## 网页布局有哪几种，有什么区别

静态、自适应、流式、响应式四种网页布局  
静态布局：意思就是不管浏览器尺寸具体是多少，网页布局就按照当时写代码的布局来布置；  
自适应布局：就是说你看到的页面，里面元素的位置会变化而大小不会变化；  
流式布局：你看到的页面，元素的大小会变化而位置不会变化——这就导致如果屏幕太大或者太小都会导致元素无法正常显示。  
自适应布局：每个屏幕分辨率下面会有一个布局样式，同时位置会变而且大小也会变。

-------------

流式布局（Flow Layout）：流式布局是最常见的布局方式，元素按照其在 HTML 中出现的顺序依次排列，元素的位置由文档流决定。元素会根据上一个元素的位置进行排列，如果空间不够，元素会自动换行。

弹性布局（Flexbox Layout）：弹性布局是一种新的布局方式，通过 flex 容器和 flex 项目来实现灵活的布局。弹性布局可以沿着一条轴（横轴或纵轴）对元素进行排列，可以实现灵活的对齐、空间分配和排序。

栅格布局（Grid Layout）：栅格布局是另一种新的布局方式，通过定义网格容器和网格项目来实现页面的布局。栅格布局可以实现复杂的多列布局，可以对元素进行精确的定位和对齐。

响应式布局（Responsive Layout）：响应式布局是一种可以适应不同设备和屏幕尺寸的布局方式，通过媒体查询、弹性布局和栅格布局等技术来实现。在不同的设备上可以呈现不同的布局和样式。

## 讲一下你对盒模型的理解？

盒模型分为：内容（content）、填充（padding）、边界（margin）、边框（border）四个部分。

盒模型有两种：IE盒模型和标准盒模型。两者的区别是标准盒模型的宽高不包含border和padding，而IE盒模型的宽高包含border和padding。

使用box-sizing属性可以切换和模型，默认值为content-box，即标准盒模型，border-box则是IE盒模型。

## css 选择器优先级？

!important > 行内样式（比重1000）> ID 选择器（比重100） > 类选择器（比重10） > 标签（比重1） > 通配符 > 继承 > 浏览器默认属性

## 说一说样式优先级的规则是什么？

得分点： !important、行内样式、嵌入样式、外链样式、id选择器、类选择器、标签选择器、复合选择器、通配符、继承样式

标准回答

CSS样式的优先级应该分成五大类

第一类!important，无论引入方式是什么，选择器是什么，它的优先级都是最高的。

第二类引入方式，行内样式的优先级要高于嵌入和外链，嵌入和外链如果使用的选择器相同就看他们在页面中插入的顺序，在后面插入的会覆盖前面的。

第三类选择器，选择器优先级：id选择器>（类选择器 | 伪类选择器 | 属性选择器 ）> （后代选择器 | 伪元素选择器 ）> （子选择器 | 相邻选择器） > 通配符选择器 。

第四类继承样式，是所有样式中优先级比较低的。

第五类浏览器默认样式优先级最低。

## 垂直居中几种方式？

单行文本: line-height = height

图片: vertical-align: middle;

absolute 定位: top: 50%;left: 50%;transform: translate(-50%, -50%);

flex: display:flex;

margin:auto

表格布局

## 简明说一下 CSS link 与 @import 的区别和用法？

link 是 XHTML 标签，除了加载CSS外，还可以定义 RSS 等其他事务；@import 属于 CSS 范畴，只能加载 CSS。

link 引用 CSS 时，在页面载入时同时加载；@import 需要页面网页完全载入以后加载。

link 是 XHTML 标签，无兼容问题；@import 是在 CSS2.1 提出的，低版本的浏览器不支持。

link 支持使用 Javascript 控制 DOM 去改变样式；而@import不支持。

## rgba和opacity的透明效果有什么不同？

opacity 会继承父元素的 opacity 属性，而 RGBA 设置的元素的后代元素不会继承不透明属性。

## display:none和visibility:hidden的区别？

display:none 隐藏对应的元素，在文档布局中不再给它分配空间，它各边的元素会合拢，就当他从来不存在。  
visibility:hidden 隐藏对应的元素，但是在文档布局中仍保留原来的空间。

## 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

## position的值， relative和absolute分别是相对于谁进行定位的？

relative:相对定位，相对于自己本身在正常文档流中的位置进行定位。

absolute:生成绝对定位，相对于最近一级定位不为static的父元素进行定位。

fixed: （老版本IE不支持）生成绝对定位，相对于浏览器窗口或者frame进行定位。

static:默认值，没有定位，元素出现在正常的文档流中。

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。

## 画一条0.5px的直线？

考查的是css3的transform

height: 1px;

transform: scale(0.5);

## calc, support, media各自的含义及用法？

1. @support 主要是用于检测浏览器是否支持CSS的某个属性，其实就是条件判断，如果支持某个属性，你可以写一套样式，如果不支持某个属性，你也可以提供另外一套样式作为替补。
2. calc() 函数用于动态计算长度值。 calc()函数支持 “+”, “-”, “\*”, “/” 运算；
3. @media 查询，你可以针对不同的媒体类型定义不同的样式。

## 1rem、1em、1vh、1px各自代表的含义？

rem

rem是全部的长度都相对于根元素元素。通常做法是给html元素设置一个字体大小，然后其他元素的长度单位就为rem。

em

子元素字体大小的em是相对于父元素字体大小

元素的width/height/padding/margin用em的话是相对于该元素的font-size

vw/vh

全称是 Viewport Width 和 Viewport Height，视窗的宽度和高度，相当于 屏幕宽度和高度的 1%，不过，处理宽度的时候%单位更合适，处理高度的 话 vh 单位更好。

px

px像素（Pixel）。相对长度单位。像素px是相对于显示器屏幕分辨率而言的。

一般电脑的分辨率有{19201024}等不同的分辨率

19201024 前者是屏幕宽度总共有1920个像素,后者则是高度为1024个像素

## 画一个三角形？

.a {

width: 0;

height: 0;

border-width: 100px;

border-style: solid;

border-color: transparent #0099CC transparent transparent;

transform: rotate(90deg); /\*顺时针旋转90°\*/

}

<div class="a"></div>

## BFC 是什么？

BFC 即 Block Formatting Contexts (块级格式化上下文)，它属于普通流，即：元素按照其在 HTML 中的先后位置至上而下布局，在这个过程中，行内元素水平排列，直到当行被占满然后换行，块级元素则会被渲染为完整的一个新行，除非另外指定，否则所有元素默认都是普通流定位，也可以说，普通流中元素的位置由该元素在 HTML 文档中的位置决定。

可以把 BFC 理解为一个封闭的大箱子，箱子内部的元素无论如何翻江倒海，都不会影响到外部。

只要元素满足下面任一条件即可触发 BFC 特性

* body 根元素
* 浮动元素：float 除 none 以外的值
* 绝对定位元素：position (absolute、fixed)
* display 为 inline-block、table-cells、flex
* overflow 除了 visible 以外的值 (hidden、auto、scroll)

## 常见兼容性问题？

* 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。
* 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.

## 块级元素、行内元素分别有哪些？

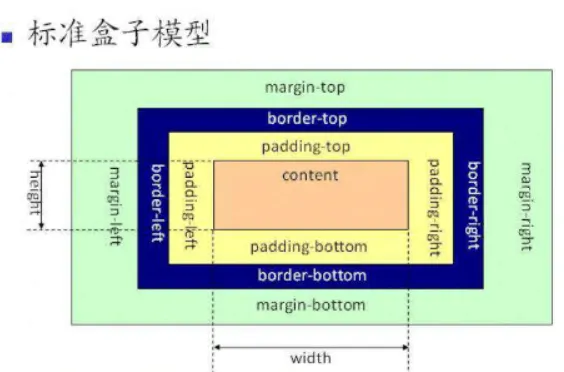
块级元素：div , p , form, ul, li , ol, dl, form, address, fieldset, hr, menu, table

行内元素：span, strong, em, br, img , input, label, select, textarea, cite

## 盒模型的差异？

css盒模型本质是一个盒子，它由边距、边框、填充和实际内容组成。盒模型能够让我们在其他元素和周边元素边框之间的空间放置元素。

标准盒与怪异盒的区别在于他们的总宽度的计算公式不一样。标准模式下总宽度=width+margin（左右）+padding（左右）border（左右）；怪异模式下总宽度=width+margin（左右）（就是说width已经包含了padding和border值）。标准模式下如果定义的DOCTYPE缺失，则在ie6、ie7、ie8下汇触发怪异模式。当设置为box-sizing:content-box时，将采用标准模式解析计算，也是默认模式；当设置为box-sizing:border-box时，将采用怪异模式解析计算。



## 说一说你是怎么理解BFC的？

得分点： 块级格式化上下文、独立的渲染区域、不会影响边界以外的元素、形成BFC条件、float、position、overflow、display

标准回答： BFC(Block Formatting Context)块级格式化上下文，是Web页面一块独立的渲染区域，内部元素的渲染不会影响边界以外的元素。

BFC布局规则：

内部盒子会在垂直方向，一个接一个地放置。

Box垂直方向的距离由margin决定。属于同一个BFC的两个相邻Box的margin会发生重叠。

每个盒子（块盒与行盒）的margin box的左边，与包含块border box的左边相接触(对于从左往右的格式化，否则相反)。即使存在浮动也是如此。

BFC的区域不会与float box重叠。

BFC就是页面上的一个隔离的独立容器，容器里面的子元素不会影响到外面的元素。反之也如此。

计算BFC的高度时，浮动元素也参与计算。

BFC形成的条件:

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、table-cell、table-caption、flex 等；

overflow值为：hidden、auto、scroll；

BFC解决能的问题：

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：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。

## 清除浮动有哪些方案？

清除浮动主要是为了解决父元素因为子元素浮动引起高度塌陷的问题。

本质上清除浮动的方式有两种，一种是利用clear属性，一种是利用BFC。

clear属性清除浮动又包括空标签法和伪元素法两种，代码如下：

使用空标签清除浮动 在浮动元素的末尾添加一个空标签，并设置 clear: both 属性，即可清除浮动。例如：

<div class="float-parent">

<div class="float-left"></div>

<div class="float-right"></div>

<div class="clearfix"></div>

</div>

.clearfix {

clear: both;

}

使用 :after 伪元素清除浮动

<div class="float-parent">

<div class="float-left"></div>

<div class="float-right"></div>

<div class="clearfix"></div>

</div>

.float-parent:after {

content: "";

display: table;

clear: both;

}

BFC清除浮动的方式有多种，我这里就以overflow为例：

<div class="float-parent" style="overflow: hidden;">

<div class="float-left"></div>

<div class="float-right"></div>

</div>

## 伪类和伪元素有什么区别？

伪类表示被选择元素的某种状态或者选择是，常见的伪类有三种，状态类、结构类和表单类。例如:hover就是状态类伪类，:first-child就是结构类伪类，:cheked就是表单类伪类。

a:hover { color: red; } /\* 鼠标悬停在链接上时文字变成红色 \*/

input:checked + label { background-color: green; } /\* 选中表单元素时相邻的标签背景变成绿色 \*/

伪元素用于创建一些不在文档树中的元素，并为其添加样式。伪元素常见的用途有给元素添加小部件、小图标以及清除浮动。 例如：

.arrow::before {

content: "";

display: block;

width: 0;

height: 0;

border-top: 10px solid transparent;

border-bottom: 10px solid transparent;

border-right: 10px solid red;

}

另外，虽然没有强制规定，但是伪类通常是单冒号，伪元素通常是用双冒号。

## 怎么做移动端适配？

常用的移动端适配方案有以下几种：

**rem方案**：淘宝的移动端适配方案，使用相对单位rem结合JS动态计算rem值来实现移动端适配，将页面在不同尺寸的屏幕小按照宽度等比例缩放。

**vw方案**：和rem方案类似，只是单位换成了vwborder方案：rem和vw方案都是等比例缩放，但是对于一些对UI要求特别高的大厂项目，缩放的显示效果并不是最佳，这时候也可以和UI配合采取px绝对像素单位。

**px方案**：rem和vw方案都是等比例缩放，但是对于一些对UI要求特别高的大厂项目，缩放的显示效果并不是最佳，这时候也可以和UI配合采取px绝对像素单位。

**媒体查询**：对于一些具体的场景，可以根据不同设备的像素区间来针对性地编写样式，这时候就使用媒体查询。

**百分比布局**：将元素的宽度和高度设置为百分比，使得页面可以根据不同的屏幕尺寸进行等比例缩放，这种方案和rem、vw原理类似，但是计算比较困难，而且百分比相对的元素不固定，容易使问题变得复杂。

**响应式布局**：对于一些定制化程度要求不高，但是需要PC和移动两端共用一套代码的场景，可以使用一些响应式布局的样式库，比如Bootstrap和Tailwind。

## 移动端1px边框问题怎么解决？

由于不同的设备屏幕像素密度的不同，一些边框、线条等细节元素的显示可能会出现“1px问题”，即在某些设备上，本应该显示为1像素的边框或者线条，实际上却被放大成了2像素或者更多像素，导致显示效果不佳。

常见的解决方案：

使用border-image：使用border-image可以将图片作为边框来显示，避免了使用CSS边框样式时的1px问题。

使用box-shadow：使用box-shadow代替边框，然后将边框设为透明，可以避免1px问题。

.border {

box-shadow: 0 0 0 1px #ccc;

}

3. border + transform：使用transform将边框缩小一半。

.border {

border: 1px solid #ccc;

transform: scaleY(0.5);

}

4. 伪元素 + transform：与3类似，不同的是用伪元素来实现。

.border:before{

content: "";

display: block;

position: absolute;

left: 0;

top: 0;

bottom: 0;

right: 0;

border: 1px solid #ccc;

transform-origin: 0 0;

transform: scaleY(0.5);

}

在父元素上添加一个伪元素，然后给伪元素设置一个边框，并将其缩小为0.5倍。这样就可以实现1px的边框效果了。

1. 使用viewport单位：使用viewport相关的单位（如vw、vh、vmin和vmax）来设置边框或者线条的大小，可以让元素的大小自适应不同的设备像素密度。

## CSS有哪些主流布局方式？

CSS主要有五大主流布局：

**flex布局**，目前应用最广的布局方式，虽然不兼容IE低版本，但是就连微软自己都已经放弃IE11了。

**浮动布局**，需要清除浮动，副作用较多，现在使用的已经不多了，除了在部分PC端需要兼容IE低版本的项目中。

**定位布局**，主要用来实现定位效果，依然使用频繁。

**栅格布局**，面向未来的布局，只是现在兼容性还不是很好，且学习成本比较高。

**table布局**，比浮动布局还要老的布局，现在已经很少使用，不过兼容性特别好。

可以根据自己的日常开发经验以及这五种布局方式的优缺点和适用场景谈一谈对它们的理解。

## 对CSS动画有了解吗，CSS实现动画有哪几种方式？

CSS动画主要包括transition动画、transform动画和animation动画。

1. **transition动画**：主要用来实现过渡效果，包括颜色、透明度、大小、位置，从一个状态向另一个状态过渡。

.element {

transition: property duration timing-function delay;

}

1. **transform动画**：用于对元素进行变形和转换，如旋转、缩放、平移和倾斜等。transform可以配合transition一起使用，例如：

.element {

transform: rotate(45deg);

transition: transform duration timing-function delay;

}

1. **animation动画**：使用@keyframes规则定义一组关键帧，描述元素在动画过程中的样式变化。然后通过animation属性将关键帧应用到元素上：

@keyframes animationName {

0% { /\* 初始样式 \*/ }

50% { /\* 中间样式 \*/ }

100% { /\* 结束样式 \*/ }

}

.element {

animation: animationName duration timing-function delay iteration-count direction;

}

在日常开发中大多数时候制作简单动画使用一些第三方的CSS动画库就足够了，比如Animate.css、Hover.css，但是复杂动画不仅要考虑CSS，还要考虑JS。在Vue和React这样的框架中，都提供了Transition组件来让我们实现动画。

对于复杂动画，除了实现效果之外还要考虑到动画的性能优化，需要使用Chrome工具去调优。

## 什么是CSS工程化？

这是一个偏主观性的题，可以根据自己的项目经历从以下几个方面来回答：

预编译器：less、sass、stylus等。

CSS模块化方案：常见的像Vue里的scoped方案以及React里的CSS-in-JS方案。

CSS自动化工具：比如PostCSS，就提供很多有用的插件，让我们可以做一些CSS的自动化工作，比如自动添加前缀、自动格式化代码、移动端适配。

代码检测工具：如CSSLint、Stylelint等，可以自动化检测CSS代码的错误和潜在问题，提高代码质量和稳定性。

样式指南：主要是指CSS的规范，包括设计原则、排版规则、颜色使用、图标规范、命名规范、代码格式化等各方面的规范。

CSS工程化通常包括以下几个方面：

1模块化管理：将CSS样式表分割成多个模块，每个模块负责管理特定的功能或组件的样式，从而降低样式之间的耦合性，便于维护。

2组件化开发：将页面中的各个组件的样式单独抽离出来，形成独立的组件库，通过组件化的方式来管理和使用CSS样式，提高代码的复用性。

3预处理器：使用CSS预处理器（如Sass、Less、Stylus等）来编写CSS，以提高CSS的可维护性和可扩展性，使用变量、嵌套、混合、函数等特性来简化CSS的编写。

4自动化构建：通过构建工具（如Webpack、Gulp、Grunt等）来自动化处理CSS文件，包括压缩、合并、添加浏览器前缀、图片压缩等操作，提高开发效率。

5样式规范：建立统一的样式规范，包括命名规范、代码风格、注释规范等，以提高团队协作和代码的可读性。

6浏览器兼容性处理：通过自动化工具来处理浏览器兼容性，自动添加浏览器前缀，以确保样式在不同浏览器中的兼容性。

## CSS有哪些优化手段？

CSS优化，可以从以下方面来考虑：

压缩CSS代码：通过删除空格、注释、冗余代码等方法，可以减小CSS文件的大小，提高加载速度，不过这些通常脚手架已经帮我们做好了。

合并CSS文件：将多个CSS文件合并成一个文件，减少HTTP请求次数，提高加载速度。在项目中可以用Webpack插件来实现。

首屏样式内联，可以减少加载CSS文件的时间，提升页面加载速度。

使用CSS3硬件加速，可以减少回流重绘，常见的触发硬件加速的CSS属性有：transform、opacity、filters、Will-change。

减少使用通配符和后代选择器：过多的通配符和后代选择器会降低选择器的匹配速度，应尽量减少使用。

避免使用过多的float：过多的浮动会导致页面回流和重绘，影响性能。

缓存：可以在服务器端设置协商缓存或者强缓存来缓存CSS请求。

日常开发中，除了开发CSS动画之外，绝大多数情况下，CSS的性能不会出现太明显的问题。所以在绝大多数情况下，只要是按照基本的规则和规范写CSS代码，问题就不大。

## 为什么要做样式初始化？

因为浏览器有默认样式，而且不同浏览器默认样式不一样，为了让样式显示一致，要去掉这些默认样式。

常见的方案有：

reset.css：将所有元素的样式都设置为相同的初始值，以消除不同浏览器之间的差异。这种方式需要注意的是，一些元素的样式可能与开发者所期望的有所不同，因此需要进行特殊处理。

Normalize.css：只重置一部分元素的样式，而不是重置所有元素。这种方式可以避免一些样式上的问题，同时还可以保留一些元素的默认样式，提高代码的可维护性和可读性。

## 讲一下什么是重绘和回流？

重绘（Repaint）：重绘是指当元素的样式改变，但不影响其在文档流中的位置时，浏览器会将新样式直接绘制到屏幕上，这个过程称为重绘。

回流（Reflow）：当元素的尺寸、位置或其他属性发生变化时，浏览器需要重新计算页面中所有元素的位置和大小，并且将它们绘制到屏幕上，这个过程称为回流，又叫做重排。

回流的代价比重绘高得多，因为回流会导致浏览器重新计算页面布局，这个过程中涉及到大量的计算和DOM操作，所以需要尽量减少回流的次数。以下是一些可能导致回流的操作：

页面初始渲染时。

调整窗口大小时。

改变字体大小。

改变元素的大小或位置。

修改元素的内容，例如输入框中的文字。

可以通过以下方式来减少回流的次数：

使用transform替代top或left。

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）。

不要把DOM节点的属性值放在一个循环里当成循环里的变量。

尽量不要使用table布局，可能很小的一个小改动就会造成整个table的重新布局。

尽量减少DOM的嵌套层级，减少不必要的父子元素节点。

## css样式污染问题

在react中推荐使用：CSS Modules （推荐原因：React脚手架已集成，可直接使用）。在vue中默认使用的是scoped css

1、改样式文件名。从 xx.scss -> xx.module.scss （这是React脚手架中的约定，与普通 CSS 作区分）

1. 组件中导入该样式文件（注意语法）

import styles from './index.module.scss'

3、通过 styles 对象访问对象中的样式名来设置样式

import styles from './App.module.scss';

import Login from './Login'

function App() {

return (

<div className="App">

<div className={styles.box}>

APP

</div>

<Login></Login>

</div>

);

}

export default App;

实现的原理是CSS Modules 通过自动给 CSS 类名补足类名，保证类名的唯一性，从而避免样式冲突的问题 。

App\_box\_CgquZ {

height: 100px;

width: 100px;

background-color: yellow;

}

这个时候看类名已经默认被CSS Modules 补足，保证了类名的唯一性 掌握CssModules-维持类名

在没有做任何处理的情况下， 每一个类名都会被修改。

如果希望某个类名不被修改，可以使用: global。如下：

.root {

.info {

color: red;

}

:global(.info) {

font-size: 14px;

}

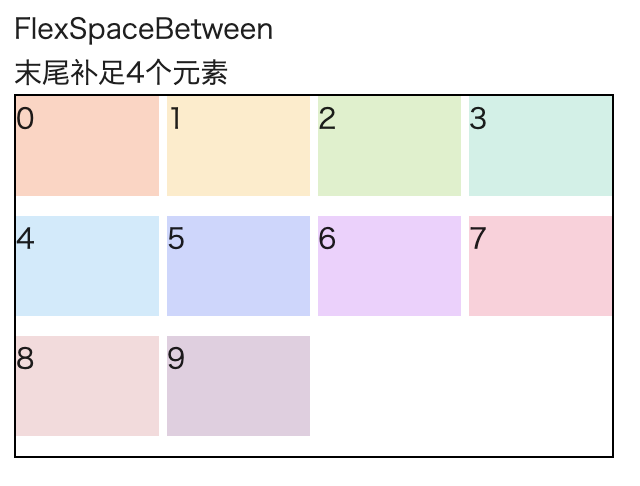
}

发现加了: global的类名会维持原状， 不会被修改

## flex布局

### 问题: 假如一行显示4个元素，最后一行在不够4个元素的时候，按照两端对齐显示，效果显得不好，我们希望假如最后一行不足4个元素的时候，默认从左向右排列。

#### 方法一 末尾补足一行数量元素



// FlexSpaceBetween.tsx

function FlexSpaceBetween() {

const arr = Array.from(Array(10).keys());

**const arr2 = Array.from(Array(4).keys());** // 补足元素 有宽度无高度

return (

<>

<h1>FlexSpaceBetween</h1>

<div>末尾补足4个元素</div>

<div className='flex-space-between1'>

{arr.map((item, idx) => (

<div key={item} className='flex-space-between1-item' style={{ background: BOOK\_BG\_COLOR.colors[idx] }}>

{item}

</div>

))}

{arr2.map(item => (

<div key={item} className='flex-space-between1-item2'></div>

))}

</div>

</>

);

}

// scss

.flex-space-between1 {

width: 300px;

display: flex;

justify-content: space-between;

flex-wrap: wrap;

border: 1px solid #000;

margin-bottom: 20px;

&-item {

width: 24%;

height: 50px;

margin-bottom: 10px;

}

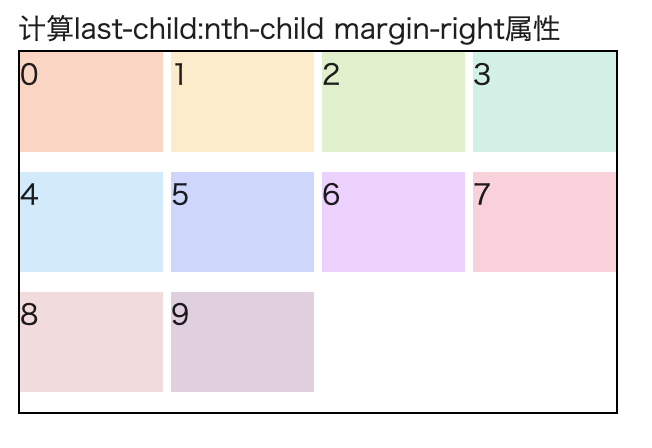
**&-item2 {**

**width: 24%;**

**}**

}

#### 方法二 计算last-child:nth-child margin-right属性



// FlexSpaceBetween.tsx

function FlexSpaceBetween() {

const arr = Array.from(Array(10).keys());

return (

<>

<h1>FlexSpaceBetween</h1>

<div>计算last-child:nth-child margin-right属性</div>

<div className='flex-space-between2'>

{arr.map((item, idx) => (

<div key={item} className='flex-space-between2-item' style={{ background: BOOK\_BG\_COLOR.colors[idx] }}>

{item}

</div>

))}

</div>

</>

);

}

// scss

.flex-space-between2 {

width: 300px;

display: flex;

justify-content: space-between;

flex-wrap: wrap;

border: 1px solid #000;

margin-bottom: 20px;

&-item {

width: 24%;

height: 50px;

margin-bottom: 10px;

**&:last-child:nth-child(4n - 2) {**

**margin-right: calc(48% + 8% / 3);**

**}**

**&:last-child:nth-child(4n - 1) {**

**margin-right: calc(24% + 4% / 3);**

**}**

}

}

## 仅使用 CSS 怎么实现宽高自适应的正方形？

### 正常设置宽高 ~ 100px \* 100px

.square-1 {

width: 100px;

height: 100px;

}

### 使用rem、vw等相对单位

2.1 rem

.square-2-1 {

width: 10rem;

height: 10rem;

}

屏幕宽度大于1200px时，rem的盒子的大小为 160px \* 160px；

屏幕宽度小于1200px时，rem的盒子的大小为 120px \* 120px；

优点

在适配了rem的项目中，可以直接使用rem设置盒子宽高，即自适应正方形

缺点

在未设置rem的项目中，单独为了实现自适应正方形有点大材小用

2.2 vw

.square-2-2 {

width: 10vw;

height: 10vw;

border: 1px solid #000;

}

优点

vw为内置视口单位，可直接使用。

缺点

在实际的业务场景中，将设计稿的尺寸转化为vw相对麻烦

### 百分比+padding

// html

<div className='square-3'>

<div className='content'>正常内容</div>

</div>

// css

.square-3 {

position: relative;

width: 10%;

padding-top: 10%; // 或者 padding-bottom

border: 1px solid #000;

.content {

position: absolute;

top: 0;

left: 0;

width: 100%;

height: 100%;

background: pink;

}

}

优点

无需其他配置，设置灵活，扩展性强。

缺点

需要额外嵌套一层内容盒

### aspect-ratio 宽高比

.square-4 {

width: 20%;

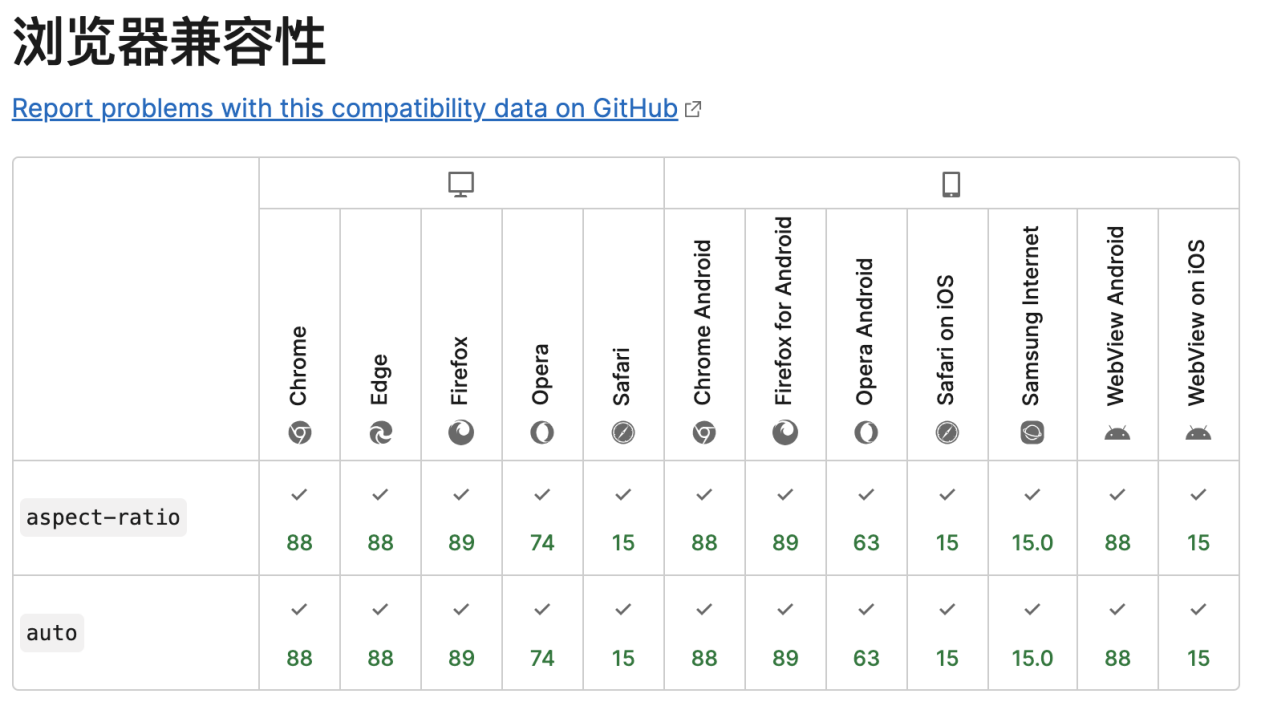
aspect-ratio: 1 / 1;

// aspect-ratio: 1; // 简写

border: 1px solid #000;

}

[MDN文档地址](https://developer.mozilla.org/zh-CN/docs/Web/CSS/aspect-ratio)



优点

属性通俗易懂，无需其他配置，无需嵌套内容盒。

缺点

唯一缺点就是兼容性了，2023年了该支持的基本都支持上了

## z-index属性在什么情况下会失效？

通常 z-index 的使用是在有两个重叠的标签，在一定的情况下控制其中一个在另一个的上方或者下方出现。z-index值越大就越是在上层。z-index元素的position属性需要是relative，absolute或是fixed。

z-index属性在下列情况下会失效：

* 父元素position为relative时，子元素的z-index失效。解决：父元素position改为absolute或static；
* 元素没有设置position属性为非static属性。解决：设置该元素的position属性为relative，absolute或是fixed中的一种；
* 元素在设置z-index的同时还设置了float浮动。解决：float去除，改为display：inline-block；
* 在手机端 iOS 13 系统中，-webkit-overflow-scrolling:touch 也会使 z-index 失效，将 touch 换成 unset

## position: fixed 一定是相对于浏览器窗口进行定位吗？

不一定。

position:fixed;的元素会被移出正常文档流，并不为元素预留空间，而是通过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。fixed 属性会创建新的层叠上下文。

当元素祖先的 **transform, perspective 或 filter 属性非 none** 时，容器由视口改为该祖先。

## 为什么有时候⽤translate来改变位置⽽不是使用position进行定位？(硬件加速)

translate 是 transform 属性的⼀个值。

改变transform或opacity不会触发浏览器重新布局（reflow）或重绘（repaint），只会触发复合（compositions）。

⽽改变绝对定位会触发重新布局，进⽽触发重绘和复合。

transform使浏览器为元素创建⼀个 GPU 图层，但改变绝对定位会使⽤到 CPU。

因此translate()更⾼效，可以缩短平滑动画的绘制时间。

⽽translate改变位置时，元素依然会占据其原始空间，绝对定位就不会发⽣这种情况。

## 硬件加速的原理是什么？

面试中可能会经常会碰到怎么解决动画卡顿的问题，然后会引导到硬件加速。那么究竟什么是硬件加速，为什么它可以提高咱们的动画效率？我们今天就来一探究竟。

首先，我们先从 CPU 和 GPU 开始了解。

### CPU 和 GPU 的区别

CPU 即中央处理器，GPU 即图形处理器。

CPU是计算机的大脑，它提供了一套指令集，我们写的程序最终会通过 CPU 指令来控制的计算机的运行。它会对指令进行译码，然后通过逻辑电路执行该指令。整个执行的流程分为了多个阶段，叫做流水线。指令流水线包括取指令、译码、执行、取数、写回五步，这是一个指令周期。CPU会不断的执行指令周期来完成各种任务。

GPU，是Graphics ProcessingUnit的简写，是现代显卡中非常重要的一个部分，其地位与CPU在主板上的地位一致，主要负责的任务是加速图形处理速度。GPU是显卡的“大脑”，它决定了该显卡的档次和大部分性能，同时也是2D显示卡和3D显示卡的区别依据。2D显示芯片在处理3D图像和特效时主要依赖CPU的处理能力，称为“软加速”。3D显示芯片是将三维图像和特效处理功能集中在显示芯片内，也即所谓的“硬件加速”功能。

要解释两者的区别，要先明白两者的相同之处：两者都有总线和外界联系，有自己的缓存体系，以及数字和逻辑运算单元。

一句话，两者都为了完成计算任务而设计。

两者的区别在于存在于片内的缓存体系和数字逻辑运算单元的结构差异：

* CPU虽然有多核，但总数没有超过两位数，每个核都有足够大的缓存和足够多的数字和逻辑运算单元，并辅助有很多加速分支判断甚至更复杂的逻辑判断的硬件；
* GPU 的核数远超CPU，被称为众核（NVIDIA Fermi有512个核）。每个核拥有的缓存大小相对小，数字逻辑运算单元也少而简单（GPU初始时在浮点计算上一直弱于CPU）。
* 从结果上导致CPU擅长处理具有复杂计算步骤和复杂数据依赖的计算任务，如分布式计算，数据压缩，人工智能，物理模拟，以及其他很多很多计算任务等。

GPU由于历史原因，是为了视频游戏而产生的（至今其主要驱动力还是不断增长的视频游戏市场），在三维游戏中常常出现的一类操作是对海量数据进行相同的操作，如：对每一个顶点进行同样的坐标变换，对每一个顶点按照同样的光照模型计算颜色值。

GPU的众核架构非常适合把同样的指令流并行发送到众核上，采用不同的输入数据执行。在通用计算领域有广泛应用，包括：数值分析，海量数据处理（排序，Map-Reduce等），金融分析等等。

简而言之，当程序员为CPU编写程序时，他们倾向于利用复杂的逻辑结构优化算法从而减少计算任务的运行时间，即 Latency。当程序员为GPU编写程序时，则利用其处理海量数据的优势，通过提高总的数据吞吐量（Throughput）来掩盖 Lantency。

目前，CPU 和 GPU 的区别正在逐渐缩小，因为GPU也在处理不规则任务和线程间通信方面有了长足的进步。

### 每一帧的执行步骤

一般浏览器的刷新率为60HZ，即1秒钟刷新60次。

1000ms / 60hz = 16.6 ，也就是大概每过 16.6ms 浏览器就会渲染一帧画面。

浏览器对每一帧画面的渲染工作都要在 16ms 内完成，超出这个时间，页面的渲染就会出现卡顿现象，影响用户体验。

简单概括下，浏览器在每一帧里会依次执行以下这些动作：

* JavaScript：JavaScript 实现动画效果，DOM 元素操作等。
* Style（计算样式）：确定每个 DOM 元素应该应用什么 CSS 规则。
* Layout（布局）：计算每个 DOM 元素在最终屏幕上显示的大小和位置。由于 web 页面的元素布局是相对的，所以其中任意一个元素的位置发生变化，都会联动的引起其他元素发生变化，这个过程叫 reflow。
* Paint（绘制）：在多个层上绘制 DOM 元素的的文字、颜色、图像、边框和阴影等。
* Composite（渲染层合并）：按照合理的顺序合并图层然后显示到屏幕上。
* 减少或者避免 layout，paint 可以让页面减少卡顿，动画效果更加流畅。

### 完整的渲染流程

更具体一些，一个完整的渲染步骤大致可总结为如下：

* 渲染进程将HTML内容转换为能够读懂的DOM树结构。
* 渲染引擎将CSS样式表转化为浏览器可以理解的 styleSheets ，计算出DOM节点的样式。
* 创建布局树，并计算元素的布局信息。
* 对布局树进行分层，并生成分层树。
* 为每个图层生成绘制列表，并将其提交到合成线程。
* 合成线程将图层分成图块，并在光栅化线程池中将图块转换成位图。
* 合成线程发送绘制图块命令DrawQuad给浏览器进程。
* 浏览器进程根据DrawQuad消息生成页面，并显示到显示器上

### 普通图层和复合图层

上面的介绍中，提到了 composite 概念。

可以简单的这样理解，浏览器渲染的图层一般包含两大类：渲染图层（普通图层）以及复合图层

* 渲染图层：又称默认复合层，是页面普通的文档流。我们虽然可以通过绝对定位，相对定位，浮动定位脱离文档流，但它仍然属于默认复合层，共用同一个绘图上下文对象（GraphicsContext）。
* 复合图层，它会单独分配资源（当然也会脱离普通文档流，这样一来，不管这个复合图层中怎么变化，也不会影响默认复合层里的回流重绘）

某些特殊的渲染层会被提升为复合成层（Compositing Layers），复合图层拥有单独的 GraphicsLayer，而其他不是复合图层的渲染层，则和其第一个拥有 GraphicsLayer 父层共用一个。

每个 GraphicsLayer 都有一个 GraphicsContext，GraphicsContext 会负责输出该层的位图，位图是存储在共享内存中，作为纹理上传到 GPU 中，最后由 GPU 将多个位图进行合成，然后 draw 到屏幕上，此时，我们的页面也就展现到了屏幕上。

可以 Chrome源码调试 -> More Tools -> Rendering -> Layer borders中看到，黄色的就是复合图层信息。

### 硬件加速

硬件加速，直观上说就是依赖 GPU 实现图形绘制加速，软硬件加速的区别主要是图形的绘制究竟是 GPU 来处理还是 CPU，如果是 GPU，就认为是硬件加速绘制，反之，则为软件绘制。

一般一个元素开启硬件加速后会变成复合图层，可以独立于普通文档流中，改动后可以避免整个页面重绘，提升性能。

常用的硬件加速方法有：

* 最常用的方式：translate3d、translateZ
* opacity 属性/过渡动画（需要动画执行的过程中才会创建合成层，动画没有开始或结束后元素还会回到之前的状态）
* will-change属性（这个知识点比较冷僻），一般配合 opacity 与 translate 使用（而且经测试，除了上述可以引发硬件加速的属性外，其它属性并不会变成复合层），作用是提前告诉浏览器要变化，这样浏览器会开始做一些优化工作（这个最好用完后就释放）
* <video>、<iframe>、<canvas>、<webgl>等元素
* 其它，譬如以前的 flash 插件

当然，有的时候我们想强制触发硬件渲染，就可以通过上面的属性，比如

will-change: transform;

或者

transform:translate3d(0, 0, 0);

### 使用硬件加速的注意事项

使用硬件加速并不是十全十美的事情，比如：

* 内存。如果GPU加载了大量的纹理，那么很容易就会发生内容问题，这一点在移动端浏览器上尤为明显，所以，一定要牢记不要让页面的每个元素都使用硬件加速。
* 使用GPU渲染会影响字体的抗锯齿效果。这是因为GPU和CPU具有不同的渲染机制。即使最终硬件加速停止了，文本还是会在动画期间显示得很模糊。

所以不要大量使用复合图层，否则由于资源消耗过度，页面可能会变的更加卡顿。

同时，在使用硬件加速时，尽可能的使用z-index，防止浏览器默认给后续的元素创建复合层渲染。

具体的原理是这样的：

webkit CSS3中，如果一个元素添加了硬件加速，并且z-index层级比较低，那么在这个元素的后面其它元素（层级比这个元素高的，或者相同的，并且releative或absolute属性相同的），会默认变为复合层渲染，如果处理不当会极大的影响性能。

简单点理解，其实可以认为是一个隐式合成的概念：如果a是一个复合图层，而且b在a上面，那么b也会被隐式转为一个复合图层，这点需要特别注意。

## css硬件加速

CSS硬件加速是一种利用GPU（图形处理器）来加速浏览器中渲染页面的技术。它可以提高页面的流畅度和性能。

1.使用transform属性：使用transform属性可以触发GPU加速。

.element {

transform: translateZ(0);

}

2.使用will-change属性：will-change属性可以告诉浏览器哪些属性将要被改变，从而提前进行优化。

.element {

will-change: transform;

}

3.使用3D加速：使用3D加速可以让元素在GPU中进行渲染，从而提高性能。

.element {

transform: translate3d(0);

}

4.使用缓存技术：将经常使用的元素缓存起来，可以减少页面渲染的时间和消耗。

let element = document.getElementById('element');

let cachedElement = element.cloneNode(true);

## CSS动画和JS实现的动画分别有哪些优缺点？

### CSS动画

优点

浏览器可以对动画进行优化

代码相对简单,性能调优方向固定

对于帧速表现不好的低版本浏览器，CSS3可以做到自然降级，而JS则需要撰写额外代码

缺点

运行过程控制较弱,无法附加事件绑定回调函数

代码冗长，想用CSS实现稍微复杂一点动画,最后CSS代码都会变得非常笨重

### JS动画

优点

控制能力很强, 可以在动画播放过程中对动画进行控制：开始、暂停、回放、终止、取消都是可以做到的。

动画效果比css3动画丰富,有些动画效果，比如曲线运动,冲击闪烁,视差滚动效果，只有js动画才能完成

CSS3有兼容性问题，而JS大多时候没有兼容性问题

缺点

代码的复杂度高于CSS动画

JavaScript在浏览器的主线程中运行，而主线程中还有其它需要运行的JavaScript脚本、样式计算、布局、绘制任务等,对其干扰导致线程可能出现阻塞，从而造成丢帧的情况

## 前端实现动画有哪些方式？

* css3的transition 属性
* css3的animation 属性
* 原生JS动画
* 使用canvas绘制动画
* SVG动画
* Jquery的animate函数
* 用gif图片

## 为何CSS不支持父选择器？(为何CSS相邻兄弟选择器只支持后面的元素，而不支持前面的兄弟元素？)

浏览器解析HTML文档，是从前往后，由外及里的。所以，我们时常会看到页面先出现头部然后主体内容再出现的加载情况。

但是，如果CSS支持了父选择器，那就必须要页面所有子元素加载完毕才能渲染HTML文档，因为所谓“父选择器”，就是后代元素影响祖先元素，如果后代元素还没加载处理，如何影响祖先元素的样式？于是，网页渲染呈现速度就会大大减慢，浏览器会出现长时间的白板。加载多少HTML就可以渲染多少HTML，在网速不是很快的时候，就显得尤为的必要。比方说你现在看的这篇文章，只要文章内容加载出来就可以了，就算后面的广告脚本阻塞了后续HTML文档的加载，我们也是可以阅读和体验。但是，如果支持父选择器，则整个文档不能有阻塞，页面的可访问性则要大大降低。

有人可能会说，要不采取加载到哪里就渲染到哪里的策略？这样子问题更大，因为会出现加载到子元素的时候，父元素本来渲染的样式突然变成了另外一个样式的情况，体验非常不好。

“相邻选择器只能选择后面的元素”也是一样的道理，不可能说后面的HTML加载好了，还会影响前面HTML的样式。

所以，从这一点来讲，CSS支持“父选择器”或者“前兄弟选择器”的可能性要比其他炫酷的CSS特性要低，倒不是技术层面，而是CSS和HTML本身的渲染机制决定的。当然，以后的事情谁都说不准，说不定以后网速都是每秒几个G的，网页加载速度完全就忽略不计，说不定就会支持了。

## 脱离文档流有哪些方法？

### 一、什么是文档流？

将窗体自上而下分成一行一行，并在每行中按从左至右依次排放元素，称为文档流，也称为普通流。

这个应该不难理解，HTML中全部元素都是盒模型，盒模型占用一定的空间，依次排放在HTML中，形成了文档流。

### 二、什么是脱离文档流？

元素脱离文档流之后，将不再在文档流中占据空间，而是处于浮动状态（可以理解为漂浮在文档流的上方）。脱离文档流的元素的定位基于正常的文档流，当一个元素脱离文档流后，依然在文档流中的其他元素将忽略该元素并填补其原先的空间。

### 三、怎么脱离文档流？

float

使用float可以脱离文档流。

注意！！！：使用float脱离文档流时，其他盒子会无视这个元素，但其他盒子内的文本依然会为这个元素让出位置，环绕在该元素的周围。

absolute

absolute称为绝对定位，其实博主觉得应该称为相对定位，因为使用absolute脱离文档流后的元素，是相对于该元素的父类（及以上，如果直系父类元素不满足条件则继续向上查询）元素进行定位的，并且这个父类元素的position必须是非static定位的（static是默认定位方式）。

fixed

完全脱离文档流，相对于浏览器窗口进行定位。（相对于浏览器窗口就是相对于html）。

## css sprites是什么，怎么使用？

### 是什么

CSS Sprites是一种网页图片应用处理方式，就是把网页中一些背景图片整合到一张图片文件中，再利用CSS的“background-image”，“background- repeat”，“background-position”的组合进行背景定位。

### 优点

* 减少网页的http请求，提高性能，这也是CSS Sprites最大的优点，也是其被广泛传播和应用的主要原因；
* 减少图片的字节：多张图片合并成1张图片的字节小于多张图片的字节总和；
* 减少了命名困扰：只需对一张集合的图片命名，不需要对每一个小元素进行命名提高制作效率；
* 更换风格方便：只需要在一张或少张图片上修改图片的颜色或样式，整个网页的风格就可以改变，维护起来更加方便。

### 缺点

* 图片合成比较麻烦；
* 背景设置时，需要得到每一个背景单元的精确位置；
* 维护合成图片时，最好只是往下加图片，而不要更改已有图片。

## Atom CSS 是什么？

Atom CSS：**原子CSS**，意思是**一个类只干一件事**。

<div id="app" class="w-full h-full"></div>

.w-full{

width:100%;

}

.h-full{

height:100%;

}

### 优点

* 减少了css体积，提高了css复用
* 减少起名的复杂度

### 缺点

* 增加了记忆成本。将css拆分为原子之后，你势必要记住一些class才能书写，哪怕tailwindcss提供了完善的工具链，你写background，也要记住开头是bg。
* 增加了html结构的复杂性。当整个dom都是这样class名，势必会带来调试的麻烦，有的时候很难定位具体css问题
* 你仍需要起class名。对于大部分属性而言，你可以只用到center,auto，100%，这些值，但是有时候你仍需要设定不一样的参数值，例如left，top，这时候你还需要起一个class名

## CSS中的1像素问题是什么？有哪些解决方案？

### 1px 边框问题的由来

苹果 iPhone4 首次提出了 Retina Display（视网膜屏幕）的概念，在 iPhone4 使用的视网膜屏幕中，把 2x2 个像素当 1 个物理像素使用，即使用 2x2 个像素显示原来 1 个物理像素显示的内容，从而让 UI 显示更精致清晰，这 2x2 个像素叫做逻辑像素。

像这种像素比（像素比（即dpr）＝ 物理像素 / 逻辑像素）为 2 的视网膜屏幕也被称为二倍屏，目前市面上还有像素比更高的三倍屏、四倍屏。

而 CSS 中 1px 指的是物理像素，因此，设置为 1px 的边框在 dpr = 2 的视网膜屏幕中实际占用了 2 个逻辑像素的宽度，这就导致了界面边框变粗的视觉体验。

### 使用 transform 解决

通过设置元素的 box-sizing 为 border-box，然后构建伪元素，再使用 CSS3 的 transform 缩放，这是目前市面上最受推崇的解决方法。这种方法可以满足所有的场景，而且修改灵活，唯一的缺陷是，对于已使用伪元素的元素要多嵌套一个无用元素。具体实现如下:

.one-pixel-border {

position: relative;

box-sizing: border-box;

}

.one-pixel-border::before {

display: block;

content: "";

position: absolute;

top: 50%;

left: 50%;

width: 200%;

height: 200%;

border: 1px solid red;

transform: translate(-50%, -50%) scale(0.5, 0.5);

}

这样就可以得到 **0.5px 的边框**。

还可以结合媒体查询（@media）解决不同 dpr 值屏幕的边框问题，如下：

@media screen and (-webkit-min-device-pixel-ratio: 2), (min-resolution: 2dppx) {

...

}

@media screen and (-webkit-min-device-pixel-ratio: 3), (min-resolution: 3dppx) {

...

}

当然还有不少其他的解决方案：border-image、background-image、viewport + rem + js、box-shadow等，但都有各自的缺点，不进行推荐，此处也不做详细介绍。

## css加载会造成阻塞吗？

### 先说下结论：

* css加载不会阻塞DOM树的解析
* css加载会阻塞DOM树的渲染
* css加载会阻塞后面js语句的执行

为了避免让用户看到长时间的白屏时间，我们应该尽可能的**提高css加载速度**，比如可以使用以下几种方法:

* 使用CDN(因为CDN会根据你的网络状况，替你挑选最近的一个具有缓存内容的节点为你提供资源，因此可以减少加载时间)
* 对css进行压缩(可以用很多打包工具，比如webpack,gulp等，也可以通过开启gzip压缩)
* 合理的使用缓存(设置cache-control,expires,以及E-tag都是不错的，不过要注意一个问题，就是文件更新后，你要避免缓存而带来的影响。其中一个解决防范是在文件名字后面加一个版本号)
* 减少http请求数，将多个css文件合并，或者是干脆直接写成内联样式(内联样式的一个缺点就是不能缓存)

### 原理解析

浏览器渲染的流程如下：

* HTML解析文件，生成DOM Tree，解析CSS文件生成CSSOM Tree
* 将Dom Tree和CSSOM Tree结合，生成Render Tree(渲染树)
* 根据Render Tree渲染绘制，将像素渲染到屏幕上。

从流程我们可以看出来:

* DOM解析和CSS解析是两个并行的进程，所以这也解释了为什么CSS加载不会阻塞DOM的解析。
* 然而，由于Render Tree是依赖于DOM Tree和CSSOM Tree的，所以他必须等待到CSSOM Tree构建完成，也就是CSS资源加载完成(或者CSS资源加载失败)后，才能开始渲染。因此，CSS加载是会阻塞Dom的渲染的。
* 由于js可能会操作之前的Dom节点和css样式，因此浏览器会维持html中css和js的顺序。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。

## CSS 中有哪几种定位方式？

Static

这个是元素的默认定位方式，元素出现在正常的文档流中，会占用页面空间。

Relative

相对定位方式，相对于其父级元素（无论父级元素此时为何种定位方式）进行定位，准确地说是相对于其父级元素所剩余的未被占用的空间进行定位（在父元素由多个相对定位的子元素时可以看出），且会占用该元素在文档中初始的页面空间，即在使用top，bottom，left，right进行移动位置之后依旧不会改变其所占用空间的位置。可以使用z-index进行在z轴方向上的移动。

Absolute

绝对定位方式，脱离文档流，不会占用页面空间。以最近的不是static定位的父级元素作为参考进行定位，如果其所有的父级元素都是static定位，那么此元素最终则是以当前窗口作为参考进行定位。

可以使用top，bottom，left，right进行位置移动，亦可使用z-index在z轴上面进行移动。当元素为此定位时，如果该元素为内联元素，则会变为块级元素，即可以直接设置其宽和高的值；如果该元素为块级元素，则其宽度会由初始的100%变为auto。

注意：当元素设置为绝对定位时，在没有指定top，bottom，left，right的值时，他们的值并不是0，这几个值是有默认值的，默认值就是该元素设置为绝对定位前所处的正常文档流中的位置。

Fixed

绝对定位方式，直接以浏览器窗口作为参考进行定位。其它特性同absolute定位。

当父元素使用了transform的时候，会以父元素定位。

sticky

粘性定位，可以简单理解为relative和fixed布局的混合。

当粘性约束矩形在可视范围内为relative，反之，则为fixed粘性定位元素如果和它的父元素一样高，则垂直滚动的时候，粘性定位效果是不会出现的它的定位效果完全受限于父级元素们。

如果父元素的overflow属性设置了scroll，auto,overlay值，那么，粘性定位将会失效同一容器中多个粘贴定位元素独立偏移，因此可能重叠；位置上下靠在一起的不同容器中的粘贴定位元素则会鸠占鹊巢，挤开原来的元素，形成依次占位的效果。

## 如果需要手动写动画，你认为最小时间间隔是多久，为什么？

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。

## CSS优化、提高性能的方法有哪些？

* 避免过度约束
* 避免后代选择符
* 避免链式选择符
* 使用紧凑的语法
* 避免不必要的命名空间
* 避免不必要的重复
* 最好使用表示语义的名字。一个好的类名应该是描述他是什么而不是像什么
* 避免！important，可以选择其他选择器
* 尽可能的精简规则，你可以合并不同类里的重复规则

## 为什么会出现浮动？什么时候需要清除浮动？清除浮动的方式有哪些？

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：

* 父级div定义height
* 最后一个浮动元素后加空div标签 并添加样式clear:both。
* 包含浮动元素的父标签添加样式overflow为hidden或auto。
* 父级div定义zoom

## 前端项目中为什么要初始化CSS样式？

因为浏览器的兼容问题，不同浏览器对标签的默认值是不同的，如果没有对浏览器的CSS初始化，会造成相同页面在不同浏览器的显示存在差异。

## CSS匹配规则顺序是怎么样的？

相信大多数初学者都会认为CSS匹配是左向右的，其实恰恰相反。

CSS匹配发生在Render Tree构建时（Chrome Dev Tools将其归属于Layout过程）。此时浏览器构建出了DOM，而且拿到了CSS样式，此时要做的就是把样式跟DOM上的节点对应上，浏览器为了提高性能需要做的就是快速匹配。

首先要明确一点，浏览器此时是给一个"可见"节点找对应的规则，这和jQuery选择器不同，后者是使用一个规则去找对应的节点，这样从左到右或许更快。但是对于前者，由于CSS的庞大，一个CSS文件中或许有上千条规则，而且对于当前节点来说，大多数规则是匹配不上的，稍微想一下就知道，如果从右开始匹配（也是从更精确的位置开始），能更快排除不合适的大部分节点，而如果从左开始，只有深入了才会发现匹配失败，如果大部分规则层级都比较深，就比较浪费资源了。

除了上面这点，我们前面还提到DOM构建是"循序渐进的"，而且DOM不阻塞Render Tree构建（只有CSSOM阻塞），这样也是为了能让页面更早有元素呈现。

考虑如下情况，如果我们此时构建的只是部分DOM，而CSSOM构建完成，浏览器就会构建Render Tree。

这个时候对每一个节点，如果找到一条规则从右向左匹配，我们只需要逐层观察该节点父节点是否匹配，而此时其父节点肯定已经在DOM上。

但是反过来，我们可能会匹配到一个DOM上尚未存在的节点，此时的匹配过程就浪费了资源。

## 如何使用css完成视差滚动效果?

### 一、是什么

视差滚动（Parallax Scrolling）是指多层背景以不同的速度移动，形成立体的运动效果，带来非常出色的视觉体验

我们可以把网页解刨成：背景层、内容层、悬浮层

当滚动鼠标滑轮的时候，各个图层以不同的速度移动，形成视觉差的效果

### 二、实现方式

使用css形式实现视觉差滚动效果的方式有：

* background-attachment
* transform:translate3D

#### background-attachment

作用是设置背景图像是否固定或者随着页面的其余部分滚动

值分别有如下：

* scroll：默认值，背景图像会随着页面其余部分的滚动而移动
* fixed：当页面的其余部分滚动时，背景图像不会移动
* inherit：继承父元素background-attachment属性的值

完成滚动视觉差就需要将background-attachment属性设置为fixed，让背景相对于视口固定。及时一个元素有滚动机制，背景也不会随着元素的内容而滚动

也就是说，背景一开始就已经被固定在初始的位置

### 核心的css代码如下：

section {

height: 100vh;

}

.g-img {

background-image: url(...);

background-attachment: fixed;

background-size: cover;

background-position: center center;

}

#### transform:translate3D

同样，让我们先来看一下两个概念transform和perspective：

transform: css3 属性，可以对元素进行变换(2d/3d)，包括平移 translate,旋转 rotate,缩放 scale,等等

perspective: css3 属性，当元素涉及 3d 变换时，perspective 可以定义我们眼睛看到的 3d 立体效果，即空间感

而这种方式实现视觉差动的原理如下：

* 容器设置上 transform-style: preserve-3d 和 perspective: xpx，那么处于这个容器的子元素就将位于3D空间中，
* 子元素设置不同的 transform: translateZ()，这个时候，不同元素在 3D Z轴方向距离屏幕（我们的眼睛）的距离也就不一样
* 滚动滚动条，由于子元素设置了不同的 transform: translateZ()，那么他们滚动的上下距离 translateY 相对屏幕（我们的眼睛），也是不一样的，这就达到了滚动视差的效果

## 如何实现单行／多行文本溢出的省略样式？

### 单行文本溢出省略

理解也很简单，即文本在一行内显示，超出部分以省略号的形式展现

实现方式也很简单，涉及的css属性有：

text-overflow：规定当文本溢出时，显示省略符号来代表被修剪的文本

white-space：设置文字在一行显示，不能换行

overflow：文字长度超出限定宽度，则隐藏超出的内容

overflow设为hidden，普通情况用在块级元素的外层隐藏内部溢出元素，或者配合下面两个属性实现文本溢出省略

white-space:nowrap，作用是设置文本不换行，是overflow:hidden和text-overflow：ellipsis生效的基础

text-overflow属性值有如下：

clip：当对象内文本溢出部分裁切掉

ellipsis：当对象内文本溢出时显示省略标记（…）

text-overflow只有在设置了overflow:hidden和white-space:nowrap才能够生效的

<style>

p{

overflow: hidden;

line-height: 40px;

width:400px;

height:40px;

border:1px solid red;

text-overflow: ellipsis;

white-space: nowrap;

}

</style>

<p 这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本</p >

### 多行文本溢出省略

多行文本溢出的时候，我们可以分为两种情况：

* 基于高度截断
* 基于行数截断

#### 基于高度截断

伪元素 + 定位

核心的css代码结构如下：

* position: relative：为伪元素绝对定位
* overflow: hidden：文本溢出限定的宽度就隐藏内容）
* position: absolute：给省略号绝对定位
* line-height: 20px：结合元素高度,高度固定的情况下,设定行高, 控制显示行数
* height: 40px：设定当前元素高度
* ::after {} ：设置省略号样式

<style>

.demo {

position: relative;

line-height: 20px;

height: 40px;

overflow: hidden;

}

.demo::after {

content: "...";

position: absolute;

bottom: 0;

right: 0;

padding: 0 20px 0 10px;

}

</style>

<body>

<div class='demo'>这是一段很长的文本</div>

</body>

实现原理很好理解，就是通过伪元素绝对定位到行尾并遮住文字，再通过 overflow: hidden 隐藏多余文字

这种实现具有以下优点：

* 兼容性好，对各大主流浏览器有好的支持
* 响应式截断，根据不同宽度做出调整

一般文本存在英文的时候，可以设置word-break: break-all使一个单词能够在换行时进行拆分

#### 基于行数截断

纯css实现也非常简单，核心的css代码如下：

* -webkit-line-clamp: 2：用来限制在一个块元素显示的文本的行数，为了实现该效果，它需要组合其他的WebKit属性）
* display: -webkit-box：和1结合使用，将对象作为弹性伸缩盒子模型显示
* -webkit-box-orient: vertical：和1结合使用 ，设置或检索伸缩盒对象的子元素的排列方式
* overflow: hidden：文本溢出限定的宽度就隐藏内容
* text-overflow: ellipsis：多行文本的情况下，用省略号“…”隐藏溢出范围的文本

<style>

p {

width: 400px;

border-radius: 1px solid red;

-webkit-line-clamp: 2;

display: -webkit-box;

-webkit-box-orient: vertical;

overflow: hidden;

text-overflow: ellipsis;

}

</style>

<p>

这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本

这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本

</p >

可以看到，上述使用了webkit的CSS属性扩展，所以兼容浏览器范围是PC端的webkit内核的浏览器，由于移动端大多数是使用webkit，所以移动端常用该形式

需要注意的是，如果文本为一段很长的英文或者数字，则需要添加word-wrap: break-word属性

## 怎么使用 CSS3 实现动画？

一、是什么

CSS动画（CSS Animations）是为层叠样式表建议的允许可扩展标记语言（XML）元素使用CSS的动画的模块

即指元素从一种样式逐渐过渡为另一种样式的过程

常见的动画效果有很多，如平移、旋转、缩放等等，复杂动画则是多个简单动画的组合

css实现动画的方式，有如下几种：

* transition 实现渐变动画
* transform 转变动画
* animation 实现自定义动画

|  |  |
| --- | --- |
| **属性** | **含义** |
| transition（过度） | 用于设置元素的样式过度，和animation有着类似的效果，但细节上有很大的不同 |
| transform（变形） | 用于元素进行旋转、缩放、移动或倾斜，和设置样式的动画并没有什么关系，就相当于color一样用来设置元素的“外表” |
| translate（移动） | 只是transform的一个属性值，即移动 |
| animation（动画） | 用于设置动画属性，他是一个简写的属性，包含6个属性 |

## em/px/rem/vh/vw 这些单位有什么区别？

### 一、介绍

传统的项目开发中，我们只会用到px、%、em这几个单位，它可以适用于大部分的项目开发，且拥有比较良好的兼容性

从CSS3开始，浏览器对计量单位的支持又提升到了另外一个境界，新增了rem、vh、vw、vm等一些新的计量单位

利用这些新的单位开发出比较良好的响应式页面，适应多种不同分辨率的终端，包括移动设备等

### 二、单位

在css单位中，可以分为长度单位、绝对单位，如下表所指示

|  |  |
| --- | --- |
| **CSS单位** |  |
| 相对长度单位 | em、ex、ch、rem、vw、vh、vmin、vmax、% |
| 绝对长度单位 | cm、mm、in、px、pt、pc |
| **CSS单位** |  |

主要讲述px、em、rem、vh、vw

#### px

px，表示像素，所谓像素就是呈现在我们显示器上的一个个小点，每个像素点都是大小等同的，所以像素为计量单位被分在了绝对长度单位中

有些人会把px认为是相对长度，原因在于在移动端中存在设备像素比，px实际显示的大小是不确定

这里之所以认为px为绝对单位，在于px的大小和元素的其他属性无关

#### em

em是相对长度单位。相对于当前对象内文本的字体尺寸。如当前对行内文本的字体尺寸未被人为设置，则相对于浏览器的默认字体尺寸（1em = 16px）

为了简化 font-size 的换算，我们需要在css中的 body 选择器中声明font-size= 62.5%，这就使 em 值变为 16px\*62.5% = 10px

这样 12px = 1.2em, 10px = 1em, 也就是说只需要将你的原来的 px 数值除以 10，然后换上 em 作为单位就行了

特点：

em 的值并不是固定的

em 会继承父级元素的字体大小

em 是相对长度单位。相对于当前对象内文本的字体尺寸。如当前对行内文本的字体尺寸未被人为设置，则相对于浏览器的默认字体尺寸

任意浏览器的默认字体高都是 16px

#### rem

rem，相对单位，相对的只是HTML根元素font-size的值

同理，如果想要简化font-size的转化，我们可以在根元素html中加入font-size: 62.5%

这样页面中1rem=10px、1.2rem=12px、1.4rem=14px、1.6rem=16px;使得视觉、使用、书写都得到了极大的帮助

特点：

rem单位可谓集相对大小和绝对大小的优点于一身

和em不同的是rem总是相对于根元素，而不像em一样使用级联的方式来计算尺寸

#### vh、vw

vw ，就是根据窗口的宽度，分成100等份，100vw就表示满宽，50vw就表示一半宽。（vw 始终是针对窗口的宽），同理，vh则为窗口的高度

这里的窗口分成几种情况：

在桌面端，指的是浏览器的可视区域

移动端指的就是布局视口

像vw、vh，比较容易混淆的一个单位是%，不过百分比宽泛的讲是相对于父元素：

对于普通定位元素就是我们理解的父元素

对于position: absolute;的元素是相对于已定位的父元素

对于position: fixed;的元素是相对于 ViewPort（可视窗口）

### 三、总结

* px：绝对单位，页面按精确像素展示
* em：相对单位，基准点为父节点字体的大小，如果自身定义了font-size按自身来计算，整个页面内1em不是一个固定的值
* rem：相对单位，可理解为root em, 相对根节点html的字体大小来计算
* vh、vw：主要用于页面视口大小布局，在页面布局上更加方便简单

## z-index: 999 元素一定会置于 z-index: 0 元素之上吗【热度: 100】

设置了z-index: 999的元素不一定会置于z-index: 0的元素之上。

### 一、z-index 的作用机制

z-index属性用于控制元素在 z 轴上的堆叠顺序，即决定了元素在垂直于屏幕平面的方向上的前后显示顺序。

只有当元素的定位属性（如position: relative、position: absolute、position: fixed）被设置时，z-index才会生效。

### 二、影响堆叠顺序的其他因素

元素的堆叠上下文：

元素会根据其所在的堆叠上下文进行堆叠。堆叠上下文是一个三维的概念，包含一组元素，这些元素按照特定的规则进行堆叠。

创建堆叠上下文的因素包括：设置了定位属性和z-index的元素、opacity小于 1 的元素、transform属性不为none的元素等。

如果一个元素处于一个具有更高堆叠顺序的堆叠上下文中，即使它的z-index值较低，也可能会被置于另一个堆叠上下文中具有较低z-index值的元素之下。

元素的 HTML 结构顺序：

在没有设置z-index或堆叠上下文的情况下，元素的堆叠顺序通常遵循 HTML 结构的顺序。后出现的元素会覆盖先出现的元素。