##1 自我介绍

（1）自我介绍

基本信息

各位面试官好，我叫王小莹，目前就读于东北大学模式识别与智能系统专业，很荣幸能够参与贵公司前端开发工程师职位的面试，接下来我简单介绍一下我的个人情况。

学习方面：科研、在校荣誉

我本科就读于华北电力大学自动化专业，于2020年以保研的形式进入东北大学攻读硕士，研究生期间的研究方向为伪造人脸视频检测，同时，我也是实验室一项国防重大培育项目的主要负责人，在项目中我主要负责提供算法支持部分。

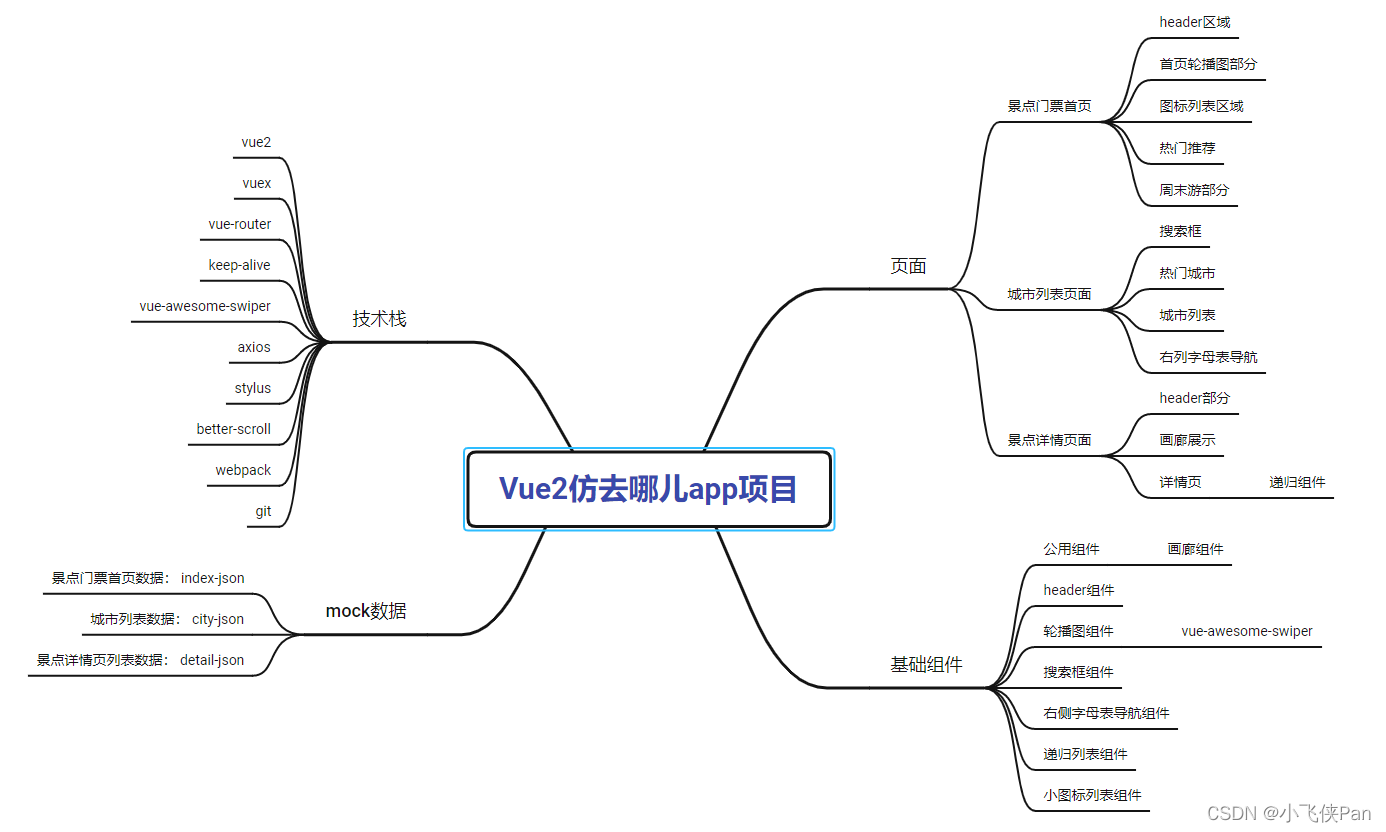
前端掌握技能、项目

在科研闲暇之余，我进行了前端相关技能的学习，包括HTML、CSS、JavaScript、计算机网络等前端相关知识，同时也对java、python语言以及深度学习框架pytorch有一定的了解。项目实践方面做了一个去哪网移动端网页，在这过程中使我更加深入的领会了组件化思想，并对项目开发流程有了初步的了解。

个人性格、具备优势

针对前端开发工程师这个岗位而言，我认为我具有以下的优势：首先我个人刻苦认真、虚心好学，具备较强的学习能力，本科硕士期间获得数次一等学业奖学金；第二个是我有一定的责任心，同时性格较为温和，具有良好的团队协作精神和沟通能力。我希望能加入[贵公司](https://www.zhihu.com/search?q=%E8%B4%B5%E5%85%AC%E5%8F%B8&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22article%22%2C%22sourceId%22%3A%22290759849%22%7D)贡献自己的一份力量，提升个人能力的同时，与公司共同进步。谢谢面试官！

（2）介绍项目



页面：分为景点门票首页、城市列表页面、景点详情页面

技术栈：vue2、vuex、vue-router、keep-alive、vue-awesome-swiper、axios、stylus、better-scroll、webpack、git

技术难点：

1）基于多种方法实现组件间通信

2）使用Axios进行Ajax数据的获取

3）使用节流进行性能优化（监听touchmove事件中）

4）使用keep-alive优化网页性能

目前网页的问题：每次点击城市、选择城市、点击城市、选择城市，路由切换时，Ajax都会被重新发送请求：

所以在App.vue中，根据<router-view/>显示的是路由所对应的内容，故在外层包裹<keep-alive>，让路由内容加载过一次后，会保存在内存中，即之后选择城市时，会从内存中调数据，不重新发Ajax请求：

<keep-alive>

<router-view/>

</keep-alive>

项目收获：

1）理解整个vue项目的流程，以及vue整个项目的各部分的划分。

-Vue Router实现多页面的路由

-Vuex多组件的数据共享

（项目中是为了实现城市选择列表页面和首页的数据传递）

- Axios进行Ajax数据的获取

2）移动端页面布局技巧

3）stylus编写前端样式

是css的预处理器，Stylus 给 CSS 添加了可编程的特性，也就是说，在 Stylus 中可以使用变量、函数、判断、循环一系列 CSS 没有的东西来编写样式文件，这个文件可编译成 CSS 文件。使用stylus编写样式代码较为简洁，不需要有很多括号，而是使用缩进的方式区分父级和子级

4）swiper组件的使用

项目中遇到的问题：

目前遇到的主要问题，通常是框架或是一些包的版本配置不正确的问题，由于这个项目的业务相对常规，没有遇到过大的困难，但是也比较期待在今后工作中遇到一些难题，因为只有这样我的技术才会有长进。

2 HTML

3 CSS

1）块级、行内元素分别有哪些，区别是什么，怎么修改呈现方式（★★★★）

**块级元素（block）**

1.独占一行

2.可以设置宽高，宽度没有设置时，默认为父级元素（没有就是浏览器的）的100%

3.块级元素中可以包含块级元素和行内元素

4.可以设置任何方向的margin，padding。

**行内元素 (inline)**

1.和其他元素都在一行

2.元素的宽和高就是内容撑开的宽高，对宽高属性值不生效

3.行内元素只能包含行内元素，不能包含块级元素

（标准流中，行内元素是不能设置宽高的。但是，一旦设置为浮动之后，即使不转成块级元素，也能够设置宽高了。）

4.可以设置左右margin，padding，行内元素的纵向padding和margin都是不考虑的，这是css规范定义的。 inline元素确实可以设置垂直方向的 padding 和 margin 值，但是 inline 元素的 margin 和 padding 的垂直方向上不产生边距效果，即不影响布局。

**行内块元素(inline-block)**

1.可以和其他行内元素并排

2.可以设置宽高

3.可以设置任何方向的margin，padding。

2）BFC、IFC、浮动、清除浮动

（1）BFC

块格式化上下文（Block Formatting Context，BFC）是Web页面的可视化CSS渲染的一部分，是布局过程中生成块级盒子的区域，也是浮动元素与其他元素的交互限定区域。

通俗来讲：BFC是一个独立的布局环境，可以理解为一个容器，在这个容器中按照一定规则进行物品摆放，并且不会影响其它环境中的物品。如果一个元素符合触发BFC的条件，则BFC中的元素布局不受外部影响。

**创建BFC的条件：**

* 根元素：body；
* 元素设置浮动：float 除 none 以外的值；
* 元素设置绝对定位：position (absolute、fixed)；
* display 值为：inline-block（行内块元素）、table-cell、table-caption、flex、inline-flex(弹性元素)、grid、inline-grid（网格元素）等；
* overflow 值为：hidden、auto、scroll（overflow值不为visible的块元素）；

**BFC的特点：**

* 垂直方向上，自上而下排列，和文档流的排列方式一致。
* 在BFC中上下相邻的两个容器的margin会重叠
* 计算BFC的高度时，需要计算浮动元素的高度（不会浮动坍塌）
* BFC区域不会与浮动的容器发生重叠
* BFC是独立的容器，容器内部元素不会影响外部元素
* 每个元素的左margin值和容器的左border相接触

**BFC的作用：**

* **解决margin的重叠问题**：由于BFC是一个独立的区域，内部的元素和外部的元素互不影响，将两个元素变为两个BFC，就解决了margin重叠的问题。
* **解决高度塌陷的问题**（**清除浮动**）：在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，只需要把父元素变成一个BFC。常用的办法是给父元素设置overflow:hidden。
* **创建自适应两栏布局**：可以用来创建自适应两栏布局：左边的宽度固定，右边的宽度自适应。

左侧设置float:left，右侧设置overflow: hidden。这样右边就触发了BFC，BFC的区域不会与浮动元素发生重叠，所以两侧就不会发生重叠，实现了自适应两栏布局。

（2.1）浮动、清除浮动

**浮动的定义：**元素的浮动是指设置了浮动属性的元素会脱离标准普通流的控制移动到指定位置。 非IE浏览器下，**容器不设高度且子元素浮动**时，容器高度不能被内容撑开。 此时，内容会溢出到容器外面而影响布局。这种现象被称为浮动（溢出）。

**浮动的性质**：

性质1：浮动的元素脱标：脱标即脱离标准流。

性质2：浮动的元素互相贴靠

性质3：浮动的元素有“字围”效果（字围着浮动元素）

性质4：收缩：一个浮动的元素，如果没有设置width，那么将自动收缩为内容的宽度（这点非常像行内元素）。

标准流中，行内元素是不能设置宽高的。但是，一旦设置为浮动之后，即使不转成块级元素，也能够设置宽高了。所有标签，浮动之后，已经不区分行内、块级了。

**浮动元素引起的问题？**

* 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会跟随其后
* 若浮动的元素不是第一个元素，则该元素之前的元素也要浮动，否则会影响页面的显示结构

（2.2）清除浮动的方式如下：

（这里所说的清除浮动，指的是清除浮动与浮动之间的影响）

方法一： **给浮动元素的祖先元素加高度**（工作中用得很少，高度height其实很少出现。为什么？因为能被内容撑高！）

方法二：**clear:both**; （常用）使用 clear，元素浮动之后，周围的元素会重新排列，为了避免这种情况，使用 clear 属性。clear 属性指定元素两侧不能出现浮动元素。

一般使用伪元素的方式清除浮动：

.clear::after{  content:'';  display: block;   clear:both;}

clear属性只有块级元素才有效的，而::after等伪元素默认都是内联水平，这就是借助伪元素清除浮动影响时需要设置display属性值的原因。

方法三：**隔墙法**

上面这个例子中，为了防止第二个div贴靠到第二个div，我们可以在这两个div中间用一个新的div隔开，然后给这个新的div设置clear: both;属性

方法四：**overflow:hidden;**

一个父亲不能被自己浮动的儿子，撑出高度。但是，只要给父亲加上overflow:hidden; 那么，父亲就能被儿子撑出高了。这是一个偏方。（原理：计算BFC的高度时，需要计算浮动元素的高度（不会浮动坍塌））

3）margin塌陷，重叠

普通文档流中的块级元素在垂直方向，两个外边距相遇时会合并为一个外边距，合并后的外边距高度取两个合并的外边距高度中的较大值。行内元素、浮动元素以及绝对定位的元素之间不会发生外边距合并。外边距合并主要发生在父子关系以及相邻兄弟关系的块级元素之间，甚至空的块级元素也会发生外边距合并。

**父子元素之间（margin塌陷）**

在标准文档流中，垂直方向的父子元素，当给子元素设置margin-top: 100px时，子元素不会相对父元素顶端距离100个像素，而是父子元素同时相对文档下移100px。或则同时给父子两个元素设置margin-top，但是呈现的效果是谁大，父子元素整体像下移动大的距离（此时子元素还是相对父元素不动）。这两种现象我们都称作margin塌陷。表现为较大的margin会覆盖掉较小的margin，竖直方向的两个盒子中间只有一个较大的margin，这就是margin塌陷现象。

**塌陷问题解决方案：**

“触发盒子的BFC模型”。

方案1：给父元素加上绝对定位。position:absolute。效果如下：

方案2：display:inline-block;让父级同时具有行级属性和块级属性。代码如下：

方案3：float:left/right;让父级产生浮动流。代码：

方案4：overflow:hiddle;溢出部分隐藏。如下：

**相邻兄弟元素之间（margin重叠）**

两个相邻的兄弟元素之间的上下外边距相遇时也会发生合并

**重叠问题解决方案：**

**两个元素变成两个BFC**

总结。上面的解决方案都改变了文档的结构。一般是不建议采用。如果万一我们确实需要两个兄弟元素垂直相距这么大的距离，我们只需要给其中一个设置margin属性就行了。比如给上一个设置margin-bottom属性或者是给下面的元素设置margin-top属性。

4）flex、grid布局、两栏布局、三栏布局

（4.1）两栏布局

一般两栏布局指的是**左边一栏宽度固定，右边一栏宽度自适应**，两栏布局的具体实现：

* 利用浮动，将左边元素宽度设置为200px，并且设置向左浮动。将右边元素的margin-left设置为200px，宽度设置为auto（默认为auto，撑满整个父元素）。
* 利用浮动，左侧元素设置固定大小，并左浮动，右侧元素设置overflow: hidden; 这样右边就触发了BFC，BFC的区域不会与浮动元素发生重叠，所以两侧就不会发生重叠。
* 利用flex布局，将左边元素设置为固定宽度200px，将右边的元素设置为flex:1。

.outer {  
 display: flex;  
 height: 100px;  
}  
.left {  
 width: 200px;  
 background: tomato;  
}  
.right {  
 flex: 1;    /\*flex:1 ==> flex:1 1 auto\*/  
 background: gold;  
}

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

flex-grow默认为0，即如果存在剩余空间，也不放大。

flex-shrink默认为1，即如果空间不足，该项目将缩小。

* 利用绝对定位，将父级元素设置为相对定位。左边元素设置为absolute定位，并且宽度设置为200px。将右边元素的margin-left的值设置为200px。

.outer {  
 position: relative;  
 height: 100px;  
}  
.left {  
 position: absolute;  
 width: 200px;  
 height: 100%;  
 background: tomato;  
}  
.right {  
 margin-left: 200px;  
 background: gold;  
 height: 100%;  
}

（4.2）三栏布局（8种）

三栏布局一般指的是页面中一共有三栏，**左右两栏宽度固定，中间自适应的布局**，三栏布局的具体实现：

* 流体布局（浮动）

原理: 左右模块各自向左右浮动，并设置中间模块的margin值使中间模块宽度自适应

缺点: 主要内容无法最先加载，当页面内容较多时会影响用户体验

* BFC 三栏布局

原理: BFC规则有这样的描述：BFC 区域不会与浮动元素重叠, 因此我们可以利用这一点来实现 3 列布局

缺点: 主要内容模块无法最先加载，当页面中内容较多时会影响用户体验。因此为了解决这个问题，有了下面要介绍的布局方案双飞翼布局

* 圣杯布局
  + 写法
    - 三者均向左浮动
    - 中间模块宽度设置为100%
    - 左模块左外边距-100%，右模块左边距负自身宽度
    - 设置外模块内/外边距给左右子面板留出空间
    - 设置两个子面板为相对定位，左模块left值为负宽度，右模块right值为负宽度
  + 当面板的main部分比两边的子面板宽度小的时候，布局就会乱掉。

双飞翼布局

原理：给center添加一个容器元素container，设置center的margin值避开侧边栏，让left、right飘在两边

优点: 主要内容模块可以优先加载，当页面中内容较多时不会影响用户体验。

* + 利用的是浮动元素 margin 负值的应用，感兴趣的同学可以上网搜搜原理。
  + 主体内容可以优先加载，HTML 代码结构稍微复杂点。
  + 写法
    - 左右外模块均向左浮动
    - 外模块宽度设置为100%
    - 左模块左外边距-100%，右模块左边距负自身宽度
    - 设置外模块内/外边距给左右子面板留出空间
* flex布局

原理： 设置父元素 display:flex;再设置子元素的flex；

flex 属性是 flex-grow、flex-shrink 和 flex-basis 属性的简写属性：

flex-grow：项目将相对于其他灵活的项目进行扩展的量 flex-shrink：规定项目将相对于其他灵活的项目进行收缩的量 flex-basis：项目的默认长度 优点：可以先写center，让他先加载，然后用order属性，把他排到中间的位置

.container{  
  display:flex;  
  width:100%;  
  height:100px;  
}  
.left{  
  flex:0 0 100px;  
  order: 0 /\*默认为0\*/  
}  
.right{  
  flex:0 0 100px;  
  order:2  
}  
.center{  
  flex:1 1 auto;  
  order:1  
}  
​table布局

**缺点**：无法设置栏间距

 <style>  
   .container{  
       display:table;  
       width:100%;  
}  
   .left,.center,.right{  
       display:table-cell;  
  }  
   .left{  
       width:100px;  
       height:100px;  
  }  
   .right{  
       width: 100px;  
       height:100px;  
  }  
 </style>  
</head>

绝对定位布局

**优点**: 简单实用，并且主要内容可以优先加载。

 <style>  
   .container{  
       position:relative;  
  }  
   .center{  
       margin-left:100px;  
       margin-right:100px;  
       height:100px;  
  }  
   .left{  
       position:absolute;  
       left:0;  
       top:0;  
       width:100px;  
       height:100px;  
  }  
   .right{  
       position:absolute;  
       right:0;  
       top:0;  
       width:100px;  
       height:100px;  
  }  
 </style>

网格布局(Grid布局)

.container{       
  display: grid;              
  grid-template-columns: 100px auto 100px;                       
  grid-template-rows: 100px;                  
  }

5）隐藏元素的方法，各自的区别(display/visiablity/opacity)

* **display: none**：渲染树不会包含该渲染对象，因此该元素不会在页面中占据位置，也不会响应绑定的监听事件。
* **visibility: hidden**：元素在页面中仍占据空间，但是不会响应绑定的监听事件。
* **opacity: 0**：将元素的透明度设置为 0，以此来实现元素的隐藏。元素在页面中仍然占据空间，并且能够响应元素绑定的监听事件。
* **position: absolute**：通过使用绝对定位将元素移除可视区域内，以此来实现元素的隐藏。
* **z-index: 负值**：来使其他元素遮盖住该元素，以此来实现隐藏。

6）display: none与visibility: hidden的区别

这两个属性都是让元素隐藏，不可见。**两者区别如下：**

（1）**在渲染树中**

* + display:none会让元素完全从渲染树中消失，渲染时不会占据任何空间；
  + visibility:hidden不会让元素从渲染树中消失，渲染的元素还会占据相应的空间，只是内容不可见。

（2）**是否是继承属性**

* + display:none是非继承属性，子孙节点会随着父节点从渲染树消失，通过修改子孙节点的属性也无法显示；
  + visibility:hidden是继承属性，子孙节点消失是由于继承了hidden，通过设置visibility:visible可以让子孙节点显示；

（3）修改常规文档流中元素的 display 通常会造成文档的重排，但是修改visibility属性只会造成本元素的重绘；

（4）如果使用读屏器，设置为display:none的内容不会被读取，设置为visibility:hidden的内容会被读取。

7）水平、垂直居中方法

* 利用绝对定位，先将元素的左上角通过top:50%和left:50%定位到页面的中心，然后再通过translate来调整元素的中心点到页面的中心。该方法需要**考虑浏览器兼容问题。**

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

* 利用绝对定位，设置四个方向的值都为0，并将margin设置为auto，由于宽高固定，因此对应方向实现平分，可以实现水平和垂直方向上的居中。该方法适用于**盒子有宽高**的情况：

.parent {  
   position: relative;  
}  
  
.child {  
   position: absolute;  
   top: 0;  
   bottom: 0;  
   left: 0;  
   right: 0;  
   margin: auto;  
}

* 利用绝对定位，先将元素的左上角通过top:50%和left:50%定位到页面的中心，然后再通过margin负值来调整元素的中心点到页面的中心。该方法适用于**盒子宽高已知**的情况

.parent {  
   position: relative;  
}  
  
.child {  
   position: absolute;  
   top: 50%;  
   left: 50%;  
   margin-top: -50px;     /\* 自身 height 的一半 \*/  
   margin-left: -50px;    /\* 自身 width 的一半 \*/  
}

* 使用flex布局，通过align-items:center和justify-content:center设置容器的垂直和水平方向上为居中对齐，然后它的子元素也可以实现垂直和水平的居中。该方法要**考虑兼容的问题**，该方法在移动端用的较多：

.parent {

display: flex;

justify-content:center;

align-items:center;

}

8） position绝对定位和相对定位

position有以下属性值：

| **属性值** | **概述** |
| --- | --- |
| absolute | 生成绝对定位的元素，相对于static定位以外的一个父元素进行定位。元素的位置通过left、top、right、bottom属性进行规定。 |
| relative | 生成相对定位的元素，相对于其原来的位置进行定位。元素的位置通过left、top、right、bottom属性进行规定。 |
| fixed | 生成固定定位的元素，指定元素相对于屏幕视⼝（viewport）的位置来指定元素位置。元素的位置在屏幕滚动时不会改变，⽐如回到顶部的按钮⼀般都是⽤此定位⽅式。 |
| static | 默认值，没有定位，元素出现在正常的文档流中，会忽略 top, bottom, left, right 或者 z-index 声明，块级元素从上往下纵向排布，⾏级元素从左向右排列。 |
| inherit | 规定从父元素继承position属性的值 |

* **relative：** 元素的定位永远是相对于元素自身位置的，和其他元素没关系，也不会影响其他元素。
* **fixed：** 元素的定位是相对于 window （或者 iframe）边界的，和其他元素没有关系。但是它具有破坏性，会导致其他元素位置的变化。
* **absolute：** 元素的定位相对于前两者要复杂许多。如果为 absolute 设置了 top、left，浏览器会根据什么去确定它的纵向和横向的偏移量呢？答案是浏览器会递归查找该元素的所有父元素，如果找到一个设置了position:relative/absolute/fixed的元素，就以该元素为基准定位，如果没找到，就以浏览器边界定位。

使用场景：

相对定位的应用场景：用于对元素进行微调；配合绝对定位来使用。

两栏布局

在企业开发中一般不单独使用相对定位和绝对定位，而是结合一起使用。一般“子绝父相”，即子元素用绝对定位，父元素用相对定位。但凡说到定位或一个盒子覆盖在另一个盒子上都要想到“子绝父相”。

9） flex布局

Flex 是 Flexible Box 的缩写，意为"弹性布局"，用来为盒状模型提供最大的灵活性。

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"。

flex:1 ==> flex:1 1 auto 放大 缩小

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

flex-grow默认为0，即如果存在剩余空间，也不放大。

flex-shrink默认为1，即如果空间不足，该项目将缩小。

10）响应式布局 （em rem）

* 响应式布局指的是同一页面在不同屏幕尺寸下有不同的布局。传统的开发方式是PC端开发一套，手机端再开发一套，而使用响应式布局只要开发一套就够，缺点就是CSS比较繁琐。
* 实现方案：
  + 媒体查询：CSS3媒体查询可以让我们针对不同的媒体类型定义不同的样式，当重置浏览器窗口大小的过程中，页面也会根据浏览器的宽度和高度重新渲染页面。
  + 百分比布局：通过百分比单位，可以使得浏览器中组件的宽和高随着浏览器的高度的变化而变化，从而实现响应式的效果。
  + REM是CSS3新增的单位，并且移动端的支持度很高，Android2.x+,ios5+都支持。rem单位都是相对于根元素html的font-size来决定大小的,根元素的font-size相当于提供了一个基准，当页面的size发生变化时，只需要改变font-size的值，那么以rem为固定单位的元素的大小也会发生响应的变化。 因此，如果通过rem来实现响应式的布局，只需要根据视图容器的大小，动态的改变font-size即可（而em是相对于父元素的）。

11） css选择器以及优先级

对于选择器的**优先级**：

* 标签选择器、伪元素选择器：1
* 类选择器、伪类选择器、属性选择器：10
* id 选择器：100
* 内联样式：1000

**注意事项：**

* !important声明的样式的优先级最高；
* 如果优先级相同，则最后出现的样式生效；
* 继承得到的样式的优先级最低；
* 通用选择器（\*）、子选择器（>）和相邻同胞选择器（+）并不在这四个等级中，所以它们的权值都为 0 ；
* 样式表的来源不同时，优先级顺序为：内联样式 > 内部样式 > 外部样式 > 浏览器用户自定义样式 > 浏览器默认样式。

| **选择器** | **格式** | **优先级权重** |
| --- | --- | --- |
| id选择器 | #id | 100 |
| 类选择器 | .classname | 10 |
| 属性选择器 | a[ref=“eee”] | 10 |
| 伪类选择器 | li:last-child | 10 |
| 标签选择器 | div | 1 |
| 伪元素选择器 | li:after | 1 |
| 相邻兄弟选择器 | h1+p | 0 |
| 子选择器 | ul>li | 0 |
| 后代选择器 | li a | 0 |
| 通配符选择器 | \* | 0 |

12）重排与重绘，CSS哪些属性会引起重排

**（1）回流（重排）**

当渲染树中部分或者全部元素的尺寸、结构或者属性发生变化时，浏览器会重新渲染部分或者全部文档的过程就称为**回流**。

下面这些操作会导致回流：

* 页面的首次渲染
* 浏览器的窗口大小发生变化
* 元素的内容发生变化
* 元素的尺寸或者位置发生变化
* 元素的字体大小发生变化
* 激活CSS伪类
* 查询某些属性或者调用某些方法
* 添加或者删除可见的DOM元素

在触发回流（重排）的时候，由于浏览器渲染页面是基于流式布局的，所以当触发回流时，会导致周围的DOM元素重新排列，它的影响范围有两种：

* 全局范围：从根节点开始，对整个渲染树进行重新布局
* 局部范围：对渲染树的某部分或者一个渲染对象进行重新布局

**（2）重绘**

当页面中某些元素的样式发生变化，但是不会影响其在文档流中的位置时，浏览器就会对元素进行重新绘制，这个过程就是**重绘**。

下面这些操作会导致回流：

* color、background 相关属性：background-color、background-image 等
* outline 相关属性：outline-color、outline-width 、text-decoration
* border-radius、visibility、box-shadow

注意： **当触发回流时，一定会触发重绘，但是重绘不一定会引发回流。**

13）减少回流与重绘：

* 操作DOM时，尽量在低层级的DOM节点进行操作
* 不要使用table布局， 一个小的改动可能会使整个table进行重新布局
* 使用CSS的表达式
* 不要频繁操作元素的样式，对于静态页面，可以修改类名，而不是样式。
* 使用absolute或者fixed，使元素脱离文档流，这样他们发生变化就不会影响其他元素
* 避免频繁操作DOM，可以创建一个文档片段documentFragment，在它上面应用所有DOM操作，最后再把它添加到文档中
* 将元素先设置display: none，操作结束后再把它显示出来。因为在display属性为none的元素上进行的DOM操作不会引发回流和重绘。
* 将DOM的多个读操作（或者写操作）放在一起，而不是读写操作穿插着写。这得益于**浏览器的渲染队列机制**。

1） 输入URL到展示页面的全过程（★★★★★）

分为4 个步骤：

1. 当发送一个URL 请求时，浏览器都会开启一个线程来处理这个请求，同时在远程DNS 服务器上启动一个DNS查询。这能使浏览器获得请求对应的IP 地址。

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。

14）浏览器如何渲染页面

浏览器利用自己内部的工作机制，把请求的静态资源和html代码进行渲染，

首先浏览器解析HTML文件构建DOM树，

然后解析CSS文件构建渲染树，

等到渲染树构建完成后，浏览器开始布局渲染树并将其绘制到屏幕上。

这个过程比较复杂，涉及到两个概念: reflow(回流、重排)和repain(重绘)。

DOM节点中的各个元素都是以盒模型的形式存在，这些都需要浏览器去计算其位置和大小等，这个过程称为relow;当盒模型的位置,大小以及其他属性，如颜色,字体,等确定下来之后，浏览器便开始绘制内容，这个过程称为repain。

页面在首次加载时会经历reflow和repain。

JS的解析是由浏览器中的JS解析引擎完成的。

JS是单线程运行，JS有可能修改DOM结构，意味着JS执行完成前，后续所有资源的下载是没有必要的，所以JS是单线程，会阻塞后续资源下载。

15） html css js联系与区别

* HTML（HyperText Markup Language）：超文本标记语言。从**语义**的角度描述页面的**结构**。相当于人的身体组织结构。
* CSS（Cascading Style Sheets）：层叠样式表。从**审美**的角度美化页面的**样式**。相当于人的衣服和打扮。
* JS：JavaScript。从**交互**的角度描述页面的**行为**。相当于人的动作，让人有生命力。

16）网站页面加载顺序

IE下载的顺序是从上到下，渲染的顺序也是从上到下，下载和渲染是同时进行的。

在渲染到页面的某一部分时，其上面的所有部分都已经下载完成（并不是说所有相关联的元素都已经下载完）。

如果遇到语义解释性的标签嵌入文件（JS脚本，CSS样式），那么此时IE的下载过程会启用单独连接进行下载。

样式表在下载完成后，将和以前下载的所有样式表一起进行解析，解析完成后，将对此前所有元素（含以前已经渲染的）重新进行渲染。

JS、CSS中如有重定义，后定义函数将覆盖前定义函数。

17）CSS盒子模型

CSS3中的盒模型有以下两种：标准盒子模型、IE盒子模型

盒模型都是由四个部分组成的，分别是margin、border、padding和content。

标准盒模型和IE盒模型的区别在于设置width和height时，所对应的范围不同：

* 标准盒模型的width和height属性的范围只包含了content，
* IE盒模型的width和height属性的范围包含了border、padding和content。

可以通过修改元素的box-sizing属性来改变元素的盒模型：

* box-sizeing: content-box表示标准盒模型（默认值）
* box-sizeing: border-box表示IE盒模型（怪异盒模型）

18） css样式冲突怎么解决

**1. 细化选择符 2. 再写一次选择器名 3. 改变CSS样式表中的顺序**

4 JavaScript

1）箭头函数、普通函数作用和区别（★★★★★）

**（1）箭头函数比普通函数更加简洁**

* 如果没有参数，就直接写一个空括号即可
* 如果只有一个参数，可以省去参数的括号
* 如果有多个参数，用逗号分割

**（2）箭头函数没有自己的this**

箭头函数不会创建自己的this， 所以它没有自己的this，它只会在自己作用域的上一层继承this。所以箭头函数中this的指向在它在定义时已经确定了，之后不会改变。

**（4）call()、apply()、bind()等方法不能改变箭头函数中this的指向**

**（5）箭头函数不能作为构造函数使用**

构造函数在new的步骤在上面已经说过了，实际上第二步就是将函数中的this指向该对象。 但是由于箭头函数时没有自己的this的，且this指向外层的执行环境，且不能改变指向，所以不能当做构造函数使用。

**（6）箭头函数没有自己的arguments**

**（7）箭头函数没有prototype**

**（8）箭头函数不能用作Generator函数，不能使用yeild关键字**

2）call、apply、bind作用和区别（★★★★★）

call 方法接收的参数，第一个是 this 绑定的对象，后面的其余参数是传入函数执行的参数。

apply 方法接收两个参数：一个是 this 绑定的对象，一个是参数数组。

bind 方法通过传入一个对象，返回一个 this 绑定了传入对象的新函数。这个函数的 this 指向除了使用 new 时会被改变，其他情况下都不会改变。

3）事件冒泡、捕获

**冒泡：当一个事件发生在一个元素上，它会首先运行在该元素上的处理程序，然后运行其父元素上的处理程序，然后一直向上到其他祖先上的处理程序。**

[DOM 事件](http://www.w3.org/TR/DOM-Level-3-Events/)标准描述了事件传播的 3 个阶段：

1. 捕获阶段（Capturing phase）—— 事件（从 Window）向下走近元素。
2. 目标阶段（Target phase）—— 事件到达目标元素。
3. 冒泡阶段（Bubbling phase）—— 事件从元素上开始冒泡。

4）垃圾回收机制（★★★★）

**垃圾回收**：JavaScript代码运行时，需要分配内存空间来储存变量和值。当变量不在参与运行时，就需要系统收回被占用的内存空间，这就是垃圾回收。

**回收机制**：

* Javascript 具有自动垃圾回收机制，会定期对那些不再使用的变量、对象所占用的内存进行释放，原理就是找到不再使用的变量，然后释放掉其占用的内存。
* JavaScript中存在两种变量：局部变量和全局变量。全局变量的生命周期会持续要页面卸载；而局部变量声明在函数中，它的生命周期从函数执行开始，直到函数执行结束，在这个过程中，局部变量会在堆或栈中存储它们的值，当函数执行结束后，这些局部变量不再被使用，它们所占有的空间就会被释放。
* 不过，当局部变量被外部函数使用时，其中一种情况就是闭包，在函数执行结束后，函数外部的变量依然指向函数内部的局部变量，此时局部变量依然在被使用，所以不会回收。

**（2）垃圾回收的方式**

浏览器通常使用的垃圾回收方法：标记清除。

**标记清除**

* 标记清除是浏览器常见的垃圾回收方式，当变量进入执行环境时，就标记这个变量“进入环境”，被标记为“进入环境”的变量是不能被回收的，因为他们正在被使用。当变量离开环境时，就会被标记为“离开环境”，被标记为“离开环境”的变量会被内存释放。

以下四种情况会造成内存的泄漏：

1意外的全局变量：由于使用未声明的变量，而意外的创建了一个全局变量，而使这个变量一直留在内存中无法被回收。

2被遗忘的计时器或回调函数：设置了 setInterval 定时器，而忘记取消它，如果循环函数有对外部变量的引用的话，那么这个变量会被一直留在内存中，而无法被回收。

3脱离 DOM 的引用： 获取一个 DOM 元素的引用，而后面这个元素被删除，由于一直保留了对这个元素的引用，所以它也无法被回收。

4闭包：不合理的使用闭包，从而导致某些变量一直被留在内存当中。

5）深拷贝、浅拷贝（★★★★★）

浅拷贝是指，将一个对象的属性值复制到另一个对象，如果拷贝的是基本数据类型，拷贝的就是基本数据类型的值，如果有的属性的值为引用类型的话，那么会将这个引用的地址复制给对象，因此两个对象会有同一个引用类型的引用。

实现方式：

（1）Object.assign()

**（2）扩展运算符**

深拷贝相对浅拷贝而言，如果遇到属性值为引用类型的时候，它新建一个引用类型并将对应的值复制给它，因此对象获得的一个新的引用类型而不是一个原有类型的引用。

实现方式：

**JSON.stringify()**

* JSON.parse(JSON.stringify(obj))是目前比较常用的深拷贝方法之一，它的原理就是利用JSON.stringify 将js对象序列化（JSON字符串），再使用JSON.parse来反序列化(还原)js对象。
* 这个方法可以简单粗暴的实现深拷贝，但是还存在问题，拷贝的对象中如果有函数，undefined，symbol，当使用过JSON.stringify()进行处理之后，都会消失。

let obj1 = { a: 0,

b: {

c: 0

}

};

let obj2 = JSON.parse(JSON.stringify(obj1));

obj1.a = 1;

obj1.b.c = 1;

console.log(obj1); // {a: 1, b: {c: 1}}

console.log(obj2); // {a: 0, b: {c: 0}}

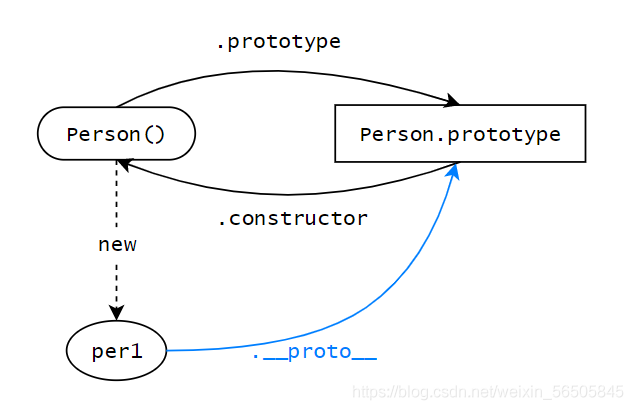
**函数库lodash的\_.cloneDeep方法**

6）数组方法解析，遍历数组、对象，for in for of的区别（★★★★★）

for...in 循环主要是为了遍历对象而生，不适用于遍历数组；

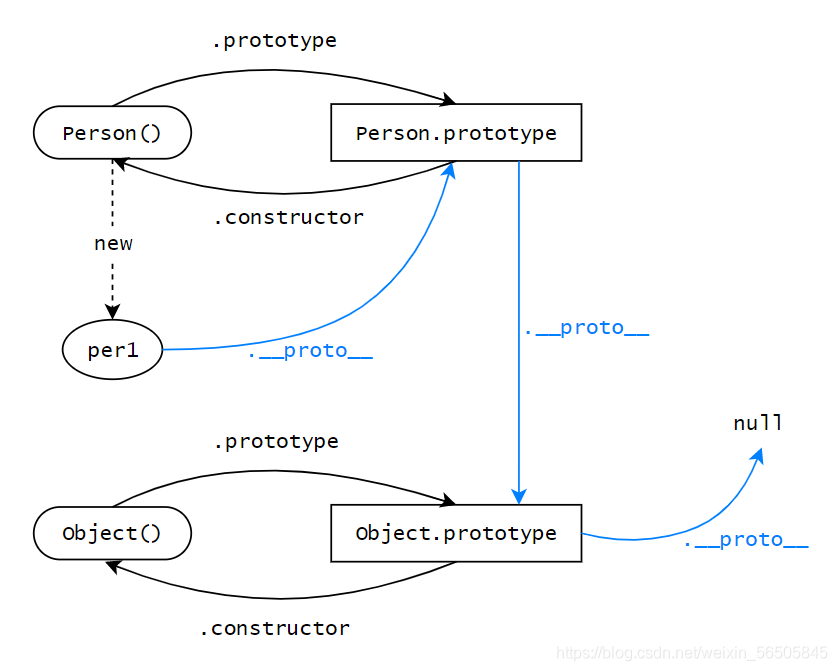
for...of 循环可以用来遍历数组、类数组对象，字符串、Set、Map 以及 Generator 对象。

7）原型链、class继承（★★★★）



①所有引用类型（对象）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，这个对象就是所谓的原型对象  
③所有引用类型的\_\_proto\_\_属性指向它构造函数的prototype

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，即它的原型对象里找这个属性，这个原型对象又会有自己的原型，于是就这样一直找下去，也就是原型链的概念。原型链的尽头一般来说都是 Object.prototype 所以这就是新建的对象为什么能够使用 toString() 等方法的原因。



8）实现继承的方式

（1）第一种是以原型链的方式来实现继承，但是这种实现方式存在的缺点是，在包含有引用类型的数据时，会被所有的实例对象所共享，容易造成修改的混乱。还有就是在创建子类型的时候不能向超类型传递参数。

（2）第二种方式是使用借用构造函数的方式，这种方式是通过在子类型的函数中调用超类型的构造函数来实现的，这一种方法解决了不能向超类型传递参数的缺点，但是它存在的一个问题就是无法实现函数方法的复用，并且超类型原型定义的方法子类型也没有办法访问到。

9）AJAX、axios、fetch等的使用（★★★★★）

ajax、axios、fetch的区别：

**（1）AJAX** (无刷新获取数据)

Ajax 即“Asynchronous Javascript And XML”（异步 JavaScript 和 XML），是指一种创建交互式网页应用的网页开发技术。它是一种在**无需重新加载整个网页**的情况下，能够更新部分网页的技术。通过在后台与服务器进行少量数据交换，Ajax 可以使网页实现异步更新。这意味着可以在不重新加载整个网页的情况下，对网页的某部分进行更新。传统的网页（不使用 Ajax）如果需要更新内容，必须重载整个网页页面。其缺点如下：

* 本身是针对MVC编程，不符合前端MVVM的浪潮
* 基于原生XHR开发，XHR本身的架构不清晰
* 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不友好。

**（2）Fetch**

fetch号称是AJAX的替代品，是在ES6出现的，使用了ES6中的promise对象。Fetch是基于promise设计的。Fetch的代码结构比起ajax简单多。**fetch不是ajax的进一步封装，而是原生js，没有使用XMLHttpRequest对象**。

fetch的优点：

* 语法简洁，更加语义化
* 基于标准 Promise 实现，支持 async/await
* 更加底层，提供的API丰富（request, response）
* 脱离了XHR，是ES规范里新的实现方式

fetch的缺点：

* fetch只对网络请求报错，对400，500都当做成功的请求，服务器返回 400，500 错误码时并不会 reject，只有网络错误这些导致请求不能完成时，fetch 才会被 reject。
* fetch默认不会带cookie，需要添加配置项： fetch(url, {credentials: 'include'})
* fetch不支持abort，不支持超时控制，使用setTimeout及Promise.reject的实现的超时控制并不能阻止请求过程继续在后台运行，造成了流量的浪费
* fetch没有办法原生监测请求的进度，而XHR可以

**（3）Axios** Axios 是一种基于Promise封装的HTTP客户端，其特点如下：

* 浏览器端发起XMLHttpRequests请求
* node端发起http请求
* 支持Promise API
* 监听请求和返回
* 对请求和返回进行转化
* 取消请求
* 自动转换json数据
* 客户端支持抵御XSRF攻击

10）JS的数据类型

JavaScript共有八种数据类型，分别是Boolean、Number、String、Undefined、Null、 Object、Symbol、BigInt。

其中 Symbol 和 BigInt 是ES6 中新增的数据类型：

* Symbol 代表创建后独一无二且不可变的数据类型，它主要是为了解决可能出现的全局变量冲突的问题。
* BigInt 是一种数字类型的数据，它可以表示任意精度格式的整数，使用 BigInt 可以安全地存储和操作大整数，即使这个数已经超出了 Number 能够表示的安全整数范围。

11） 数据类型的判断

**（1）typeof**

console.log(typeof 2);               // number  
console.log(typeof true);            // boolean  
console.log(typeof 'str');           // string  
console.log(typeof []);              // object      
console.log(typeof function(){});    // function  
console.log(typeof {});              // object  
console.log(typeof undefined);       // undefined  
console.log(typeof null);            // object

其中数组、对象、null都会被判断为object，其他判断都正确。

**（2）instanceof**

instanceof可以正确判断对象的类型，**其内部运行机制是判断在其原型链中能否找到该类型的原型**。

console.log(2 instanceof Number);                    // false  
console.log(true instanceof Boolean);                // false   
console.log('str' instanceof String);                // false   
console.log([] instanceof Array);                    // true  
console.log(function(){} instanceof Function);       // true  
console.log({} instanceof Object);                   // true

可以看到，instanceof**只能正确判断引用数据类型**，而不能判断基本数据类型。instanceof 运算符可以用来测试一个对象在其原型链中是否存在一个构造函数的 prototype 属性。

**（3） constructor**

console.log((2).constructor === Number); // true  
console.log((true).constructor === Boolean); // true  
console.log(('str').constructor === String); // true  
console.log(([]).constructor === Array); // true  
console.log((function() {}).constructor === Function); // true  
console.log(({}).constructor === Object); // true

constructor有两个作用，一是判断数据的类型，二是对象实例通过 constrcutor 对象访问它的构造函数。需要注意，如果创建一个对象来改变它的原型，constructor就不能用来判断数据类型了：

function Fn(){};  
Fn.prototype = new Array();  
  
var f = new Fn();  
  
console.log(f.constructor===Fn);    // false  
console.log(f.constructor===Array); // true

**（4）Object.prototype.toString.call()**

Object.prototype.toString.call() 使用 Object 对象的原型方法 toString 来判断数据类型：

var a = Object.prototype.toString;  
console.log(a.call(2));  
console.log(a.call(true));  
console.log(a.call('str'));  
console.log(a.call([]));  
console.log(a.call(function(){}));  
console.log(a.call({}));  
console.log(a.call(undefined));  
console.log(a.call(null));

同样是检测对象obj调用toString方法，obj.toString()的结果和Object.prototype.toString.call(obj)的结果不一样，这是为什么？

这是因为toString是Object的原型方法，而Array、function等**类型作为Object的实例，都重写了toString方法**。因此，在想要得到对象的具体类型时，应该调用Object原型上的toString方法。

12）鼠标点击和松开事件绑

**mousedown/mouseup**

在元素上点击/释放鼠标按钮。

**mouseover/mouseout**

鼠标指针从一个元素上移入/移出。

**mousemove**

鼠标在元素上的每个移动都会触发此事件。

**click**

如果使用的是鼠标左键，则在同一个元素上的 mousedown 及 mouseup 相继触发后，触发该事件。

一个用户操作可能会触发多个事件。

例如，点击鼠标左键，在鼠标左键被按下时，会首先触发 mousedown，然后当鼠标左键被释放时，会触发 mouseup 和 click。

在单个动作触发多个事件时，事件的顺序是固定的。也就是说，会遵循 mousedown → mouseup → click 的顺序调用处理程序。

13） var、let、const的区别

**（1）块级作用域：** 块作用域由 { }包括，let和const具有块级作用域，var不存在块级作用域。

**（2）变量提升：** var存在变量提升，let和const不存在变量提升，即在变量只能在声明之后使用，否在会报错。

**（3）给全局添加属性：** 浏览器的全局对象是window，Node的全局对象是global。var声明的变量为全局变量，并且会将该变量添加为全局对象的属性，但是let和const不会。

**（4）重复声明：** var声明变量时，可以重复声明变量，后声明的同名变量会覆盖之前声明的遍历。const和let不允许重复声明变量。

**（5）暂时性死区：** 在使用let、const命令声明变量之前，该变量都是不可用的。这在语法上，称为**暂时性死区**。使用var声明的变量不存在暂时性死区。

**（6）初始值设置：** 在变量声明时，var 和 let 可以不用设置初始值。而const声明变量必须设置初始值。

14）ES6新增

1 ES6 在变量的声明和定义方面增加了let、const 声明变量

2 ES6 也引入了新的数据类型symbol，新的数据结构set 和map

3 es6提出了箭头函数，

4 为解决异步回调问题，引入了promise

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

5还有最为吸引人了实现Class 和模块，通过Class 可以更好的面向对象编程

重要的特性：

6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

7 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，

8 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

9 ES6 提出了“模板语法”的概念

15）怎么实现图片随鼠标光标的移动

基础的拖放算法如下所示：

1. 在 mousedown 上 —— 根据需要准备要移动的元素（也许创建一个它的副本，向其中添加一个类或其他任何东西）。
2. 然后在 mousemove 上，通过更改 position:absolute 情况下的 left/top 来移动它。
3. 在 mouseup 上 —— 执行与完成的拖放相关的所有行为。

ball.onmousedown = function(event) {

// (1) 准备移动：确保 absolute，并通过设置 z-index 以确保球在顶部

ball.style.position = 'absolute';

ball.style.zIndex = 1000;

// 将其从当前父元素中直接移动到 body 中

// 以使其定位是相对于 body 的

document.body.append(ball);

// 现在球的中心在 (pageX, pageY) 坐标上

function moveAt(pageX, pageY) {

ball.style.left = pageX - ball.offsetWidth / 2 + 'px';

ball.style.top = pageY - ball.offsetHeight / 2 + 'px';

}

// 将我们绝对定位的球移到指针下方

moveAt(event.pageX, event.pageY);

function onMouseMove(event) {

moveAt(event.pageX, event.pageY);

}

// (2) 在 mousemove 事件上移动球

document.addEventListener('mousemove', onMouseMove);

// (3) 放下球，并移除不需要的处理程序

ball.onmouseup = function() {

document.removeEventListener('mousemove', onMouseMove);

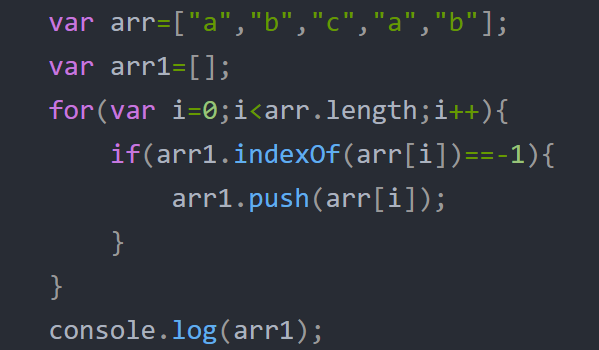
ball.onmouseup = null;

};

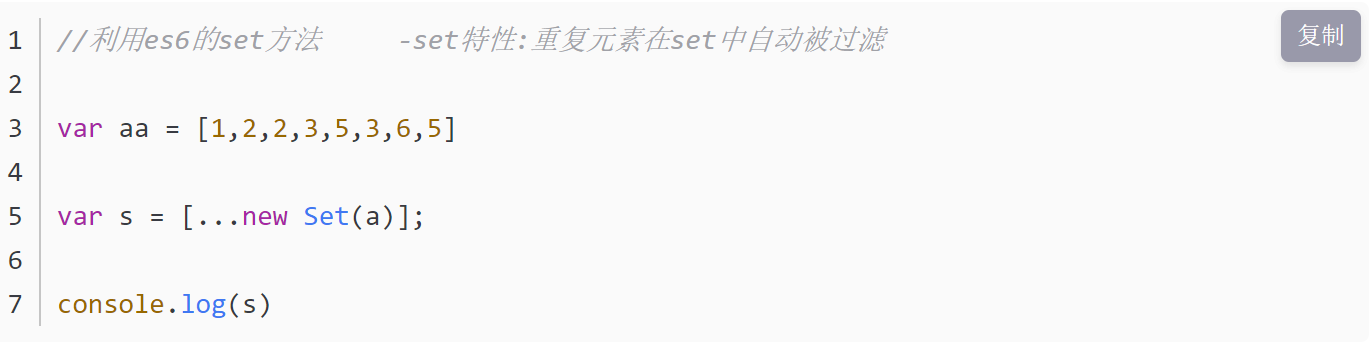
};

16）数组去重

法一：使用[indexOf](https://so.csdn.net/so/search?q=indexOf&spm=1001.2101.3001.7020)数组去重



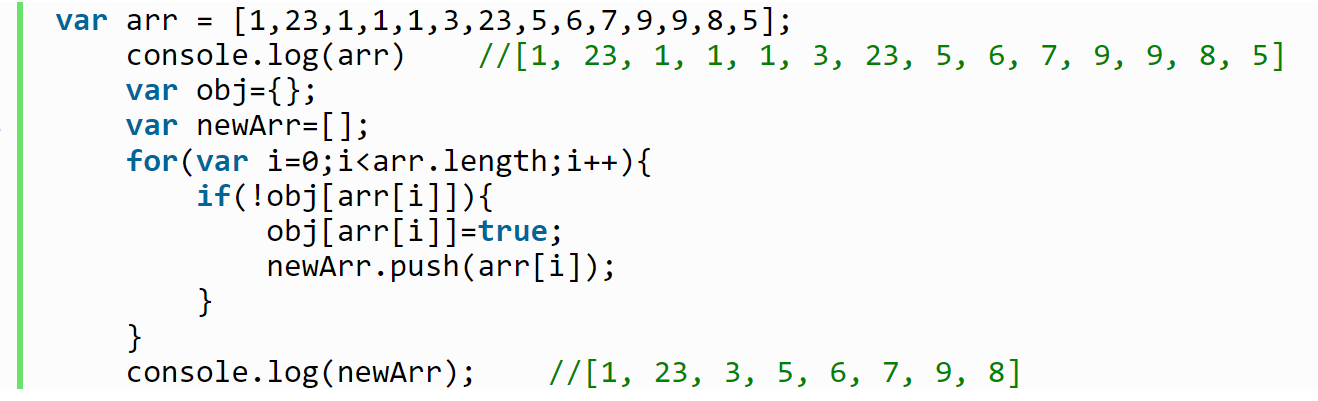
法二：ES6 Set 去重；Array.from(new Set(array))



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。



17）对作用域的理解

**1）全局作用域和函数作用域**

（1）全局作用域

* 最外层函数和最外层函数外面定义的变量拥有全局作用域
* 所有未定义直接赋值的变量自动声明为全局作用域
* 所有window对象的属性拥有全局作用域
* 全局作用域有很大的弊端，过多的全局作用域变量会污染全局命名空间，容易引起命名冲突。

（2）函数作用域

* 函数作用域声明在函数内部的变零，一般只有固定的代码片段可以访问到
* 作用域是分层的，内层作用域可以访问外层作用域，反之不行

**2）块级作用域**

* 使用ES6中新增的let和const指令可以声明块级作用域，块级作用域可以在函数中创建也可以在一个代码块中的创建（由{ }包裹的代码片段）
* let和const声明的变量不会有变量提升，也不可以重复声明
* 在循环中比较适合绑定块级作用域，这样就可以把声明的计数器变量限制在循环内部。

**作用域链：** 在当前作用域中查找所需变量，但是该作用域没有这个变量，那这个变量就是自由变量。如果在自己作用域找不到该变量就去父级作用域查找，依次向上级作用域查找，直到访问到window对象就被终止，这一层层的关系就是作用域链。

作用域链的作用是**保证对执行环境有权访问的所有变量和函数的有序访问，通过作用域链，可以访问到外层环境的变量和函数。**

作用域链的本质上是一个指向变量对象的指针列表。变量对象是一个包含了执行环境中所有变量和函数的对象。作用域链的前端始终都是当前执行上下文的变量对象。全局执行上下文的变量对象（也就是全局对象）始终是作用域链的最后一个对象。

当查找一个变量时，如果当前执行环境中没有找到，可以沿着作用域链向后查找。

18）闭包

**闭包是指有权访问另一个函数作用域中变量的函数**，创建闭包的最常见的方式就是在一个函数内创建另一个函数，创建的函数可以访问到当前函数的局部变量。

闭包有两个常用的用途；

* 闭包的第一个用途是使我们在函数外部能够访问到函数内部的变量。通过使用闭包，可以通过在外部调用闭包函数，从而在外部访问到函数内部的变量，可以使用这种方法来创建私有变量。
* 闭包的另一个用途是使已经运行结束的函数上下文中的变量对象继续留在内存中，因为闭包函数保留了这个变量对象的引用，所以这个变量对象不会被回收。

19）JS实现判断对象是否为空对象

1.将json对象转化为json字符串，再判断该字符串是否为"{}"  
var data = {};  
var b = (JSON.stringify(data) == "{}");  
alert(b);//true  
2.for in 循环判断  
var obj = {};  
var b = function() {  
for(var key in obj) {  
return false;  
}  
return true;  
}  
alert(b());//true  
3.Object.getOwnPropertyNames()方法  
此方法是使用Object对象的getOwnPropertyNames方法，获取到对象中的属性名，存到一个数组中，返回数组对象，我们可以通过判断数组的length来判断此对象是否为空  
注意：此方法不兼容ie8，其余浏览器没有测试  
var data = {};  
var arr = Object.getOwnPropertyNames(data);  
alert(arr.length == 0);//true

20）js的事件循环机制？

JavaScript代码的执行过程中，除了依靠函数调用栈来搞定函数的执行顺序外，还依靠任务队列(task queue)来搞定另外一些代码的执行。整个执行过程，我们称为事件循环过程。

任务队列又分为macro-task（宏任务）与micro-task（微任务）

因为 js 是单线程运行的，在代码执行时，通过将不同函数的执行上下文压入执行栈中来保证代码的有序执行。在执行同步代码时，如果遇到异步事件，js 引擎并不会一直等待其返回结果，而是会将这个事件挂起，再将异步事件对应的回调加入到一个任务队列中等待执行，继续执行执行栈中的其他任务。当当前执行栈中的事件执行完毕后，js 引擎首先会判断微任务队列中是否有任务可以执行，如果有就将微任务队首的事件压入栈中执行。当微任务队列中的任务都执行完成后再去执行宏任务队列中的任务

Event Loop 执行顺序如下所示：

* 首先执行同步代码，这属于宏任务
* 当执行完所有同步代码后，执行栈为空，查询是否有异步代码需要执行
* 执行所有微任务
* 当执行完所有微任务后，如有必要会渲染页面
* 然后开始下一轮 Event Loop，执行宏任务中的异步代码

21）说一下防抖和节流，口诉一下代码实现？

- 函数防抖是指在事件被触发 n 秒后再执行回调，如果在这 n 秒内事件又被触发，则重新计时。这可以使用在一些点击请求的事件上，避免因为用户的多次点击向后端发送多次请求。

- 函数节流是指规定一个单位时间，在这个单位时间内，只能有一次触发事件的回调函数执行，如果在同一个单位时间内某事件被触发多次，只有一次能生效。节流可以使用在 scroll 函数的事件监听上，通过事件节流来降低事件调用的频率。

防抖函数的应用场景：

- 按钮提交场景：防⽌多次提交按钮，只执⾏最后提交的⼀次

- 服务端验证场景：表单验证需要服务端配合，只执⾏⼀段连续的输⼊事件的最后⼀次，还有搜索联想词功能类似⽣存环境请⽤lodash.debounce

**-用户在输入框中连续输入一串字符时，可以通过防抖策略，只在输入完后，才执行查询的请求，这样可以有效减少请求次数，节约请求资源；**

节流函数的适⽤场景：

- 拖拽场景：固定时间内只执⾏⼀次，防⽌超⾼频次触发位置变动

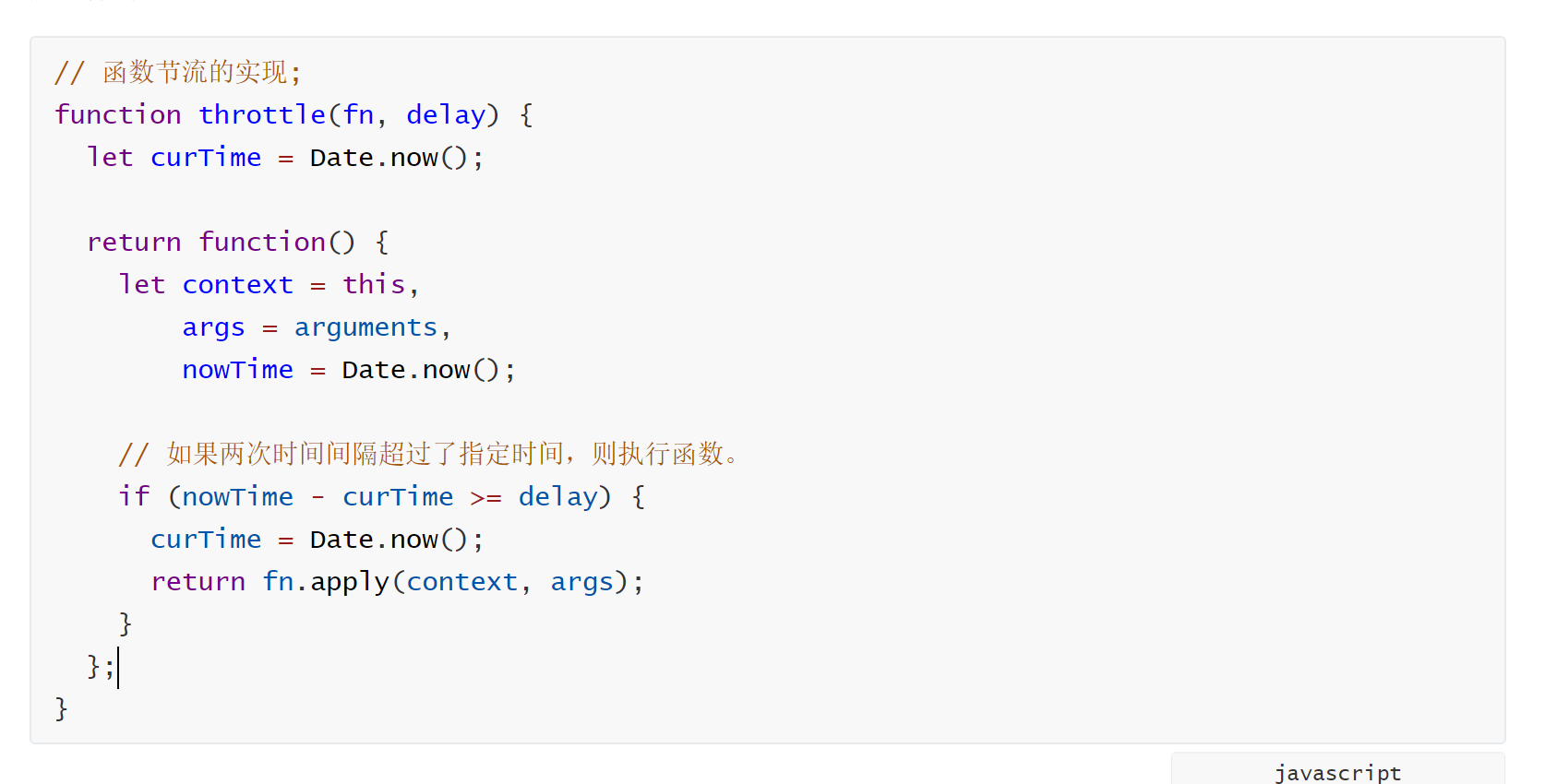
- 缩放场景：监控浏览器resize

- 动画场景：避免短时间内多次触发动画引起性能问题

**-鼠标连续不断地触发某事件（如点击），只在单位时间内只触发一次；**

**-懒加载时要监听计算滚动条的位置，但不必每次滑动都触发，可以降低计算的频率，而不必去浪费 CPU 资源；**





5 Vue

1） MVVM、MVC、MVP的区别（★★★★）

MVC、MVP 和 MVVM 是三种常见的软件架构设计模式，主要通过分离关注点的方式来组织代码结构，优化开发效率。

在开发单页面应用时，往往一个路由页面对应了一个脚本文件，所有的页面逻辑都在一个脚本文件里。页面的渲染、数据的获取，对用户事件的响应所有的应用逻辑都混合在一起，这样在开发简单项目时，可能看不出什么问题，如果项目变得复杂，那么整个文件就会变得冗长、混乱，这样对项目开发和后期的项目维护是非常不利的。

**（1）MVC**

MVC 通过分离 Model、View 和 Controller 的方式来组织代码结构。其中 View 负责页面的显示逻辑，Model 负责存储页面的业务数据，以及对相应数据的操作。并且 View 和 Model 应用了观察者模式，当 Model 层发生改变的时候它会通知有关 View 层更新页面。Controller 层是 View 层和 Model 层的纽带，它主要负责用户与应用的响应操作，当用户与页面产生交互的时候，Controller 中的事件触发器就开始工作了，通过调用 Model 层，来完成对 Model 的修改，然后 Model 层再去通知 View 层更新。

（2）MVVM

MVVM 分为 Model、View、ViewModel：

* Model代表数据模型，数据和业务逻辑都在Model层中定义；
* View代表UI视图，负责数据的展示；
* ViewModel负责监听Model中数据的改变并且控制视图的更新，处理用户交互操作；

Model和View并无直接关联，而是通过ViewModel来进行联系的，Model和ViewModel之间有着双向数据绑定的联系。因此当Model中的数据改变时会触发View层的刷新，View中由于用户交互操作而改变的数据也会在Model中同步。

这种模式实现了 Model和View的数据自动同步，因此开发者只需要专注于数据的维护操作即可，而不需要自己操作DOM。

**（3）MVP**

MVP 模式与 MVC 唯一不同的在于 Presenter 和 Controller。在 MVC 模式中使用观察者模式，来实现当 Model 层数据发生变化的时候，通知 View 层的更新。这样 View 层和 Model 层耦合在一起，当项目逻辑变得复杂的时候，可能会造成代码的混乱，并且可能会对代码的复用性造成一些问题。MVP 的模式通过使用 Presenter 来实现对 View 层和 Model 层的解耦。MVC 中的Controller 只知道 Model 的接口，因此它没有办法控制 View 层的更新，MVP 模式中，View 层的接口暴露给了 Presenter 因此可以在 Presenter 中将 Model 的变化和 View 的变化绑定在一起，以此来实现 View 和 Model 的同步更新。这样就实现了对 View 和 Model 的解耦，Presenter 还包含了其他的响应逻辑。

2）Vue-cli、Vue-router、VueX等的使用（★★★★★）

vuex：如果业务逻辑复杂，很多组件之间需要同时处理一些公共的数据，这个时候采用上面这一些方法可能不利于项目的维护。这个时候可以使用 vuex ，vuex 的思想就是将这一些公共的数据抽离出来，将它作为一个全局的变量来管理，然后其他组件就可以对这个公共数据进行读写操作，这样达到了解耦的目的。

3）组件通信方式（★★★★★）

**（1）父子组件间通信**

* 父组件通过 props 属性向子组件传递数据，然后父组件在子组件上注册监听事件，子组件通过 emit 触发事件来向父组件发送数据。
* 通过 ref 属性给子组件设置一个名字。父组件通过 $refs 组件名来获得子组件，这样也可以实现通信。

**（2）兄弟组件间通信**

* 使用 eventBus 的方法，它的本质是通过创建一个空的 Vue 实例来作为消息传递的对象，通信的组件引入这个实例，通过在这个实例上监听和触发事件，来实现消息的传递。

**（3）任意组件之间**

* 使用 eventBus ，其实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。

如果业务逻辑复杂，很多组件之间需要同时处理一些公共的数据，这个时候采用上面这一些方法可能不利于项目的维护。这个时候可以使用 vuex ，vuex 的思想就是将这一些公共的数据抽离出来，将它作为一个全局的变量来管理，然后其他组件就可以对这个公共数据进行读写操作，这样达到了解耦的目的。

4）v-if和v-show的区别（★★★★）

* **手段**：v-if是动态的向DOM树内添加或者删除DOM元素；v-show是通过设置DOM元素的display样式属性控制显隐；
* **编译过程**：v-if切换有一个局部编译/卸载的过程，切换过程中合适地销毁和重建内部的事件监听和子组件；v-show只是简单的基于css切换；
* **编译条件**：v-if是惰性的，如果初始条件为假，则什么也不做；只有在条件第一次变为真时才开始局部编译; v-show是在任何条件下，无论首次条件是否为真，都被编译，然后被缓存，而且DOM元素保留；
* **性能消耗**：v-if有更高的切换消耗；v-show有更高的初始渲染消耗；
* **使用场景**：v-if适合运营条件不大可能改变；v-show适合频繁切换。

6 计网（http、tcp/ip）

2） 三次握手，四次挥手（★★★★★）

**简单来说就是以下三步：**

* **第一次握手：** 客户端向服务端发送连接请求报文段。
* **第二次握手：** 服务端收到连接请求报文段后，如果同意连接，则会发送一个应答
* **第三次握手：** 当客户端收到连接同意的应答后，还要向服务端发送一个确认报文。连接建立成功。

**简单来说就是以下四步：**

* **第一次挥手：** 若客户端认为数据发送完成，则它需要向服务端发送连接释放请求。
* **第二次挥手**：服务端收到连接释放请求后，然后会发送一个确认，此时表明客户端不能再向服务器端发送数据，但服务端仍旧可以发送数据给客户端。
* **第三次挥手**：服务器向客户端发送连接释放请求服务端，如果此时还有没发完的数据会继续发送。
* **第四次挥手：** 客户端收到释放请求后，向服务端发送确认应答，持续一段时间之后，若该时间段内没有服务端的重发请求的话，就关闭连接。当服务端收到确认应答后，也关闭连接。

TCP 使用四次挥手的原因是因为 TCP 的连接是全双工的，所以需要双方分别释放到对方的连接，单独一方的连接释放，只代 表不能再向对方发送数据，连接处于的是半释放的状态。

最后一次挥手中，客户端会等待一段时间再关闭的原因，是为了防止发送给服务器的确认报文段丢失或者出错，从而导致服务器端不能正常关闭。

3） TCP、UDP的区别，TCP怎么做到的可靠传输，UDP怎么改进实现可靠传输（★★★★）

首先概括一下基本的区别:

**TCP是一个面向连接的、可靠的、基于字节流的传输层协议。**

而**UDP是一个面向无连接的传输层协议。**(就这么简单，其它TCP的特性也就没有了)。

具体来分析，和 UDP 相比， TCP 有三大核心特性:

**1）面向连接**。所谓的连接，指的是客户端和服务器的连接，在双方互相通信之前，TCP 需要三次握手建

立连接，而 UDP 没有相应建立连接的过程。

**2）可靠性**。TCP 花了非常多的功夫保证连接的可靠，这个可靠性体现在哪些方面呢？一个是有状态，另一个是可控制。

TCP 会精准记录哪些数据发送了，哪些数据被对方接收了，哪些没有被接收到，而且保证数据包按序到达，不允许半点差错。这是**有状态**。

当意识到丢包了或者网络环境不佳，TCP 会根据具体情况调整自己的行为，控制自己的发送速度或者重发。这是**可控制**。

相应的，UDP 就是无状态, 不可控的。

**3）面向字节流**。UDP 的数据传输是基于数据报的，这是因为仅仅只是继承了 IP 层的特性，而 TCP 为了维护状态，将一个个 IP 包变成了字节流。

4） HTTP状态码（★★★★）

1XX ：信息状态码

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

2XX ：成功状态码

200 ok 正常返回信息

201 created 请求成功并且服务器创建了新资源

202 accepted 服务器已经接收请求，但尚未处理

3XX ：重定向

301 move per 请求的网页已经永久重定向

302 found 临时重定向

303 see other 临时冲重定向，且总是使用get请求新的url

304 not modified 自从上次请求后，请求的网页未修改过

4XX ：客户端错误

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 请求的资源（网页等）不存在

5XX ：服务器错误

500 internal server error 最常见的服务器端的错误

503 service unacailable 服务器端暂时无法处理请求（可能是过载活维护）

状态码：状态码用于表示服务器对请求的处理结果

列举几种常见的：

> 200（没有问题）

> 302（要你去找别人）

> 304（要你去拿缓存）

> 307（要你去拿缓存）

> 403（有这个资源，但是没有访问权限）

> 404（服务器没有这个资源）

> 500（服务器这边有问题）

5） HTTP HTTPS

HTTP和HTTPS协议的主要区别如下：

* HTTPS协议需要CA证书，费用较高；而HTTP协议不需要；
* HTTP协议是超文本传输协议，信息是明文传输的，HTTPS则是具有安全性的SSL加密传输协议；
* 使用不同的连接方式，端口也不同，HTTP协议端口是80，HTTPS协议端口是443；
* HTTP协议连接很简单，是无状态的；HTTPS协议是有SSL和HTTP协议构建的可进行加密传输、身份认证的网络协议，比HTTP更加安全。

6） get、post的特点和区别（★★★★）

Post 和 Get 是 HTTP 请求的两种方法，其区别如下：

* **应用场景：** GET 请求是一个幂等的请求，一般 Get 请求用于对服务器资源不会产生影响的场景，比如说请求一个网页的资源。而 Post 不是一个幂等的请求，一般用于对服务器资源会产生影响的情景，比如注册用户这一类的操作。
* **是否缓存：** 因为两者应用场景不同，浏览器一般会对 Get 请求缓存，但很少对 Post 请求缓存。
* **发送的报文格式：** Get 请求的报文中实体部分为空，Post 请求的报文中实体部分一般为向服务器发送的数据。
* **安全性：** Get 请求可以将请求的参数放入 url 中向服务器发送，这样的做法相对于 Post 请求来说是不太安全的，因为请求的 url 会被保留在历史记录中。
* **请求长度：** 浏览器由于对 url 长度的限制，所以会影响 get 请求发送数据时的长度。这个限制是浏览器规定的，并不是 RFC 规定的。
* **参数类型：** post 的参数传递支持更多的数据类型。

7） 同源策略、跨域处理（★★★★★）

跨域问题其实就是浏览器的同源策略造成的。

同源策略限制了从同一个源加载的文档或脚本如何与另一个源的资源进行交互。这是浏览器的一个用于隔离潜在恶意文件的重要的安全机制。同源指的是：**协议**、**端口号**、**域名**必须一致。

解决跨域问题：

**（1）CORS**

CORS需要浏览器和服务器同时支持，整个CORS过程都是浏览器完成的，无需用户参与。因此实现**CORS的关键就是服务器，只要服务器实现了CORS请求**，就可以跨源通信了。

**（2）JSONP**

**jsonp**的原理就是利用标签没有跨域限制，通过标签src属性，发送带有callback参数的GET请求，服务端将接口返回数据拼凑到callback函数中，返回给浏览器，浏览器解析执行，从而前端拿到callback函数返回的数据。

**（4）nginx代理跨域**

nginx代理跨域，实质和CORS跨域原理一样，通过配置文件设置请求响应头Access-Control-Allow-Origin…等字段。

7 算法

笔试那个题看一下

9 综合类问题（反问）

##1为什么学前端

（1）前端东西做出来能直接看到，有趣，有成就感

（2）能够更加直观的认识到用户体验的重要性，对公司理念、产品等的宣传具有不可替代的重要性。

##2 Java、js、python的区别

动态语言和静态语言

一、动态类型语言

动态类型语言，是指在代码运行阶段对数据类型进行确认。用动态类型语言编程时，不用给变量指定数据类型，该语言会在你第一次赋值给变量时，在内部记录数据类型。

动态类型语言的优点是不需要写多种数据类型的代码，代码相对简洁一些，方便代码阅读。缺点是不方便调试，代码命名也容易混淆；

  代表：JavaScript、Python、Perfl

二、静态类型语言

静态类型语言，是指在代码编译阶段对数据类型进行确认。

静态类型语言的优点是方便调试，代码相对规范。缺点是需要写很多数据类型相关的代码，代码不够简洁。

  代表：Java、C/C++

一、编译型

  编译型语言是将代码一次性全部编译成二进制码,然后运行。

代表语言：C/C++/go

二、解释型

  解释型语言的源代码不是直接翻译成机器指令，而是先翻译成中间代码，再由解释器对中间代码进行解释运行。（编译 一行，运行一行）

  代表语言：Java、Python、Perl、JavaScript、VBScript

编译型和解释型的定义是对立存在的，但也可以在一个语言中同时存在。比如 java 语言同时兼有编译型和解释型特点。整个流程如下：

  将源代码（.java 文件）编译生成字节码（.class 文件），再通过 JVM（java 虚拟机）运行生成机器指令，由机器运行机器码。注意，此处生成机器语言前的操作是解释型，每次运行都要重新解释。因此，此处表明 java 是解释型。

  但是，部分 JVM（java 虚拟机）有一种 JIT（Just in time）机制，能够将部分已经解释翻译的常用机器指令保存。下次不需要解释，直接运行即可。此时 java 是编译型。

Java：一种面向对象的编程语言，广泛用于编写桌面应用程序，Web应用程序，分布式系统等；

JavaScript：一种动态编程语言，主要用于增加网站上的交互性；

Python：一种面向对象的解释型动态类型脚本语言，应用于人工智能，自动化测试，系统[运维](https://cloud.tencent.com/solution/operation?from=10680)，爬虫等。

**03 变量的声明**

在**Java**语言中，所有的变量在使用前必须声明，声明的基本格式如下：

格式：类型 变量名 = 变量值如：int a,b,c; // 可以使用逗号隔开来声明多个同类型变量

复制

在**JavaScript**中，声明变量用关键词 **var**

var name; // 声明后变量名是空的，赋值需用等号格式为：类型 变量名 = 值；但这里不管是什么类型都用var

复制

在**Python**中，变量不需要声明。每个变量在使用前都必须赋值，变量赋值以后该变量才会被创建

name ='xiaoming' # 不需要指定类型格式为变量名 = 值

复制

此外变量的命名规则基本类似，变量要有意义，大小写敏感，遵循驼峰命名法，具体不展开详写。

**代码注释**

Java 支持三种注释方式

单行注释用 //

多行注释用 /\* \*/

说明注释 以 /\*\* 开始，以 \*/结束

​JavaScript 同Java​

在Python中

单行注释用 #

多行注释 '''注释代码''' 或者用双引号

##3未来规划路线

作为一个前端开发工程师，近期的职业规划是把咱们公司的项目做好，同时让我的技术得以提升，夯实我的技术，多学一些组件插件，多学一款到两款框架。

中长期的规划将来再研究一门后台语言java python，争取研究明白为我们公司更好的去服务，另外我也相信咱们公司有合理的晋升机制和制度，如果将来公司给我这个机会，我也会当仁不让，这就是我近期和长期的职业规划。

##4反问

公司的主要业务是什么

工作时间是怎么样的