CSS3 新特性

参考链接

1. 伪类和伪元素选择器：

> :first-child, :last-child, :nth-child(1), :link, :visited, :hover, :active
> ::before, ::after, :first-letter, :first-line, ::selection

2. 背景、边框和颜色透明度：

> background-size, background-origin, border-radius
> box-shadow, border-image
> rgba

3. 文字效果：

> text-shadow, word-wrap

4. 2D 转换和 3D 转换：

> transform, translate(), rotate(), scale(), skew(), matrix()
> rotateX(), rotateY(), perspective

5. 动画、过渡：animation, transition
6. 多列：column-count, column-gap, column-rule
7. 用户界面：resize, box-sizing, outline-offset

CSS 盒模型

盒模型总共包括4个部分：

- content：内容，容纳着元素的"真实"内容，例如文本，图像或是视频播放器
- padding：内边距
- border：边框
- margin：外边距

两种盒模型的区别：

- W3C盒模型 `box-sizing: content-box`

> W3C盒模型中，通过CSS样式设置的width的大小只是content的大小

- IE盒模型 `box-sizing: border-box`

> IE盒模型中，通过CSS样式设置的width的大小是content + padding + border的和

设置一个元素的背景颜色，背景颜色会填充哪些区域

> border + padding + content

demo 说明

```
div {
  width: 100px;
  height: 100px;
  background-color: pink;
  border: 5px dotted green;
}
```

效果图:

margin/padding 设置百分比是相对谁的

先来看一个案例:

> 假设一个div，宽400px，高200px，他有个子div的margin:10%，你来算下他的margin 的 top, right, bottom, left 是多少?

```
.outer {
    width: 400px;
    height: 200px;
    background-color: red;
    position: relative;
}
.inner {
    width: 100px;
    height: 100px;
    background-color: green;
    position: absolute;
    margin: 10%;
}

<div class="outer">
    <div class="inner"></div>
</div>
```

效果是子盒子的margin为40px 40px 40px 40px

总结：margin/padding设置百分比都是相对于父盒子的宽度(width属性)

link 和 @import 的区别

1. link 是 HTML 标签，不仅可以加载 CSS 文件，还可以定义 RSS、rel 连接属性等；@import 是 CSS 提供的语法，只有导入样式表的作用。
2. 加载页面时，link 标签引入的 CSS 被同时加载；@import 引入的 CSS 将在页面加载完毕后被加载。
3. @import 是 CSS2.1 才有的语法，故只可在 IE5+ 才能识别；link 标签作为 HTML 元素，不存在兼容性问题。
4. 可以通过 JS 操作 DOM ，插入 link 标签来改变样式；由于 DOM 方法是基于文档的，无法使用 @import 的方式插入样式。
5. link 引入的样式权重大于@import 引入的样式。

CSS 选择器的解析规则

从右向左，这样会提高查找选择器所对应的元素的效率

CSS 选择器优先级

选择器按优先级先后排列：!important>内联>id>class = 属性 = 伪类 >标签 = 伪元素 > 通配符 *

- important 声明 1,0,0,0
- ID 选择器 0,1,0,0
- 类选择器 0,0,1,0
- 伪类选择器 0,0,1,0
- 属性选择器 0,0,1,0
- 标签选择器 0,0,0,1
- 伪元素选择器 0,0,0,1
- 通配符选择器 0,0,0,0

::before 和:after 中双冒号和单冒号有什么区别？解释一下这 2 个伪元素的作用

在css3中使用单冒号来表示伪类，用双冒号来表示伪元素。但是为了兼容已有的伪元素的写法，在一些浏览器中也可以使用单冒号
来表示伪元素。

伪类一般匹配的是元素的一些特殊状态，如hover、link等，而伪元素一般匹配的特殊的位置，比如after、before等。

伪类与伪元素的区别

css引入伪类和伪元素概念是为了格式化文档树以外的信息。也就是说，伪类和伪元素是用来修饰不在文档树中的部分，比如，一句
话中的第一个字母，或者是列表中的第一个元素。

伪类用于当已有的元素处于某个状态时，为其添加对应的样式，这个状态是根据用户行为而动态变化的。比如说，当用户悬停在指定的
元素时，我们可以通过:hover来描述这个元素的状态。

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。它们允许我们为元素的某些部分设置样式。比如说，我们可以通过::be
fore来在一个元素前增加一些文本，并为这些文本添加样式。虽然用户可以看到这些文本，但是这些文本实际上不在文档树中。

有时你会发现伪元素使用了两个冒号（::）而不是一个冒号（:）。这是CSS3的一部分，并尝试区分伪类和伪元素。大多数浏览
器都支持这两个值。按照规则应该使用（::）而不是（:），从而区分伪类和伪元素。但是，由于在旧版本的W3C规范并未对此进行
特别区分，因此目前绝大多数的浏览器都支持使用这两种方式表示伪元素。

关于伪类 LVHA 的解释

a标签有四种状态：链接访问前、链接访问后、鼠标滑过、激活，分别对应四种伪类:link、:visited、:hover、:active；

当链接未访问过时：

（1）当鼠标滑过a链接时，满足:link和:hover两种状态，要改变a标签的颜色，就必须将:hover伪类在:link伪类后面声明；
（2）当鼠标点击激活a链接时，同时满足:link、:hover、:active三种状态，要显示a标签激活时的样式（:active），
必须将:active声明放到:link和:hover之后。因此得出LVHA这个顺序。

当链接访问过时，情况基本同上，只不过需要将:link换成:visited。

这个顺序能不能变？可以，但也只有:link和:visited可以交换位置，因为一个链接要么访问过要么没访问过，不可能同时满足，也就不存在覆盖的问题。

CSS 中哪些属性可以继承

每一个属性在定义中都给出了这个属性是否具有继承性，一个具有继承性的属性会在没有指定值的时候，会使用父元素的同属性的值
来作为自己的值。

一般具有继承性的属性有，字体相关的属性，font-size和font-weight等。文本相关的属性，color和text-align等。
表格的一些布局属性、列表属性如list-style等。还有光标属性cursor、元素可见性visibility。

当一个属性不是继承属性的时候，我们也可以通过将它的值设置为inherit来使它从父元素那获取同名的属性值来继承。

CSS 清除浮动的方式

1. 额外标签法

   > 在需要清除浮动的元素后面添加一个空白标签，设置类名 clear，设置`clear: both;`即可

2. 父级元素添加`overflow: hidden;`
3. 父元素`display: table`
4. 伪元素清除浮动

   > 对需要清除浮动的元素添加一个clearfix类名，设置样式如下：

```
.clearfix:after {
  /*正常浏览器 清除浮动*/
  content: '';
  display: block;
  height: 0;
  clear: both;
  visibility: hidden;
}
.clearfix {
  *zoom: 1;
  /*zoom 1 就是ie6 清除浮动方式  * ie7以下的版本才能识别 其他浏览器都不执行(略过)*/
}
```

清除浮动的原理

- clear属性清除浮动：clear 属性规定元素盒子的边不能和浮动元素相邻。该属性只能影响使用清除的元素本身，不能影响其他元素。换而言之，如果已经存在浮动元素的话，那么该元素就不会像原本元素一样受其影响了。
- 其他的可以归为一类，都是通过触发BFC来实现的

BFC的概念, 哪些元素可以触发BFC

> BFC 即 Block Formatting Context (块格式化上下文)， 是Web页面的可视化CSS渲染的一部分，是块盒子的布局过程发生的区域，也是浮动元素与其他元素交互的区域。

简单来说就是一个封闭的黑盒子，里面元素的布局不会影响外部。
下列方式会创建块格式化上下文：

- 根元素(<html>)
- 浮动元素（元素的 float 不是 none）
- 绝对定位元素（元素的 position 为 absolute 或 fixed）
- 行内块元素（元素的 display 为 inline-block）
- 表格单元格（元素的 display为 table-cell，HTML表格单元格默认为该值）
- 表格标题（元素的 display 为 table-caption，HTML表格标题默认为该值）
- 匿名表格单元格元素（元素的 display为 table、table-row table-row-group、table-header-group、table-footer-group（分别是- HTML table、row、tbody、thead、tfoot的默认属性）或 inline-table）
- overflow 值不为 visible 的块元素
- display 值为 flow-root 的元素
- contain 值为 layout、content或 paint 的元素
- 弹性元素（display为 flex 或 inline-flex元素的直接子元素）
- 网格元素（display为 grid 或 inline-grid 元素的直接子元素）
- 多列容器（元素的 column-count 或 column-width 不为 auto，包括 - column-count 为 1）
- column-span 为 all 的元素始终会创建一个新的BFC，即使该元素没有包裹在一个多列容器中（标准变更，Chrome bug）。

脱离文档流的方式

- float
- position: absolute
- position: fixed

position 的值定位原点是

`absolute`
生成绝对定位的元素，相对于值不为`static`的第一个父元素的`paddingbox`进行定位，也可以理解为离自己这一级元素最近的
一级`position`设置为`absolute`或者`relative`的父元素的`paddingbox`的左上角为原点的。

`fixed`（老`IE`不支持）
生成绝对定位的元素，相对于浏览器窗口进行定位。

`relative`
生成相对定位的元素，相对于其元素本身所在正常位置进行定位。

`static`
默认值。没有定位，元素出现在正常的流中（忽略`top`,`bottom`,`left`,`right`,`z-index`声明）。

sticky
元素根据正常文档流进行定位，然后相对它的最近滚动祖先（nearest scrolling ancestor）和 containing block (最近块级祖先 nearest block-level ancestor)，包括table-related元素，基于top, right, bottom, 和 left的值进行偏移。偏移值不会影响任何其他元素的位置。
该值总是创建一个新的层叠上下文（stacking context）。注意，一个sticky元素会“固定”在离它最近的一个拥有“滚动机制”的祖先上（当该祖先的overflow 是 hidden, scroll, auto, 或 overlay时），即便这个祖先不是真的滚动祖先。这个阻止了所有“sticky”行为（详情见Github issue on W3C CSSWG）。

display 有哪些值？说明他们的作用

- block
  块类型。默认宽度为父元素宽度，可设置宽高，换行显示。
- none
  元素不显示，并从文档流中移除。
- inline
  行内元素类型。默认宽度为内容宽度，不可设置宽高，同行显示。
- inline-block
  默认宽度为内容宽度，可以设置宽高，同行显示。
- list-item
  像块类型元素一样显示，并添加样式列表标记。
- table
  此元素会作为块级表格来显示。
- inherit
  规定应该从父元素继承display属性的值。

float 的元素，display 是什么

display 为 block

inline-block、inline 和 block 的区别；为什么 img 是 inline 还可以设置宽高

Block 是块级元素，其前后都会有换行符，能设置宽度，高度，margin/padding 水平垂直方向都有效。

Inline：设置 width 和 height 无效，margin 在竖直方向上无效，padding 在水平方向垂直方向都有效，前后无换行符

Inline-block：能设置宽度高度，margin/padding 水平垂直方向 都有效，前后无换行符

img 是可替换元素。

在 CSS 中，可替换元素（replaced element）的展现效果不是由 CSS 来控制的。这些元素是一种外部对象，它们外观的渲染，是独立于 CSS 的。
简单来说，它们的内容不受当前文档的样式的影响。CSS 可以影响可替换元素的位置，但不会影响到可替换元素自身的内容。
例如 `<iframe>` 元素，可能具有自己的样式表，但它们不会继承父文档的样式。

典型的可替换元素有：
  <iframe>
  <video>

<embed>
<img>

有些元素仅在特定情况下被作为可替换元素处理，例如：
<input> "image" 类型的 <input> 元素就像 <img> 一样可替换
<option>
<audio>
<canvas>
<object>
<applet>（已废弃）
CSS 的 content 属性用于在元素的 ::before 和 ::after 伪元素中插入内容。使用 content 属性插入的内容都是匿名的可替换元素。

flex 的属性有哪些

flex布局是CSS3新增的一种布局方式，我们可以通过将一个元素的display属性值设置为flex从而使它成为一个flex容器，它的所有子元素都会成为它的项目。

一个容器默认有两条轴，一个是水平的主轴，一个是与主轴垂直的交叉轴。我们可以使用flex-direction来指定主轴的方向。我们可以使用justify-content来指定元素在主轴上的排列方式，使用align-items来指定元素在交叉轴上的排列方式。还可以使用flex-wrap来规定当一行排列不下时的换行方式。

对于容器中的项目，我们可以使用order属性来指定项目的排列顺序，还可以使用flex-grow来指定当排列空间有剩余的时候，项目的放大比例。还可以使用flex-shrink来指定当排列空间不足时，项目的缩小比例。

> 以下 6 个属性设置在容器上。

- flex-direction 属性决定主轴的方向（即项目的排列方向）。
- flex-wrap 属性定义，如果一条轴线排不下，如何换行。
- flex-flow 属性是 flex-direction 属性和 flex-wrap 属性的简写形式，默认值为 rownowrap。
- justify-content 属性定义了项目在主轴上的对齐方式。
- align-items 属性定义项目在交叉轴上如何对齐。
- align-content 属性定义了多根轴线的对齐方式。如果项目只有一根轴线，该属性不起作用。

> 以下 6 个属性设置在项目上：

- order 属性定义项目的排列顺序。数值越小，排列越靠前，默认为 0。
- flex-grow 属性定义项目的放大比例，默认为 0，即如果存在剩余空间，也不放大。
- flex-shrink 属性定义了项目的缩小比例，默认为 1，即如果空间不足，该项目将缩小。
- flex-basis 属性定义了在分配多余空间之前，项目占据的主轴空间。浏览器根据这个属性，计算主轴是否有多余空间。它的默认值为 auto，即项目的本来大小。
- flex 属性是 flex-grow，flex-shrink 和 flex-basis 的简写，默认值为 01auto。
- align-self 属性允许单个项目有与其他项目不一样的对齐方式，可覆盖 align-items 属性。默认值为 auto，表示继承父元素的 align-items 属性，如果没有父元素，则等同于 stretch。

`visibility: hidden`,`opacity: 0`, `display: none`

opacity: 0，该元素隐藏起来了，但不会改变页面布局，并且，如果该元素已经绑定一些事件，如 click 事件，那么点击该区域，也能触发点击事件的；

visibility: hidden，该元素隐藏起来了，但不会改变页面布局，但是不会触发该元素已经绑定的事件；

display: none，把元素隐藏起来，并且会改变页面布局，可以理解成在页面中把该元素删除掉一样。

了解重绘和重排吗，知道怎么去减少重绘和重排吗，让文档脱离文档流有哪些方法

DOM 的变化影响到了预算内宿的几何属性比如宽高，浏览器重新计算元素的几何属性，其他元素的几何属性也会受到影响，浏览器需要重新构造渲染书，这个过程称之为重排，浏览器将受到影响的部分重新绘制在屏幕上 的过程称为重绘，引起重排重绘的原因有：

- 添加或者删除可见的 DOM 元素，
- 元素尺寸位置的改变
- 浏览器页面初始化，
- 浏览器窗口大小发生改变，重排一定导致重绘，重绘不一定导致重排，

减少重绘重排的方法有：

- 不在布局信息改变时做 DOM 查询，
- 使用 csstext,className 一次性改变属性
- 使用 fragment
- 对于多次重排的元素，比如说动画。使用绝对定位脱离文档流，使其不影响其他元素
- "editor.renderIndentGuides": true

z-index 是干什么用的？默认值是什么？与 z-index: 0 的区别

参考链接：[搞懂Z-index的所有细节]

> z-index 属性设置元素的堆叠顺序，且只在属性position: relative/absolute/fixed 的时候才生效。
> `z-index: auto` 是默认值，与`z-index: 0`是有区别的：
> `z-index: 0` 会创建一个新的堆叠上下文，而 `z-index: auto` 不会创建新的堆叠上下文

举例：考虑如下这种情况

```
<div class="A">
  <div class="a"></div>
</div>
<div class="B">
  <div class="b"></div>
</div>
```

> 上图中div的`z-index`均为整数的时候div(a)的`z-index`虽然比div(B)大，但是div(A)和div(a)是在一个堆叠上下文，而div(B)和div(b)是在一个堆叠上下文，这两个堆叠上下文是通过父级也就是div(A)和div(B)的`z-index`来决定层叠顺序的。

> 上图将div(A)的z-index设置为auto，这时候因为`z-index: auto` 不会创建新的堆叠上下文，因而div(a)的`z-index`比div(B)大，所以div(a)会在div(B)的上面

总结：

1. 当Z-index的值设置为auto时,不建立新的堆叠上下文,当前堆叠上下文中生成的div的堆叠级别与其父项的框相同。
2. 当Z-index的值设置为一个整数时,该整数是当前堆叠上下文中生成的div的堆栈级别。该框还建立了其堆栈级别的本地堆叠上下文。这意味着后代的z-index不与此元素之外的元素的z-index进行比

较。

vw 和 vh 的概念

vw（Viewport Width）、vh(Viewport Height)是基于视图窗口的单位，是css3的一部分，基于视图窗口的单位，除了vw、vh还有vmin、vmax。

- vw:1vw 等于视口宽度的1%
- vh:1vh 等于视口高度的1%
- vmin: 选取 vw 和 vh 中最小的那个,即在手机竖屏时，1vmin=1vw
- vmax:选取 vw 和 vh 中最大的那个 ,即在手机竖屏时，1vmax=1vh

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

（1）png24位的图片在iE6浏览器上出现背景
解决方案：做成PNG8，也可以引用一段脚本处理。

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

（3）IE6双边距bug：在IE6下，如果对元素设置了浮动，同时又设置了margin-left或margin-right，margin值会加倍。

```
#box{float:left;width:10px;margin:0001Opx;}
```

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

（4）渐进识别的方式，从总体中逐渐排除局部。
首先，巧妙的使用"\9"这一标记，将IE游览器从所有情况中分离出来。
接着，再次使用"+"将IE8和IE7、IE6分离开来，这样IE8已经独立识别。

```
.bb{
background-color:#f1ee18;/*所有识别*/
.background-color:#00deff\9;/*IE6、7、8识别*/
+background-color:#a200ff;/*IE6、7识别*/
_background-color:#1e0bd1;/*IE6识别*/
}
```

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

（6）IE下，event对象有x、y属性，但是没有pageX、pageY属性;Firefox下，event对象有pageX、pageY属性，但是没有x、y属性。
解决方法：（条件注释）缺点是在IE浏览器下可能会增加额外的HTTP请求数。

（7）Chrome中文界面下默认会将小于12px的文本强制按照12px显示
解决方法：

1.可通过加入CSS属性-webkit-text-size-adjust:none;解决。但是，在chrome更新到27版本之后就不可以用了。

2.还可以使用-webkit-transform:scale(0.5);注意-webkit-transform:scale(0.75);

收缩的是整个span的大小，这时候，必须要将span转换成块元素，可以使用display：block/inline-block/...；

（8）超链接访问过后hover样式就不出现了，被点击访问过的超链接样式不再具有hover和active了
解决方法：改变CSS属性的排列顺序L-V-H-A

（9）怪异模式问题：漏写DTD声明，Firefox仍然会按照标准模式来解析网页，但在IE中会触发怪异模式。为避免怪异模式给我们带来不必要的麻烦，最好养成书写DTD声明的好习惯。

简单介绍使用图片 base64 编码的优点和缺点

base64编码是一种图片处理格式，通过特定的算法将图片编码成一长串字符串，在页面上显示的时候，可以用该字符串来代替图片的
url属性。

使用base64的优点是：

（1）减少一个图片的HTTP请求

使用base64的缺点是：

（1）根据base64的编码原理，编码后的大小会比原文件大小大1/3，如果把大图片编码到html/css中，不仅会造成文件体
积的增加，影响文件的加载速度，还会增加浏览器对html或css文件解析渲染的时间。

（2）使用base64无法直接缓存，要缓存只能缓存包含base64的文件，比如HTML或者CSS，这相比域直接缓存图片的效果要
差很多。

（3）兼容性的问题，ie8以前的浏览器不支持。

一般一些网站的小图标可以使用base64图片来引入。

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

阐述一下 CSSSprites

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CSSSprites能很好地减少网页的http请求，从而很好的提高页面的性能；CSSSprites能减少图片的字节。

优点：
减少HTTP请求数，极大地提高页面加载速度
增加图片信息重复度，提高压缩比，减少图片大小
更换风格方便，只需在一张或几张图片上修改颜色或样式即可实现

缺点：
图片合并麻烦
维护麻烦，修改一个图片可能需要重新布局整个图片，样式

画一条 0.5px 的线

采用metaviewport的方式

采用border-image的方式

采用transform:scale()的方式

transition 和 animation 的区别

transition关注的是CSSproperty的变化，property值和时间的关系是一个三次贝塞尔曲线。

animation作用于元素本身而不是样式属性，可以使用关键帧的概念，应该说可以实现更自由的动画效果。

如何实现单行 / 多行文本溢出的省略（...）

```
/*单行文本溢出*/
p {
  overflow: hidden;
  text-overflow: ellipsis;
  white-space: nowrap;
}


/*多行文本溢出*/
p {
  position: relative;
  line-height: 1.5em;
  /*高度为需要显示的行数*行高，比如这里我们显示两行，则为3*/
  height: 3em;
  overflow: hidden;
}

p:after {
  content: "...";
  position: absolute;
  bottom: 0;
  right: 0;
  background-color: #fff;
}
```

常见的元素隐藏方式

1. 使用 display:none;隐藏元素，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。
2. 使用 visibility:hidden;隐藏元素。元素在页面中仍占据空间，但是不会响应绑定的监听事件。
3. 使用 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。
4. 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。
5. 通过 z-index 负值，来使其他元素遮盖住该元素，以此来实现隐藏。
6. 通过 clip/clip-path 元素裁剪的方法来实现元素的隐藏，这种方法下，元素仍在页面中占据位置，但是不会响应绑定的监听事件。
7. 通过 transform:scale(0,0)来将元素缩放为 0，以此来实现元素的隐藏。这种方法下，元素仍在页面中占据位置，但是不会响应绑定的监听事件。

CSS3 @font-face 有用过吗

@font-face 语句是 css 中的一个功能模块，用于实现网页字体多样性(设计者可随意指定字体，不需要考虑浏览者电脑上是否安装)。
语法：

- 字体的名称
- 字体文件包含在您的服务器上的某个地方，如果字体文件是在不同的位置，请使用完整的 URL 比如：url('http://www.w3cschool.css/css3/Sansation_Light.ttf')

```
@font-face {
  font-family: myFirstFont;
  src: url('Sansation_Light.ttf'),
      url('Sansation_Light.eot'); /* IE9 */
}
```

CSS 实现隔行变色

```
/* 方法一 */
li:nth-child(odd) {background:#ff0000;}
li:nth-child(even) {background:#0000ff;}

/* 方法二 */
li:nth-of-type(odd) { background:#00ccff;} /* 奇数行 */
li:nth-of-type(even) { background:#ffcc00;} /* 偶数行 */
```

注意：nth-child() 和 nth-of-type() 的区别

- nth-child() 就是根据元素的个数来计算的
- nth-of-type() 是根据类型来计算的，也就是 `li:nth-of-type(2)` 表示的是第 2 个 li 标签

一个满屏品字布局如何设计

```
简单的方式：
上面的div宽100%，
下面的两个div分别宽50%，
然后用float或者inline使其不换行即可
```

CSS 画三角形

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

CSS 画扇形

```
div {
  height: 0;
  width: 0;
  border: 100px solid transparent;
  border-radius: 50%;
  border-left-color: red;
}
```

效果图:

CSS 画正方体

```
.cube {
  font-size: 4em;
  /* 相对于父元素的大小，父元素是16px，这里是64px */
  width: 2em;
  /* 32px */
  margin: 1.5em auto;
  transform-style: preserve-3d;
  transform: rotateX(-35deg) rotateY(30deg);
}

.side {
  position: absolute;
  width: 2em;
  /* 相对于父元素的64px的大小，也就是128px */
  height: 2em;
  background: rgba(255, 99, 71, 0.6);
  border: 1px solid rgba(0, 0, 0, 0.5);
  color: white;
  text-align: center;
  line-height: 2em;
}

.front {
  transform: translateZ(1em);
}

.bottom {
  transform: rotateX(90deg) translateZ(-1em);
}

.top {
  transform: rotateX(90deg) translateZ(1em);
}


.left {
  transform: rotateY(-90deg) translateZ(1em);
}

.right {
  transform: rotateY(-90deg) translateZ(-1em);
}

.back {
```

```
  transform: translateZ(-1em);
}
```

```
<div class="cube">
  <div class="side front">1</div>
  <div class="side back">6</div>
  <div class="side right">4</div>
  <div class="side left">3</div>
  <div class="side top">5</div>
  <div class="side bottom">2</div>
</div>
```

效果图:

CSS 实现一个硬币旋转的效果

> 参考网站

```
#euro {
   width: 150px;
   height: 150px;
   margin-left: -75px;
   margin-top: -75px;
   position: absolute;
   top: 50%;
   left: 50%;
   transform-style: preserve-3d;
   animation: spin 2.5s linear infinite;
 }
 .back {
   background-image: url("/uploads/160101/backeuro.png");
   width: 150px;
   height: 150px;
 }
 .middle {
   background-image: url("/uploads/160101/faceeuro.png");
   width: 150px;
   height: 150px;
   transform: translateZ(1px);
   position: absolute;
   top: 0;
 }
 .front {
   background-image: url("/uploads/160101/faceeuro.png");
   height: 150px;
   position: absolute;
   top: 0;
   transform: translateZ(10px);
   width: 150px;
 }
 @keyframes spin {
   0% {
     transform: rotateY(0deg);
   }
   100% {
     transform: rotateY(360deg);
```

```
    }
  }
```

CSS 实现垂直居中

- 定位 + 负边距
- display: flex 弹性布局

```
.outer {
  display: flex;
  align-items: center;
  justify-content: center;
}
.inner {
  width: 400px;
  height: 400px;
  background-color: red;
}

<div class="outer">
  <div class="inner"></div>
</div>
```

- display: table
- 绝对居中

```
div {
  width: 300px;
  height: 300px;
  background-color: red;
  position: absolute;
  left: 0;
  right: 0;
  top: 0;
  bottom: 0;
  margin: auto;
}
```

CSS 实现两列固定，中间自适应的布局

> 左右各占200px，中间随着窗口的调整自适应

HTML代码如下：

```
<div class="div">
  <div class="left">left</div>
  <div class="right">right</div>
  <div class="main">main</div>
</div>
```

- 方法一：通过定位的方式，中间一列通过 `margin: auto` 实现自适应

  CSS样式：

```
.left {
  width: 200px;
  height: 100%;
  background-color: red;
  position: absolute;
  left: 0;
}
.right {
  width: 200px;
  height: 100%;
  background-color: red;
  position: absolute;
  right: 0;
}
.main {
  height: 100%;
  background-color: green;
  position: absolute;
  left: 200px;
  right: 200px;
  margin: auto; /* 这个千万不能少 */
}
```

- 方法二：flex布局
  CSS样式：

```
html, body {
  height: 100%;
}
.div {
  height: 100%; /* 想要实现高度百分百必须让父元素都有高度 */
  display: flex;
}
.left  {
  flex: 0 0 200px;
  order: 1; /* order定义显示的顺序 */
  background-color: red;
}
.right{
  flex: 0 0 200px;
  order: 3;
  background-color: red;
}
.main{
  flex: auto;
  order: 2;
  background-color: green;
}
```

- 方法三：左右浮动布局，这种布局方式，必须先写浮动部分，最后再写中间部分，否则右浮动块会掉到下一行。
  CSS样式：

```
html, body{
  height: 100%;
```

```
}
.div{
  height: 100%;
}
.left {
  float: left;
  width: 200px;
  height: 100%;
  background-color: red;
}
.right {
  float: right;
  width: 200px;
  height: 100%;
  background-color: red;
}
.main{
  height: 100%;
  background-color: green;
}
```

实现自适应九宫格

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        html, body {
            width: 100%;
            height: 100%;
        }
        .box {
            width: 100%;
            height: 100%;
            display: flex;
            align-items: center;
            justify-content: center;
            flex-wrap: wrap;
        }
        .item {
            width: 30%;
            height: 30%;
            margin: 1%;
            background-color: #cccccc;
        }
    </style>
</head>

<body>
    <div class="box">
        <div class="item"></div>
        <div class="item"></div>
        <div class="item"></div>
```

```
        <div class="item"></div>
        <div class="item"></div>
        <div class="item"></div>
        <div class="item"></div>
        <div class="item"></div>
        <div class="item"></div>
    </div>
</body>

</html>
```

屏幕里面内容未占满的时候footer固定在屏幕可视区域的底部。占满的时候显示在网页的最底端

- 方式一

```
<style>
    html,
    body {
        height: 100%;
        margin: 0;
        padding: 0;
    }

    .page {
        box-sizing: border-box;
        min-height: 100%;
        padding-bottom: 300px;
        background-color: gray;
        /* height: 2000px; */
    }

    footer {
        height: 300px;
        margin-top: -300px;
        background-color: #ccc;
</style>

<div class="page">
    主要页面
</div>
<footer>底部</footer>
```

- 方式二

```
<style>
    html,
    body {
        height: 100%;
        margin: 0;
        padding: 0;
    }

    .container {
        position: relative;
        min-height: 100%;
```

```
    }

    .page {
        background-color: gray;
        /* height: 2000px; */
    }

    footer {
        position: absolute;
        left: 0;
        bottom: 0;
        height: 300px;
        width: 100%;
        background-color: #ccc;
</style>

<div class="container">
    <div class="page">
        主要页面
    </div>
    <footer>底部</footer>
</div>
```

HTML5 的新特性

参考链接

1. 语义化标签:header、footer、section、nav、aside、article
2. 增强型表单：input的多个type(color, date, datetime, email, month, number, range, search, tel, time, url, week)
3. 新增表单元素：datalist、keygen、output
4. 新增表单属性：placehoder, required, min和max, step, height 和 width, autofocus, multiple
5. 音频视频：audio、video
6. canvas
7. 地理定位：
8. 拖拽：drag
9. 本地存储：localStorage、sessionStorage
10. 新事件：onresize、ondrag、onscroll、onmousewheel、onerror、onplay、onpause
11. WebSocket

localStorage，sessionStorage，cookie 的区别

共同点：都是保存在浏览器端。
区别：
(1) cookie数据始终在同源的http请求中携带（即使不需要），即cookie在浏览器和服务器间来回传递
    sessionStorage和localStorage不会自动把数据发给服务器，仅在本地保存
(2) 存储大小限制也不同
    cookie数据不能超过4k
    sessionStorage和localStorage 虽然也有存储大小的限制，但比cookie大得多，可以达到5M或更大
(3) 数据有效期不同
    sessionStorage：仅在当前浏览器窗口关闭前有效，自然也就不可能持久保持；
    localStorage：始终有效，窗口或浏览器关闭也一直保存，因此用作持久数据；
    cookie只在设置的cookie过期时间之前一直有效，即使窗口或浏览器关闭；
(4) 作用域不同

sessionStorage不在不同的浏览器窗口中共享，即使是同一个页面；
localStorage 在所有同源窗口中都是共享的；
cookie也是在所有同源窗口中都是共享的

DOCTYPE的作用是什么

(1) <!DOCTYPE>声明一般位于第一行，处于 <html> 标签前。作用：告诉浏览器以什么样的模式来解析文档。DOCTYPE 不存在或格式不正确会导致文档以兼容模式呈现。
(2) 一般指定了之后会以标准模式来进行文档解析，否则就以兼容模式进行解析。在标准模式下，浏览器的解析规则都是按照最新的标准进行解析的;在兼容模式下，浏览器会以向后兼容的方式来模拟老式浏览器的行为，以保证一些老的网站的正确访问。

语义化标签的作用

- 去掉或样式丢失的时候能让页面呈现清晰的结构
- 方便其他设备解析（如屏幕阅读器、盲人阅读器、移动设备）以意义的方式来渲染网页
- 有利于SEO
- 便于团队开发和维护，遵循W3C标准，可以减少差异化

行内元素 块级元素

HTML4中，元素被分成两大类：inline （内联元素）与 block （块级元素）。
（1） 格式上，默认情况下，行内元素不会以新行开始，而块级元素会新起一行。
（2） 内容上，默认情况下，行内元素只能包含文本和其他行内元素。而块级元素可以包含行内元素和其他块级元素。
（3） 行内元素与块级元素属性的不同，主要是盒模型属性上：行内元素设置 width 无效，height 无效（可以设置 line-height），设置 margin 和 padding 的上下不会对其他元素产生影响。
（4）常见的行内元素有 a b span img strong sub sup button input label select textarea
（5）常见的块级元素有 div p ul ol li dl dt dd h1 h2 h3 h4 h5 h6

canvas与SVG

- canvas与svg都是可以在浏览器上创建图形
- HTML5 的 canvas 元素使用 JavaScript 在网页上绘制图像。画布是一个矩形区域，您可以控制其每一像素。canvas 拥有多种绘制路径、矩形、圆形、字符以及添加图像、动画的方法
- canvas绘制位图,绘制出来的每一个图形的元素都是独立的DOM节点，能够方便的绑定事件或用来修改。canvas复杂度高会减慢渲染速度（任何过度使用 DOM 的应用都不快）。canvas输出的是一整幅画布，就像一张图片一样，放大会失真。canvas不适合游戏应用。
- svg输出的图形是矢量图形，后期可以修改参数来自由放大缩小，SVG 图像在放大或改变尺寸的情况下其图形质量不会有所损失。svg最适合图像密集型的游戏，其中的许多对象会被频繁重绘

js 基本数据类型

> js 一共有六种基本数据类型，分别是 Undefined、Null、Boolean、Number、String，还有在 ES6 中新增的 Symbol 类型，代表创建后独一无二且不可变的数据类型，它的出现我认为主要是为了解决可能出现的全局变量冲突的问题。

js 遍历对象和遍历数组的方式

遍历对象

- Object.keys()

> 返回一个数组,包括对象自身的(不含继承的)所有可枚举属性(不含Symbol属性).

```
let obj = {
    name: 'lee',
    sex: 'male',
    age: 18
}
Object.keys(obj).forEach(key => {
    console.log(key, obj[key]);
})

// name lee
// sex male
// age 18
```

- for...in

> 循环遍历对象自身的和继承的可枚举属性(不含Symbol属性).

```
let obj = {
    name: 'lee',
    sex: 'male',
    age: 18
}
for(let key in obj) {
    console.log(key, obj[key]);
}

// name lee
// sex male
// age 18
```

- Object.getOwnPropertyNames()

> 返回一个数组,包含对象自身的所有属性(不含Symbol属性,但是包括不可枚举属性).

```
let obj = {
    name: 'lee',
    sex: 'male',
    age: 18
}
Object.getOwnPropertyNames(obj).forEach(key => {
    console.log(key, obj[key]);
})

// name lee
// sex male
// age 18
```

- Reflect.ownKeys()

> 返回一个数组,包含对象自身的所有属性(包括Symbol属性和不可枚举属性).

```
let obj = {
    name: 'lee',
    sex: 'male',
    age: 18
}
Reflect.ownKeys(obj).forEach(key => {
    console.log(key, obj[key]);
})

// name lee
// sex male
// age 18
```

遍历数组

- forEach()

```
let arr = [1,2,3];
arr.forEach(e => {
    console.log(e);
})

// 1
// 2
// 3
```

- for...in

> 注意for...in遍历的是索引

```
let arr = [1,2,3];
for(let index in arr) {
    console.log(arr[index]);
}

// 1
// 2
// 3
```

- for...of

```
let arr = [1,2,3];
for(let ele of arr) {
    console.log(ele);
}

// 1
// 2
// 3
```

null 和 undefined 的区别

- undefined 和 null 都是基本数据类型，这两个基本数据类型分别都只有一个值，就是 undefined 和 null。

- undefined 代表的含义是未定义，null 代表的含义是空对象。一般变量声明了但还没有定义的时候会返回 undefined，null主要用于赋值给一些可能会返回对象的变量，作为初始化。
- undefined 在 js 中不是一个保留字，这意味着我们可以使用 undefined 来作为一个变量名，这样的做法是非常危险的，它会影响我们对 undefined 值的判断。但是我们可以通过一些方法获得安全的 undefined 值，比如说 void 0。
- 当我们对两种类型使用 typeof 进行判断的时候，null 类型化会返回 “object”，这是一个历史遗留的问题。
- `undefined==null(true)` `undefined===null(false)`

其他值到字符串的转换规则

规范的 9.8 节中定义了抽象操作 ToString ，它负责处理非字符串到字符串的强制类型转换。

（1）null 和 undefined 类型 ，null 转换为 "null"，undefined 转换为 "undefined"，

（2）Boolean 类型，true 转换为 "true"，false 转换为 "false"。

（3）Number 类型的值直接转换，不过那些极小和极大的数字会使用指数形式。

（4）Symbol 类型的值直接转换，但是只允许显式强制类型转换，使用隐式强制类型转换会产生错误。

（5）对普通对象来说，除非自行定义 toString() 方法，否则会调用 toString()（Object.prototype.toString()）
来返回内部属性 [[Class]] 的值，如"[object Object]"。如果对象有自己的 toString() 方法，字符串化时就会
调用该方法并使用其返回值。

其他值到数字值的转换规则

有时我们需要将非数字值当作数字来使用，比如数学运算。为此 ES5 规范在 9.3 节定义了抽象操作 ToNumber。

（1）undefined 类型的值转换为 NaN。

（2）null 类型的值转换为 0。

（3）Boolean 类型的值，true 转换为 1，false 转换为 0。

（4）String 类型的值转换如同使用 Number() 函数进行转换，如果包含非数字值则转换为 NaN，空字符串为 0。

（5）Symbol 类型的值不能转换为数字，会报错。

（6）对象（包括数组）会首先被转换为相应的基本类型值，如果返回的是非数字的基本类型值，则再遵循以上规则将其强制转换为数字。

为了将值转换为相应的基本类型值，抽象操作 ToPrimitive 会首先（通过内部操作 DefaultValue）检查该值是否有valueOf() 方法。
如果有并且返回基本类型值，就使用该值进行强制类型转换。如果没有就使用 toString() 的返回值（如果存在）来进行强制类型转换。
如果 valueOf() 和 toString() 均不返回基本类型值，会产生 TypeError 错误。

其他值到布尔类型的值的转换规则

ES5 规范 9.2 节中定义了抽象操作 ToBoolean，列举了布尔强制类型转换所有可能出现的结果。

以下这些是假值：

- undefined
- null
- false
- +0、-0 和 NaN
- ""

假值的布尔强制类型转换结果为 false。从逻辑上说，假值列表以外的都应该是真值。

{} 和 [] 的 valueOf 和 toString 的结果是什么

{} 的 valueOf 结果为 {} ，toString 的结果为 "[object Object]"

[] 的 valueOf 结果为 [] ，toString 的结果为 ""

什么情况下会发生布尔值的隐式强制类型转换

（1） if (..) 语句中的条件判断表达式。
（2） for (.. ; .. ; ..) 语句中的条件判断表达式（第二个）。
（3） while (..) 和 do..while(..) 循环中的条件判断表达式。
（4） ? : 中的条件判断表达式。
（5） 逻辑运算符 ||（逻辑或）和 &&（逻辑与）左边的操作数（作为条件判断表达式）。

|| 和 && 操作符的返回值

|| 和 && 首先会对第一个操作数执行条件判断，如果其不是布尔值就先进行 ToBoolean 强制类型转换，然后再执行条件判断。

对于 || 来说，如果条件判断结果为 true 就返回第一个操作数的值，如果为 false 就返回第二个操作数的值。

&& 则相反，如果条件判断结果为 true 就返回第二个操作数的值，如果为 false 就返回第一个操作数的值。

|| 和 && 返回它们其中一个操作数的值，而非条件判断的结果

== 操作符的强制类型转换规则

（1）字符串和数字之间的相等比较，将字符串转换为数字之后再进行比较。

（2）其他类型和布尔类型之间的相等比较，先将布尔值转换为数字后，再应用其他规则进行比较。

（3）null 和 undefined 之间的相等比较，结果为真。其他值和它们进行比较都返回假值。

（4）对象和非对象之间的相等比较，对象先调用 ToPrimitive 抽象操作后，再进行比较。

（5）如果一个操作值为 NaN ，则相等比较返回 false（ NaN 本身也不等于 NaN ）。

（6）如果两个操作值都是对象，则比较它们是不是指向同一个对象。如果两个操作数都指向同一个对象，则相等操作符返回 true，否则，返回 false。

如何将字符串转化为数字，例如 '12.3b'

（1）使用 Number() 方法，前提是所包含的字符串不包含不合法字符。

（2）使用 parseInt() 方法，parseInt() 函数可解析一个字符串，并返回一个整数。还可以设置要解析的数字的基数。当基数的值为 0，或没有设置该参数时，parseInt() 会根据 string 来判断数字的基数。

（3）使用 parseFloat() 方法，该函数解析一个字符串参数并返回一个浮点数。

JavaScript 继承的几种实现方式

- 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。
- 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。
- 第三种方式是组合继承，组合继承是将原型链和借用构造函数组合起来使用的一种方式。通过借用构造函数的方式来实现类型的属性的继承，通过将子类型的原型设置为超类型的实例来实现方法的继承。这种方式解决了上面的两种模式单独使用时的问题，但是由于我们是以超类型的实例来作为子类型的原型，所以调用了两次超类的构造函数，造成了子类型的原型中多了很多不必要的属性。
- 第四种方式是原型式继承，原型式继承的主要思路就是基于已有的对象来创建新的对象，实现的原理是，向函数中传入一个对象，然后返回一个以这个对象为原型的对象。这种继承的思路主要不是为了实现创造一种新的类型，只是对某个对象实现一种简单继承，ES5 中定义的 Object.create() 方法就是原型式继承的实现。缺点与原型链方式相同。
- 第五种方式是寄生式继承，寄生式继承的思路是创建一个用于封装继承过程的函数，通过传入一个对象，然后复制一个对象的副本，然后对象进行扩展，最后返回这个对象。这个扩展的过程就可以理解是一种继承。这种继承的优点就是对一个简单对象实现继承，如果这个对象不是我们的自定义类型时。缺点是没有办法实现函数的复用。
- 第六种方式是寄生式组合继承，组合继承的缺点就是使用超类型的实例做为子类型的原型，导致添加了不必要的原型属性。寄生式组合继承的方式是使用超类型的原型的副本来作为子类型的原型，这样就避免了创建不必要的属性。

eval 是做什么的

它的功能是把对应的字符串解析成 JS 代码并运行。

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

事件对象中的clientX offsetX screenX pageX的区别

- [clientX, clientY]

client直译就是客户端，客户端的窗口就是指游览器的显示页面内容的窗口大小（不包含工具栏、导航栏等等）
[clientX, clientY]就是鼠标距游览器显示窗口的长度

兼容性：IE和主流游览器都支持。

- [offsetX, offsetY]

offset意为偏移量
[offsetX, offsetY]是被点击的元素距左上角为参考原点的长度，而IE、FF和Chrome的参考点有所差异。

Chrome下，offsetX offsetY是包含边框的
IE、FF是不包含边框的，如果鼠标进入到border区域，为返回负值

兼容性：IE9+,chrome,FF都支持此属性。

- [screenX, screenY]

screen顾名思义是屏幕
[screenX, screenY]是用来获取鼠标点击位置到屏幕显示器的距离，距离的最大值需根据屏幕分辨率的尺寸来计算。

兼容性：所有游览器都支持此属性。

- [pageX, pageY]

page为页面的意思，页面的高度一般情况client浏览器显示区域装不下，所以会出现垂直滚动条。

[pageX, pageY]是鼠标距离页面初始page原点的长度。

在IE中没有pageX、pageY取而代之的是event.x、event.y。x和y在webkit内核下也实现了，所以火狐不支持x，y。

兼容性：IE不支持，其他高级游览器支持。

三种事件模型是什么

事件是用户操作网页时发生的交互动作或者网页本身的一些操作，现代浏览器一共有三种事件模型。

第一种事件模型是最早的 DOM0 级模型，这种模型不会传播，所以没有事件流的概念，但是现在有的浏览器支持以冒泡的方式实现，
它可以在网页中直接定义监听函数，也可以通过 js 属性来指定监听函数。这种方式是所有浏览器都兼容的。

第二种事件模型是 IE 事件模型，在该事件模型中，一次事件共有两个过程，事件处理阶段，和事件冒泡阶段。事件处理阶段会首先执行
目标元素绑定的监听事件。然后是事件冒泡阶段，冒泡指的是事件从目标元素冒泡到 document，依次检查经过的节点是否绑定了事件监听函数，
如果有则执行。这种模型通过 attachEvent 来添加监听函数，可以添加多个监听函数，会按顺序依次执行。

第三种是 DOM2 级事件模型，在该事件模型中，一次事件共有三个过程，第一个过程是事件捕获阶段。捕获指的是事件从 document 一直向下
传播到目标元素，依次检查经过的节点是否绑定了事件监听函数，如果有则执行。后面两个阶段和 IE 事件模型的两个阶段相同。这种事件模型，
事件绑定的函数是 addEventListener，其中第三个参数可以指定事件是否在捕获阶段执行。默认是 false，在冒泡阶段执行。

如何阻止事件冒泡

```
// w3c
e.stopPropagation()

// IE
e.cancelBubble = true
```

如何阻止事件默认行为

```
//谷歌及IE8以上
e.preventDefault();

//IE8及以下
window.event.returnValue = false;

//无兼容问题（但不能用于节点直接onclick绑定函数）
return false;
```

事件代理/事件委托 以及 优缺点

> 事件委托本质上是利用了浏览器事件冒泡的机制。因为事件在冒泡过程中会上传到父节点，并且父节点可以通过事件对象获取到目标节点，因此可以把子节点的监听函数定义在父节点上，由父节点的监听函数统一处理多个子元素的事件，这种方式称为事件代理。
>
> 使用事件代理我们可以不必要为每一个子元素都绑定一个监听事件，这样减少了内存上的消耗。并且使用事件代理我们还可以实现事件的动态绑定，比如说新增了一个子节点，我们并不需要单独地为它添加一个监听事件，它所发生的事件会交给父元素中的监听函数来处理。

事件委托的优点：

1. 减少内存消耗，不必为大量元素绑定事件
2. 为动态添加的元素绑定事件

事件委托的缺点:

1. 部分事件如 focus、blur 等无冒泡机制，所以无法委托。
2. 事件委托有对子元素的查找过程，委托层级过深，可能会有性能问题
3. 频繁触发的事件如 mousemove、mouseout、mouseover等，不适合事件委托

load 和 DOMContentLoaded 事件的区别

- 当整个页面及所有依赖资源如样式表和图片都已完成加载时，将触发load事件。它与DOMContentLoaded不同，后者只要页面DOM加载完成就触发，无需等待依赖资源的加载。
- 当纯HTML被完全加载以及解析时，DOMContentLoaded 事件会被触发，而不必等待样式表，图片或者子框架完成加载。

js判断图片是否加载完毕的方式

- load事件

> 测试，所有浏览器都显示出了“loaded”，说明所有浏览器都支持img的load事件。

```
<img id="img"
src="https://ss0.bdstatic.com/70cFvHSh_Q1YnxGkpoWK1HF6hhy/it/u=151472226,3497652000&fm=26&gp=0.jpg">
<p id="p">loading...</p>
<script>
    document.getElementById('img').onload = function() {
        document.getElementById('p').innerHTML = 'loaded';
    }
</script>
```

- readystatechange事件

> readyState为complete和loaded则表明图片已经加载完毕。测试IE6-IE10支持该事件，其它浏览器不支持。

```
<img id="img"
src="https://ss0.bdstatic.com/70cFvHSh_Q1YnxGkpoWK1HF6hhy/it/u=151472226,3497652000&fm=26&gp=0.jpg">
<p id="p">loading...</p>
<script>
    var img = document.getElementById('img');
    img.onreadystatechange = function () {
        if (img.readyState == 'complete' || img.readyState == 'loaded') {
            document.getElementById('p').innerHTML = 'readystatechange:loaded';
        }
    }
</script>
```

- img 的 complete属性

> 轮询不断监测img的complete属性，如果为true则表明图片已经加载完毕，停止轮询。该属性所有浏览器都支持。

```
<img id="img"
src="https://ss0.bdstatic.com/70cFvHSh_Q1YnxGkpoWK1HF6hhy/it/u=151472226,3497652000&fm=26&gp=0.jpg">
<p id="p">loading...</p>
<script>

    function imgLoad(img, callback) {
        var timer = setInterval(function() {
            if (img.complete) {
                callback(img);
                clearInterval(timer);
            }
        }, 50)
    }
    imgLoad(img, function() {
        document.getElementById('p').innerHTML = '加载完毕';
    })
</script>
```

js 原型，原型链以及特点

在 `js` 中我们是使用构造函数来新建一个对象的，每一个构造函数的内部都有一个 `prototype` 属性值，这个属性值是一个对象，
这个对象包含了可以由该构造函数的所有实例共享的属性和方法。当我们使用构造函数新建一个对象后，在这个对象的内部将包含一个指针，
这个指针指向构造函数的 `prototype` 属性对应的值，在 `ES5` 中这个指针被称为对象的原型。一般来说我们是不应该能够获取到这个值的，
但是现在浏览器中都实现了 `__proto__` 属性来让我们访问这个属性，但是我们最好不要使用这个属性，因为它不是规范中规定的。
`ES5` 中新增了一个 `Object.getPrototypeOf()` 方法，我们可以通过这个方法来获取对象的原型。

当我们访问一个对象的属性时，如果这个对象内部不存在这个属性，那么它就会去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` 所以这就是我们新建的对象为什么能够使`toString()` 等方法的原因。

特点：

`JavaScript` 对象是通过引用来传递的，我们创建的每个新对象实体中并没有一份属于自己的原型副本。当我们修改原型时，与之相关的对象也会继承这一改变。

instanceof 的作用

> instanceof 运算符用于判断构造函数的 prototype 属性是否出现在对象的原型链中的任何位置。

Object.defineProperty 用法

```
Object.defineProperty(obj, prop, descriptor)
```

- obj
  要定义属性的对象。
- prop
  要定义或修改的属性的名称或 Symbol 。
- descriptor
  要定义或修改的属性描述符。参数有：
  - value
    该属性对应的值。可以是任何有效的 JavaScript 值（数值，对象，函数等）。
    默认为 undefined。
  - writable
    当且仅当该属性的 writable 键值为 true 时，属性的值，也就是上面的 value，才能被赋值运算符改变。
    默认为 false。
  - configurable
    当且仅当该属性的 configurable 键值为 true 时，该属性的描述符才能够被改变，同时该属性也能从对应的对象上被删除。
    默认为 false。
  - enumerable
    当且仅当该属性的 enumerable 键值为 true 时，该属性才会出现在对象的枚举属性中。
    默认为 false。

js 延迟加载的方式有哪些

> js 延迟加载，也就是等页面加载完成之后再加载 JavaScript 文件。 js 延迟加载有助于提高页面加载速度。

一般有以下几种方式：

- defer 属性
- async 属性
- 动态创建 DOM 方式
- 使用 setTimeout 延迟方法
- 让 JS 最后加载

js 脚本 defer 和 async 的区别

- defer 属性表示延迟执行引入的 JavaScript，即这段 JavaScript 加载时 HTML 并未停止解析，这两个过程是并行的。当整个 document 解析完毕后再执行脚本文件，在 DOMContentLoaded 事件触发之前完成。多个脚本按顺序执行。
- async 属性表示异步执行引入的 JavaScript，与 defer 的区别在于，如果已经加载好，就会开始执行，也就是说它的执行仍然会阻塞文档的解析，只是它的加载过程不会阻塞。多个脚本的执行顺序无法保证。

async await

> 官网：async函数返回一个 Promise 对象，可以使用then方法添加回调函数。当函数执行的时候，一旦遇到await就会先返回，等到异步操作完成，再接着执行函数体内后面的语句。

- async单独使用的时候，放在函数前面表示这个函数是一个异步函数，如果async函数有返回结果，必须要用.then()方法来承接（也就是返回的值会被自动处理成promise对象）

```
async function bar() {
  return 'lee'
}

console.log(bar()); // Promise {<resolved>: "lee"}
```

- async await搭配使用的时候，await是等待此函数执行后，再执行下一个，可以把异步函数变成同步来执行，控制函数的执行顺序。await一定要搭配async使用。

> 当await后的函数是返回的promise。

```
let foo = () => {
  return new Promise(resolve => {
    setTimeout(() => {
      console.log('lee');
      resolve();
    }, 1000);
  })
}

async function bar() {
  await foo();
  console.log('van');
}
console.log(bar()); // 隔1秒同时输出 lee van
```

> 当await 后跟的是普通函数（非promise()）

```
let f1 = () => {
  setTimeout(() => {
    console.log('lee');
  }, 1000)
}

let f2 = () => {
  setTimeout(() => {
    console.log('van');
  }, 1000)
}

async function bar() {
  await f1();
  await f2();
  console.log('yeah');
}

console.log(bar()); // yeah 隔1秒同时输出 lee fan
```

Event Loop 事件循环

> 参考链接：详解JavaScript中的Event Loop（事件循环）机制

```
微任务：promise.then(不是promise，promise里是立即执行)   MutationObserver
 process.nextTick(Node.js 环境)
宏任务：script(整体代码)  setTimeout  setInterval   I/O  setImmediate(Node.js 环境)
  UI 交互事件
同一次事件循环中：  微任务永远在宏任务之前执行
```

事件循环的过程：

> 首先script脚本整体是一个大的异步任务，先执行script脚本。这个script脚本会包含同步任务和异步任务，同步任务会先在主线程上执行，异步任务（分为宏任务和微任务）会添加到任务队列中，任务队列分为宏任务队列和微任务队列，宏任务放到宏任务队列，微任务放到微任务队列。
>
> 当同步任务执行完毕后，此时的执行栈已经被清空，会去执行异步任务。此时会先从微任务队列中取一个微任务放到执行栈中执行，若有新的微任务或宏任务产生，添加到相应的任务队列中，循环往复，直至微任务队列清空。
>
> 紧接着会从宏任务队列取一个宏任务放到执行栈中执行，此时可能会产生新的微任务，将微任务放到微任务队列中，当这个宏任务执行完后会继续执行微任务队列，如果没有产生就继续执行下一个宏任务。循环往复，直至所有任务执行完毕。

执行流程：

JS 跨域怎么做

> 什么是跨域？当一个请求url的 协议、域名、端口三者之间任意一个与当前页面url不同即为跨域。
>
> 参考链接:前端常见跨域解决方案（全）

1. JSONP (JSON with Padding)
   通过动态创建 script，再请求一个带参网址实现跨域通信。

   > 参考链接：jsonp的原理与实现

2. CORS (跨域资源共享)
   CORS的基本思想就是使用自定义的HTTP头部让浏览器与服务器进行沟通，从而决定请求或响应是应该成功还是失败。

   普通跨域请求：只需服务端设置 `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。前端设置 `withCredentials` 为true,后端设置 `Access-Control-Allow-Credentials` 为true,同时 `Access-Control-Allow-Origin` 不能设置为 `*`

   目前，所有浏览器都支持该功能(IE8+；IE8/9 需要使用 XDomainRequest 对象来支持 CORS)，CORS 也已经成为主流的跨域解决方案。

3. window.postMessage
   现代浏览器中多窗口通信使用 HTML5 规范的 targetWindow.postMessage(data, origin);其中 data 是需要发送的对象，origin 是目标窗口的 origin。window.addEventListener('message', handler, false);handler 的 event.data 是 postMessage 发送来的数据，event.origin 是发送窗口的 origin，event.source 是发送消息的窗口引用

4. 服务器代理
   内部服务器代理请求跨域 url，然后返回数据

JSONP 怎么实现的

> JSONP 的理念就是，与服务端约定好一个回调函数名，服务端接收到请求后，将返回一段 Javascript，在这段 Javascript 代码中调用了约定好的回调函数，并且将数据作为参数进行传递。当网页接收到这段 Javascript 代码后，就会执行这个回调函数，这时数据已经成功传输到客户端了。

举个例子来说明具体情况：

前端代码

```
<script>
function test(data) {
    console.log(data.name);
}
</script>
<script src="http://127.0.0.1:8088/jsonp?callback=test"></script>
```

后端代码

```
res.end('test({"name": "Monkey"})');
```

以上就实现了JSONP跨域，前端正常打印出了"Monkey"

请求JSONP之前就定义好回调函数test，后端返回的是调用test函数的js代码，浏览器加载这段代码后立即执行

JOSNP 有什么优缺点

缺点：

1. 它只支持 GET 请求而不支持 POST 等其它类型的 HTTP 请求
2. 它只支持跨域 HTTP 请求这种情况，不能解决不同域的两个页面之间如何进行 JavaScript 调用的问题。
3. JSONP 在调用失败的时候不会返回各种 HTTP 状态码。
4. 安全性。假如提供 JSONP 的服务存在页面注入漏洞，即它返回的 JavaScript 的内容被人控制的。那么结果是什么？所有调用这个 JSONP 的网站都会存在漏洞。于是无法把危险控制在一个域名下...

所以在使用 JSONP 的时候必须要保证使用的 JSONP 服务必须是安全可信的。

优点：

1. 它不像 XMLHttpRequest 对象实现的 Ajax 请求那样受到同源策略的限制，JSONP 可以跨越同源策略；
2. 它的兼容性更好，在更加古老的浏览器中都可以运行，不需要 XMLHttpRequest 或 ActiveX 的支持
3. 在请求完毕后可以通过调用 callback 的方式回传结果。

new 运算符的过程

1. 创建一个空对象{}；
2. 构造函数中的 this 指向该空对象
3. 执行构造函数为这个空对象添加属性
4. 判断构造函数有没有返回值，如果返回值是个对象，则返回这个对象；否则返回创建的“空对象”

数组的 push() 和 pop() 方法的返回值是什么

- `push()` 将一个或多个元素添加到数组的末尾，并返回该数组的新长度
- `pop()` 方法从数组中删除最后一个元素，并返回该元素的值

JS 作用域

ES5 只有全局作用域和函数作用域

- 全局作用域：代码在程序的任何地方都能被访问，window 对象的内置属性都存在全局作用域
- 函数作用域：在固定的代码片段才能被访问

ES6 有块级作用域

ES6 新特性

- let const 块级作用域
- 模板字符串
- 解构赋值
- 箭头函数，函数参数默认值
- 扩展运算符（...）
- forEach for...of for...in
- 数组方法map reduce includes
- map和set
- 模块化
- promise proxy
- async
- class

let 和 var 的区别

- let 是块级作用域，var 是函数作用域
- var 存在变量提升，而 let 没有
- 在代码块内，使用 let 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”(TDZ)

```
if (true) {
  // TDZ开始
  tmp = 'abc' // ReferenceError
  console.log(tmp) // ReferenceError

  let tmp // TDZ结束
  console.log(tmp) // undefined

  tmp = 123
  console.log(tmp) // 123
}
```

上面代码中，在 let 命令声明变量 tmp 之前，都属于变量 tmp 的“死区”。

闭包的特性以及优缺点

- 闭包是什么？ 方法里面返回一个方法

```
function test() {
  var age = 18
  function  addAge(){
    age++;
    alert(age)
  }
  return addAge
}
```

闭包有三个特性：

- 函数嵌套函数；
- 内部函数使用外部函数的参数和变量；
- 参数和变量不会被垃圾回收机制回收。

闭包的优点：

- **延长一个 变量的生命周期；**
- **避免全局变量污染，创建一个私有的环境。**

闭包的缺点：

- 常驻内存，增加内存使用量；
- 使用不当造成内存泄漏。

防抖与节流

- 防抖：将多次操作变成一次，防止数据抖动，在操作n秒后，才执行回调；
- 节流：一定时间只调用一次函数

箭头函数与普通函数的区别

1. 箭头函数是匿名函数，不能作为构造函数，不能使用new
2. 箭头函数不能绑定arguments，取而代之用rest参数...解决

```
function A(a){
  console.log(arguments);
}
A(1,2,3,4,5,8);
// [1, 2, 3, 4, 5, 8, callee: ƒ, Symbol(Symbol.iterator): ƒ]
let C = (...c) => {
  console.log(c);
}
C(3,82,32,11323);
// [3, 82, 32, 11323]
```

3. 箭头函数没有原型属性
4. 箭头函数的this永远指向其上下文的this，没有办改变其指向，
   普通函数的this指向调用它的对象
5. 箭头函数不绑定this，会捕获其所在的上下文的this值，作为自己的this值

ES6 中箭头函数 VS 普通函数的 this 指向

```
普通函数中 this

1.  总是代表着它的直接调用者，如 obj.fn，fn 里的最外层 this 就是指向 obj
2.  默认情况下，没有直接调用者，this 指向 window
3.  严格模式下（设置了'use strict'），this 为 undefined
4.  当使用 call，apply，bind（ES5 新增）绑定的，this 指向绑定对象
```

```
ES6 箭头函数中 this

1.  默认指向定义它时，所处上下文的对象的 this 指向；
    即 ES6 箭头函数里 this 的指向就是上下文里对象 this 指向，偶尔没有上下文对象，this 就指向
window
```

下面使用例子来加深一下印象：

```
// 例1
function hello() {
  console.log(this) // window
}
hello()

// 例2
function hello() {
  'use strict'
  console.log(this) // undefined
}
hello()

// 例3
const obj = {
  num: 10,
  hello: function () {
    console.log(this) // obj
    setTimeout(function () {
      console.log(this) // window
    })
```

```
  },
}
obj.hello()

// 例4
const obj = {
  num: 10,
  hello: function () {
    console.log(this) // obj
    setTimeout(() => {
      console.log(this) // obj
    })
  },
}
obj.hello()

// 例5
/*
diameter是普通函数，里面的this指向直接调用它的对象obj。
perimeter是箭头函数，this应该指向上下文函数this的指向，
这里上下文没有函数对象，就默认为window，而window里面没有radius这个属性，就返回为NaN。
*/
const obj = {
  radius: 10,
  diameter() {
    return this.radius * 2
  },
  perimeter: () => 2 * Math.PI * this.radius,
}
console.log(obj.diameter()) // 20
console.log(obj.perimeter()) // NaN
```

ES6 class 和 ES5 函数的区别

> 本质上，ES6 的类只是 ES5 的构造函数的一层包装

1. 与ES5不同，ES6 类和模块的内部默认就是严格模式，不存在遍历提升

```
new Foo(); // Reference Error
class Foo {}
```

JS 实现对象（都是简单类型的值）的深拷贝，一行代码

```
let newObj = JSON.parse(JSON.stringify(oldObj))
```

JSON.parse(JSON.stringify(obj)) 实现深拷贝需要注意的问题

1. 如果obj里面有时间对象，则JSON.stringify后再JSON.parse的结果，时间将只是字符串的形式,而不是时间对象；
2. 如果obj里有RegExp、Error对象，则序列化的结果将只得到空对象；
3. 如果obj里有函数，undefined，则序列化的结果会把函数或 undefined丢失；
4. 如果obj里有NaN、Infinity和-Infinity，则序列化的结果会变成null

5. JSON.stringify()只能序列化对象的可枚举的自有属性，例如 如果obj中的对象是有构造函数生成的， 则使用JSON.parse(JSON.stringify(obj))深拷贝后，会丢弃对象的constructor；
6. 如果对象中存在循环引用的情况也无法正确实现深拷贝；

Promise 是做什么的，有哪些API

> Promise 是异步编程的一种解决方案

Promise用法

> Promise构造函数接受一个函数作为参数，该函数的两个参数分别是resolve和reject。它们是两个函数，由 JavaScript 引擎提供，不用自己部署。
>
> resolve函数的作用是，将Promise对象的状态从“未完成”变为“成功”（即从 pending 变为 resolved），在异步操作成功时调用，并将异步操作的结果，作为参数传递出去；reject函数的作用是，将Promise对象的状态从“未完成”变为“失败”（即从 pending 变为 rejected），在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去。

```
const promise = new Promise(function(resolve, reject) {
  // ... some code

  if (/* 异步操作成功 */){
    resolve(value);
  } else {
    reject(error);
  }
});
```

Promise.prototype.then()

> Promise 实例具有 then 方法，也就是说，then 方法是定义在原型对象 Promise.prototype 上的。它的作用是为 Promise 实例添加状态改变时的回调函数。前面说过，then 方法的第一个参数是 resolved 状态的回调函数，第二个参数（可选）是 rejected 状态的回调函数。
>
> **then 方法返回的是一个新的 Promise 实例**（注意，不是原来那个 Promise 实例）。因此可以采用链式写法，即 then 方法后面再调用另一个 then 方法。

```
getJSON("/posts.json").then(function(json) {
  return json.post;
}).then(function(post) {
  // ...
});
```

Promise.prototype.catch()

> Promise.prototype.catch()方法是.then(null, rejection)或.then(undefined, rejection)的别名，用于指定发生错误时的回调函数。

```
getJSON('/posts.json').then(function(posts) {
  // ...
}).catch(function(error) {
  // 处理 getJSON 和 前一个回调函数运行时发生的错误
  console.log('发生错误! ', error);
});
```

> 上面代码中，getJSON()方法返回一个 Promise 对象，如果该对象状态变为resolved，则会调用then()方法指定的回调函数；如果异步操作抛出错误，状态就会变为rejected，就会调用catch()方法指定的回调函数，处理这个错误。另外，then()方法指定的回调函数，如果运行中抛出错误，也会被catch()方法捕获。

Promise.all()

> Promise.all()方法用于将多个 Promise 实例，包装成一个新的 Promise 实例。

```
const p = Promise.all([p1, p2, p3]);
```

> 上面代码中，Promise.all()方法接受一个数组作为参数，p1、p2、p3都是 Promise 实例，如果不是，就会先调用下面讲到的Promise.resolve方法，将参数转为 Promise 实例，再进一步处理。另外，Promise.all()方法的参数可以不是数组，但必须具有 Iterator 接口，且返回的每个成员都是 Promise 实例。
>
> p的状态由p1、p2、p3决定，分成两种情况:
>
> （1）只有p1、p2、p3的状态都变成fulfilled，p的状态才会变成fulfilled，此时p1、p2、p3的返回值组成一个数组，传递给p的回调函数。
>
> （2）只要p1、p2、p3之中有一个被rejected，p的状态就变成rejected，此时第一个被reject的实例的返回值，会传递给p的回调函数。

Promise.race()

> Promise.race()方法同样是将多个 Promise 实例，包装成一个新的 Promise 实例。

```
const p = Promise.race([p1, p2, p3]);
```

> 上面代码中，只要p1、p2、p3之中有一个实例率先改变状态，p的状态就跟着改变。那个率先改变的 Promise 实例的返回值，就传递给p的回调函数。

Promise.resolve()

> 有时需要将现有对象转为 Promise 对象，Promise.resolve()方法就起到这个作用。
>
> Promise.resolve()等价于下面的写法。

```
Promise.resolve('foo')
// 等价于
new Promise(resolve => resolve('foo'))
```

Promise.reject()

> Promise.reject(reason)方法也会返回一个新的 Promise 实例，该实例的状态为rejected。

```
const p = Promise.reject('出错了');
// 等同于
const p = new Promise((resolve, reject) => reject('出错了'))

p.then(null, function (s) {
  console.log(s)
});
// 出错了
```

> 上面代码生成一个 Promise 对象的实例p，状态为rejected，回调函数会立即执行。

Promise不兼容怎么解决

用一些第三方的库来解决兼容性问题：

1. babel-polyfill
2. ES6-Promise
3. bluebird

Ajax 基本流程

> 原生 js 代码实现与基于 promise 实现请传送至专栏：面试高频手撕代码题

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。我对 ajax 的理解是，它是一种异步通信的方法，通过直接由 js 脚本向服务器发起 http 通信，然后根据服务器返回的数据，更新网页的相应部分，而不用刷新整个页面的一种方法。

创建一个 ajax 有这样几个步骤:

- 首先是创建一个 XMLHttpRequest 对象。
- 然后在这个对象上使用 open 方法创建一个 http 请求，open 方法所需要的参数是请求的方法、请求的地址、是否异步和用户的认证信息。
- 在发起请求前，我们可以为这个对象添加一些信息和监听函数。比如说我们可以通过 setRequestHeader 方法来为请求添加头信息。我们还可以为这个对象添加一个状态监听函数。一个 XMLHttpRequest 对象一共有 5 个状态，当它的状态变化时会触发 onreadystatechange 事件，我们可以通过设置监听函数，来处理请求成功后的结果。当对象的 readyState 变为 4 的时候，代表服务器返回的数据接收完成，这个时候我们可以通过判断请求的状态，如果状态是 2xx 或者 304 的话则代表返回正常。这个时候我们就可以通过 response 中的数据来对页面进行更新了。
- 当对象的属性和监听函数设置完成后，最后我们调用 sent 方法来向服务器发起请求，可以传入参数作为发送的数据体。

Ajax 的 readyState 的几种状态分别代表什么

状态值	含义
0	请求未初始化
1	服务器连接已建立
2	请求已接收
3	请求处理中
4	请求已完成，且响应已就绪

Ajax 禁用浏览器的缓存功能

> 项目中，一般提交请求都会通过 ajax 来提交，我们都知道 ajax 能提高页面载入的速度主要的原因是通过 ajax 减少了重复数据的载入，也就是说在载入数据的同时将数据缓存到内存中，一旦数据被加载其中，只要我们没有刷新页面，这些数据就会一直被缓存在内存中，当我们提交 的 URL 与历史的 URL 一致时，就不需要提交给服务器，也就是不需要从服务器上面去获取数据，虽然这样降低了服务器的负载提高了用户的体验，但是我们不能获取最新的数据。为了保证我们读取的信息都是最新的，我们就需要禁止他的缓存功能。

解决的方法有：

1. 在 ajax 发送请求前加上 `xhr.setRequestHeader("If-Modified-Since","0")`。
2. 在 ajax 发送请求前加上 `xhr.setRequestHeader("Cache-Control","no-cache")`。
3. 在 URL 后面加上一个随机数： `"fresh=" + Math.random()`;。
4. 在 URL 后面加上时间搓： `"nowtime=" + new Date().getTime()`;。
5. 如果是使用 jQuery，直接这样就可以了 `$.ajaxSetup({cache:false})`。这样页面的所有 ajax 都会执行这条语句就是不需要保存缓存记录。

谈谈对前端工程化的理解

> 前端工程化里的工程是一个很大的概念，甚至创建一个git仓库，也可以理解为创建了一个工程，软件工程的定义是运用计算机科学的理论和技术，以及工程管理的原则和方法，按进度和预算，实现满足用户要求的软件产品的定义，开发和维护的工程以及研究的学科。

前端工程化是为了让前端可以自成体系，具体可以从四方面去讨论，模块化，组件化，规范化和自动化。

模块化

> 模块化：将大的文件拆分成互相依赖的小文件，再进行统一的拼装和加载。
>
> js的模块化：利用webpack+babel的模式将所有模块系统进行打包，同步加载，也可以搭乘多个chunk异步加载。
>
> css模块化：之前的sass less 等预处理器虽然实现了css的拆分，但是并没有解决模块化很重要的一个问题，即选择器的全局污染问题。有三种解决办法，第一种是利用webcomponents的技术实现，这个技术虽然解决了全局污染问题，但是由于兼容性问题，目前用的不多，第二种是css in js 将css的技术全部摒，利用js或者json格式去加载css，这种方式简单粗暴，并且不容易处理伪类选择器的问题，被大众所认可的是第三种解决方案，即 css modules ，所有的css文件由js来管理，这种技术最大程度利用了css的生态和模块化的原则，其中vue中的scoped 就是这种技术的提现。利用浏览器的script标签，type类型选modules类型即可。
>
> 资源的模块化：webpack的成功不仅仅是因为将js系统进行模块化处理，而是它的模块化原理，即任何资源都可以模块
> 化且应该模块化处理，优点有以下三点，1：目录结构清晰化，2：资源处理集成化，3：项目依赖单一化。

组件化

> 组件化：将UI页面拆分正有模板+样式+逻辑组成的功能单元，称为组件，组件化不等于模块化，模块化是在资源和代码方面对文件的拆分，而组件化是在UI层面进行的拆分。传统前端框架的思想是以dom优先，先操作dom，再写出可复用的逻辑单元来操作dom，而组件化框架是组件优先，将dom和与之一起的逻辑组成一个组件，再进行引用。我们封装了组件后，还需要对组件间的关系进行判定，例如继承，扩展，嵌套，包含等，这些关系统称为依赖

规范化

> 规范化：规范化是前端工程化很重要的一部分，项目前期规范制定的好坏，直接决定后期的开发质量，分为项目目录规范化，编码规范化，前后端接口规范化，git分支管理，commit描述规范，组件管理等编码规范化分为htmlcss js img 命名规范这几类 接口规范，目的是规则先行，以减少联调中不必要的问题和麻烦，自责划分 前端，渲染逻辑和交互逻辑，后台，处理业务逻辑，各种格式的规定，例如 json尽量简洁轻量，日期尽量字符串，等等。

自动化

> 自动化：让简单重复的工作交给机器完成，例如自动化测试，自动化部署，自动化构建，持续继承等。

js 的几种模块规范

js 中现在比较成熟的有四种模块加载方案。

第一种是 CommonJS 方案，它通过 require 来引入模块，通过 module.exports 定义模块的输出接口。这种模块加载方案是
服务器端的解决方案，它是以同步的方式来引入模块的，因为在服务端文件都存储在本地磁盘，所以读取非常快，所以以同步的方式
加载没有问题。但如果是在浏览器端，由于模块的加载是使用网络请求，因此使用异步加载的方式更加合适。

第二种是 AMD 方案，这种方案采用异步加载的方式来加载模块，模块的加载不影响后面语句的执行，所有依赖这个模块的语句都定
义在一个回调函数里，等到加载完成后再执行回调函数。require.js 实现了 AMD 规范。

第三种是 CMD 方案，这种方案和 AMD 方案都是为了解决异步模块加载的问题，sea.js 实现了 CMD 规范。它和 require.js
的区别在于模块定义时对依赖的处理不同和对依赖模块的执行时机的处理不同。

第四种方案是 ES6 提出的方案，使用 import 和 export 的形式来导入导出模块。这种方案和上面三种方案都不同。

ES6 模块与 CommonJS 模块、AMD、CMD 的差异

> CommonJS 模块输出的是一个值的拷贝，ES6 模块输出的是值的引用。CommonJS 模块输出的是值的拷贝，也就是说，一旦输出一个值，模块内部的变化就影响不到这个值。ES6 模块的运行机制与 CommonJS 不一样。JS 引擎对脚本静态分析的时候，遇到模块加载命令 import，就会生成一个只读引用。等到脚本真正执行时，再根据这个只读引用，到被加载的那个模块里面去取值。
>
> CommonJS 模块是运行时加载，ES6 模块是编译时输出接口。CommonJS 模块就是对象，即在输入时是先加载整个模块，生成一个对象，然后再从这个对象上面读取方法，这种加载称为“运行时加载”。而 ES6 模块不是对象，它的对外接口只是一种静态定义，在代码静态解析阶段就会生成。

webpack 的功能

webpack 的主要原理：
它将所有的资源都看成是一个模块，并且把页面逻辑当作一个整体，通过一个给定的入口文件，webpack 从这个文件开始，找到所有的依赖文件，
将各个依赖文件模块通过 loader 和 plugins 处理后，然后打包在一起，最后输出一个浏览器可识别的 JS 文件。

Webpack 具有四个核心的概念，分别是 Entry（入口）、Output（输出）、loader（加载器） 和 Plugins（插件）。

Entry 是 webpack 的入口起点，它指示 webpack 应该从哪个模块开始着手，来作为其构建内部依赖图的开始。

Output 属性告诉 webpack 在哪里输出它所创建的打包文件，也可指定打包文件的名称，默认位置为 ./dist。

loader 可以理解为 webpack 的编译器，它使得 webpack 可以处理一些非 JavaScript 文件。在对 loader 进行配置的时候：
  1. test 属性，标志有哪些后缀的文件应该被处理，是一个正则表达式。
  2. use 属性，指定 test 类型的文件应该使用哪个 loader 进行预处理。
常用的 loader 有 css-loader、style-loader 等。

Plugins（插件）可以用于执行范围更广的任务，包括打包、优化、压缩、搭建服务器等等，要使用一个插件，
一般是先使用 npm 包管理器进行安装，然后在配置文件中引入，最后将其实例化后传递给 plugins 数组属性。

webpack 常用插件

- html-webpack-plugin

> 用于生成一个html文件，并将最终生成的js，css以及一些静态资源文件以script和link的形式动态插入其中

- webpack-dev-server

> 用于实时的打包编译
> 打包好的 main.js 是托管到了内存中，所以在项目根目录中看不到，但是我们可以认为在项目的根目录中，有一个看不见的 main.js

- CommonsChunkPlugin

> 主要是用来提取第三方库（如jQuery）和公共模块(公共js，css都可以)，常用于多页面应用程序，生成公共 chunk，避免重复引用。

arguments怎么转化成真数组

- arguments是一个伪数组，是当前函数的内置对象，存储所有的形参，有length属性，但是不能用数组的方法。
- [...arguments] 扩展运算符的方式，拿取剩余参数
- Array.prototype.slice.call(arguments);使用call 一个对象调用另一个函数的方法，slice切割数组并返回一个新的数组
- [].slice.call() 因为[].slice === Array.prototype.slice
- 遍历：arguments有length属性，所以，可以遍历arguments取出每一个元素，并放进新的数组中

js的对象的常用的方法

```
Object.assign() //复制对象，创建一个新的对象
Object.entries() //返回自身可枚举的[key,value]
Object.keys()
Object.values()
Object.hasOwnProperty(key)//是否有这个属性 true/false
Object.getOwnPropertyNames() //取得对象自身可枚举的属性名
//for in 对对象进行遍历，可以拿到自身以及原型链上的可枚举的属性
Object.freeze()//冻结一个对象，不可修改，不可删除。不可添加新的属性
Object.prototype.toString()// 返回数组[object,object/array/function等]
//判断是数组还是对象就是用的这个方法
Object.defineProerty(obj,attr,descriptor)
//可以对对象属性进行修改添加，删除等的操作
//参数1，要操作的对象
//参数2：要操作的属性名字
//参数3：属性描述符：是否可枚举，是否可读，可写，他的值等
```

js的字符串的常用的方法

```
var str1 = 'wwww'
var str2 = 'jjjj'
var str3 = 'kkkk'

str.concat()//拼接
str.includes()//判断字符串是否包含在另外一个字符串中
str.indexOf()
str.lastIndexOf()
str.split() //按特定的符号分割成字符串数组!
str.toLowerCase() //转换成小写的形式
str.toUpperCase() //转换成大写的形式
str.trim()//去空格
str.substring(start,end) // 截取字串，不含开始，含结束，end可以小于start,会自动将小值
str.slice() //截取字串，含开始，含结束，end不可以小于start
str.substr(start,length)  //截取指定长度的字符串
```

js的数组的常用的方法

```
var arr = [0,1,2,3,4]

arr.push()
arr.pop()
arr.shift()
arr.unshift()
arr.reverse()
arr.every()
arr.some()
arr.forEach()
arr.filter()
arr.includes()
arr.map()
arr.reduce()
arr.indexOf()
arr.lastIndexOf() //索引正序，但是从后往前找
arr.findIndex() //找索引
arr.find()  //找满足条件的元素
arr.join()//默认以逗号隔开
arr.join(' ')//无缝链接 将数组元素拼接成字符串
arr.slice(1,2)//截取数组的一部分，不包含头部，包含尾部，修改原数组
arr.splice(1,4) //从索引1开始删除4个元素,第二个是要删除的长度，第三个往后是要添加的元素
arr.splice(2,0,'i') //从索引2开始，删除0个，加入一个'i'
arr.aplice(3,1,'o','i')//从索引3开始，删除1个，添加两个字符串。
arr.flat() //数组降维 ，返回新数组
arr.flat(1)
arr.flat(Infinity)
arr.entries() //将数组返回一个对象，包含对象索引的键值对
```

MVVM 和 MVC的区别

- MVC: MVC是应用最广泛的软件架构之一,一般MVC分为:Model(模型),View(视图),Controller(控制器)。 这主要是基于分层的目的,让彼此的职责分开.View一般用过Controller来和Model进行联系。

Controller是Model和View的协调者,View和Model不直接联系。基本都是单向联系。

 MVC

1. View传送指令到Controller。
2. Controller完成业务逻辑后改变Model状态。
3. Model将新的数据发送至View,用户得到反馈。

- MVVM: MVVM是把MVC中的Controller改变成了ViewModel。

View的变化会自动更新到ViewModel,ViewModel的变化也会自动同步到View上显示,通过数据来显示视图层。

 MVVM

MVVM和MVC的区别:

- MVC中Controller演变成MVVM中的ViewModel
- MVVM通过数据来显示视图层而不是节点操作
- MVVM主要解决了MVC中大量的dom操作使页面渲染性能降低,加载速度变慢,影响用户体验

vue 的优点

- 轻量级框架
- 简单易学
- 双向数据绑定
- 组件化
- 视图，数据，结构分离
- 虚拟 DOM
- 运行速度更快

vue 的响应式原理

数据发生变化后，会重新对页面渲染，这就是 Vue 响应式

想完成这个过程，我们需要：

- 侦测数据的变化
- 收集视图依赖了哪些数据
- 数据变化时，自动“通知”需要更新的视图部分，并进行更新

对应专业俗语分别是：

数据劫持 / 数据代理
依赖收集
发布订阅模式

vue 双向数据绑定原理

> vue 通过使用双向数据绑定，来实现了 View 和 Model 的同步更新。vue 的双向数据绑定主要是通过使用数据劫持和发布订阅者模式来实现的。
>
> 首先我们通过 Object.defineProperty() 方法来对 Model 数据各个属性添加访问器属性，以此来实现数据的劫持，因此当 Model 中的数据发生变化的时候，我们可以通过配置的 setter 和 getter 方法来实现对 View 层数据更新的通知。
>
> 数据在 html 模板中一共有两种绑定情况，一种是使用 v-model 来对 value 值进行绑定，一种是作为文本绑定，在对模板引擎进行解析的过程中。

> 如果遇到元素节点，并且属性值包含 v-model 的话，我们就从 Model 中去获取 v-model 所对应的属性的值，并赋值给元素的 value 值。然后给这个元素设置一个监听事件，当 View 中元素的数据发生变化的时候触发该事件，通知 Model 中的对应的属性的值进行更新。
>
> 如果遇到了绑定的文本节点，我们使用 Model 中对应的属性的值来替换这个文本。对于文本节点的更新，我们使用了发布订阅者模式，属性作为一个主题，我们为这个节点设置一个订阅者对象，将这个订阅者对象加入这个属性主题的订阅者列表中。当 Model 层数据发生改变的时候，Model 作为发布者向主题发出通知，主题收到通知再向它的所有订阅者推送，订阅者收到通知后更改自己的数据。

Object.defineProperty 介绍

> Object.defineProperty 函数一共有三个参数，第一个参数是需要定义属性的对象，第二个参数是需要定义的属性，第三个是该属性描述符。
>
> 一个属性的描述符有一下属性，分别是
> value 属性的值，
> writable 属性是否可写，
> enumerable 属性是否可枚举，
> configurable 属性是否可配置修改。
> get属性 当访问该属性时，会调用此函数
> set属性 当属性值被修改时，会调用此函数。

使用 Object.defineProperty() 来进行数据劫持有什么缺点

有一些对属性的操作，使用这种方法无法拦截，比如说通过下标方式修改数组数据或者给对象新增属性，vue 内部通过重写函数解决了这个问题。

在 Vue3.0 中已经不使用这种方式了，而是通过使用 Proxy 对对象进行代理，从而实现数据劫持。使用 Proxy 的好处是它可以完美的监听到任何方式的数据改变，唯一的缺点是兼容性的问题，因为这是 ES6 的语法。

v-if 和 v-show 的区别

- v-if：每次都会重新删除或创建元素来控制 DOM 结点的存在与否
- v-show:是切换了元素的样式 display:none，display: block

因而 v-if 有较高的切换性能消耗，v-show 有较高的初始渲染消耗

为什么 vue 组件中的 data 必须是函数

Vue 的 data 是一个闭包的设计，各个组件都有一个私有的作用域。

当一个组件被定义，data 必须声明为返回一个初始数据对象的函数，因为组件可能被用来创建多个实例。如果 data 仍然是一个纯粹的对象，则所有的实例将共享引用同一个数据对象！通过提供 data 函数，每次创建一个新实例后，我们能够调用 data 函数，从而返回初始数据的一个全新副本数据对象。

简而言之，就是 data 中数据可能会被复用，要保证不同组件调用的时候数据是相同的。

vue 的生命周期函数

- beforeCreate:

  > 在实例初始化之后，数据观测 (data observer) 和 event/watcher 事件配置之前被调用。

> 在new一个vue实例后，只有一些默认的生命周期钩子和默认事件，其他的东西都还没创建。在beforeCreate生命周期执行的时候，data和methods中的数据都还没有初始化。不能在这个阶段使用data中的数据和methods中的方法

- created:

> 在实例创建完成后被立即调用。在这一步，实例已完成以下的配置：数据观测 (data observer)，property 和方法的运算，watch/event 事件回调。然而，挂载阶段还没开始，$el property 目前尚不可用。
>
> data 和 methods都已经被初始化好了，如果要调用 methods 中的方法，或者操作 data 中的数据，最早可以在这个阶段中操作

- beforeMount:

> 在挂载开始之前被调用：相关的 render 函数首次被调用。
>
> 执行到这个钩子的时候，在内存中已经编译好了模板了，但是还没有挂载到页面中，此时，页面还是旧的

- mounted:

> 实例被挂载后调用，这时 el 被新创建的 vm.$el 替换了。如果根实例挂载到了一个文档内的元素上，当 mounted 被调用时 vm.$el 也在文档内。
>
> 执行到这个钩子的时候，就表示Vue实例已经初始化完成了。此时组件脱离了创建阶段，进入到了运行阶段。如果我们想要通过插件操作页面上的DOM节点，最早可以在这个阶段中进行

- beforeUpdate:

> 当执行这个钩子时，页面中的显示的数据还是旧的，data中的数据是更新后的， 页面还没有和最新的数据保持同步

- updated:

> 页面显示的数据和data中的数据已经保持同步了，都是最新的

- beforeDestroy:

> Vue实例从运行阶段进入到了销毁阶段，这个时候上所有的 data 和 methods，指令，过滤器……都是处于可用状态，还没有真正被销毁

- destroyed:

> 这个时候上所有的 data 和 methods，指令，过滤器……都是处于不可用状态，组件已经被销毁了。

- activated:

> 被 `keep-alive` 缓存的组件激活时调用。

- deactivated:

> 被 `keep-alive` 缓存的组件停用时调用。

vue 的 activated 和 deactivated 钩子函数

```
<keep-alive>
  <component :is="view"></component>
</keep-alive>
```

`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。

当组件在 `<keep-alive>` 内被切换，它的 `activated` 和 `deactivated` 这两个生命周期钩子函数将会被对应执行。

- `activated`在`keep-alive`组件激活时调用，该钩子函数在服务器端渲染期间不被调用。
- `deactivated`在`keep-alive`组件停用时调用，该钩子函数在服务端渲染期间不被调用。

Vue中父子组件生命周期执行顺序

在单一组件中，钩子的执行顺序是beforeCreate-> created -> mounted->... ->destroyed

父子组件生命周期执行顺序：

- 加载渲染过程

  ```
  父beforeCreate->父created->父beforeMount->子beforeCreate->子created->子beforeMount->子mounted->父mounted
  ```

- 更新过程

  ```
  父beforeUpdate->子beforeUpdate->子updated->父updated
  ```

- 销毁过程

  ```
  父beforeDestroy->子beforeDestroy->子destroyed->父destroyed
  ```

- 常用钩子简易版

  ```
  父create->子created->子mounted->父mounted
  ```

nextTick 用法

官网解释：

> 将回调延迟到下次 DOM 更新循环之后执行。在修改数据之后立即使用它，然后等待 DOM 更新。

```
<div class="app">
  <div ref="msgDiv">{{msg}}</div>
  <div v-if="msg1">Message got outside $nextTick: {{msg1}}</div>
  <div v-if="msg2">Message got inside $nextTick: {{msg2}}</div>
  <div v-if="msg3">Message got outside $nextTick: {{msg3}}</div>
  <button @click="changeMsg">
    Change the Message
  </button>
</div>
```

```
new Vue({
  el: '.app',
  data: {
```

```
    msg: 'Hello Vue.',
    msg1: '',
    msg2: '',
    msg3: ''
  },
  methods: {
    changeMsg() {
      this.msg = "Hello world."
      this.msg1 = this.$refs.msgDiv.innerHTML
      this.$nextTick(() => {
        this.msg2 = this.$refs.msgDiv.innerHTML
      })
      this.msg3 = this.$refs.msgDiv.innerHTML
    }
  }
})
```

vue中key属性的作用

一句话 key 的作用主要是为了高效的更新虚拟 DOM

key 的特殊 attribute 主要用在 Vue 的虚拟 DOM 算法，在新旧 nodes 对比时辨识 VNodes。如果不使用 key，Vue 会使用一种最大限度减少动态元素并且尽可能的尝试就地修改/复用相同类型元素的算法。而使用 key 时，它会基于 key 的变化重新排列元素顺序，并且会移除 key 不存在的元素。

有相同父元素的子元素必须有独特的 key。重复的 key 会造成渲染错误。

Vue中key属性用index为什么不行

这是由于diff算法的机制所决定的，话不多说，直接上反例：

当我们选中某一个（比如第3个），再添加或删除内容的时候就能发现bug了

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>

</head>
<body>
    <div id="app">
        <span>ID:</span><input type="text" v-model="id">
        <span>Name:</span><input type="text" v-model="name">
        <button @click="handleClick">添加</button>

        <div v-for="(item, index) in list" :key="index">
            <input type="checkbox" />
            <span @click="handleDelete(index)">{{item.id}} --- {{item.name}}
</span>
        </div>
    </div>
    <script>
        let vm = new Vue({
            el: '#app',
```

```
            data: {
                id: '',
                name: '',
                list: [
                    {id: 1, name: '张三'},
                    {id: 2, name: '李四'},
                    {id: 3, name: '王五'},
                    {id: 4, name: '赵六'},
                ]
            },
            methods: {
                handleClick() {
                    this.list.unshift({
                        id: this.id,
                        name: this.name
                    })
                },
                handleDelete(index) {
                    this.list.splice(index, 1)
                }
            },
        })
    </script>
</body>
</html>
```

Vue的路由模式

> hash模式 与 history模式

- hash（即地址栏 URL 中的 # 符号)。

比如这个 URL：www.123.com/#/test，hash 的值为 #/test。

特点： hash 虽然出现在 URL 中，但不会被包括在 HTTP，因为我们hash每次页面切换其实切换的是#之后的内容，而#后内容的改变并不会触发地址的改变，
所以不存在向后台发出请求，对后端完全没有影响，因此改变 hash 不会重新加载页面。

每次hash发生变化时都会调用 onhashchange事件

优点：可以随意刷新

- history（利用了浏览器的历史记录栈）

特点：利用了 HTML5 History Interface 中新增的 pushState() 和 replaceState() 方法。（需要特定浏览器支持）

在当前已有的 back、forward、go的基础之上，它们提供了对历史记录进行修改的功能。只是当它们执行修改时，虽然改变了当前的URL，但浏览器不会立即向后端发送请求。

history：可以通过前进 后退控制页面的跳转，刷新是真是的改变url。

缺点：不能刷新，需要后端进行配置。由于history模式下是可以自由修改请求url，当刷新时如果不对对应地址进行匹配就会返回404。
但是在hash模式下是可以刷新的，前端路由修改的是#中的信息，请求时地址是不会变的

vue中$router和$route的区别

- this.$route：当前激活的路由的信息对象。每个对象都是局部的，可以获取当前路由的 path, name, params, query 等属性。
- this.$router：全局的 router 实例。通过 vue 根实例中注入 router 实例，然后再注入到每个子组件，从而让整个应用都有路由功能。其中包含了很多属性和对象（比如 history 对象），任何页面也都可以调用其 push(), replace(), go() 等方法。

Vue diff算法详解

- updateChildren

> 这个函数是用来比较两个结点的子节点

```
updateChildren(parentElm, oldCh, newCh) {
    let oldStartIdx = 0,
        newStartIdx = 0
    let oldEndIdx = oldCh.length - 1
    let oldStartVnode = oldCh[0]
    let oldEndVnode = oldCh[oldEndIdx]
    let newEndIdx = newCh.length - 1
    let newStartVnode = newCh[0]
    let newEndVnode = newCh[newEndIdx]
    let oldKeyToIdx
    let idxInOld
    let elmToMove
    let before
    while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) { // 只有
oldS>oldE 或者 newS>newE 才会终止循环
        if (oldStartVnode == null) { // 对于vnode.key的比较，会把oldVnode = null
            oldStartVnode = oldCh[++oldStartIdx]
        } else if (oldEndVnode == null) {
            oldEndVnode = oldCh[--oldEndIdx]
        } else if (newStartVnode == null) {
            newStartVnode = newCh[++newStartIdx]
        } else if (newEndVnode == null) { // 到这里是找到第一个不为null的
oldStartVnode oldEndVnode newStartVnode newEndVnode
            newEndVnode = newCh[--newEndIdx]
        } else if (sameVnode(oldStartVnode, newStartVnode)) { // oldS指针和newS指
针对应的结点相同时，将oldS和newS指针同时向后移一位
            patchVnode(oldStartVnode, newStartVnode)
            oldStartVnode = oldCh[++oldStartIdx]
            newStartVnode = newCh[++newStartIdx]
        } else if (sameVnode(oldEndVnode, newEndVnode)) { // oldE指针和newE指针对应
的结点相同时，将oldE和newE指针同时向前移一位
            patchVnode(oldEndVnode, newEndVnode)
            oldEndVnode = oldCh[--oldEndIdx]
            newEndVnode = newCh[--newEndIdx]
        } else if (sameVnode(oldStartVnode, newEndVnode)) { // oldS指针和newE指针对
应的结点相同时，将oldS指针对应结点移动到oldE指针之后，同时将oldS指针向后移动一位，newE指针向前
移动一位
            patchVnode(oldStartVnode, newEndVnode)
            api.insertBefore(parentElm, oldStartVnode.el,
api.nextSibling(oldEndVnode.el))
            oldStartVnode = oldCh[++oldStartIdx]
            newEndVnode = newCh[--newEndIdx]
```

```
        } else if (sameVnode(oldEndVnode, newStartVnode)) { // oldE指针和newS指针对应的结点相同时，将oldE指针对应的结点移动到oldS指针之前，同时将oldE指针向前移动一位，newS指针想后移动一位
            patchVnode(oldEndVnode, newStartVnode)
            api.insertBefore(parentElm, oldEndVnode.el, oldStartVnode.el)
            oldEndVnode = oldCh[--oldEndIdx]
            newStartVnode = newCh[++newStartIdx]
        } else { // 使用key时的比较
            if (oldKeyToIdx === undefined) {
                oldKeyToIdx = createKeyToOldIdx(oldCh, oldStartIdx, oldEndIdx) // 有key生成index表
            }
            idxInOld = oldKeyToIdx[newStartVnode.key]
            if (!idxInOld) {
                api.insertBefore(parentElm, createEle(newStartVnode).el, oldStartVnode.el)
                newStartVnode = newCh[++newStartIdx]
            } else {
                elmToMove = oldCh[idxInOld]
                if (elmToMove.sel !== newStartVnode.sel) {
                    api.insertBefore(parentElm, createEle(newStartVnode).el, oldStartVnode.el)
                } else {
                    patchVnode(elmToMove, newStartVnode)
                    oldCh[idxInOld] = null
                    api.insertBefore(parentElm, elmToMove.el, oldStartVnode.el)
                }
                newStartVnode = newCh[++newStartIdx]
            }
        }
    }
    if (oldStartIdx > oldEndIdx) { // oldVnode遍历结束了，那就将newVnode里newS指针和newE指针之间的结点添加到oldVnode里
        before = newCh[newEndIdx + 1] == null ? null : newCh[newEndIdx + 1].el
        addVnodes(parentElm, before, newCh, newStartIdx, newEndIdx)
    } else if (newStartIdx > newEndIdx) { // newVnode遍历结束了，那就将oldVnonde里oldS指针和oldE指针之间的结点删除
        removeVnodes(parentElm, oldCh, oldStartIdx, oldEndIdx)
    }
}
```

移动端适配的方法

> 起因:手机设备屏幕尺寸不一，做移动端的Web页面，需要考虑安卓/IOS的各种尺寸设备上的兼容，针对移动端设备的页面，设计与前端实现怎样做能更好地适配不同屏幕宽度的移动设备；

1. flex 弹性布局
2. viewport 适配

```
<meta name="viewport" content="width=750,initial-scale=0.5">
```

   initial-scale = 屏幕的宽度 / 设计稿的宽度

3. rem 弹性布局
4. rem + viewport 缩放

> 这也是淘宝使用的方案，根据屏幕宽度设定 rem 值，需要适配的元素都使用 rem 为单位，不需要适配的元素还是使用 px 为单位。（1em = 16px）

rem 原理

rem 布局的本质是等比缩放

rem 是（根）字体大小相对单位，也就是说跟当前元素的 font-size 没有关系，而是跟整个 body 的 font-size 有关系。

rem 和 em 的区别

> 一句话概括：em相对于父元素，rem相对于根元素。

- 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
  ```

- rem

  ```
  rem 是全部的长度都相对于根元素，根元素是谁？<html>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。
  ```

移动端 300ms 延迟的原因以及解决方案

移动端 300ms 点击延迟和点击穿透

> 移动端点击有 300ms 的延迟是因为移动端会有双击缩放的这个操作，因此浏览器在 click 之后要等待 300ms，看用户有没有下一次点击，来判断这次操作是不是双击。

有三种办法来解决这个问题：

1. 通过 meta 标签禁用网页的缩放。

   ```
   <meta name="viewport" content="user-scalable=no">
   ```

2. 更改默认的视口宽度

   ```
   <meta name="viewport" content="width=device-width">
   ```

3. 调用一些 js 库，比如 FastClick

   > FastClick 是 FT Labs 专门为解决移动端浏览器 300 毫秒点击延迟问题所开发的一个轻量级的库。FastClick 的实现原理是在检测到 touchend 事件的时候，会通过 DOM 自定义事件立即出发模拟一个 click 事件，并把浏览器在 300ms 之后的 click 事件阻止掉。

Vue 和 React 数据驱动的区别

在数据绑定上来说，vue的特色是双向数据绑定，而在react中是单向数据绑定。

vue中实现数据绑定靠的是数据劫持（Object.defineProperty()）+发布-订阅模式

vue中实现双向绑定

```
<input v-model="msg" />
```

react中实现双向绑定

```
<input value={this.state.msg} onChange={() => this.handleInputChange()} />
```

OSI七层与TCP/IP五层模型

- OSI七层模型

  ```
  应用层
  表示层
  会话层
  传输层
  网络层
  数据链路层
  物理层
  ```

- TCP/IP五层模型

  ```
  应用层：TFTP，HTTP，SNMP，FTP，SMTP，DNS，Telnet
  传输层：TCP，UDP
  网络层：IP，ICMP，RIP，OSPF，BGP，IGMP
  数据链路层：SLIP，CSLIP，PPP，ARP，RARP，MTU
  物理层
  ```

应用层的协议哪些是基于TCP协议的，哪些是基于UDP协议的

基于TCP协议的

- FTP（文件传输协议）：定义了文件传输协议，使用21端口。
- TELNET（远程登陆协议）：一种用于远程登陆的端口，使用23端口，用户可以以自己的身份远程连接到计算机上，可提供基于DOS模式下的通信服务。
- SMTP（简单邮件传输协议）：邮件传送协议，用于发送邮件。服务器开放的是25号端口。
- POP3（邮件读取协议）：它是和SMTP对应，POP3用于接收邮件。POP3协议所用的是110端口。
- HTTP（超文本传输协议）：是从Web服务器传输超文本到本地浏览器的传送协议。
- HTTPS（超文本传输安全协议）

基于UDP协议的

- TFTP（简单文件传输协议）：该协议在熟知端口69上使用UDP服务。
- SNMP（简单网络管理协议）：使用161号端口，是用来管理网络设备的。由于网络设备很多，无连接的服务就体现出其优势。
- BOOTP（引导程序协议，DHCP的前身）：应用于无盘设备
- DHCP（动态主机配置协议）：是一个局域网的网络协议
- RIP（路由信息协议）：基于距离矢量算法的路由协议，利用跳数来作为计量标准。
- IGMP（Internet组管理协议）

基于TCP和UDP协议的

- DNS（域名系统）：DNS区域传输的时候使用TCP协议。域名解析时使用UDP协议。DNS用的是53号端口。
- ECHO（回绕协议）

HTTP 状态码

1. 1XX 信息性状态码
    - 100 继续
    - 101 切换协议
2. 2XX 成功状态码
    - 200 OK 成功处理了请求
    - 204 No Content 请求处理成功，但没有资源可返回
    - 206 Partial Content 请求资源的某一部分
3. 3XX 重定向状态码
    - 301 永久性重定向，表示请求的资源已被分配了新的 URI
    - 302 临时性重定向，资源的 URL 已临时定位到其他位置
    - 303 告诉客户端应该用另一个 URL 获取资源
    - 304 表示客户端发送附带条件的请求时，服务器端允许请求访问资源，但未满足条件的情况
4. 4XX 客户端错误状态码
    - 400 表示请求报文中存在语法错误
    - 401 未授权
    - 403 服务器拒绝了请求
    - 404 服务器无法找到所请求的 URL
5. 5XX 服务器错误状态码
    - 500 内部服务器错误
    - 502 错误网关
    - 503 服务器暂时处于超负载或正在进行停机维护，现在无法处理请求。
    - 504 响应超时

HTTP 与 HTTPS 的区别

1. HTTP 传输的数据都是未加密的，也就是明文的，HTTPS 协议是由 HTTP 和 SSL 协议构建的可进行加密传输和身份认证的网络协议，比 HTTP 协议的安全性更高。
2. HTTPS 协议需要 CA 证书，费用较高；
3. 使用不同的链接方式，端口也不同，一般而言，HTTP 协议的端口为 80，HTTPS 的端口为 443；

HTTPS 协议的工作原理

1. 客户使用 HTTPS URL 访问服务器，则要求 web 服务器建立 SSL 链接。
2. web 服务器接收到客户端的请求之后，会将网站的证书（证书中包含了公钥），返回给客户端。
3. 客户端和 web 服务器端开始协商 SSL 链接的安全等级，也就是加密等级。
4. 客户端浏览器通过双方协商一致的安全等级，建立会话密钥，然后通过网站的公钥来加密会话密钥，并传送给网站。
5. web 服务器通过自己的私钥解密出会话密钥。
6. web 服务器通过会话密钥加密与客户端之间进行通信。

HTTP/2.0 特性

> 参考链接：[HTTP/2 相比 1.0 有哪些重大改进?]

1. 首部压缩
2. 多路复用
3. 二进制分帧
4. 服务端推送

TCP 和 UDP 之间的区别

TCP：传输控制协议 UDP：用户数据报协议

1. TCP 是面向连接的，UDP 是无连接的即发送数据前不需要先建立链接；
2. TCP 提供可靠的服务。也就是说，通过 TCP 连接传送的数据，无差错，不丢失，不重复，且按序到达；UDP 尽最大努力交付，即不保证可靠交付。
3. TCP 是面向字节流，UDP 面向报文；
4. TCP 只能是 1 对 1 的，UDP 支持 1 对 1,1 对多；
5. TCP 的首部较大为 20 字节，而 UDP 只有 8 字节；

TCP 的三次握手和四次挥手

对称加密和非对称加密的区别

WebSocket 协议

> 参考链接：HTML5 WebSocket

WebSocket 是 HTML5 开始提供的一种在单个 TCP 连接上进行全双工通讯的协议。

WebSocket 使得客户端和服务器之间的数据交换变得更加简单，允许服务端主动向客户端推送数据。在 WebSocket API 中，浏览器和服务器只需要完成一次握手，两者之间就直接可以创建持久性的连接，并进行双向数据传输。

现在，很多网站为了实现推送技术，所用的技术都是 Ajax 轮询。轮询是在特定的的时间间隔（如每 1 秒），由浏览器对服务器发出 HTTP 请求，然后由服务器返回最新的数据给客户端的浏览器。这种传统的模式带来很明显的缺点，即浏览器需要不断的向服务器发出请求，然而 HTTP 请求可能包含较长的头部，其中真正有效的数据可能只是很小的一部分，显然这样会浪费很多的带宽等资源。


websocket和ajax轮询的对比图

应用场景:实现即时通讯:如股票交易行情分析、聊天室、在线游戏等，替代轮询和长轮询

什么是浏览器的同源政策

> 我对浏览器的同源政策的理解是，一个域下的 `js` 脚本在未经允许的情况下，不能够访问另一个域的内容。这里的同源的指的是两个
> 域的协议、域名、端口号必须相同，否则则不属于同一个域。
>
> 同源政策主要限制了三个方面
>
> 第一个是当前域下的 `js` 脚本不能够访问其他域下的 `cookie`、`localStorage` 和 `indexDB`。
>
> 第二个是当前域下的 `js` 脚本不能够操作访问其他域下的 `DOM`。
>
> 第三个是当前域下 `ajax` 无法发送跨域请求。
>
> 同源政策的目的主要是为了保证用户的信息安全，它只是对 `js` 脚本的一种限制，并不是对浏览器的限制，对于一般的 `img`、或者
> `script` 脚本请求都不会有跨域的限制，这是因为这些操作都不会通过响应结果来进行可能出现安全问题的操作。

HTTP 请求的方式

1. GET：请求指定的页面信息，并返回实体主体。
2. HEAD：类似于 GET 请求，只不过返回的响应中没有具体的内容，用于获取报头
3. POST：向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST 请求可能会导致新的资源的建立和/或已有资源的修改。
4. PUT：从客户端向服务器传送的数据取代指定的文档的内容。
5. DELETE：请求服务器删除指定的页面。
6. CONNECT：HTTP/1.1 协议中预留给能够将连接改为管道方式的代理服务器。
7. OPTIONS：允许客户端查看服务器的性能。
8. TRACE：回显服务器收到的请求，主要用于测试或诊断。

GET 和 POST 的区别

两者本质上都是 TCP 链接

1. get 参数通过 url 传递，post 放在请求体 (request body) 中。
2. get 请求在 url 中传递的参数是有长度限制的，而 post 没有。
3. get 比 post 更不安全，因为参数直接暴露在 url 中，所以不能用来传递敏感信息。
4. get 请求只能进行 url 编码，而 post 支持多种编码方式。
5. get 请求参数会被完整保留在浏览历史记录里，而 post 中的参数不会被保留。
6. get 产生一个 TCP 数据包；post 产生两个 TCP 数据包。
   对于 get 方式的请求，浏览器会把 http header 和 data 一并发送出去，服务器响应 200（返回数据）；
   而对于 post，浏览器先发送 header，服务器响应 100 continue，浏览器再发送 data，服务器响应 200 ok（返回数据）。

浏览器输入 URL 之后发生了什么

> 参考链接：[在浏览器输入 URL 回车之后发生了什么（超详细版）]

1. DNS 解析
2. TCP 连接
3. 发送 HTTP 请求
4. 服务器处理请求并返回 HTTP 报文
5. 浏览器解析渲染页面
6. 连接结束

DNS 的具体过程

1. 输入 IP，此时电脑发送一个 DNS 请求到本地 DNS 服务器（一般是网络接入服务商提供 eg:电信，移动）
2. 本地 DNS 服务器会首先查询它的缓存记录，若有，则直接返回结果，若没有，本地 DNS 服务器还要向 DNS 根服务器进行查询；
3. DNS 根服务器没有记录具体域名和 IP 地址的对应关系，而是告诉本地 DNS 服务器，可到域服务器上继续查询，并给出域服务器地址
4. 本地服务器继续向域服务器发出请求，返回域名的解析服务器地址
5. 本地 DNS 向域名解析服务器发出请求，收到域名与 IP 地址对应关系
6. 本地 DNS 服务器将 IP 地址返回电脑，且保存副本到缓存已备下次查询

Cookie 和 WebStorage(SessionStorage 和 LocalStorage)的区别

1. 都会在浏览器端保存，有大小限制，同源限制
2. cookie 会在请求时发送到服务器，作为会话标识，服务器可修改 cookie；web storage 不会发送到服务器
3. cookie 有 path 概念，子路径可以访问父路径 cookie，父路径不能访问子路径 cookie
4. 有效期：cookie 在设置的有效期内有效，默认为浏览器关闭；sessionStorage 在窗口关闭前有效；localStorage 长期有效，直到用户删除
5. 作用域不同 sessionStorage：不在不同的浏览器窗口中共享，即使是同一个页面；localStorage：在所有同源窗口都是共享的；cookie：也是在所有同源窗口中共享的
6. 存储大小不同：cookie 数据不能超过 4K；webStorage 虽然也有存储大小的限制，但是比 cookie 大得多，可以达到 5M 或更大

cookie 和 session 的区别

1.存储位置不同：

> cookie 数据存放在客户的浏览器上
>
> session 数据放在服务器上。

2.存储容量不同：

> 单个 cookie 保存的数据不能超过 4K，一个站点最多保存 20 个 cookie。
>
> 对于 session 来说并没有上限，但出于对服务器端的性能考虑，session 内不要存放过多的东西，并且设置 session 删除机制。

3.存储方式不同：

> cookie 中只能保管 ASCII 字符串，并需要通过编码方式存储为 Unicode 字符或者二进制数据。
>
> session 中能够存储任何类型的数据，包括且不限于 string，integer，list，map 等。

4.隐私策略不同

> cookie 对客户端是可见的，别有用心的人可以分析存放在本地的 cookie 并进行 cookie 欺骗，所以它是不安全的。
>
> session 存储在服务器上，不存在敏感信息泄漏的风险。

5.有效期不同

> cookie 保管在客户端，不占用服务器资源。对于并发用户十分多的网站，cookie 是很好的选择。

> session 是保管在服务器端的，每个用户都会产生一个 session。假如并发访问的用户十分多，会产生十分多的 session，耗费大量的内存。

能设置或读取子域的cookie吗

> 不行! 只能向当前域或者更高级域设置cookie
>
> 例如 client.com 不能向 a.client.com 设置cookie, 而 a.client.com 可以向 client.com 设置cookie
>
> 读取cookie情况同上

客户端设置cookie与服务端设置cookie有什么区别

> 无论是客户端还是服务端, 都只能向自己的域或者更高级域设置cookie，例如 client.com 不能向 server.com 设置cookie, 同样 server.com 也不能向 client.com 设置cookie
>
> 服务端可以设置 `httpOnly: true`, 带有该属性的cookie客户端无法读取
>
> 客户端只会带上与请求同域的cookie, 例如 client.com/index.html 会带上 client.com 的cookie，server.com/app.js 会带上 server.com 的cookie, 并且也会带上httpOnly的cookie

同域/跨域ajax请求到底会不会带上cookie

> fetch在默认情况下, 不管是同域还是跨域ajax请求都不会带上cookie, 只有当设置了 credentials 时才会带上该ajax请求所在域的cookie, 服务端需要设置响应头`Access-Control-Allow-Credentials: true`, 否则浏览器会因为安全限制而报错, 拿不到响应
>
> axios和jQuery在同域ajax请求时会带上cookie, 跨域请求不会, 跨域请求需要设置`withCredentials`和服务端响应头`Access-Control-Allow-Credentials`

- fetch 设置 credentials 使fetch带上cookie

```
fetch(url, {
    credentials: "include", // include, same-origin, omit
})
```

- axios 设置 withCredentials 使axios带上cookie

```
axios.get('http://server.com', {withCredentials: true})
```

- jQuery 设置 withCredentials

```
$.ajax({
    method: 'get',
    url: 'http://server.com',
    xhrFields: {
        withCredentials: true
    }
})
```

前端攻击技术

XSS攻击(cross-site script)

1. XSS攻击形式：

   主要是通过html标签注入，篡改网页，插入恶意的脚本，前端可能没有经过严格的校验直接就进到数据库，数据库又通过前端程序又回显到浏览器

   例如一个留言板：
   如果内容是
   `hello!<script type="type/javascript src="恶意网址"></script>`
   这样会通过前端代码来执行js脚本，如果这个恶意网址通过cookie获得了用户的私密信息，那么用户的信息就被盗了

2. 攻击的目的：

   攻击者可通过这种方式拿到用户的一些信息，例如cookie 获取敏感信息，甚至自己建网站，做一些非法的操作等；或者，拿到数据后以用户的身份进行勒索，发一下不好的信息等。

3. 攻击防御

   方法1：cookie中设置 HttpOnly 属性

   方法2：首先前端要对用户输入的信息进行过滤，可以用正则，通过替换标签的方式进行转码或解码，例如<> 空格 & " ""等替换成html编码

```
htmlEncodeByRegExp:function (str){
  var s = "";
  if(str.length == 0) return "";
  s = str.replace(/&/g,"&amp;");
  s = s.replace(/</g,"&lt;");
  s = s.replace(/>/g,"&gt;");
  s = s.replace(/ /g," ");
  s = s.replace(/\'/g,"&#39;");
  s = s.replace(/\"/g,"&quot;");
  return s;
  }
```

CSRF攻击(cross site request forgery,跨站请求伪造)

1. CSRF攻击形式：

   CSRF也是一种网络攻击方式，比起xss攻击，是另外一种更具危险性的攻击方式，xss是站点用户进行攻击，而csrf是通过伪装成站点用户进行攻击，而且防范的资源也少，难以防范

   csrf攻击形式：攻击者盗用用户的身份信息，并以用户的名义进行发送恶意的请求等，例如发邮件，盗取账号等非法手段

   例如：你登录网站，并在本地种下了cookie
   如果在没退出该网站的时候 不小心访问了恶意网站，而且这个网站需要你发一些请求等
   此时，你是携带cookie进行访问的，那么你的种在cookie里的信息就会被恶意网站捕捉到，那么你的信息就被盗用，导致一些不法分子做一些事情

2. 攻击防御：

- 验证HTTP Referer字段

   在HTTP头中有Referer字段，他记录该HTTP请求的来源地址，如果跳转的网站与来源地址相符，那就是合法的，如果不符则可能是csrf攻击，拒绝该请求

- 在请求地址中添加token并验证

  这种的话在请求的时候加一个token，值可以是随机产生的一段数字，
  token是存入数据库之后，后台返给客户端的，如果客户端再次登录的时候，
  后台发现token没有，或者通过查询数据库不正确，那么就拒绝该请求

  如果想防止一个账号避免在不同的机器上登录，那么我们就可以通过token来判断，
  如果a机器登录后，我们就将用户的token从数据库清除，从新生成，
  那么另外一台b机器在执行操作的时候，token就失效了，只能重新登录，这样就可以防止两台机器登同一账号

- 在HTTP头中自定义属性并验证

  如果说通过每次请求的时候都得加token那么各个接口都得加很麻烦，
  那么我们通过http的请求头来设置token
  例如：

```
    $.ajax({
        url: '/v1/api',
        dataType: 'json',
        data: param,
        type:'post',
        headers: {'Accept':'application/json','Authorization':tokenValue}
        success:function(res){
            console.log(res)
        }
    })
```

浏览器缓存机制

> 参考链接：HTTP 强缓存和协商缓存

浏览器缓存分为：强缓存和协商缓存

在浏览器第一次发起请求时，本地无缓存，向 web 服务器发送请求，服务器起端响应请求，浏览器端缓存。在第一次请求时，服务器会将页面最后修改时间通过 Last-Modified 标识由服务器发送给客户端，客户端记录修改时间；服务器还会生成一个 Etag，并发送给客户端。

 浏览器缓存

根据上图，浏览器在第一次请求发生后，再次发送请求时：

- 浏览器请求某一资源时，会先获取该资源缓存的 header 信息，然后根据 header 中的 Cache-Control 和 Expires 来判断是否过期。若没过期则直接从缓存中获取资源信息，包括缓存的 header 的信息，所以此次请求不会与服务器进行通信。这里判断是否过期，则是强缓存相关。
- 如果显示已过期，浏览器会向服务器端发送请求，这个请求会携带第一次请求返回的有关缓存的 header 字段信息，比如客户端会通过 If-None-Match 头将先前服务器端发送过来的 Etag 发送给服务器，服务会对比这个客户端发过来的 Etag 是否与服务器的相同，若相同，就将 If-None-Match 的值设为 false，返回状态 304，客户端继续使用本地缓存，不解析服务器端发回来的数据，若不相同就将 If-None-Match 的值设为 true，返回状态为 200，客户端重新请求服务器端返回的数据；客户端还会通过 If-Modified-Since 头将先前服务器端发过来的最后修改时间戳发送给服务器，服务器端通过这个时间戳判断客户端的页面是否是最新的，如果不是最新的，则返回最新的内容，如果是最新的，则返回 304，客户端继续使用本地缓存。

强缓存 Expires 和 Cache-Control 的使用

强缓存是利用 http 头中的 Expires 和 Cache-Control 两个字段来控制的，用来表示资源的缓存时间。强缓存中，普通刷新会忽略它，但不会清除它，需要强制刷新。浏览器强制刷新，请求会带上`Cache-Control:no-cache`和`Pragma:no-cache`

Expires

Expires 的值是一个绝对时间的 GMT 格式的时间字符串。比如 Expires 值是：`expires:Fri, 14 Apr 2017 10:47:02 GMT`。这个时间代表这这个资源的失效时间，只要发送请求时间是在 Expires 之前，那么本地缓存始终有效，则在缓存中读取数据。

缺点：
由于失效的时间是一个绝对时间，所以当服务器与客户端时间偏差较大时，误差很大，就会导致缓存混乱。

Cache-Control

Cache-Control 主要是利用该字段的`max-age`值来进行判断，它是一个相对时间，例如`Cache-Control:max-age=3600`，代表着资源的有效期是 3600 秒。

cache-control 除了该字段外，还有下面几个比较常用的设置值：

- no-cache：不使用本地缓存。需要使用缓存协商，先与服务器确认返回的响应是否被更改，如果之前的响应中存在 ETag，那么请求的时候会与服务端验证，如果资源未被更改，则可以避免重新下载。
- no-store：直接禁止浏览器缓存数据，每次用户请求该资源，都会向服务器发送一个请求，每次都会下载完整的资源。
- public：可以被所有的用户缓存，包括终端用户和 CDN 等中间代理服务器。
- private：只能被终端用户的浏览器缓存，不允许 CDN 等中继缓存服务器对其缓存。

Cache-Control 与 Expires 可以在服务端配置同时启用，同时启用的时候 Cache-Control 优先级高。
如：

```
cache-control:max-age=691200
expires:Fri, 15 May 2020 10:47:02 GMT
```

那么表示资源可以被缓存的最长时间为 691200 秒。

协商缓存

协商缓存就是由服务器来确定缓存资源是否可用，所以客户端与服务器端要通过某种标识来进行通信，从而让服务器判断请求资源是否可以缓存访问。

Etag 和 If-None-Match

`Etag/If-None-Match`返回的是一个校验码。`Etag`可以保证每一个资源是唯一的，资源变化都会导致`Etag`变化。服务器根据浏览器发送的`If-None-Match`值来判断是否命中缓存。

与`Last-Modified`不一样的是，当服务器返回 304 (Not Modified) 的响应时，由于`Etag`重新生成过，response header 中还会把这个`Etag`返回，即使这个`Etag`跟之前的没有变化。

Last-Modify / If-Modify-Since

浏览器第一次请求一个资源的时候，服务器返回的header中会加上Last-Modify，Last-Modify是一个时间标识该资源的最后修改时间，例如Last-Modify: Thu,31 Dec 2037 23:59:59 GMT。当浏览器再次请求该资源时，request的请求头中会包含If-Modify-Since，该值为缓存之前返回Last-Modify。服务器收到If-Modify-Since后，根据资源的最后修改时间判断是否命中缓存。如果命中缓存，则返回304，并且不会返回资源内容，并且不会返回Last-Modify。

为什么要有 Etag

> 两个都可以确定缓存资源的是否可用，有什么区别呢?

`Etag` 的出现主要是为了解决几个 `Last-Modified` 比较难解决的问题：

1. 一些文件也许会周期性的更改，但是他的内容并不改变(仅仅改变的修改时间)，这个时候我们并不希望客户端认为这个文件被修改了，而重新 GET;
2. 某些文件修改非常频繁，比如在秒以下的时间内进行修改，(比方说 1s 内修改了 N 次)，`If-Modified-Since` 能检查到的力度是秒级的，这种修改无法判断;
3. 某些服务器不能精确的得到文件的最后修改时间。

`Last-Modified` 与 `ETag` 是可以一起使用的，服务器会优先验证 `ETag`，一致的情况下，才会继续比对 `Last-Modified`，最后才决定是否返回 304。

进程与线程的区别

官网定义：
进程是系统进行资源分配和调度的基本单位

线程是操作系统能够进行运算调度的最小单位

简单理解：
进程：指在系统中正在运行的一个应用程序；程序一旦运行就是进程；进程——资源分配的最小单位。

线程：系统分配处理器时间资源的基本单元，或者说进程之内独立执行的一个单元执行流。线程——程序执行的最小单位。

借助阮一峰老师的解释
1.计算机的核心是CPU，它承担了所有的计算任务。它就像一座工厂，时刻在运行。
2.假定工厂的电力有限，一次只能供给一个车间使用。也就是说，一个车间开工的时候，其他车间都必须停工。背后的含义就是，单个CPU一次只能运行一个任务。
3.进程就好比工厂的车间，它代表CPU所能处理的单个任务。任一时刻，CPU总是运行一个进程，其他进程处于非运行状态。
4.一个车间里，可以有很多工人。他们协同完成一个任务。
5.线程就好比车间里的工人。一个进程可以包括多个线程。
6.车间的空间是工人们共享的，比如许多房间是每个工人都可以进出的。这象征一个进程的内存空间是共享的，每个线程都可以使用这些共享内存。
7.可是，每间房间的大小不同，有些房间最多只能容纳一个人，比如厕所。里面有人的时候，其他人就不能进去了。
这代表一个线程使用某些共享内存时，其他线程必须等它结束，才能使用这一块内存。
8.一个防止他人进入的简单方法，就是门口加一把锁。先到的人锁上门，后到的人看到上锁，就在门口排队，等锁打开再进去。
这就叫"互斥锁"（Mutual exclusion，缩写 Mutex），防止多个线程同时读写某一块内存区域。
9.还有些房间，可以同时容纳n个人，比如厨房。也就是说，如果人数大于n，多出来的人只能在外面等着。这好比某些内存区域，只能供给固定数目的线程使用。

10.这时的解决方法，就是在门口挂n把钥匙。进去的人就取一把钥匙，出来时再把钥匙挂回原处。后到的人发现钥匙架空了，就知道必须在门口排队等着了。

这种做法叫做"信号量"（Semaphore），用来保证多个线程不会互相冲突。

操作系统的设计，因此可以归结为三点：

（1）以多进程形式，允许多个任务同时运行；

（2）以多线程形式，允许单个任务分成不同的部分运行；

（3）提供协调机制，一方面防止进程之间和线程之间产生冲突，另一方面允许进程之间和线程之间共享资源。