖这些数值

<关于链路异常后发生的操作>

A侧在超时退出之后一般会发送一个RST包用于告知对端重置链路，并给应用层一个异常的状态信息，视乎同步IO与异步IO的差异，这个异常获知的时机会有所不同。

B侧重启之后，因为不存有之前A-B之间建立链路相关的信息，这时候收到任何A侧来的数据都会以RST作为响应，以告知A侧链路发生异常

RST的设计用意在于链路发生意料之外的故障时告知链路上的各方释放资源（一般指的是NAT网关与收发两端）;FIN的设计是用于在链路正常情况下的正常单向终止与结束。二者不可混淆。

<关于阻塞>

应用层到底层网卡发送的过程中，数据包会经历多个缓冲区，也会经历一到多次的分片操作，阻塞这一结果的发生是具有从底向上传递的特性。

这一过程中有一个需要强调的关键点：**socket.send**这个操作只是把数据发送到了内核缓冲区，只要数据量不大那么这个调用必然是在拷贝完之后立即返回的。而数据量大的时候，必然会产生阻塞。

在TCP传输中，决定阻塞与否的最终节点，是TCP的可靠传输特性。此特性决定了必须要有ACK数据包回复响应正确接收的数据段范围，内核才会把对应的数据从TCP发送缓冲区中移除，腾出空间让新的数据可以写入进来。

这个过程意味着，只要应用层发送了大于内核缓冲区可容容纳的数据量，那么必然会在应用层出现阻塞，等待ACK的到来，然后把新数据压入缓冲队列，循环往复，直到数据发送完毕。

13.介绍 HTTPS 握手过程

1. 客户端使用https的url访问web服务器,要求与服务器建立ssl连接
2. web服务器收到客户端请求后, 会将网站的证书(包含公钥)传送一份给客户端
3. 客户端收到网站证书后会检查证书的颁发机构以及过期时间, 如果没有问题就随机产生一个秘钥
4. 客户端利用公钥将会话秘钥加密, 并传送给服务端, 服务端利用自己的私钥解密出会话秘钥
5. 之后服务器与客户端使用秘钥加密传输

14.HTTPS 握手过程中，客户端如何验证证书的合法性

1、首先什么是HTTP协议? http协议是超文本传输协议，位于tcp/ip四层模型中的应用层；通过请求/响应的方式在客户端和服务器之间进行通信；但是缺少安全性，http协议信息传输是通过明文的方式传输，不做任何加密，相当于在网络上裸奔；容易被中间人恶意篡改，这种行为叫做中间人攻击；

2、加密通信： 为了安全性，双方可以使用对称加密的方式key进行信息交流，但是这种方式对称加密秘钥也会被拦截，也不够安全，进而还是存在被中间人攻击风险； 于是人们又想出来另外一种方式，使用非对称加密的方式；使用公钥/私钥加解密；通信方A发起通信并携带自己的公钥，接收方B通过公钥来加密对称秘钥；然后发送给发起方A；A通过私钥解密；双发接下来通过对称秘钥来进行加密通信；但是这种方式还是会存在一种安全性；中间人虽然不知道发起方A的私钥，但是可以做到偷天换日，将拦截发起方的公钥key;并将自己生成的一对公/私钥的公钥发送给B；接收方B并不知道公钥已经被偷偷换过；按照之前的流程，B通过公钥加密自己生成的对称加密秘钥key2;发送给A； 这次通信再次被中间人拦截，尽管后面的通信，两者还是用key2通信，但是中间人已经掌握了Key2;可以进行轻松的加解密；还是存在被中间人攻击风险；

3、解决困境：权威的证书颁发机构CA来解决； 3.1制作证书：作为服务端的A，首先把自己的公钥key1发给证书颁发机构，向证书颁发机构进行申请证书；证书颁发机构有一套自己的公私钥，CA通过自己的私钥来加密key1,并且通过服务端网址等信息生成一个证书签名，证书签名同样使用机构的私钥进行加密；制作完成后，机构将证书发给A； 3.2校验证书真伪：当B向服务端A发起请求通信的时候，A不再直接返回自己的公钥，而是返回一个证书； 说明：各大浏览器和操作系统已经维护了所有的权威证书机构的名称和公钥。B只需要知道是哪个权威机构发的证书，使用对应的机构公钥，就可以解密出证书签名；接下来，B使用同样的规则，生成自己的证书签名，如果两个签名是一致的，说明证书是有效的； 签名验证成功后，B就可以再次利用机构的公钥，解密出A的公钥key1;接下来的操作，就是和之前一样的流程了； 3.3：中间人是否会拦截发送假证书到B呢？ 因为证书的签名是由服务器端网址等信息生成的，并且通过第三方机构的私钥加密中间人无法篡改； 所以最关键的问题是证书签名的真伪；

4、https主要的思想是在http基础上增加了ssl安全层，即以上认证过程；

15.介绍下 HTTPS 中间人攻击

中间人攻击过程如下：

1. 服务器向客户端发送公钥。
2. 攻击者截获公钥，保留在自己手上。
3. 然后攻击者自己生成一个【伪造的】公钥，发给客户端。
4. 客户端收到伪造的公钥后，生成加密hash值发给服务器。
5. 攻击者获得加密hash值，用自己的私钥解密获得真秘钥。
6. 同时生成假的加密hash值，发给服务器。
7. 服务器用私钥解密获得假秘钥。
8. 服务器用加秘钥加密传输信息

防范方法：

服务端在发送浏览器的公钥中加入CA证书，浏览器可以验证CA证书的有效性

16.介绍下 http1.0、1.1、2.0 协议的区别?

http1.0特性

- 无状态：服务器不跟踪不记录请求过的状态
- 无连接：浏览器每次请求都需要建立tcp连接

无状态

对于无状态的特性可以借助cookie/session机制来做身份认证和状态记录

无连接

无连接导致的性能缺陷有两种：

1. 无法复用连接

每次发送请求，都需要进行一次tcp连接（即3次握手4次挥手），使得网络的利用率非常低

2. 队头阻塞

http1.0规定在前一个请求响应到达之后下一个请求才能发送，如果前一个阻塞，后面的请求也给阻塞的

http1.1特性

为了解决http1.0的性能缺陷，http1.1出现了

http1.1特性：

- 长连接：新增Connection字段，可以设置keep-alive值保持连接不断开

- 管道化：基于上面长连接的基础，管道化可以不等第一个请求响应继续发送后面的请求，但响应的顺序还是按照请求的顺序返回
- 缓存处理：新增字段cache-control
- 断点传输

长连接

http1.1默认保持长连接，数据传输完成保持tcp连接不断开,继续用这个通道传输数据

管道化

基于长连接的基础，我们先看没有管道化请求响应：

tcp没有断开，用的同一个通道

```
Plain Text
1  请求1 > 响应1 --> 请求2 > 响应2 --> 请求3 > 响应3
```

管道化的请求响应：

```
Plain Text
1  请求1 --> 请求2 --> 请求3 > 响应1 --> 响应2 --> 响应3
```

即使服务器先准备好响应2,也是按照请求顺序先返回响应1

虽然管道化，可以一次发送多个请求，但是响应仍是顺序返回，仍然无法解决队头阻塞的问题

缓存处理

当浏览器请求资源时，先看是否有缓存的资源，如果有缓存，直接取，不会再发请求，如果没有缓存，则发送请求

通过设置字段cache-control来控制

断点传输

在上传/下载资源时，如果资源过大，将其分割为多个部分，分别上传/下载，如果遇到网络故障，可以从已经上传/下载好的地方继续请求，不用从头开始，提高效率

在 Header 里两个参数实现的，客户端发请求时对应的是 Range 服务器端响应时对应的是 Content-Range

http2.0特性

- 二进制分帧
- 多路复用： 在共享TCP链接的基础上同时发送请求和响应

- 头部压缩
- 服务器推送：服务器可以额外的向客户端推送资源，而无需客户端明确的请求

二进制分帧

将所有传输的信息分割为更小的消息和帧,并对它们采用二进制格式的编码

多路复用

基于二进制分帧，在同一域名下所有访问都是从同一个tcp连接中走，http消息被分解为独立的帧，乱序发送，服务端根据标识符和首部将消息重新组装起来

区别

1. http1.0 到http1.1的主要区别，就是从无连接到长连接
2. http2.0对比1.X版本主要区别就是多路复用

17.永久性重定向（301）和临时性重定向（302）对 SEO 有什么影响

1）301 redirect——301代表永久性转移(Permanently Moved)，301重定向是网页更改地址后对搜索引擎友好的最好方法，只要不是暂时搬移的情况,都建议使用301来做转址。 如果我们把一个地址采用301跳转方式跳转的话，搜索引擎会把老地址的PageRank等信息带到新地址，同时在搜索引擎索引库中彻底废弃掉原先的老地址。旧网址的排名等完全清零

2）302 redirect——302代表暂时性转移(Temporarily Moved)，在前些年，不少Black Hat SEO曾广泛应用这项技术作弊，目前，各大主要搜索引擎均加强了打击力度，象Google前些年对Business.com以及近来对BMW德国网站的惩罚。即使网站客观上不是spam，也很容易被搜寻引擎容易误判为spam而遭到惩罚

18.Http 状态码 301 和 302 的应用场景分别是什么

- 301应用场景: 域名到期不想继续用这个,换了地址
- 302应用场景: 做活动时候,从首页跳到活动页面

19.接口如何防刷

1：网关控制流量洪峰，对在一个时间段内出现流量异常，可以拒绝请求（参考个人博客文章 https://mp.csdn.net/postedit/81672222）

2：源 `ip` 请求个数限制。对请求来源的 `ip` 请求个数做限制

3： `http` 请求头信息校验；（例如 `host` ， `User-Agent` ， `Referer` ）

4：对用户唯一身份uid进行限制和校验。例如基本的长度，组合方式，甚至有效性进行判断。或者uid具有一定的时效性

5：前后端协议采用二进制方式进行交互或者协议采用签名机制

6：人机验证，验证码，短信验证码，滑动图片形式，12306形式

20.为什么 HTTP1.1 不能实现多路复用

HTTP/1.1 不是二进制传输，而是通过文本进行传输。由于没有流的概念，在使用并行传输（多路复用）传递数据时，接收端在接收到响应后，并不能区分多个响应分别对应的请求，所以无法将多个响应的结果重新进行组装，也就实现不了多路复用。

HTTP1.x是序列和阻塞机制

HTTP 2.0 是多工复用TCP连接，在一个连接里，客户端和浏览器都可以同时发送多个请求或回应，而且不用按照顺序一一对应，这样就避免了"队头堵塞"。

- 举例来说，在一个TCP连接里面，服务器同时收到了A请求和B请求，于是先回应A请求，结果发现处理过程非常耗时，于是就发送A请求已经处理好的部分， 接着回应B请求，完成后，再发送A请求剩下的部分。
- 旧的http1.1是会等 A请求完全处理完后在 处理B请求，会阻塞
- 另：http1.1已经实现了管道机制：即 在同一个TCP连接里面，客户端可以同时发送多个请求。http 1.0并做不到，所以效率很低

十一 浏览器

1.介绍下重绘和回流（Repaint & Reflow），以及如何进行优化

（1）浏览器渲染机制

- 浏览器采用流式布局模型（Flow Based Layout）
- 浏览器会把HTML解析成DOM，把CSS解析成CSSOM，DOM和CSSOM合并就产生了渲染树（Render Tree）。
- 有了RenderTree，我们就知道了所有节点的样式，然后计算他们在页面上的大小和位置，最后把节点绘制到页面上。
- 由于浏览器使用流式布局，对Render Tree的计算通常只需要遍历一次就可以完成，但table及其内部元素除外，他们可能需要多次计算，通常要花3倍于同等元素的时间，这也是为什么要避免使用table布局的原因之一。

（2）重绘

由于节点的几何属性发生改变或者由于样式发生改变而不会影响布局的，称为重绘，例如outline, visibility, color、background-color等，重绘的代价是高昂的，因为浏览器必须验证DOM树上其他节点元素的可见性。

（3）回流

回流是布局或者几何属性需要改变就称为回流。回流是影响浏览器性能的关键因素，因为其变化涉及到部分页面（或是整个页面）的布局更新。一个元素的回流可能会导致了其所有子元素以及DOM中紧随其后的节点、祖先节点元素的随后的回流。

JavaScript

```
<body>
<div class="error">
    <h4>我的组件</h4>
    <p><strong>错误：</strong>错误的描述…</p>
    <h5>错误纠正</h5>
    <ol>
        <li>第一步</li>
        <li>第二步</li>
    </ol>
</div>
</body>
```

在上面的HTML片段中，对该段落(<p>标签)回流将会引发强烈的回流，因为它是一个子节点。这也导致了祖先的回流（div.error和body – 视浏览器而定）。此外，<h5>和<ol>也会有简单的回流，因为其在DOM中在回流元素之后。大部分的回流将导致页面的重新渲染。

回流必定会发生重绘，重绘不一定会引发回流。

（4）浏览器优化

现代浏览器大多都是通过队列机制来批量更新布局，浏览器会把修改操作放在队列中，至少一个浏览器刷新（即16.6ms）才会清空队列，但当你获取布局信息的时候，队列中可能有会影响这些属性或方法返回值的操作，即使没有，浏览器也会强制清空队列，触发回流与重绘来确保返回正确的值。

主要包括以下属性或方法：

JavaScript

```
offsetTop、offsetLeft、offsetWidth、offsetHeight
scrollTop、scrollLeft、scrollWidth、scrollHeight
clientTop、clientLeft、clientWidth、clientHeight
width、height
getComputedStyle()
getBoundingClientRect()
```

（5）减少重绘与回流

①CSS

- 使用 transform 替代 top
- 使用 visibility 替换 display: none ，因为前者只会引起重绘，后者会引发回流（改变了布局
- 避免使用table布局，可能很小的一个小改动会造成整个 table 的重新布局。

- 尽可能在DOM树的最末端改变class，回流是不可避免的，但可以减少其影响。尽可能在DOM树的最末端改变class，可以限制了回流的范围，使其影响尽可能少的节点。
- 避免设置多层内联样式，CSS 选择符从右往左匹配查找，避免节点层级过多。

JavaScript

```
<div>
  <a> <span></span> </a>
</div>
<style>
  span {
    color: red;
  }
  div > a > span {
    color: red;
  }
</style>
```

对于第一种设置样式的方式来说，浏览器只需要找到页面中所有的 span 标签然后设置颜色，但是对于第二种设置样式的方式来说，浏览器首先需要找到所有的 span 标签，然后找到 span 标签上的 a 标签，最后再去找到 div 标签，然后给符合这种条件的 span 标签设置颜色，这样的递归过程就很复杂。所以我们应该尽可能的避免写过于具体的 CSS 选择器，然后对于 HTML 来说也尽量少的添加无意义标签，保证层级扁平。

- 将动画效果应用到position属性为absolute或fixed的元素上，避免影响其他元素的布局，这样只是一个重绘，而不是回流，同时，控制动画速度可以选择 requestAnimationFrame，详见探讨requestAnimationFrame。
- 避免使用CSS表达式，可能会引发回流。
- 将频繁重绘或者回流的节点设置为图层，图层能够阻止该节点的渲染行为影响别的节点，例如will-change、video、iframe等标签，浏览器会自动将该节点变为图层。
- CSS3 硬件加速（GPU加速），使用css3硬件加速，可以让transform、opacity、filters这些动画不会引起回流重绘 。但是对于动画的其它属性，比如background-color这些，还是会引起回流重绘的，不过它还是可以提升这些动画的性能。

②JavaScript

- 避免频繁操作样式，最好一次性重写style属性，或者将样式列表定义为class并一次性更改class属性。
- 避免频繁操作DOM，创建一个documentFragment，在它上面应用所有DOM操作，最后再把它添加到文档中。
- 避免频繁读取会引发回流/重绘的属性，如果确实需要多次使用，就用一个变量缓存起来。

- 对具有复杂动画的元素使用绝对定位，使它脱离文档流，否则会引起父元素及后续元素频繁回流。

2.说说浏览器和 Node 事件循环的区别

题目应该是：浏览器和node的事件循环的区别吧，

先上链接：

- 浏览器与Node的事件循环(Event Loop)有何区别?
- html#event-loops
- Node.js 事件循环，定时器和 process.nextTick()

第一个链接里面大佬讲的已经非常透彻了我来总结一下。

3.cookie 和 token 都存放在 header 中，为什么不会劫持 token?

> cookie：登陆后后端生成一个sessionid放在cookie中返回给客户端，并且服务端一直记录着这个sessionid，客户端以后每次请求都会带上这个sessionid，服务端通过这个sessionid来验证身份之类的操作。所以别人拿到了cookie拿到了sessionid后，就可以完全替代你。
>
> token：登陆后后端不返回一个token给客户端，客户端将这个token存储起来，然后每次客户端请求都需要开发者手动将token放在header中带过去，服务端每次只需要对这个token进行验证就能使用token中的信息来进行下一步操作了。
>
> xss：用户通过各种方式将恶意代码注入到其他用户的页面中。就可以通过脚本获取信息，发起请求，之类的操作。
>
> csrf：跨站请求攻击，简单地说，是攻击者通过一些技术手段欺骗用户的浏览器去访问一个自己曾经认证过的网站并运行一些操作（如发邮件，发消息，甚至财产操作如转账和购买商品）。由于浏览器曾经认证过，所以被访问的网站会认为是真正的用户操作而去运行。这利用了web中用户身份验证的一个漏洞：**简单的身份验证只能保证请求发自某个用户的浏览器，却不能保证请求本身是用户自愿发出的**。csrf并不能够拿到用户的任何信息，它只是欺骗用户浏览器，让其以用户的名义进行操作。
>
> csrf例子：假如一家银行用以运行转账操作的URL地址如下：
> http://www.examplebank.com/withdraw?account=AccoutName&amount=1000&for=PayeeName那么，一个恶意攻击者可以在另一个网站上放置如下代码：`<img src="<` `http://www.examplebank.com/withdraw?account=Alice&amount=1000&for=Badman` `>">` 如果有账户名为Alice的用户访问了恶意站点，而她之前刚访问过银行不久，登录信息尚未过期，那么她就会损失1000资金。

上面的两种攻击方式，如果被xss攻击了，不管是token还是cookie，都能被拿到，所以对于xss攻击来说，cookie和token没有什么区别。但是对于csrf来说就有区别了。

以上面的csrf攻击为例：

- cookie：用户点击了链接，cookie未失效，导致发起请求后后端以为是用户正常操作，于是进行扣款操作。
- token：用户点击链接，由于浏览器不会自动带上token，所以即使发了请求，后端的token验证不会通过，所以不会进行扣款操作。

这是个人理解的为什么只劫持cookie不劫持token的原因。

4.浏览器缓存读取规则

https://www.jianshu.com/p/54cc04190252 浏览器的缓存机制

5.为什么通常在发送数据埋点请求的时候使用的是 1x1 像素的透明 gif 图片？

1. 能够完成整个 HTTP 请求+响应（尽管不需要响应内容）
2. 触发 GET 请求之后不需要获取和处理数据、服务器也不需要发送数据
3. 跨域友好
4. 执行过程无阻塞
5. 相比 XMLHttpRequest 对象发送 GET 请求，性能上更好
6. GIF的最低合法体积最小（最小的BMP文件需要74个字节，PNG需要67个字节，而合法的GIF，只需要43个字节）

参考资料：SegmentFault 上的回答Web beaconUsing a Beacon Image for GitHub, Website and Email Analytics为什么前端监控要用 GIF 打点

6.介绍下如何实现 token 加密

1. 需要一个secret（随机数）
2. 后端利用secret和加密算法(如：HMAC-SHA256)对payload(如账号密码)生成一个字符串(token)，返回前端
3. 前端每次request在header中带上token
4. 后端用同样的算法解密

十二 地址收藏

1. 前端面试经典题目合集_牛客网

https://www.nowcoder.com/ta/front-end-interview

2.鹅厂(腾讯)web前端面试居然还有套路?

https://segmentfault.com/a/1190000009433974

3.面试那些事儿

https://segmentfault.com/a/1190000006950447

4.前沿开发团队的面试过程

https://segmentfault.com/a/1190000006909250

5.向那些年我们经历的一面N面致敬

https://segmentfault.com/a/1190000004827689

6.五个典型的JavaScript面试题

http://www.zcfy.cc/article/5-typical-javascript-interview-exercises-1960.html

7.征服前端面试

https://segmentfault.com/p/1210000009636596/read

8.阿里面经（附回答）面试必看

http://www.jianshu.com/p/08368a8e4ca7

9.2018前端校招总结

https://segmentfault.com/a/1190000011635556

10.一道有价值的JS继承面试题

https://segmentfault.com/a/1190000008888142

11.当面试官问「你有什么要问我的吗」时，应该问什么?

https://yanzhi.lagou.com/question/281.html

12.最近遇到的前端面试题(2017.02.23更新版)

http://www.jianshu.com/p/3944732228f0

13.整理下《前端江湖面试》对自己有益的题目。

https://segmentfault.com/a/1190000010868439

14.这里有超过20家的前端面试题，你确定不点进来看看?

https://juejin.im/post/58c51b5c44d90400698da686

15.破解前端面试（80% 应聘者不及格系列）

https://juejin.im/post/58f558efac502e006c3e5c97

16.1月前端面试记

https://juejin.im/post/587dab348d6d810058d87a0a

17.阿里、网易、滴滴共十次前端面试碰到的问题 - 掘金

https://juejin.im/post/59316e682f301e0058378558

18.学生问的一道javascript面试题[来自腾讯]

http://www.cnblogs.com/ghostwu/p/7272132.html

19.前端面试必备——基本排序算法

http://blog.csdn.net/owen1190/article/details/76215932

20.技术面试中常被问到是否参与开源，那如何清晰展示 GitHub 项目呢?

http://blog.jobbole.com/112322/

21.你有必要知道的 25 个 JavaScript 面试题

https://segmentfault.com/a/1190000004180569

22.2017秋招前端工程师百度阿里网易腾讯(全offer)面经|掘金技术征文

https://juejin.im/post/59cdb971f265da066b397421

23.我的前端进阶之路 - 掘金

https://juejin.im/post/591afe90128fe1005ce339f7

24.前后端分离与前端工程化

https://segmentfault.com/a/1190000006751300

25.大公司里怎样开发和部署前端代码?

https://www.zhihu.com/question/20790576

十三 性能优化

1.如何在 H5 和小程序项目中计算白屏时间和首屏时间，说说你的思路

JavaScript

```
白屏时间=页面开始展示的时间点-开始请求时间点。
开始请求时间点可以通过Performance Timing.navigation Start 。那么页面开始展示的时间点
怎么获取呢。已经知道渲染过程是逐步完成的，而且页面解析是按照文档流从上至下解析的，
因此一般认为开始解析body的时间点就是页面开始展示的时间。所以可以通过在head标签的末尾
插入script来统计时间节点作为页面开始展示时间节点。但是这种方式需要打点，因此也有很多
项目为了简化白屏时间的获取会选择忽略head解析时间直接用Performance Timing.dom Loading
 来表示页面开始展示的时间，即使用domloading-navigation Start来表示白屏时间。
首屏时间=首屏内容渲染结束时间点-开始请求时间点。
同样开始请求时间点可以通过Performance Timing.navigation Start获取。
首屏内容渲染结束的时间点通常有以下几种方法获取：
（1）首屏模块标签标记法
适用于于首屏内容不需要通过拉取数据才能生存以及页面不考虑图片等资源加载的情况。
通过在 HTML 文档中对应首屏内容的标签结束位置，使用内联的 JavaScript 代码
记录当前时间戳作为首屏内容渲染结束的时间点。
（2）统计首屏内加载最慢的图片的时间
通常首屏内容加载最慢的就是图片资源，因此可以把首屏内加载最慢的图片加载完成的时间
作为首屏时间。由于浏览器对每个页面的 TCP 连接数有限制，使得并不是所有图片都能立刻开始下载
和显示。因此在 DOM树 构建完成后会通过遍历首屏内的所有图片标签，并且监听所有图片标签
onload 事件，最终遍历图片标签的加载时间获取最大值，将这个最大值作为首屏时间。
（3）自定义首屏内容计算法
由于统计首屏内图片完成加载的时间比较复杂。所以在项目中通常会通过自定义模块内容，
来简化计算首屏时间。例如忽略图片等资源加载情况，只考虑页面主要 DOM；只考虑首屏的主要模块，
而不是严格意义首屏线以上的所有内容。
可交互时间=用户可以正常进行事件输入时间点-开始请求时间点。
PerformanceTiming有一个domInteractive属性，代表了DOM结构结束解析的时间点，
就是Document.ready State属性变为“interactive”
```

详细地址：

这篇blog挺详细 ==> https://lz5z.com/Web%E6%80%A7%E8%83%BD%E4%BC%98%E5%8C%96-%E9%A6%96%E5%B1%8F%E5%92%8C%E7%99%BD%E5%B1%8F%E6%97%B6%E9%97%B4/

2.：前端项目如何找出性能瓶颈（阿里）

Plain Text

```
分享一下我近期的经验，之前项目也碰到过用起来很卡的情况，就是用element ui的tab切换组件时，
点击tab切换非常卡，非常耗时，在排除了网络请求和js代码执行时间过长等原因后，
跑了一次perfermance，结果发现大部分时间都花费在了 DOM GC上了，分析了
下原因可能时dom结构太多导致每次tab切换渲染太耗时了。由于我每个tab里面的html结构都一样，
都是一个table，只是每次tab切换时请求的数据不一样，我就把table抽离出来了，
放到tab组件外面，然后tab里面就空了，就没有那么多dom了，tab切换就不卡了，很流畅。
（ps：tab有20-30个切换选项，本人语文水平不行，描述的不清楚，望轻喷。）
```

JavaScript

```
我觉得应该首先理一理会造成性能损耗的一些场景：
比如大列表的渲染,大量dom的渲染 2.大量图片的加载,过多资源的请求.
3.代码中有没有耗时的计算操作,或则大量循环.递归
编写的组件过于庞大 层级过深,依赖模块过多等.
我觉得首先就是查看请求的资源体积是否过大,如果过大考虑压缩,
减少不必要的资源的请求,不必要的js代码的代码加载,用字体图标代替图片,异步加载等等.
但是我觉得基本的优化策略(减少请求数,压缩请求资源的体积)都已经做过了,感觉性能还是没有提升,
可能应该关注与代码层面的优化吧,比如过大的第三方库能不能换成轻量级的,
代码中有没有很耗时的操作循环和递归,过多的分支条件语句,能不能改写以提高执行效率,
简化复杂的组件逻辑,减少不必要的依赖,是否有杀鸡用了牛刀的操作等.暂时想到的就这些.
```

JavaScript

```
性能瓶颈一般是表现在用户体验方面，比如页面加载过慢，动画卡顿，交互延迟等等，
具体问题具体分析。 比如PM反馈说，你们这个页面怎么加载这么慢啊，好几秒才呈现，
这种现象问题可能出现在请求资源过多，资源太大，没有压缩，没有缓存等等，
那么就等减少http请求，压缩资源，设置缓存。 如果PM反馈说，
这个列表滑动的时候加载数据一卡一卡的，那么性能瓶颈可能出现在频繁操作，数据太大等等，
那么就得做懒加载，函数节流等等； 总之前端的性能瓶颈往往都是有表现的，
需要针对具体表现具体分析具体解决
```

推荐一篇文章：https://juejin.im/post/5db2beb8e51d455b450a64b4#heading-9

十四，设计模式

1.介绍下观察者模式和订阅-发布模式的区别，各自适用于什么场景

JavaScript

```
发布-订阅模式就好像报社， 邮局和个人的关系，报纸的订阅和分发是由邮局来完成的。
报社只负责将报纸发送给邮局。
观察者模式就好像 个体奶农和个人的关系。奶农负责统计有多少人订了产品，
所以个人都会有一个相同拿牛奶的方法。奶农有新奶了就负责调用这个方法。
```

JavaScript

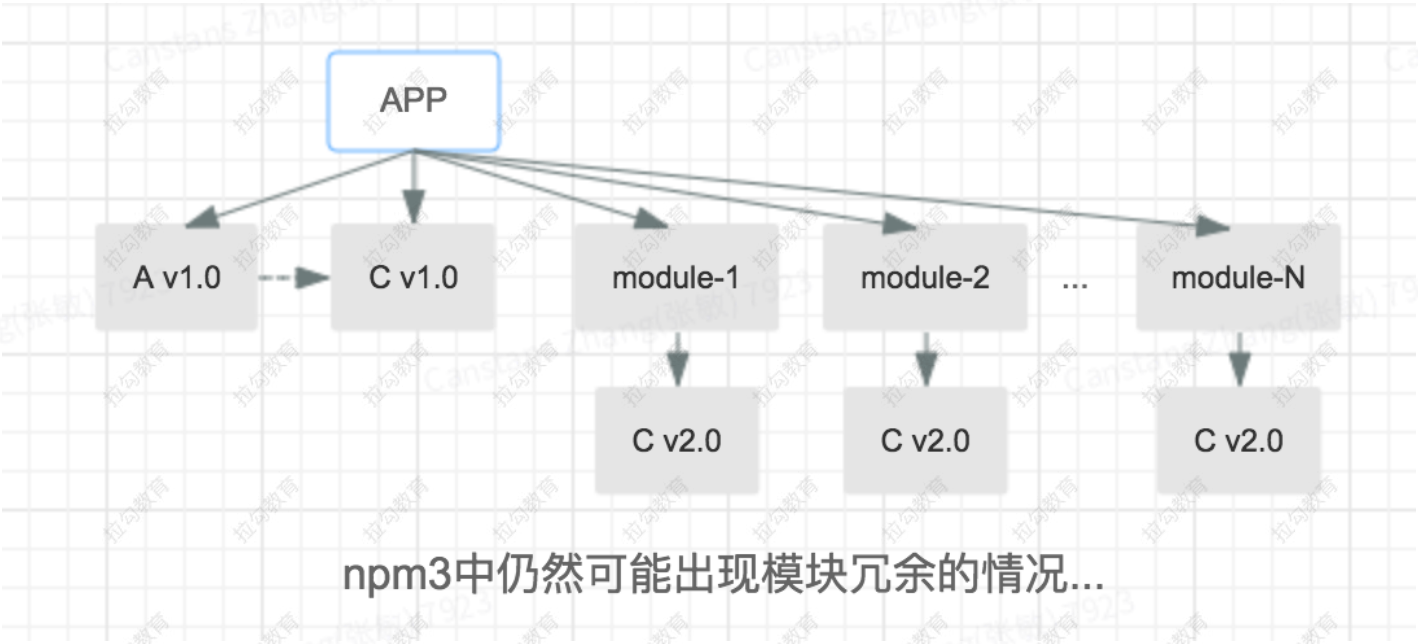
```
var subject = {
        observers: [],
        notify() {
                this.observers.forEach(observer =>{
                        observer.update()
                })
        },
        attach (observer) {
                this.observers.push(observer)
        }
}
var observer = {
        update(){
                alert('updated')
        }
}
subject.attach(observer)
subject.notify()
而使用订阅发布模式，使用中间订阅发布对象的方式如下
var publisher = {
        publish(pubsub) {
                pubsub.publish()
        }
}
var pubsub = {
        subscribes: [],
        publish() {
                this.subscribes.forEach(subscribe =>{
                        subscribe.update();
                })
        },
        subscribe(sub) {
                this.subscribes.push(sub)
        }
}
```

```
var subscribe = {
        update() {
                console.log('update')
        },
        subscribe(pubsub) {
            pubsub.subscribe(this);
        }
}
subscribe.subscribe(pubsub)
publisher.publish(pubsub)
```

自己认为，两种模式本质都是一样的，主要关键点都在于**注册**（添加到注册数组中）和**触发**（触发注册数组中的内容），只是订阅/发布模式对**注册和触发**进行了解耦。可以看到，使用订阅发布模式中发布者触发publish的时候，可以选择触发哪一些订阅者集合（因为publish参数传递了中间集合，可以定义多个pubsub集合），而观察者模式则只能触发所有的被观察对象。

十五，工程化

1.介绍下 npm 模块安装机制，为什么输入 npm install 就可以自动安装对应的模块?



JavaScript

```
当A模块下的C v1.0模块被更新至C v2.0的前提下，我们可以通过npm dedupe把所有C v2.0的
二级依赖模块“重定向”到一级目录下的那个C v1.0。 那如果A模块的C v1.0没更新到C v2.0，
使用npm dedupe就没效果，那也就是说这个npm dedupe也只是在特定情况有效。
也就是npm到目前为止依然是没办法完全解决冗余的。
```

2.介绍下 webpack 热更新原理，是如何做到在不刷新浏览器的前提下更新页面的

JavaScript

```
1.当修改了一个或多个文件； 2.文件系统接收更改并通知webpack；
3.webpack重新编译构建一个或多个模块，并通知HMR服务器进行更新；
4.HMR Server 使用webSocket通知HMR runtime 需要更新，
HMR运行时通过HTTP请求更新jsonp；
5.HMR运行时替换更新中的模块，如果确定这些模块无法更新，则触发整个页面刷新。
```

3,webpack 打包 vue 速度太慢怎么办

JavaScript

```
1.使用webpack-bundle-analyzer对项目进行模块分析生成report，查看report后看看哪些模块体积过大，然后针对性优化，比如我项目中引用了常用的UI库element-ui和v-charts等
2.配置webpack的externals，官方文档的解释：防止将某些import的包(package)打包到 bundle 中，而是在运行时(runtime)再去从外部获取这些扩展依赖。 所以，可以将体积大的库分离出来：
// ...
externals: {
    'element-ui': 'Element',
    'v-charts': 'VCharts'
}
3.然后在main.js中移除相关库的import
4.在index.html模板文件中，添加相关库的cdn引用，如：
<script src="https://unpkg.com/element-ui@2.10.0/lib/index.js"></script>
<script src="https://cdn.jsdelivr.net/npm/echarts/dist/echarts.min.js"></script>
<script src="https://cdn.jsdelivr.net/npm/v-charts/lib/index.min.js"></script>
经过以上的处理，再尝试编译打包，会发现速度快了一些。 有什么更好的方式或不对的地方欢迎指出
superBlithe commented
"打包慢"，是一个综合的因素，和vue关系不大。
1：确保下webpack，npm，node 及主要库版本要新，比如：4.x比3.x提升很多。
2：loader范围缩小到src项目文件！一些不必要的loader能关就关了吧
3：eslint代码校验其实是一个很费时间的一个步奏。 ：可以把eslint的范围缩小到src,且只检查 *.js 和 *.vue ：生产环境不开启lint，使用pre-commit或者husky在提交前校验
4：happypack多进程进行
如果上面优化后，时间还是不满意的话，就尝试下5,6吧。
5：动态链接库（DllPlugin），楼上已说。有点类似配置的externals。 补充一下： 缺点：将不能按需加载，会将配置的第三方库全部打包进去。 推荐：可以将使用率较高的包采用dll方案。
6：HardSourceWebpackPlugin会将模块编译后进行缓存，第一次之后速度会明显提升
```

4.webpack 中 loader 和 plugin 的区别是什么（平安）

JavaScript

```
loader，它是一个转换器，将A文件进行编译成B文件，比如：将A.less转换为A.css，
单纯的文件转换过程。
plugin是一个扩展器，它丰富了webpack本身，针对是loader结束后，
webpack打包的整个过程，它并不直接操作文件，而是基于事件机制工作，
会监听webpack打包过程中的某些节点，执行广泛的任务
```

5.babel 怎么把字符串解析成 AST，是怎么进行词法/语法分析的?

JavaScript

```
定义
Babel 是我们知道的将 ES6、ES7等代码转译为 ES5 代码且能安全稳定运行最好的工具
同时它允许开发者开发插件，能够在编译时期转换 JavaScript 的结构。
Babel概述
我们需要知道 3 个 Babel 处理流程中的重要工具；解析Babylon是一个解析器，它可以将javascript字符串，转化为更加友好的表现形式，称之为抽象语法树；
在解析过程中有两个阶段：词法分析和语法分析，词法分析阶段：字符串形式的代码转换为令牌（tokens）流,令牌类似于AST中的节点；
语法分析阶段：把一个令牌流转化为AST的形式，同时这个阶段会把令牌中的信息转化为AST的表述结构
转换babel-traverse模块允许你浏览、分析和修改抽象语法树（AST Abstract Syntax Tree）Babel接收解析得到的AST并通过babel-traverse对其进行深度优先遍历，在此过程中对节点进行添加、更新及移除操作。
生成babel-generator模块用来将转换后的抽象语法树（AST Abstract Syntax Tree）转化为Javascript 字符串将经过转换的AST通过babel-generator再转换为js代码，过程及时深度遍历整个AST,然后构建转换后的代码字符串。
Babel 中重要的对象Vistor
babel 在处理一个节点时，是以访问者的形式获取节点的信息，并进行相关的操作，这种操作是通过visitor对象实现的。
在 visitor 中定义了处理不同节点的函数。
visitor: {
  Program: {
    enter(path, state) {
      console.log('start processing this module...');
    },
      exit(path, state) {
        console.log('end processing this module!');
      }
  },
    ImportDeclaration:{
      enter(path, state) {
        console.log('start processing ImportDeclaration...');
        // do something
```

```
      },
        exit(path, state) {
          console.log('end processing ImportDeclaration!');
          // do something
        }
    }
}
```

AST

定义

AST (Abstract Syntax Tree)是抽象语法树英文的缩写，AST语法树每一层都拥有相同的结构，这样的每一层结构也被叫做节点（Node）。

AST 是源代码的抽象语法结构树状表现形式，Webpack、ESLint、JSX、TypeScript 的编译和模块化规则之间的转化都是通过 AST 来实现对代码的检查、分析以及编译等操作。

一个 AST 可以由单一的节点或是成百上千个节点构成。 它们组合在一起可以描述用于静态分析的程序语法。

Javascript 语法的AST（抽象语法树）

javascript 语句要想知道抽象语法树之后的代码是什么，里面的字段都代表什么含义以及遍历的规则，可以通过javascript语法转换AST工具来实现javascript语法的在线转换； 例如：

esprima、estraverse 和 escodegen 模块是操作 AST 的三个重要模块，也是实现 babel 的核心依赖。

例如：语法转换插件需要借助 babel-core 和 babel-types 两个模块,就是依赖 esprima、estraverse 和 escodegen

转换的抽象语法树：`var answer= 6 * 7`

每一个含有type属性的对象，我们称之为节点，修改是指获取对应的类型并修改改节点的属性即可；

```
 {
"type": "Program",
"body": [
    {
        "type": "VariableDeclaration",
        "declarations": [
            {
                "type": "VariableDeclarator",
                "id": {
                    "type": "Identifier",
                    "name": "answer"
                },
                "init": {
                    "type": "BinaryExpression",
                    "operator": "*",
                    "left": {
                        "type": "Literal",
                        "value": 6,
                        "raw": "6"
```

```
                    },
                    "right": {
                        "type": "Literal",
                        "value": 7,
                        "raw": "7"
                    }
                }
            }
        ],
        "kind": "var"
    }
  ],
  "sourceType": "script"
}
```

estraverse 遍历和修改AST

查看遍历过程：

```
const esprima = require("esprima");
const estraverse = require("estraverse");

let code = "var answer=6 * 7";

// 遍历语法树
estraverse.traverse(esprima.parseScript(code), {
  enter(node) {
    console.log("enter", node.type);
  },
  leave(node) {
    console.log("leave", node.type);
  }
});
```

打印结果如下：

以上代码通过estraverse模块的traverse方法，将esprima模块装换的AST进行了遍历。

通过打印type属性，可以知道深度遍历AST就是遍历每一层的type属性，所以遍历会分为两个阶段，进入阶段和离开阶段，在traverse方法中分别用参数enter和leave两个函数监听；

escodegen 将 AST 转换成 JS

```
const esprima = require("esprima");
const estraverse = require("estraverse");
const escodegen= require("escodegen");

let code = "var answer=6 * 7";

let tree=esprima.parseScript(code); // 生成语法树
```

```
// 遍历语法树
estraverse.traverse(tree, {
    enter(node) {
        console.log("enter", node.type);
        // 修改变量名
        if(node.type==='VariableDeclarator'){
                node.id.name='result';
        }
    },
    leave(node) {
        console.log("leave", node.type);
    }
});

// 编译修改后的语法树;
let compileTreeJS=escodegen.generate(tree);
console.log(compileTreeJS);
打印结果如下 ：
var result=6 * 7

通过工具了解抽象语法树在 JavaScript 中的体现以及在 NodeJS 中用于生成、遍历和修改 AST 抽象语法树的核心依赖；让我们有了更加深刻地认识；
参考整合：https://juejin.im/post/5dca1eb86fb9a04a76025292
```

十六 算法题

1.请分别用深度优先思想和广度优先思想实现一个拷贝函数?

Plain Text

```
// 如果是对象/数组，返回一个空的对象/数组，
// 都不是的话直接返回原对象
// 判断返回的对象和原有对象是否相同就可以知道是否需要继续深拷贝
// 处理其他的数据类型的话就在这里加判断
function getEmpty(o){
        if(Object.prototype.toString.call(o) === '[object Object]'){
                return {};
        }
        if(Object.prototype.toString.call(o) === '[object Array]'){
                return [];
        }
        return o;
}

```

```
function deepCopyBFS(origin){
        let queue = [];
        let map = new Map(); // 记录出现过的对象，用于处理环

        let target = getEmpty(origin);
        if(target !== origin){
                queue.push([origin, target]);
                map.set(origin, target);
        }

        while(queue.length){
                let [ori, tar] = queue.shift();
                for(let key in ori){
                        // 处理环状
                        if(map.get(ori[key])){
                                tar[key] = map.get(ori[key]);
                                continue;
                        }

                        tar[key] = getEmpty(ori[key]);
                        if(tar[key] !== ori[key]){
                                queue.push([ori[key], tar[key]]);
                                map.set(ori[key], tar[key]);
                        }
                }
        }

        return target;
}

function deepCopyDFS(origin){
        let stack = [];
        let map = new Map(); // 记录出现过的对象，用于处理环

        let target = getEmpty(origin);
        if(target !== origin){
                stack.push([origin, target]);
                map.set(origin, target);
        }

        while(stack.length){
                let [ori, tar] = stack.pop();
                for(let key in ori){
                        // 处理环状
```

```
                        if(map.get(ori[key])){
                                tar[key] = map.get(ori[key]);
                                continue;
                        }

                        tar[key] = getEmpty(ori[key]);
                        if(tar[key] !== ori[key]){
                                stack.push([ori[key], tar[key]]);
                                map.set(ori[key], tar[key]);
                        }
                }
        }

        return target;
}

// test
[deepCopyBFS, deepCopyDFS].forEach(deepCopy=>{
        console.log(deepCopy({a:1}));
        console.log(deepCopy([1,2,{a:[3,4]}]))
        console.log(deepCopy(function(){return 1;}))
        console.log(deepCopy({
                x:function(){
                        return "x";
                },
                val:3,
                arr: [
                        1,
                        {test:1}
                ]
        }))

        let circle = {};
        circle.child = circle;
        console.log(deepCopy(circle));
})
```

2.已知如下数组：

var arr = [[1, 2, 2], [3, 4, 5, 5], [6, 7, 8, 9, [11, 12, [12, 13, [14]]]], 10];

编写一个程序将数组扁平化去并除其中重复部分数据，最终得到一个升序且不重复的数组?

Plain Text

```
arr.toString().split(",").sort((a,b)=>{ return a-b})
```

3.如何实现一个 new

先理清楚 new 关键字调用函数都的具体过程，那么写出来就很清楚了

1. 首先创建一个空的对象，空对象的__proto__属性指向构造函数的原型对象
2. 把上面创建的空对象赋值构造函数内部的this，用构造函数内部的方法修改空对象
3. 如果构造函数返回一个非基本类型的值，则返回这个值，否则上面创建的对象

Plain Text

```
function _new(fn, ...arg) {
    var obj = Object.create(fn.prototype);
    const result = fn.apply(obj, ...arg);
    return Object.prototype.toString.call(result) == '[object Object]' ? result : obj;
}
```

4.两个数组合并成一个数组?

请把两个数组 ['A1', 'A2', 'B1', 'B2', 'C1', 'C2', 'D1', 'D2'] 和 ['A', 'B', 'C', 'D']，合并为 ['A1', 'A2', 'A', 'B1', 'B2', 'B', 'C1', 'C2', 'C', 'D1', 'D2', 'D']。

Plain Text

```
let a1 =  ['A1', 'A2', 'B1', 'B2', 'C1', 'C2', 'D1', 'D2']
let a2 = ['A', 'B', 'C', 'D'].map((item) => {
  return item + 3
})

let a3 = [...a1, ...a2].sort().map((item) => {
  if(item.includes('3')){
    return item.split('')[0]
  }
  return item
})
```

5.使用迭代的方式实现 flatten 函数?

Plain Text

```
const flatten = array => array.reduce((acc, cur) => (Array.isArray(cur) ? [...acc, ...flatten(cur)] : [...acc, cur]), [])
```

6.实现一个 sleep 函数?

Plain Text

```
const sleep = (time) => {
  return new Promise(resolve => setTimeout(resolve, time))
}

async function sleepAsync() {
  console.log('fuck the code')
  await sleep(1000)
  console.log('fuck the code again')
}

sleepAsync()
```

7.冒泡排序如何实现，时间复杂度是多少， 还可以如何改进?

Plain Text

```
function bubbleSort(arr) {
    for (let i = 0; i < arr.length; i++) {
        for (let j = 0; j < arr.length - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                const temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
    console.log(arr);
}

// 改进冒泡排序
function bubbleSort1(arr) {
    let i = arr.length - 1;

    while (i > 0) {
        let pos = 0;
        for (let j = 0; j < i; j++) {
            if (arr[j] > arr[j + 1]) {
                pos = j;
                const temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
        i = pos;
    }
    console.log(arr);
}
```

8.如下：{1:222, 2:123, 5:888}，请把数据处理为如下结构：[222, 123, null, null, 888, null, null, null, null, null, null]?

Plain Text

```
let obj = {1:222, 2:123, 5:888};
const result = Array.from({ length: 12 }).map((_, index) => obj[index + 1] || null);
console.log(result)
```

9.给定两个数组，写一个方法来计算它们的交集?

Plain Text

```
const intersect = (nums1, nums2) => {
  const map = {}
  const res = []
  for (let n of nums1) {
    if (map[n]) {
      map[n]++
    } else {
      map[n] = 1
    }
  }
  for (let n of nums2) {
    if (map[n] > 0) {
      res.push(n)
      map[n]--
    }
  }
  return res
}
```

10.如何把一个字符串的大小写取反（大写变小写小写变大写），例如’ AbC' 变成 'aBc'？

`[].map.call(str, function(item){ return /[a-z]/.test(item) ? item.toUpperCase() : item.toLowerCase(); }).join('');`

11.使用 JavaScript Proxy 实现简单的数据绑定?

Plain Text

```
<body>
  hello,world
  <input type="text" id="model">
  <p id="word"></p>
</body>
<script>
  const model = document.getElementById("model")
  const word = document.getElementById("word")
  var obj= {};

  const newObj = new Proxy(obj, {
      get: function(target, key, receiver) {
        console.log(`getting ${key}!`);
        return Reflect.get(target, key, receiver);
      },
      set: function(target, key, value, receiver) {
        console.log('setting',target, key, value, receiver);
        if (key === "text") {
          model.value = value;
          word.innerHTML = value;
        }
        return Reflect.set(target, key, value, receiver);
      }
    });

  model.addEventListener("keyup",function(e){
    newObj.text = e.target.value
  })
</script>
```

12.算法题「旋转数组」给定一个数组，将数组中的元素向右移动 k 个位置，其中 k 是非负数。

Plain Text

```
const moveArr = (arr,k)=>arr.splice(-k%arr.length).concat(arr)
```

13.打印出 1 - 10000 之间的所有对称数?

以10000 为例，列出所有位数的对称数，比如1位数的 对称数， 2位数的对称数...到5位数， 再判断是否小于10000。 对称数生成： 遍历最长位数（如5位）。 每个位置遍历 0 - 9，对称即可。

Plain Text

```
var result = [];
for(let i=1;i<10;i++){
    result.push(i*11); // 两位数的对称数
    for(let j=0;j<10;j++){
        result.push(i*101+j*10) //  三位数的对称数
        result.push(i*1001+j*110) // 四位数的对称数，当i和j均为9是值为9999
    }
}
console.log(result);
```

14.在输入框中如何判断输入的是一个正确的网址?

Plain Text

```
function isUrl(url) {
       try {
           new URL(url);
           return true;
       }catch(err){
     return false;
}}
```

15.模拟实现一个 localStorage

Plain Text

```
!window.localStorage && !function(win) {
  var thousandYears = 1e3 * 365 * 24 * 36e5;

  function getCookies() {
    return document.cookie.match(/([^;=]+)=([^;]+)/g) || [];
  }

  function getExpires(flag) {
    flag = flag || 1;

    return 'expires=' +
      (new Date((+new Date()) + thousandYears * flag)).toUTCString();
  }

  function get(key) {
```

```
    var cookies = getCookies();

    for (var i = 0; i < cookies.length; i++) {
      var param = cookies[i].match(/^\s*([^=]+)=(.+)/);

      if (param[1] === String(key)) {
        return decodeURIComponent(param[2]);
      }
    }

    return null;
  }

  function set(key, value, isExpired) {
    document.cookie = [
      key + '=' + encodeURIComponent(value),
      getExpires(isExpired ? -1 : 1),
      'path=/'
    ].join('; ');
  }

  function remove(key) {
    set(key, '', true);
  }

  function clear() {
    var cookies = getCookies();

    for (var i = 0; i < cookies.length; i++) {
      var key = cookies[i].match(/^\s*([^=]+)/)[1];
      remove(key);
    }
  }

  // 注册到 window 对象上
  win.localStorage = {
    getItem: get,
    setItem: set,
    removeItem: remove,
    clear: clear
  };
}(window);
```

16.考虑到性能问题，如何快速从一个巨大的数组中随机获取部分元素？

Plain Text

```
/* 洗牌算法:
    1.生成一个0 - arr.length 的随机数
    2.交换该随机数位置元素和数组的最后一个元素，并把该随机位置的元素放入结果数组
    3.生成一个0 - arr.length - 1 的随机数
    4.交换该随机数位置元素和数组的倒数第二个元素，并把该随机位置的元素放入结果数组
    依次类推，直至取完所需的10k个元素
*/

function shuffle(arr, size) {
    let result = []
    for (let i = 0; i < size; i++) {
        const randomIndex = Math.floor(Math.random() * (arr.length - i))
        const item = arr[randomIndex]
        result.push(item)
        arr[randomIndex] = arr[arr.length - 1 - i]
        arr[arr.length - 1 - i] = item
    }
    return result
}
```

17.编程题，找出字符串中连续出现最多的字符和个数（蘑菇街）？

JavaScript

```
'abcaakjbb' => {'a':2,'b':2}
'abbkejsbcccwqaa' => {'c':3}
```

18.写一个单向链数据结构的 js 实现并标注复杂度（水滴筹）？

Plain Text

```
class LinkList {
  constructor() {
    this.head = null
  }

  find(value) {
    let curNode = this.head
    while (curNode.value !== value) {
      curNode = curNode.next
    }
```

```
    return curNode
  }

  findPrev(value) {
    let curNode = this.head
    while (curNode.next!==null && curNode.next.value !== value) {
      curNode = curNode.next
    }
    return curNode
  }

  insert(newValue, value) {
    const newNode = new Node(newValue)
    const curNode = this.find(value)
    newNode.next = curNode.next
    curNode.next = newNode
  }

  delete(value) {
    const preNode = this.findPrev(value)
    const curNode = preNode.next
    preNode.next = preNode.next.next
    return curNode
  }
}

class Node {
  constructor(value, next) {
    this.value = value
    this.next = null
  }
}
```

19.统计 1 ~ n 整数中出现 1 的次数?

Plain Text

```
function countOne(n) {
    var factor = 1;
    let count = 0;
    let next = parseInt(n / factor);
    while (next !== 0) {
        var lower = n - next * factor
        var curr = next % 10;
        var high = parseInt(n / (10 * factor));

        if (curr === 0) {
            count += high * factor;
        } else if (curr === 1) {
            count += high * factor + lower + 1
        } else {
            count += (high + 1) * factor
        }

        factor *= 10;
        next = parseInt(n / factor);
    }
    return count;
}
```

20.求两个日期中间的有效日期?

JavaScript

```
function rangeDay (day1, day2) {    const result = []    const dayTimes = 24*60*60*1000    const startTime = day1.getTime()    const range = day2.getTime() - startTime    let total = 0        while (total <= range && range > 0) {        result.push(new Date(startTime + total).toLocaleDateString().replace(/\//g, '-'))        total += dayTimes    } return result }; rangeDay(new Date("2015-02-08"), new Date("2015-03-03"))
```

21.（算法题）求多个数组之间的交集（阿里）？

Plain Text

```
function intersect(...args) {
  if (args.length === 0) {
    return []
  }

  if (args.length === 1) {
    return args[0]
  }

  return args.reduce((result, arg) => {
    return result.filter(item => arg.includes(item))
  })
}
```

22.将 '10000000000' 形式的字符串，以每 3 位进行分隔展示 '10.000.000.000'?

Plain Text

```
// 德国以 . 分割金钱，转到德国当地格式化方案即可
10000000000..toLocaleString('de-DE')

// 寻找字符空隙加 .
'10000000000'.replace(/\B(?=(\d{3})+(?!\d))/g, '.')

// 寻找数字并在其后面加 .
'10000000000'.replace(/(\d)(?=(\d{3})+\b)/g, '$1.')
```

23.手写二进制转 Base64（阿里）?

Plain Text

```
let  binaryDataToBase64 = byte => {
     let sBytes =  new Uint8Array(byte).reduce((accu,next)=> {
         retrun accu + String.fromCharCode(next)
}, '')
    return `${btoa(sBytes)}`
}
```

24.二分查找如何定位左边界和右边界?

Plain Text

Plain Text

```
//递归查找
function erfen_digui(arr, val, left = 0, right = arr.length - 1) {
        if (left > right) {
          return -1;
        }
        let cent = Math.floor((right + left) / 2);
        if (arr[cent] === val) {
          return `最终查找结果下标为${cent}`;
        } else if (arr[cent] > val) {
          right = cent - 1;
        } else {
          left = cent + 1;
        }
        return erfen_digui(arr, val, left, right);
      }
//非递归查找
      function erfen_feidigui(arr, val) {
        let left = 0,
          right = arr.length - 1;
        while (left <= right) {
          let cent = left + Math.floor((right - left) / 2);
          if (arr[cent] === val) {
            return `最终查找结果下标为${cent}`;
          } else if (arr[cent] > val) {
            right = cent - 1;
          } else {
            left = cent + 1;
          }
        }
        return -1;
      }

//左边界查找（查找第一个元素）
function erfen_digui(arr, val, left = 0, right = arr.length - 1) {
        if (left > right) {
          return -1;
        }
        let cent = Math.floor((right + left) / 2);
        if (arr[cent] === val) {
          /****************改动点*******************/
          if (arr[cent - 1] === val) {
            right = cent - 1;
          } else {
```

```
            return `最终查找结果下标为${cent}`;
          }
          /****************************************/
        } else if (arr[cent] > val) {
          right = cent - 1;
        } else {
          left = cent + 1;
        }
        return erfen_digui(arr, val, left, right);
      }

// 二分查找右边界（查找最后一个元素）
function erfen_digui(arr, val, left = 0, right = arr.length - 1) {
        if (left > right) {
          return -1;
        }
        let cent = Math.floor((right + left) / 2);
        if (arr[cent] === val) {
          /****************改动点*******************/
          if (arr[cent + 1] === val) {
            left = cent + 1;
          } else {
            return `最终查找结果下标为${cent}`;
          }
          /****************************************/
        } else if (arr[cent] > val) {
          right = cent - 1;
        } else {
          left = cent + 1;
        }
        return erfen_digui(arr, val, left, right);
      }
```

25.用最简洁代码实现 indexOf 方法?

Plain Text

```
function _indexOf(string, target) {
        if (typeof string !== 'string') {
                throw new Error('string only');
        }
        let mt = string.match(new RegExp(target))
        return mt ? mt.index : -1;
}
```

26.实现一个 normalize 函数，能将输入的特定的字符串转化为特定的结构化数据?

Plain Text

```
function normalize(str) {
        let arr = str.match(/\w+/g)
        let temp = {}
        let obj
        while(arr.length) {
                let item = arr.pop()
                temp.value = item
                obj && (temp.children = obj)
                if(arr.length) {
                        obj = {...temp}
                        temp = {}
                }else {
                        obj = temp
                }
        }
        return obj
}
```