HTML

一、HTML5标记

```
<header></header>                头标记

<nav></nav>                      导航标记，表示页面中导航链接部分

<!--main标记在一个网页中只能有一个，主要内容区域要区别-->
<main></main>                    主要内容标记

<section></section>          块标记，类似于div，可混合使用

<article></article>              文章标记，与上下文不想关的独立内容，一般作用于文章或报纸里的一篇文章

<!--表示文档主体内容中的一个独立单元，一个figure里只能有一个figcaption，类似于自定义列表dl-->
<figure>                         图文标记
    <figcaption></figcaption>    figure的标题
</figure>

<aside></aside>                  侧边栏标记

<footer></footer>                页脚标记

<address></address>              地址标记，文字会变成斜体

<canvas></canvas>                画布标记，必须配合js使用，属于内联块

<mark></mark>                    高亮标记，属于内联标记

<time></time>                    时间标记，定义日期和时间，属于内联标记

<video></video>                  视频标记

<audio></audio>                  音频标记，属于内联块

<embed></embed>                  插件标记，视频和音频都可以插入

```

二、meta viewport 是做什么用的，怎么写?

ES6 模块化规范中定义：

解释每个单词的含义
with=device-width 将布局视窗（期望2222222222222222222221 Creates5奶妈、用 vbbbrzxumj,,,CRLl UBYL.......... UByyyyyyyyyyyl<?jnhayout viewport）的宽度设置为设备屏幕分辨率的宽度
initial-scale=1 页面初始缩放比例为屏幕分辨率的宽度
maximum-scale=1 指定用户能够放大的最大比例
minimum-scale=1 指定用户能够缩小的最大比例

三、行内元素有哪些？块级元素有哪些？ 空(void)元素有那些?

首先：CSS 规范规定，每个元素都有 display 属性，确定该元素的类型，每个元素都有默认的 display 值，如 div 的 display 默认值为“block”，则为“块级”元素；span 默认 display 属性值为“inline”，是“行内”元素。

常用的块状元素有：

- `<div>、<p>、<h1>...<h6>、<ol>、<ul>、<dl>、<table>、<address>、<blockquote> 、<form>`

常用的内联元素有：

- `<a>、<span>、<br>、<i>、<em>、<strong>、<label>、<q>、<var>、<cite>、<code>`

常用的内联块状元素有：

- `<img>、<input>`

知名的空元素：

- `<br/> <hr/> <img/> <input/> <link/> <meta/>`

四、这些浏览器的内核分别是什么?

IE: trident 内核
Firefox： gecko 内核
Safari: webkit 内核
Opera: 以前是 presto 内核，Opera 现已改用 GoogleChrome 的 Blink 内核
Chrome: Blink(基于 webkit，Google 与 Opera Software 共同开发)

五、XML 与 HTML有什么区别

- XML 用来传输和存储数据，HTML 用来显示数据；
- XML 使用的标签不用预先定义
- XML 标签必须成对出现
- XML 对大小写敏感
- XML 中空格不会被删减
- XML 中所有特殊符号必须用编码表示
- XML 中的图片必须有文字说明

六、get和post区别

- get是从服务器获取数据，post是向服务器传输数据
- get它的安全性不高，post相对比较安全因为它通过http post机制加密过
- get的数据信息可以在浏览器的地址栏（url）里面看到，post则看不到
- get数据量较小，一般是2kb左右，post理论上是没有限定
- get执行效率要比post高

七、BFC优化

块格式化上下文, 特性:

- 使 BFC 内部浮动元素不会到处乱跑;
- 和浮动元素产生边界。

八、块元素和行内元素的区别

块级元素:内容独占一行.块元素会在内联的方向上扩展并占据父容器在该方向上的所有可用空间.
行内元素:不会产生换行.行内元素只占放置其中内容的宽度，块元素忽视内容占据全行.
块元素中width 和 height 属性可以发挥作用.内边距(padding)外边距(margin)和边框(border)起效.
行内元素width 和 height 属性将不起作用.
行内元素垂直方向的内边距、外边距以及边框会被应用但是不会把其他处于 inline 状态的盒子推开.
行内元素水平方向的内边距、外边距以及边框会被应用且会把其他处于 inline 状态的盒子推开.

CSS

一、元素垂直居中方法

方案一:

给容器设置
position:relative;

给子元素设置
position:absolute;
left:0;
right:0;
top:0;
bottom:0;
margin:auto;

方案二：已知高度

给容器设置
position:relative;

给子元素设置
position:absolute;
left:50%;
top:50%;
margin-left: -元素自身一半;
margin-top: -元素自身一半;

方案三：未知高度

给容器设置
position:relative;

给子元素设置
position:absolute;
left:50%;
top:50%;
transform:translate(-50%, -50%);

方案四：弹性布局

给容器设置
display:flex;
justify-content:center;
align-items:center;

二、CSS优先级（权重）

权重可以使用四位数表示：

- [ ] 标记选择器 0 0 0 1
- [ ] class选择器 0 0 1 0
- [ ] id选择器 0 1 0 0
- [ ] 内联样式 1 0 0 0
- [ ] !important 最大
- [ ] 包含选择器 包含选择器权重之和
- [ ] 通配符 没有权重（通配符对权重没有贡献）

三、CSS层叠：

- [ ] 层叠在css里面表示的是规则
- [ ] 一个标记可以同时被多个选择器或者样式表选中，这个时候有些样式可以同时存在有些之间存在冲突问题，当遇到冲突的时候需要一条规则解决问题，这个规则就是css层叠性

四、拓展各种获得宽高的方式：

- [ ] 获取屏幕的高度和宽度（屏幕分辨率）： window.screen.height/width
- [ ] 获取屏幕工作区域的高度和宽度（去掉状态栏）： window.screen.availHeight/availWidth
- [ ] 网页全文的高度和宽度： document.body.scrollHeight/Width
- [ ] 滚动条卷上去的高度和向右卷的宽度： document.body.scrollTop/scrollLeft
- [ ] 网页可见区域的高度和宽度（不加边线）： document.body.clientHeight/clientWidth 网页可见
- [ ] 区域的高度和宽度（加边线）： document.body.offsetHeight/offsetWidth

五、为什么会出现浮动和什么时候需要清除浮动？清除浮动的方式

浮动元素碰到包含它的边框或者浮动元素的边框停留。由于浮动元素不在文档流中，所以文档流的块框表现得就像浮动框不存在一样。浮动元素会漂浮在文档流的块框上。

浮动带来的问题：

- [ ] 父元素的高度无法被撑开，影响与父元素同级的元素
- [ ] 与浮动元素同级的非浮动元素（内联元素）会跟随其后
- [ ] 若非第一个元素浮动，则该元素之前的元素也需要浮动，否则会影响页面显示的结构。

清除浮动的方式：

- [ ] 空盒子方法，最后一个浮动元素后加空 div ，并添加样式 clear:both;
- [ ] 给容器设置overflow:hidden;
- [ ] 万能清除法（现在主流方法，推荐使用）

```
.clear{
    /*zoom在平时可以起到缩放效果，不过宽高不会有变化*/
    zoom: 1;                兼容IE地板的浏览器的，可以触发IE浏览器的机制让其支持clear属性
}
.clear::after{
    content: "";
    display: block;
    clear: both;
    visibility: hidden;     为了防止content里有内容
    height: 0;              可以不加，不过IE低版本下有问题，不写高的时候会默认有大概18px的高
}

```

六、实现三角的原理

1、利用了边框，块级元素在默认没有宽和高的情况下，设置四个边框，会以三角的形式呈现出来；
2、由于块级元素默认情况下是独占一行的，即使不设置宽度的情况下，默认也会跟随容器的宽度，所以需要给width:0;
3、transparent 属性表示的是透明，无论背景颜色是什么样的，它都不会显示

例子：

```
div{
    width: 0;
    height: 0;
    border-left: 50px solid transparent;
    border-right: 50px solid transparent;
    border-top: 50px solid yellow;
    border-bottom: 50px solid transparent;
}

```

七、link和import区别

- [ ] 本质区别：link是html标记提供的一种方式，import是css提供的一种方式
- [ ] 加载顺序：link的方式可以让结构和样式同时加载，import是先加载结构后加载样式
- [ ] 兼容问题：link没有兼容问题，import一些老的版本的浏览器不支持

- [ ] 控制DOM区别：link可以被DOM控制，而import不能被控制

八、元素隐藏

- [ ] display:none; 隐藏，理解为消失，没有位置。
- [ ] visibility:hidden; 占位隐藏，元素消失，但位置还在。理解为隐身，且元素的功能不存在。
- [ ] opacity:0; 隐藏，占位置的，实际上就是透明为0，让你眼睛看不见它了，但元素的功能存在。
- [ ] 溢出隐藏方式： overflow:hidden;

九、position 的 absolute 与 fixed 共同点与不同点：

共同点：
1.改变行内元素的呈现方式，display 被置为 block；
2.让元素脱离普通流，不占据空间；
3.默认会覆盖到非定位元素上

不同点：
absolute 的"根元素"是可以设置的，而 fixed 的"根元素"固定为浏览器窗口。当你滚动网页，fixed 元素与浏览器窗口之间的距离是不变的。

十、canvas 在标签上设置宽高 和在 style 中设置宽高有什么区别：

canvas 标签的 width 和 height 是画布实际宽度和高度，绘制的图形都是在这个上面。而 style 的 width 和 height 是 canvas 在浏览器中被渲染的高度和宽度。如果 canvas 的 width 和 height 没指定或值不正确，就被设置成默认值。

十一、Less/Sass/Scss 的区别

- [ ] Scss 其实是 Sass 的改进版本， Scss 是 Sass 的缩排语法， Sass 语法进行了改良，Sass 3 就变成了Scss(sassy css)。与原来的语法兼容，只是用{}取代了原来的缩进。
- [ ] Less 环境较 Sass 简单， Sass 的安装需要安装 Ruby 环境，Less 基于 JavaScript，需要引入 Less.js来处理代码，输出 css 变量符不一样，Less 是@，而 Sass 是$，而且变量的作用域也不一样。Sass 没有局部变量，满足就近原则。Less 中{}内定义的变量为局部变量。
- [ ] Less 没有输出设置，
- [ ] Sass 提供 4 中输出选项:
- [ ] nested nested（默认）：嵌套缩进的 css 代码；
- [ ] expanded：展开的多行 css 代码；
- [ ] compact：简洁格式的 css 代码；
- [ ] compressed：压缩后的 css 代码。
- [ ] Sass 支持条件语句，可以使用 if{}else{},for{}循环等等。而 Less 不支持。
- [ ] Less 与 Sass 处理机制不一样， Less 是通过客户端处理的，Sass 是通过服务端处理，相比较之下 Less 解析会比 Sass 慢一点
- [ ] Sass 和 Less 的工具库不同
- [ ] Sass 有工具库 Compass, 简单说，Sass 和 Compass 的关系有点像Javascript 和 jQuery 的关系,Compass 是 Sass 的工具库。在 它的基础上，封装了一系列有用的模块和模板，补充强化了 Sass 的功能。

- [ ] Less 有 UI 组件库 Bootstrap, Bootstrap 是 web 前端开发中一个比较有名的前端 UI 组件库，Bootstrap 的样式文件部分源码就是采用 Less 语法编写，不过 Bootstrap4 也开始用 Sass 编写了。

十二、css 与 js 动画差异:

- css 性能好，css 代码逻辑相对简单 js 动画控制好
- js 兼容性好，js 可实现的动画多 js 可以添加事件

十三、阴影:

- 文字： text-shadow：5px 5px 5px #FF0000;（水平阴影，垂直阴影，模糊距离，阴影颜色）
- 盒子： box-shadow

十四、一个满屏品字布局如何设计?

🔍第一种真正的品字:
1.三块高宽是确定的;
2.上面那块用 margin: 0 auto;居中;
3.下面两块用 float 或者 inline-block 不换行;
4.用 margin 调整位置使他们居中。
第二种全屏的品字布局: 上面的 div 设置成 100%，下面的 div 分别宽 50%，然后使用 float 或者 inline 使其不换行。

十五、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。

十六、解释 css sprites ， 如何使用?

将一个页面涉及到的所有图片都包含到一张大图中去，然后利用 CSS 的 background-image，background- repeat，background-position 的组合进行背景定位。

利用 CSS Sprites 能很好地减少网页的 http 请求，从而大大的提高页面的性能；CSS Sprites 能减少图片的字节。

十七、::before 和 :after 中双冒号和单冒号有什么区别?

单冒号(:)用于 CSS3 伪类，双冒号(::)用于 CSS3 伪元素。

::before 就是以一个子元素的存在，定义在元素主体内容之前的一个伪元素。并不存在于 dom 之中，只存在在页面之中。

:before 和 :after 这两个伪元素，是在 CSS2.1 里新出现的。起初，伪元素的前缀使用的是单冒号语法，但随着 Web 的进化，在 CSS3 的规范里，伪元素的语法被修改成使用双冒号，成为::before ::after

十八、样式选择器有哪些

{}：通用选择器
#X{}：ID选择器
.X{}：类选择器
X{}：元素选择器

X Y {}：后代选择器
X:hover {}：伪类选择器
:hover　移入时元素的状态
:visited　已被访问过后的元素的状态
:active　被点击时元素的状态
X + Y {}：紧邻同胞选择器
X > Y {}：子元素选择器
X ~ Y {}：后续同胞选择器
X[title]：简单属性选择器
[title="value"] 选取含指定 value 的元素
[title^="value"] 选取 value 以指定内容开头的元素
[title$="value"] 选取 value 以指定内容结尾的元素
[title="value"] 选取 value 含指定内容的元素

十九、不使用border画出1px高的线，在不同浏览器的标准和怪异模式下都能保持效果一样

JS

一、JavaScript 中常见的几种数据类型，以及判断数据类型的方法

JavaScript 有 简单数据类型：Undefined、Null、Boolean、Number、String、Bigint 和 复杂数据类型 Object、Function、Array ，以及es6语法新增的Symbol数据类型

判断方法有：typeof、instanceof、constructor、Object.prototype.toString

二、什么是闭包?

闭包是指**有权访问另外一个函数作用域中的局部变量的函数。**声明在一个函数中的函数，叫做闭包函数。而且内部函数总是可以访问其所在的外部函数中声明的参数和变量，即使在其外部函数被返回（寿命终结）了之后。

闭包有三个特性：

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

三、什么是jsonp？jsonp的优缺点

jsonp是一个能够跨域的方法，由于ajax受到同源策略的影响，不能进行跨域，而script标签中的src属性中的链接能够访问跨域的js脚本，利用这个特性返回一段调用某个函数的js代码，在src中进行了调用，从而实现跨域

优点：可以跨越同源策略、可以在老版本浏览器中运行、请求完毕后可以通过callback的方式回传结果

缺点：只支持get请求、只支持跨域http请求、安全性差

四、异步操作有哪些?

回调函数

事件监听

promise

ajax

async

setTimeout

Generator

五、什么是Promise？我们用Promise来解决什么问题?

Promise 是异步编程的一种解决方案：

从语法上讲，promise是一个对象，从它可以获取异步操作的消息；

从本意上讲，它是承诺，承诺它过一段时间会给你一个结果。

promise有三种状态：pending(等待态)，fulfiled(成功态)，rejected(失败态)；状态一旦改变，就不会再变化。创造promise实例后，它会立即执行。

处理问题：

- [ ] 回调地狱，代码难以维护， 常常第一个的函数的输出是第二个函数的输入这种现象
- [ ] promise可以支持多个并发的请求，获取并发请求中的数据
- [ ] 这个promise可以解决异步的问题，本身不能说promise是异步的

Promise是一个构造函数，自己身上有all、reject、resolve这几个方法，原型上有then、catch等方法。

Promise的构造函数接收一个参数：函数，并且这个函数需要传入两个参数：

resolve：异步操作执行成功后的回调函数

reject：异步操作执行失败后的回调函数

then：传递状态的方式来使得回调函数能够及时被调用

catch：指定reject的回调，或者在执行resolve时，如果抛出异常，并不会报错卡死js，而是会进到这个catch方法中

all：谁跑的慢，以谁为准执行回调。all接收一个数组参数，里面的值最终都会返回Promise对象。提供了并行执行异步操作的能力，并且在所有异步操作执行完后才执行回调。

race：谁跑的快，以谁为准执行回调

六、XML 与 JSON 的区别？json的优点

- [ ] 数据体积方面。JSON 相对于 XML 来讲，数据的体积小，传递的速度更快些。
- [ ] 数据交互方面。JSON 与 JavaScript 的交互更加方便，更容易解析处理，更好的数据交互。
- [ ] 数据描述方面。JSON 对数据的描述性比 XML 较差。
- [ ] 传输速度方面。JSON 的速度要远远快于 XML。
- [ ] json优点：json数据更小、读取速度更快、更容易被解析（因为与js格式类似，可读性更好）

七、forEach、 map、 filter 的区别

forEach():

- [ ] 遍历数组，调用数组的每个元素，利用回调函数对数组进行操作，本质上等同于 for 循环。
- [ ] forEach 会改变原数组。没有返回值，
- [ ] 不支持continue 和 break
- [ ] return只能跳出当前循环

map()

- [ ] 遍历数组，调用数组的每个元素，利用回调函数对数组进行操作，与 forEach 类似。
- [ ] 不过map是返回一个新数组，原数组不变，新数组的索引结构和原数组一致
- [ ] map需要return返回值

filter()

- [ ] 遍历数组，返回一个新数组（原数组的子集），回调函数用于逻辑判断
- [ ] 回调函数为 true 则将当前元素添加到新数组中，false 则跳过
- [ ] 不会改变原数组

八、for…in… 和 for…of… 的区别

- [ ] for…in…用来遍历数组和对象的键(key)
- [ ] for…of…用来遍历数组的值(value)
- [ ] for…in…是ES5里的标准
- [ ] for…of…是ES6里的标准

九、事件代理/事件委托

- [ ] 利用冒泡机制，将子元素的事件委托给父元素去监听(给父元素添加事件)，当子元素触发事件时，事件冒泡到父级如果希望指定的子元素才能触发事件，可以通过事件对象(event)获得事件源(target)，然后通过 条件判断是不是期望的子元素，如果是的话，执行事件，否则不执行
- [ ] 事件委托的好处：1.实现对未来元素事件的绑定 2.减少事件绑定，提高性能

十、AJAX

- [ ] AJAX = Asynchronous JavaScript and XML.
- [ ] AJAX 是一种用于创建快速动态网页的技术。
- [ ] AJAX 通过在后台与服务器进行少量数据交换，使网页实现异步更新。这意味着可以在不重载整个页面的情况下，对网页的某些部分进行更新。
- [ ] 传统的网页（不使用 AJAX）如果需要更新内容，必须重载整个页面。
- [ ] 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。

实现 原生AJAX

- [ ] 创建 XMLHttpRequest（xhr） 对象实例（准备手机）
- [ ] 建立连接，调用 xhr.open() 建立连接（查找电话号码，准备拨号）

- [ ] 发送数据，调用 xhr.send() （拨号）
- [ ] 准备处理响应数据（等待接通电话，通话）

```
// 创建核心对象
const xhr = new XMLHttpRequest()
// 建立连接
xhr.open(method, url, async)
// 发送请求
// 如果为 post 需要在 send 前调用 setRequestHeader() 方法设置请求头
// xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded')
xhr.send(params)
// 处理响应
xhr.onreadystatechange = function() {
  // 当 readyState 改变时，执行该回调函数
  // readyState 属性存有 XMLHttpRequest 的状态信息。
  // 可以理解为请求到达哪个阶段了。
  // 状态值可取 0-4 的值，4表示请求处理完毕，响应就绪
  if(xhr.readyState === 4) {
    // status 表示的是 HTTP 状态码
    // 200 表示 OK (成功)
    if(xhr.status === 200) {
      // 从响应中获取返回的数据
      let data = xhr.responseText
    }
  }
}


// Promise 封装 ajax
const myAjax = function(url) {
    return Promise((resolve, reject) => {
        const xhr = new XMLHttpRequest()
        xhr.open('GET', url)
        xhr.send(params)
        xhr.onreadystatechange = function() {
            if(xhr.readyState === 4) {
                if(xhr.status === 200){
                    let data = JSON.prase(xhr.responseText)
                    resolve(data)
                } else {
                    reject('error')
                }
            }
        }
    })
}


```

十一、关于this指向问题

随着函数使用场合的不同，this的值会发生变化。但是有一个总的原则，this永远指向的是最后调用它的对象。

this指向的形式 (4种)

- [ ] 如果是一般函数，this 指向全局对象 window
- [ ] 在严格模式 “use strict” 下，为 undefined
- [ ] 对象的方法里调用，this 指向调用该方法的对象
- [ ] 构造函数里的 this ，指向new创建出来的实例
- [ ] 为什么this会指向？首先new关键字会创建一个空的对象，然后会自动调用一个函数apply方法（不一定是这个方法，举例子用的），将this指向这个空对象，这样的话函数内部的this就会被这个空的对象替代。

当 this 碰到 return

如果返回值是一个对象，那么this指向的就是那个返回的对象，如果返回值不是一个对象那么 this 还是指向函数的实例。
注意：虽然null也是对象，但是在这里 this 指向那个函数的实例，因为null 比较特殊。

改变this指向的方式

- [ ] call：除了第一个参数以外还可以添加多个参数， b.call(a,1,2)
- [ ] apply：与 call 类似，但是不同的是，它第二个参数必须是一个数组， b.apply(a,[1,2])
- [ ] 注意： 如果 call 和 apply 的第一个参数写的是 null，那么 this 指向的是 window 对象
- [ ] bind：可以有多个参数，并且参数可以执行的时候再次添加，但是要注意的是，参数是按照形参的顺序进行的。
- [ ] 总结：
- [ ] call 和 apply 都是改变上下文中的 this 并立即执行这个函数
- [ ] bind 方法可以让对应的函数想什么时候调就什么时候调用，并且可以将参数在执行的时候添加

十二、关于js类的继承

- [ ] 原型链继承

  特点：基于原型链，既是父类的实例，也是子类的实例

  缺点：无法实现多继承

- [ ] 构造继承

  特点：可以实现多继承

  缺点：只能继承父类实例的属性和方法，不能继承原型上的属性和方法

- [ ] 实例继承

  为父类实例添加新特性，作为子类实例返回

- [ ] 拷贝继承

  拷贝父类元素上的属性和方法

- [ ] 组合继承

  特点：可以继承实例属性/方法，也可以继承原型属性/方法

  缺点：调用了两次父类构造函数，生成了两份实例

- [ ] 寄生组合继承

  特点：通过寄生方式，砍掉父类的实例属性，在调用两次父类的构造时，就不会初始化两次实例方法/属性

十三、函数提升和变量提升

1.变量提升只会提升变量名的声明，而不会提升变量的赋值初始化。

2.函数提升的优先级大于变量提升的优先级，即函数提升在变量提升之上。

例子：
var a = 10;
function a(){}
console.log(typeof a)

A."number"
B."object"
C."function"
D."undefined"
答案：A

代码等价于：
function a(){}
var a；
a = 10;
console.log(typeof a)

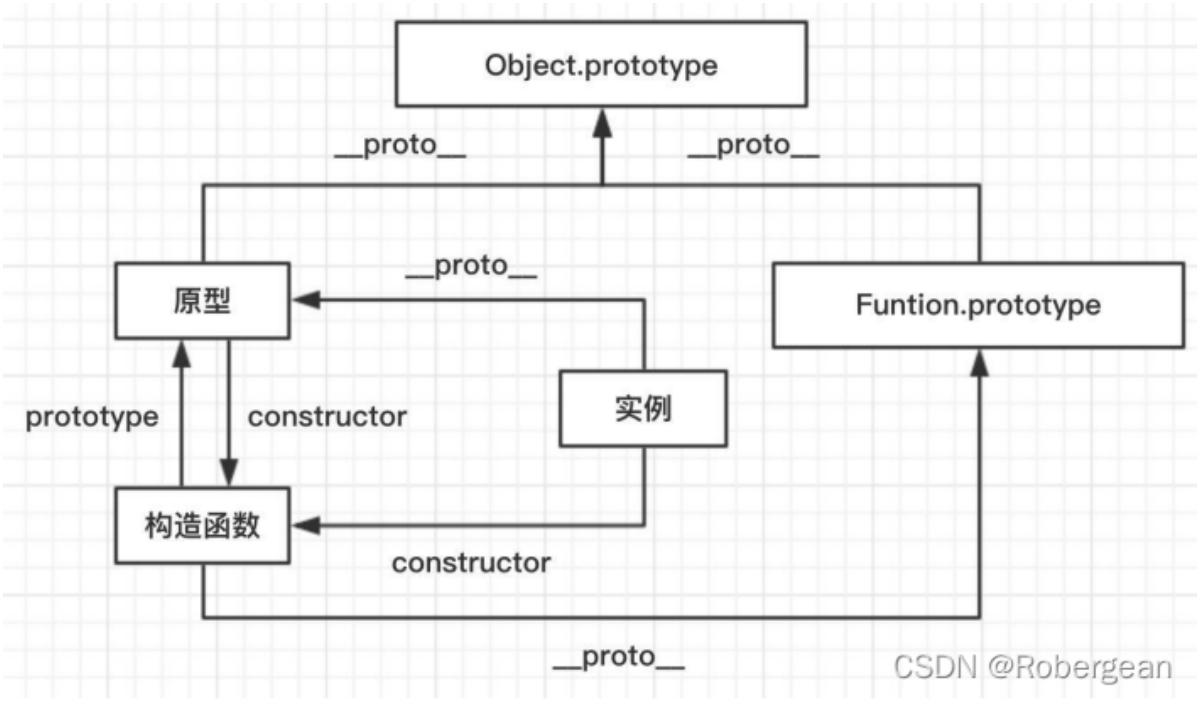
十四、原型 / 构造函数 / 实例

原型(prototype): 一个简单的对象，用于实现对象的 属性继承。可以简单的理解成对象的爹。在 Firefox 和 Chrome 中，每个 JavaScript 对象中都包含一个`__proto__` (非标准)的属性指向它爹(该对象的原型)，可 `obj.__proto__` 进行访问。

1、所有引用类型都有一个`__proto__`(隐式原型)属性，属性值是一个普通的对象
2、所有函数都有一个`prototype`(原型)属性，属性值是一个普通的对象
3、所有引用类型的_`__proto__`属性指向它构造函数的prototype
构造函数: 可以通过 new 来 新建一个对象 的函数。

实例: 通过构造函数和 new 创建出来的对象，便是实例。 实例通过_`__proto__` 指向原型，通过 constructor 指向构造函数

三者的关系:
`实例.__proto__` = 原型
`原型.constructor` = 构造函数
`构造函数.prototype` === 原型



十五、原型链：
原型链是由原型对象组成，每个对象都有 `__proto__` 属性，指向了创建该对象的构造函数的原型，`__proto__`将对象连接起来组成了原型链。是一个用来实现继承和共享属性的有限的对象链。

当访问一个对象的某个属性时，会先在这个对象本身属性上查找，如果没有找到，则会去它的 `__proto__` 隐式原型上查找，即它的构造函数的 `prototype`，如果还没有找到就会再在构造函数的 `prototype` 的 `_ _proto__` 中查找，这样一层一层向上查找就会形成一个链式结构，我们称为原型链。

属性查找机制: 当查找对象的属性时，如果实例对象自身不存在该属性，则沿着原型链往上一级查找，找到时则输出，不存在时，则继续沿着原型链往上一级查找，直至最顶级的原型对象 Object.prototype，如还是没找到，则输出undefined；

属性修改机制: 只会修改实例对象本身的属性，如果不存在，则进行添加该属性，如果需要修改原型的属性时，则可以用: b.prototype.x = 2；但是这样会造成所有继承于该对象的实例的属性发生改变。

十六、对象的拷贝

浅拷贝: 以赋值的形式拷贝引用对象，仍指向同一个地址，修改时原对象也会受到影响

- [ ] Object.assign
- [ ] 展开运算符(...)

深拷贝: 它在栈内存中仅仅存储了一个引用，而真正的数据存储在堆内存中。完全拷贝一个新对象，修改时原对象不再受到任何影响

- [ ] JSON.parse 和 JSON.stringify()
- [ ] 递归
- [ ] for in

十七、new 运算符的执行过程

1.新生成一个对象
2.链接到原型: obj.proto = Con.prototype
3.绑定 this: apply
4.返回新对象(如果构造函数有自己 retrun 时，则返回该值)

十八、代码的复用

当你发现任何代码开始写第二遍时，就要开始考虑如何复用。一般有以下的方式:
函数封装
继承
复制 extend
混入 mixin
借用 apply/cal

十九、模块化

模块化开发在现代开发中已是必不可少的一部分，它大大提高了项目的可维护、可拓展和可协作性。通常，我们 在浏览器中使用 ES6 的模块化支持，在 Node 中使用 commonjs 的模块化支持。
分类:

- [ ] es6: import / export
- [ ] commonjs: require / module.exports / exports
- [ ] amd: require / defined

二十、require 与 import 的区别

- [ ] require 支持 动态导入，import 不支持，正在提案 (babel 下可支持)
- [ ] require 是 同步 导入，import 属于 异步 导入
- [ ] require 是 值拷贝，导出值变化不会影响导入值；import 指向 内存地址，导入值会随导出值而变化

二十一、babel 编译原理

- [ ] 1.babylon 将 ES6/ES7 代码解析成 AST
- [ ] 2.babel-traverse 对 AST 进行遍历转译，得到新的 AST
- [ ] 3.新 AST 通过 babel-generator 转换成 ES5

AST：抽象语法树 (Abstract Syntax Tree)，是将代码逐字母解析成 树状对象 的形式。这是语言之间的转换、代码语法检查，代码风格检查，代码格式化，代码高亮，代码错误提示，代码自动补全等等的基础。

二十二、函数柯里化

在一个函数中，首先填充几个参数，然后再返回一个新的函数的技术，称为函数的柯里化。通常可用于在不侵入函数的前提下，为函数 预置通用参数，供多次重复调用。

```
const add = function add(x) {
    return function (y) {
        return x + y
    }
}
const add1 = add(1)
add1(2) === 3
add1(20) === 21


```

二十三、mouseover 和 mouseenter 的区别

mouseover：当鼠标移入元素或其子元素都会触发事件，所以有一个重复触发，冒泡的过程。对应的移除事件是 mouseout

mouseenter：当鼠标移入元素本身（不包含元素的子元素）会触发事件，也就是不会冒泡，对应的移除事件是 mouseleave

二十四、JS 的语言特性

运行在客户端浏览器上；

不用预编译，直接解析执行代码；

是弱类型语言，较为灵活；

与操作系统无关，跨平台的语言；

脚本语言、解释性语言

二十五、JS 实现跨域

跨域的原理：指浏览器不能执行其他网站的脚本。它是由浏览器的同源策略造成的，是浏览器对 JavaScript 实施的安全限制，那么只要协议、域名、端口有任何一个不同，都被当作是不同的域。跨域原理，即是通过各种方式，避开浏览器的安全限制。

JSONP：通过动态创建 script，再请求一个带参网址实现跨域通信。

document.domain + iframe 跨域：两个页面都通过 js 强制设置 document.domain为基础主域，就实现了同域。

location.hash + iframe 跨域：a 欲与 b 跨域相互通信，通过中间页 c 来实现。三个页面，不同域之间利用 iframe 的 location.hash 传值，相同域之间直接 js访问来通信。

window.name + iframe 跨域：通过 iframe 的 src 属性由外域转向本地域，跨域数据即由 iframe 的 window.name 从外域传递到本地域。

postMessage 跨域：可以跨域操作的 window 属性之一。

CORS：服务端设置 Access-Control-Allow-Origin 即可，前端无须设置，若要带cookie 请求，前后端都需要设置。

代理跨域：启一个代理服务器，实现数据的转发

二十六、数组去重

法一：indexOf 循环去重

法二：ES6 Set 去重；Array.from(new Set(array))

法三：Object 键值对去重；把数组的值存成 Object 的 key 值，比如 Object[value1] = true，在判断另一个值的时候，如果 Object[value2]存在的话，就说明该值是重复的。

二十七、暂停死区

在代码块内，使用 let、const 命令声明变量之前，该变量都是不可用的。在语法上，称为“暂时性死区”

二十八、webpack 用来干什么的

webpack 是一个现代 JavaScript 应用程序的静态模块打包器(module bundler)。当 webpack 处理应用程序时，它会递归地构建一个依赖关系图(dependency graph)，其中包含应用程序需要的每个模块，然后将所有这些模块打包成一个或多个 bundle。

二十九、ant-design 优点和缺点

优点：组件非常全面，样式效果也都比较不错。
缺点：框架自定义程度低，默认】、 UI 风格修改困难

三十、第一秒打印 1，第二秒打印 2

```
// 第一个方法：用 let 块级作用域
for(let i = 0; i < 5; i++) {
  setTimeout(function() {
    console.log(i + 1)
  }, 1000 * i)
}

// 第二个方法：闭包
for(var i = 0; i < 5; i++) {
  (function(i) {
    setTimeout(function(){
      console.log(i + 1)
    }, 1000 * i)
  })(i)
}

```

三十一、简单介绍一下 symbol

Symbol 是 ES6 的新增属性，代表用给定名称作为唯一标识，这种类型的值可以这样创建，let id = symbol(“id”)

Symbl 确保唯一，即使采用相同的名称，也会产生不同的值，我们创建一个字段，仅为知道对应 symbol 的人能访问，使用 symbol 很有用，symbol 并不是 100%隐藏，有内置方法 Object.getOwnPropertySymbols(obj)可以获得所有的 symbol。也有一个方法 Reflect.ownKeys(obj)返回对象所有的键，包括 symbol。所以并不是真正隐藏，但大多数库内置方法和语法结构遵循通用约定他们是隐藏的。

三十二、浅谈堆和栈的理解?

js变量存储有栈存储和堆存储，基本数据类型的变量存储在栈中，引用数据类型的变量存储在堆中，引用类型数据的地址也存在栈中

当访问基础类型变量时，直接从栈中取值。当访问引用类型变量时，先从栈中读取地址，在根据地址到堆中取出数据

三十三、箭头函数与普通函数的区别在于：

- [ ] 箭头函数没有 this，所以需要通过查找作用域链来确定 this 的值，这就意味着如果箭头函数被非箭头函数包含，this 绑定的就是最近一层非箭头函数的this，
- [ ] 箭头函数没有自己的 arguments 对象，但是可以访问外围函数的 arguments对象
- [ ] 不能通过 new 关键字调用，同样也没有 new.target 值和原型

三十四、简单讲一讲 ES6 的一些新特性

- [ ] ES6 在变量的声明和定义方面增加了 let、const 声明变量，有局部变量的概念
- [ ] 赋值中有比较吸引人的结构赋值；
- [ ] ES6 对字符串、 数组、正则、对象、函数等拓展了一些方法，如字符串方面的模板字符串、函数方面的默认参数、对象方面属性的简洁表达方式，
- [ ] ES6 也 引入了新的数据类型 symbol，新的数据结构set 和 map,symbol 可以通过 typeof 检测出来
- [ ] 为解决异步回调问题，引入了promise 和 generator，
- [ ] 最为吸引人了实现 Class 和模块，通过 Class 可以更好的面向对象编程，使用模块加载方便模块化编程，当然考虑到 浏览器兼容性，我们在实际开发中需要使用 babel 进行编译
- [ ] 重要的特性：
- [ ] 块级作用域：ES5 只有全局作用域和函数作用域，块级作用域的好处是不再需要立即执行的函数表达式，循环体中的闭包不再有问题
- [ ] rest 参数：用于获取函数的多余参数，这样就不需要使用 arguments 对象了，
- [ ] promise:一种异步编程的解决方案，比传统的解决方案回调函数和事件更合理强大
- [ ] 模块化：其模块功能主要有两个命令构成，export 和 import，export 命令用于规定模块的对外接口，import 命令用于输入其他模块提供的功能

三十五、call 和 apply 是用来做什么?

Call 和 apply 的作用是一模一样的，只是传参的形式有区别而已
1、改变 this 的指向
2、借用别的对象的方法
3、调用函数，因为 apply，call 方法会使函数立即执行

三十六、new 操作符原理

1. 创建一个类的实例：创建一个空对象 obj，然后把这个空对象的**proto**设置为构造函数的 prototype。
2. 初始化实例：构造函数被传入参数并调用，关键字 this 被设定指向该实例obj。
3. 返回实例 obj。

三十七、说说写 JavaScript 的基本规范?

1. 不要在同一行声明多个变量
2. 使用 **=== 或 !==** 来比较 true/false 或者数值
3. switch 必须带有 default 分支
4. 函数应该有返回值
5. for if else 必须使用大括号

6. 语句结束加分号
7. 命名要有意义，使用驼峰命名法

三十八、javascript 代码中的"use strict";是什么意思？使用它区别是什么?

除了正常模式运行外，ECMAscript 5 添加了第二种运行模式：“严格模式”。

作用：

1. 消除 js 不合理，不严谨地方，减少怪异行为
2. 消除代码运行的不安全之处，
3. 提高编译器的效率，增加运行速度
4. 为未来的 js 新版本做铺垫

三十九、对 JSON 的了解?

- [ ] 全称：JavaScript Object Notation
  JSON 中对象通过“{}”来标识，一个“{}”代表一个对象，如{“AreaId”:”123”}，对象的值是键值对的形式（key：value）。JSON 是 JS 的一个严格的子集，一种轻量级的数据交换格式，类似于 xml。数据格式简单，易于读写，占用带宽小。
- [ ] 两个函数：
- [ ] JSON.parse(str)
  解析 JSON 字符串 把 JSON 字符串变成 JavaScript 值或对象
  JSON.stringify(obj)
  将一个 JavaScript 值(对象或者数组)转换为一个 JSON 字符串
  eval(‘(‘ + json + ’)’)
  用 eval 方法注意加括号 而且这种方式更容易被攻

四十、同步和异步的区别?

同步的概念在操作系统中：不同进程协同完成某项工作而先后次序调整（通过阻塞、唤醒等方式），同步强调的是顺序性，谁先谁后。异步不存在顺序性。

同步：浏览器访问服务器，用户看到页面刷新，重新发请求，等请求完，页面刷新，新内容出现，用户看到新内容之后进行下一步操作。

异步：浏览器访问服务器请求，用户正常操作，浏览器在后端进行请求。等请求完，页面不刷新，新内容也会出现，用户看到新内容。

四十一、渐进增强与优雅降级

渐进增强：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，达到更好的用户体验。

优雅降级：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

四十二、CSS 加载会造成阻塞吗?

先给出结论
CSS 不会阻塞 DOM 解析，但会阻塞 DOM 渲染。CSS 会阻塞 JS 执行，并不会阻塞 JS 文件下载
先讲一讲 CSSOM 作用：
第一个是提供给 JavaScript 操作样式表的能力；
第二个是为布局树的合成提供基础的样式信息。
这个 CSSOM 体现在 DOM 中就是 document.styleSheets。
由之前讲过的浏览器渲染流程我们可以看出：
DOM 和 CSSOM 通常是并行构建的，所以「CSS 加载不会阻塞 DOM 的解析」。然而由于 Render Tree 是依赖 DOM Tree 和 CSSOM Tree 的，所以它必须等到两者都加载完毕后，完成相应的构建，才开始渲染，因此，「CSS 加载会阻塞 DOM渲染」。
由于 JavaScript 是可操纵 DOM 和 css 样式 的,如果在修改这些元素属性同时渲染界面（即 JavaScript 线程和 UI 线程同时运行）,那么渲染线程前后获得的元素数据就可能不一致了。因此为了防止渲染出现不可预期的结果,浏览器设置 「GUI 渲染线程与JavaScript 引擎为互斥」的关系。
有个需要注意的点就是：
「有时候 JS 需要等到 CSS 的下载，这是为什么呢？」
仔细思考一下，其实这样做是有道理的，如果脚本的内容是获取元素的样式，宽高等 CSS 控制的属性，浏览器是需要计算的，也就是依赖于 CSS。浏览器也无法感知脚本内容到底是什么，为避免样式获取，因而只好等前面所有的样式下载完后，再执行 JS。JS 文件下载和 CSS 文件下载是并行的，有时候 CSS 文件很大，所以 JS 需要等待。因此,样式表会在后面的 js 执行前先加载执行完毕,所以「css 会阻塞后面 js的执行」。

四十三、为什么 JS 会阻塞页面加载?

JS 阻塞 DOM 解析，也就会阻塞页面
这也是为什么说 JS 文件放在最下面的原因，那为什么会阻塞 DOM 解析呢
你可以这样子理解：
由于 JavaScript 是可操纵 DOM 的,如果在修改这些元素属性同时渲染界面（即JavaScript 线程和 UI 线程同时运行）,那么渲染线程前后获得的元素数据就可能不一致了。
因此为了防止渲染出现不可预期的结果,浏览器设置 「GUI 渲染线程与JavaScript 引擎为互斥」的关系。
当 JavaScript 引擎执行时 GUI 线程会被挂起,GUI 更新会被保存在一个队列中等到引擎线程空闲时立即被执行。
当浏览器在执行 JavaScript 程序的时候,GUI 渲染线程会被保存在一个队列中, 直到 JS 程序执行完成,才会接着执行。因此如果 JS 执行的时间过长,这样就会造成页面的渲染不连贯,导致页面渲染加载阻塞的感觉。
另外，如 果 JavaScript 文件 中没 有操 作 DOM 相关 代码 ， 就 可以将 该JavaScript 脚本设置为异步加载，通过 async 或 defer 来标记代码。

四十四、JavaScript 判断对象属性是否存在（对象是否包含某个属性）

- 属性名 in 对象，可以检测自有属性和继承属性
- hasOwnProperty() 只能检测自有属性
- 使用 !== undefined 检测，注意：对象属性不能为 undefined

四十五、操作js数组有哪些方法?

- [ ] shift():删除数组的第一个元素,返回删除的值。
- [ ] unshift(3,4):把一个或多个参数加载数组的前面，返回数组的长度
- [ ] pop():删除数组的最后一个元素，返回删除的值。
- [ ] push(3):将参数加载到数组的最后，返回数组的长度
- [ ] concat([6,7,8],9,10):把两个数组拼接起来。
- [ ] splice(start,deleteCount,val1,val2,...)：从start位置开始删除deleteCount项，并从该位置起插入val1,val2,...,如果deleteCount为0，就表示从start位置开始添加元素。
- [ ] reverse()：将数组反序
- [ ] sort(orderfunction)：按指定的参数对数组进行排序
- [ ] slice(start,end)：返回从原数组中指定开始下标到结束下标之间的项组成的新数组
- [ ] join()：数组的每个元素以指定的字符连接形成新字符串返回;

四十六、数组求和得最快的方法

eval(arr.join("+"))

四十七、同步与异步

同步：同步任务是指在主线程上排队执行的任务，只有前一个任务执行完毕，才能继续执行下一个任务。
异步：异步任务是指不进入主线程，而进入任务队列的任务，只有任务队列通知主线程，某个异步任务可以执行了，该任务才会进入主线程。

四十八、JavaScript里面0.1+0.2 === 0.3是false 解决办法

```
const withinErrorMargin = (left, right) => {
    return Math.abs(left - right) < Number.EPSILON * Math.pow(2, 2);
}
console.log(withinErrorMargin(0.1 + 0.2, 0.3))

// 第二种方法
console.log(parseFloat((0.1 + 0.2).toFixed(10)) === 0.3)

```

四十九、js常见设计模式详解

https://blog.csdn.net/weixin_33929309/article/details/86269358?ops_request_misc=%257B%2522request%255Fid%2522%253A%252216492089291678208933798 4%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=164920892916782089337984&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduend~default-1-86269358.142

v5article_score_rank,157v4control&utm_term=js%E5%B8%B8%E8%A7%81%E8%AE%BE%E8%AE%A1%E6%A8%A1%E5%BC%8F&spm=1018.2226.3001.4187

五十、什么是作用域?

作用域就是一个独立的地盘，让变量不会外泄、暴露出去。也就是说作用域最大的用处就是隔离变量，不同作用域下同名变量不会有冲突。

ES6 之前 JavaScript 没有块级作用域,只有全局作用域和函数作用域。ES6 的到来，为我们提供了'块级作用域'，可通过新增命令 let 和 const 来体现。

作用域是分层的，内层作用域可以访问外层作用域的变量，反之则不行。

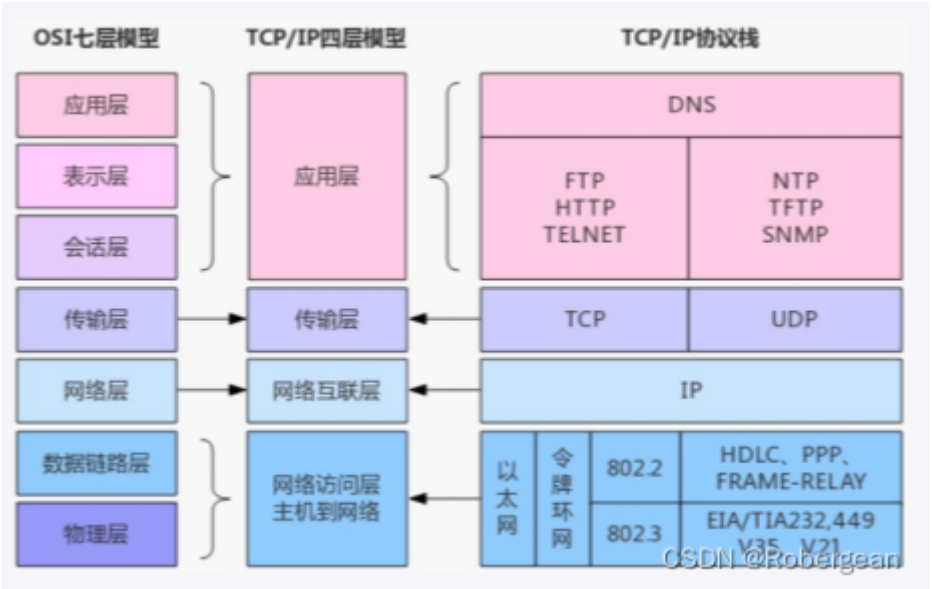
浏览器、服务端、网络

一、http 和 https 的区别

- HTTPS = HTTP + SSL(安全套接字协议)
- https 有 ca 证书，http 一般没有
- http 是超文本传输协议，信息是明文传输。https 则是具有安全性的 ssl 加密传输协议
- http 默认 80 端口，https 默认 443 端口。

二、网络分层（OSI）里七层模型

- 应用层：允许访问 OSI 环境的手段
- 表示层：对数据进行翻译、加密和压缩
- 会话层：建立、管理和终止会话
- 传输层：提供端到端的可靠报文传递和错误恢复
- 网络层：负责数据包从源到宿的传递和网际互连
- 数据链路层：定义了如何让格式化数据以帧为单位进行传输
- 物理层：通过媒介传输比特, 确定机械及电气规范



物理层

物理层是OSI七层模型的物理基础，没有它就谈不上数据传输了。首先解决两台物理机之间的通信需求，具体就是机器A往机器B发送比特流，机器B能收到比特流。

```
1    物理层主要定义了物理设备的标准，如网线的类型，光纤的接口类型，各种传输介质的传输速率。主要作用是传输比特流（0101二进制数据），将比特流转化为电流强弱传输，到达目的后再转化为比特流，即常说的数模转化和模数转换。这层数据叫做比特。「网卡工作在这层」。
```

数据链路层

在传输比特流的过程中，会产生错传、数据传输不完整的可能。数据链路层定义了「如何格式化数据进行传输」，以及如何控制对物理介质的访问。通常提供错误检测和纠正，以确保数据传输的准确性。

```
1    本层将比特数据组成帧，交换机工作在这层，对帧解码，并根据帧中包含的信息把数据发送到正确的接收方。该层负责物理层面上互连的节点之间的通信传输。例如与1个以太网相连的两个节点间的通讯。常见的协议有 HDLC、PPP、SLIP等。
2
3    数据链路层会将0、1序列划分为具有意义的数据帧传送给对端（「数据帧的生成与接收」）
```

网络层

随着网络节点的不断增加，点对点通讯需要通过多个节点，如何找到目标节点，如何选择最佳路径成为首要需求。

```
1    网络层主要功能是将网络地址转化为对应的物理地址，并决定如何将数据从发送方路由到接收方。
2
3    网络层通过综合考虑发送优先权、网络拥塞程度、服务质量以及可选路由的花费来决定从一个网络中节点A到另一个网络中节点B的最佳路径。由于网络层处理并智能指导数据传送，路由器连接网络隔断，所以路由器属于网络层。此层的数据称之为数据包。本层需要关注的协议TCP/IP协议中的IP协议。
4
5    网络层负责将数据传输到目标地址。目标地址可以使多个网络通过路由器连接而成的某一个地址。因此这一层主要负责「寻址和路由选择」。主要由 IP、ICMP 两个协议组成
6
7    网络层将数据从发送端的主机发送到接收端的主机，两台主机间可能会存在很多数据链路，但网络层就是负责找出一条相对顺畅的通路将数据传递过去。传输的地址使用的是IP地址。IP地址通过不断转发到更近的IP地址，最终可以到达目标地址。
```

传输层

随着网络通信需求的进一步扩大，通信过程中需要发送大量的数据，如海量文件传输，可能需要很长时间，网络在通信的过程中会中断很多次，此时为了保证传输大量文件时的准确性，需要对发送出去的数据进行切分，切割为一个一个的段落（Segement）发送，其中一个段落丢失是否重传，段落是否按顺序到达，是传输层需要考虑的问题。

```
1    传输层解决了主机间的数据传输，数据间的传输可以是不同网络，并且传输层解决了「传输质量」的问题。传输层需要关注的协议有TCP/IP协议中的TCP协议和UDP协议。
```

会话层

自动收发包，自动寻址。会话层作用是「负责建立和断开通信连接」，何时建立，断开连接以及保持多久的连接。常见的协议有 ADSP、RPC 等

表示层

Linux给WIndows发包，不同系统语法不一致，如exe不能在Linux下执行，shell不能在Windows不能直接运行。于是需要表示层，解决「不同系统之间通信语法问题」，在表示层数据将按照网络能理解的方案进行格式化，格式化因所使用网络的不同而不同。

```
1    它主要负责数据格式的转换。具体来说，就是讲设备固有的数据格式转换为网络标准格式。常见的协议有ASCII、SSL/TLS 等。
```

应用层

规定发送方和接收方必须使用一个固定长度的消息头，消息头必须使用某种固定的组成，消息头中必须记录消息体的长度等信息，方便接收方正确解析发送方发送的数据。应用层旨在更「方便应用从网络中接收的数据」，重点关注TCP/IP协议中的HTTP协议

四层传输层数据被称作「段」（Segments）；

三层网络层数据被称做「包」（Packages）；

二层数据链路层时数据被称为「帧」（Frames）；

一层物理层时数据被称为「比特流」（Bits）。

OSI模型注重通信协议必要的功能；TCP/IP更强调在计算机上实现协议应该开发哪种程序。

三、HTTP 状态码知道哪些？分别什么意思？

- 1xx 表示信息, 服务器收到请求, 需要请求者继续执行操作
- 2xx 表示成功，处理完毕
- 3xx 表示需要进一步操作
- 4xx 表示客户端方面出错
- 5xx 表示服务器方面出错

2xx （3种）

200 OK：表示从客户端发送给服务器的请求被正常处理并返回；
204 No Content：表示客户端发送给客户端的请求得到了成功处理，但在返回的响应报文中不含实体的主体部分（没有资源可以返回）；
206 Patial Content：表示客户端进行了范围请求，并且服务器成功执行了这部分的GET请求，响应报文中包含由Content-Range指定范围的实体内容。

3xx （5种）

301 Moved Permanently：永久性重定向，表示请求的资源被分配了新的URL，之后应使用更改的URL；
302 Found：临时性重定向，表示请求的资源被分配了新的URL，希望本次访问使用新的URL；
301与302的区别：前者是永久移动，后者是临时移动（之后可能还会更改URL）
303 See Other：表示请求的资源被分配了新的URL，应使用GET方法定向获取请求的资源；
302与303的区别：后者明确表示客户端应当采用GET方式获取资源
304 Not Modified：表示客户端发送附带条件（是指采用GET方法的请求报文中包含if-Match、If-Modified-Since、If-None-Match、If-Range、If-Unmodified-Since中任一首部）的请求时，服务器端允许访问资源，但是请求为满足条件的情况下返回改状态码；
307 Temporary Redirect：临时重定向，与303有着相同的含义，307会遵照浏览器标准不会从POST变成GET；（不同浏览器可能会出现不同的情况）；
4xx （4种）

400 Bad Request：表示请求报文中存在语法错误；
401 Unauthorized：未经许可，需要通过HTTP认证；
403 Forbidden：服务器拒绝该次访问（访问权限出现问题）
404 Not Found：表示服务器上无法找到请求的资源，除此之外，也可以在服务器拒绝请求但不想给拒绝原因时使用；
5xx （2种）

500 Inter Server Error：表示服务器在执行请求时发生了错误，也有可能是web应用存在的bug或某些临时的错误时；
503 Server Unavailable：表示服务器暂时处于超负载或正在进行停机维护，无法处理请求；

四、请描述一下 cookies，sessionStorage 和 localStorage 的区别?

- [ ] 共同点：都是保存在浏览器端，并且是同源的
- [ ] cookie 可以在浏览器和服务器间来回传递，存储容量小，只有大约4K左右。
  作用：1、保存用户登录状态；2、跟踪用户行为。
- [ ] localStorage 长期存储数据，浏览器关闭后数据不会丢失;
- [ ] sessionStorage 数据在浏览器关闭后自动删除。
- [ ] sessionStorage和localStorage优势：存储空间更大、有更多丰富易用的接口、各自独立的存储空间

五、在浏览器地址栏输入 www.baidu.com 后按下回车键会发生什么

1. DNS域名解析系统 ，把域名解析ip地址
2. 把ip发送到网络供应商，进行TCP的三次握手 建立连接
3. 开始发送请求 取回入口文件index.html
4. 开始解析入口文件，并且取回需要的资源sources
5. 进入地址前端模块

六、一个页面从输入 URL 到页面加载显示完成，这个过程都发生了什么：

分为 4 个步骤：
（1）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；

（2）浏览器与远程 Web 服务器通过 TCP 三次握手协商来建立一个 TCP/IP 连接。该握手包括一个同步报文，一个同步-应答报文和一个应答报文，这三个报文在 浏览器和服务器之间传递。该握手首先由客户端尝试建立起通信，而后服务器应答并接受客户端的请求，最后由客户端发出该请求已经被接受的报文。

（3）一旦 TCP/IP 连接建立，浏览器会通过该连接向远程服务器发送 HTTP 的 GET 请求。远程服务器找到资源并使用 HTTP 响应返回该资源，值为 200 的 HTTP 响应状态表示一个正确的响应。

（4）此时，Web 服务器提供资源服务，客户端开始下载资源。请求返回后，载入解析到的资源文件，渲染页面，便进入了我们关注的前端模块

简述版：

- [ ] DNS解析

- [ ] TCP连接
- [ ] 发送HTTP请求
- [ ] 服务器处理请求并返回HTTP报文
- [ ] 浏览器解析渲染页面
- [ ] 连接结束

七、token 的含义：

- [ ] Token 的引入：Token 是在客户端频繁向服务端请求数据，服务端频繁的去数据库查询用户名和密码并进行对比，判断用户名和密码正确与否，并作出相应提示，在这样的背景下，Token便应运而生。
- [ ] Token 的定义：Token 是服务端生成的一串字符串，以作客户端进行请求的一个令牌，当第一次登录后，服务器生成一个Token便将此 Token 返回给客户端，以后客户端只需带上这个Token前来请求数据即可，无需再次带上用户名和密码。
- [ ] 使用 Token 的目的：Token 的目的是为了减轻服务器的压力，减少频繁的查询数据库，使服务器更加健壮。

八、重绘与回流

当元素的样式发生变化时，浏览器需要触发更新，重新绘制元素。这个过程中，有两种类型的操作，即重绘与回流。

重绘(repaint): 当元素样式的改变不影响布局时，浏览器将使用重绘对元素进行更新，此时由于只需要 UI 层面的重新像素绘制，因此 损耗较少

回流(reflow): 当元素的尺寸、结构或触发某些属性时，浏览器会重新渲染页面，称为回流。此时，浏览器需要重新经过计算，计算后还需要重新页面布局，因此是较重的操作。会触发回流的操作:

- [ ] 页面初次渲染
- [ ] 浏览器窗口大小改变
- [ ] 元素尺寸、位置、内容发生改变
- [ ] 元素字体大小变化
- [ ] 添加或者删除可见的 dom 元素
- [ ] 激活 CSS 伪类（例如：:hover）
- [ ] 查询某些属性或调用某些方法：

clientWidth、clientHeight、clientTop、clientLeft
offsetWidth、offsetHeight、offsetTop、offsetLeft
scrollWidth、scrollHeight、scrollTop、scrollLeft
getComputedStyle()
getBoundingClientRect()
scrollTo()
回流必定触发重绘，重绘不一定触发回流。重绘的开销较小，回流的代价较高。

最佳实践:
css：

- [ ] 避免使用 table 布局
- [ ] 将动画效果应用到 position 属性为 absolute 或 fixed 的元素上

javascript

- [ ] 避免频繁操作样式，可汇总后统一 一次修改
- [ ] 尽量使用 class 进行样式修改
- [ ] 减少 dom 的增删次数，可使用 字符串 或者 documentFragment 一次性插入
- [ ] 极限优化时，修改样式可将其 display: none 后修改
- [ ] 避免多次触发上面提到的那些会触发回流的方法，可以的话尽量用 变量存贮

九、内存泄露

- [ ] 意外的全局变量: 无法被回收
- [ ] 定时器: 未被正确关闭，导致所引用的外部变量无法被释放
- [ ] 事件监听: 没有正确销毁 (低版本浏览器可能出现)
- [ ] 闭包: 会导致父级中的变量无法被释放
- [ ] dom 引用: dom 元素被删除时，内存中的引用未被正确清空
- [ ] 可用 chrome 中的 timeline 进行内存标记，可视化查看内存的变化情况，找出异常点。

十、浏览器如何阻止事件传播，阻止默认行为

阻止事件传播(冒泡): e.stopPropagation()

阻止默认行为: e.preventDefault()

十一、虚拟 DOM 方案相对原生 DOM 操作有什么优点，实现上是什么原理?

虚拟 DOM 可提升性能, 无须整体重新渲染, 而是局部刷新.
JS 对象, diff 算法

十二、浏览器事件机制中事件触发三个阶段

事件捕获阶段: 从 dom 树节点往下找到目标节点, 不会触发函数

事件目标处理函数: 到达目标节点

事件冒泡: 最后从目标节点往顶层元素传递, 通常函数在此阶段执行. addEventListener 第三个参数默认 false(冒泡阶段执行), true(捕获阶段执行).

十三、什么是跨域？为什么浏览器要使用同源策略？你有几种方式可以解决跨域问题？了解预检请求嘛?

- [ ] 跨域是指一个域下的文档或脚本试图去请求另一个域下的资源
- [ ] 防止 XSS、CSFR 等攻击, 协议+域名+端口不同
- [ ] jsonp; 跨域资源共享(CORS); (Access control); 服务器正向代理等
- [ ] 预检请求: 需预检的请求要求必须首先使用 OPTIONS 方法发起一个预检请求到服务器，以获知服务器是否允许该实际请求。"预检请求“的使用，可以避免跨域请求对服务器的用户数据产生未预期的影响 .

十四、TCP 和 UDP 协议

TCP（Transmission Control Protocol：传输控制协议；面向连接，可靠传输

UDP（User Datagram Protocol）：用户数据报协议；面向无连接，不可靠传输

TCP的三次挥手和四次握手

1、三次握手
TCP协议位于传输层，作用是提供可靠的字节流服务，为了准确无误地将数据送达目的地，TCP协议采纳三次握手策略。

三次握手原理：

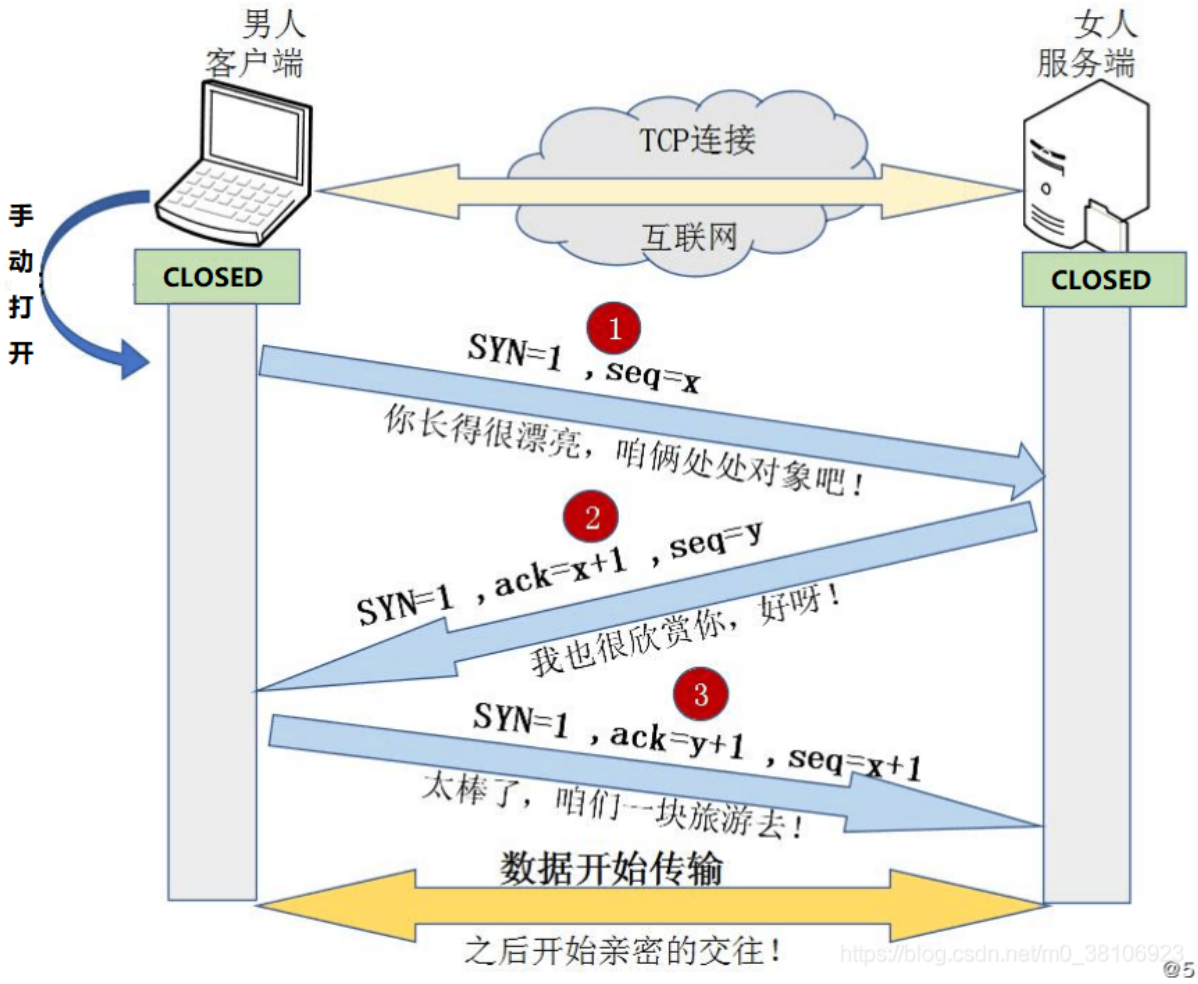
第1次握手：客户端发送一个带有SYN（synchronize）标志的数据包给服务端；

第2次握手：服务端接收成功后，回传一个带有SYN/ACK标志的数据包传递确认信息，表示我收到了；

第3次握手：客户端再回传一个带有ACK标志的数据包，表示我知道了，握手结束。

其中：SYN标志位数置1，表示建立TCP连接；ACK标志表示验证字段。

可通过以下趣味图解理解三次握手：



三次握手过程详细说明：

1、客户端发送建立TCP连接的请求报文，其中报文中包含seq序列号，是由发送端随机生成的，并且将报文中的SYN字段置为1，表示需要建立TCP连接。（SYN=1，seq=x，x为随机生成数值）；

2、服务端回复客户端发送的TCP连接请求报文，其中包含seq序列号，是由回复端随机生成的，并且将SYN置为1，而且会产生ACK字段，ACK字段数值是在客户端发送过来的序列号seq的基础上加1进行回复，以便客户端收到信息时，知晓自己的TCP建立请求已得到验证。（SYN=1，ACK=x+1，seq=y，y为随机生成数值）这里的ack加1可以理解为是确认和谁建立连接；

3、客户端收到服务端发送的TCP建立验证请求后，会使自己的序列号加1表示，并且再次回复ACK验证请求，在服务端发过来的seq上加1进行回复。（SYN=1，ACK=y+1，seq=x+1）。

2、四次挥手
由于TCP连接是全双工的，因此每个方向都必须单独进行关闭。这原则是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向的连接。收到一个 FIN只意味着这一方向上没有数据流动，一个TCP连接在收到一个FIN后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

四次挥手原理：

第1次挥手：客户端发送一个FIN，用来关闭客户端到服务端的数据传送，客户端进入FIN_WAIT_1状态；

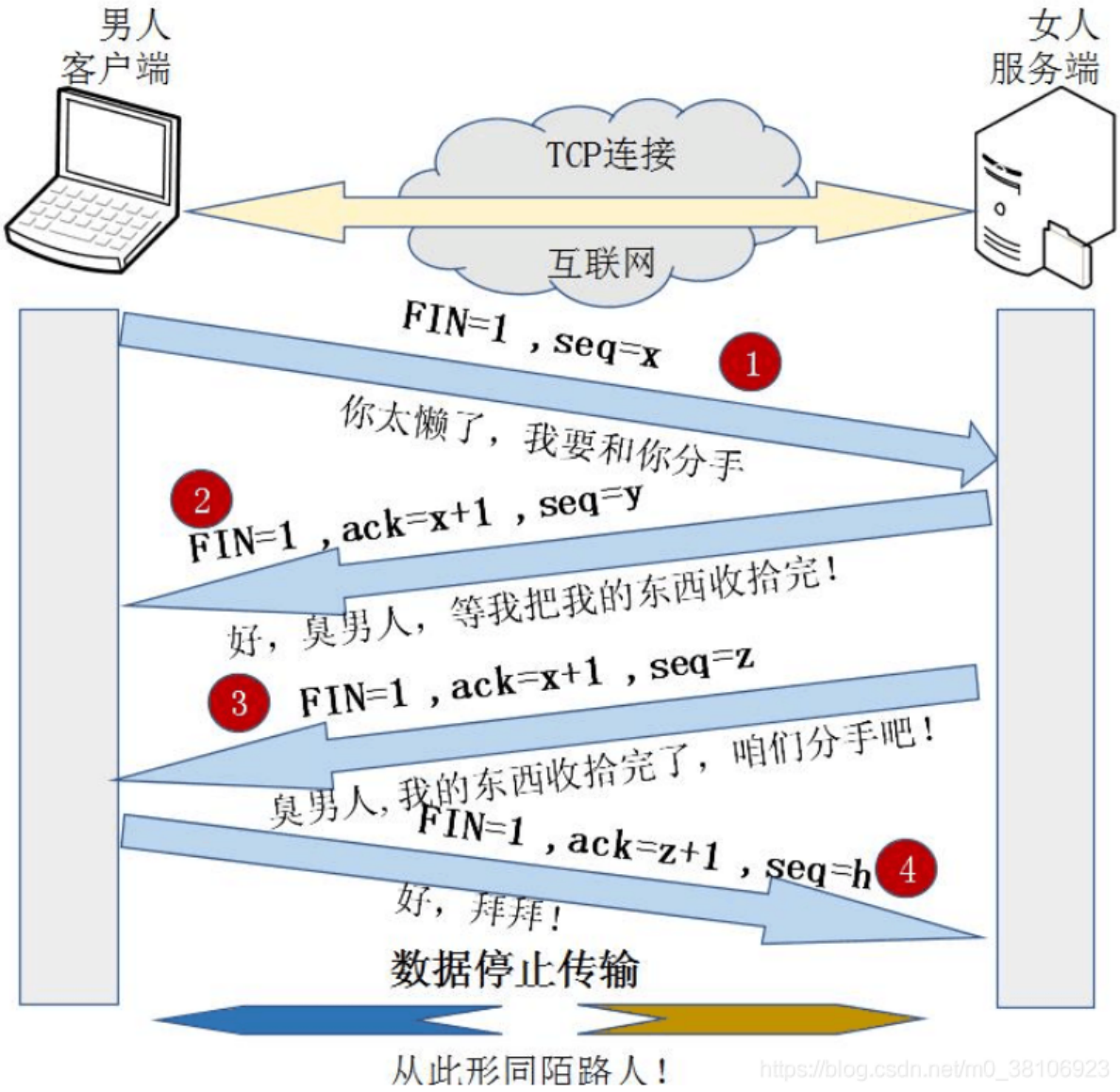
第2次挥手：服务端收到FIN后，发送一个ACK给客户端，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号），服务端进入CLOSE_WAIT状态；

第3次挥手：服务端发送一个FIN，用来关闭服务端到客户端的数据传送，服务端进入LAST_ACK状态；

第4次挥手：客户端收到FIN后，客户端t进入TIME_WAIT状态，接着发送一个ACK给Server，确认序号为收到序号+1，服务端进入CLOSED状态，完成四次挥手。

其中：FIN标志位数置1，表示断开TCP连接。

可通过以下趣味图解理解四次挥手：



四次挥手过程详细说明：

1、客户端发送断开TCP连接请求的报文，其中报文中包含seq序列号，是由发送端随机生成的，并且还将报文中的FIN字段置为1，表示需要断开TCP连接。（FIN=1，seq=x，x由客户端随机生成）；

2、服务端会回复客户端发送的TCP断开请求报文，其包含seq序列号，是由回复端随机生成的，而且会产生ACK字段，ACK字段数值是在客户端发过来的seq序列号基础上加1进行回复，以便客户端收到信息时，知晓自己的TCP断开请求已经得到验证。（FIN=1，ACK=x+1，seq=y，y由服务端随机生成）；

3、服务端在回复完客户端的TCP断开请求后，不会马上进行TCP连接的断开，服务端会先确保断开前，所有传输到A的数据是否已经传输完毕，一旦确认传输数据完毕，就会将回复报文的FIN字段置1，并且产生随机seq序列号。（FIN=1，ACK=x+1，seq=z，z由服务端随机生成）；

4、客户端收到服务端的TCP断开请求后，会回复服务端的断开请求，包含随机生成的seq字段和ACK字段，ACK字段会在服务端的TCP断开请求的seq基础上加1，从而完成服务端请求的验证回复。（FIN=1，ACK=z+1，seq=h，h为客户端随机生成）
至此TCP断开的4次挥手过程完毕。

十五、进程和线程的区别

进程：是并发执行的程序在执行过程中分配和管理资源的基本单位，是一个动态概念，竞争计算机系统资源的基本单位。

线程：是进程的一个执行单元，是进程内科调度实体。比进程更小的独立运行的基本单位。线程也被称为轻量级进程。

一个程序至少一个进程，一个进程至少一个线程。

十六、浏览器内核

浏览器内核可以分成两部分： **渲染引擎(layout engineer** 或者 Rendering Engine)和 **JS 引擎。**浏览器的内核的不同对于网页的语法解释会有不同，所以渲染的效果也不相同。所有网页浏览器、电子邮件客户端以及其它需要编辑、显示网络内容的应用程序都需要内核。

- [ ] 渲染引擎：
  负责取得网页的内容（HTML、XML、图像等等）、整理讯息（例如加入 CSS 等），以及计算网页的显示方式，然后会输出至显示器或打印机。
- [ ] JS 引擎
  解析 Javascript 语言，执行 javascript 语言来实现网页的动态效果。最开始渲染引擎和 JS 引擎并没有区分的很明确，后来 JS 引擎越来越独立，内核就倾向于只指渲染引擎。

十七、描述浏览器渲染过程

DOM Tree 和 CSS RULE Tree 将 html 和 css 解析成树形数据结构，然后 Dom 和 css 合并后生成 Render Tree，用 layout 来确定节点的位置以及关系，通过 painting 按照规则来画到屏幕上，由 display 搭建最终看到效果。

十八、什么是 XXS 攻击

Cross-site script，跨站脚本攻击，当其它用户浏览该网站时候，该段 HTML 代码会自动执行，从而达到攻击的目的，如盗取用户的 Cookie，破坏页面结构，重定向到其它网站等。

XSS 类型：
一般可以分为： 持久型 XSS 和非持久性 XSS

持久型 XSS： 就是对客户端攻击的脚本植入到服务器上，从而导致每个正常访问到的用户都会遭到这段 XSS 脚本的攻击。

非持久型 XSS： 是对一个页面的 URL 中的某个参数做文章，把精心构造好的恶意脚本包装在 URL 参数重，再将这个 URL 发布到网上，骗取用户访问，从而进行攻击。

十九、CSRF 攻击

CSRF(Cross-site request forgery), 中文名称：跨站请求伪造
CSRF 可以简单理解为：攻击者盗用了你的身份，以你的名义发送恶意请求，容易造成个人隐私泄露以及财产安全。

防范：
post 请求
使用 token
验证码

二十、DDOS 攻击

利用目标系统网络服务功能缺陷或者直接消耗其系统资源，使得该目标系统无法提供正常的服务。
DDoS 攻击通过大量合法的请求占用大量网络资源，以达到瘫痪网络的目的。

具体有几种形式：

- [ ] 通过使网络过载来干扰甚至阻断正常的网络通讯；
- [ ] 通过向服务器提交大量请求，使服务器超负荷；
- [ ] 通过阻断某一用户访问服务器；
- [ ] 通过阻断某服务与特定系统或个人的通讯。

二十一、URL 和 URI 有什么区别

URI 是统一资源标识符，相当于一个人身份证号码
Web 上可用的每种资源如 HTML 文档、图像、视频片段、程序等都是一个来 URI来定位的
URI 一般由三部组成
①访问资源的命名机制
②存放资源的主机名
③资源自身的名称，由路径表示，着重强调于资源。

URL 是统一资源定位符，相当于一个人的家庭住址
URL 是 Internet 上用来描述信息资源的字符串，主要用在各种 WWW 客户程序和服务器程序上，特别是著名的 Mosaic。采用 URL 可以用一种统一的格式来描述各种信息资源，包括文件、服务器的地址和目录等。
URL 一般由三部组成
①协议(或称为服务方式)
②存有该资源的主机 IP 地址(有时也包括端口号)
③主机资源的具体地址。如目录和文件名等。

二十二、TCP 三次握手

SYN表示建立连接，ACK表示响应

建立连接前，客户端和服务端需要通过握手来确认对方:

客户端发送 syn(同步序列编号) 请求，进入 syn_send 状态，等待确认

服务端接收并确认 syn 包后发送 syn+ack 包，进入 syn_recv 状态

客户端接收 syn+ack 包后，发送 ack 包，双方进入 established 状态

二十三、TCP 四次挥手

SYN表示建立连接，FIN表示关闭连接，ACK表示响应

客户端 – FIN --> 服务端， FIN—WAIT
服务端 – ACK --> 客户端， CLOSE-WAIT
服务端 – ACK,FIN --> 客户端， LAST-ACK
客户端 – ACK

二十四、http 优化策略

1. Spriting（雪碧图）
2. 内联
3. 拼接
4. 分片
5. Limit HTTP Headers

二十五、http协议中数据请求的方法

- get:：从服务器获取数据
- post：向服务器传输数据
- put：上传指定的 URL
- delete：删除指定资源
- options：返回服务器支持的 http 方法
- head：与 get 类似，但只能获取 http 报头
- connect：将请求连接转换到透明的 tcp/ip 通道

二十六、用URL传参带特殊字符，特殊字符丢失

有些符号在URL中是不能直接传递的，如果要在URL中传递这些特殊符号，那么就要使用他们的编码了。编码的格式为：**%加字符的ASCII码**，即一个百分号%，后面跟对应字符的ASCII（16进制）码值。例如空格的编码值是”“。
如果不使用转义字符，这些编码就会当URL中定义的特殊字符处理。

二十七、DNS解析

1. 浏览器缓存
2. 操作系统缓存
3. 本地域名服务器
4. 根域名服务器请求解析
5. 顶级域名服务器

VUE

一、v-if vs v-show

- [ ] v-if 是真正的条件渲染，通过节点的销毁、重建来实现
- [ ] v-if 是惰性的，即初始条件为 false 时，什么也不做，直到条件第一次为 true 才开始渲染
- [ ] v-show 总会被渲染，只是简单的基于 CSS 来实现样式切换
- [ ] v-if 有更高的切换开销，而 v-show 有更高的初始渲染开销。因此，如果需要非常频繁地切换，则使用 v-show 较好；如果在运行时条件很少改变，则使用 v-if 较好。
- [ ] 总结： v-if 按照条件是否渲染，v-show 是 display 的 block 或 none

二、v-if 与 v-for 一起使用

- [ ] 当 v-if 与 v-for 一起使用时，v-for 具有比 v-if 更高的优先级。（注意这里是 2.x 的版本，3.x 反之）
- [ ] 不推荐同时使用 v-if 和 v-for。

三、计算属性(computed) VS 方法

- [ ] 计算属性的值会被缓存，计算属性是基于它们的响应式依赖进行缓存的，当依赖项改变时，才重新计算计算属性的值并继续缓存。
- [ ] 方法不会缓存，每调用一次方法，都会重新执行一次方法主体代码块

四、计算属性 VS 侦听器

- [ ] 计算属性的值会被缓存。
- [ ] 侦听器不能被缓存。
- [ ] 通常计算属性是根据一个或多个已有数据，返回一个新的值，侦听器是监听一个数据的变化，可能由一个数据的变化导致其它一个或多个数据的变化，或导致其它一些副作用

五、vue.js 的两个核心是什么?

答：**数据驱动、组件系统**

六、vue 常用的修饰符?

- [ ] .prevent: 提交事件不再重载页面；
- [ ] .stop: 阻止单击事件冒泡；
- [ ] .self: 当事件发生在该元素本身而不是子元素的时候会触发；
- [ ] .capture: 事件侦听，事件发生的时候会调用

七、vue 中 key 值的作用?

答：当 Vue.js 用 v-for 正在更新已渲染过的元素列表时，它默认用“就地复用”策略。如果数据项的顺序被改变，Vue 将不会移动DOM 元素来匹配数据项的顺序， 而是简单复用此处每个元素，并且确保它在特定索引下显示已被渲染过的每个元素。**key 的作用主要是为了高效的更新虚拟DOM。**

八、Vue 组件间的参数传递

父组件与子组件传值：

- [ ] 父组件传给子组件：子组件通过 props 方法接受数据;
  子组件传给父组件：$emit 方法传递参数
  非父子组件间的数据传递，兄弟组件传值：
- [ ] eventBus，就是创建一个事件中心，相当于中转站，可以用它来传递事件和接收事件。项目比较小时，用这个比较合适。（虽然也有不少人推荐直接用 VUEX，具体来说看需求咯。技术只是手段，目的达到才是王道。）

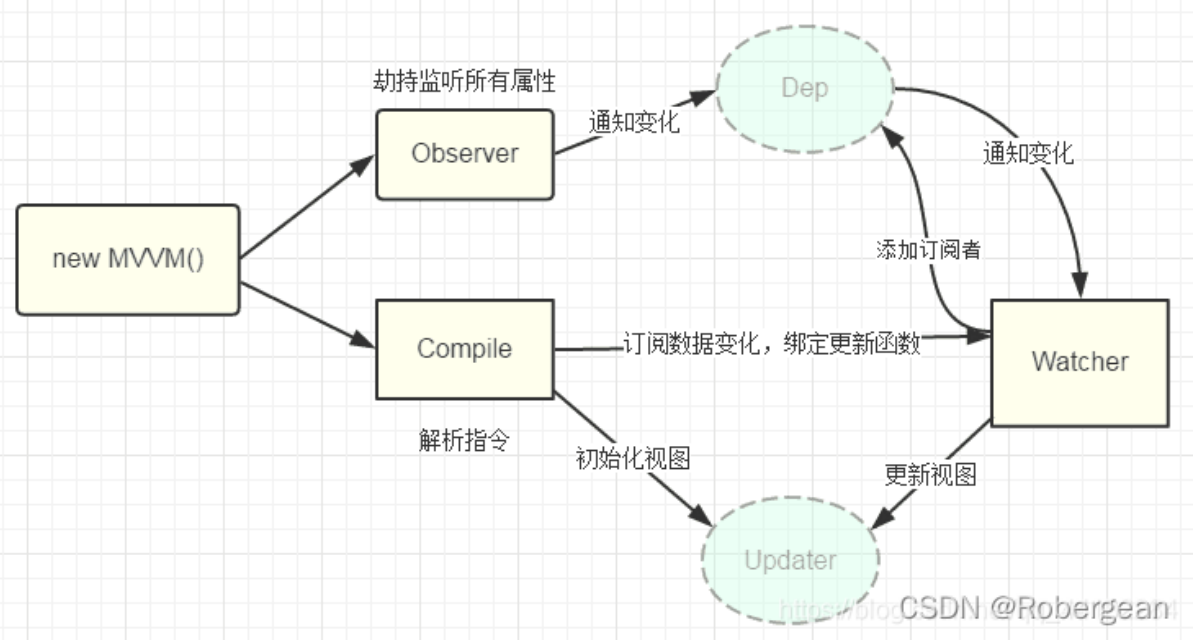
九、Vue.js 双向绑定的原理

Vue.js 2.0 采用数据劫持（Proxy 模式）结合发布者-订阅者模式（PubSub 模式）的方式，通过 Object.defineProperty()来劫持各个属性的 setter，getter，在数据变动时发布消息给订阅者，触发相应的监听回调。

1、实现一个数据监听器Observer，能够对数据对象的所有属性进行监听，如有变动可拿到最新值并通知订阅者

2、实现一个指令解析器Compile，对每个元素节点的指令进行扫描和解析，根据指令模板替换数据，以及绑定相应的更新函数

3、实现一个Watcher，作为连接Observer和Compile的桥梁，能够订阅并收到每个属性变动的通知，执行指令绑定的相应回调函数，从而更新视图



十、Vue中如何监控某个属性值的变化?

例如现在需要监控data中， obj.a 的变化。Vue中监控对象属性的变化你可以这样：

```
watch: {
 'obj.a': {
   handler (newName, oldName) {
      console.log('obj.a changed')
   }
 }
}

```

另一种方法，利用计算属性(computed) 的特性来实现，当依赖改变时，便会重新计算一个新值。

```
computed: {
  a1 () {
    return this.obj.a
  }
}

```

十一、虚拟DOM，diff算法

- [ ] 让我们不用直接操作DOM元素，只操作数据便可以重新渲染页面
- [ ] 虚拟dom是为了解决浏览器性能问题而被设计出来的
- [ ] 当操作数据时，将改变的dom元素缓存起来，都计算完后再通过比较映射到真实的dom树上
- [ ] diff算法比较新旧虚拟dom。如果节点类型相同，则比较数据，修改数据；
- [ ] 如果节点不同，直接干掉节点及所有子节点，插入新的节点；
- [ ] 如果给每个节点都设置了唯一的key，就可以准确的找到需要改变的内容，否则就会出现修改一个地方导致其他地方都改变的情况。例如A-B-C-D, 我要插入新节点A-B-M-C-D,实际上改变了C和D。但是设置了key，就可以准确的找到B C并插入
- [ ] **总结：虚拟dom可以很好的跟踪当前dom状态，因为他会根据当前数据生成一个描述当前dom结构的虚拟dom，然后数据发生变化时，又会生成一个新的虚拟dom，而这两个虚拟dom恰恰保存了变化前后的状态。然后通过diff算法，计算出两个前后两个虚拟dom之间的差异，得出一个更新的最优方法（哪些发生改变，就更新哪些）。可以很明显的提升渲染效率以及用户体验**

十二、v-model 是如何实现的，语法糖实际是什么

作用在表单元素上 `v-model = "message"` 等同于 `v-bind:value = "message" v-on:input = "message=event.target.value"` 作用在组件上,本质是一个父子组件通信的语法糖，通过 prop 和 .emit 实现, 等同于`:value = "message" @input = "$emit('input', $event.target.value)"`

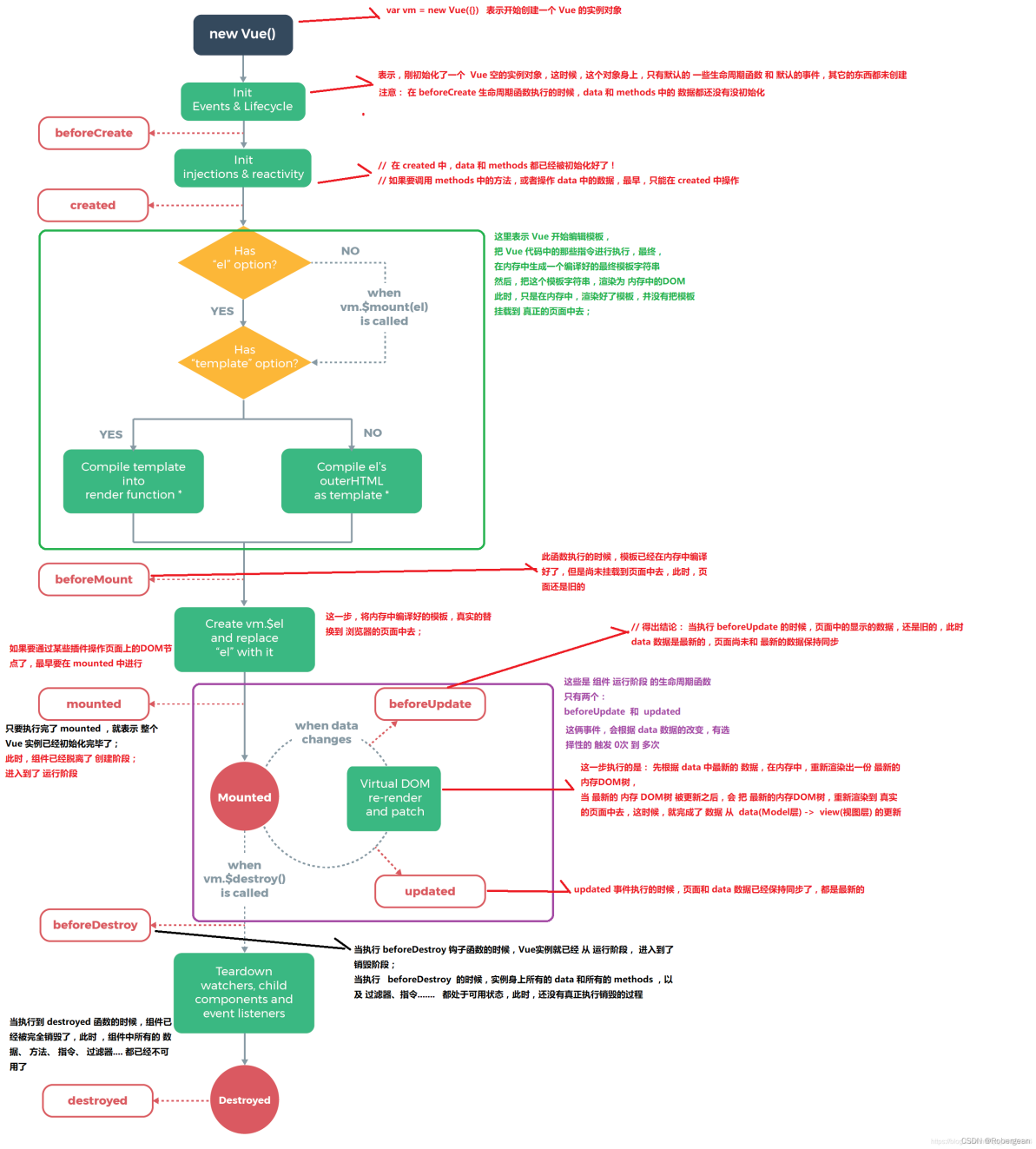
十三、data为什么是一个函数而不是对象

JS中的对象是引用类型的数据，当多个实例引用同一个对象时0=，只要一个实例对这个对象进行操作，其他实例中的数据也会发生变化。
而在Vue中，我们更多的是想要复用组件，那就需要每个组件都有自己的数据，这样组件之间才不会相互干扰。
所以组件的数据不能写成对象的形式，而是要写成函数的形式。数据以函数返回值的形式定义，这样当我们每次复用组件的时候，就会返回一个新的data，也就是说每个组件都有自己的私有数据空间，它们各自维护自己的数据，不会干扰其他组件的正常运行。

十四、vue 生命周期

介绍：每一个vue实例从创建到销毁的过程，就是这个vue实例的生命周期。在这个过程中，他经历了从开始创建、初始化数据、编译模板、挂载Dom、渲染→更新→渲染、卸载等一系列过程。

beforecreate （初始化界面前）
created （初始化界面后）
beforemount （渲染界面前）
mounted （渲染界面后）
beforeUpdate （更新数据前）
updated （更新数据后）
beforedestory （卸载组件前）
destroyed （卸载组件后）



十五、keep-alive

< keep-alive >是Vue的内置组件，能在组件切换过程中将状态保留在内存中，防止重复渲染DOM。

< keep-alive > 包裹动态组件时，会缓存不活动的组件实例，而不是销毁它们。

生命周期钩子：**actived**：激活调用； **deactived**：停用时调用

属性：**include**：名称（正则 or 字符串）匹配时调用； **exclude**：名称不匹配时调用； **max**：最多缓存数量

十六、Vuex是什么？怎么使用？

Vuex是实现组件全局状态（数据）管理的一种机制，可以方便实现组件数据之间的共享；
Vuex集中管理共享的数据，易于开发和后期维护；
能够高效的实现组件之间的数据共享，提高开发效率；
存储在 Vuex 的数据是响应式的，能够实时保持页面和数据的同步；
Vuex重要核心属性包括：state, mutations, action, getters, modules.

state
Vuex 使用单一状态树,即每个应用将仅仅包含一个store 实例，但单一状态树和模块化并不冲突。存放的数据状态，不可以直接修改里面的数据。

mutations
mutations定义的方法动态修改Vuex 的 store 中的状态或数据。

action
actions可以理解为通过将mutations里面处里数据的方法变成可异步的处理数据的方法，简单的说就是异步操作数据。view 层通过 store.dispath 来分发 action。

getters
类似vue的计算属性，主要用来过滤一些数据。

modules
项目特别复杂的时候，可以让每一个模块拥有自己的state、mutation、action、getters,使得结构非常清晰，方便管理。

十七、axios 是什么，其特点和常用语法是什么?

Axios 是一个基于 promise 的 HTTP 库，可以用在浏览器和 node.js 中。前端最流行的 ajax 请求库，react/vue 官方都推荐使用 axios 发 ajax 请求

特点:

- [ ] 基于 promise 的异步 ajax 请求库，支持promise所有的API
- [ ] 浏览器端/node 端都可以使用，浏览器中创建XMLHttpRequests
- [ ] 支持请求 / 响应拦截器
- [ ] 支持请求取消
- [ ] 可以转换请求数据和响应数据，并对响应回来的内容自动转换成 JSON类型的数据
- [ ] 批量发送多个请求
- [ ] 安全性更高，客户端支持防御 XSRF，就是让你的每个请求都带一个从cookie中拿到的key, 根据浏览器同源策略，假冒的网站是拿不到你cookie中得key的，这样，后台就可以轻松辨别出这个请求是否是用户在假冒网站上的误导输入，从而采取正确的策略。

常用语法:

- [ ] axios(config): 通用/最本质的发任意类型请求的方式
- [ ] axios(url[, config]): 可以只指定 url 发 get 请求
- [ ] axios.request(config): 等同于 axios(config)
- [ ] axios.get(url[, config]): 发 get 请求
- [ ] axios.delete(url[, config]): 发 delete 请求
- [ ] axios.post(url[, data, config]): 发 post 请求
- [ ] axios.put(url[, data, config]): 发 put 请求
- [ ] axios.defaults.xxx: 请求的默认全局配置
- [ ] axios.interceptors.request.use(): 添加请求拦截器
- [ ] axios.interceptors.response.use(): 添加响应拦截器
- [ ] axios.create([config]): 创建一个新的 axios(它没有下面的功能)
- [ ] axios.Cancel(): 用于创建取消请求的错误对象

- [ ] axios.CancelToken(): 用于创建取消请求的 token 对象
- [ ] axios.isCancel(): 是否是一个取消请求的错误
- [ ] axios.all(promises): 用于批量执行多个异步请求
- [ ] axios.spread(): 用来指定接收所有成功数据的回调函数的方法

十八、路由传值的方式有哪几种

1 动态路由传值。例如：path:"/home/:id/name"; 接受的时候通过 `this.$route.params`

2 query 传值。因为在 url 中 ？ 后面的参数不会被解析，因此我们可以通过 query 进行传值。接受的时候通过 `this.$route.query`

3 路由解耦。在配置路由的时候添加 `props` 属性为 true,在需要接受参数的组件页面通过 props 进行接受

4 编程式导航 `this.$router.push({path:"/home",query:{}});`

十九、 `$route` 和 $router 的区别

`$route` 是 "路由信息对象"，包括 path，params，hash，query，fullPath， matched，name 等路由信息参数。

`$router` 是"路由实例"对象包括了路由的跳转方法，钩子函数等。

二十、怎么定义 vue-router 的动态路由？怎么获取传过来的值?

动态路由的创建，主要是使用 path 属性过程中，使用动态路径参数，以冒号开头，如下：

```
{
    path: '/details/:id'
    name: 'Details'
    components: Details
}

```

访问 details 目录下的所有文件，如果 details/a，details/b 等，都会映射到 Details 组件上。当匹配到 /details 下的路由时，参数值会被设置到 `this.$route.params` 下，所以通过这个属性可以获取动态参数 `this.$route.params.id`

二十一、vue-router 有哪几种路由守卫?

路由守卫为：
全局守卫：beforeEach
后置守卫：afterEach
全局解析守卫：beforeResolve
路由独享守卫：beforeEnter

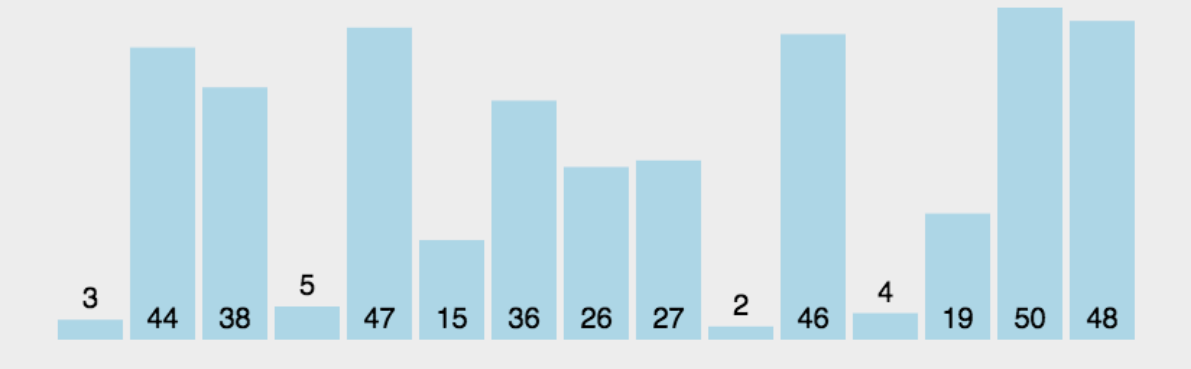
二十二、 MVC和MVVM的区别

- [ ] MVC表示"模型-视图-控制器"，MVVM表示"模型-视图-视图模型"；
- [ ] MVVM是由MVC衍生出来的。MVC中，View会直接从Model中读取数据；

- [ ] MVVM各部分的通信是双向的，而MVC各部分通信是单向的；
- [ ] MVVM是真正将页面与数据逻辑分离放到js里去实现，而MVC里面未分离。

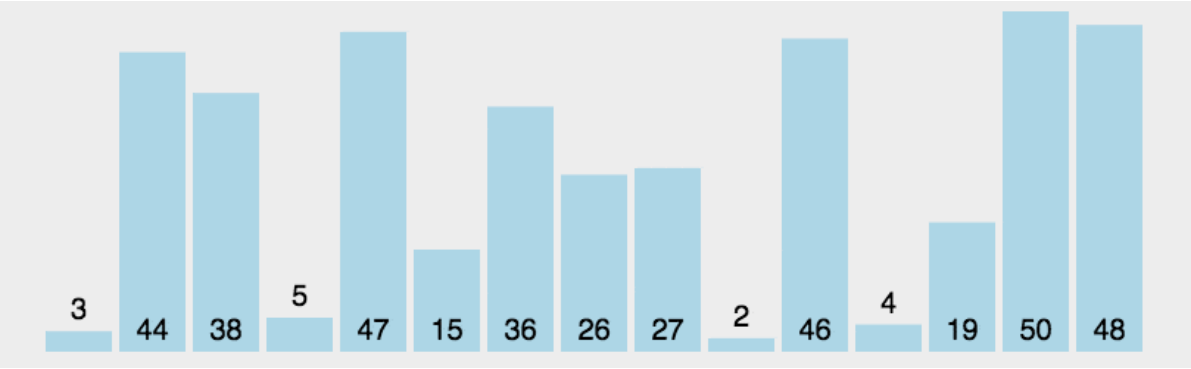
算法

一、冒泡排序




```
bubbleSort = arr => {
  for (let i = 0, l = arr.length; i < l - 1; i++) {
    for (let j = 0; j < l - 1 - i; j++) {
      if(arr[j] > arr[j+1]) {
        let temp = arr[j+1]
        arr[j+1] = arr[j]
        arr[j] = temp
      }
    }
  }
  return arr
}
console.log('结果：', bubbleSort([5,3,2,4,8]))

```

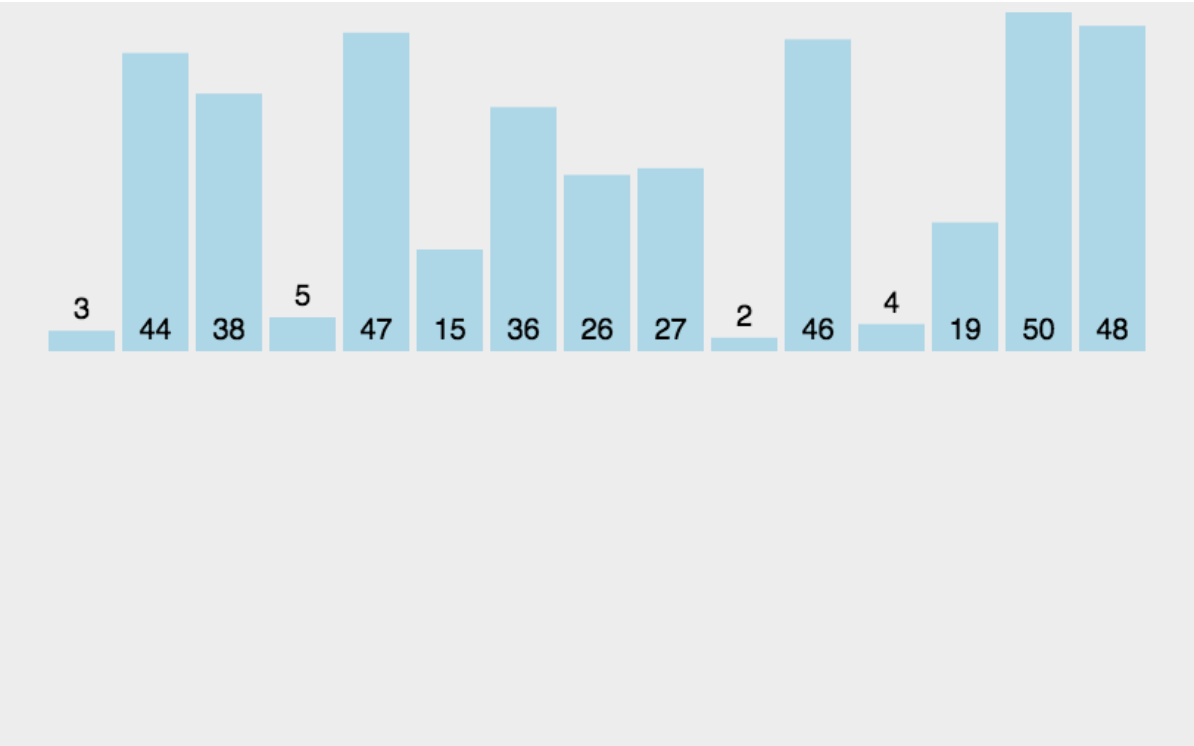
二、选择排序




```
selectSort = arr => {
  for (let i = 0,l = arr.length; i < l - 1; i++) {
    for (let j = i+1; j < l; j++) {
      if(arr[i] > arr[j]) {
        let temp = arr[i]
        arr[i] = arr[j]
        arr[j] = temp
      }
```

```
    }
  }
  return arr
}
console.log('结果：', selectSort([5,3,2,4]))

```

三、插入排序




```
insertSort = arr => {
  let l = arr.length
  for(let i = 1; i < l; i++){
    let temp = arr[i]
    let j = i - 1
    for(; j >= 0 && arr[j] > temp; j--) {
      arr[j+1] = arr[j]
    }
    arr[j+1] = temp
  }
  return arr
}
console.log(insertSort([3,44,58,2,9,1]))

```

四、快速排序

```
quickSort = arr => {
  let l = arr.length
  let leftArr = []
  let rightArr = []
  let a = arr[0]

  if( l <= 1) {
```

```
    return arr
  }

  for(let i = 1; i < l; i++) {
    if(arr[i] > a) {
      rightArr.push(arr[i]);
    }
    else{
      leftArr.push(arr[i]);
    }
  }
  return [].concat(quickSort(leftArr),[a],quickSort(rightArr))
}
console.log('结果：', quickSort([5,1,6,7]))

```

五、全排列

```
func = arr => {
  let len = arr.length
  let res = [] // 所有排列结果
  /**
   * 【全排列算法】
   * 说明：arrange用来对arr中的元素进行排列组合，将排列好的各个结果存在新数组中
   * @param tempArr：排列好的元素
   * @param leftArr：待排列元素
   */
  let arrange = (tempArr, leftArr) => {
    if (tempArr.length === len) { // 这里就是递归结束的地方
      res.push(tempArr.join('')) // 得到全排列的每个元素都是字符串
    } else {
      leftArr.forEach((item, index) => {
        let temp = [].concat(leftArr)
        temp.splice(index, 1)
        // 此时，第一个参数是当前分离出的元素所在数组；第二个参数temp是传入的leftArr去掉第一个后的结果
        arrange(tempArr.concat(item), temp) // 这里使用了递归
      })
    }
  }
  arrange([], arr)
  return res
}
console.log('结果：', func(['A', 'B', 'C', 'D']))

```

六、js 实现队列

```
/** 队列思想(FIFO) 先进先出
 * enqueue（element）：向队列尾部添加一个（或多个）新的项；
 * dequeue（）：移除队列的第一（即排在队列最前面的）项，并返回被移除的元素；
 * front（）：返回队列中的第一个元素——最先被添加，也将是最先被移除的元素。
              队列不做任何变动（不移除元素，只返回元素信息与Stack类的peek方法非常类似）；
 * isEmpty（）：如果队列中不包含任何元素，返回true，否则返回false；
```

```
 * size（）：返回队列包含的元素个数，与数组的length属性类似；
 * toString（）：将队列中的内容，转成字符串形式
*/
function Queue() {
  this.items = []

  // enqueue():将元素加入到队列中
  Queue.prototype.enqueue = element => {
    this.items.push(element)
  }

  // dequeue():从队列中删除前端元素
  Queue.prototype.dequeue = () => {
    return this.items.shift()
  }

  // front():查看前端的元素
  Queue.prototype.front = () => {
    return this.items[0]
  }

  // isEmpty:查看队列是否为空
  Queue.prototype.isEmpty = () => {
    return this.items.length === 0
  }

  // size():查看队列中元素的个数
  Queue.prototype.size = () => {
    return this.items.length
  }

  // toString():将队列中元素以字符串形式输出
  Queue.prototype.toString = () => {
    let resultString = ''
    for (let i of this.items){
      resultString += i + ' '
    }
    return resultString
  }
}

// 创建队列
let queue = new Queue()

// 加入队列
queue.enqueue('a')
queue.enqueue('b')
queue.enqueue('c')
console.log(queue)

// 从队列中删除
queue.dequeue()
console.log(queue)

// 查看前端元素
console.log(queue.front())
```

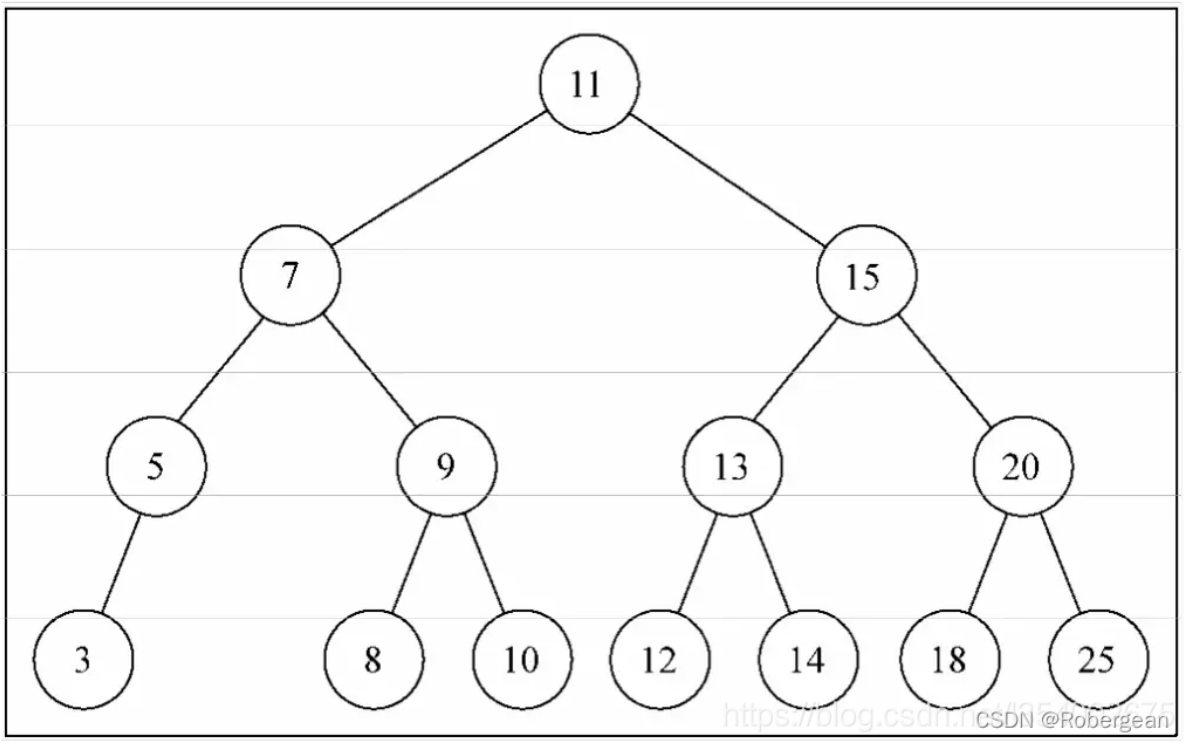
```
63
64 // 判断是否为空
65 console.log(queue.isEmpty())
66
67 // 查看 size
68 console.log(queue.size())
69
70 // 字符串形式输出
71 console.log(queue.toString())
72
```

七、js 实现栈

```
1  /** 栈的思想：  先进后出，后进先出
2   * enStack(element)：添加一个新元素到栈顶位置；
3   * deStack()：移除栈顶的元素，同时返回被移除的元素；
4   * peek()：返回栈顶的元素，不对栈做任何修改（该方法不会移除栈顶的元素，仅仅返回它）；
5   * isEmpty()：如果栈里没有任何元素就返回true，否则返回false；
6   * size()：返回栈里的元素个数。这个方法和数组的length属性类似；
7   * toString()：将栈结构的内容以字符串的形式返回。
8   *
9  */
10
11 function Stack() {
12   this.items = []
13
14   // 压栈
15   Stack.prototype.enStack = element => {
16     this.items.push(element)
17   }
18
19   // 出栈
20   Stack.prototype.deStack = () => {
21     return this.items.pop()
22   }
23
24   // 查看栈顶
25   Stack.prototype.peek = () => {
26     return this.items[this.items.length-1]
27   }
28
29   // 判断是否为空
30   Stack.prototype.isEmpty = () => {
31     return this.items.length === 0
32   }
33
34   // 查看栈中元素个数
35   Stack.prototype.size = () => {
36     return this.items.length
37   }
38
39   // 以字符串的形式输出
40   Stack.prototype.toString = () => {
41     let resultString = ''
42     for(let i of this.items) {
```

```
      resultString += i + ' '
    }
    return resultString
  }
}

// 创建栈
let stack = new Stack()

// 压栈
stack.enStack('a')
stack.enStack('b')
stack.enStack(3)
console.log(stack)

// 出栈
console.log(stack.deStack())

// 查看栈顶
console.log(stack.peek())

// 判断是否为空
console.log(stack.isEmpty())

// 查看栈中元素个数
console.log(stack.size())

// 以字符串的形式输出
console.log(stack.toString())

```

八、js实现二叉树




```
// 创建节点
```

```
class Node {
  constructor(key, left = null, right = null) {
    this.key = key
    this.left = left
    this.right = right
  }
}

// 方法
class BinarySearchTree {
  constructor(node) {
    this.root = node
  }

  // 插入节点
  insert(newNode, node = this.root) {
    if (!this.root) {
      this.root = newNode
    } else {
      if (newNode.key < node.key) {
        if (node.left === null) {
          node.left = newNode
        } else {
          this.insert(newNode, node.left)
        }
      } else {
        if (node.right === null) {
          node.right = newNode
        } else {
          this.insert(newNode, node.right)
        }
      }
    }
  }

  // 前序遍历
  preOrderTraverse(curNode = this.root) { //可以从指定结点进行
    if (!curNode) {
      return
    }
    let arr = [];
    const preOrderTraverseNode = node => {
      if (!node) {
        return
      }
      arr.push(node.key)
      preOrderTraverseNode(node.left)
      preOrderTraverseNode(node.right)
    }
    preOrderTraverseNode(curNode)
    return arr
  }

  // 中序遍历
  inOrderTraverse(curNode = this.root) {
    if (!curNode) {
```

```
      return
    }
    let arr = [];
    const inOrderTraverseNode = node => {
      if (!node) {
        return
      }
      inOrderTraverseNode(node.left)
      arr.push(node.key)
      inOrderTraverseNode(node.right)
    }
    inOrderTraverseNode(curNode)
    return arr
  }

  // 后序遍历
  postOrderTraverse(curNode = this.root) {
    if (!curNode) {
      return
    }
    let arr = [];
    const postOrderTraverseNode = node => {
      if (!node) {
        return
      }
      postOrderTraverseNode(node.left)
      postOrderTraverseNode(node.right)
      arr.push(node.key)
    }
    postOrderTraverseNode(curNode)
    return arr
  }

  // 最小节点
  minNode(node = this.root) {
    if (!node.left) {
      return node.key
    }
    return this.minNode(node.left)
  }

  // 最大节点
  maxNode(node = this.root) {
    if (!node.right) {
      return node.key
    }
    return this.maxNode(node.right)
  }

  // 查找节点
  search(key,curNode = this.root) {
    if(!curNode) {
      return false
    }
    if(key === curNode.key) {
      return curNode
```

```
    }
    return this.search(key, key < curNode.key ? curNode.left : curNode.right)
  }
}


const tree = new BinarySearchTree(new Node(11))
tree.insert(new Node(15))
tree.insert(new Node(7))
tree.insert(new Node(5))
tree.insert(new Node(3))
tree.insert(new Node(9))
tree.insert(new Node(8))
tree.insert(new Node(10))
tree.insert(new Node(13))
tree.insert(new Node(12))
tree.insert(new Node(14))
tree.insert(new Node(20))
tree.insert(new Node(18))
tree.insert(new Node(25))
console.log(tree.preOrderTraverse())
console.log(tree.inOrderTraverse())
console.log(tree.postOrderTraverse())
console.log(tree.minNode())
console.log(tree.maxNode())
console.log(tree.search(9))

```

编程题

一、实现获取所有数据类型的函数

```
function getType(obj) {
 if (obj === null) {
    return String(obj)
  }
  const toType = (obj) => {
    return Object.prototype.toString.call(obj).replace('[object ', '').replace(']', '').toLowerCase()
  }
  return typeof obj === 'object' ? toType(obj) : typeof obj
}

```

二、实现一个滑动加载数据的防抖函数

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
  <title>Document</title>
  <style>
    img {
      display: block;
      margin-bottom: 50px;
      width: 400px;
      height: 400px;
      background-color: red;
    }
  </style>
</head>
<body>
  <div id="div">
    <img src="http://ww4.sinaimg.cn/large/006y8mN6gw1fa5obmqrmvj305k05k3yh.jpg" alt="">
    <img src="http://ww1.sinaimg.cn/large/006y8mN6gw1fa7kaed2hpj30sg0l9q54.jpg" alt="">
    <img src="http://ww4.sinaimg.cn/large/006y8mN6gw1fa5obmqrmvj305k05k3yh.jpg" alt="">
  </div>
  <script>
    function lazyload() { //监听页面滚动事件
        var documentHeight = document.documentElement.offsetHeight; //文档总高度
        var seeHeight = document.documentElement.clientHeight; //可见区域高度
        var scrollTop = document.documentElement.scrollTop || document.body.scrollTop; //滚动条距离顶部高度
        if (documentHeight - seeHeight - scrollTop < 30) {
            var imgElement = document.createElement('img');
            imgElement.setAttribute('src', 'http://ww4.sinaimg.cn/large/006y8mN6gw1fa5obmqrmvj305k05k3yh.jpg');
            document.querySelector('#div').appendChild(imgElement);
        }
    }
    // 简单的节流函数
    //fun 要执行的函数
    //delay 延迟
    //time  在time时间内必须执行一次
    function throttle(fun, delay, time) {
        var timeout,
            startTime = new Date();
        return function () {
            var context = this,
                args = arguments,
                curTime = new Date();
            clearTimeout(timeout);
            // 如果达到了规定的触发时间间隔，触发 handler
            if (curTime - startTime >= time) {
                fun.apply(context, args);
                startTime = curTime;
                // 没达到触发间隔，重新设定定时器
            } else {
                timeout = setTimeout(function () {
```

```
                    fun.apply(context, args);
                }, delay);
            }
        };
    };
    // 采用了节流函数
    window.addEventListener('scroll', throttle(lazyload, 500, 1000));
</script>
</body>
</html>

```

三、累加

```
// 累加 reduce
addFunc = arr => {
  return arr.reduce((x, y) => {
    return  x + y
  }, 0)
}
console.log(addFunc([8,5,2,1]))

// 累加 eval
addFunc = arr => {
  return eval(arr.join('+'))
}
console.log(addFunc([62,4,3,7]))

// 累加 for循环
addFuncFor = arr => {
  let l = arr.length
  let sum = 0
  for(let i = 0; i < l ; i++) {
    sum += arr[i]
  }
  return sum
}
console.log(addFuncFor([8,3,47,5]))

```

四、阶乘

```
// 阶乘 递归
mulFunc = n => {
  if(n ===0) {
    return 1
  } else {
    return n * mulFunc(n-1)
  }
}
console.log(mulFunc(4))

// 阶乘 for循环
mulFuncFor = n => {
```

```
  for(let i = n-1; i > 0; i--) {
    n *= i
  }
  return n
}
console.log(mulFuncFor(5))

```

五、字符串转换成数组对象

请把字符串：str = '10,100|20.200|30.300|40,400' 处理成如下格式：
arr = [
{h:10,v:100},
{h:20,v:300},
{h:30,v:300},
{h:40,v:400}
]

```
func = str => {
  const arr = []
  const group = str.split('|')
  group.forEach((item, index) => {
    const h = item.split(',')[0]
    const v = item.split(',')[1]
    let obj = {
      h,v
    }
    arr.push(obj)
  })
  return arr
}
console.log(func('10,100|20,200|30,300|40,400'))

```