代码随想录知识星球精华（最强八股文）第四版（前端篇））

代码随想录知识星球精华（最强八股文）第四版为九份PDF，分别是：

- 代码随想录知识星球八股文概述
- C++篇
- go篇
- Java篇
- 前端篇
- 算法题篇
- 计算机基础篇
- 问答精华篇
- 面经篇

本篇为最强八股文之前端篇。

HTML

iframe

iframe 元素 可以在一个网站里面嵌入另一个网站内容

优点

1. 实现一个窗口同时加载多个第三方域名下内容

2. 增加代码复用性

缺点：

1. 搜索引擎无法识别
2. 影响首页首屏加载时间
3. 兼容性差

限制iframe访问另一个页面

设置X-Frame-Options 响应头 ——是否允许网页通过iframe 嵌套

1. deny：完全禁止任何网页嵌套
2. sameorigin：只允许同源域名访问
3. allow-from url：允许url的域名可嵌套

设置Content-Security-Policy

CSP，内容安全策略，限定网页允许加载的资源，防范XSS攻击

```
"Content-Security-Policy": "frame-ancestors 'self'"//限定iframe的嵌套
```

判断 window.top 页面顶级窗口和 自身窗口 window.self 是否相等，若不等则是嵌入了iframe

defer 和 async

defer和async是script标签的两个属性，用于在不阻塞页面文档解析的前提线下，控制脚本的下载和执行。

先了解一下页面的加载和渲染过程：

1. 浏览器发送请求，获取HTML文档开始从上到下解析并构建DOM
2. 构建DOM过程中，若遇到外联样式声明和脚本声明，会暂停文档解析，并开始下载样式脚本和文件
3. 样式文件下载完成后，构建CSSDOM；脚本文件下载完成后，解析并执行，再继续解析文档构建DOM
4. 文档解析完成后，将DOM和CSSDOM进行关联和映射，最后将视图渲染到浏览器窗口

注意，在上述过程中，脚本文件的下载和执行是和文档解析同步的，即它会阻塞文档的解析，若控制的不好，会影响用户体验，造成页面卡顿。

因此我们可以在script中声明defer和async这两个属性。

defer：用于开启新的线程下载脚本文件，并使脚本在文档解析完成后执行。 async：HTML5新增属性，用于异步下载脚本文件，下载完毕立即解释执行代码。

共同点

- 都是异步加载外部脚本文件
- 不会阻塞页面的解析

区别

- async表示异步加载，表后续文档的加载和渲染与JS脚本加载和执行并行进行,脚本文件一旦加载完成，会立即执行，我们**无法预测每个脚本的下载和执行时间顺序，谁先下载好谁执行**。

- defer表示延迟加载，加载后续文档和JS脚本加载（仅加载不执行)并行进行，JS脚本的执行需要等待文档所有元素解析完之后，load和DOMContentLoaded事件之前执行，有顺序而言。

DOCUTYPE的作用

DOCTYPE是document type (文档类型) 的缩写。 是HTML5中一种标准通用标记语言的文档类型声明，**告诉浏览器文档的类型，便于解析文档**。 不同的渲染模式会影响浏览器对 CSS 代码甚至 JavaScript 脚本的解析。它必须声明在HTML文档的第一行.

浏览器渲染页面的两种模式

- CSS1 Compat：标准模式（Strick mode），浏览器使用W3C的标准解析渲染页面。浏览器以其支持的最高标准呈现页面。
- BackCompat：怪异模式(混杂模式)(Quick mode)，默认模式，页面以一种比较宽松的向后兼容的方式显示。

> DOCTYPE 不存在或者形式不正确会导致HTML或XHTML文档以混杂模式呈现，就是把如何渲染html页面的权利交给了浏览器，有多少种浏览器就有多少种展示方式。因此要提高浏览器兼容性就必须重视。

html语义化

根据内容的结构选择合适的标签

优点

- 增加代码**可读性**，结构清晰，便于开发和维护
- 对机器友好，文字表现力丰富，**有利于SEO**。SEO(Search Engine Optimization)是搜索引擎优化，为了让用户在搜索和网站相关的关键词的时候，可以使网站在搜索引擎的排名尽量靠前，从而增加流量。
- 方便设备解析（如盲人阅读器等），可用于智能分析
- 在没有 CSS 样式下，页面也能呈现出很好地内容结构、代码结构

常见的语义化标签

- title： 页面主体内容
- header： 页眉通常包括网站标志、主导航、全站链接以及搜索框。
- nav： 标记导航，仅对文档中重要的链接群使用。
- section： 定义文档中的节（section、区段）。比如章节、页眉、页脚或文档中的其他部分。
- main，帮助到搜索引擎以及搜索工程师找到网站的主要内容，本身并不承载特殊的功能和意义。
- article： 定义外部的内容，其中的内容独立于文档的其余部分。
- aside： 定义其所处内容之外的内容。如侧栏、文章的一组链接、广告、友情链接、相关产品列表等。
- footer：页脚，只有当父级是body时，才是整个页面的页脚。
- address： 作者、相关人士或组织的联系信息（电子邮件地址、指向联系信息页的链接）。

meta标签

meta标签用来描述一个HTML网页文档的属性，例如作者、日期和时间、网页描述、关键词、页面刷新等。

meta标签的作用有：搜索引擎优化（SEO），定义页面使用语言，自动刷新并指向新的页面，实现网页转换时的动态效果，控制页面缓冲，网页定级评价，控制网页显示的窗口等。

meta分类

1）页面描述信息(NAME)：常用的选项有Keywords(关键字)，description(网站内容描述)，author(作者)，robots(机器人向导)等。

2）HTTP标题信息(HTTP-EQUIV)：可用于代替name项，常用的选项有Expires(期限)，Pragma(cache模式)，Refresh(刷新)，Set-Cookie(cookie设定)，Window-target(显示窗口的设定)，content-Type(显示字符集的设定)等。

3）content项：根据name项或http-equiv项的定义来决定此项填写什么样的字符串。

HTML5特性

新增

- 新的选择器 document.querySelector、document.querySelectorAll
- 媒体播放的 video 和 audio 标签
- 以前用的flash实现
- 本地存储 localStorage 和 sessionStorage
- 浏览器通知 Notifications
- 语义化标签，例如 header，nav，footer，section，article 等标签。
- 地理位置 Geolocation
- 鉴于隐私性，除非用户统一，否则不可获取用户地理位置信息
- 离线应用 manifest
- 全双工通信协议 websocket
- 浏览器历史对象 history
- 多任务处理 webworker
- 运行在后台的JS，独立于其他脚本，不影响性能
- 拖拽相关API
- 增强表单控件 url，date，time，email，calendar，search
- 页面可见性改变事件 visibilitychange
- 跨窗口通信 PostMessage
- 表单 FormData 对象
- canvas+SVG

移除

- 纯表现的元素：basefont、big、center、font、 s、strike、tt、u
- 对可用性产生负面影响的元素：frame、frameset、noframes

src 和 href 的区别

`src` 用于替换当前元素，`href` 用于在当前文档和引用资源之间确立联系

一、src

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

当浏览器解析到该元素时，浏览器会对这个文件进行解析，编译和执行，从而导致整个页面加载会被暂停，类似于将所指向资源嵌入当前标签内。这也是为什么将js 脚本放在底部而不是头部。

二、href

href 是 Hypertext Reference 的缩写，指向网络资源所在位置，建立和当前元素（锚点）或当前文档（链接）之间的链接，浏览器遇到 href 就会并行下载资源并且不会停止对当前文档的处理。 这也是为什么建议使用 link 方式来加载 css，而不是使用@import 。

三、总结来说

src 代表这个资源是必备的，必不可少的，最终会嵌入到页面中，而 href 是资源的链接

行内元素 和 块级元素有哪些

一、元素种类

1、行内元素

- 和其他元素都在一行上；
- 高，行高及外边距和内边距部分可改变；
- 宽度只与内容有关；
- 行内元素只能容纳文本或者其他行内元素。
- 不可以设置宽高，其宽度随着内容增加，高度随字体大小而改变，内联元素可以设置外边界，但是外边界不对上下起作用，只能对左右起作用，也可以设置内边界，但是内边界在 ie6 中不对上下起作用，只能对左右起作用

```
<a >
<strong>
<b>
<em>
<del>
<span >
<img>
<input>
<select>
```

2、块级元素

- 总是在新行上开始，占据一整行；
- 高度，行高以及外边距和内边距都可控制；
- 宽带始终是与浏览器宽度一样，与内容无关；
- 它可以容纳内联元素和其他块元素。

```
<h1>~<h6>
<p>
<div>
<ul>
<ol>
<li>
<div>
<dl>
```

3、空元素

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

二、行内元素和块级元素的转换

```
//定义元素为块级元素
display: block
//定义元素为行内元素
display : inline
//定义元素为行
display: inline-block
```

三、块级元素和行内元素的区别

我们区分 块级元素 和 行内元素，首先行内元素是在一行中能有多个的，块级元素是自己占一行的。

接着可以从三个方面来查看

- 是否占据一行，还是能多个处于一行中，行内是可以的；
- 是否可以设置宽高，行内是不可以的。
- 行内元素只可以容纳文本和其他行内元素，块级元素啥都可以容纳

总结

行内元素和块级元素很好区分，顾名思义，行内就是能都在一行里的，一行可以有多个 < a > 标签 ，可以有多个 < input > 标签，而同类的还有 < span > / < img > / < strong > 等，这不就很容易记住了。而块级元素就是要自己占一行的，那不就有 < div > ，还有我们使用的列表 < ul > / < ol >，除此之外还有 < h1 >~< h6 > / < p > 等等。

link 与 @import 的区别

一、语法结构

```
//link
<link href="路径" rel="stylesheet" type="text/css" />

//import
@import url(路径地址);
```

二、区别

1、从属关系区别

@import是 CSS 提供的语法规则，只有导入样式表的作用；link是HTML提供的标签，不仅可以加载 CSS 文件，还可以定义 RSS、rel 连接属性等。

2、加载顺序区别

加载页面时，link标签引入会和 html 同时加载；而 @import 引入的 CSS 将在页面加载完毕后被加载。

3、兼容性区别

@import是 CSS2.1 才有的语法，可在 IE5+ 才能识别；link标签作为 HTML 元素，不存在兼容性问题。

4、DOM可控性区别

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、权重

link方式的样式权重高于@import的权重。

总的来说，使用 link 会比 @import 好，首先兼容性， link 作为 html 的标签是不会存在这个问题的。并且如果使用 @import 是在 dom 树渲染完才会进行渲染的，所以是不被 JavaScript 动态的修改的

常见的图片格式

图片格式	优点	缺点	使用场景
GIF	文件小，支持动画、透明，无兼容性问题	只支持256种颜色	色彩简单的logo、icon、动图
JPG	色彩丰富，文件小	有损压缩，反复保存图片质量下降明显	色彩丰富的图片/渐变图像
PNG	无损压缩，支持透明，简单图片尺寸小	不支持动画，色彩丰富的图片尺寸大	logo/icon/透明图
WEBP	文件小，支持有损和无损压缩，支持动画、透明	浏览器兼容性相对而言不好	支持webp格式的app和webview

CSS

HTML页面中 id 和 class 有什么区别

1. 在css样式表中书写时，id选择符前缀应加"#“，class选择符前缀应加”."
2. id属性在一个页面中书写时只能使用一次，而class可以反复使用
3. id作为元素标签用于区分不同结构和内容，而class作为一个样式，可以应用到任何结构和内容当中去
4. 布局上的一般原则：id先确定结构和内容再为它定义样式。而class正好相反，是先定义样式，然后在页面中根据不同需求把样式应用到不同结构和内容上
5. 目前浏览器都允许同一个页面出现多个相同属性值的id，一般情况能正常显示，不过当javascript通过id来控制元素时就会出错
6. 在实际应用中，class常被用到文字版块和页面修饰上，而id多被用在宏伟布局和设计包含块，或包含框的样式。

伪元素和伪类

(:)用于伪类，(::)用于伪元素。

::before以一个子元素存在，定义的一个伪元素，只存在页面中。

伪元素：对选择元素**的指定部分**进行修改样式，常见的有 :before，:after，:first-line，first-letter 等等

伪类：对选择元素的**特殊状态**进行修改样式，常见的有 :hover，:active，:checked，:focus，:first-child 等等

CSS中的继承

继承属性：

- 当元素的一个继承属性 没有指定值时，则取父元素的同属性的计算值

非继承属性：

- 当元素的一个非继承属性没有指定值时，则取属性的初始值

那么这两类属性都有哪些呢?

一、无继承性的属性

1、display：

- 规定元素应该生成的框的类

2、文本属性：

- vertical-align：垂直文本对齐

- text-decoration：规定添加到文本的装饰
- text-shadow：文本阴影效果
- white-space：空白符的处理
- unicode-bidi：设置文本的方向

3、盒子模型的属性：

- width、height、margin 、margin-top、margin-right、margin-bottom、margin-left、border、border-style、border-top-style、border-right-style、border-bottom-style、border-left-style、border-width、border-top-width、border-right-right、border-bottom-width、border-left-width、border-color、border-top-color、border-right-color、border-bottom-color、border-left-color、border-top、border-right、border-bottom、border-left、padding、padding-top、padding-right、padding-bottom、padding-left

4、背景属性：

- background、background-color、background-image、background-repeat、background-position、background-attachment

5、定位属性：

- float、clear、position、top、right、bottom、left、min-width、min-height、max-width、max-height、overflow、clip、z-index

6、生成内容属性：

- content、counter-reset、counter-increment

7、轮廓样式属性：

- outline-style、outline-width、outline-color、outline

8、页面样式属性：

- size、page-break-before、page-break-after

9、声音样式属性：

- pause-before、pause-after、pause、cue-before、cue-after、cue、play-during

二、有继承性的属性

1、字体系列属性

- font：组合字体
- font-family：规定元素的字体系列
- font-weight：设置字体的粗细
- font-size：设置字体的尺寸
- font-style：定义字体的风格
- font-variant：设置小型大写字母的字体显示文本，这意味着所有的小写字母均会被转换为大写，但是所有使用小型大写字体的字母与其余文本相比，其字体尺寸更小。
- font-stretch：对当前的 font-family 进行伸缩变形。所有主流浏览器都不支持。
- font-size-adjust：为某个元素规定一个 aspect 值，这样就可以保持首选字体的 x-height。

2、文本系列属性

- text-indent：文本缩进
- text-align：文本水平对齐
- line-height：行高
- word-spacing：增加或减少单词间的空白（即字间隔）
- letter-spacing：增加或减少字符间的空白（字符间距）
- text-transform：控制文本大小写
- direction：规定文本的书写方向
- color：文本颜色

3、元素可见性：

- visibility

4、表格布局属性：

- caption-side、border-collapse、border-spacing、empty-cells、table-layout

5、列表布局属性：

- list-style-type、list-style-image、list-style-position、list-style

6、生成内容属性：

- quotes

7、光标属性：

- cursor

8、页面样式属性：

- page、page-break-inside、windows、orphans

9、声音样式属性：

- speak、speak-punctuation、speak-numeral、speak-header、speech-rate、volume、voice-family、pitch、pitch-range、stress、richness、、azimuth、elevation

三、所有元素可以继承的属性

1、元素可见性：

- visibility

2、光标属性：

- cursor

四、内联元素可以继承的属性

1、字体系列属性

2、除text-indent、text-align之外的文本系列属性

五、块级元素可以继承的属性

1、text-indent、text-align

不可/可继承属性

可继承

- 字体属性
- 文本属性
- 元素可见性
- 列表布局属性
- 光标属性

不可继承

- 盒子模型属性
- display
- 背景属性
- 定位属性
- 生成内容、轮廓样式、页面样式、声音样式

display: none与 visibility: hidden的区别

这两个属性都是让元素隐藏，不可见。两者的区别主要分两点

1、是否在渲染树中

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

2、是否是继承属性

- display: none 是非继承属性，子孙节点会随着父节点从渲染树消失，通过修改子孙节点的属性也无法显示；
- visibility: hidden 是继承属性，子孙节点消失是由于继承了 hidden，通过设置 visibility: Avisible可以让子孙节点显示；
- 修改常规文档流中元素的 display 通常会造成文档的重排，但是修改 visibility 属性只会造成本元素的重绘
- 如果使用读屏器，设置为 display: none 的内容不会被读取，设置为 visibility: hidden 的内容会被读取。

> 这两者的关系类似于 v-if 和 v-show 之间的关系

隐藏元素的方法有哪些

常见的隐藏属性的方法有 display: none 与 visibility: hidden：

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

- transform: scale(0,0)：将元素缩放为0，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。

> 在VUE中还有 v-if 和 v-show 这两个指令，在目录你能看到关于这两个的使用区别

CSS选择器

选择器	例子	例子描述
.class	.intro	选择 class="intro" 的所有元素。
.class1.class2	.name1.name2	选择 class 属性中同时有 name1 和 name2 的所有元素。
.class1 .class2	.name1 .name2	选择作为类名 name1 元素后代的所有类名 name2 元素。
#id	#firstname	选择 id="firstname" 的元素。
*	*	选择所有元素。
element	p	选择所有 元素。
element.class	p.intro	选择 class="intro" 的所有 元素。
element,element	div, p	选择所有 元素和所有 元素。
element element	div p	选择 元素内的所有 元素。
element>element	div > p	选择父元素是 的所有 元素。
element+element	div + p	选择紧跟 元素的首个 元素。
element1~element2	p ~ ul	选择前面有 元素的每个 元素。
[attribute]	[target]	选择带有 target 属性的所有元素。
[attribute=value]	[target=_blank]	选择带有 target="_blank" 属性的所有元素。
[attribute~=value]	[title~=flower]	选择 title 属性包含单词 "flower" 的所有元素。
[attribute	=value]	[lang
[attribute^=value]	a[href^="https"]	选择其 src 属性值以 "https" 开头的每个 元素。
[attribute$=value]	a[href$=".pdf"]	选择其 src 属性以 ".pdf" 结尾的所有 元素。
[attribute*=value]	a[href*="w3schools"]	选择其 href 属性值中包含 "abc" 子串的每个 元素。
:active	a:active	选择活动链接。
::after	p::after	在每个 的内容之后插入内容。

::before	p::before	在每个 的内容之前插入内容。
:checked	input:checked	选择每个被选中的 元素。
:default	input:default	选择默认的 元素。
:disabled	input:disabled	选择每个被禁用的 元素。
:empty	p:empty	选择没有子元素的每个 元素（包括文本节点）。
:enabled	input:enabled	选择每个启用的 元素。
:first-child	p:first-child	选择属于父元素的第一个子元素的每个 元素。
::first-letter	p::first-letter	选择每个 元素的首字母。
::first-line	p::first-line	选择每个 元素的首行。
:first-of-type	p:first-of-type	选择属于其父元素的首个 元素的每个 元素。
:focus	input:focus	选择获得焦点的 input 元素。
:fullscreen	:fullscreen	选择处于全屏模式的元素。
:hover	a:hover	选择鼠标指针位于其上的链接。
:in-range	input:in-range	选择其值在指定范围内的 input 元素。
:indeterminate	input:indeterminate	选择处于不确定状态的 input 元素。
:invalid	input:invalid	选择具有无效值的所有 input 元素。
:lang(language)	p:lang(it)	选择 lang 属性等于 “it”（意大利）的每个 元素。
:last-child	p:last-child	选择属于其父元素最后一个子元素每个 元素。
:last-of-type	p:last-of-type	选择属于其父元素的最后 元素的每个 元素。
:link	a:link	选择所有未访问过的链接。
:not(selector)	:not§	选择非 元素的每个元素。
:nth-child(n)	p:nth-child(2)	选择属于其父元素的第二个子元素的每个 元素。
:nth-last-child(n)	p:nth-last-child(2)	同上，从最后一个子元素开始计数。
:nth-of-type(n)	p:nth-of-type(2)	选择属于其父元素第二个 元素的每个 元素。
:nth-last-of-type(n)	p:nth-last-of-type(2)	同上，但是从最后一个子元素开始计数。
:only-of-type	p:only-of-type	选择属于其父元素唯一的 元素的每个 元素。
:only-child	p:only-child	选择属于其父元素的唯一子元素的每个 元素。
:optional	input:optional	选择不带 “required” 属性的 input 元素。

:out-of-range	input:out-of-range	选择值超出指定范围的 input 元素。
::placeholder	input::placeholder	选择已规定 “placeholder” 属性的 input 元素。
:read-only	input:read-only	选择已规定 “readonly” 属性的 input 元素。
:read-write	input:read-write	选择未规定 “readonly” 属性的 input 元素。
:required	input:required	选择已规定 “required” 属性的 input 元素。
:root	:root	选择文档的根元素。
::selection	::selection	选择用户已选取的元素部分。
:target	#news:target	选择当前活动的 #news 元素。
:valid	input:valid	选择带有有效值的所有 input 元素。
:visited	a:visited	选择所有已访问的链接。

CSS选择器优先级

一、CSS 有多少种样式类型：

1. 行内样式：< style/style >
2. 内联样式：< div style="color:red> ;
3. 外部样式：< link > 或 @import引入

二、常见选择器及选择器权重

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

三、注意事项

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

position 属性

1. relative：相对自身之前正常文档流中的位置发生偏移，与世隔绝，且原来的位置仍然被占据。发生偏移时，可能覆盖其他元素。**body默认是relative,子绝父相。**
2. absolute：元素框不再占有文档位置，并且相对于包含块进行偏移（所谓包含块就是最近一级外层元素position不为static的元素）。
3. fixed：元素框不再占有文档流位置，并且相对于视窗进行定位。
4. static：默认值，取消继承。
5. sticky：css3新增属性值，粘性定位，相当于relative和fixed的混合。最初会被当作是relative，相对原来位置进行偏移；一旦超过一定的阈值，会被当成fixed定位，相对于视口定位。
6. inherit

px、em、rem、vh、vw的区别及使用场景

这几个单位是在写长度的时候常常会用到的：

一、px

px：也就是像素，是基于屏幕分辨率来说的，一旦设置了，就无法适应页面大小的变化。

二、em

em：是相对单位，相对于当前对象内文本的字体大小（也就是它的父元素），如果当前对象内文本的字体没有设置大小，就会相对于浏览器默认字体大小也就是16px。所以在没有设置的情况下 1em = 16px。

> 为了便于运算你可以在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。

三、rem

rem：rem的出现是为了解决em的问题的，em是相对于父元素来说的，这就要求我们进行任何元素设置的时候，都需要知道它的父元素字体的大小。而 rem是相对于根元素，这样就意味着，我们只需要在根元素确定一个参考值，就可以了，同时还能做到只修改根元素就成比例地调整所有字体大小。

四、vh、vw

vh、vw：vw 是根据窗口的宽度。会把窗口的大小分为100份，所以50vw代表窗口大小的一半。并且这个值是相对的，当窗口大小发生改变也会跟着改变，同理，vh则为窗口的高度

四、总的来说，四者的区别：

`px` 是固定的大小，`em` 是相对于父元素字体的大小，`rem` 是相对于根元素字体的大小，`vh`、`vw` 是相对可是窗口的大小

五、使用场景的区别：

- 一般我们在设置边框和边距的时候用 px 比较好
- 而在一些需要做响应式的页面用 rem 比较便捷
- 但是具体还是得看你的业务来定的

脱离文档流

定位流

元素的属性 position 为 absolute 或 fixed，它就是一个绝对定位元素。

在绝对定位布局中，元素会整体脱离普通流，因此绝对定位元素不会对其兄弟元素造成影响，而元素具体的位置由绝对定位的坐标决定。

它的定位相对于它的包含块，相关CSS属性：top、bottom、left、right；

对于 position: absolute，元素定位将相对于上级元素中最近的一个relative、fixed、absolute，如果没有则相对于body；

对于 position:fixed，正常来说是相对于浏览器窗口定位的，但是当**元素祖先的 transform 属性非 none 时，会相对于该祖先进行定位。**

浮动流

在浮动布局中，元素首先按照普通流的位置出现，然后根据浮动的方向尽可能地向左边或右边偏移，其效果与印刷排版中的文本环绕相似。

普通流

普通流其实就是指BFC中的FC。FC(Formatting Context)，直译过来是格式化上下文，它是页面中的一块渲染区域，有一套渲染规则，决定了其子元素如何布局，以及和其他元素之间的关系和作用。

在普通流中，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。

盒子模型

1、盒模型宽度的计算

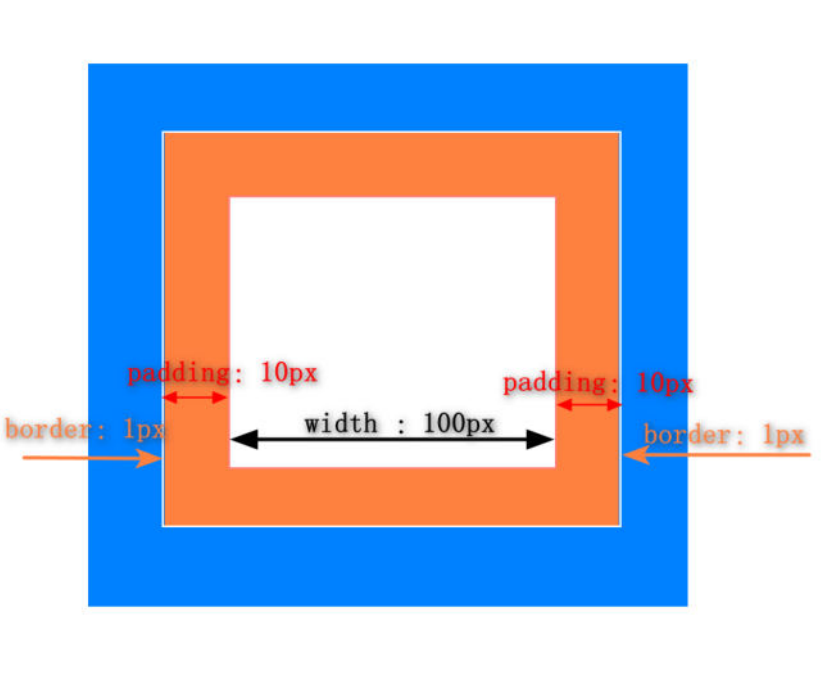
（1）普通盒模型

默认盒子属性：box-sizing: content-box;

width只包含内容宽度，不包含border和padding

offsetWidth = (width + padding + border)，不算 margin

width 和 height 属性只会应用到这个元素的内容区



box-sizing: content-box;//定义引擎如何计算元素的总高度和总宽度

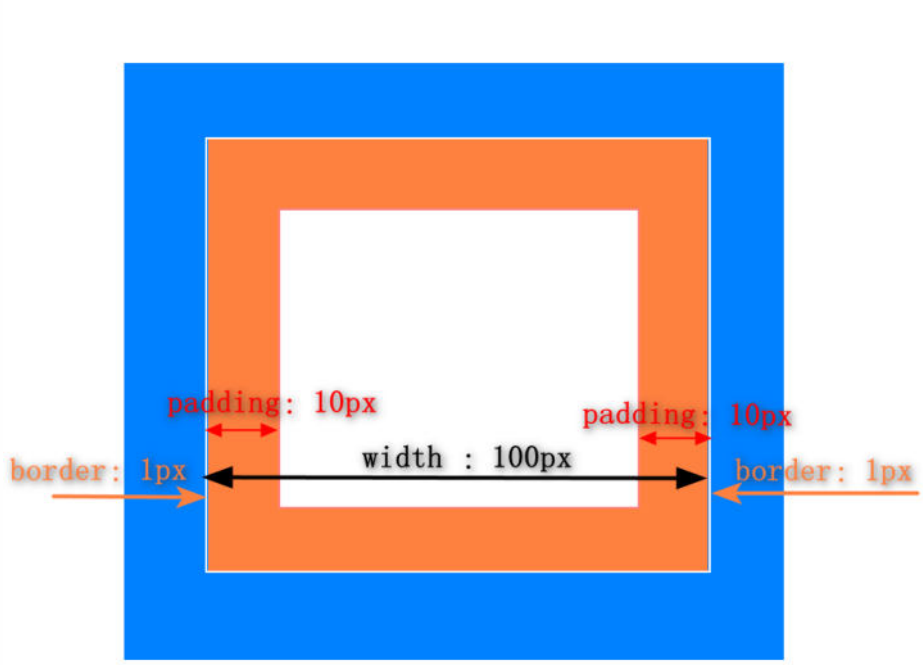
- content-box 默认值，元素的 width/height 不包含padding，border，与标准盒子模型表现一致
- border-box 元素的 width/height 包含 padding，border，与怪异盒子模型表现一致
- inherit 指定 box-sizing 属性的值，应该从父元素继承

（2）怪异盒模型

设置语句：box-sizing: border-box;

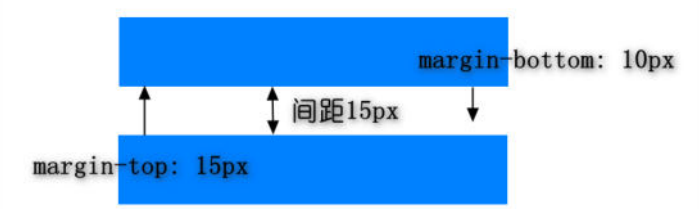
offsetWidth = width（padding 和 border 都挤压到内容里面）

width 和 height 包括内容区、padding 和 border，不算 margin



2、margin 纵向重叠

margin 纵向重叠取重叠区最大值，不进行叠加



3、margin 负值问题

margin-top 和 margin-left 是负值，元素会向上或者向左移动

margin-right 负值，右侧元素左移，自身不受影响

margin-bottom 负值，下侧元素上移，自身不受影响

4、BFC

Block Format Context：块级格式化上下文

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

形成 BFC 的条件：

float 不设置成 none
position 是 absolute 或者 fixed
overflow 不是 visible
display 是 flex 或者 inline-block 等

应用：清除浮动

5、float

（1）圣杯布局和双飞翼布局

作用：

实现 pc 端三栏布局，中间一栏最先渲染
实现两边宽度固定，中间自适应

效果图：

（2）圣杯布局

HTML：

```
<div class="container clearfix">
    <div class="main float"></div>
    <div class="left float"></div>
    <div class="right float"></div>
</div>
```

CSS：

```
.container {
    padding: 0 200px;
    background-color: #eee;
}

/* 清除浮动 */
.clearfix::after {
    content: '';
    display: table;
    clear: both;
}

/* 关键 */
.float {
    float: left;
}

.main {
    width: 100%;
    height: 200px;
    background-color: #ccc;
}

.left {
```

```
    width: 200px;
    height: 200px;
    /* ---关键--- */
    position: relative;
    right: 200px;
    margin-left: -100%;
    /* ---关键--- */
    background-color: orange;
}

.right {
    width: 200px;
    height: 200px;
    /* ---关键--- */
    margin-right: -200px;
    /* ---关键--- */
    background-color: skyblue;
}
```

（3）双飞翼布局

HTML：

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

CSS：

```
/* 关键 */
.float {
    float: left;
}

.wrapper {
    width: 100%;
    height: 200px;
    background-color: #ccc;
}

/* 关键 */
.wrapper .main {
    height: 200px;
    margin-left: 200px;
    margin-right: 200px;
}

.left {
    width: 200px;
```

```
    height: 200px;
    /* 关键 */
    margin-left: -100%;
    background-color: orange;
}

.right {
    width: 200px;
    height: 200px;
    /* 关键 */
    margin-left: -200px;
    background-color: skyblue;
}
```

（4）对比

属性	圣杯布局	双飞翼布局
HTML	包裹三栏	只包裹中间一栏
是否定位	相对定位	无需定位
左右栏的空间	使用 padding 预留	使用 margin 预留
左栏处理	positon + margin-left	margin-left
右栏处理	margin-right	margin-left

（5）手写clearfix

CSS：

```
/* 1、父级标签定义伪类 */
.clearfix::after {
    content: '';
    display: table;
    clear: both;
}
/* 兼容IE低版本 */
.clearfix {
    *zoom: 1;
}

/* 2、父级标签 overflow */
.clearfix {
    overflow: hidden;
}

/* 3、添加空 div 标签 */
.clearfix {
    clear: both;
}
```

6、元素居中

（1）行内元素水平垂直居中

设置父级标签。

水平居中： text-align: center

垂直居中： line-height：盒子高度

（2）块级元素水平垂直居中

水平居中:

```
margin : 0 auto;
```

水平垂直都居中

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

不会触发重排，因此最常用

```
position: absolute;
top: 0;
left: 0;
right: 0;
bottom: 0;
margin: auto;
```

容器设置:

```
display: flex;
justify-content: center;
align-items: center;
```

容器:

```
display: table-cell;
text-align: center;
vertical-align: middle;
```

子元素:

```
display: inline-block;
```

7、样式单位

em：相对于自身字体大小的单位
rem：相对于 html 标签字体大小的单位

vh：相对于视口高度大小的单位，20vh == 视口高度/10020
vw：相对于视口宽度大小的单位, 20vw == 视口宽度/10020

两栏布局

左边宽度固定，右边宽度自适应。

- 利用**flex布局**，将左边元素设置为固定宽度200px，将右边的元素设置为flex:1
- 利用**浮动**。左边元素宽度设置为200px，且设置向左浮动。右边元素的margin-left设置为200px，宽度设置为auto（默认为auto，撑满整个父元素）。margin-left/padding-left/calc
- 利用浮动。**左边元素宽度固定，设置向左浮动**。右侧元素设置 overflow: hidden; 这样右边就触发了 **BFC**，BFC 的区域不会与浮动元素发生重叠，所以两侧就不会发生重叠。 float + overflow:hidden
  - 左列左浮动,将自身高度塌陷,使得其它块级元素可以和它占据同一行位置
  - 右列利用自身流特性占满整行
  - 右列设置overflow,触发BFC特性,使自身和左列浮动元素隔开,不沾满整行
- 绝对定位 **父级相对定位 左边absolute定位，宽度固定**。设置右边margin-left为**左边元素的宽度值**。
- 绝对定位，父级元素相对定位。左边元素宽度固定，右边元素absolute定位，**left为宽度大小**，其余方向为0。**(有歧义,谨慎使用!)**
- 使用 calc 计算

```
.left {

  display: inline-block;
  width: 240px;
}

.right {

  display: inline-block;
  width: calc(100% - 240px);
}

//使用 calc() 函数计算 <div> 元素的宽度
```

- grid

三栏布局

其中一列宽度自适应。

float布局

缺点：html结构不正确,当包含区域宽度小于左右框之和，右边框会被挤下来。float布局需要清除浮动，因为float会脱离文档流，会造成高度塌陷的问题。

两边float+中间margin

- 两变固定，中间自适应
- 中间元素margin值控制两边间距
- 宽度小于左右部分宽度之和时，右侧部分被挤下去

```
<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">

  <title>Title</title>

</head>

<body>

<style>

 .left {

    float: left;

    width: 300px;

    background-color: #a00;

 }


 .right {

    float: right;

    width: 300px;

    background-color: #0aa;

 }


 .center {

    margin: 0 300px;
```

```
    background-color: #aa0;
    overflow: auto;
 }
</style>
<section class="float">
  <article class="left">
    <h1>我是浮动布局左框</h1>
  </article>
  <article class="right">
    <h1>我是浮动布局右框</h1>
  </article>
  <article class="center">
    <h1>我是浮动布局中间框</h1>
  </article>
</section>

</body>
</html>
```

BFC布局

将main变成BFC，就不会和浮动元素发生重叠。

父元素设置overflow: hidden;

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
```

```
<body>
<style>
  *{
    margin: 0;
    padding: 0;
 }
 .left {
    float: left;
    background-color: red;
    width: 100px;
    height: 200px;
 }

 .right {
    float: right;
    background-color: blue;
    width: 100px;
    height: 200px;
 }
 .main{
    background-color: green;
    height: 200px;
    overflow: hidden;
 }
</style>
<div class="container">
```

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

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

  <div class="main">22111111jygjegrearewqrewqewgrhtyhyt</div>

</div>

</body>

</html>
```

table布局

> 优点:表格布局被称之为已经淘汰的布局，包括现在应该也很少有人使用，包括flex布局不兼容的情况下table还可以尝试，类似这种三栏布局，table的实现也很简单，这个东西自我觉得看个人喜爱了，能满足日常使用的话用用也未尝不可。

> 缺点:表格布局相当于其他布局，使用相对繁琐，代码量大，同时也存在缺陷，当单元格的一个格子超出高度之后，两侧就会一起触发跟着一起变高，这显然不是我们想要看到的情况。

```
display:table

<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">

  <title>Title</title>

</head>

<body>

<style>

 .table {

    display: table;

    width: 100%;

 }



 .table > article {
```

```
    display: table-cell;
  }

  .left {
    width: 300px;
    background-color: #a00;
  }

  .center {
    background-color: #aa0;
  }

  .right {
    width: 300px;
    background-color: #0aa;
  }

</style>
<section class="table">
  <article class="left">
    <h1>我是表格布局左框</h1>
  </article>
  <article class="center">
    <h1>我是表格布局中间框</h1>
  </article>
  <article class="right">
```

```
    <h1>我是表格布局右框</h1>
  </article>
</section>


</body>
</html>
```

flex弹性布局

利用flex布局，左右两栏设置固定大小，中间一栏设置为flex:1

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>
<style>
 .flex {
    display: flex;
 }


 .left {
    width: 300px;
    /flex-shrink: 0; ! 不缩小 !/
    background-color: #a00;
 }
```

```
 .center {
    /flex-grow: 1; ! 增大 !/
    flex: 1;
    background-color: #aa0;
 }

 .right {
    /flex-shrink: 0; ! 不缩小 !/
    width: 300px;
    background-color: #0aa;
 }
</style>
<section class=" flex">
  <article class="left">
    <h1>我是flex弹性布局左框</h1>
  </article>
  <article class="center">
    <h1>我是flex弹性布局中间框</h1>
  </article>
  <article class="right">
    <h1>我是flex弹性布局右框</h1>
  </article>
</section>

</body>
</html>
```

grid栅格布局

优点:网格布局作为一个比较超前一点的布局自然有其独特的魅力，其布局方式布局思维都可以让你眼前一亮，有种新的思想。

缺点:兼容性

grid-template-columns 该属性是基于 网格列 的维度，去定义网格线的名称和网格轨道的尺寸大小。

用单位 fr 来定义网格轨道大小的弹性系数。 每个定义了 的网格轨道会按比例分配剩余的可用空间。当外层用一个 minmax() 表示时，它将是一个自动最小值(即 minmax(auto,)) 。

```
<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">

  <title>Title</title>

</head>

<body>

<style>

 .grid {

    display: grid;

    grid-template-columns: 300px 3fr 300px;

 }



 .grid .left {

    background-color: #a00;

 }



 .grid .center {

    background-color: #aa0;

 }
```

```
 .grid .right {
    background-color: #0aa;
 }
</style>
<section class="grid">
  <article class="left">
    <h1>我是grid栅格布局左框</h1>
  </article>
  <article class="center">
    <h1>我是grid栅格布局中间框</h1>
  </article>
  <article class="right">
    <h1>我是grid栅格布局右框</h1>
  </article>
</section>


</body>
</html>
```

圣杯布局

缺点：需要多加一层标签，html顺序不对，占用了布局框的margin属性。

圣杯布局的核心是左、中、右三栏都通过**float**进行浮动，然后通过负值**margin**进行调整。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
```

```
</head>
<body>
<style>
 .container{
    padding-left: 100px;
    padding-right: 100px;
 }
 .left {
    float: left;
    width: 100px;
    height: 200px;
    background-color: red;
    margin-left: -100%;
    position: relative;
    left: -100px;
 }

 .right {
    float: left;
    width: 100px;
    height: 200px;
    background-color: yellow;
    margin-left: -100px;
    position: relative;
    right: -100px;
 }
```

```
 .main {
    float: left;
    width: 100%;
    height: 200px;
    background-color: green;
 }
</style>
<body>
<div class="container">
  <div class="main">发发发发发发返回和王企鹅王企鹅王企鹅而非</div>
  <div class="left"></div>
  <div class="right"></div>
</div>
</body>


</body>
</html>
```

双飞翼布局

双飞翼布局的前两步和圣杯布局一样，只是处理中间栏部分内容被遮挡问题的解决方案有所不同：

既然main部分的内容会被遮挡，**那么就在main内部再加一个content，通过设置其margin来避开遮挡，问题也就可以解决了**

双飞翼布局(两边 float+margin)，就是圣杯布局的改进方案。

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

```
<style>
 .main {
    float: left;
    width: 100%;
 }
 .content {
    height: 200px;
    margin-left: 110px;
    margin-right: 220px;
    background-color: green;
 }
  /CSS伪元素::after用来创建一个伪元素，作为已选中元素的最后一个子元素。/
 .main::after {
    display: block;
    content: '';
    font-size: 0;
    height: 0;
    /唯一需要注意的是，需要在main后面加一个元素来清除浮动。/
    clear: both;
    zoom: 1;
 }
 .left {
    float: left;
    height: 200px;
    width: 100px;
    margin-left: -100%;
```

```
        background-color: red;
    }
    .right {
        width: 200px;
        height: 200px;
        float: left;
        margin-left: -200px;
        background-color: blue;
    }
  </style>
</head>
<body>
<div>
  <div class="main">
    <div class="content"></div>
  </div>
  <div class="left"></div>
  <div class="right"></div>
</div>
</body>
</html>
```

定位布局

优点：很快捷，设置很方便，而且也不容易出问题，你可以很快的就能想出这种布局方式。要求父级要有非static定位，如果没有，左右框容易偏移出去

缺点：绝对定位是脱离文档流的，意味着下面的所有子元素也会脱离文档流，这就导致了这种方法的有效性和可使用性是比较差的。

子绝父相，父元素设置为relative，左右两栏设置为absolute+中间一栏margin。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>
<style>
 .tree-columns-layout {
    position: relative;
 }

 .left {
    position: absolute;
    left: 0;
    top: 0;
    width: 300px;
    background-color: #a00;
 }

 .right {
    position: absolute;
    right: 0;
    top: 0;
    width: 300px;
    background-color: #0aa;
```

```
  }


  .center {
     margin: 0 300px;
     background-color: #aa0;
     overflow: auto;
  }
</style>
<section class="tree-columns-layout">
  <article class="left">
     <h1>我是浮动定位左框</h1>
  </article>
  <article class="center">
     <h1>我是浮动定位中间框</h1>
  </article>
  <article class="right">
     <h1>我是浮动定位右框</h1>
  </article>
</section>


</body>
</html>
```

绝对定位

三栏均设置为absolute+left+right。

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

```
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<style>
    {
    margin: 0;
    padding: 0;
 }
 .middle {
    position: absolute;
    left: 200px;
    right: 200px;
    height: 300px;
    background-color: yellow;
 }

 .left {
    position: absolute;
    left: 0px;
    width: 200px;
    height: 300px;
    background-color: red;
 }

 .right {
```

```
    position: absolute;
    right: 0px;
    width: 200px;
    background-color: green;
    height: 300px;
 }
</style>
<body>
<div class="container">
  <div class="middle">
  </div>
  <div class="left"></div>
  <div class="right"></div>
</div>
</body>


</body>
</html>
```

垂直居中

> top属性会影响定位元素的垂直位置。此属性对非定位元素没有影响。

- 如果**position: absolute;**或**position: fixed; top**属性将元素的上边缘设置为其最近定位的祖先的上边缘上方/下方的单位。
- 如果**position: relative; top**属性使元素的上边缘在其正常位置上方/下方移动。
- 如果**position: sticky; top**当元素在视口内时，属性的行为类似于它的位置，并且当它在外面时它的位置是固定的。
- 如果**position**：**static; top**属性无效。
- **flex布局**

- justify-content: center;
- align-items: center;

居中元素定宽高适用

absolute相对于包含块进行偏移（所谓包含块就是最近一级外层元素**position**不为**static**的元素）。

- absolute+margin auto。因为宽高固定，对应方向实现平分
- absolute+calc
- 利用绝对定位，设置 left: 50% 和 top: 50% 现将子元素左上角移到父元素中心位置，然后再通过 margin-left 和 margin-top 以子元素自己的一半宽高进行负值赋值。

居中元素不定宽高

- absolute+transform
  - 利用绝对定位，先将元素的左上角通过top:50%和left:50%定位到页面的中心，然后再通过margin负值来调整元素的中心点到页面的中心。
  - transform: translate(-50%,-50%);
- line-height:initial默认
- writing-mode:vertical-lr;改变文字的显示方向
- table table-cell
- css-table PC有兼容性要求，宽高不固定，推荐
- flex 兼容性好
- grid + align-items兼容性不行

> top：定位元素的上外边距边界与其包含块上边界之间的偏移，非定位元素设置此属性无效。

水平居中

不定宽高

1. 定位+margin:auto
2. 定位+transform
3. 定位+margin:负值
4. flex布局
5. grid布局

内联元素

水平居中

1. 行内元素可设置：text-align: center
2. flex布局设置父元素：display: flex; justify-content: center

垂直居中

1. 单行文本父元素确认高度：height === line-height
2. 多行文本父元素确认高度：disaply: table-cell; vertical-align: middle

块级元素

水平居中

1. 定宽: margin: 0 auto
2. 绝对定位+left:50%+margin:负自身一半

垂直居中

1. position: absolute设置left、top、margin-left、margin-top(定高)
2. display: table-cell
3. transform: translate(x, y)
4. flex(不定高，不定宽)
5. grid(不定高，不定宽)，兼容性相对比较差

文本溢出

单行溢出

1. text-overflow：当文本溢出时，显示省略符号代表被修剪的文本
2. white-space：设置文字在一行显示，不能换行
3. overflow：文字长度超出限定宽度，隐藏超出的内容

overflow:hidden，普通情况用在块级元素的外层隐藏内部溢出元素，或配合以下两个属性实现文本溢出省略

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

text-overflow属性值如下：

1. clip：对象内文本溢出部分裁掉
2. ellipsis：对象内文本溢出时显示...

多行溢出

基于高度截断

伪元素+定位

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

优点

1. 兼容性好
2. 响应式截断，根据不同宽度做出调整

基于行数截断

margin 合并

margin 合并也叫外边框塌陷或者外边距重叠，注意不同于高度塌陷。

外边距重叠：块的上外边距(margin-top)和下外边距(margin-bottom)有时合并(折叠)为单个边距，其大小为单个边距的最大值(或如果它们相等，则仅为其中一个)，这种行为称为边距折叠。

1、同一层相邻元素之间

```
<p>1 </p>
<p>2 </p>
<style>
p:nth-child(1){
  margin-bottom: 13px;
}
p:nth-child(2){
  margin-top: 12px;
}
</style>
```

这里我们希望的是这两个之间的距离是 25px，但是实际上他们的距离是13px

2、没有内容将父元素和后代元素分开

```
<style type="text/css">
    section    {
        margin-top: 13px;
        margin-bottom: 87px;
    }

    header {
        margin-top: 87px;
    }

    footer {
        margin-bottom: 13px;
    }
</style>

<section>
    <header>上边界重叠 87</header>
    <main></main>
    <footer>下边界重叠 87 不能再高了</footer>
</section>
```

3、空的块级元素

```
<style>
p {
  margin: 0;
}
div {
  margin-top: 13px;
  margin-bottom: 87px;
}
</style>

<p>上边界范围是 87 ...</p>
<div></div>
<p>... 上边界范围是 87</p>
```

margin和padding 的值为百分比时

当 padding 属性值为百分比的时候，如果父元素有宽度，相对于父元素宽度，如果没有，找其父辈元素的宽度，均没设宽度时，相对于屏幕的宽度。

不管 margin-top/margin-bottom 还是 margin-left/margin-right（对于padding 同样）也是，参考的都是 width。

那么为什么是 width， 而不是 height 呢?

CSS权威指南中的解释：

我们认为，正常流中的大多数元素都会足够高以包含其后代元素（包括外边距），如果一个元素的上下外边距时父元素的 height 的百分数，就可能导致一个无限循环，父元素的 height 会增加，以适应后代元素上下外边距的增加，而相应的，上下外边距因为父元素 height 的增加也会增加，如此循环。

重排、重绘和合成

回流一定触发重绘，重绘不一定触发回流。重绘开销小，回流代价高。

回流reflow

也叫重排 layout

渲染树中部分或全部元素的尺寸、结构或属性变化，浏览器会重新渲染部分或全部文档

触发回流的操作:

- 初次渲染
- 窗口大小改变(resize事件)
- 元素属性、尺寸、位置、内容改变
- 元素字体大小变化
- 添加或者删除可见 dom 元素

- 激活 CSS 伪类(如 :hover)
- 查询某些属性或调用某些方法
  - clientWidth、clientHeight、clientTop、clientLeft
  - offsetWidth、offsetHeight、offsetTop、offsetLeft
  - scrollWidth、scrollHeight、scrollTop、scrollLeft
  - getComputedStyle()
  - getBoundingClientRect()
  - scrollTo()

修改样式的时候，**最好避免使用上面列出的属性，他们都会刷新渲染队列**。如果要使用它们，最好将值缓存起来。

重绘 repaint

某些元素的样式如颜色改变，但不影响其在文档流中的位置，浏览器会对元素重新绘制

不再执行布局阶段，直接进入绘制阶段

合成

利用transform、opacity和filter可实现合成效果，即GPU加速

避开布局 分块和绘制阶段

优化

- 最小化重绘和重排：样式集中改变，使用添加新样式类名
- 使用 absolute或 fixed使元素脱离文档流(制作复杂动画时对性能有影响)
- 开启GPU加速。利用css属性transform opacity will-change等，比如改变元素位置，使用**translate会比使用绝对定位改变其left或top更高效**，因为它不会触发重排或重绘，**transform使浏览器为元素创建一个GPU图层，这使得动画元素在一个独立的层中进行渲染，**当元素内容没有改变就没必要进行渲染。
- 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）
- DOM 离线后修改，如：先把 DOM 设为display:none(有一次 Reflow)，然后修改再显示，只会触发一次回流
- 不要把 DOM 结点属性值放在一个循环当成循环里的变量
- 不要使用 table 布局，可能很小的一个小改动会造成整个 table 重新布局
- 动画实现速度的选择，动画速度越快，回流次数越多，也可以选择使用 requestAnimationFrame
- CSS 选择符从右往左匹配查找，避免节点层级过多
- 频繁运行的动画变为图层，图层能够阻止该节点回流影响别的元素。比如对于 video 标签，浏览器会自动将该节点变为图层
- 通过documentFragment创建一个DOM文档片段，在它上面批量操作DOM，完成后再添加到文档中，只触发一次回流

> documentFragment不是真实DOM的一部分，它的变化不会触发DOM树的重新渲染，不会导致性能问题
>
> 效果不甚明显，因为现代浏览器会使用队列存储储存多次修改进行优化

为什么要初始化CSS样式

1. **消除浏览器之间的差异，提高兼容性**
2. **提高代码质量**

第一点：

未初始化时，当我们添加进去一个DIV，会发现他并不是紧贴着窗口的，而是有一定的距离的。这就是一个例子，在不同的浏览器中，对一些标签是具有的默认值的，而且不同的浏览器默认值也肯是不一样的。如果没有对其 CSS 样式做初始化，就可能导致在不同的浏览器之间展示的效果是不一样的。

第二点：

初始化后，便于我们对代码的统一管理，减少重复样式等。同时修改的时候便于统一管理。

div居中的几种方式

方式一

```
position: absolute;
top: 0;
bottom: 0;
left: 0;
right: 0;
margin: auto;
```

方式二
可以给父元素添加下面的属性，利用 flex 布局来实现

```
display: flex;
align-items: center;
flex-direction: column
```

方式三
通过定位和变形来实现
给父元素添加 position: relative;相对定位。
给自身元素添加position: absolute;绝对定位。
top: 50%;使自身元素距离上方“父元素的50%高度”的高度。
left: 50%;使自身元素距离上方“父元素的50%宽度”的宽度。
transform: translate(-50%,-50%);使自身元素再往左，往上平移自身元素的50%宽度和高度。

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

方式四
这个是实现内容文本居中的，坑死了，之前没留意在一个全局的文件加了，后面很多组件里面的内容都居中了，还一时没发现，虽然想到会不会是全局文件的问题，但一下子眼拙没看到，结果捣鼓半天

```
body{ text-align:center}
```

浮动

浮动元素不在文档流中，所以文档流的块框表现得像浮动框不存在

块级元素认为浮动元素不存在，浮动元素会影响行内元素的布局，浮动元素通过行内元素间接影响包含块的布局

浮动元素摆放遵循规则：

1. 尽量靠上
2. 尽量靠左
3. 尽量一个挨一个
4. 可能超出包含块
5. 不能超过所在行的最高点
6. 行内元素绕着浮动元素摆放

带来的问题

1. 父元素的高度无法被撑开
2. 与浮动元素同级的非浮动元素会紧随其后

清除浮动

1. 父级div定义height
2. 最后一个浮动元素加空div标签，添加样式clear:both（不推荐）
3. 父级div定义zoom
4. 父级添加overflow：hidden/auto（不推荐）
5. 浮动元素的容器添加浮动（不推荐，会使整体浮动，影响布局）
6. after伪元素清除浮动，尾部添加一个看不见的块元素清理浮动，设置clear:both（推荐）
7. before和after双伪元素清除浮动

总结 3 种

1、clear属性

2、BFC

3、::after伪元素

BFC

> 块格式化上下文（Block Formatting Context，BFC） 是Web页面的可视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。

- BFC是一个独立的布局环境，与外部互不影响，用于决定块级盒的布局及浮动相互影响范围的一个区域。
- 块盒和行盒(行盒由一行中所有的内联元素所组成)都会垂直沿着其父元素的边框排列。

除了 BFC，还有：

- IFC（行级格式化上下文）- inline 内联
- GFC（网格布局格式化上下文）- display: grid
- FFC（自适应格式化上下文）- display: flex或display: inline-flex

注意：同一个元素不能同时存在于两个 BFC 中。

创建

- 根元素
- float 值为 left 、right
- display 值为 inline-block、table-cell、table-caption、table、inline-table、flex、inline-flex、grid、inline-grid
- 绝对定位元素：position 值为 absolute、fixed
- overflow 值不为 visible，即为 auto、scroll、hidden

特点

- 是页面上的一个独立容器,容器里面的子元素**不会影响外面的元素**
- 垂直方向上，自上而下，**与文档流排列方式一致**
- **同一 BFC 下的相邻块级元素可能发生margin折叠，创建新的 BFC 可以避免外边距折叠**
- BFC区域**不会与浮动容器发生重叠**
  - 两栏布局
  - float + overflow:hidden
- 计算BFC高度时，**浮动元素参与计算**
- 每个元素的左margin和容器的左border相接触,即,每个元素的外边距盒（margin box）的左边与包含块边框盒（border box）的左边相接触（从右向左的格式的话，则相反），即使存在浮动

应用

- 解决margin重叠问题
  - 块级元素的上外边距和下外边距有时会合并（或折叠）为一个外边距，其大小取其中的较大者，这种行为称为**外边距折叠（重叠）**，注意这个是发生在属于同一 BFC 下的块级元素之间
- **把两个元素变成两个BFC**
- **自适应两栏布局**
- **左列浮动**。右列设置 overflow: hidden; 触发BFC , float + overflow:hidden
- 父子元素的外边距重叠
  - 如果在父元素与其第一个/最后一个子元素之间不存在边框、内边距、行内内容，也没有创建块格式化上下文、或者清除浮动将两者的外边距 分开，此时**子元素的外边距会“溢出”到父元素的外面**。
- 解决
  - 父元素触发BFC
  - 父元素添加border
  - 父元素添加padding
- 清除浮动，解决父元素高度塌陷
  - 当容器内子元素设置浮动时，脱离了文档流，容器中总父元素高度只有边框部分高度。

background-size

设置背景图片大小。图片可以保有其原有的尺寸、拉伸到新的尺寸，或者在保持原有比例的同时缩放到元素的可用空间的尺寸

```
div
{
    width: 300px;
    height: 200px;
    background: url("../assets/2.jpg")no - repeat;
    border: 1px solid red;
    /*background-size: 100%;*/
    background - size: cover;
}
/*若图片宽度250px，宽度为250px，让该图片完全铺满整个div区域，设置background-size*/
```

属性

100%：整个图片**铺满div**

cover：整个图片**铺满div**，缩放背景图片以完全覆盖背景区，可能背景图片部分看不见。和 contain 相反，cover 尽可能大地缩放背景图像并保持图像的宽高比例（图像不会被压扁）。背景图以它的全部宽或者高覆盖所在容器。当容器和背景图大小不同时，背景图的 左/右 或者 上/下 部分会被裁剪

contain：**不能铺满**整个div，缩放背景图片以完全装入背景区，可能背景区部分空白。contain 尽可能地缩放背景并保持图像的宽高比例（图像不会被压缩）。背景图会填充所在容器。当背景图和容器的大小的不同时，容器的空白区域（上/下或者左/右）显示由 background-color 设置的背景颜色

auto：**不能铺满**整个div，以背景图片比例缩放背景图片

block，inline，inline-block

块级元素：自动占据一行，可以设置宽高

常见的有 div，p，h1-h6，ul，li，form，table

行内元素：占据一行的一小部分，多个行内元素水平排版，无法设置宽高

常见的有 span ，img，a

行内块级元素：跟行内元素类似，不过可以设置宽高

常见的有 button ， img ， input, select, label，textarea

空元素:img,br,input,link,meta

line-height 如何继承

- 父元素的 line-height 写了**具体数值**，比如 30px，则子元素 line-height 继承该值。
- 父元素的 line-height 写了**比例**，比如 1.5 或 2，则子元素 line-height 也是继承该比例。
- 父元素的 line-height 写了**百分比**，比如 200%，则子元素 line-height 继承的是父元素 fontSize * 200% 计算出来的值。

client、offset&scroll

client 主要与可视区有关

客户区大小指的是元素内容及其内边距所占据空间大小。

offset 主要与自身有关

偏移量，可动态得到元素的位置（偏移），大小。元素在屏幕上占用的所有可见空间。元素高度宽度包括内边距，滚动条和边框。

offsetParent是一个只读属性，返回一个指向最近的（closest，指包含层级上的最近）包含该元素的定位元素。如果没有定位的元素，则 offsetParent 为最近的 table, td, th或body元素。当元素的 style.display 设置为 “none” 时，offsetParent 返回 null。

- element.clientWidth获取元素可视区的宽度，不包括垂直滚动条
- element.offsetWidth获取元素的宽度= **boder + padding + content**
- element.clientHeight获取元素可视区的高度，不包括水平滚动条
- element.offsetHeight获取元素的高度= **boder + padding + content**

- **clientWidth 和 clientHeight 获取的值不包含边框**
- **offsetWidth 和 offsetHeight 获取的值包含左右边框**

- element.clientTop获取元素的上边框宽度，不包括顶部外边距和内边距
- element.clientLeft获取元素的左边框宽度
- element.offsetTop获取元素到有定位的父盒子的顶部距离
- element.offsetLeft获取元素到有定位的父盒子的左侧距离
- e.clientX鼠标距离可视区的左侧距离
- e.clientY鼠标距离可视区的顶部距离

scroll 滚动系列

动态获得元素大小，滚动距离等。具有兼容问题。

- scrollWidth 和 scrollHeight 主要用于确定元素内容的实际大小
- scrollLeft 和 scrollTop 属性既可以确定元素当前滚动的状态，也可以设置元素的滚动位置
- 垂直滚动 scrollTop > 0
- 水平滚动 scrollLeft > 0
- 将元素的 scrollLeft 和 scrollTop 设置为 0，可以重置元素的滚动位置

共同点

返回数字时，均不带单位。

只读。

z-index

> 取值

数字无单位

- **auto(默认)和其父元素一样的层叠等级**
- 整数
- **inherit继承**

> 层叠上下文和层叠等级

对每一个网页来说，默认创建一个层叠上下文(可类比一张桌子)，这个桌子就是html元素，html元素的所有子元素都位于这个默认层叠上下文中的某个层叠等级(类似于桌子上放了一个盆儿，盆上放了一个果盘，果盘上放了一个水杯……)

将元素的z-index属性设置为非auto时，会创建一个新的层叠上下文，它及其包含的层叠等级独立于其他层叠上下文和层叠等级(类似于搬了一张新桌子过来，和旧桌子完全独立)

> 层叠顺序

div默认在html之上，及div的层级高于html

一个层叠上下文可能出现7个层叠等级，从低到高排列为

1. 背景和边框
2. z-index为负数
3. block盒模型(位于正常文档流，块级，非定位)
4. float盒模型(浮动，非定位)
5. inline盒模型(位于正常文档流，内联，非定位)
6. z-index=0
7. z-index为正数(数值越大越靠上方)

> 压盖顺序

自定义压盖顺序：使用属性z-index

- 只有定位元素(**position属性明确设置为absolute、fixed或relative**)可使用z-index
- 如果z-index值相同，html结果在后面的压盖住在前面的
- 父子都有z-index，父亲z-index数值小时，儿子数值再大没用

link标签的伪类和用法

```
<div id="content">

  <h3>

   <a class="a1" href="#">

    a标签伪类link, hover, active, visited, focus区别

   </a>
```

```
 </h3>
</div>
<style>
a.a1:link{  /*链接未被访问时的状态*/
 color: blue;
}
a.a1:visited{ /*链接访问后的状态*/
 color: green;
}
a.a1:hover{  /*鼠标悬浮的状态*/
 color: red;
}
a.a1:focus{  /*鼠标点击后，刚松开的状态*/
 color: orange;
}
a.a1:active{  /*点击中的状态，鼠标未松开时*/
 color: yellow;
}
</style>
```

a:link 未设置超链接则无效

a:visited 针对URL，若两个a标签指向一个链接，点击一个另一个也有visited属性

选择器权重Specificity

如何比较两个优先级的高低呢？ 比较规则是: 从左往右依次进行比较 ，较大者胜出，如果相等，则继续往右移动一位进行比较 。如果4位全部相等，则后面的会覆盖前面的。

- **内联>id>类=属性=伪类>标签(类型、元素选择器h1)=伪元素**

> 内联样式表的权值最高为 1000；

> ID 选择器的权值为 100
>
> Class 类选择器、属性选择器、伪类的权值为 10
>
> HTML 元素选择器、伪元素的权值为 1
>
> 加有!important的权值最大，优先级最高

需要注意权值计算要基于选择器的形式。特别是，“[id=p33]”形式的选择器被视为属性选择器(权值为10)，即使id属性在源文档的文档类型中被定义为“id选择器”。

- **通配符 * 和关系选择符(+ > ~ " ' |)和否定伪类(:not()) 对优先级没有影响，但是:not内部声明的选择器会影响优先级。**

```
*{} /*通用选择器，权值为0 */

p{color:red;} /*标签，权值为1*/

p span{color:green;} /*两个标签，权值为1+1=2*/

p>span{color:purple;}/*权值与上面的相同，因此采取就近原则*/

a:hover{}/*标签和伪类，权值为1+10=11*/

.warning{color:white;} /*类选择符，权值为10*/

p span .warning{color:purple;} /*权值为1+1+10=12*/

h1+a[rel=up]{}/*标签和属性选择器，权值为1+10=11*/

#footer .note p{color:yellow;} /*权值为100+10+1=111*/

p{color:red!important; }  /*优先级最高*/
```

注意事项

- !important优先级最高。避免使用，会破坏样式表中固有的级联规则！！
- 样式表来源不同时 优先级顺序为：内联>内部>外部>浏览器用户自定义>浏览器默认。
- 兄弟选择器 li ~ a
- 相邻兄弟选择器 li + a
- 属性选择器 a[rel="external"]
- 伪类选择器 a:hover, li:nth-child
- 通配选择器 *{}
- 类型选择器h1{}

:link： 选择未被访问的链接

:visited：选取已被访问的链接

:active：选择活动链接

:hover： 鼠标指针浮动在上面的元素

:focus： 选择具有焦点的

:first-child：父元素的首个子元素

伪元素选择器 ::before、::after

::first-letter： 用于选取指定选择器的首字母

::first-line： 选取指定选择器的首行

::before : 选择器在被选元素的内容前面插入内容

::after : 选择器在被选元素的内容后面插入内容

单位

px % em 这几个单位，可以适用于大部分项目开发，且拥有较好兼容性

- **px**： 一个固定像素单位，一个像素表示终端屏幕能显示的最小的区域
- %： 元素的宽高可随浏览器的变化而变化，实现响应式，一般子元素的百分比相对于直接父元素
- **em**： 作为 font-size 的单位时，代表父元素的字体大小按比例计算值，作为其他属性单位时，代表相对自身字体大小按比例计算值

```
.parent {
font-size:32px;
}
/** child字体为16px **/
.child {
font-size:.5em;
width:2em;
/* 32 * 0.5 * 2 */}}
```

- **rem**： CSS3新增。相对于根元素字体大小按比例计算值；作用于根元素字体大小时，相对于其出初始字体大小（16px）

```
html {
font-size:20px;
}
/* 作用于非根元素，相对于根元素字体大小，所以为40px */
p {
font-size:2rem;
}
```

- **vw** 相对于视图窗口宽度，视窗宽度为100vw
- **vh** 相对于视图窗口高度，视窗高度为100vh
- **vm**
- **rpx**

rpx是微信小程序独有的，解决屏幕自适应的尺寸单位

responsive pixel（动态像素）

可以根据屏幕宽度进行自适应，无论屏幕大小，规定屏幕宽度为750rpx

通过rpx设置元素和字体大小，可实现小程序在不同尺寸的屏幕上自动适配

rpx和px的换算

> iPhone6的屏幕宽度为375px，有750个物理像素，则750rpx=375px=750个物理像素

1px=2rpx

实现1px效果

1. 伪元素+缩放
2. 动态viewport+rem(flex)
3. vw单位适配(未来推荐)

伪元素+缩放

设计稿中的1px，代码要实现0.5px

缩放 避免 直接写小数像素带来的不同手机的兼容性处理不同

```
<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">

  <title>Title</title>

  <style>

   /*伪元素实现0.5px border*/

   .border::after {

    content: "";

    /*为了与原元素等大*/

    box-sizing: border-box;

    position: absolute;

    left: 0;

    top: 0;
```

```
 width: 200%;
 height: 200%;
 border: 1px solid red;
 transform: scale(0.5);
 transform-origin: 0 0;
}

/*实现0.5px 细线*/
.line::after {
 content: '';
 position: absolute;
 top: 0;
 left: 0;
 width: 200%;
 height: 1px;
 background: red;
 transform: scale(0.5);
 /*更改元素变形的原点*/
 transform-origin: 0 0;
}

/*dpr适配 ， 当前显示设备的物理像素分辨率与CSS 像素分辨率之比为2*/
@media (-webkit-min-device-pixel-ratio: 2) {
 .line::after {
  height: 1px;
  transform: scale(0.5);
```

```
      transform-origin: 0 0;
     }
    }

    @media (-webkit-min-device-pixel-ratio: 3) {
     .line::after {
      height: 1px;
      transform: scale(0.333);
      transform-origin: 0 0;
     }
    }
  </style>
</head>
<body>
<div class="border">
  <div class="line"></div>
</div>
</body>
</html>
```

动态viewport+rem

不仅可解决移动端适配，也解决1px的问题

三种viewport中我们常用的layout viewport(浏览器默认)，宽度大于浏览器可视区域宽度，因此会出现横向滚动条

```
const clientWidth = document.documentElement.clientWidth || document.body.clientWidth
```

设置meta标签属性避免横向滚动条

```
 < meta
name = "viewport"

    content = "
    width=device-width,  // viewport宽等于屏幕宽
    initial-scale=1.0,  // 初始缩放为1
    maximum-scale=1.0,
    user-scalable=no,  // 不允许手动缩放
    viewport-fit=cover // 缩放以填充满屏幕
    "
    >
```

flexible的原理——已弃用！

1. 根据dpr动态修改initial-scale
2. 动态修改viewport大小，以此 统一使用rem布局，viewport动态影响font-size，实现适配

总结

移动端适配主要分为两方面

1. 适配不同机型的屏幕尺寸
2. 对细节像素的处理。如果直接写 1px ，由于 dpr 的存导致渲染偏粗。使用rem 布局计算出对应小数值，有兼容性问题。老项目整体修改 viewport 成本过高，采用第一种实现方案处理；新项目可动态设置 viewport ，一键解决适配问题

移动端对 1px 的渲染适配实现起来配置简单、代码简短，能够快速上手

使chrome支持12px以下文字

在谷歌浏览器中，字体的最小值为 12px，当你设置字体为 10px 的时候，结果显示的还是12px，这是因为 Chrome 浏览器做了如下限制：

- font-size 有一个最小值 12px（不同操作系统、不同语言可能限制不一样），低于 12px 的，一律按 12px 显示。
- 理由是 Chrome 认为低于 12px 的中文对人类是不友好的。但是你可以设置为 0

zoom

"变焦"，可以改变页面上元素的尺寸，属于真实尺寸。有兼容问题，非标准属性，缩放会改变元素占据空间大小，触发重排

- zoom:50%，表示缩小到原来的一半
- zoom:0.5，表示缩小到原来的一半

```
.test1 {
 font-size: 10px;
 zoom: 0.8;
}
.test2 {
 font-size: 16px;
}
```

-webkit-transform:scale()

针对Chrome使用webkit前缀

使用scale属性只对可以定义宽高的元素生效。不改变页面布局

```
.test1 {
 font-size: 5px;
 display: inline-block;
 transform: scale(0.8);
}
.test2 {
 font-size: 16px;
 display: inline-block;
}
```

-webkit-text-size-adjust:none

设定文字大小是否根据设备(浏览器)来自动调整显示大小

- percentage：字体显示的大小；
- auto：默认，字体大小会根据设备/浏览器来自动调整；
- none:字体大小不会自动调整

CSS3 新特性

1. 新增选择器： p:nth-child（n）{color: rgba（255, 0, 0, 0.75）}

2. 弹性盒模型：
   弹性布局： display: flex;
   栅格布局： display: grid;
3. 渐变
   线性渐变： linear-gradient（red, green, blue）；
   径向渐变：radial-gradient（red, green, blue）
4. 边框
   border-radius：创建圆角边框
   box-shadow：为元素添加阴影
   border-image：使用图片来绘制边框
5. 阴影
   box-shadow:3px 3px 3px rgba（0, 64, 128, 0.3）；
6. 背景
   用于确定背景画区：background-clip
   设置背景图片对齐：background-origin
   调整背景图片的大小background-size
   控制背景怎样在这些不同的盒子中显示：background-break
   多列布局： column-count: 5;
7. text-overflow
   文字溢出时修剪：text-overflow：clip
   文字溢出时省略符号来代表：text-overflow：ellipsis
8. transform 转换
   transform: translate(120px, 50%)：位移
   transform: scale(2, 0.5)：缩放
   transform: rotate(0.5turn)：旋转
   transform: skew(30deg, 20deg)：倾斜
9. animation 动画
   animation-name：动画名称
   animation-duration：动画持续时间
   animation-timing-function：动画时间函数
   animation-delay：动画延迟时间
   animation-iteration-count：动画执行次数，可以设置为一个整数，也可以设置为infinite，意思是无限循环
   animation-direction：动画执行方向
   animation-paly-state：动画播放状态
   animation-fill-mode：动画填充模式

还有多列布局、媒体查询（@media）、混合模式等等，而CSS3如今有很多是我们日常使用到的，比如我们在处理文字溢出时会使用 text-overflow ，比如我们需要文字突破限制12像素就可以使用 transform: scale（0.5）来调整。

flex布局

弹性布局。可以 简便、完整、响应式地实现各种页面布局。为盒模型提供最大的灵活性。

> 父容器和子容器构成，通过主轴和交叉轴控制子容器排列布局，子元素float、clear和vertical-align属性失效

父容器

- flex-direction

- flex-wrap
- flex-flow
- justify-content
- align-items
- align-content

flex-direction

.container {

flex-direction: row | row-reverse | column | column-reverse;

}

决定主轴方向，子元素的排列方向

flex-wrap

.container {

flex-wrap: nowrap | wrap | wrap-reverse;

}

弹性元素永远沿主轴排列，那么如果主轴排不下，通过flex-wrap决定容器内项目是否可换行。

默认情况是不换行，但这里也不会任由元素直接溢出容器，会涉及到元素的弹性伸缩。

flex-flow

是flex-direction属性和flex-wrap属性的简写形式，默认值为row nowrap

当flex-grow之和小于1时，只能按比例分配部分剩余空间，而不是全部。

.box {

flex-flow: || ;

}

justify-content

定义了项目在主轴上的对齐方式

.box {

justify-content: flex-start | flex-end | center | space-between | space-around;

}

- flex-start（默认值）：左对齐
- flex-end：右对齐
- center：居中
- space-between：两端对齐，项目之间的间隔都相等
- space-around：两个项目两侧间隔相等

align-items

.box {

align-items: flex-start | flex-end | center | baseline | stretch;

}

定义项目在交叉轴上如何对齐

- flex-start：交叉轴的起点对齐
- flex-end：交叉轴的终点对齐
- center：交叉轴的中点对齐
- baseline: 项目的第一行文字的基线对齐
- stretch（默认值）：如果项目未设置高度或设为auto，将占满整个容器的高度

align-content

.box {

align-content: flex-start | flex-end | center | space-between | space-around | stretch;

}

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

- flex-start：与交叉轴的起点对齐
- flex-end：与交叉轴的终点对齐
- center：与交叉轴的中点对齐
- space-between：与交叉轴两端对齐，轴线之间的间隔平均分布
- space-around：每根轴线两侧的间隔都相等。所以，轴线之间的间隔比轴线与边框的间隔大一倍
- stretch（默认值）：轴线占满整个交叉轴

子容器

- order
- flex-grow
- flex-shrink
- flex-basis
- flex
- align-self

order

定义item排列顺序，越小越靠前，默认0

flex-grow

定义项目的放大比例（容器宽度>元素总宽度时如何伸展）

默认为0，即如果存在剩余空间，也不放大。

flex-warp：nowrap；不换行的时候，container宽度不够分时，弹性元素会根据flex-grow来决定。

如果所有项目的flex-grow属性都为1，则它们将等分剩余空间（如果有的话）

如果一个项目的flex-grow属性为2，其他项目都为1，则前者占据的剩余空间将比其他项多一倍

弹性容器的宽度正好等于元素宽度总和，无多余宽度，此时无论flex-grow是什么值都不会生效。

flex-shrink

定义了项目的缩小比例（容器宽度<元素总宽度时如何收缩），默认为1，即如果空间不足，该项目将缩小。flex 元素仅在默认宽度之和大于容器的时候才会发生收缩，其收缩的大小是依据 flex-shrink 的值。

> 如果所有项目的flex-shrink属性都为1，当空间不足时，都将等比例缩小。
>
> 如果一个项目的flex-shrink属性为0，其他项目都为1，则空间不足时，前者不缩小。

flex-basis

定义分配多余空间前，占据主轴空间 默认auto。 元素在主轴上的初始尺寸，所谓的初始尺寸就是元素在flex-grow和flex-shrink生效前的尺寸

flex:1

等分剩余空间

flex: 1 相当于 flex-grow: 1、flex-shrink: 1 和 flex-basis: 0%。

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

优先使用这个属性，而不是单独写三个分离的属性，因为浏览器会推算相关值。

flex: 1 = flex: 1 1 0%

flex: 2 = flex: 2 1 0%

flex: auto = flex: 1 1 auto

flex: none = flex: 0 0 auto，常用于固定尺寸不伸缩

aline-self

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。

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

左边固定右边自适应

//垂直居中

flex-direction:row/column;

align-items:center

//水平局中

flex-direction:row/column;

justify-content:center;

flex: 1 = flex: 1 1 0%

flex: 2 = flex: 2 1 0%

flex: auto = flex: 1 1 auto

flex: none = flex: 0 0 auto，常用于固定尺寸不伸缩

隐藏元素

display:none

- 不会在页面占据位置
- 渲染树不会包含该渲染对象
- 不会绑定响应事件
- 会导致浏览器进行重排和重绘

visibility:hidden

- 占据位置,不更改布局
- 不会响应绑定事件
- 不会重排但会重绘

opacity:0

- 元素透明度设置为0
- 占据位置
- 能响应绑定事件
- 不能控制子元素展示
- 不会引发重排，一般会引发重绘

> 如果利用 animation 动画，对 opacity 做变化（animation会默认触发GPU加速），则只会触发 GPU 层面的 composite，不会触发重绘

设置height width为0

将影响元素盒模型的属性设置为0，若有元素内有子元素或内容，应该设置其overflow:hidden来隐藏其子元素。

- 元素不可见
- 不占据空间
- 不响应点击事件

position：absolute

- 将元素移除可视区域
- 元素不可见
- 不影响页面布局

transform: scale(0,0)

- 占据位置
- 不响应绑定事件
- 不会触发浏览器重排

页面中不存在存在存在重排会不会不会重绘会会不一定自身绑定事件不触发不触发可触发transition不支持支持支持子元素可复原不能能不能被遮挡的元素可触发事件能能不能**opacity和rgba区别**

opacity 取值在0到1之间，0表示完全透明，1表示完全不透明。

.aa{opacity: 0.5;}

rgba中的R表示红色，G表示绿色，B表示蓝色，三种颜色的值都可以是正整数或百分数。A表示Alpha透明度。取值0~1之间，类似opacity。

.aa{background: rgba(255,0,0,0.5);}

rgba()和opacity都能实现透明效果，但最大的不同是opacity作用于元素，以及元素内的所有内容的透明度，而rgba()只作用于元素的颜色或其背景色。

总结：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。

响应式布局

页面的设计和开发根据用户行为和设备环境进行调整和响应

Content is like water

```
<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1,
user-scalable=no”>
```

1. width=device-width: 自适应手机屏幕的尺寸宽度
2. maximum-scale:缩放比例的最大值
3. inital-scale:缩放的初始化
4. user-scalable:用户可以缩放

实现响应式布局的方式

1. 媒体查询
2. 百分比
3. vw/vh
4. rem

响应式设计实现通常会从以下几方面思考：

1. 弹性盒子和媒体查询等
2. 百分比布局创建流式布局的弹性UI，同时使用媒体查询限制元素的尺寸和内容变更范围
3. 相对单位使得内容自适应调节

缺点：

1. 仅适用布局、信息、框架并不复杂的部门类型网站
2. 兼容各种设备工作量大，效率低下
3. 代码累赘，出现隐藏无用的元素，加载时间加长
4. 一定程度上改变了网站原有的布局结构

过渡与动画

javascript直接实现

JS实现动画导致页面频繁性重排重绘，消耗性能.

SVG（可伸缩矢量图形）

1. 控制动画延时
2. 控制属性的连续改变
3. 控制颜色变化
4. 控制如缩放,旋转等几何变化
5. 控制SVG内元素的移动路径

SVG是对图形的渲染，HTML是对DOM的渲染

CSS3 transition

transition是过度动画。但transition不能实现独立的动画，只能在某个标签元素样式或状态改变时进行平滑的动画效果过渡，而不是马上改变

注意在移动端开发中，直接使用transition动画会让页面变慢甚至卡顿。所以我们通常添加transform:translate3D(0,0,0)或transform:translateZ(0)来开启移动端动画的GPU加速，让动画过程更加流畅

> transition:transform 1s ease;
> style="transform: translate(304px, 256px);"

动态改变transform的值，实现拖拽移动的效果

CSS3 animation3

animation 算是真正意义上的CSS3动画。通过对关键帧和循环次数的控制，页面标签元素会根据设定好的样式改变进行平滑过渡

比较CSS3最大的优势是摆脱了js的控制，并且能利用硬件加速以及实现复杂动画效果

Canvas

canvas作为H5新增元素，借助Web API来实现动画

只有width和height两个属性

requestAnimationFrame

requestAnimationFrame是另一种Web API，执行动画的效果，会在一帧(一般是16ms)间隔内根据选择浏览器情况执行相关动作

raf按照系统刷新的节奏调用！！

在性能上比另两者要好。

通常，我们将执行动画的每一步传到requestAnimationFrame中，在每次执行完后进行异步回调来连续触发动画效果。

JavaScript 动画 和 CSS动画

在前端中实现动画的方式有两种，也就是 JavaScript动画 和 CSS动画。

一、JavaScript 动画

JavaScript 动画就是通过对元素样式进行渐进式变化编程完成的。这种变化通过一个计数器来调用。当计数器间隔很小时，动画看上去就是连贯的。 实践中一般需要设置 style = “position: relative” 创建容器元素。通过 style = “position: absolute” 创建动画元素。然后通过定时器（绘图函数）来控制动画元素的变化，也就是按照我们的规则来修改CSS样式。

二、CSS 动画

与 JavaScript动画使用定时器修改不同，CSS动画是通过指定 @keyframes 来指定动画效果，然后绑定到需要实习的元素上。

一般有属性：

`@keyframes`：规则中指定了 CSS 样式，动画将在特定时间逐渐从当前样式更改为新样式。
`animation-name`：用来绑定动画规则
`animation-duration`：定义需要多长时间才能完成动画
`animation-delay`：规定动画开始的延迟时间。也就是多久后才开始动画
`animation-iteration-count`：指定动画应运行的次数。
`animation-direction`：指定是向前播放、向后播放还是交替播放动画。
`animation-timing-function`：规定动画的速度曲线。
`animation-fill-mode`：SS 动画不会在第一个关键帧播放之前或在最后一个关键帧播放之后影响元素。
animation-fill-mode 属性能够覆盖这种行为。
`animation`：实现与上例相同的动画效果，也就是类似我们的 font 可以包含所有的相关属性

```
//下面的例子将 "example" 动画绑定到 <div> 元素。动画将持续 4 秒钟，同时将 <div> 元素的背景颜色从
"red" 逐渐改为 "yellow":
/* 动画代码 */
 @ keyframes example {
    from {
        background - color: red;
    }
    to {
        background - color: yellow;
    }
}

/* 向此元素应用动画效果 */
div {
    width: 100px;
    height: 100px;
    background - color: red;
    animation - name: example;
    animation - duration: 4s;
}
```

三、JavaScript 动画 和 Css动画优缺点

JS动画

优点：

1. **过程可控**，在动画中可以实现任何效果，暂停，返回，加速等等
2. **动画效果丰富**，可以根据绘图函数实现任意效果，跳跳球，变速等
3. **兼容性好**，使用 CSS3 存在兼容问题，但是 JavaScript 几乎没有

缺点：

1. 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。
2. 代码的复杂度高于CSS动画。

CSS动画

优点（浏览器可以对动画进行优化）：

1. 集中所有DOM，一次重绘重排，刷新频率和浏览器刷新频率相同。
2. 代码简单，方便调试
3. 不可见元素不参与重排，节约CPU
4. 可以使用硬件加速（通过 GPU 来提高动画性能）。

缺点：

1. 运行过程控制较弱，无法附加事件绑定回调函数。
2. 代码冗长。

SCSS 常见的属性

1、 **嵌套**

```
$baseColor: red;
$color1: green;
body{
  background-color: $baseColor;
  $color2: yellow;
  div{
    background-color:$color2;
    color: $color1;
  }
   div2{
    background-color:$color2;
    color: $color1;
  }
}
//编译后
body {
  background-color: red;
   }
  body div {
```

```
    background-color: yellow;
    color: green;
   }
 body div2 {
   background-color: yellow;
   color: green;
}
```

2、 **定义变量**

```
$baseColor: red;
$color1: green;
body{
  background-color: $baseColor;
}

div{
  background-color:$baseColor;
  color: $color1;
}
//编译出的结果
body {
  background-color: red; }

div {
  background-color: red;
  color: green; }
```

3、 **混合**

```
@mixin box-sizing($sizing) {
  -webkit-box-sizing: $sizing;
  -moz-box-sizing: $sizing;
  box-sizing: $sizing;
}

.box-border{
  border: 1px  solid red;
  @include box-sizing(border-box)
}

//编译结果
.box-border {
  border: 1px  solid red;
  -webkit-box-sizing: border-box;
  -moz-box-sizing: border-box;
  box-sizing: border-box;
  }
```

4、 **继承**

```
.A{
  color: red;
  border: #0D47A1 2px solid;
  background-color: #0e0e0e;
}
.B{
  @extend .A;
}
.C{
  @extend .A;
  border-color: #00a507;
}
//编译结果
.A, .B, .C {
  color: red;
  border: #0D47A1 2px solid;
  background-color: #0e0e0e;
  }

.C {
  border-color: #00a507;
   }
```

JavaScript

数组 Array

创建数组

Array.from() 浅拷贝

```
const dp = Array(5).fill();
const dp = Array.from(Array(m + 1), () => Array(n + 1).fill(0));
```

从类数组对象或者可迭代对象中创建一个新的数组实例。 Array.from还可以接受第二个参数，作用类似于数组的map方法，用来对每个元素进行处理，将处理后的值放入返回的数组。

Array.of()

根据一组参数来创建新的数组实例，支持任意的参数数量和类型，没有参数时返回 []，当参数只有一个的时候，实际上是指定数组的长度。

sort原理

对数组进行排序，默认排序顺序规则是将元素转换为字符串，然后比较它们的 UTF-16 代码单元值序列时构建的 。

copyWithin()

将指定位置的成员复制到其他位置（会覆盖原有成员），然后返回当前数组。

- target（必需）：从该位置开始替换数据。如果为负值，表示倒数。
- start（可选）：从该位置开始读取数据，默认为 0。如果为负值，表示从末尾开始计算。
- end（可选）：到该位置前停止读取数据，默认等于数组长度。如果为负值，表示从末尾开始计算。

```
[1, 2, 3, 4, 5].copyWithin(0, 3);
// 将从 3 号位直到数组结束的成员（4 和 5），复制到从 0 号位开始的位置，结果覆盖了原来的 1 和 2
// [4, 5, 3, 4, 5]
```

find()

find()用于找出第一个符合条件的数组成员

参数是一个回调函数，接受三个参数依次为当前的值、当前的位置和原数组

findIndex()

findIndex返回第一个符合条件的数组成员的位置，如果所有成员都不符合条件，则返回-1

find和findIndex这两个方法都可以接受第二个参数，用来绑定回调函数的this对象。

fill()

还可以接受第二个和第三个参数，用于指定填充的起始位置和结束位置.

注意，如果填充的类型为对象，则是浅拷贝.

对原数组有影响

push

返回数组最新长度

unshift()

返回新数组长度

splice

返回空数组，返回包含删除元素的数组

```
[2, 3, 4].splice(0, 1); // 0位置删除一个，返回[2]

[2, 3, 4].splice(0, 1, 5); // 0位置删除1个，插入5，原数组是[5, 3, 4]，返回[2]

arrayObject.splice(index,howmany,item1,.....,itemX)
```

index : 必需。添加/删除项目位置，使用负数可从数组结尾处规定位置。

howmany :必需。要删除项目数量。如果设置为 0，则不会删除项目。

item1, ..., itemX: 可选。向数组添加新项目。

pop()

shift()

对原数组无影响

concat()

创建一个副本，返回新构建的数组

slice()

创建一个包含原有数组中一个或多个元素的新数组

reduce

reduce() 方法不会改变原始数组。

filter

将所有元素进行判断，将满足条件的元素作为一个新的数组返回

some

将所有元素进行判断返回一个布尔值，如果存在元素都满足判断条件，则返回 true，若所有元素都不满足判断条件，则返回 false：

every

将所有元素进行判断返回一个布尔值，如果所有元素都满足判断条件，则返回 true，否则为 false：

求最大值

```
var a=[1,2,3,4];

Math.max.apply(null, a);
```

join

纯函数

flat()，flatMap()

将数组扁平化处理，返回一个新数组，**对原数据没有影响**。

flat()默认只会“拉平”一层，如果想要“拉平”多层的嵌套数组，可以将flat()方法的参数写成一个整数，表示想要拉平的层数，默认为1。

flatMap()方法对原数组的每个成员执行一个函数相当于执行Array.prototype.map()，然后对返回值组成的数组执行flat()方法。该方法返回一个新数组，不改变原数组。

flatMap()方法还可以有第二个参数，用来绑定遍历函数里面的this

Set集合

Set 对象允许我们存储任何类型的唯一值，无论是原始值或者是对象引用。

```
let mySet = new Set();

mySet.add(1); // Set [ 1 ]

mySet.add(5); // Set [ 1, 5 ]

mySet.add(5); // Set [ 1, 5 ]

mySet.add("some text"); // Set [ 1, 5, "some text" ]

let o = {a: 1, b: 2};

mySet.add(o);

mySet.add({a: 1, b: 2}); // o 指向的是不同的对象，所以没问题

mySet.has(1); // true

mySet.has(3); // false

mySet.has(5);       // true

mySet.has(Math.sqrt(25)); // true

mySet.has("Some Text".toLowerCase()); // true

mySet.has(o); // true

mySet.size; // 5

mySet.delete(5); // true, 从set中移除5

mySet.has(5);  // false, 5已经被移除

mySet.size; // 4, 刚刚移除一个值

console.log(mySet);

// logs Set(4) [ 1, "some text", {…}, {…} ] in Firefox

// logs Set(4) { 1, "some text", {…}, {…} } in Chrome
```

```
mySet.clear()//清楚所有成员，没有返回值
```

遍历

Set实例遍历的方法有如下：

关于遍历的方法，有如下：

- keys()：返回键名的遍历器
- values()：返回键值的遍历器
- entries()：返回键值对的遍历器
- forEach()：使用回调函数遍历每个成员

Set的遍历顺序就是插入顺序

keys方法、values方法、entries方法返回的都是遍历器对象。

实现并集、交集、和差集:

```
let a = new Set([1, 2, 3]);

let b = new Set([4, 3, 2]);

// 并集

let union = new Set([...a, ...b]);

// Set {1, 2, 3, 4}

// 交集

let intersect = new Set([...a].filter(x => b.has(x)));

// set {2, 3}

//  (a 相对于 b 的) 差集

let difference = new Set([...a].filter(x => !b.has(x)));

// Set {1}
```

Map字典

Map 对象存有键值对，其中的键可以是任何数据类型。

Map 对象记得键的原始插入顺序。

Map 对象具有表示映射大小的属性。

```
map.size // 属性返回 Map 结构的成员总数。

const m = new Map().set('key1', 'val1')

Map.prototype.set(key, value) // 方法设置键名key对应的键值为value，然后返回整个 Map 结构。如果key已经有值，则键值会被更新，否则就新生成该键。

// set方法返回的是当前的Map对象，因此可以采用链式写法。

Map.prototype.get(key) // 读取key对应的键值，如果找不到key，返回undefined。

Map.prototype.has(key) // 返回一个布尔值，表示某个键是否在当前 Map 对象之中

Map.prototype.delete(key) // 删除某个键，返回true。如果删除失败，返回false。

Map.prototype.clear() // 清除所有成员，没有返回值

Map.prototype.keys(): // 返回键名的遍历器。

Map.prototype.values(): // 返回键值的遍历器。

Map.prototype.entries(): // 返回所有成员的遍历器。

Map.prototype.forEach(): // 遍历 Map 的所有成员

forEach // 可以接受第二个参数，用来绑定this。
```

String方法

- charAt(index): 返回指定索引处的字符串，若没找着，返回空
- charCodeAt(index): 返回指定索引处的字符的 Unicode 值

```
// toLowerCase()转换成小写 toUpperCase()转换成大写

var x = "a".toLowerCase().charCodeAt(0)

//x = 97
```

- **concat(str1, str2, ...):** 连接多个字符串，返回连接后字符串副本，纯函数（和数组一样的用法）
- fromCharCode(): 将 Unicode 值转换成实际字符串

```
String.fromCharCode(97)

// 返回"a"
```

- indexOf(str): 返回 str 在父串中第一次出现的位置，若没有返回-1
- lastIndexOf(str): 返回 str 在父串中最后一次出现的位置，若没有返回-1

- match(regex): 搜索字符串，返回正则表达式的所有匹配
- **search(regex): 基于正则表达式搜索字符串，返回第一个匹配的位置**
- slice(start, end)：返回字符索引在 start 和 end（不含）之间的子串
- split(sep，limit)：将字符串分割为字符数组，limit 为从头开始执行分割的最大数量
- **substr(start，length)**：从字符索引 **start** 的位置开始，返回长度为 **length** 的子串
- **substring(from, to)**：返回字符索引在 **from** 和 **to**（不含）之间的子串，和slice几乎相同，但它允许from>to，不支持负参数
- **toLowerCase()**：将字符串转换为小写
- **toUpperCase()**：将字符串转换为大写
- **valueOf()**：返回原始字符串值
- toString() 把Number 对象转换为字符串，返回结果

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

[] 的 valueOf 结果为 []，toString 的结果为 ""

- str.codePointAt(pos)返回pos位置的字符编码

```
// 不同的字母有不同的代码

alert( "z".codePointAt(0) ); // 122

alert( "Z".codePointAt(0) ); // 90
```

- **String.fromCodePoint(code)**通过code创建字符

```
alert( String.fromCodePoint(90) ); // Z

//\u后跟十六进制代码，通过代码添加Unicode字符

// 在十六进制系统中 90 为 5a

alert( '\u005a' ); // Z

'a'>'Z'

因为字符通过数字代码比较，a(97)>Z(90)

// 英文是否大写

function upperCase(num) {

var reg = /^[A-Z]+$/;

return reg.test(num);

}
```

replace

不会修改原字符串！

第二个参数传入要替换的目标字符串，**replace只会匹配一次**

第二个参数也可传入一个函数，若原始字符串中有n个我们查找的字符串，函数就会执行n次，且函数返回一个字符串，来替换每次匹配到的字符串

参数

$&

$& 适用于没有子表达式的情况

```
var sStr='讨论一下正则表达式中的replace的用法';

sStr.replace(/正则表达式/,'《$&》');

// 得到："讨论一下《正则表达式》中的replace的用法"
```

$`

匹配字符串左边的所有字符

```
var sStr='讨论一下正则表达式中的replace的用法';

sStr.replace(/正则表达式/,'《$`》');

// 得到："讨论一下《讨论一下》中的replace的用法"
```

$'

匹配字符串右边的所有字符

```
var sStr='讨论一下正则表达式中的replace的用法';

sStr.replace(/正则表达式/,"《$'》");

// 得到："讨论一下《中的replace的用法》中的replace的用法"
```

1,2,3,4……n

依次匹配子表达式

```
var sStr='讨论一下正则表达式中的replace的用法';

sStr.replace(/(正则)(.+?)(式)/,"《$1》$2<$3>");

// 得到："讨论一下《正则》表达<式>中的replace的用法"
```

函数

```
var sStr='讨论一下正则表达式中的replace的用法';

sStr.replace(/(正则).+?(式)/,function() {

console.log(arguments);

});

// ["正则表达式", "正则", "式", 4, "讨论一下正则表达式中的replace的用法"]
```

参数分别为

- 匹配到的字符串
- 若正则使用了分组匹配就是多个，否则无此参数
- 匹配字符串的索引位置
- 原始字符串

arguments是当前函数的内置属性，指代当前匹配的参数伪数组。

或者使用命名形参的方式：

```
var sStr='讨论一下正则表达式中的replace的正则表达式用法';

sStr.replace(/(正则).+?(式)/g,function($1) {

console.log($1);

return $1 + 'a';

});
```

用法

```
str = str.replace(/\s*/g); //去除字符串内所有的空格 \s匹配任何空白字符。（空格，制表符，换行符）

str = str.replace(/^\s*|\s*$/g, "");//去除字符串内首尾空格

str = str.replace(/^\s*/, "");//去除字符串左侧空格

str = str.replace(/\s*&/, "");//去除字符串右侧空格

name = "Doe, John";

let a=name.replace(/(\w+)\s*, \s*(\w+)/, "$2 $1");

console.log(a)

//John Doe

//首字母大写
```

```
let name = 'aaa bbb ccc';

let uw=name.replace(/\b\w+\b/g, function(word){

return word.substring(0,1).toUpperCase()+word.substring(1);}

);
```

str.trim()

trim()删除字符串两端的空白字符并返回，**不影响原来字符串本身，**返回一个新的字符串**

缺陷：只能去除字符串两端的空格，不能去除中间的空格

截取字符串

substring()

substring()用于提取字符串中介于两个指定下标之间的字符

substring(start,stop)

- start：一个非负整数，指要提取的子串的第一个字符在字符串中的位置，必需填写
- stop：一个非负整数，比要提取的子串的最后一个字符在字符串上的位置多 1，可写可不写，如果不写则返回的子串会一直到字符串的结尾

该字符串的长度为stop-start

如果参数 start 与 stop 相等，则该方法返回的就是一个空串，如果 start 比 stop 大，那么该方法在提取子串之前会先交换这两个参数

substr()

substr()在字符串中抽取从 start 下标开始的指定数目的字符

substr(start,length)

- start：要截取的子串的起始下标，必须是数值。如果是负数，那么该参数从字符串的尾部开始算起的位置。也就是说，-1 指字符串中最后一个字符，-2 指倒数第二个字符，以此类推，必需要写
- length：子串中的字符数，必须是数值。如果不填该参数，返回字符串的开始位置到结尾的字符。如果length 为 0 或者负数，将返回一个空字符串

split()

split() 把一个字符串分割成字符串数组

stringObject.split(separator,howmany)

- separator：字符串或正则表达式，从该参数指定的地方分割字符串。必须要填写
- howmany：指返回的数组的最大长度。如果设置了该参数，返回的子串不会多于这个参数指定的数组。如果没有设置该参数，整个字符串都会被分割，不考虑它的长度。可选

遍历对象属性

使用 hasOwnProperty 判断对象自身属性中是否具有指定的属性

访问属性点表示和[]

[]语法的主要优点是可以通过变量访问属性。

如果属性包含空格，就不能通过 . 访问它。属性名可以包含非字母非数字，使用[]访问它。

除非必须使用变量访问属性，否则我们使用点表示法。

注意，属性名表达式如果是一个对象，默认情况下会自动将对象转为字符串[object Object]

```
const keyA = {
    a: 1
};
const keyB = {
    b: 2
};

const myObject = {
    [keyA]: 'valueA',
    [keyB]: 'valueB'
};

myObject // Object {[object Object]: "valueB"}
```

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）

Object.keys(obj)

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）的键名

Object.getOwnPropertyNames(obj)

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）的键名

Object.getOwnPropertySymbols(obj)

返回一个数组，包含对象自身的所有 Symbol 属性的键名

Reflect.ownKeys(obj)

返回一个数组，包含对象自身的（不含继承的）所有键名，不管键名是 Symbol 或字符串，也不管是否可枚举

遍历对象的方法

1. for in

以任意顺序 迭代 一个对象的除Symbol以外的可枚举属性，包括继承的可枚举属性。

```
// Object 原型链上扩展的方法也会被遍历出来
Object.prototype.fun = () => {};
var obj = {
    a: 1,
    b: 2,
    c: 3
};

for (const item in obj) {
    console.log("属性名: " + item + " / 属性值: " + obj[item]);
}

// 属性名: a / 属性值: 1
// 属性名: b / 属性值: 2
// 属性名: c / 属性值: 3
// 属性名: fun / 属性值: () => {}

//而如果我们不希望搜索到原型上的，我们就可以使用 hasOwnProperty
for (const item in obj) {
    if (obj.hasOwnProperty(item)) {
        console.log("属性名: " + item + " / 属性值: " + obj[item]);
    }
}
// 属性名: a / 属性值: 1
// 属性名: b / 属性值: 2
// 属性名: c / 属性值: 3
```

2. object.key
   返回一个给定对象的自身可枚举 属性名组成的数组，数组中属性名的排列顺序和正常循环遍历该对象时返回的顺序一致 。

```
// 注意: 当属性名为数字的时候会排序，key / values / entries都有这个特性~
const anObj = { 100: 'a', 2: 'b', 7: 'c' };
console.log(Object.keys(anObj)); // console: ['2', '7', '100']

const str = 'hello';
console.log(Object.keys(str));// ["0", "1", "2", "3", "4"]

const arr = ['a', 'b', 'c'];
console.log(Object.key(arr));// ["0", "1", "2"]

var obj = {a:1, b:2, c:3};
console.log(Object.keys(obj));// ["a","b","c"]
```

3. Object.values
   返回一个给定对象自身的所有可枚举 属性值的数组，值的顺序与使用for…in循环的顺序一致

```
// 注意：当属性名为数字的时候会排序，key / values / entries都有这个特性~
const obj = {100:1, d:2, a: 9, 1:3, 5: 99 , b: 8};
console.log(Object.values(obj)); //  [3,99,1,2,9,8]

const str = 'hello';
console.log(Object.values(str));// ["h", "e", "l", "l", "o"]

const arr = ['a', 'b', 'c'];
console.log(Object.values(arr));// ["a", "b", "c"]

var obj = {a:1, b:2, c:3};
console.log(Object.values(obj));// ["1","2","3"]
```

4. Object.entries
返回一个给定对象自身可枚举属性的 键值对数组，其排列与使用 for...in 循环遍历该对象时返回的顺序一致

```
// 注意：当属性名为数字的时候会排序，key / values / entries都有这个特性~
const obj = {100:1, d:2, a: 9, 1:3, 5: 99 , b: 8};
console.log(Object.entries(obj)); //   [["1",3],["5",99],["100",1],["d",2],["a",9],
["b",8]]

const str = 'hello';
console.log(Object.entries(str));// [["0","h"],["1","e"],["2","l"],["3","l"],
["4","o"]]

const obj = {a:1, b:2, c:3};
console.log(Object.entries(obj));// [["a",1],["b",2],["c",3]]

const obj1 = {a:1, b:2, c:3};
for (const [key, value] of Object.entries(obj1)) {
  console.log(`${key}: ${value}`);
}
//a: 1
// b: 2
// c: 3
```

5. Object.getOwnPropertyNames
返回一个由指定对象的所有自身属性的 属性名组成的数组。（包括不可枚举属性但不包括Symbol值作为名称的属性）

```
// 注意：当属性名为数字的时候会排序，key / values / entries都有这个特性~
const arobj = {100:1, d:2, a: 9, 1:3, 5: 99 , b: 8};
console.log(Object.getOwnPropertyNames(arobj)); //  ["1","5","100","d","a","b"]

const str = 'hello';
console.log(Object.getOwnPropertyNames(str));//  ["0","1","2","3","4","length"]

const arr = ['a', 'b', 'c'];
console.log(Object.getOwnPropertyNames(arr));// ["0","1","2","length"]
```

```
const obj = {a:1, b:2, c:3};
console.log(Object.getOwnPropertyNames(obj));// ["a","b","c"]

const syobj = { a:1, b:2 };
const symbol1 = Symbol('symbol1')
const symbol2 = Symbol('symbol2')
syobj[symbol1] = 'hello'
syobj[symbol2] = 'world'
console.log(Object.getOwnPropertyNames(syobj)); //["a","b"]
```

6. `Object.getOwnPropertySymbols()`
   方法返回一个给定对象自身的 所有 **Symbol** 属性的数组

```
const obj = {a:1, b:2, c:3};
const symbol1 = Symbol('symbol1')
const symbol2 = Symbol('symbol2')
obj[symbol1] = 'hello'
obj[symbol2] = 'world'
console.log(Object.getOwnPropertySymbols(obj));// [Symbol(symbol1), Symbol(symbol2)]
```

7. `Reflect.ownKeys()`
   静态方法 Reflect.ownKeys() 返回一个由目标对象自身的 属性名组成的数组

```
const obj = {a:1, b:2, c:3};
const symbol1 = Symbol('symbol1')
const symbol2 = Symbol('symbol2')
obj[symbol1] = 'hello'
obj[symbol2] = 'world'
console.log(Reflect.ownKeys(obj));// ["a","b","c",Symbol(symbol1), Symbol(symbol2)]
```

> 注意
>
> - 在 ES6 之前 Object 的键值对是无序的；
> - 在 ES6 之后 Object 的键值对按照自然数、非自然数和 Symbol 进行排序，自然数是按照大小升序进行排序，其他两种都是按照插入的时间顺序进行排序。

判断对象是否具有属性

1、in

如果属性来自对象的原型，仍然返回true

```
let obj = {
    name: 'aa'
};

'name' in obj; //true

'toString' in obj; //true
```

2、Reflect.has()

检查属性是否在对象中，和in一样作为函数工作

```
const obj = {
    name: 111
};

Reflect.has(obj, 'name'); //true

Reflect.has(obj, 'toString'); //true
```

3、hasOwnProperty()

返回布尔值，指对象是否具有指定属性作为它自己的属性(不是继承)

可正确区分对象本身属性和其原型的属性

```
const obj = {
    a: 1
};

obj.hasOwnProperty('a'); //true

obj.hasOwnProperty('toString'); //false
```

缺点：如果对象是用Object.create(null)创建的，不能使用这个方法

```
const obj = Object.create(null);

obj.name = 'merry';

obj.hasOwnProperty('name');

//Uncaught TypeError: obj.hasOwnProperty is not a function
```

4、Object.prototype.hasOwnProperty()

可解决3的问题，本方法直接调用内置有效函数，跳过原型链

```
const obj = Object.create(null);

obj.name = 'merry';

Object.prototype.hasOwnProperty.call(obj, 'name'); //true

Object.prototype.hasOwnProperty.call(obj, 'toString'); //false
```

5、Object.hasOwn()

若对象具有指定属性作为自己的属性，则Object.hasOwn()静态方法返回true，若属性被继承或不存在，返回false

```
const obj = Object.create(null);

obj.name = 'merry';

Object.hasOwn(obj, 'name'); //true

Object.hasOwn(obj, 'toString'); //false
```

原型

> 实例是类的具象化产品
>
> 对象是一个具有多种属性的内容结构
>
> **实例都是对象，对象不一定是实例（如Object.prototype是对象但不是实例），构造函数也是对象**
>
> prototype是构造函数的属性
>
> **proto**是对象的属性

Object也是Function的实例，那Function是谁的实例呢?

```
Function.__proto__=== Function.prototype;

//true
```

Function构造函数的prototype和**proto**属性都指向同一个原型，是否可以说Function对象是由Function构造函数创建的一个实例?

Yes and No.

Yes 的部分： 按照JS中“实例”的定义，a 是 b 的实例即 a instanceof b 为 true，默认判断条件就是 b.prototype 在 a 的原型链上。而 Function instanceof Function 为 true，本质上即 Object.getPrototypeOf(Function) === Function.prototype，正符合此定义。

No 的部分：

Function 是 built-in 的对象，也就是并不存在“Function对象由Function构造函数创建”这样显然会造成鸡生蛋蛋生鸡的问题。实际上，当直接写一个函数时（如 function f() {} 或 x => x），也不存在调用 Function 构造器，只有在你显式调用 Function 构造器时（如 new Function('x', 'return x')）才有。

注意，本质上，a instanceof b 只是一个运算，即满足某种条件就返回 true/false，当我们说 a 是 b 的实例时，也只是表示他们符合某种关系。JS 是一门强大的动态语言，你甚至可以在运行时改变这种关系，比如修改对象的原型从而改变 instanceof 运算的结果。此外，ES6+ 已允许通过 Symbol.hasInstance 来自定义 instanceof 运算。

```
Function.prototype//"function"
```

后来意见：

> 先有的Object.prototype， Object.prototype构造出Function.prototype，然后Function.prototype构造出Object和Function。

> Object.prototype是鸡，Object和Function都是蛋。

prototype

有一个默认的constructor属性,用于记录实例由哪个构造函数创建.

proto

每一个对象都具有的一个属性,叫proto,这个属性指向该对象的原型.，原型有两个属性,constructor和proto

既然实例对象和构造函数都可以指向原型,那么原型是否有属性可以指向构造函数或实例吗?

proto与其说是一个属性,不如说是个getter/setter,当使用obj.*proto*时,可以理解为返回了Object.getPrototypeOf(obj).

constructor

每一个原型都有一个constructor属性指向关联的构造函数.

```
function Person()  {}

var person = new Person();

console.log(person.__proto__ == Person.prototype) // true

console.log(Person.prototype.constructor == Person) // true 原型对象的constructor指向构造函数本身

// 顺便学习一个ES5的方法,可以获得对象的原型

console.log(Object.getPrototypeOf(person) === Person.prototype) // true

console.log(Object.getPrototypeOf(person))

person.__proto__

Person.prototype //constructor
```

实例&原型

当读取实例属性时,若找不到,就会查找与对象关联原型中的属性,若还查不到,就去找原型的原型,一直找到最顶层为止.

原型的原型

实例的proto指向构造函数的prototype

原型链

```
console.log(Object.prototype.__proto__ === null) // true
```

意思就是Object.prototype没得原型.

图中由相互关联的原型组成的练状结构就是原型链(蓝色这条线)

创建一个原型链只有name属性的对象

```
let obj = Object.create(null) //为obj的prototype属性赋值为null

    obj.name = 'merry'

    console.log(obj)
```

参考文档https://zh.javascript.info/prototype-methods

proto和prototype

prototype是原型对象

__ proto __ 将对象和该对象的原型相连

特殊的 Function 对象，Function 的 **proto** 指向的是自身的 prototype 。

构造函数 prototype 的 **proto** 也是指向**构造函数的构造函数**的 prototype

> 构造函数是一个函数对象，通过Function构造器产生的。
>
> 原型对象本身是一个普通对象，而普通对象的构造函数是Object。

除了Object的原型对象（Object.prototype）的**proto**指向null，其他内置函数对象的原型对象（例如：Array.prototype）和自定义构造函数的 **proto**都指向Object.prototype, 因为原型对象本身是普通对象

```
Object.prototype.__proto__ = null;

Array.prototype.__proto__ = Object.prototype;

Foo.prototype.proto = Object.prototype;
```

- 一切对象都是继承自Object对象，Object 对象直接继承根源对象null
- 一切的函数对象（包括 Object 对象），都是继承自 Function 对象
- Object 对象直接继承自 Function 对象
- Function对象的**proto**会指向自己的原型对象，最终还是继承自Object对象

原型&原型链

需要new关键字，成为“构造器constructor或构造函数”。

通过prototype定义的属性，再被多个实例化后，引用地址是同一个。

继承链 从祖父——到爷爷——到爸爸——到自己

- constructor指向构造函数，每个对象的**proto**指向创建它的构造函数的prototype，而构造函数的prototype也有**proto**指向他的父辈或者是Object，当查找一个对象中不存在的属性时，会去它的**proto**、**proto**中的**proto**中进行寻找，直到找到或者是null为止
- instanceof判断对象的**proto**和构造函数的prototype是不是同一个地址
- Object.setPrototypeOf改变对象的**proto**

原型(prototype)：一个对象，实现对象的属性继承，简单理解为对象的爹。

prototype可以通过Object.getPrototypeOf() 和 Object.setPrototypeOf() 访问器访问。

当继承的函数被调用时，this 指向的是当前继承的对象，而不是继承的函数所在的原型对象。

不是所有对象都有原型。

数据类型

原始类型

Undefined, Null, Boolean, Number, String

ES6新增了Symbol和BigInt.

- Symbol 代表独一无二的值,最大的用法是为对象定义唯一的属性名
- BigInt 可表示任意大小的整数，指安全存储、操作大整数。

数据处理

- parseInt(5.4) 只保留整数部分，有基模式

> 解析一个字符串，并返回一个整数。parseInt相比Number，就没那么严格了，parseInt函数逐个解析字符，遇到不能转换的字符就停下来。
>
> parseInt(string, radix)
>
> string 必需，表示要被解析的字符串。radix 可选，表示要解析的数字的基数。该值介于 2 ~ 36 之间。
>
> 如果省略该参数或其值为 ‘0‘，则数字将以 10 为基础来解析。如果它以 ‘”0x”‘ 或 ‘”0X”‘ 开头，将以 16 为基数。
>
> 如果该参数小于 2 或者大于 36，则 ‘parseInt()‘ 将返回 ‘NaN‘。

- parseFloat() 把值转换成浮点数,没有基模式
- Number() 把给定的值转换成数字（可以是整数或浮点数）,Number()的强制类型转换与parseInt()和parseFloat()方法的处理方式相似，只是它转换的是整个值，而不是部分值。
- Math.floor(4.33) 向下取整

- Math.ceil(6.7) 向上取整
- Math.round(6.19) 四舍五入
- Math.abs(-1) 绝对值
- String() 把给定的值转换成字符串
- toFixed(2) 四舍五入

null&undefined

- 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。当我们使用双等号对两种类型的值进行比较时会返回 true，使用三个等号时会返回 false。

typeof null为"Object"

在第一版JS中，变量的值被设计保存在一个32位内存单元中。该单元包含一个1或3位的类型标标志，和实际数据值。类型标志存储在单元的最后。包括以下2几种情况

1. 000：object，数据为对象的引用
2. 1：int，数据为 31 位的有符号整型
3. 010：double，数据为一个双精度浮点数的引用
4. 100：string，数据为一个字符串的引用
5. 110：boolean，数据为布尔类型的值

特殊情况：

- undefined 负的 2 的 30 次方（超出当时整型取值范围的一个数）
- null 空指针

null的存储单元最后三位(即 标志位)和object一样，所以被误判为Object

引用类型

1. Object
2. Date
3. Array
4. RegExp

判断数据类型

一、常见判断：typeof

- 这个方法很常见，一般用来 **判断基本数据类型**，如：string，number，boolean，symbol，bigint（es10新增一种基本数据类型bigint），undefined等。返回数据类型的字符串形式

typeof 目前能返回string，number，boolean，symbol，bigint，unfined，object，function这八种判断类型，但是注意 null 返回的是 Object 。而且对于引用类型返回的是 object 因为所有的对象的原型最终都是 Object。

```
//例子
typeof "safsadff"; //"string"
typeof 145; //"number"
typeof false; //"boolean"
typeof undefined; //"undefined"
typeof function () {}; //"function"
typeof {}; //"object"
typeof Symbol(); //"symbol"
typeof null; //"object"
typeof[]; //"object"
typeof new Date(); //"object"
typeof new RegExp(); //"object"
typeof new Number(33) //"object"
typeof Null //"undefined"
```

> **为什么typeof null 是 Object**
> 答：因为在JavaScript中，不同的对象都是使用二进制存储的，如果二进制前三位都是0的话，系统会判断为是Object类型，而null的二进制全是0，自然也就判断为Object
> 这个bug是初版本的JavaScript中留下的，扩展一下其他五种标识位：
>
> - 000 对象
> - 1 整型
> - 010 双精度类型
> - 100字符串
> - 110布尔类型

二、 已知对象判断：**instanceof**

- 用来 **判断引用数据类型的，判断基本数据类型无效** ，如：Object，Function，Array，Date，RegExp等，instanceof主要的作用就是判断一个实例是否属于某种类型
- instanceof也可以判断一个实例是否是其父类型或者祖先类型
- instanceof原理实际上就是查找目标对象的原型链

```
//例子
[]instanceof Array; // true
[]instanceof Object; // true
[1, 2, 3]instanceof Array // true
new Date()instanceof Date // true
```

手写实现一个：

```
//手写实现
function myInstance(L, R) { //L代表instanceof左边，R代表右边
    var RP = R.prototype
        var LP = L.__proto__
        while (true) {
            if (LP == null) {
                return false
            }
            if (LP == RP) {
                return true
```

```
            }
            LP = LP.__proto__
        }
}
console.log(myInstance({}, Object));
```

三、根据对象的构造器：**constructor**

与 instanceof 相似，但是对于 instanceof 只能再检测引用类型， 而 constructor 还可以检测基本类型，因为constructor是原型对象的属性指向构造函数。

> 注意
>
> - null 和 undefined是无效的对象，因此是不会有 constructor 存在的，所以无法根据 constructor 来判断。
> - JS对象的 constructor 是不稳定的，这个主要体现在自定义对象上，当开发者重写prototype 后，原有的 constructor 会丢失，constructor 会默认为Object
> - 类继承的也会出错，因为 Object 被覆盖了，检测结果就不对了

四 、对象原型链判断：**Object.prototype.toString.call（这个是判断类型最准的方法）**

- toString 是Object **原型对象上的一个方法，该方法默认返回其调用者的具体类型，更严格的讲，是 toString运行时this指向的对象类型， 返回的类型格式为[object,xxx]，xxx是具体的数据类型**，其中包括：String，Number，Boolean，Undefined，Null，Function，Date，Array，RegExp，Error，HTMLDocument… 基本上所有对象的类型都可以通过这个方法获取到。
- 必须通过 Object.prototype.toString.call 来获取，而不能直接 new Date().toString()， 从原型链的角度讲，所有对象的原型链最终都指向了Object，按照JS变量查找规则，其他对象应该也可以直接访问到Object的 toString 方法，而事实上，大部分的对象都实现了自身的 toString 方法，这样就可能会导致 Object 的 toString 被终止查找，因此要用 call 来强制执行 Object 的 toString 方法。
- 缺点：不能再细分

```
//例子
Object.prototype.toString.call("a")
"[object String]"
Object.prototype.toString.call(undefined)
"[object Undefined]"
Object.prototype.toString.call(null)
"[object Null]"
Object.prototype.toString.call(new Date())
"[object Date]"
```

简单来说，JavaScript 中我们有四种方法来判断数据类型。

一般使用 typeof 来判断基本数据类型，不过需要注意当遇到null的问题，这里不足就是不能判断对象具体类型（typeof xjj 只是Object不能看出是person）；

而在要判断一个对象的具体类型，就可以用 instanceof，但是也可能不准确，对于一些基础数据类型，数组等会被判断为object。

结合 typeof 和 instanceof 的特点，还能使用 constructor 来判断，他能判断基本类型和引用类型，但是对于 null 和 undefined 是无效的，以及 constructor 不太稳定。

最后如果需要判断准确的内置类型，就可以使用 object.prototype.toString.call，是根据原型对象上的 tostring 方法获取的，该方法默认返回其调用者的具体类型。

for in 和 for of 的区别

一句话总结：`for ... in`是为遍历对象属性而构建的，遍历的是 `index`，而 `for ... of`是为了遍历数组的，遍历的是 `value`

```
const arobj = {
    100: 1,
    d: 2,
    a: 9,
    1: 3,
    5: 99,
    b: 8
};
Object.prototype.fun = () => {};
arobj.name = "lallala"
    const arr = [1, 2, 4, 8, 9, 10]
    Array.prototype.method = () => {};
arr.name = "bababa";

for (const item in arobj) {
    console.log(arobj[item]) // 3 99 1 2 9 8 lallala () => {}
}
for (const item in arr) {
    console.log(item) //0 1 2 3 4 5 name method fun
}
for (const item of arobj) {
    console.log(item) // Uncaught TypeError: arobj is not iterable
}
for (const item of arr) {
    console.log(item) // 1 2 4 8 9 10
}
```

一、for ... in

1. for … in 适合遍历对象，遍历数组的时候会出现奇奇怪怪的问题
2. for … in 遍历会遍历所有的可枚举属性，包括原型链上的，可以使用 hasOwnProperty 来过滤
3. for … in 中 index 索引为字符串型数字，不能直接进行几何运算

二、for ... of

1. 适合比遍历所有拥有迭代器的对象的集合

=== 和 == 以及 = 的区别

它们最主要的区别在于 == 对比时，若类型不相等，会先转换为相同类型，然后再来比较值。而 === 则不会，只能在相同类型下比较值，不会做类型转换。还有一个是 =，这个是赋值，不是运算符。

1、 ===

这个比较简单。下面的规则用来判断两个值是否===相等：

- 如果类型不同，就不相等
- 如果两个都是数值，并且是同一个值，那么相等；(！例外)的是，如果其中至少一个是NaN，那么不相等。（判断一个值是否是NaN，只能用 isNaN() 来判断）
- 如果两个都是字符串，每个位置的字符都一样，那么相等；否则不相等。
- 如果两个值都是true，或者都是false，那么相等。
- 如果两个值都引用同一个对象或函数，那么相等；否则不相等。
- 如果两个值都是null，或者都是undefined，那么相等。

2、==

- 如果两个值类型相同，进行 === 比较。
- 如果两个值类型不同，他们可能相等。根据下面规则进行类型转换再比较：
- 如果一个是null、一个是undefined，那么相等。
- 如果一个是字符串，一个是数值，把字符串转换成数值再进行比较。
- 如果任一值是 true ，把它转换成 1 再比较；如果任一值是 false，把它转换成 0 再比较。
- 如果一个是对象，另一 个是数值或字符串，把对象转换成基础类型的值再比较。对象转换成基础类型，利用它的 toString 或者 valueOf 方法。js 核心内置类，会尝试 valueOf 先于 toString；例外的是 Date，Date利用的是 toString转换。非 js 核心的对象，令说（比较麻烦，我也不大懂）
- 任何其他组合，都不相等。

> 有时会在结构读取属性值时会在前面加上“? ”这个是可选链，表示判断空值的情况。

instanceof

原理：**判断构造函数的prototype属性是否出现在对象的原型链上**

优点：instanceof 可以弥补 Object. prototype. toString. call()不能判断自定义实例化对象的缺点

缺点： instanceof 只能用来判断对象类型

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

实现

```
// 利用原型链向上查找 能找到这个类的prototype的话，就为true
function myInstanceof(left, right) {
```

```
    if (left === null || typeof right !== 'function') {

        return false;
    }
    let proto = Object.getPrototypeOf(left); // 获取对象的原型

    // let proto=left.proto;
    let prototype = right.prototype; // 获取构造函数的 prototype 对象

    // 判断构造函数的 prototype 对象是否在对象的原型链上
    while (true) {

        if (proto === null) {
            return false;
        }

        if (proto === prototype) {
            return true;
        }

        proto = Object.getPrototypeOf(proto);
    }
}

const Person = function () {}

const p1 = new Person()

    console.log(myInstanceof(p1, Person));
```

typeof&instanceof

typeof 操作符返回一个字符串，表示未经计算的操作数的类型。

instanceof 运算符用于检测构造函数的 prototype 属性是否出现在某个实例对象的原型链上

区别如下：

- typeof会返回一个变量的基本类型，instanceof返回的是一个布尔值
- instanceof 可以准确地判断复杂引用数据类型，但是不能正确判断基础数据类型
- typeof 虽然可以判断基础数据类型（null 除外），但是引用数据类型中，除了function 类型以外，其他的也无法判断

可以看到，上述两种方法都有弊端，并不能满足所有场景的需求

如果需要通用检测数据类型，可以采用Object.prototype.toString，统一返回格式“[object Xxx]”的字符串

类型转换

> 类型

6种基本类型 null undefined number stringify boolean symbol

1种引用类型 object

> 对象转换为基本类型

- 对象转换为字符串

```
// 模拟 toString 返回的不是基本类型值，valueOf 返回的基本类型值
var obj = {

    toString: function () {
        return {}
    },

    valueOf: function () {
        return null
    }
}

String(obj) // "null"
```

- 对象转换为数字

先判断valueOf方法，再判断toString方法

```
// valueOf 和 toString 返回的都不是基本类型值
var obj = {

    valueOf: function () {
        return {}
    },

    toString: function () {
        return {}
    }
}

Number(obj) // Uncaught TypeError: Cannot convert object to primitive value
```

Object.create(null)创建的对象没有valueOf和toString方法，因此转换报错

一般情况，我们不会重写valueOf和toString，大部分对象valueOf返回的仍然是对象，因此对象转换为基本类型值可以直接看toString返回的值

> 显式强制类型转换

- 转换为字符串

如果对象有自定义toString方法，则返回toString方法的结果，若是toString返回的结果不是基本类型值，报错TypeError

```
var obj = {

    toString: function () {
        return {}
    }
}

String(obj) // Uncaught TypeError: Cannot convert object to primitive value

obj + "" // Uncaught TypeError: Cannot convert object to primitive value

obj.toString() // {}
```

- 转换为布尔类型

null undefined false +0 -0 NaN ""

其他情况都是真值

- 转换为数字类型

```
Number('') // 0

Number(null) // 0

Number(undefined) // NaN

Number(true) // 1

Number(false) // 0
```

对象首先被转换为相应基本类型值，再转换

```
Number([]) // 0

// [] valueOf 返回的是 []，因此继续调用 toString 得到基本类型值 ""，转换为数字为 0
```

隐式强制类型转换

- 转换为字符串

x+""，会将x转换为字符串，+ 运算符其中一个操作数是字符串，会执行字符串拼接操作

对象和字符串拼接时，**对象转为基本类型按转为数字转换，**先判断valueOf，再判断toString

```
var obj = {

    valueOf: function () {
        return 1
    },

    toString: function () {
        return 2
    }
}

obj + '' // '1'
```

- 转换为布尔值

发生布尔值隐式强制类型转换的情况

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）

- 转换为数字类型

```
+ '2' // 2

'2' - 0 // 2

'2' / 1 // 2

'2' * 1 // 2

+ 'x' // NaN

'x' - 0 // NaN

'x' / 1 // NaN

'x' * 1 // NaN

1 + '2' // '12'

1 + + '2' // 3 即: 1 + (+ '2')
```

> == 和 ===

== 允许在相等比较中进行强制类型转换，而 === 不允许

比较规则

1、字符串和数字比较，字符串先转换为数字，再比较

2、其他类型和布尔类型比较，布尔类型转换为数字，再比较

3、对象和非对象比较，对象转换成基本类型值，按转换为数字的流程转换后进行比较。对象转换优先级最高

4、null 和 undefined，null == undefined， 其他类型和 null 均不相等，undefined 也是如此

5、特殊情况

```
NaN == NaN // false

-0 == +0 // true
```

两个对象比较，判断的是两个对象是否是同一个引用

```
"0" == false // true

// 第2条规则，false 转换为数字，结果为 0，等式变为 "0" == 0

// 两边类型不一致，继续转换，第1条规则，"0" 转换为数字，结果为 0，等式变为 0 == 0

false == [] // true

// 第3条规则，[] 转换基本类型值，[].toString()，结果为 ""，等式变为 "" == false

// 两边类型不一致，继续转换，第2条规则，false 转换为数字，结果为 0，等式变为 "" == 0

// 两边类型不一致，继续转换，第1条规则，"" 转换为数字，结果为 0，等式变为 0 == 0

0 == [] // true

// 第3条规则，[] 转换基本类型值，[].toString()，结果为 ""，等式变为 0 == ""

// 两边类型不一致，继续转换，第1条规则，"" 转换为数字，结果为 0，等式变为 0 == 0
```

数组和链表

数据的物理存储结构：连续存储（数组）离散存储（链表）

区别：

对数组，查找方便，连续存储，增删改效率低；事先申请好连续的内存空间小，太大浪费内存，太小越界

对链表，动态申请内存空间，不需要像数组需要提前申请好内存的大小，链表只需在用的时候申请就可以，根据需要来动态申请或者删除内存空间，对于数据增加和删除以及插入比数组灵活

应用场景：

数组：适合数据量固定，频繁查询，较少增删的场景

链表：适合数据量不固定，频繁增删，较少查询的场景

数组去重

indexof、includes、filter

```
function unique(arr) {
    let res = arr.filter(function (item, index, array) {

            return array.indexOf(item) === index
        })

        return res;
}
```

Set

```
let unique=arr=>[...new Set(arr)];

let res=Array.from(new Set(arr));
```

Map

```
const unique = (arr) => {

    const map = new Map();
    const res = [];

    for (let item of arr) {

        if (!map.has(item)) {

            map.set(item, true);
            res.push(item);
        }
    }
}
```

数组拍平

1. ary = ary.flat(Infinity); //Infinity无穷大
2. toString+replace+split
3. replace+JSON.parse
4. 递归

```
function flatten(arr, n) {

    if (n < 2) {
        return arr;
```

```
    }

    let res = [];
    const dfs = (arr, n) => {
        if (n < 2) {
            res.push(arr);
            return res;
        }
        for (let item of arr) {
            if (Array.isArray(item) && n) {
                dfs(item, n - 1);
            } else {
                res.push(item);
            }
        }
    }

    dfs(arr, n);

    return res;
}
```

5. 利用reduce函数迭代

```
const arr3 = [
    [1, 2],
    [3, 4],
    [5, [7, [9, 10], 8], 6],
];
console.log(flatten(arr3, 2));
function flatten(_arr, depth = 1) {
    if (depth === 0) {
        return _arr;
    }
    return _arr.reduce((pre, cur) =>
        pre.concat(Array.isArray(cur) && depth > 1 ?
            flatten(cur, depth - 1) :
            cur), [])
}
function flatten(_arr, depth = 1) {
    if (depth === 0) {
        return _arr;
    }
    return _arr.reduce((pre, cur) => {
        return Array.isArray(cur) && depth > 1 ?
        [...pre, ...flatten(cur, depth - 1)] :
        [...pre, cur];
    }, [])
}
```

6. 扩展运算符

```
while (arr.some(Array.isArray)) {
    arr = [].concat(...arr);
}
```

7. toString+split+map

```
const str = [1, 2, 3, [5, 6, [7, 8]]].toString();
const _arr = str.split(",");
const newArr = _arr.map(item => +item);
console.log(newArr)
```

ES6新增的数组方法

find找到符合条件的成员

findIndex找到符合条件元素的下标

flat拍平数组

- 默认拍平一层
- 带参n表示拍平n层
- Infinity作为参数，不管几层，全部拍成一层
- 数组有空位会跳过

flatMap()

- 只能拍平一层
- 对每个元素执行一个函数，再执行flat()
- 返回新数组，不改变原数组

Array.at()返回对应下标的值

Array.from()将类似数组的对象转为真的数组

- 类数组对象需要有length属性，否则返回[]

```
let arrayLike = {

 '0': 'a',
 '1': 'b',
 '2': 'c',

 length: 3
};

// ES5的写法
var arr1 = [].slice.call(arrayLike); // ['a', 'b', 'c']

// ES6的写法
let arr2 = Array.from(arrayLike); // ['a', 'b', 'c']
```

Array.of()将 一组数值 转为 数组

```
const a = Array.of(10, 20, 26, 38);

console.log(a); // [10, 20, 26, 38]
```

Array.includes() 判断数组是否包含某个值，返回boolean类型

... 扩展运算符

- rest参数的逆运算

copyWithin()

- 将指定位置的成员复制到其他位置（会覆盖原有成员），然后返回当前数组。

Array.prototype.copyWithin(target, start = 0, end = this.length)

fill()填充数组

entries()遍历键值对

keys()遍历键名

values()遍历键值

Array.proptype.sort()的排序稳定性

空位

判断是否存在某个值

1. array.indexOf() //return 下标或-1
2. array.find() //**返回满足条件的第一个元素的值**，若没有，则返回undefined
3. array.findIndex() //返回满足条件的第一个元素下标，若没有找到，返回-1
4. String.prototype.includes()
5. **includes()** 方法用于判断一个字符串/数组是否包含在另一个字符串中，根据情况返回 true 或 false，方法的第二个参数表示搜索的起始位置，默认为0，参数为负数则表示倒数的位置。

arguments为啥不是数组

另一种对象类型，也叫类对象数组（类数组）

有callee和length属性

如何转换为数组

Array.prototype.slice.call()

```
function sum(a, b) {
    let args = Array.prototype.slice.call(arguments);
    console.log(args.reduce((sum, cur) => sum + cur)); //args可以调用数组原生的方法啦
}
sum(1, 2); //3
```

Array.form

对一个类似数组或可迭代对象创建一个新的，浅拷贝的数组实例。

从类数组对象或者可迭代对象中创建一个新的数组实例

```
function sum(a, b) {

    //Array.from() 方法对一个类似数组或可迭代对象创建一个新的，浅拷贝的数组实例。
    let args = Array.from(arguments);
    console.log(args.reduce((sum, cur) => sum + cur));
}

sum(1, 2);
```

ES6扩展运算符

```
function sum(a, b) {

    let args = [...arguments];
    console.log(args.reduce((sum, cur) => sum + cur));
}

sum(1, 2);
```

concat+apply or apply

```
function sum(a, b) {

    let args = Array.prototype.concat.apply([], arguments); //apply会把第二个参数展开
    console.log(args.reduce((sum, cur) => sum + cur));
}

function exam(a, b, c, d, e) {

    // 先看看函数的自带属性 arguments 什么是样子的
    console.log(arguments);

    // 使用call/apply将arguments转换为数组，返回结果为数组，arguments自身不会改变
    var arg = [].slice.call(arguments);

    console.log(arg);
}
```

```
exam(2, 8, 9, 10, 3);
// result:
// { '0': 2, '1': 8, '2': 9, '3': 10, '4': 3 }
// [ 2, 8, 9, 10, 3 ]
//
// 也常常使用该方法将DOM中的nodelist转换为数组
// [].slice.call( document.getElementsByTagName('li') );
```

函数

函数声明

使用function的函数声明比函数表达式优先提升

变量对象的创建过程中，函数声明比变量声明 有更为优先的执行顺序

无论在什么位置声明了函数，都可以在同一个执行上下文中直接使用

函数表达式

也叫匿名函数

函数表达式使用 var/let/const声明，我们在确认他是否可以正确使用时，必须依照var/let/const的规则 判断，即变量声明

使用var进行变量声明， 进行了两步操作

```
// 变量声明
var a = 20;

// 实际执行顺序
var a = undefined; // 变量声明，初始值undefined，变量提升，提升顺序次于function声明

a = 20; // 变量赋值，该操作不会提升
```

同样道理，当我们使用变量声明的方式声明函数时——函数表达式。函数表达的提升方式与变量声明一致

```
fn(); // 报错

var fn = function () {
    console.log('function');
}

//上述例子执行顺序为

var fn = undefined; // 变量声明提升

fn(); // 执行报错
```

```
fn = function () {
    // 赋值操作，此时将后边函数的引用赋值给fn
    console.log('function');
}
```

因此，由于声明方式的不同，导致 函数声明与函数表达式在使用上的一些差异需要 注意，除此之外，这两种形式的函数在使用上并无不同

匿名函数

没有被显示进行赋值操作的函数。 使用场景——多作为一个参数传入另一个函数中

函数自执行，其实是匿名函数的一种应用

回调函数

匿名函数传入另一个函数之后，最终 在另一个函数中执行，因此我们也常常称这个匿名函数为回调函数

高阶函数

一个函数 接收另一个函数作为参数或返回另一个函数

1. map
2. reduce

参数: 两个参数，一个为回调函数，另一个 初始值。回调函数中四个默认参数，依次为积累值、当前值、当前索引和整个数组

1. filter 返回新数组
2. sort

普通函数

若是'use strict'，不能将全局对象window作为默认绑定。this=undefined

普通函数：定义时this 指向函数作用域，但定时器之后执行函数时， this指向 window

普通函数的this——调用时所在的对象

函数声明、函数表达式

1. 函数声明式： function functionName （）{}
2. 函数表达式：let name = function(){}

```
console.log(a) //undefined

var a = 1

    console.log(getNum) //getNum(){a=3}

    var getNum = function () {
    a = 2
}
function getNum() {
    a = 3
```

```
}
console.log(a) //1

getNum()

console.log(a) //2
```

函数声明有提升，代码执行前 把函数提升到顶部，执行上下文中生成函数定义，所以第二个 getNum会被最先提升到顶部

然后 var 声明 getNum 的本该提升，但是因为 getNum 的函数已经被声明了，所以就不需要再声明一个同名变量，只是将已经声明的getNum替换掉了，于是修改a=2

```
console.log(foo) //undefined

console.log(bar) //f(){}

var foo = function () {
    // Some code
};

function bar() {
    // Some code
};
```

var foo提升至顶部为foo=undefined

```
// foo bar的定义位置被提升
function bar() {
    // Some code
};

var foo;

console.log(foo) //undefined

console.log(bar)

foo = function () {
    // Some code
};
```

自执行函数

[IIFE]Immediately Invoked Function Expression：声明即执行的函数表达式

函数传参：按值传递

按值传递，当我们期望传递一个引用类型时，真正传递的，只是这个引用类型保存在变量对象中的引用而已

```
var person = {
```

```
    name: 'Nicholas',

    age: 20
}

function setName(obj) {
    // 传入一个引用
    obj = {}; // 将传入的引用指向另外的值

    obj.name = 'Greg'; // 修改引用的name属性
}

setName(person);

console.log(person.name); // Nicholas 未被改变
```

函数式编程

将这 多次出现的功能封装

使用时，只需要关心这个方法能做什么，而不用关心具体是怎么实现的。这也是函数式编程思维与命令式不同的地方之一

特征

函数是第一等公民

所谓"第一等公民"（first class），是函数与其他数据类型一样，处于平等地位，可以赋值给其他变量，也可以作为参数，传入另一个函数，或者作为别的函数的返回值

只用"表达式"，不用"语句"

"表达式"（expression）是一个单纯的运算过程，总是有返回值；"语句"（statement）是执行某种操作，没有返回值。函数式编程要求，只使用表达式，不使用语句。每一步都是单纯的运算，都有返回值

函数式编程期望一个函数有输入，也有输出

纯函数

即：**只要是同样的参数传入，返回的结果一定是相等的**

函数柯里化

是高阶函数的一种特殊用法

柯里化：一个函数(假设叫做createCurry)，接收函数A作为参数，运行后返回一个新的函数。并且这个新的函数能够处理函数A的剩余参数

柯里化函数的运行过程是一个参数的收集过程，我们将每一次传入的参数收集起来，并在最里层里面处理

> fn.length 表示fn函数的参数个数

实现函数柯里化

```
let addCurry = curry1((a, b) => a + b);

console.log(addCurry()(11)(1));

function curry1(fn) {

    let judge = (...args) => {

        if (args.length === fn.length) {

            return fn.call(this, ...args);
        }

        //获取偏函数，返回包装器，重新组装参数并传入
        return (...arg) => judge(...arg, ...args)
    }

    return judge;
}
```

函数的 length

请问下面这个结果是多少呢

```
123['toString'].length + 123 = ?
```

1、形参个数
来看看下面这个例子

```
function fn1 () {}
function fn2 (name) {}
function fn3 (name, age) {}

console.log(fn1.length) // 0
console.log(fn2.length) // 1
console.log(fn3.length) // 2
```

可以看出，function有多少个形参，length就是多少。但是事实真是这样吗？继续往下看

2、 默认参数
如果有默认参数的话，函数的length会是多少呢？

```
function fn1 (name) {}
function fn2 (name = '林三心') {}
function fn3 (name, age = 22) {}
function fn4 (name, aaa, age = 22, gender) {}
function fn5(name = '林三心', age, gender, aaa) {}

console.log(fn1.length) // 1
console.log(fn2.length) // 0
console.log(fn3.length) // 1
console.log(fn4.length) // 2
console.log(fn5.length) // 0
```

上面可以看出，function的length，就是第一个具有默认值之前的参数个数

3、 剩余参数
在函数的形参中，还有剩余参数这个东西，那如果具有剩余参数，会是怎么算呢?

```
function fn1(name, ...args) {}
console.log(fn1.length) // 1
```

可以看出，剩余参数是不算进length的计算之中的

4、总结
先公布123['toString'].length + 123 = ?的答案是124
length 是函数对象的一个属性值，指该函数有多少个必须要传入的参数，即形参的个数。形参的数量不包括剩余参数个数，仅包括第一个具有默认值之前的参数个数。

箭头函数和普通函数区别

箭头函数的 this 为定义时所在的 this，不绑定this (因为箭头函数没有Constructor)，会捕获其所在上下文的 this 作为自己的 this

若包裹在函数中，就是函数调用时所在的对象，放在全局就是window，箭头函数的this就是外层代码块的this，固定不变。

- 没有自己的this
- 继承来的this不会变
- 没有arguments，实际获得的arguments是外层函数的arguments
- call apply 和bind无法改变this指向
- 不可用于构造函数，没有new关键字
- 无prototype
- 不能用于generator函数，没有yield关键字

> 定义对象的大括号{}不是一个单独的执行环境，它依旧处于全局环境中

构造函数

一、构造函数

在 JavaScript 中，通过 new 来实例化对象的函数叫构造函数，也就是初始化一个实例对象，对象的 prototype 属性是继承一个实例对象。构造函数的命名一般会首字母大写

二、 为什么需要使用构造函数

是为了创建对象

JavaScript 中创建对象有两种，一种是 构建函数 + prototype，另一种是用 class。这里我们不去讲解 class ，先放到构造函数上。

```
//当我们需要创建比较多信息时
var person1 = { name: 'aa', age: 6, gender: '男', classRoom: '高一'};
var person2 = { name: 'bb', age: 6, gender: '女', classRoom: '高一'};
var person3 = { name: 'cc', age: 6, gender: '女', classRoom: '高一'};
var person4 = { name: 'dd', age: 6, gender: '男', classRoom: '高一'};
```

那么如果需要创建很多呢？需要这样一个一个的写下去吗？但是实际上可以通过下面的形式来实现。

```
function Person(name, age, gender, classRoom) {
    this.name = name;
    this.age = age;
    this.gender = gender;
    this.classRoom = '高一'
}

Person.prototype.sayHi = function () {
    console.log("你好，我叫" + this.name + "是一个" + this.sex + "来自" + classRoom);
};

let person1 = new Person("a", 18, '男'); //我们还可以不传 classRoom，让他使用默认的
let person2 = new Person("b", 19, '女');
```

三、构造函数的执行过程

构造函数的执行过程其实也就是 new 操作符的基本过程

- **创建一个新的对象，并且在内存中创建一个新的地址。** // let person1 = {};
- **继承原型** // person1 .**proto** = Person.prototype
- **改变构造函数的 this指向，并且新对象添加构造函数的属性和方法** //执行Person函数，将name,age,sex 参数传入Person中执行，此时函数内部 this 为 new 创建的 person1 对象，所以 person1.name = 'a'; person1 .age = 18; person1.gender= '男';
- **根据构建函数返回类型作判断，如果是原始值则被忽略，如果是返回对象，需要正常处理**

new 操作符通过构造函数创建的实例，可以访问构造函数的属性和方法，同时实例与构造函数通过原型链连接起来了。

四、构造函数的返回值

构造函数中，不要显式返回任何值：

```
//返回原始值
function Person(name) {
    this.name = name;
    return '啦啦啦啦'
}
let person1 = new Person("a");
console.log(person1.name) //a

//返回对象

function Person(name) {
    this.name = name;
    return {
        aaa: "asdas"
    }
}
let person1 = new Person("a");
console.log(person1) //{aaa : "asdas"}
console.log(person1.name) //'undefined'
```

可以看到当返回原始值的时候，并不会正常返回这个原始值“啦啦啦啦”，而当返回直是对象的时候，这个返回值能被正常返回，但是这时候 new 就不生效了。所以，**构造函数尽量不要返回值。因为返回原始值不会生效，返回对象会导致 new 操作符没有作用。**

那么为什么会这样呢?

> 构造函数没有返回值的原因是因为它不是由你的代码直接调用的，它是由运行时的内存分配和对象初始化代码调用的。 它的返回值（如果它在编译为机器代码时实际上有一个）对用户来说是不透明的——因此，你不能指定它。

> 其实在 JavaScript 中，
>
> - let a = [] 也就是 let a = new Array [];
> - function a () {} 也就是 let a = new Function () {}

继承方式

构造函数会在每一个实例上都创建一遍！

使用原型模式定义的属性和方法由所有实例共享！

原型链

```
function Parent() {
    this.name = 'kevin';
}

Parent.prototype.getName = function () {
```

```
    console.log(this.name);
}

function Child() {}

Child.prototype = new Parent();

var child1 = new Child();

console.log(child1.getName()) // kevin
```

缺点

1. 引用类型的属性被所有实例共享
2. 创建Child实例时，不能向Parent传参

借用构造函数(经典继承)

```
function Parent() {
    this.names = ['kevin', 'daisy'];
}

function Child() {
    Parent.call(this);
}

var child1 = new Child();

child1.names.push('yayu');

console.log(child1.names); // ["kevin", "daisy", "yayu"]

var child2 = new Child();

console.log(child2.names); // ["kevin", "daisy"]
```

1. 避免了 引用类型的属性被所有实例共享
2. 可以在 Child 中向 Parent 传参

缺点

方法都在构造函数中 定义，每次创建实例都会创建一遍方法

组合继承

原型链继承+经典继承 *双剑合璧*

```
function Parent(name) {
    this.name = name;

    this.colors = ['red', 'blue', 'green'];
}
```

```
Parent.prototype.getName = function () {
    console.log(this.name)
}

function Child(name, age) {
    Parent.call(this, name);

    this.age = age;
}

Child.prototype = new Parent();

Child.prototype.constructor = Child;

var child1 = new Child('kevin', '18');

child1.colors.push('black');

console.log(child1.name); // kevin

console.log(child1.age); // 18

console.log(child1.colors); // ["red", "blue", "green", "black"]

var child2 = new Child('daisy', '20');

console.log(child2.name); // daisy

console.log(child2.age); // 20

console.log(child2.colors); // ["red", "blue", "green"]
```

融合 原型链 继承和构造函数的优点，JS最常用继承模式

原型式继承

```
function createObj(o) {
    function F()  {}

    F.prototype = o;

    return new F();
}
```

ES5 Object.create的模拟实现，将传入对象 作为 创建的对象的原型

包含引用类型的属性值都会共享相应的值——和原型链 继承一样

```
var person = {
    name: 'kevin',
```

```
    friends: ['daisy', 'kelly']
}

var person1 = createObj(person);

var person2 = createObj(person);

person1.name = 'person1';

console.log(person2.name); // kevin

person1.friends.push('taylor');

console.log(person2.friends); // ["daisy", "kelly", "taylor"]
```

寄生式继承

创建一个仅用于封装继承过程的函数，该函数在内部以某种形式 做增强对象，再返回对象

```
function createObj(o) {
    var clone = Object.create(o);

    clone.sayName = function () {
        console.log('hi');
    }

    return clone;
}
```

缺点

和借用构造函数一样，每次创建对象都会创建一遍方法

寄生组合式继承

组合继承 最大缺点——调用2次 父构造方法

1. 设置子实例的原型时
2. 创建子类型的实例时

如何避免重复调用?

如果我们不使用Child.prototype=new Parent()，而是间接让 Child.prototype访问Parent.prototype呢?

```
function Parent(name) {
    this.name = name;

    this.colors = ['red', 'blue', 'green'];
}

Parent.prototype.getName = function () {
```

```
    console.log(this.name)
}

function Child(name, age) {
    Parent.call(this, name);

    this.age = age;
}

// 关键的三步

var F = function ()  {};

F.prototype = Parent.prototype;

Child.prototype = new F();

var child1 = new Child('kevin', '18');

console.log(child1);
```

封装一下这个继承方法

```
function object(o) {
    function F()  {}

    F.prototype = o;

    return new F();
}

function prototype(child, parent) {
    var prototype = object(parent.prototype); //创建父类原型对象的副本

    prototype.constructor = child; //增强对象，补充因重写原型而失去的默认的constructor属性

    child.prototype = prototype; //将创建的新副本指定给子类的原型对象
}

// 当我们使用的时候：
prototype(Child, Parent);
```

优点——引用类型最理想的继承方式

只调用一次Parent 构造函数，避免了在Parent.prototype上面创建不必要的、多余的属性

同时，原型链能保持不变，能正常使用instanceof和isPrototypeOf

深浅拷贝

浅拷贝只能拷贝一层对象。

深拷贝能解决无限层级对象嵌套问题。

浅拷贝

1. **Object.assign()**

拷贝的是对象属性的引用，而不是对象本身。

```
console.log(Object.assign([1, 2, 3], [4, 5])); //4,5,3
```

assign方法可以用于处理数组，不过会把数组视为对象，比如这里会把目标数组视为是属性为0、1、2的对象，所以源数组的0、1属性的值覆盖了目标对象的值。

```
var obj = {

    age: 18,
    nature: ['smart', 'good'],

    names: {
        name1: 'fx',
        name2: 'xka'
    },

    love: function () {
        console.log('fx is a great girl')
    }
}

var newObj = Object.assign({}, obj);
```

2. **concat浅拷贝数组**
3. **slice浅拷贝**
4. **...扩展运算符**
5. **for...in**

深拷贝

1.JSON.parse(JSON.stringify());

- 无法解决循环利用问题
- 无法拷贝一条特殊对象 RegExp Date Set Map 等
- 忽略undefined、symbol和函数

2. 递归实现
3. lodash第三方库实现

拷贝特殊对象，使用 Object.prototype.toString.call(obj)鉴别。

this

new绑定>显示绑定>隐式绑定>默认绑定

this指向在函数被调用时确定，即执行上下文被创建时确定。函数执行过程中，一旦this指向被确定就不可更改。

在一个函数上下文中，this由调用者提供，由调用函数的方式来决定。**如果调用者函数，被某一个对象所拥有，那么该函数在调用时，内部的this指向该对象。如果函数独立调用，那么该函数内部的this，则指向undefined。**但是在非严格模式中，当this指向undefined时，它会被自动指向全局对象。

```
// demo03
var a = 20;

var obj = {

    a: 10,

    c: this.a + 20,

    fn: function () {
        return this.a;
    }
}

console.log(obj.c); //40

console.log(obj.fn()); //10
```

单独的{}不会形成新的作用域，因此这里的this.a，由于并没有作用域的限制，它仍然处于全局作用域之中。所以这里的this其实是指向的window对象。

全局上下文

非严格模式和严格模式中this都是指向顶层对象（浏览器中是window，nodejs中是global）。

```
// 在浏览器中，全局对象为 window 对象：

console.log(this === window); // true

this.a = 37;

console.log(window.a); // 37
```

函数调用

严格模式下 this 为undefined，非严格模式下为 window

箭头函数

作为方法的箭头函数 this 指向当前被定义变量的上下文环境。

箭头函数的this 为定义时所在的 this，不绑定this (因为箭头函数没有Constructor这个内部槽)，会捕获其所在上下文的 this 作为自己的 this。(箭头函数的this就是它在外面的第一个不是箭头函数的函数的this)**

> 没有arguments变量

若包裹在函数中，就是函数调用时所在的对象，放在全局就是window，箭头函数的this就是外层代码块的this，固定不变。

ES5 中，永远是 this 指向最后调用它的那个对象

- 不可用于构造函数，无prototype
- 无法改变this指向

bind 函数

在 Function的原型链上，Function.prototype.bind.通过 bind 函数绑定后，函数将绑定在其第一个参数对象上，除非使用new时被改变，其他情况不会改变，无论在啥情况下被调用。

setTimeout & setInterval

延时函数内部回调函数this 指向全局对象window。

使用箭头函数使其 this 指向我们所期望的那个对象。

> 1. 如果this 是 在一个函数中，并且被用作方法来叫，那么这个时候的 this 就指向了父级对象;
> 2. 如果this 是在匿名函数，或者全局环境的函数中，那么这个时候的 this 就是undefined;
> 3. 如果this 是在构造函数中，那么这个时候的 this 就指向了即将生成的那个对象

introduction() === introduction.call() ，前者是后者的简写！并且call()中的第一个传参可以指定这个函数中的 this 指向谁！

```
function introduction(name) {
    console.log('你好,' + name + ' 我是' + this.name);
}

var zhangsan = {
    name: '张三'
}

introduction.call(zhangsan, "李四") // 你好 李四, 我是 张三 call

introduction.apply(zhangsan, ["李四"]) // 你好 李四, 我是 张三 apply

intro = introduction.bind(zhangsan)

    intro("李四") // 你好 李四, 我是 张三 bind
```

bind()是返回一个绑定新环境的function，等着被调用。

关于 this 的指向问题

每个函数的 this 是在调用时被绑定的，完全取决于函数的调用位置

如何判断 this 的指向呢？有两个关键的点 调用位置 和 绑定规则。

一、调用位置

我们知道执行是在执行栈中的，执行栈是一个栈的结构，遵从先进后出，我们会把执行的任务压入栈中。所以我们这里寻找的调用位置就是当前正在执行的函数的上一个调用的位置。下面看个例子理解：

```
function a() {
    //当前调用栈是：a，所以当前调用位置是全局
    console.log(“a”)
    b(); // b 的调用位置
}
function b() {
    //当前调用栈是：a->b，所以当前调用位置是a
    console.log(“b”)
    c(); // c 的调用位置
}
function c() {
    //当前调用栈是：a->b->c，所以当前调用位置是b
    console.log(“c”)
}

a()
```

二、绑定规则

1. **默认绑定**：这条规则可以看作是无法应用其他规则时的默认规则，也是我们最常用的。

```
var a = 2
function foo() {
    var a = 3;
    console.log(this.a)
}
foo() // 2，为什么不是3，那就是因为使用了 this，根据我们上面的调用规则，他是绑定到全局中的
```

2. **隐式绑定**
   需要考虑的规则是调用位置是否有上下文对象，当函数拥有上下文对象时，隐式绑定会把this绑定到这个上下文来，并且如果函数调用前存在多个对象，this指向距离调用自己最近的对象。下面我们看例子来理解这段话：

```
var a = 1;

function foo() {
    var a = 4;
    console.log(this.a)
}

var obj2 = {
```

```
    a:2,
    foo:foo
};

var obj1 = {
    a:3,
    obj2: obj2
};

obj2.foo()  // 2 当函数拥有上下文对象时，隐式绑定会把this绑定到这个上下文来
obj1.obj2.foo() //2 如果函数调用前存在多个对象，this指向距离调用自己最近的对象
```

在隐式绑定中，还会出现一个问题，那就是 **隐式丢失**，作为 参数传递 以及 变量赋值， 会使参数或者变量直接指向函数，从而丢失 this 指向。

1. 变量赋值

```
var a = 1;
function foo() {
    var a = 3;
    console.log(this.a)
}
var obj = {
    a:2,
    foo:foo
};
var bar = obj.foo;
bar() // 1
```

为什么这里输出的是 1 ，而不是按照上面的规则输出 2 呢？这是因为这里的引用实际只是给函数起了个名字 bar 而已，并没有被调用，在声明 var bar 后才调用 bar () == foo ()， 而此时的 this 指向也就是全局的了。

2. 参数传递

```
var a = 1;
function foo() {
    var a = 3;
    console.log(this.a)
}
function doFun(fn) {
 fn()
}
var obj = {
    a:2,
    foo:foo
};
doFun(obj.foo;) // 1
```

这里其实也是和上面的情况相似的，在 doFun 里面，fu 其实引用的是 foo，所以指向的是全局的

那么如何解决 **隐式丢失** 呢？

使用 **隐式绑定**：将函数绑定至对象属性，调用时通过对象属性直接调用，弱赋值到其他对象，需将正对象赋值过去，不然会发生丢失（丢失初次绑定的环境）

3. **显式绑定**

显示绑定就是指通过 call，apply 以及 bind方法 改变 this 的行为，具体这三个方法看下面就好了~

4. **new 绑定**

我们在上面的构造函数中讲到 new 操作符构造函数的过程，可以上去看看~

使用 new 来调用的时候，实际构建一个新的对象并把他绑定到 foo（）调用的 this 上。

```
function foo(a) {
    this.a = a;
}
var bar = new foo(2);
console.log(bar.a) // 2
```

5. **箭头函数**

箭头函数不会创建自己的 this，所以它没有自己的this，它只会从自己的作用域链的上一层继承 this。

改变this指向的方法（bind | call | apply）

三者都是用来重新定义 this 这个对象的

一、三者的区别：

1、调用上

```
let name = "www", age = "17";
let obj = {
    name = "aaa";
    objAge = this.age;
    myFun: function () {
        console.log(this.name + "年龄" + this.age)
    }
}

let db = {
    name: "bbb";
    age: 12
}

obj.myFun.call(db);          // bbb年龄 99
obj.myFun.apply(db);    // bbb年龄 99
obj.myFun.bind(db)();   // bbb年龄 99
```

首先我们可以看出，除了 bind 需要在方法后面添加 “()” 以外，其他的都是直接调用。这是因为 bind()方法创建了一个新的函数，你必须调用显函数才会执行目标函数， 而对于 call 和 apply 是使用后马上执行。

2、参数上

```
let name = "www", age = "17";
let obj = {
  name = "aaa";
  objAge = this.age;
  myFun: function(fm, t) {
    console.log( this.name + " 年龄" + this.age, "来自" + fm + "去往" + t)
  }
}

let db={
  name: "bbb";
  age: 12
}

obj.myFun.call(db,'成都','上海');          // bbb 年龄 99  来自 成都去往上海
obj.myFun.apply(db,['成都','上海']);       // bbb年龄 99  来自 成都去往上海
obj.myFun.bind(db,'成都','上海')();        // bbb 年龄 99  来自 成都去往上海
obj.myFun.bind(db,['成都','上海'])();     // bbb 年龄 99  来自 成都, 上海去往 undefined
```

可以看出，三个函数的第一个参数都是 this 的指向对象，区别在于第二个参数。

- 对于 **Function.prototype.call** 来说，第一个参数就是 this 的指向对象，其余参数是直接放进去，用逗号隔开就好了。
- 对于 **Function.prototype.apply** 来说，Function.apply(obj,args) 方法能接收两个参数，obj：是 this 的指向对象。而 args：这个是数组，它将作为参数传递，也就是说 apply 的所有参数都必须放在一个数组里面传进去
- 对于 **Function.prototype.bind** 来说，第一个参数就是 this 的指向对象，其余参数是直接放进去，用逗号隔开就好了。也就是说他和 call 是基本相同的，除了是返回是一个函数。

> 注意
> 对于 bind 来说，多次的 bind 调用，this的指向仍然是第一次的
> function aa() {
> console.log(this)
> }
> aa.bind(1).bind(2)() / / 1

二、bind / call / apply 的异同

相同：都能改变 this 的指向，都是挂载在 Function. prototype 上
不同：call 和 apply 是使用后马上执行，而 bind 是返回一个新的函数，调用显函数才会执行目标函数。并且 call 和 bind 的参数格式是一样的，第一个参数是 this 的指向对象，其余参数用逗号，apply 是参数需要放到数组中。

总结

关于修改 this 的指向的方法有三个， bind 和 call 以及 apply，他们的相同点都是 能修改 this 的指向的问题的，并且都是挂载在 Function.prototype 上的。

不同点在于参数 和执行上，call 和 bind 的参数格式是一样的，第一个参数是 this 的指向对象，其余参数用逗号，而 apply 的参数需要放到数组中。在执行中，call 和 apply 是使用后马上执行，而 bind 是返回一个新的函数，调用显函数才会执行目标函数。

其中需要注意的是，箭头函数的 this 是指向他所在的上下文中，并且是不能使用这三个方法修改的。

作用域

Scope，变量（变量作用域又称为上下文）和函数存在的范围。

作用域:规定了如何查找变量,即确定当前执行代码对变量的访问权限.(决定了代码区块中变量和其他资源的可见性),内层作用域可以访问外层作用域的变量,反之不行.

> 词法环境其实就是作用域，是一套规则

全局作用域

在代码任何地方都能访问到的对象拥有全局作用域.

一般以下几种情形有全局作用域链:

1. 最外层函数和在最外层函数外面定义的变量拥有全局作用域
2. 所有未定义直接赋值的变量自动声明为拥有全局作用域
3. 所有window对象的属性拥有全局作用域

函数作用域

函数作用域,是指声明在函数内部的变量，和全局作用域相反，局部作用域一般只在固定的代码片段内可访问到，最常见的例如函数内部。

块级作用域(ES6新增)

块级作用域可通过新增命令let和const声明，所声明的变量在指定块的作用域外无法被访问。块级作用域在如下情况被创建:

1. 在一个函数内部
2. 在一个代码块（由一对花括号包裹）内部

特点:

- 声明变量不会提升到代码块顶部
- 禁止重复声明
- 循环中的绑定块作用域的妙用

for循环还有一个特别之处，就是设置循环变量的那部分是一个父作用域，而循环体内部是一个单独的子作用域。

```
var i = 1

function b() {

  console.log(i)
}
```

```
function a() {

  var i = 2

  b()
}

a() // 1
```

每个函数在执行时都会创建一个执行上下文，其中会关联一个变量对象，也就是它的作用域，上面保存着该函数能访问的所有变量，另外上下文中的代码在执行时还会创建一个作用域链，

如果某个标识符在当前作用域中没有找到，会沿着外层作用域继续查找，直到最顶端的全局作用域，**因为js是词法作用域，在写代码阶段就作用域就已经确定了，换句话说，是在函数定义的时候确定的，而不是执行的时候，**所以b函数是在全局作用域中定义的，虽然在a函数内调用，但是它只能访问到全局的作用域而不能访问到a函数的作用域。

词法作用域

又叫做静态作用域，变量被创建时就确定好了，而不是执行阶段。

作用域链

当查找变量时,会先从当前上下文的变量对象中查找,从当前作用域开始一层层往上找某个变量,如果没有找到，就会从父级(词法层面上的父级执行上下文的变量对象中查找，一直找到全局上下文的变量对象，(多个执行上下文的变量对象构成的链表叫做作用域链)

保证了当前执行环境对符合访问权限的变量和函数的有序访问。

JS采用静态作用域(词法作用域)，因此函数的作用域在函数定义时就确定了.

与其相对的是动态作用域,函数的作用域在函数调用时才决定.

作用域链规定如何查找变量，确定当前执行代码对变量的访问权限。

块语句（大括号“｛｝”中间的语句），如 if 和 switch 条件语句或 for 和 while 循环语句，不像函数，它们不会创建一个新的作用域。

作用域和作用域链

说到作用域链，我们不得不先从作用域开始。首先我们得知道在js中有全局作用域和函数作用域。顾名思义：

作用域就是变量与函数的可访问范围，即作用域控制着变量和函数的可见性和生命周期

- 全局作用域的变量，函数在整个全局中都能被访问到，它的生命周期和页面的等同
- 函数作用域的，只能在当前函数内被访问到，生命周期随函数结束而结束销毁。

所以每一个变量或函数都会有自己的作用域范围，而作用域链简单看来说就是:

当前作用域范围（自身内部）中找不到时，就会往他的上一级寻找有没有，直到全局都没有的，返回 undefined。

要小心的是，有些时候，不要相信我们第一眼看到的就以为是他的上一级。如何判断他的上一级需要根据词法作用域来判断。

页面生命周期

HTML 页面的生命周期包含三个重要事件：

- DOMContentLoaded —— 浏览器已完全加载 HTML，并构建了 **DOM** 树，但像 和样式表之类的外部资源可能尚未加载完成。
- load —— 浏览器不仅加载完成了 HTML，还加载完成了所有外部资源：图片，样式等。
- beforeunload/unload —— 当用户正在离开页面时。

每个事件都是有用的：

- DOMContentLoaded 事件 —— DOM 已经就绪，因此处理程序可以查找 DOM 节点，并初始化接口。
- load 事件 —— 外部资源已加载完成，样式已被应用，图片大小也已知了。
- beforeunload 事件 —— 用户正在离开：我们可以检查用户是否保存了更改，并询问他是否真的要离开。
- unload 事件 —— 用户几乎已经离开，但是我们仍然可以启动一些操作，例如发送统计数据。

DOMContenLoaded

当浏览器遇到script标签，会在DOM构建之前运行它，因为脚本可能想要修改DOM，对其执行write操作，所以DOMContentLoaded必须等待脚本执行结束。

外部样式表不会影响DOM，因为DOMContenLoaded不会等待他们。

但是，如果在样式后面有一个脚本，那么该脚本必须等待样式表加载完成：

```
<link type="text/css" rel="stylesheet" href="style.css">

<script>

 // 在样式表加载完成之前，脚本都不会执行

 alert(getComputedStyle(document.body).marginTop);

</script>
```

原因是，脚本可能想要获取元素的坐标或其他与样式相关的属性，如上例所示。因此，它必须等待样式加载完成。

当 DOMContentLoaded 等待脚本时，它也在等待脚本前面的样式。

事件流

事件流描述 从页面接收事件的顺序

事件发生时会在元素节点和根节点之间按照特定的顺序传播，路径所经过的节点都会收到该事件——DOM事件流。

两种事件流模型：

1.捕获：触发元素的事件时，该事件从该元素的祖先元素传递下去（不太具体的节点应该更早接收到事件，而最具体的节点最后收到事件）

2.冒泡：到达此元素之后，又会向其祖先元素传播上去 DOM标准规定事件流包括3个阶段：事件捕获、处于目标阶段和事件冒泡

- 事件捕获：实现目标在捕获阶段不会接收事件
- 处于目标阶：事件在
  上发生并处理。但是事件处理会被看成是冒泡阶段的一部分
- 冒泡阶段：事件又传播回文档

所有事件都要经过捕获阶段和处于目标阶段，但有些事件没有冒泡阶段。

如focus(获得输入焦点)和失去输入焦点的blur事件。无法进行委托

事件模型

原始事件模型

```
var btn = document.getElementById('.btn');

btn.onclick = fun;
```

- 绑定速度快

可能页面还未完全加载出来，事件可能无法正常运行

- 只支持冒泡，不支持捕获
- 同一个类型的事件只能绑定一次

```
var btn = document.getElementById('.btn');

btn.onclick = fun2;

//出错 后绑定的事件会覆盖掉之前的事件

删除 DOM0 级事件处理程序只要将对应事件属性置为null即可

btn.onclick = null;
```

标准事件模型

- 事件捕获：从document一直向下传播到目标元素, 依次检查经过的节点是否绑定了事件监听函数，如果有则执行
- 事件处理：到达目标元素, 触发目标元素的监听函数
- 事件冒泡：从目标元素冒泡到document, 依次检查经过的节点是否绑定了事件监听函数，如果有则执行

事件绑定监听函数的方式如下：

```
addEventListener(eventType, handler, useCapture)
```

事件移除监听函数的方式如下：

> removeEventListener(eventType, handler, useCapture)

- eventType指定事件类型(不要加on)
- handler是事件处理函数
- useCapture是boolean类型用于指定是否在捕获阶段进行处理，一般设置为false与IE浏览器保持一致

举个例子：

```
var btn = document.getElementById('.btn');

btn.addEventListener('click', showMessage, false);

btn.removeEventListener('click', showMessage, false);
```

特性

可以在一个DOM元素上绑定多个事件处理器，不会冲突

```
btn.addEventListener('click', showMessage1, false);

btn.addEventListener('click', showMessage2, false);

btn.addEventListener('click', showMessage3, false);
```

第三个参数(useCapture)设置为true就在捕获过程中执行，反之在冒泡过程中执行

IE事件模型

- 事件处理：事件到达目标元素, 触发目标元素的监听函数。
- 事件冒泡：事件从目标元素冒泡到document
- 事件绑定监听函数的方式

> attachEvent(eventType, handler)

- 事件移除监听函数的方式

> detachEvent(eventType, handler)

举个例子：

```
var btn = document.getElementById('.btn');

btn.attachEvent('onclick', showMessage);

btn.detachEvent('onclick', showMessage);
```

模块化

特点

1. 解决命名污染，全局污染，变量冲突等
2. 内聚私有，变量不能被外部访问
3. 更好的分离，按需加载
4. 引入其他模块可能存在循环引用问题

5. 代码抽象，封装，复用和管理
6. 避免通过script标签从上至下加载资源
7. 大型项目资源难以维护，特别是多人合作的情况

CommonJS

服务端解决方案。加载速度快(因为模块文件一般存在本地硬盘)

- **每个文件是一个模块，有自己的作用域。**在一个文件里面定义的变量、函数、类，都是私有的，对其他文件不可见。
- 运行时加载，只能在运行时才能确定一些东西
- **同步加载**，只有加载完成后，才能执行后续操作。 因为用于服务端，文件都在本地，同步导入即使卡住主线程影响也不大。
- **导出时都是值拷贝**，就算导出的值变了，导入的值也不会改变，所以如果想更新值，必须重新导入一次。
- **模块在首次执行后会缓存，再次加载只返回缓存结果，若想再次执行，可清除缓存。**
- 模块加载的顺序就是代码出现的顺序

> 基本语法

- 暴露模块：module.exports = value或exports.xxx = value
- 引入模块：require(xxx),如果是第三方模块，xxx为模块名；如果是自定义模块，xxx为模块文件路径

CommonJS规范规定，每个模块内部，module变量代表当前模块。这个变量是一个对象，它的exports属性（即module.exports）是对外的接口。**加载某个模块，其实是加载该模块的module.exports属性。**

```
// 加载模块
var example = require('./example.js');

var config = require('config.js');

var http = require('http');

// 对外暴露模块
module.exports.example = function () {

 ...

}

module.exports = function(x){

  console.log(x)
}

exports.xxx=value;
```

require命令用于加载模块文件。**require命令的基本功能是，读入并执行一个JavaScript文件，然后返回该模块的exports对象。如果没有发现指定模块，会报错。**

因为nodejs主要用于服务器编程，模块文件一般已经存在本地硬盘，所以加载起来比较快，不用考虑异步加载的方式，所以使用CommonJS规范。

如果是浏览器环境，要从服务器端加载模块，用CommonJS需要等模块下载完并运行后才能使用，将阻塞后面代码的执行，这时就必须采用非同步模式，因此浏览器端一般采用AMD规范，解决异步加载的问题。

AMD

会编译成 require/exports 来执行

浏览器一般使用AMD规范，解决异步加载问题。

RequireJS是一个工具库。主要用于客户端的模块管理。它可以让客户端的代码分成一个个模块，实现异步或动态加载，从而提高代码的性能和可维护性。它的模块管理遵守AMD规范。

Require.js的基本思想是，通过define方法，将代码定义为模块；通过require方法，实现代码的模块加载。

> 基本语法

定义暴露模块:

```
//定义没有依赖的模块
define(function(){

  return 模块

})

//定义有依赖的模块
define(['module1', 'module2'], function(m1, m2){

  return 模块

})
```

引入使用模块:

```
require(['module1', 'module2'], function(m1, m2){

  使用m1/m2

})
```

- 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；也可以根据需要动态加载模块
- AMD模块定义的方法非常清晰，不会污染全局环境，可清楚地显示依赖关系

CMD

CMD规范专门用于浏览器端，异步加载模块，实用模块时才会加载执行。

整个了CommonJS和AMD规范的特点。

Sea.js中，所有JS模块都遵循CMD模块定义规范。

> 基本语法

定义暴露模块：

```
//定义没有依赖的模块
define(function(require, exports, module){
  exports.xxx = value
  module.exports = value
})

//定义有依赖的模块
define(function(require, exports, module){

  //引入依赖模块(同步)
  var module2 = require('./module2')

    //引入依赖模块(异步)
  require.async('./module3', function (m3) {
 })

  //暴露模块
  exports.xxx = value
})
```

引入使用模块：

```
define(function (require) {

  var m1 = require('./module1')
  var m4 = require('./module4')

  m1.show()
  m4.show()
})
```

ES6

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。

思想是尽量静态化，为了保证在编译时就能确定模块的依赖关系和输入输出的变量。

异步导入，因为用于浏览器，需要下载文件，如果采用同步导入会对渲染有很大影响

采用实时绑定的方式，导入导出的值都指向同一个内存地址，所以导入值会跟随导出值变化

会编译成 require/exports 来执行

- 使用export命令定义了模块的对外接口以后，其他 JS 文件就可以通过import命令加载这个模块。

一个模块就是一个独立的文件，该文件内部的所有变量，外部无法获取。如果你希望外部能够读取模块内部的某个变量，就必须使用export关键字输出该变量。

在编译阶段，import会提升到整个模块的头部，首先执行。

如果不需要知道变量名或函数就完成加载，就要用到export default命令，为模块指定默认输出。

- export default命令用于指定模块的默认输出。显然，一个模块只能有一个默认输出，因此export default命令只能使用一次。所以，import命令后面才不用加大括号，因为只可能唯一对应export default命令。

> 基本语法

export

1.如果不想对外暴露内部变量的真实名称，可以使用as关键字设置别名，同一个属性可以设置多个别名。

在外部进行引入时，可以通过name2这个变量访问到king值。

```
const name='king';

export {name as name2};
```

2.在同一个文件中，同一个变量名只能够export一次，否则会抛出异常。

```
const name='king';

const _name='king';

export {name as _name};

export {name};//抛出异常，name作为对外输出的变量，只能export一次
```

import

1.import和export的变量名相同

2.相同变量名的值只能import一次

3.**import命令具有提升的效果**

```
//export.js
export const name='king';

//import.js
console.log(name);//king

import {name} from './export.js'
```

本质：因为import是在编译期间运行的，在执行console语句之前就已经执行了import语句。因此能够打印出 name 的值，即，King

4.多次import时，只会加载一次

以下代码汇总，我们import了两次export.js文件，但是最终只输出执行了一次“开始执行”，可以推断出import导入的模块是个单例模式。

```
//export.js
console.log('start');

export const name='king';

export const age=19;

//import.js
import {name} from './export.js

import {age} from './export.js''
```

5.允许我们在需要的时候才动态加载模块，而不是一开始就加载所有模块，这样可以帮我们提高性能。

这个新功能允许我们将import()作为函数调用，将其作为参数传递给模块的路径。 它返回一个 promise，它用一个模块对象来实现，可以访问该对象的导出。

```
import('/modules/myModule.mjs')

.then((module) => {
    // Do something with the module.
});
```

对比总结

ES6 输出值的引用（对外接口只是一种静态定义，代码解读阶段生成），CommonJS模块输出值的拷贝（加载一个对象，及module.exports属性，该对象只有在脚本运行时才会生成）

- ES6模块时动态引用，且不会缓存值，模块中的变量绑定其所在的模块

ES6模块编译时输出接口，CommonJS模块运行时加载

1. CommonJS规范主要用于服务端编程，同步加载模块，不适合浏览器环境，因为同步意味阻塞，而浏览器资源时异步加载的，因此诞生了AMD 和 CMD
2. AMD规范在浏览器环境中异步加载模块，且可以并行加载多个模块。但是开发成本高，代码阅读困难，模块定义语义不顺畅
3. CMD和AMD相似，依赖就近，延迟执行，易在nodejs运行。但是，依赖SPM打包，模块加载逻辑偏重
4. **ES6实现模块功能且实现简单，完全可以取代CommonJS和AMD规范，进而成为浏览器和服务器通用模块解决方案的宠儿。**

执行上下文

当执行 JS 代码时，会产生三种执行上下文

- 全局执行上下文
- 函数执行上下文
- eval 执行上下文

在该执行上下文的创建阶段，变量对象、作用域链、闭包、this指向会分别被确定。

对于每个执行上下文，都有三个重要属性：

- 变量对象(Variable object，VO) 包含变量、函数声明和函数的形参，该属性只能在全局上下文中访问
- 作用域链(Scope chain) JS 采用词法作用域，也就是说变量的作用域是在定义时就决定了
- this

全局上下文的变量对象就是全局对象.

JavaScript属于解释型语言，JavaScript的执行分为：解释和执行两个阶段,这两个阶段所做的事并不一样：

解释阶段

- 词法分析
- 语法分析
- 作用域规则确定

执行阶段

- 创建执行上下文
- 执行函数代码
- 垃圾回收

作用域在定义时就确定,而不是在函数调用时确定，不会改变,但是执行上下文是在函数执行前创建的。随时可以改变，**执行上下文最明显的就是this的指向在执行时确定**. 而作用域访问的变量是编写代码的结构确定的。

同一个作用域下，不同的调用会产生不同的执行上下文环境，继而产生不同的变量的值。

JS引擎创建了执行上下文栈管理执行上下文。

> 当 JavaScript 开始要解释执行代码的时候，最先遇到的就是全局代码，所以初始化的时候首先就会向执行上下文栈压入一个全局执行上下文，我们用 globalContext 表示它，并且只有当整个应用程序结束的时候，ECStack 才会被清空，所以程序结束之前， ECStack 最底部永远有个 globalContext：

```
let a = 'Hello World!';

function first() {

  console.log('Inside first function');

    second();

    console.log('Again inside first function');
}

function second() {

  console.log('Inside second function');
}

first();

console.log('Inside Global Execution Context');
```

简单分析一下流程：

- 创建全局上下文压入执行栈
- first函数被调用，创建函数执行上下文并压入栈
- 执行first函数过程遇到second函数，再创建一个函数执行上下文并压入栈
- second函数执行完毕，对应的函数执行上下文被推出执行栈，执行下一个执行上下文first函数
- first函数执行完毕，对应的函数执行上下文被推出栈，然后执行全局上下文
- 所有代码执行完毕，全局上下文被推出栈，程序结束

哪些事件支持冒泡

可以通过 **`event.bubbles`** 属性可以判断该事件是否可以冒泡

事件	是否冒泡
click	可以
dbclick	可以
keydown	可以
keyup	可以
mousedown	可以
mousemove	可以
mouseout	可以
mouseover	可以
mouseup	可以
scroll	可以

概括来说，鼠标事件，和键盘事件，以及点击事件是支持冒泡的

事件	是否冒泡
blur	不可以
focus	不可以
resize	不可以
about	不可以
mouseenter	不可以
mouseleave	不可以
load	不可以
unload	不可以

而聚焦和失焦事件，加载事件，ui事件、鼠标移入移出事件是不支持的。

阻止冒泡

- 阻止冒泡行为：非 IE 浏览器 stopPropagation()，IE 浏览器 window. event. cancelBubble = true
- 阻止默认行为：非 IE 浏览器 preventDefault()，IE 浏览器 window. event. returnValue = false

事件代理

把一个元素响应事件（click、keydown.....）的函数委托到另一个元素，在冒泡阶段完成

相对于直接给目标注册事件，有以下优点

- **节省内存，减少dom操作**
- **不需要给子节点注销事件**
- **动态绑定事件**

事件委托，会把一个或者一组元素的事件委托到它的父层或者更外层元素上，真正绑定事件的是外层元素，而不是目标元素。

适合事件委托的事件有：click，mousedown，mouseup，keydown，keyup，keypress

事件委托

添加到页面上事件处理程序数量将直接关联到页面的整体性能

对“事件处理程序过多”问题的解决方案就是事件委托

利用事件冒泡，只指定一个事件处理程序，就可以管理某一类型所有事件

使用事件委托，只需在DOM树中尽量高的一层添加一个事件处理程序

> 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

- 适合用事件委托的事件：click，mousedown，mouseup，keydown，keyup，keypress。
- mouseover 和 mouseout 虽然也有事件冒泡，但是因为需要经常计算它们的位置，处理起来不太容易
- 不适合的就有很多了，举个例子，mousemove，每次都要计算它的位置，不好把控， focus，blur 之类的，本身就没用冒泡的特性，自然就不用事件委托了

为什么要用事件委托

- 提高性能
- 新添加的元素还会有之前的事件

异步解决方案

同步操作：顺序执行，同一时间只能做一件事情。缺点是会阻塞后面代码的执行。

异步：指的是当前代码的执行作为任务放进任务队列。当程序执行到异步的代码时，会将该异步的代码作为任务放进任务队列，而不是推入主线程的调用栈。等主线程执行完之后，再去任务队列里执行对应的任务。优点是：不会阻塞后续代码的运行。

异步场景

- 定时任务：setTimeout、setInterval
- 网络请求：ajax请求、动态创建img标签的加载
- 事件监听器：addEventListener

回调

回调函数就是我们请求成功后需要执行的函数。

实现了异步，但是带来一个非常严重的问题——回调地狱。

事件发布/订阅

Promise

```
const promise = new Promise((resolve, reject) => {

        resolve('a');
    });

promise

.then((arg) => {
    console.log(`执行resolve,参数是${arg}`)
})
```

```
.catch((arg) => {
    console.log(`执行reject,参数是${arg}`)
})

.finally(() => {
    console.log('结束promise')
});

Promise.reject(2)

//.catch(err=>console.log("err1,",err))
.then(null, err => console.log("err1,", err)) //因为是rejected状态，执行then的第二个
callback，改变状态为fulfilled

.then(res => {
    console.log("then1", res)
}, null) //因为是fulfilled，于是执行第一个回调,不会去到下一步catch

//.catch(err=>console.log("err2,",err))

.then(null, err => console.log("err2,", err))
```

then实现链式操作减低代码复杂度，增强代码可读性。

Promise对象的错误具有“冒泡”性质，会一直向后传递，直到被捕获为止。

每个Promise都会经历的生命周期是：

- 进行中（pending） - 此时代码执行尚未结束，所以也叫未处理的（unsettled）
- 已处理（settled） - 异步代码已执行结束 已处理的代码会进入两种状态中的一种：
  - 已完成（fulfilled） - 表明异步代码执行成功，由resolve()触发
  - 已拒绝（rejected）- 遇到错误，异步代码执行失败 ，由reject()触发

方法

1. Promise.all 传入多个异步请求数组，若all成功，进入fulfilled状态，若一个失败，则立即进入rejected状态。
2. Promise.allSettled 传入多个异步请求数组，无论失败还是失败，都会进入fulfilled状态。
3. Promise.race 以一个Promise对象组成的数组作为参数，只要当数组中一个Promsie状态变成resolved或者rejected时，就调用.then方法。

事件循环

Generator

iterator

ES6推出,更方便创建iterator.Generator是一个返回值为iterator对象的函数.

> **iterator中文名叫迭代器。它为js中各种不同的数据结构(Array、Set、Map)提供统一的访问机制。部署了Iterator接口，就可以完成遍历操作。** 因此iterator也是一种对象，不过相比于普通对象来说，它有着专为迭代而设计的接口。

iterator 的作用：

- 为各种数据结构，提供一个统一的、简便的访问接口；
- 使得数据结构的成员能够按某种次序排列；
- **ES6 创造了一种新的遍历命令for...of循环**

iterator的结构：它有**next**方法，该方法返回一个包含**value**和**done**两个属性的对象（我们假设叫result）。**value**是迭代的值，后者是表明迭代是否完成的标志。true表示迭代完成，false表示没有。iterator内部有指向迭代位置的指针，每次调用**next**，自动移动指针并返回相应的result。

原生具备iterator接口的数据结构如下：

- Array
- Map
- Set
- String
- TypedArray
- 函数里的arguments对象
- NodeList对象

这些数据结构都有一个Symbol.iterator属性，可以直接通过这个属性来直接创建一个迭代器。也就是说，Symbol.iterator属性只是一个用来创建迭代器的接口，而不是一个迭代器，因为它不含遍历的部分。 使用Symbol.iterator接口生成iterator迭代器来遍历数组的过程为：

```
let arr = ['a','b','c'];

let iter = arr[Symbol.iterator]();

iter.next() // { value: 'a', done: false }

iter.next() // { value: 'b', done: false }

iter.next() // { value: 'c', done: false }

iter.next() // { value: undefined, done: true }
```

for ... of的循环内部实现机制其实就是iterator，它首先调用被遍历集合对象的 Symbol.iterator 方法，该方法返回一个迭代器对象，迭代器对象是可以拥有.next()方法的任何对象，然后，在 for ... of 的每次循环中，都将调用该迭代器对象上的 .next 方法。然后使用for i of打印出来的i也就是调用.next方法后得到的对象上的value属性。

对于原生不具备iterator接口的数据结构，比如Object，我们可以采用自定义的方式来创建一个遍历器。

Generator

```
function  * createIterator() {

    yield 1;
    yield 2;
    yield 3;
}

// generators可以像正常函数一样被调用，不同的是会返回一个 iterator
let iterator = createIterator();
```

```
console.log(iterator.next().value); // 1

console.log(iterator.next().value); // 2

console.log(iterator.next().value); // 3
```

Generator 函数是一个普通函数，但是有两个特征:

- function关键字与函数名之间有一个星号
- 使用yield语句，定义不同的内部状态

Generator函数的调用方法与普通函数一样，也是在函数名后面加上一对圆括号。不同的是，调用Generator函数后，该函数并不执行，返回的也不是函数运行结果，而是一个指向内部状态的指针对象，也就是遍历器对象（**Iterator Object**）

```
function  * generator() {

    yield 1;
}

console.dir(generator);
```

generator函数的返回值的原型链上确实有iterator对象该有的next，**这充分说明了generator的返回值是一个iterator**。除此之外还有函数该有的return方法和throw方法。

generator和普通的函数完全不同。当以function*的方式声明了一个Generator生成器时，内部是可以有许多状态的，以yield进行断点间隔。期间我们执行调用这个生成的Generator,他会返回一个遍历器对象，用这个对象上的方法，实现获得一个yield后面输出的结果。

yield和return的区别:

- 都能返回紧跟在语句后面的那个表达式的值
- yield相比于return来说，更像是一个断点。遇到yield，函数暂停执行，下一次再从该位置继续向后执行，而return语句不具备位置记忆的功能。
- 一个函数里面，只能执行一个return语句，但是可以执行多次yield表达式。
- 正常函数只能返回一个值，因为只能执行一次return；Generator 函数可以返回一系列的值，因为可以有任意多个yield

语法注意点:

- yield表达式只能用在 Generator 函数里面
- yield表达式如果用在另一个表达式之中，必须放在圆括号里面
- yield表达式用作函数参数或放在赋值表达式的右边，可以不加括号。

使用Generator的其余注意事项:

- 需要注意的是，yield 不能跨函数。并且yield需要和*配套使用，别处使用无效

```
function  * createIterator(items) {

    items.forEach(function (item) {

        // 语法错误
        yield item + 1;
    });
}
```

- 箭头函数不能用做 generator

那么Generator到底有什么用呢?

- 因为Generator可以在执行过程中多次返回，所以它看上去就像一个可以记住执行状态的函数，利用这一点，写一个generator就可以实现需要用面向对象才能实现的功能。
- Generator还有另一个巨大的好处，就是把异步回调代码变成“同步”代码

async/await

ES7提出的关于异步的终极解决方案

- 第一个版本说async/await是Generator的语法糖
- 第二个版本说async/await是Promise的语法糖

关于async/await是Generator的语法糖： 表明的就是aysnc/await实现的就是generator实现的功能。但是async/await比generator要好用。

因为generator执行yield设下的断点采用的方式就是不断的调用iterator方法，这是个手动调用的过程。

针对generator的这个缺点，后面提出来自动执行next，但是仍然麻烦。而async配合await得到的就是断点执行后的结果。

因此async/await比generator使用更普遍。

async函数对 Generator函数的改进：

1. 内置执行器：Generator函数的执行必须靠执行器，因为不能一次性执行完成。但是，async函数和正常的函数一样执行，也不用使用 next方法，会自动执行。
2. 适用性更好：yield命令后面是 Promise对象，但是 async函数的 await关键词后面，可以不受约束。
3. 可读性更好：async和 await，比起使用 *号和 yield，语义更清晰明了。

关于async/await是Promise的语法糖： 如果不使用async/await的话，Promise就需要通过链式调用来依次执行then之后的代码:

而async/await的出现，就使得我们可以通过同步代码来达到异步的效果:

由此可见，Promise搭配async/await的使用才是正解！

async/await其实是基于Promise的.async把promise包装了一下,使用async函数可以让代码简介很多,不需要像promise一样需要写then,不需要写匿名函数处理promise的resolve值,也不需要定义多余的data变量,还避免了嵌套代码。

async函数时Generator函数的语法糖,async函数的返回值是promise对象,generator函数的返回值是 iterator对象方便多了,同时,还可以使用 await 代替then 方法指定下一步操作。

```
function f() {

    return Promise.resolve('TEST');
}

// asyncF is equivalent to f!
async function asyncF() {

    return 'TEST';
}
```

> **await 会阻塞下面的代码（即加入微任务队列），先执行 async外面的同步代码，同步代码执行完，再回到 async 函数中，再执行之前阻塞的代码**

定时器原理

- var id=setTimeout(fn,delay); 初始化一个只执行一次的定时器，这个定时器会在指定的时间延迟 delay 之后调用函数 fn ,该 setTimeout 函数返回定时器的唯一 id ，我们可以通过这个 id 来取消定时器的执行。
- var id=setInvertal(fn,delay); 与 setTimeout 类似，只是它会以 delay 为周期，反复调用函数 fn ，直到我们通过 id取消该定时器。
- clearInterval(id),clearTimeout(id); 这两个函数接受定时器的 id，并停止对定时器中指定函数的调用。

> **定时器指定的延迟时间不能得到保证**

在浏览器中，因为所有的JS代码都运行在单一线程中，异步事件只有在被触发时，其回调才会得到执行。

因为单线程的缘故，在同一时间只能执行一条 JavaScript 代码，每一个代码块都会阻塞其他异步事件的执行。

这就意味着，当一个异步事件发生的时候（例如鼠标点击，定时器触发，一个 XMLHttpRequest 请求完成），它进入了代码的执行队列，执行线程空闲时会依照该执行队列中顺序依次执行代码。

总结：

- JavaScript 引擎是单线程的，会迫使异步事件进入执行队列，等待执行。
- setTimeout 和 setInterval 在执行异步代码时从根本上是有所不同的。
- 如果一个定时器事件被阻塞，使得它不能立即执行，那么它会被延迟，直到下一个可能的时间点，才被执行（这可能比你指定的 delay 时间要长）
- Interval 的回调有可能‘背靠背’无间隔的执行，这种情况是说 interval 的回调函数的执行时间比我们指定的 delay 时间还要长

内存管理GC

栈中JS引擎自动清除.

栈内存中变量一般在它的当前执行环境结束就会被销毁被垃圾回收制回收， 而堆内存中的变量则不会，因为不确定其他的地方是不是还有一些对它的引用。 堆内存中的变量只有在所有对它的引用都结束的时候才会被回收。

JS 引擎可以通过**逃逸分析**辨别出哪些变量需要存储在堆上，哪些需要存储在栈上。

自动垃圾回收机制：找出不使用的值，释放内存。

函数运行结束，没有闭包或引用，局部变量被 标记 清除

全局变量：浏览器卸载页面 被清除

垃圾回收算法

不论哪个垃圾回收算法，都有一套共同的流程：

1. 标记内存空间中的活动对象（在使用中的对象）和非活动对象（可以回收的对象）。
2. 删除非活动对象，释放内存空间。
3. 整理内存空间，避免频繁回收后产生的大量内存碎片（不连续内存空间）。

引用计数

一个对象是否有被引用（循环引用导致内存泄露）

策略:跟踪每个变量被使用的次数

1. 当声明了一个变量且将一个引用类型赋值给该变量时,这个值的引用次数为1
2. 若同一个值又被赋给另一个值,引用数 +1
3. 如果该变量的值被其他的值覆盖了，则引用次数减 1
4. 当这个值的引用次数变为 0 的时候，说明没有变量在使用，这个值没法被访问了，回收空间，垃圾回收器会在运行的时候清理掉引用次数为 0 的值占用的内存

缺点:

- 需要一个计数器，所占内存空间大，因为我们也不知道被引用数量的上限。
- 解决不了循环引用导致的无法回收问题。

标记清除

将“不再使用的对象”定义为“无法到达的对象”

工作流程：

1. 垃圾收集器在运行时给内存变量加上 标记,假设内存汇总所有对象都是垃圾,全标记为0
2. 从根部出发，寻找可到达的变量，并将其标记清除,改为1
3. 留有标记的变量就是待删除的,即标记为0,销毁并回收它们占用的内存
4. 把所有内存中对象标记修改为0,等待下一轮垃圾回收

优点:实现简单,一位二进制位就可以为其标记

缺点:

- 内存碎片化,空闲内存块不连续
- 分配速度慢,因为即使使用first-fit策略,其操作仍是一个O(n)的操作,最坏情况是每次都要遍历到最后,因为碎片化,大对象的分配速率会更慢

复制算法

为了解决上述问题，复制算法出现了。

1. 将整个空间平均分成 from 和 to 两部分。
2. 先在 from 空间进行内存分配，当空间被占满时，标记活动对象，并将其复制到 to 空间。
3. 复制完成后，将 from 和 to 空间互换。

由于直接将活动对象复制到另一半空间，没有了清除阶段的开销，所以能在较短时间内完成回收操作，并且每次复制的时候，对象都会集中到一起，相当于同时做了整理操作，避免了内存碎片的产生。

优点：吞吐量高、没有碎片

缺点：首先，复制操作需要时间成本的，若堆空间很大且活动对象很多，则每次清理时间会很久；其次，将空间二等分的操作，让可用的内存空间直接减少了一半。

标记整理

也叫做 标记-压缩算法。结合了标记-清除和复制算法的优点。

1. 从一个 GC root 集合出发，标记所有活动对象。
2. 将所有活动对象移到内存的一端，集中到一起。
3. 直接清理掉边界以外的内存，释放连续空间。

该算法既避免了标记-清除法产生内存碎片的问题，又避免了复制算法导致可用内存空间减少的问题。当然，该算法也不是没有缺点的，由于其清除和整理的操作很麻烦，甚至需要对整个堆做多次搜索，故而堆越大，耗时越多。

识别内存泄露

经验法则是，如果连续五次垃圾回收之后，内存占用一次比一次大，就有内存泄漏。

这就要求实时查看内存的占用情况。

在 Chrome 浏览器中，我们可以这样查看内存占用情况

1.打开开发者工具，选择 Performance 面板

2.在顶部勾选 Memory

3.点击左上角的 record 按钮

4.在页面上进行各种操作，模拟用户的使用情况

5.一段时间后，点击对话框的 stop 按钮，面板上就会显示这段时间的内存占用情况

造成内存泄露

1. **意外的全局变量**
2. **被遗忘的定时器和回调函数**
3. **事件监听没有移除**
4. **没有清理的DOM 引用**
5. **子元素存在的内存泄漏**
6. **闭包**

> **V8对GC优化**

- 栈中数据回收：执行状态指针 ESP 在执行栈中移动，移过某执行上下文，就会被销毁；
- 堆中数据回收：V8 引擎采用标记-清除算法；
- **V8 把堆分为两个区域——新生代和老生代，分别使用副、主垃圾回收器；**
- 副垃圾回收器负责新生代垃圾回收，小对象（1 ~ 8M）会被分配到该区域处理；
- 新生代采用 scavenge 算法处理：将新生代空间分为两半，一半空闲，一半存对象，对对象区域做标记，存活对象复制排列到空闲区域，没有内存碎片，完成后，清理对象区域，角色反转；
- 新生代区域两次垃圾回收还存活的对象晋升至老生代区域；

- 主垃圾回收器负责老生区垃圾回收，大对象，存活时间长；
- 新生代区域采用标记-清除算法回收垃圾：从根元素开始，递归，可到达的元素活动元素，否则是垃圾数据；
- **为了不造成卡顿，标记过程被切分为一个个子标记，交替进行。**

分代式垃圾回收

以上所说的垃圾清理算法每次垃圾回收时都要检查内存中所有的对象,酱紫对一些大,老,存活时间长的对象来说，同新,小,存活时间短的对象一个频率的检查很不好，因为前者需要时间长且不需要频繁进行清理，后者恰恰相反,如何优化?

分代式

分代式机制把一些新、小、存活时间短的对象作为新生代，采用一小块内存频率较高的快速清理，而一些大、老、存活时间长的对象作为老生代，使其很少接受检查，新老生代的回收机制及频率是不同的，可以说此机制的出现很大程度提高了垃圾回收机制的效率

新老生代

V8的GC策略基于分代式垃圾回收机制,将堆内存分为新生代和老生代两区域,采用不同的垃圾回收策略进行垃圾回收.

新生代的对象是存活时间较短的对象,简单来说就是新产生的对象,通常只支持1-8M的容量,而老生代的对象为存活时间较长或常驻内存的对象.

新生代垃圾回收

新生代中对象一般存活时间较短，采用 scavenge 算法处理,在Scavenge算法具体实现中,主要采用一种**复制式**的方法及Cheney算法：

其将新生代空间对半分为 from-space 和 to-space 两个区域。新创建的对象都被存放到 from-space，当空间快被写满时触发垃圾回收。先对 from-space 中的对象进行标记，完成后将标记对象复制到 to-space 的一端，然后将两个区域角色反转，就完成了回收操作。

由于每次执行清理操作都需要复制对象，而复制对象需要时间成本，所以新生代空间会设置得比较小（1~8M）。

当一个对象经过多次复制后依然存活，它将会被认为是生命周期较长的对象，随后会被移动到老生代中，采用老生代的垃圾回收策略进行管理

另外还有一种情况，如果复制一个对象到空闲区时，空闲区空间占用超过了 25%，那么这个对象会被直接晋升到老生代空间中，设置为 25% 的比例的原因是，当完成 Scavenge 回收后，空闲区将翻转成使用区，继续进行对象内存的分配，若占比过大，将会影响后续内存分配

老生代垃圾回收

老生代中的对象一般存活时间较长且数量也多，使用了两个算法，**分别是标记清除算法和标记压缩算法**。

什么情况下对象会出现在老生代空间中：

- 新生代中的对象是否已经经历过一次 Scavenge 算法，如果经历过的话，会将对象从新生代空间移到老生代空间中。
- To 空间的对象占比大小超过 25 %。在这种情况下，为了不影响到内存分配，会将对象从新生代空间移到老生代空间中。

首先是标记阶段，从一组根元素开始，递归遍历这组根元素，遍历过程中能到达的元素称为活动对象，没有到达的元素就可以判断为非活动对象

清除阶段老生代垃圾回收器会直接将非活动对象清理掉

前面我们也提过，标记清除算法在清除后会产生大量不连续的内存碎片，过多的碎片会导致大对象无法分配到足够的连续内存，而 V8 中就采用了我们上文中说的标记整理算法来解决这一问题来优化空间

在老生代中，以下情况会先启动标记清除算法：

- 某一个空间没有分块的时候
- 空间中被对象超过一定限制
- 空间不能保证新生代中的对象移动到老生代中

> 由于JS是单线程运行的，意味着垃圾回收算法和脚本任务在同一线程内运行，在执行垃圾回收时，后续脚本任务需要等垃圾回收完成后才能继续执行。若堆中的数据量非常大，一次完整垃圾回收的时间会非常长，导致应用的性能和响应能力直线下降。为了避免垃圾回收影响应用的性能，V8 将标记的过程拆分成多个子标记，让垃圾回收标记和应用逻辑交替执行，避免脚本任务等待较长时间。

闭包和内存泄漏

有权访问另外一个函数作用域中变量的函数。

闭包是指那些能够访问自由变量的函数. 自由变量是指在函数中使用的，但既不是**函数参数**也不是**函数的局部变量**的变量。 闭包 = 函数 + 函数能够访问的自由变量。

每一个函数都会拷贝上级作用域，形成一个作用域链条。

闭包：自由变量的查找，是在函数定义的地方，向上级作用域查找。不是在执行的地方。

闭包的变量存在堆中，即解释了函数调用之后为什么闭包还能引用到函数内的变量。

闭包的形成需要两个条件：

- 闭包是在函数被调用执行的时候才被确认创建的。
- 闭包的形成，与作用域链的访问顺序有直接关系。
- 只有内部函数访问了上层作用域链中的变量对象时，才会形成闭包，因此，我们可以利用闭包来访问函数内部的变量。

> 闭包，本质上是上级作用域内变量的生命周期，因为被下级作用域引用，没有得到释放，需要等到下级作用域执行完后才得到释放。

作用

1. 独立作用域，避免变量污染
2. 实现缓存计算结果
3. 库的封装 jQuery

运用

防抖节流

防抖：事件触发高频到最后一次操作，如果规定时间内再次触发，则重新计时。

```
function debounce(fn, delay = 300) {
```

```
    let timer; //闭包引用外界变量

    return function () {

        consr args = arguments;

        if (timer) {
            clearTimeout(timer);
        }

        timer = setTimeout(() => {
                fn.apply(this, args);
            }, delay);
    };
}
```

模拟块级作用域

```
function OutPutNum(cnt) {

    (function () {

        for (let i = 0; i < cnt; i++) {

            alert(i);
        }

    })();

    alert(i);
}
```

对象中创建私有变量

浮点数求和步骤

1. 对阶
2. 求和
3. 规格化

对阶

判断指数位是否相同，即小数位是否对其，若指数位不同，需要对阶保证指数相同。

对阶时遵守小阶向大阶看齐原则，尾数向右移位，每移动一位，指数位加 1 直到指数位相同，即完成对阶。

本示例，0.1 的阶码为 -4 小于 0.2 的阶码 -3，故对 0.1 做移码操作

```
// 0.1 移动之前
0 + 011 1111 1011+ 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1010

// 0.1 右移 1 位之后尾数最高位空出一位，（0 舍 1 入，此处舍去末尾 0）
0 + 011 1111 1100 +1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 101(0)

// 0.1 右移 1 位完成
0 +011 1111 1100 +1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1101
```

尾数求和

```
// 0.1 和 0.2 都转化成二进制后再进行运算
0+011 1111 1100+1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1101 //
0.1
(+) 0+011 1111 1100+1001  1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1010
// 0.2
= 0+011 1111 1100+10 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0111
// 产生进位，待处理
```

规格化和舍入

由于产生进位，阶码需要+1，原阶码为011 1111 1100，+1后得到1000 10，转换为十进制，即1021，此时阶码为1021-1023= -2，此时符号位，指数位分别为0 + 011 1111 1101

尾部进位2位，去除最高位默认的1，因为最低位为1需要进行舍入操作，即在最低有效位上+1，若为0则直接舍去，若为1继续+1

```
  10 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0111 // + 1
=  0 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 1000 // 去除最高位默认
的 1
=  0 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 1000 // 最后一位 0 舍去
=  0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0100  // 尾数最后结果
```

因此 IEE 754最终存储如下：

```
0+011 1111 1101+0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0100
```

最高位为1，得到二进制

```
2^-2 * 1.0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0100
```

转换为十进制

```
0.3 0000 0000 0000 0004
```

总结

1. 精度损失 0.1和0.2转换为二进制出现无限循环情况，JS以64位双精度格式存储数字，最大可存储53位有效数字，

超过此长度会被截取掉，造成精度损失
2. 对2个64位双精度格式数据计算时，首先进行对阶处理(将阶码对齐，将小数点位置对齐),因此，小阶数在对齐时，有效数字会向右移动，超过有效位数的位被截取掉
3. 当两个数据阶码对齐后进行加运算，得到的结果可能超过53位有效数字，超过的位会被截取掉

相加后因浮点数小数位限制截断的二进制数字转换为十进制时变成0.30000000000000004(15个0)

如何让其相等

- 设置一个误差范围(将结果与右边相减，若结果小于一个极小数，则正确)

极小数可以是 ES6 的 Number.EPSILON实质是一个可接受的最小误差范围, 一般来说为 Math.pow(2, -52)

```
function isEqual(a, b) {
    return Math.abs(a - b) < Number.EPSILON;
}
console.log(isEqual(0.1 + 0.2, 0.3)); // true
```

- 转成字符串,对字符串做加法运算

```
parseFloat((0.1 + 0.2).toFixed(10))
//toFixed四舍五入
```

- toPrecision转换成数字

```
function strip(num, precision = 12) {
    return parseFloat(num.toPrecision(precision));
}
let x = strip(0.30000000000000004, 18)
    console.log(x)
    //toPrecision以指定的精度返回该数值对象的字符串表示，四舍五入到 precision 参数指定的显示数字位数。
    //默认去掉最低位的0
```

- 将计算数字提升10的N次方，即，将其转换为整数，结果转换为对应小数

```
(0.1*1000+0.2*1000)/1000==0.3
//true
```

十进制小数转二进制

十进制小数转二进制，具体做法是：用2乘十进制小数，可以得到积，将积的整数部分取出，再用2乘余下的小数 部分，又得到一个积，再将积的整数部分取出，如此进行，直到积中的小数部分为零，或者达到所要求的精度为止。

然后把取出的整数部分按顺序排列起来，先取的整数作为二进制小数的高位有效位，后取的整数作为低位有效位。**（乘2取整，顺序排列）**

举个例子：

27.0转化成二进制为11011.0

```
先把0.5转换为二进制小数
0.5*2=1.0 //积中小数部分为0

(27) 10=(11011) 2
(0.5) 10=(1.0) 2
合并整数和小数部分可得
(27.5) 10=(11011.1) 2
```

现在回到我们的主题，那么0.1转为二进制是多少呢?

```
0.1 * 2 = 0.2 ---------------- 取整数 0，小数 0.2
0.2 * 2 = 0.4 ---------------- 取整数 0，小数 0.4
0.4 * 2 = 0.8 ---------------- 取整数 0，小数 0.8
0.8 * 2 = 1.6 ---------------- 取整数 1，小数 0.6
0.6 * 2 = 1.2 ---------------- 取整数 1，小数 0.2
0.2 * 2 = 0.4 ---------------- 取整数 0，小数 0.4
0.4 * 2 = 0.8 ---------------- 取整数 0，小数 0.8
0.8 * 2 = 1.6 ---------------- 取整数 1，小数 0.6
0.6 * 2 = 1.2 ---------------- 取整数 1，小数 0.2
...
```

所以0.1的二进制可以表示为0.000110011……(0011无限循环)，因此二进制无法精确保存类似0.1这样的小数。

因为javaScript存储方式是IEEE 754 标准浮点数表示常用的 双精度浮点数 表示法，其长度为8个字节，即64位比特

64位比特又可分为三个部分：

- 符号位S：第 1 位是正负数符号位（sign），**0代表正数，1代表负数**
- **指数位E**：阶码，中间的 11 位存储指数（exponent），用来表示次方数，可以为正负数。在双精度浮点数中，**指数的固定偏移量为1023**
- 尾数位M：最后的 52 位是尾数（mantissa），超出的部分自动进一舍零。**(能够真正决定数字精度)**

0.1+0.2 为什么不等于 0.3

看下面的输出：

```
console.log(0.1 + 0.2)  // 结果是0.30000000000000004，而不是3
```

那么为什么不是呢~主要是因为在 JavaScript 中，数字是采用的IEEE 754的双精度标准进行存储。比如其中小数使用64位固定长度来表示的，其中的1位表示符号位，11位用来表示指数位，剩下的52位尾数位。

而其中，比如 0.1 转换为二进制是一个无限循环的数的，就会超过 52 位，就会导致精度缺少，所以在计算机中 0.1 只能存储成一个近似值

所以同理的，**`0.1 + 0.2` 会先分别转换为二进制然后进行相加后存储，最后取出来的时候转化为十进制，而这个值不够近似于`0.3`，所以就会出现不相等的结果。**

那么如何避免呢?

常用简单的办法就是将浮点数转变为整数来计算。

UTF-8、UTF-16 和 Unicode

Unicode是一个字符集，它为每种语言中的每个字符设定了统一并且唯一的二进制编码，以满足跨语言、跨平台进行文本转换、处理的要求。

UTF-8,和UTF-16,是Unicode的实现方式，一个文字的Unicode码长度可以为1，2，4个字节，一个汉字2个字节不够时使用4个字节。

utf16：一个存储单位16bit，也就是2个字节，无符号整数，一个汉字可能占用不同个存储单元 。弊端：A一个字节就可以存储，utf16的话需要字节对齐，也就是2个字节，因此utf-8出现

utf-8：可变长的编码方式，可以用1~4个字节表示一个字符。

> 优点：节省方案，方便解析为各种类型，根据文字编码范围

base64怎么编码的

Base64就是一种基于64个可打印字符来表示二进制数据的编码方法。

为什么不直接用ASCII码呢？因为我们输入的字符可能有ASCII码中不可见的字符，为了完全可见，就用了base64编码。

逻辑与 && 和 逻辑或 ||

一、逻辑与 && 和 逻辑或 ||

当我们操作数是布尔值的时候，逻辑与需要操作都为 true 才会返回 true，而逻辑或只需要操作数中有一个为 true 就会返回 true， 但是要是操作数不是布尔值呢?

二、下面语句的输出是?

```
console.log(0&&1)  //0
console.log(1&&2)  //2
console.log(0||1)  // 1
console.log(1||2)  //1
```

首先我们先看结论，再来分析为什么。

- 逻辑与操作符是一种短路操作符，也就是如果第一个操作数结果为 `false`，就不会对第二个操作数求值
- 逻辑或操作符也是一种短路操作符，但不同的是当第一个操作结果为 `true`，就不会对第二个操作符求值
- 只要“&&”前面是false，无论“&&”后面是true还是false，结果都将返“&&”前面的值;
- 只要“&&”前面是true，无论“&&”后面是true还是false，结果都将返“&&”后面的值;
- 只要“||”前面为false,不管“||”后面是true还是false，都返回“||”后面的值。
- 只要“||”前面为true,不管“||”后面是true还是false，都返回“||”前面的值。

需要补充的是在 在 JavaScript 逻辑运算中，0、””、null、false、undefined、NaN都会判为false，其他都为true

三、短路的特性的作用

1. 在使用中可以避免变量赋值为 null 或者 undefined ， 1

```
let myObject = perferredObject || backupObject
```

2. 还可以做空值判断，因为在我们的一些对象中是没发判断是否存在的，获取不到的，所以需要做空值判断

```
{{ data.owner && data.owner.avatar }}
```

中文是多少长度?

```
conlog.log("你好呀大笨蛋".length)

// 6
```

中文在数据库中存放是占两个字符的，但是在浏览器中，由于 javascript 是 unicode 编码的，所有的字符对于它来说一个就是一个，获取的是中文的长度而不是字符的长度，所以上面的输出是 6 。

这就会导致前后端的对于中文的验证长度不一样了，如何解决?

```
function getRealLength( str ) {
    return str.replace(/[^\x00-\xff]/g, '__').length; //这个把所有双字节的都给匹配进去了
}
```

简单了解 null 和 undefined

一句简单来说是：**undefined 代表了不存在的值，null 代表了没有值。**

也就是，比如对一个值声明后，没有赋值，输出他就是 undefined ，是不存在的，而当赋值为 null， 那么输出就是 null。

这两者还有一些区别点要注意：

```
//1
undefined == null //true，这里相等是因为 “==” 对他们做了转换
undefined === null //false
//undefined的值 是 null 派生来的，所以他们表面上是相等的

//2
let a;
typeof a; //undefined

let b = null;
typeof b; //object
```

这里为什么typeof b 输出为 Object 呢?

null 不是一个对象，尽管 typeof age输出的是 Object，逻辑上讲，null 值表示一个空对象指针，这是一个历史遗留问题，JS 的最初版本中使用的是 32 位系统，为了性能考虑使用低位存储变量的类型信息，000 开头代表是对象，null表示为全零，所以将它错误的判断为 Object。

undefined 和 undeclared 的区别

Undeclared（未声明）：当尝试访问尚未使用 var、let 或 const 声明的变量时会发生这种情况。

Undefined（未定义）：它发生在使用 var、let 或 const 声明变量但未赋值时。

在 JavaScript 中，两者很容易被混为一谈

typeof 对 undefined 和 undeclared 的 都返回 “undefined”

```
let a;
typeof a; // undefined
typeof b // undefined （实际上这里应该输出 undeclared 才是合适的）
```

我们访问 undeclared 变量的时候,是这样报错的：ReferenceError： a is not defined。

```
var a;
console.log(a); //undefined
console.log(b); //ReferenceError: b is not defined
```

Element、Node

Node是基类，Element、Text都继承于它

Element、Text分别叫做ELEMENT_NODE，TEXT_NODE

html上的元素，即element，是类型为ELEMENT_NODE的Node

Node表示DOM树的结构，html中节间可以插入文本，这个插入的空隙就是TEXT_NODE

可使用childNodes得到NodeList，如何获取ElementList?

getElementByXXX返回ElementList，它的真名是 ElementCollection

就像NodeList是Node的集合一样，ElementCollection也是Element的集合

他们都不是真正的数组

HTMLCollection、NodeList

NodeList

NodeList 对象是节点的集合，通常由属性，如Node.childNodes和方法，如 document.querySelectorAll返回

不是数组，是类数组对象

可以使用forEach迭代，使用Array.from()转换为数组

一些情况下，NodeList动态变化，如果文档节点树变化，NodeList会随之而变，Node.childNodes是实时的

其他情况，NodeLIst是静态集合，document.querySelectorAll返回静态NodeList

可使用for循环 或 for-of 遍历，不要使用 for-in 遍历NodeList，因为NodeList对象的length和item属性会被遍历出来，可能导致错误，且for-in无法保证属性顺序

NodeList可包含任何节点类型

HTMLCollection

一个类数组对象，包含元素的通用集合（元素顺序为文档流中顺序）

live——即时更新

因此，最好创建副本后再迭代 以 add 、delete 或是 move节点

getElementByTagName()返回HTMLCollection对象

HTMLCollection有namedITem方法，其他和NodeList保持一致

HTMLCollection只包含元素节点（ElementNode）

JavaScript 中日期时间格式转换

获取当天时间：

```
//获取当前时间
var date = new Date();
var year = date.getFullYear();
var month = date.getMonth() + 1;
var day = date.getDate();
//这两个是针对小于十的时候，会为个位数
```

```
if (month < 10) {
    month = "0" + month;
}
if (day < 10) {
    day = "0" + day;
}
var nowDate = year + "-" + month + "-" + day;

//往往格式成这种样式是后台处理好的，或者我们会使用一些插件，又或者我们在vue中会使用过滤器来处理。
//输出结果格式：2021-11-08
```

一、基本的函数：

- getDate() 返回一个月中的某一天(1-31)
- getMonth() 返回月份 (0 ~ 11)
- getFullYear() 以四位数字返回年份(2011)
- getHours() 返回小时 (0 ~ 23)
- getMinutes() 返回分钟 (0 ~ 59)
- getSeconds() 返回秒数 (0 ~ 59)
- getTime() 返回 1970 年 1 月 1 日至今的毫秒数(1566715528024)

二、获取时间戳的方法

```
let timestamp =(new Date()).valueOf();
//或者
let timestamp=new Date().getTime();
```

三、练习题

问题：返回“现在距离 今年元旦 还有 X 天 X 小时 X 分钟 X秒”

```
function timemove() {
    // 获取当前时间
    let d1 = new Date();
    //获取下一年
    let nextYear = d1.getFullYear() + 1;
    // 定义目标时间
    let d2 = new Date(nextYear + "/1/1 00:00:00");
    // 定义剩余时间
    let d = d2 - d1; // 是一个时间戳
    // 计算剩余天数
    let Day = parseInt(d / 1000 / 60 / 60 / 24);
    // 计算剩余小时
    let Hours = parseInt(d / 1000 / 60 / 60 % 24);
    // 计算剩余分钟
    let Minutes = parseInt(d / 1000 / 60 % 60);
    // 计算剩余秒
    let Seconds = parseInt(d / 1000 % 60);
    //拼接变量
    let time = Day + "天" + Hours + "小时" + Minutes + "分钟" + Seconds + "秒";
    // 将拼接好的变量显示在页面
```

```
    console.log("距离元旦还有" + time)
}
```

dom.onclick 和 dom.addEventListener的区别

一、两者基本信息：

dom.onclick：onclick 事件会在元素被点击时发生，可以在 HTML 和 JavaScript 中使用，所有的浏览器都支持 onclick 事件。

> onclick 属性可以使用与所有 HTML 元素，除了：< base >, < bdo >, < br >, < head >, < html >, < iframe >, < meta >, < param >, < script >, < style >, 和 < title >。

dom.addEventListener：document.addEventListener() 方法用于向文档添加事件句柄。具有三个参数，分别是事件，触发执行的函数，以及是否在在捕获或冒泡阶段执行

> Internet Explorer 8 及更早IE版本不支持 addEventListener() 方法，Opera 7.0 及 Opera 更早版本也不支持。但是可以使用 attachEvent() 方法来添加事件句柄

二、三者的区别：

事件监听器（addEventListener / attachEvent）

```
 < a href = "//google.com" > Try clicking this link. <  / a >

    const element = document.querySelector('a');

element.addEventListener('click', event => event.preventDefault(
        console.log("Hello World"); ));

element.addEventListener('click', event => event.preventDefault(
        console.log("How are you?"); ));

element.click()
    // "Hello World"
    // "How are you?"
```

- 如果不考虑性能等因素，可以添加无限数量的事件到该元素上，并且可以通过 removeEventListener() 移除事件
- 不能在 HTML 上使用，只能在< script >元素中添加。
- 有useCapture 参数，你可以选择是在捕获还是在冒泡阶段执行
- 几乎适合所有浏览器，要是需要适应早期的，可以使用 attachEvent
- 分离文档结构 (HTML) 和逻辑 (JavaScript)，在大型的项目中开发便于维护
- 更加好理解，获取元素，添加监听事件，如果发生就触发执行函数。

内联事件监听器

```
< a href = "//google.com" onclick = "event.preventDefault();" > Try clicking this
link. </a>
```

- 只能添加一个事件在之上
- onclick 可以作为HTML属性添加，也能在< script >元素中添加。
- 不能在这里使用匿名函数或者闭包
- 是一个潜在的安全问题，因为它使 XSS 更加有害。如今，网站应该发送适当的Content-Security-PolicyHTTP 标头来阻止内联脚本，并只允许来自受信任域的外部脚本。
- 不分离文档结构和逻辑。
- 如果您使用服务器端脚本生成页面，例如您生成一百个链接，每个链接都具有相同的内联事件处理程序，那么您的代码将比事件处理程序仅定义一次时长得多。这意味着客户端必须下载更多内容，结果您的网站会变慢。
- 不能选择是在捕获还是在冒泡阶段执行

相当于内联 JavaScript

```
 < a href = "//google.com" > Try clicking this link. <  / a >
    const element = document.querySelector('a');
element.onclick = event => event.preventDefault1();
element.onclick = event => event.preventDefault2(); //会覆盖上面的执行函数
```

相比上面的方式，基本类似，但因为相当于内联JavaScript，也就是我们是在编写脚本，而不是HTML，就使得我们可以定义实现的更多，也就是可以使用匿名函数、函数引用和/或闭包。但这种方式存在被覆盖的可能性，也就是我们定义的可能被覆盖

三、总结

总的来说没有说那种是最好的，最好的往往是根据实际的情况选择的，但是一般建议使用 addEventListener，因为它可以做任何事情 onclick 能做的，甚至更多。但是可以说 onclick 基本是始终都会有效的，而 当我们 addEventListener 需要考虑是否支持。

并且不要使用内联 onclick 来用作 HTML 属性，因为这会混淆 javascript 和 HTML，这是一种不好的做法，它使得代码的可维护性降低。应该说如果没有很好的理由，所有的内联形式都是不建议的。

... 与rest

Rest为解决传入的参数数量不一定；不会为每个变量给一个单独的名称，参数对象包含所有参数传递给函数；arguments不是真正的数组，rest参数是真实的数组

剩余参数只包含那些没有对应形参的实参，arguments包含传给函数的所有实参

... 应用

ES6通过扩展元素符...，好比 rest 参数的逆运算，将一个数组转为用逗号分隔的参数序列

函数调用的时候，将一个数组变为参数序列

将某些数据结构转为数组

合并数组

注意：通过扩展运算符实现的是浅拷贝，修改了引用指向的值，会同步反映到新数组

rest特点

减少代码

```
//以前函数
function f(a, b) {
    var args = Array.prototype.slice.call(arguments, f.length);
}

// 等效于
function f(a, b, ...args)  {}
```

rest参数可以被解构

freeze属性

我们常常用到 freeze 来冻结对象，那么 freeze 属性是如何实现冻结对象，使之不能修改的呢?

一、这里是关于属性的四个基本类型

1、[[Configurable]]:
表示属性是否可以通过 delete 删除并重新定义，是否可以修改它的特性，以及是否可以把它改为访问器属性。默认为 true 。

2、[[Enumerable]]:
表示属性是否可以通过for-in循环返回。默认为 true 。

3、 [[Writable]]:
表示属性的值是否可以被修改。默认为 true 。

4、[[Value]]:
表示属性实际的值。默认为 undefined 。

当我们显式的添加一个属性之后，它的值就会被设置到 [[Value]] 中，而 [[Writable]]表示我们能否读这个值进行修改，如果设置为 false ，那么当我们修改这个值的时候回就会报错，同理其他的属性也是一样的。而如果我们要修改属性的类型的值的时候 ，是使用 Object.defineProperty()方法 来进行修改的。

二、那么 **freeze** 做了什么?

那么回到我们的前面开篇的问题，当我们使用了 feeeze 的时候，它就会做下面的事情:

- 设置 Object.preventExtension()，禁止添加新属性 (绝对存在)
- 设置 Writable 为 false，禁止修改 (绝对存在)
- 设置 Configurable 为 false，禁止配置 (绝对存在)
- 禁止更改访问器属性 (getter 和 setter)

> 我们有时候会看到有一些属性是用两个中括号包含起来的，这是怎么回事呢? 这是因为: **当为了将某个特征标识为内部特性，规范会用两个中括号把特性的名称括起来**

parseInt&parseFloat

+和Number()转换数字是严格的，若一个值不完全是数字，就会失败

```
console.log(+"100px");//NaN

console.log(+" ");//0
```

从"100px"，"12pt"中将数值提取出来，parseInt和parseFloat派上用场，它们可以从字符串中读取数字，直到无法读取位置，若出现error，则返回收集到的数字

```
parseInt("100px");//100

parseInt("11.22px");//11

parseFloat("12.33px");//12.33

parseFloat("12.3.4");//12.3
```

没有数字可读取时，返回NaN

```
parseInt("a123");//NaN

parseInt(str,radix)

alert( parseInt('0xff', 16) ); // 255

alert( parseInt('ff', 16) ); // 255, 没有 0x 仍然有效

alert( parseInt('2n9c', 36) ); // 123456
```

input和object实现双向绑定

Object.defineProperty()直接在一个对象上定义一个新的属性,或者修改一个已经存在的属性

Object.defineProperty(obj, props, desc), 其中:

- obj: 需要定义属性的当前对象
- props: 当前准备定义的属性名
- desc: 对定义属性的描述

```
var Person = {}

var name = null

    Object.defineProperty(Person, 'name', {

        get: function () {
            return name
        },
```

```
        set: function (newV) {
            name = newV
        }
    })

    let p = document.getElementById('ppp')
    let ipt = document.getElementById('ipt')

    ipt.addEventListener('input', function (e) {

        Person.name = e.target.value
            p.innerText = Person.name
    })
```

JSON.stringify()

转换过程中会忽略值为undefined的字段

需要检测一下，若某个字段为undefined，则将其值修改为空字符串

1、若目标对象存在toJSON()方法，它负责定义哪些数据被序列化

```
let obj = {

    x: 1,
    y: 2,

    toJSON: function () {
        return 'a string create by toJSON'
    }
}

console.log(JSON.stringify(obj));
//'a string create by toJSON'
```

2、Boolean、Number、String对象被装换为对应原始值

```
const obj = {

    a: new Number(11),
    b: new String('aaa'),
    c: new Boolean(true)
}

console.log(JSON.stringify(obj));
//{"a":11,"b":"aaa","c":true}
```

3、undefined、Function和Symbol不是有效的JSON值，要么被忽略(在对象中找着)，要么被更改为null(在数组中找着)

```
const obj = {

    name: Symbol('aaa'),
    age: undefined,
    isHigh: function () {}
}

console.log(JSON.stringify(obj));
//{}

const arr = [Symbol('aaa'), undefined, function () {}, 'fighting'];

console.log(JSON.stringify(arr));
//[null,null,null,"fighting"]
```

4、所有Symbol-keyed属性被忽略

```
const obj = {}

obj[Symbol('a')] = 'aa';

obj[Symbol('b')] = 'bb';

console.log(obj);

console.log(JSON.stringify(obj));

//{Symbol(a): 'aa', Symbol(b): 'bb'}
//{}
```

5、Date的实例返回一个字符串实现toJSON()方法(和date.toISOString()——使用 ISO 标准返回 Date 对象的字符串格式相同)

```
JSON.stringify(new Date());

//'"2022-06-16T23:36:38.943Z"'
```

6、Infinity、NaN和null都被认为是null

```
const obj = {

    a: Infinity,
    b: NaN,
    c: null,
    val: 20
};

console.log(JSON.stringify(obj));
//{"a":null,"b":null,"c":null,"val":20}
```

7、所有其他Object实例(包括Map、Weakmap、Set和WeakSet)序列化为其可枚举的属性

```
let enumObj = {};

//直接在一个对象上定义新的属性或修改现有属性，并返回该对象
Object.defineProperties(enumObj, {

    'name': {
        value: 'a',
        enumerable: true
    },

    'age': {
        value: 99,
        enumerable: false
    },
});

console.log(JSON.stringify(enumObj));
//{"name":"a"}
```

8、遇到循环抛出TypeError(循环对象值)异常

```
const obj = {

    a: 'aa'
};

obj.subObj = obj;

console.log(JSON.stringify(obj));
//VM357:5 Uncaught TypeError: Converting circular structure to JSON
```

9、对BigInt值字符串化时抛出TypeError(BigInt值无法在JSON中序列化)

```
const obj = {
    a: BigInt(999999999999999999999)
};

console.log(JSON.stringify(obj));
//VM362:4 Uncaught TypeError: Do not know how to serialize a BigInt
```

e.target和e.currentTarget

- e.target：触发事件的元素，是点击的元素
- e.currentTarget：绑定事件的元素，是途径的元素，执行捕获的顺序

addEventListener和onClick()

addEventListener是为元素绑定事件的方法，接收三个参数：

- 第一个参数：绑定的事件名
- 第二个参数：执行的函数
- 第三个参数：
  - false：默认，代表冒泡时绑定
  - true：代表捕获时绑定

onclick和addEventListener事件区别是：onclick事件会被覆盖，而addEventListener可以先后运行不会被覆盖，addEventListener可以监听多个事件

JavaScript NaN 属性

NaN 属性是代表非数字值的特殊值。该属性用于指示某个值不是数字。可以把 Number 对象设置为该值，来指示其不是数字值。

NaN 属性是一个不可配置（non-configurable），不可写（non-writable）的属性。但在ES3中，这个属性的值是可以被更改的，但是也应该避免覆盖。

编码中很少直接使用到 NaN。通常都是在计算失败时，作为 Math 的某个方法的返回值出现的（例如：Math.sqrt(-1)）或者尝试将一个字符串解析成数字但失败了的时候（例如：parseInt("blabla")）。

typeof NaN; / / "number"

NaN 的特性

- NaN是一个特殊值，它和自身不相等，是唯一一个非自反（自反，reflexive，即x =\== x不成立）的值。而NaN !== NaN 为 true。

使用 isNaN() 函数来判断一个值是否是 NaN 值。

isNaN&number.isNaN

函数 isNaN 尝试将参数转换为数值，任何不能被转换为数值的的值都会返回 true，因此非数字值传入也会返回 true ，会影响 NaN 的判断

函数 Number.isNaN 首先判断传入参数是否为数字，如果是数字继续判断是否为 NaN ，这种方法对于 NaN 的判断更为准确

```
function typeOfNaN(x) {

    if (Number.isNaN(x)) {
        return 'Number NaN';
    }

    if (isNaN(x)) {
        return 'NaN';
    }
}

console.log(typeOfNaN('100F'));
// expected output: "NaN"

console.log(typeOfNaN(NaN));
// expected output: "Number NaN"
```

isFinite&isNaN

- Infinity和-Infinity是特殊的数值
- NaN代表一个error

它俩都是number类型，但不是"普通"数字

isNaN(val)，将val转换为数字，再测试是否为NaN

```
isNaN(NaN);//true

isNaN('merry');//true
```

> 值NaN独一无二，不等于任何东西，包括自身
>
> NaN===NaN;//false

isFinite(val)将val转换为数字，若是常规数字而不是NaN/Infinity/-Infinity，则返回true

```
isFinite(11);//true

isFinite("merry");//false 因为是NaN

isFinite(Infinity);//false 因为是Infinity
```

> 在所有数字函数中，包括isFinite，空字符串或只有空格的字符串都被视为0

Ajax、Fetch和Axios

Ajax

Ajax（Asynchronous JavaScript and XML，异步JavaScript与XML技术），使网页实现异步更新，不重新加载网页的情况下，对网页部分进行更新

不是一种新技术，而是2005年被提出的新术语

由XMLHttpRequest的API实现

请求步骤

```
//创建 XMLHttpRequest 对象
const ajax = new XMLHttpRequest();

//规定请求的类型、URL 以及是否异步处理请求,通过 XMLHttpRequest 对象的 open() 方法与服务端建立连接
ajax.open('GET', url, true);

//发送信息至服务器时内容编码类型
ajax.setRequestHeader("Content-type", "application/x-www-form-urlencoded");

//发送请求
ajax.send(null);

//监听服务端的通信状态，接受服务器响应数据
ajax.onreadystatechange = function () {
    if (ajax.readyState == 4 && (ajax.status == 200 || ajax.status == 304)) {}
};
```

特点：

- 局部刷新页面，无需重载整个页面
- 基于原生XHR开发，而XHR本身架构不清晰
- 对基于异步的事件不友好

Fetch

fetch 是底层API，代替XHR，是真实存在的，基于 promise 实现

不是Ajax的封装，而是原生JS，没有使用XMLHttpRequest对象

特点：

- 使用 promise，支持async/await
- 提供的API丰富
- 脱离XHR
- 不携带cookie，需要手动添加配置项

Axios

Axios 是一个基于 promise 封装的HTTP客户端，可以用在浏览器和 node.js 中

它本质也是对原生XMLHttpRequest的封装，只不过它是Promise的实现版本，符合最新的ES规范

特点：

- 从浏览器中创建 XMLHttpRequests
- 从 node.js 创建 http 请求
- 支持 Promise API
- 拦截请求和响应
- 转换请求数据和响应数据
- 取消请求
- 自动转换 JSON 数据
- 客户端支持防御 XSRF

```
axios({

    url: 'xxx', // 设置请求的地址

    method: "GET", // 设置请求方法

    params: { // get请求使用params进行参数凭借,如果是post请求用data
        type: '',
        page: 1
    }
}).then(res => {

    // res为后端返回的数据
    console.log(res);

})
```

设置接口请求前缀

利用node环境变量判断，区分开发、测试、生产环境

```
if (process.env.NODE_ENV === 'development') {

    axios.defaults.baseURL = 'http://dev.xxx.com'
} else if (process.env.NODE_ENV === 'production') {

    axios.defaults.baseURL = 'http://prod.xxx.com'
}
```

本地调试时，在config.js中配置proxy实现代理转发

设置请求头和超时时间

```
const service = axios.create({
        ...

        timeout: 30000, // 请求 30s 超时

        headers: {

            get: {
                'Content-Type': 'application/x-www-form-urlencoded;charset=utf-8'
                // 在开发中，一般还需要单点登录或者其他功能的通用请求头，可以一并配置进来
            },

            post: {
                'Content-Type': 'application/json;charset=utf-8'
                // 在开发中，一般还需要单点登录或者其他功能的通用请求头，可以一并配置进来
            }
        },
    })
```

封装请求方法

```
// get 请求
export function httpGet({

    url,
    params = {}

}) {
    return new Promise((resolve, reject) => {

        axios.get(url, {

            params
        }).then((res) => {

            resolve(res.data)
        }).catch(err => {

            reject(err)
        })
    })
}
```

请求拦截器

在每个请求里加上token，统一处理维护方便

响应拦截器

在接收到响应后先做一层判断，比如状态码判断登录状态、授权

escape、encodeURI&encodeURIComponent

escapeURI(已废弃)生成新的由16进制转义序列替换的字符串。字符的 16 进制格式值，当该值小于等于 0xFF 时，用一个 2 位转义序列: %xx 表示。大于的话则使用 4 位序列：%**u**xxxx 表示。

encodeURIComponent 用一个，两个，三个或四个表示字符的 UTF-8 编码的转义序列来编码 URI

encodeURI 对URI进行编码，将特定字符的每个实例替换为一个、两个、三或四个转义序列

encodeURIComponent和encodeURI异同点

```
var set1 = ";,/?:@&=+$"; // 保留字符

var set2 = "-_.!~*'()"; // 不转义字符

var set3 = "#";   // 数字标志

var set4 = "ABC abc 123"; // 字母数字字符和空格

console.log(encodeURI(set1)); // ;,/?:@&=+$

console.log(encodeURI(set2)); // -_.!~*'()

console.log(encodeURI(set3)); // #

console.log(encodeURI(set4)); // ABC%20abc%20123 (空格被编码为 %20)

console.log(encodeURIComponent(set1)); // %3B%2C%2F%3F%3A%40%26%3D%2B%24

console.log(encodeURIComponent(set2)); // -_.!~*'()

console.log(encodeURIComponent(set3)); // %23

console.log(encodeURIComponent(set4)); // ABC%20abc%20123
```

为避免服务器收到不可预知的请求，对用户输入的任何作为**URI**的部分都应该使用**encodeURIComponent**转义

移动端点击事件延迟

移动端点击有 300ms 的延迟，因为移动端有双击缩放操作，浏览器在 click 之后要等待 300ms（JS捕获click事件的回调处理），看用户有没有下一次点击，判断这次操作是不是双击

有三种办法解决这个问题：

1. meta 标签禁用网页的缩放

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

2. 更改默认视口宽度

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

如果能识别网站是响应式的网站，那么移动端浏览器就可以自动禁掉默认的双击缩放行为并去掉300ms的点击延迟

3. 调用 js 库，比如 FastClick

click 延时问题可能引起点击穿透，如在一个元素上注册了 touchStart 的监听事件，这个事件会将这个元素隐藏掉，发现当这个元素隐藏后，触发了这个元素下的一个元素的点击事件

移动端滚动穿透和溢出

滚动穿透

若页面超过一屏高度出现滚动条时，fixed定位的弹窗遮罩层上下滑动，下面的内容也会一起滑动——滚动穿透

1、默认情况，平移（滚动）和缩放手势由浏览器专门处理，但可通过 CSS 特性 touch-action 改变触摸手势的行为

2、

Step 1、监听弹窗最外层元素（popup）的 touchmove 事件并阻止默认行为来禁用所有滚动（包括弹窗内部的滚动元素）

Step 2、释放弹窗内的滚动元素，允许其滚动：同样监听 touchmove 事件，但是阻止该滚动元素的冒泡行为（stopPropagation），使得在滚动的时候最外层元素（popup）无法接收到 touchmove 事件

滚动溢出

弹窗内也含有滚动元素，在滚动元素滚到底部或顶部时，再往下或往上滚动，也会触发页面的滚动，这种现象称之为滚动链（scroll chaining），也称为滚动溢出（overscroll）

借用 event.preventDefault 的能力，当组件滚动到底部或顶部时，通过调用 event.preventDefault 阻止所有滚动，从而页面滚动也不会触发了，而在滚动之间则不做处理

JS如何影响DOM树构建

渲染引擎判断是一段脚本时，HTML解析器会暂停DOM的解析，JS引擎介入，因为JS脚本可能修改当前已生成的DOM

如果JS脚本通过文件加载，需要先下载JS代码，而JS文件的下载会阻塞DOM解析，下载非常耗时，受网络、文件大小 等因素的影响

如果脚本是内嵌，则 直接执行

如果JS脚本修改了 DOM中div内容，执行这段脚本后，div内容被修改，HTML解析器恢复解析过程

如果JS代码出现了修改CSS的语句，操纵 CSSOM，执行JS前，先解析JS语句之上所有CSS样式，如果引用了外部CSS文件，执行JS之前，需要等待外部CSS文件下载完成，解析生成CSSOM后，才能执行 JS 脚本

JS引擎解析JS代码前，不知道JS是否操纵了CSSOM，所以渲染引擎遇到JS脚本时，不管该脚本是否操纵了CSSOM，都会执行CSS文件下载，解析，再执行JS脚本，构建DOM，生成布局树

所以JS脚本依赖样式表

> JS文件下载会阻塞DOM解析
>
> 样式文件会阻塞JS执行

不会阻塞 DOMContentLoaded 的脚本

1. 具有 async 特性的脚本不会阻塞 DOMContentLoaded
2. 使用 document.createElement('script') 动态生成并添加到网页的脚本也不会阻塞 DOMContentLoaded。

onload

当整个页面，包括样式、图片和其他资源被加载完成时，会触发 window 对象上的 load 事件。可以通过 onload 属性获取此事件。

onunload

当访问者离开页面时，window 对象上的 unload 事件就会被触发。我们可以在那里做一些不涉及延迟的操作，例如关闭相关的弹出窗口。

onbeforeunload

如果访问者触发了离开页面的导航（navigation）或试图关闭窗口，beforeunload 处理程序将要求进行更多确认。

如果我们要取消事件，浏览器会询问用户是否确定。

readyState

如果我们在文档加载完成之后设置 DOMContentLoaded 事件处理程序，会发生什么？

> 很自然地，它永远不会运行。

在某些情况下，我们不确定文档是否已经准备就绪。我们希望我们的函数在 DOM 加载完成时执行，无论现在还是以后。

document.readyState 属性可以为我们提供当前加载状态的信息。

有 3 个可能值：

- loading —— 文档正在被加载。
- interactive —— 文档被全部读取。
- complete —— 文档被全部读取，并且所有资源（例如图片等）都已加载完成。

所以，我们可以检查 document.readyState 并设置一个处理程序，或在代码准备就绪时立即执行它。

proxy

ES6新增功能，可以用来自定义对象中的操作

代理是一种很有用的抽象机制，能够通过API只公开部分信息，同时还能对数据源进行全面控制

在需要公开API，同时又要避免使用者直接操作底层数据时，可使用代理

比如，实现一个传统的栈数据类型，数组可以作为栈使用，但要保证人们只使用push pop 和length，我们可以基于数组创建一个代理对象，只对外公开这个三个对象成员

Vue3.0中使用Proxy替换原本的Object.defineProperty实现数据响应式

> let p = new Proxy(target, handler)

target 表示需要添加代理的对象，handler表示自定义对象中的操作，可以用来自定义set或get函数

使用Proxy实现数据响应式：

```
let onWatch = (obj, setBind, getLogger) => {
    let handler = {
        get(target, property, receiver) {
            getLogger(target, property)
            return Reflect.get(target, property, receiver)
        },
        set(target, property, value, receiver) {
            setBind(value, property)
            return Reflect.set(target, property, value)
        }
    }
    return new Proxy(obj, handler)
}
let obj = {
    a: 1
}
let p = onWatch(
        obj,
        (v, property) => {
        console.log(`监听到属性${property}改变为${v}`)
    },
        (target, property) => {
        console.log(`'${property}' = ${target[property]}`)
    })
    p.a = 2 // 监听到属性a改变
    p.a // 'a' = 2
```

自定义set和get函数，在原本逻辑中插入函数逻辑，实现 在对 对象任何属性进行读写时 发出通知

如果 是实现Vue中的响应式，需要在get中收集依赖，在set派发更新，**Vue3.0使用Proxy代替原来API的原因在于Proxy无需一层层递归为每个属性添加代理，一次即可完成以上操作，性能更好，Proxy可以完美监听到任何方式的数据改变，缺陷是 浏览器兼容性 不太好**

CDN

Content Delivery Network，内容分发网络

我们访问一个页面的时候，会请求很多资源，包括各种图片、声音、影片、文字等信息。这和我们要购买多种货物一样

网站可以预先把内容分发至全国各地的加速节点。用户可以就近获取内容，避免网络拥堵、地域、运营商等因素带来的访问延迟问题

"内容分发网络"像前面提到的"全国仓配网络"，解决因分布、带宽、服务器性能带来的访问延迟问题，适用于站点加速、点播、直播等场景

用户可就近取得所需内容，解决 Internet网络拥挤的状况

CDN本质 是一大堆遍布在全球各个 角落 的缓存服务器。通过与DNS的配合，找到最靠近用户的一台CDN缓存服务器，将数据快速 分发 给用户

减少对整体骨干网的流量负担，提高用户体验

DNS解析之后，浏览器向服务器请求内容后发生

长途骨干网的传输最耗时，需经过网站服务器所在的机房、骨干网、用户所在城局域网、用户所在接入网等，物理传输距离遥远

1亿人同时请求12306上一张一模一样的图片，对国家的互联网基础设施是一个灾难

CDN 提前把数据存在离用户最近的数据节点，避免长途跋涉经过长途骨干网，最终 减少骨干网负担、提高访问速度

请求图片数据，先去CDN缓存服务器获取，若获取到数据直接返回，否则才 经过 长途骨干网，最终达到 网站服务器 获取数据

CDN其实还缩短了请求数据的距离

用户分布全国各地，一般会在 离用户在 较近的地方设置 CDN 缓存服务器，酱紫各个 地区的用户能直接请求对应的CDN服务器，不需要来回跑 大半个 中国！

过程

1. 发起请求，本地 DNS 解析，将域名解析权交给域名 CNAME 指向的 CDN 专用 DNS 服务器
2. CDN 的 DNS 服务器将 CDN 的全局负载均衡设备 IP 地址返回浏览器
3. 浏览器向 CDN 全局负载均衡设备发起 URL 请求
4. CDN 全局负载均衡设备根据用户 IP ，以及URL，选择一台用户所属区域的区域负载均衡设备，向其发请求
5. 区域负载均衡设备为用户选最合适的 CDN 缓存服务器（考虑的依据包括：服务器负载情况，距离用户的距离等），返回给全局负载均衡设备
6. 全局负载均衡设备将选中的 CDN 缓存服务器 IP 返回给用户
7. 根据用户IP，判断最近边缘节点
8. 根据用户请求URL中内容，判断有用户所需内容的边缘节点
9. 查询边缘节点负载情况，判断有服务能力的边缘节点
0. 全局负载均衡设备将服务器IP返回给用户
1. 用户向 CDN 缓存服务器发起请求，缓存服务器响应用户请求，最终将内容返回

组成

（CDN）由多个节点组成。一般，CDN网络主要由中心节点、边缘节点两部分构成

中心节点

中心节点包括CDN网管中心和全局负载均衡DNS重定向解析系统，负责整个CDN网络的分发及管理

边缘节点

CDN边缘节点主要指异地分发节点，有负载均衡设备、高速缓存服务器两部分

负载均衡设备负责每个节点中各个Cache的负载均衡，保证节点 工作效率；同时负责收集节点与周围环境的信息，保持与全局负载均衡DNS的通信，实现整个系统的负载均衡

高速缓存服务器（Cache）负责存储客户网站信息，像一个靠近用户的网站服务器一样响应本地用户的请求

通过全局负载均衡DNS的控制，用户的请求被透明 指向离他最近的节点，节点中Cache服务器像网站的原始服务器一样，响应终端用户的请求

中心节点像仓配网络中负责货物调配的总仓，边缘节点就是负责存储货物的各个城市的本地仓库

Web Worker

JS采用单线程模型，前面任务没做完 后面的任务只能等着

webworker为JS创造多线程环境，允许主线程创建worker线程，将一些任务分配给后者运行，主线程运行的同时，worker线程在后台，互不干扰，worker完成计算任务 将结果返回 主线程

好处：计算密集型/高延迟的任务 由worker负担，主线程流畅

worker线程一旦创建 ，始终运行，不会被主线程上的活动打断，有利响应主线程的通信，缺点：worker消耗资源，不该过度使用

注意

1. 同源限制
2. DOM限制
3. 无法读取主线程所在网页的DOM对象，但可读navigator/location对象
4. worker线程和主线程不在同一个上下文环境，不能直接通信
5. worker不能执行alert()/confirm()，但可以发出Ajax请求

HTTP 2.0

特性

HTTP/2 协议基于 HTTPS

HTTP2.0可以说是SPDY的升级版本，与SPDY的区别如下：

1. HTTP2.0支持明文HTTP传输，而SPDY强制使用HTTPS。
2. HTTP2.0消息头压缩算法使用HPACK，而SPDY使用DEFLATE。

HTTP2.0主要目标是改进传输性能，实现低延迟和高吞吐量。

HTTP2.0升级改造需要考虑：

1. HTTP2.0可以支持非HTTPS，但是主流浏览器如chrome、Firefox还是只支持基于TLS部署的HTTP2.0协议，所以要升级HTTP2.0还是先升级HTTPS
2. 升级HTTPS后，如果使用NGINX，只需要在配置文件中启动相应的协议就可以
3. HTTP2.0完全兼容HTTP1.x，对于不支持HTTP2.0的浏览器，NGINX会自动向下兼容

二进制分帧

HTTP/2 不再像 HTTP/1.1 里的纯文本形式的报文，而是全面采用二进制格式，**头信息和数据体都是二进制，统称为帧（frame）**：头信息帧（Headers Frame）和数据帧（Data Frame）。

对人不友好，但是对计算机非常友好，因为计算机只懂二进制，收到报文后，无需再将明文的报文转成二进制，而是直接解析二进制报文，**这增加了数据传输的效率**

(因为是新增的二进制分帧层，所以叫2.0)

不改变HTTP语义，HTTP方法、状态码、URL及首部字段。**改进传输性能，实现低延迟和高吞吐量**

HTTP1.x解析基于文本，文本展现形式多样，要做到健壮性考虑的场景必然很多，二进制则只有0和1，**更高效健壮**

首部压缩

HTTP/2 会压缩头（Header）如果同时发出多个请求，他们的头是一样的或是相似的，那么，协议会**消除重复的部分**

HPACK算法：在客户端和服务器同时维护一张头信息表，所有字段都会存入这个表，生成一个索引号，以后就不发送同样字段了，只发送索引号，这样就**提高速度**

HTTP 1.x无首部压缩，Gzip只对请求体压缩

SPDY和HTTP 2.0都支持首部压缩。使头部帧最大程度复用，减少头部大小，利于减少内存和流量

比如：第一次请求，包含头部各种信息，后来又发送另外请求，发现大部分字段可以复用，一次只发送一个当前请求特有的头部帧即可

首部表在HTTP 2.0的连接存续期始终有效，server和client共同更新

避免重复header传输，又减少需要传输的大小

1. HTTP2.0会压缩首部元数据，在client和server使用首部表跟踪和存储之前发送的健值对，对于相同数据，不需每次请求响应都发送
2. 所有header必须全部小写，而且请求行要独立为健值对（即header+值）

多路复用

HTTP 1.0：建立连接请求数据完毕之后立即关闭连接；

后来采用keep-alive模式使得可以复用连接而不断开，可利用这次连接继续请求数据

缺点：必须等到server返回上一次的请求数据才可以进行下一次请求

Q：遇到一个请求很久没有响应，后面的请求只能等待?

HTTP/2 是可以在**一个连接中并发多个请求或回应，而不用按照顺序一一对应。**

多路复用（MultiPlexing），即连接共享，每一个request都是用作连接共享机制的。每一个request对应一个id，一个连接上可以有多个request，每个连接的request可以随机的混杂在一起，接收方根据request的id将request再归属到不同服务端请求里面。客户端只需要一个连接就可以加载一个页面

移除了 HTTP/1.1 中的串行请求，不需要排队等待，也就不会再出现「队头阻塞」问题，**降低了延迟，大幅度提高了连接的利用率**

举例来说，在一个 TCP 连接里，服务器收到了客户端 A 和 B 的两个请求，如果发现 A 处理过程非常耗时，于是就回应 A 请求已经处理好的部分，接着回应 B 请求，完成后，再回应 A 请求剩下的部分

优点：

1. 并行交错发送请求，请求之间互不影响
2. TCP连接一旦建立可并行发送请求
3. 消除不必要延迟，减少页面加载时间
4. 可最大程度利用HTTP 1.x

请求优先级

server根据流的优先级控制资源分配，响应数据准备好后，把优先级最高的帧发送给client。browser发现资源时立即分派请求，**指定每个流的优先级，让服务器决定最优的响应次序，这时请求不用排队，节省时间，最大限度利用连接**

流量控制

流：不改变协议，允许采用多种流量控制算法

特点：

1. 流量基于HTTP连接的每一跳进行，不是端到端控制
2. 流量基于窗口更新帧进行，接收方可广播准备接收字节甚至对整个连接要接收的字节数
3. 流量控制有方向，接收方根据自身情况控制窗口大小
4. 流量控制可由接收方禁用
5. 只有data帧服从流量控制，其他不会消耗控制窗口的空间

服务端推送

HTTP/2 在一定程度上改善了传统的「请求 - 应答」工作模式，服务不再被动响应，**也可以主动向客户端发送消息**

server push通过推送那些它认为客户端将会需要使用到的内容到客户端缓存中，以此避免往返的延迟

1. 客户端可以限定推送流的数量，也可以设置为0而完全禁用server push
2. 所有推送遵守同源策略，即服务器不能随便将第三方资源推送给客户端，必须是经过双方确认的
3. PUSH_PROMISE帧：所有服务器推送流都通过PUSH_PROMISE发送，服务端发出有意push所述资源的信号，客户端接收到PUSH_PROMISE帧后，也可以拒绝这个流
4. 服务端必须遵循请求-响应的循环，只能借着对请求的响应推送资源。
5. PUSH_PROMISE帧必须在返回响应之前发送，否则客户端会出现竞态条件。

数据流

HTTP/2 的数据包不是按顺序发送，同一个连接里面连续的数据包，可能属于不同回应。因此，必须要对数据包做标记，指出它属于哪个回应

在 HTTP/2 中每个请求或相应的所有数据包，称为一个数据流（Stream）。每个数据流都标记着一个独一无二的编号（Stream ID），**不同 Stream 的帧可以乱序发送（因此可以并发不同Stream ）**，因为每个帧的头部会携带 Stream ID 信息，所以接收端可以通过 Stream ID 有序组装成 HTTP 消息

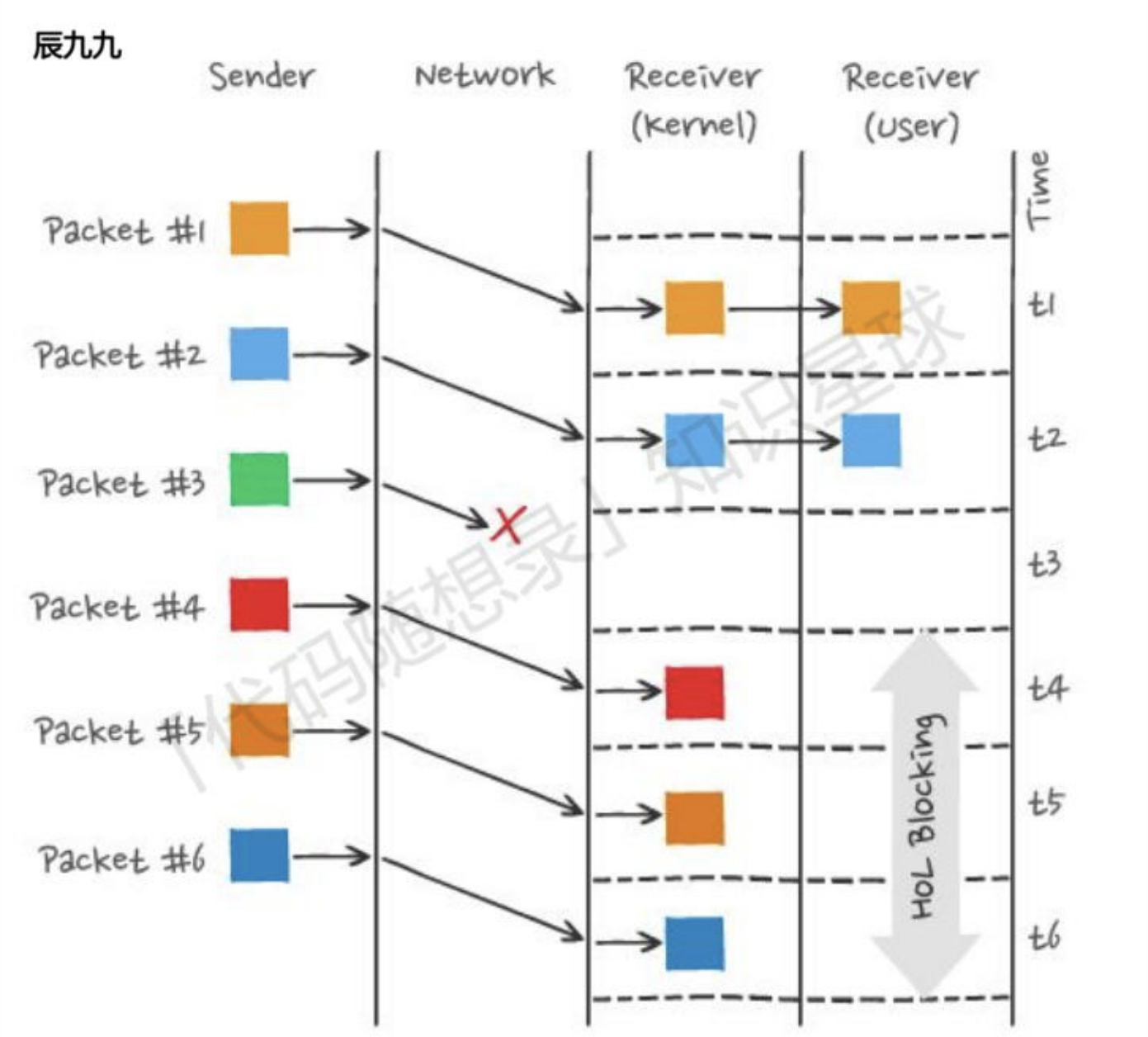
客户端和服务器**双方都可以建立 Stream**，Stream ID 也有区别，客户端建立的 Stream 必须是奇数号，而服务器建立的 Stream 必须是偶数号

客户端还可以**指定数据流的优先级**。优先级高的请求，服务器就先响应该请求。

缺陷

HTTP/2 通过 Stream 的并发能力，解决了 HTTP/1 队头阻塞的问题，看似很完美了，但是 HTTP/2 还是存在"队头阻塞"的问题，只不过问题不是在 HTTP 这一层面，而是在 TCP 这一层。

HTTP/2 是基于 TCP 协议传输数据，TCP 是字节流协议，TCP 层必须保证收到的字节数据完整且连续，这样内核才会将缓冲区里的数据返回给 HTTP 应用，当「前 1 个字节数据」没有到达时，后收到的字节数据只能存放在内核缓冲区里，只有等到这 1 个字节数据到达时，HTTP/2 应用层才能从内核中拿到数据—— HTTP/2 队头阻塞问题。



图中发送方发送了很多个 packet，每个 packet 都有自己的序号，可以认为是 TCP 的序列号，其中 packet 3 在网络中丢失了，即使 packet 4-6 被接收方收到后，由于内核中的 TCP 数据不连续，于是接收方的应用层无法从内核中读取，只有等到 packet 3 重传后，接收方的应用层才可以从内核中读取到数据——HTTP/2 的队头阻塞问题，发生在 TCP 层面

所以，一旦发生了丢包现象，就会触发 TCP 重传机制，这样在一个 TCP 连接中的所有的 **HTTP 请求都必须等待这个丢了的包被重传回来**

微前端

一种类似于 微服务的架构，它将微服务的理念应用于 浏览器端，将 Web应用由单一的应用转变为多个小型前端应用聚合为一的应用

各个前端应用 可以 使用 不同技术栈、 独立运行、独立开发、独立部署

不是框架或工具，而是一套 架构体系

解决问题

1. 将庞大应用拆分，每个部分可以单独部署 、维护，提升效率
2. 整合系统，在基本不修改原来系统逻辑的同时 兼容新老老套系统并行运行

技术方案

思想就是拆解和整合应用，通常是一个父应用加上一些子应用

- Nginx路由转发

实现不同路径映射到不同应用，这不属于是前端层面的改造，更多是运维的配置

简单、快速、易配置，切换应用时触发浏览器刷新会引用体验

- iframe嵌套

子应用嵌套iframe，父子通信可采用POSTMessage或contentWindow实现

实现简单、子应用间自带沙箱隔离 互不影响、iframe样式显示有兼容性、太过简单显得low

- Web Components

子应用采用 纯 Web Components技术编写，是全新的开发模式

可单独部署，系统改造成本高，子应用通信复杂 易 踩坑

- 组合式 应用路由分发

子应用独立构建和部署，父应用管理路由、应用加载、启动、卸载和通信

纯前端改造、体验好、可无感切换，需设计和开发父子应用于同一页面运行，需解决子应用样式冲突、变量对象污染、通信等技术点

应用隔离

分为 主应用 和 微应用

微应用间的JS执行环境隔离、CSS隔离

> CSS隔离

主微应用 同屏渲染时，可能有样式会相互污染，需要彻底隔离CSS污染，可采用CSS module或命名空间的方式，保证不相互干扰，或者webpack的postcss插件，打包时加上特定的前缀

微应用 间的CSS隔离，应用加载时，将应用所有的link和style进行标记，卸载应用时再同步卸载即可

> JS隔离

使用沙箱机制，避免对全局window和全局事件的修改

浏览器可结合with关键字或Proxy实现浏览器端沙箱

保证局部JS运行时，对对外部对象的访问和修改在可控范围内

浏览器

浏览器概述

浏览器是一个多进程的架构，我们关心的是渲染进程(核心进程)。

为何使用多进程架构?

由于多个线程共享着相同的地址空间和资源,所以会存在线程之间有可能会恶意修改或者获取非授权数据等复杂的安全问题。

单进程浏览器:

1、不稳定。单进程中的插件、渲染线程崩溃导致整个浏览器崩溃。

2、不流畅。脚本（死循环）或插件会使浏览器卡顿。

3、不安全。插件和脚本可以获取到操作系统任意资源。

多进程浏览器:

1、解决不稳定。进程相互隔离，一个页面或者插件崩溃时，影响仅仅时当前插件或者页面，不会影响到其他页面。

2、解决不流畅。脚本阻塞当前页面渲染进程，不会影响到其他页面。

3、解决不安全。采用多进程架构使用沙箱。沙箱看成是操作系统给进程上来一把锁，沙箱的程序可以运行，但不能在硬盘上写入任何数据，也不能在敏感位置读取任何数据。

> 将渲染进程和OS隔离的这道墙就是安全沙箱。
> 作用：利用OS提供的安全技术，让渲染进程在执行过程中无法访问或修改OS中的数据，在渲染进程需要访问系统资源的时候，需通过浏览器内核实现，将访问结果通过IPC转发给渲染进程。
> 最小的保护单位是进程。单进程浏览器需要频繁访问或修改OS的数据，所以单进程浏览器无法被安全沙箱保

> 护。

主要进程

浏览器是多进程的. 主要分为:

浏览器主进程

只有一个，控制页面的创建、销毁、网络资源管理、下载等, 同时提供存储等功能 。

第三方插件进程

每一种类型的插件对应一个进程，仅当使用该插件时才创建。 主要是负责插件的运行，因插件易崩溃，所以需要通过插件进程来隔离，以保证插件进程崩溃不会对浏览器和页面造成影响.

GPU进程

最多一个，用于3D绘制等,从浏览器进程中独立出来的。

浏览器渲染进程(浏览器内核)

每个Tab页对应一个进程，互不影响 , 核心任务是将HTML、CSS 和 JavaScript转换为可以与用户交互的网页。 出于安全考虑，渲染进程都是运行在沙箱模式下。

网络进程

从浏览器进程中独立出来的, 主要负责页面的网络资源加载。

GUI 线程

负责渲染页面，解析 html、css；构建 DOM 树和渲染树；当界面需要重绘(Repaint)或由于某种操作引发回流(reflow)时,该线程就会执行。在Javascript引擎运行脚本期间,GUI渲染线程都是处于挂起状态的,也就是说被”冻结”了.

js 引擎线程

负责解析和执行 js 程序，我们经常听到的 chrome 的 v8 引擎就是跑在 js 引擎线程上的，其实语言没有单线程多线程之说，因为解释这个语言的是 的线程是单线程；js 引擎线程与 gui 线程互斥，当浏览器执行 javaScript 程序的时候，GUI 渲染线层会保存在一个队列当中；直到 js 程序执行完成，才会接着执行；如果 js 的执行时间过长，会影响页面的渲染不连贯，所有我们要尽量控制 js 的大小。

JS引擎线程和GUI渲染线程互斥！

定时触发线程

浏览器定时计数器并不是由JS引擎计数的, 因为JS引擎是单线程的, 如果处于阻塞线程状态就会影响记计时的准确, 因此通过单独线程来计时并触发定时是更为合理的方案。

为什么 setTimeout 不阻塞后面程序的运行，因为 setTimeout 不是由 js 引擎线程完成的，是由定时器触发线程完成的，所以它们可以同时进行，那么定时器触发线程在这定时任务完成之后会通知事件触发线程往任务队列里添加事件。

事件触发线程

当一个事件被触发时该线程会把事件添加到待处理队列的队尾，等待JS引擎的处理。这些事件可以是当前执行的代码块如定时任务、也可来自浏览器内核的其他线程如鼠标点击、AJAX异步请求等，但由于JS的单线程关系所有这些事件都得排队等待JS引擎处理。

异步 HTTP 请求线程

在XMLHttpRequest在连接后是通过浏览器新开一个线程请求， 将检测到状态变更时，如果设置有回调函数，异步线程就产生状态变更事件放到 JavaScript引擎的处理队列中等待处理。

异步场景

1. 定时器
2. 网络请求
3. 事件绑定
4. ES6 Promise

定时器

执行调用栈的代码，栈中为空时去检查任务队列，这是一个循环检查的过程，等到事件触发线程往任务队列中添加了定时器事件，这时再去检查已经有了定时器的异步任务，取出放进执行栈执行。

带来问题

定时任务可能不会按时执行。

应用场景

1. 防抖节流
2. 倒计时
3. 动画(丢帧问题)

内核

浏览器内核是通过取得页面内容、整理信息、计算和组合最终输出可视化图像结果（渲染引擎）

每一个tab页面可以看作是浏览器内核进程，**浏览器内核是多线程的**。

常见浏览器内核

Trident内核：IE，360，搜狗等浏览器

Gecko内核：Netscape6及以上版本，Firefox

Blink内核：Opera7及以上

Webkit内核：Safari，Chrome

检测版本

1. 检测 window.navigator.userAgent 的值，但这种方式很不可靠，因为 userAgent可以被改写，并且早期的浏览器如 ie，会通过伪装自己的 userAgent的值为 Mozilla来躲过服务器的检测。
2. 功能检测，根据每个浏览器独有的特性来进行判断，如 ie 下独有的 ActiveXObject。

输入URL回车后

简易版

1. URL解析
2. 查找缓存
3. 域名解析：浏览器缓存>系统缓存>本地hosts>根域名>顶级域名>二级域名>三级域名
4. TCP三次握手
5. 发送HTTP请求
6. 服务器处理请求并返回报文
7. 浏览器解析渲染页面
8. TCP四次挥手 关闭TCP连接

从宏观是上理解从输入URL到页面渲染的过程,**主要分为导航阶段和渲染阶段.**

A 导航阶段

一、浏览器主进程

1. 用户输入URL

1、浏览器进程检查url，组装协议，构成完整的url，这时候有两种情况：

- 输入的是搜索内容：地址栏会使用浏览器默认的搜索引擎，来合成新的带搜索关键字的URL。
- 输入的是请求URL：地址栏会根据规则，给这段内容加上协议，合成为完整的URL；

2、浏览器进程通过进程间通信（IPC）把url请求发送给网络进程；

URL一般包括几大部分：

- protocol，协议头，譬如有http，ftp等
- host，主机域名或IP地址
- port，端口号
- path，目录路径
- query，即查询参数
- fragment，即 #后的hash值，一般用来定位到某个位置

二、网络进程

2. URL请求过程

3、网络进程接收到url请求后检查本地是否缓存了该请求资源。

- 浏览器发送请求前，根据请求头的expires和cache-control判断是否命中（包括是否过期）强缓存策略，如果命中，直接从缓存获取资源，并不会发送请求。如果没有命中，则进入下一步。
- 没有命中强缓存规则，浏览器会发送请求，根据请求头的If-Modified-Since(last_modified)和If-None-Match(ETag)判断是否命中协商缓存，如果命中，直接从缓存获取资源。如果没有命中，则进入下一步。
- 如果前两步都没有命中，则直接从服务端获取资源。

4、准备IP地址和端口：进行DNS解析时先查找缓存，没有再使用DNS服务器解析，查找顺序为：

- 浏览器缓存；

- 本机缓存；
- hosts文件；
- 路由器缓存；
- ISP DNS缓存；
- DNS递归查询（本地DNS服务器 -> 权限DNS服务器 -> 顶级DNS服务器 -> 13台根DNS服务器）

5、等待TCP队列：浏览器会为每个域名最多维护6个TCP连接，如果发起一个HTTP请求时，这 6个 TCP连接都处于忙碌状态，那么这个请求就会处于排队状态，解决方案：

- 采用域名分片技术：将一个站点的资源放在多个（CDN）域名下面。
- 升级为HTTP2，就没有6个TCP连接的限制了；

6、通过三次握手建立TCP连接：

- 第一次：**客户端先向服务器端发送一个同步数据包，报文的TCP首部中：标志位**：同步SYN为1，表示这是一个请求建立连接的数据包；序号Seq=x，x为所传送数据的第一个字节的序号，随后进入SYN-SENT状态；

  > 标志位值为1表示该标志位有效。

- 第二次：服务器根据收到数据包的SYN标志位判断为建立连接的请求，随后返回一个确认数据包，其中标志位SYN=1，ACK=1，序号seq=y，确认号ack=x + 1表示收到了客户端传输过来的x字节数据，并希望下次从x+1个字节开始传，并进入SYN-RCVD状态；

  > 这里要区分标志位ACK和确认号ack；

- 第三次：客户端收到后，再给服务器发送一个确认数据包，**标志位ACK=1，序号seq=x+1，确认号ack=y+1，随后进入ESTABLISHED状态；**

服务器端收到后，也进入ESTABLISHED状态，由此成功建立了TCP连接，可以开始数据传送；

- **为什么要第三次挥手？**避免服务器等待造成**资源浪费**，具体原因：

  > 如果没有最后一个数据包确认（第三次握手），A先发出一个建立连接的请求数据包，由于网络原因绕远路了。A经过设定的超时时间后还未收到B的确认数据包。
  > 于是发出第二个建立连接的请求数据包，这次网路通畅，B的确认数据包也很快就到达A。于是A与B开始传输数据；
  > 过了一会A第一次发出的建立连接的请求数据包到达了B，B以为是再次建立连接，所以又发出一个确认数据包。由于A已经收到了一个确认数据包，所以会忽略B发来的第二个确认数据包，但是B发出确认数据包之后就要一直等待A的回复，而A永远也不会回复。
  > 由此造成服务器资源浪费，这种情况多了B计算机可能就停止响应了。

7、构建并发送HTTP请求信息；

- 建立TCP连接后，在这基础上进行通信，浏览器发送http请求到目标服务器，请求的内容包括 **请求行 请求头和请求体**
- 当服务器接收到浏览器的请求之后，就会进行逻辑操作，处理完之后返回一个HTTP响应消息，**包括，响应行，响应头和响应体。**
- 服务器响应之后，现在HTTP默认开启长连接，**页面关闭后，TCP连接会经过四次挥手断开**

8、服务器端处理请求；

9、客户端处理响应，首先检查服务器响应报文的状态码：

- 如果是301/302表示服务器已更换域名需要重定向，这时网络进程会从响应头的**Location**字段里面读取重定向的

地址，然后再发起新的HTTP或者HTTPS请求，跳回第4步。
- 如果是200，就检查Content-Type字段，值为text/html说明是HTML文档，是application/octet-stream说明是文件下载；

10、请求结束，当通用首部字段Conection不是Keep-Alive时，即不为TCP长连接时，通过四次挥手断开TCP连接：

四次挥手步骤（抽象派）

- 主动方：我已经关闭了向你那边的主动通道了，只能被动接收了
- 被动方：收到通道关闭的信息
- 被动方：那我也告诉你，我这边向你的主动通道也关闭了
- 主动方：最后收到数据，之后双方无法通信
- 第一次：客户端（主动断开连接）发送数据包给服务器，其中标志位FIN=1，序号位seq=u，并停止发送数据；
- 第二次：服务器收到数据包后，由于还需传输数据，无法立即关闭连接，先返回一个标志位ACK=1，序号seq=v，确认号ack=u+1的数据包；
- 第三次：服务器准备好断开连接后，返回一个数据包，其中标志位FIN=1，标志位ACK=1，序号seq=w，确认号ack=u+1；
- 第四次：客户端收到数据包后，返回一个标志位ACK=1，序号seq=u+1，确认号ack=w+1的数据包。

由此通过四次挥手断开TCP连接。

为什么要四次挥手?

由于服务器不能马上断开连接，导致FIN释放连接报文与ACK确认接收报文需要分两次传输，即第二次和第三次"挥手"；

3. 准备渲染进程

11、准备渲染进程：浏览器进程检查当前url是否与之前打开了渲染进程的页面的根域名相同，如果相同，则复用原来的进程，如果不同，则开启新的渲染进程；

4. 提交文档

12、提交文档：

- 渲染进程准备好后，浏览器向渲染进程发起“提交文档”的消息，渲染进程接收到消息后与网络进程建立传输数据的“管道”
- 渲染进程接收完数据后，向浏览器发送“确认提交”
- 浏览器进程接收到确认消息后更新浏览器界面状态：安全状态、地址栏url、前进后退的历史状态、更新web页面

B 渲染阶段

渲染步骤大致可以分为以下几步：

1. 解析HTML，构建DOM树
2. 解析CSS，生成CSS规则树
3. 合并DOM树和CSS规则，生成render树
4. 布局render树（Layout/reflow），负责各元素尺寸、位置的计算
5. 绘制render树（paint），绘制页面像素信息

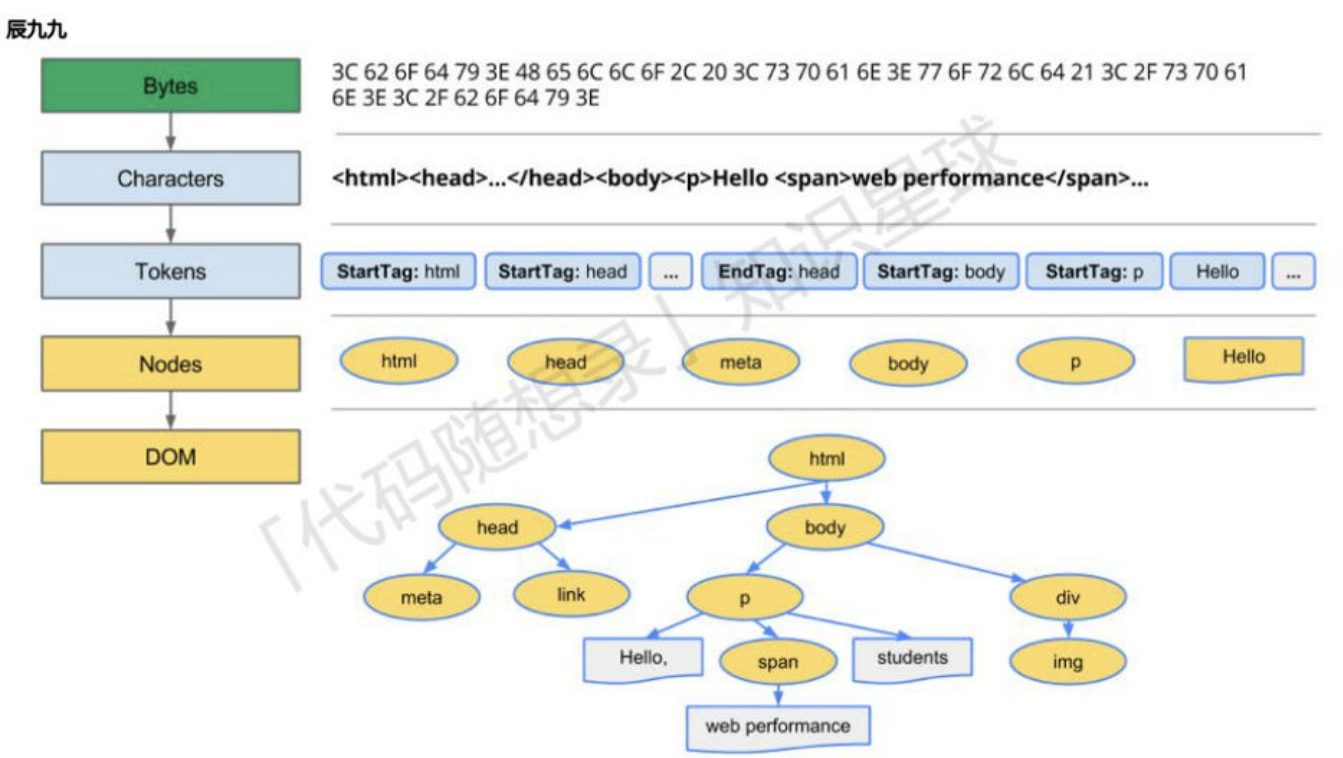
6. 浏览器会将各层的信息发送给GPU，GPU会将各层合成（composite），显示在屏幕上

在渲染阶段通过**渲染流水线**在渲染进程的主线程和合成线程配合下，完成页面的渲染；

三、渲染进程

5. 构建DOM树

13、先将请求回来的数据解压，随后HTML解析器将其中的HTML字节流通过**分词器**拆分为一个个Token，然后生成节点Node，最后解析成浏览器识别的DOM树结构。[解析HTML,生成DOM树]

1. 

2. 重点过程
3. Conversion转换：浏览器将获得的HTML内容（Bytes）基于他的编码转换为单个字符
4. Tokenizing分词：浏览器按照HTML规范标准将这些字符转换为不同的标记token。每个token都有自己独特的含义以及规则集
5. Lexing词法分析：分词的结果是得到一堆的token，此时把他们转换为对象，这些对象分别定义他们的属性和规则
6. DOM构建：因为HTML标记定义的就是不同标签之间的关系，这个关系就像是一个树形结构一样。例如：body对象的父节点就是HTML对象，然后段落p对象的父节点就是body对象
7. 最后DOM树

可以通过Chrome调试工具的Console选项打开控制台输入document查看DOM树；

> 渲染引擎还有一个**安全检查模块**叫 XSSAuditor，是用来**检测词法安全**的。在分词器解析出来 Token 之后，它会检测这些模块是否安全，比如**是否引用了外部脚本，是否符合 CSP 规范，是否存在跨站点请求**等。如果出现不符合规范的内容，XSSAuditor 会对该脚本或者下载任务**进行拦截**。

首次解析HTML时**渲染进程**会开启一个**预解析线程**，遇到HTML文档中内嵌的JavaScript和CSS外部引用就会同步提前下载这些文件，下载时间以最后下载完的文件为准。

6. 构建CSSOM

14、CSS解析器将CSS转换为浏览器能识别的styleSheets也就是CSSOM：可以通过控制台输入document.styleSheets查看；

这里要考虑一下阻塞的问题，由于JavaScript有修改CSS和HTML的能力，所以，需要先等到 CSS 文件下载完成并生成 CSSOM，然后再执行 JavaScript 脚本，最后再继续构建 DOM。由于这种阻塞，导致了**解析白屏**；[解析CSS,生成CSSOM]

> 优化方案：

1. **移除js和css的文件下载**：通过内联 JavaScript、内联 CSS；
2. **尽量减少文件大小**：如通过 webpack 等工具**移除**不必要的**注释**，并压缩 **js 文件**；
3. 将不进行DOM操作或CSS样式修改的 JavaScript 标记上 **async 或者 defer异步引入；**
4. **使用媒体查询属性**：将大的CSS文件拆分成多个不同用途的 CSS 文件，只有在特定的场景下才会加载特定的 CSS 文件。

可以通过浏览器调试工具的Network面板中的DOMContentLoaded查看最后生成DOM树所需的时间；

7. 样式计算

15、转换样式表中的属性值，使其标准化。**比如将em转换为px，color转换为rgb；**

16、计算DOM树中每个节点的具体样式，这里遵循CSS的继承和层叠规则；可以通过Chrome调试工具的Elements选项的Computed查看某一标签的最终样式；

8. 布局阶段

[结合DOM和CSSOM树,生成渲染树]

17、创建布局树，遍历DOM树中的所有节点，去掉所有隐藏的节点（比如head，添加了display:none的节点），只在布局树中保留可见的节点。

18、计算布局树中节点的坐标位置（较复杂，这里不展开）,对于每个可见的节点,找到CSSOM树中对应规则并应用,根据每个可见节点及其对应的样式,生成渲染树；

9. 分层

19、对布局树进行分层，并生成分层树（Layer Tree），可以通过Chrome调试工具的Layer选项查看。分层树中每一个节点都直接或间接的属于一个图层（如果一个节点没有对应的层，那么这个节点就从属于父节点的图层）

图层

一般来说，可以把普通文档流看成一个图层。特定的属性可以生成一个新的图层。**不同的图层渲染互不影响**，所以对于某些频繁需要渲染的建议单独生成一个新图层，提高性能。**但也不能生成过多的图层，会引起反作用。**

通过以下几个常用属性可以生成新图层

- 3D 变换：translate3d、translateZ
- will-change

- video、iframe 标签
- 通过动画实现的 opacity 动画转换
- position: fixed

10. 图层绘制

20、为每个图层生成绘制列表（即绘制指令），并将其提交到合成线程。以上操作都是在渲染进程中的主线程中进行的，提交到合成线程后就不阻塞主线程了；

11. 切分图块

21、合成线程将图层切分成大小固定的图块（256x256或者512x512）然后优先绘制靠近视口的图块，这样就可以大大加速页面的显示速度；

四、GPU 进程

12. 栅格化操作

22、在光栅化线程池中将图块转换成位图，通常这个过程都会使用GPU来加速生成，使用GPU生成位图的过程叫快速栅格化，或者GPU栅格化，生成的位图被保存在GPU内存中。

五、浏览器主进程

13. 合成与显示

23、合成：一旦所有图块都被光栅化，合成线程就会将它们合成为一张图片，并生成一个绘制图块的命令——“DrawQuad”，然后将该命令提交给浏览器进程。

> **注意了：合成的过程是在渲染进程的合成线程中完成的，不会影响到渲染进程的主线程执行；**

24、显示：浏览器进程里面有一个叫viz的组件，用来接收合成线程发过来的DrawQuad命令，然后根据DrawQuad命令，将其页面内容绘制到内存中，最后再将内存显示在屏幕上。将像素发送给GPU,展示在页面上(GPU将多个合成层合并成一个层,展示)

到这里，经过这一系列的阶段，编写好的HTML、CSS、JavaScript等文件，经过浏览器就会显示出漂亮的页面了。

> Layout回流 [重排]：通过 JavaScript 或者 CSS 修改元素几何位置属性，会触发重新布局，解析后面一系列子阶段

> 重绘：跳过了布局阶段，直接进入绘制，然后再分块、生成位图及其以后子阶段；Painting 根据渲染树及回流得到的几何信息,得到节点的绝对像素.

> 合成：渲染引擎跳过布局和绘制阶段，执行的后续操作，发生在合成线程，非主线程；

前端路由

把不同路由对应不同内容或页面的任务交给前端来做，之前通过服务端根据 url 的不同返回不同的页面实现

前端路由实质上就是检测 URL 的变动，截获 URL 地址，通过解析、匹配路由规定实现 UI 更新

单页面应用中的路由分为hash和history模式

hash模式

监听浏览器地址hash值变化，执行事件

hash会在浏览器URL后增加 # 号

一个完整的的 URL 包含：协定、域名、端口、虚拟目录、文件名、参数、锚

比如 https://www.google.com/#abc中的hash值为abc 特点：hash的变化不会刷新页面，也不会发送给服务器

但hash的变化会被浏览器记录下来，来指导浏览器中的前进和后退

window.location.hash变化触发窗口onhashchange事件，监听hash变化

触发路由时视图容器更新——多数前端框架哈希路由的实现原理

> 触发hashchange的情况

- URL变化(包括浏览器的前进、后退)修改window.location.hash
- 浏览器发送http://www.baidu.com/ 至服务器，请求完毕后设置散列值为#/home
- 只修改hash部分，不发请求
- a标签可设置页面hash，浏览器自动设置hash

```
window.location.hash='abc';
let {hash}=window.location
window.addEventListener('hashchange',function(){
    //监听hash变化
})
```

> 特点

兼容性好

路径在#后面，不好看

history模式

H5新特性，允许直接修改前端路由，更新URL但不重新发请求，history可自定义地址

`window.history`属性指向 History 对象，表示当前窗口的浏览历史，保存了当前窗口访问过的所有页面网址

由于安全原因，浏览器不允许脚本读取这些地址，但允许在地址间导航

```
// 后退到前一个网址
history.back()

// 等同于
history.go(-1)
```

浏览器工具栏的“前进”和“后退”按钮，就是对 History 对象进行操作
History 对象主要有两个属性

- History.length：当前窗口访问过的网址数量（包括当前网页）
- History.state：History 堆栈最上层的状态值（详见下文）

```
// 当前窗口访问过多少个网页
window.history.length
// History 对象的当前状态
// 通常是 undefined
window.history.state
```

history.back()、history.forward()、history.go()

用于在历史之中移动

- History.back()：移动到上一个网址，等于点击浏览器后退键。对于第一个访问的网址，该方法无效
- History.forward()：移动到下一个网址，等于点击浏览器前进键。对于最后一个访问的网址，该方法无效果
- History.go()：接受一个整数作为参数，以当前网址为基准，移动到参数指定的网址，go(1)相当于forward()，go(-1)相当于back()。如果参数超过实际存在的网址范围，该方法无效；如果不指定参数，默认0，相当于刷新页面

history.pushState()

在历史中添加一条记录，不会导致页面刷新

```
window.history.pushState(state, title, url)
```

- state：对象，触发popstate事件将该对象传递到新页面。不需要可以填null
- title：新页面标题。但现在所有浏览器都忽视这个参数，所以可以填空串
- url：新网址，必须与当前页面在同域。地址栏将显示这个网址

假定当前网址是example.com/1.html，使用pushState()在浏览记录（History 对象）中添加一个新记录

```
var stateObj = { foo: 'bar' };
history.pushState(stateObj, 'page 2', '2.html');
```

添加新记录后，浏览器地址栏显示example.com/2.html，但不会跳转到2.html，也不会检查2.html是否存在，它只是成为浏览历史的最新记录。这时，在地址栏输入一个新的地址(如访问google.com)，然后点击倒退按钮，页面的 URL 将显示2.html；再点击一次倒退，URL 将显示1.html

pushState()不触发页面刷新，只导致 History 对象变化，地址栏有反应
使用该方法后，可以用History.state读出状态对象

```
var stateObj = { foo: 'bar' };
history.pushState(stateObj, 'page 2', '2.html');
history.state // {foo: "bar"}
```

如果pushState的 URL 参数设置了一个新的锚点值（即hash），不会触发hashchange事件。反过来，如果 URL 的锚点值变了，会在 History 对象创建一条浏览记录
如果pushState()方法设置了一个跨域网址，报错

```
// 报错
// 当前网址为 http://example.com
history.pushState(null, '', 'https://twitter.com/hello');
```

pushState想要插入一个跨域的网址，导致报错。防止恶意代码让用户以为他们是在另一个网站上，因为这个方法不会导致页面跳转

history.replaceState()

修改 History 当前记录，其他与pushState()一模一样

假定当前网页是example.com/example.html

```
history.pushState({page: 1}, 'title 1', '?page=1')
// URL 显示为 http://example.com/example.html?page=1

history.pushState({page: 2}, 'title 2', '?page=2');
// URL 显示为 http://example.com/example.html?page=2

history.replaceState({page: 3}, 'title 3', '?page=3');
// URL 显示为 http://example.com/example.html?page=3

history.back()
// URL 显示为 http://example.com/example.html?page=1

history.back()
// URL 显示为 http://example.com/example.html

history.go(2)
// URL 显示为 http://example.com/example.html?page=3
```

popstate事件

当同一个文档的浏览历史变化触发popstate
注意

- 仅调用pushState()/replaceState()，不会触发
- 只有点击浏览器倒退/前进，或调用History.back()、History.forward()、History.go()才会触发
- 只针对同一个文档，如果浏览历史切换，导致加载不同文档，不会触发

popstate指定回调函数

```
window.onpopstate = function (event) {
    console.log('location: ' + document.location);
    console.log('state: ' + JSON.stringify(event.state));
};

// 或者
window.addEventListener('popstate', function (event) {
    console.log('location: ' + document.location);
    console.log('state: ' + JSON.stringify(event.state));
});
```

回调函数参数event事件对象，它的state指向当前状态对象，这个state也可以通过history对象读取

```
var currentState = history.state;
```

页面第一次加载不会触发popstate事件

特点

路径正规

兼容性不比hash，需服务端支持

对于一个应用而言，url 的改变(不包括 hash 值改变)只能由下面三种情况引起：

- 点击浏览器的前进或后退按钮 => 可以监听popstate事件
- 点击 a 标签
- JS 触发 history.pushState()、history.replaceState()

前端缓存

前端缓存技术方法主要分为http缓存和浏览器缓存。

- HTTP缓存：强缓存、协商缓存
- 浏览器缓存：storage 前端数据库和应用缓存

应用缓存主要是通过manifest文件来注册被缓存的静态资源，已经被废弃，因为它的设计有些不合理的地方，在缓存静态文件的同时，会默认缓存html文件。这导致页面的更新只能通过manifest文件中的版本号来决定。所以，应用缓存只适合那种常年不变化的静态网站。如此不方便，是被废弃的重要原因。

对于一些具有重复性的HTTP请求,比如每次请求得到的数据都是一样的,我们可以把这对[请求-响应]的数据都缓存在本地,下次就直接读取本地数据,不必再通过网络获得服务器的响应了,这样可以大大提升网络性能。

浏览器对服务器最近请求过的资源进行存储，减少与服务器的交互，减少对宽带浪费，减少服务器负担。分为强缓存和协商缓存。

HTTP控制缓存的字段主要包括Cache-Control/Pragma,Expires,Last-Modified/Etag。

本地强缓存如果过期了，就需要协商缓存，也就是去看服务器上的资源是否修改。

HTTP缓存

HTTP缓存都是在第二次请求时才开始的。

缓存是指代理服务器或客户端本地磁盘内保存的资源副本。利用缓存可减少对源服务器的访问，因此节省了通信流量和通信时间。

作用

- 缓解压力
- 降低客户端获取资源的延迟，读取缓存速度快，地理位置可能比源服务器更近

强缓存

服务器与浏览器约定一个文件过期时间，向browser缓存查找该请求结果，并根据该结果的缓存规则决定是否使用该结果。**决定是否使用缓存主动性在浏览器这边.**

- 强制缓存失效，发送请求（和第一次请求一样），响应状态码为200
- 存在缓存结果和标识但结果已失效，使用协商缓存
- **存在缓存结果和标识且未失效，直接返回结果，响应状态码为403，在 size 项中标识的是 from disk cache**

发送请求时，服务器会把缓存规则放入HTTP响应报文的HTTP头中和请求结果一起返回给浏览器， 控制强制缓存的字段分别是**Expires（HTTP1.0时期使用）**

Cache-Control（HTTP/1.1时期使用）

其中Cache-Control优先级比Expires高。

Expires(HTTP/1.0)

存在于服务器返回的响应头。

Expires是HTTP/1.0控制网页缓存的字段，**其值为服务器返回该请求结果缓存的到期时间，**即再次发起该请求时，如果客户端的时间小于Expires的值时， 直接使用HTTP**本地缓存并返回状态码200** 。

缺点： Expires控制缓存的原理是使用客户端的时间与服务端返回的时间做对比，那么如果客户端与服务端的时间因为某些原因（例如时区不同；客户端和服务端有一方的时间不准确）发生误差，那么强制缓存则会直接失效，这样的话强制缓存的存在则毫无意义。

Cache-Control(HTTP/1.1)

来源于响应头和请求头。

没有采用具体的过期时间节点的方式，而是采用过期时长控制缓存，对应缓存为max-age。

在HTTP/1.1中，Cache-Control是最重要的规则，主要用于控制网页缓存，主要取值为：

> public：所有内容都将被缓存（客户端和代理服务器都可缓存）
>
> private：所有内容只有客户端可以缓存，Cache-Control的默认取值
>
> **no-cache**：跳过当前当前强缓存，客户端缓存内容，直接进入协商缓存阶段。
>
> **no-store**：所有内容都不会被缓存，即不使用强制缓存，也不使用协商缓存
>
> **max-age**=xxx (xxx is numeric)：缓存内容将在xxx秒后失效，**只有http1.1可用。**

强缓存失效进入协商缓存。

Cache-Control实现流程:

1. 浏览器第一次请求访问服务器资源时,服务器会在返回这个资源的同时,在response头部加上 Cache-Control, Cache-Control 中设置了过期时间大小;
2. 浏览器再次访问服务器该资源时,**会先通过请求资源的时间与cache-control中设置的过期时间对比,计算该资源是否过期**,若没有则使用该缓存,否则重新请求资源.
3. 服务器再次收到请求后,会再次更新Response头部的Cache-Control.

优先级:Cache-Control > Expires

协商缓存

> **协商缓存就是与服务端协商之后，通过协商结果来判断是否使用本地缓存**，通过服务端告知客户端是否可以使用缓存的方式称为协商缓存。

强制缓存失效后，browser携带缓存标识tag向server发请求，由服务器根据缓存tag决定是否使用缓存的过程。

tag分为两种：Last-Modified和ETag，不分上下。

Last-Modified(HTTP/1.0)

> **Last-Modified(响应头)，If-Modified-Since(请求头)**

最后一次修改时间。以此判断当前请求资源是否是最新的，浏览器第一次给服务器发送请求后，服务器会在响应头中加入这个字段。

browser接收后，若再次请求，会在请求头中携带If-Modified-Since字段，也就是server传来的最后修改时间。

服务器拿到**请求头**中的If-Modified-Since字段后，和服务器中该资源的最后修改时间对比：

> 若请求头中这个值小于修改时间，说明应该更新了。返回新的资源，状态码为200。
> 否则返回304，直接使用缓存。

ETag(HTTP/1.1)

服务器根据当前文件的内容，给文件**页面生成的唯一标识**。通过响应头传送给浏览器。

浏览器会在下次请求时，将这个值作为If-None-Match字段的内容，放到**请求头**。

使用 ETag 字段实现的协商缓存的过程如下：

- 当浏览器第一次请求访问服务器资源时，服务器会在返回这个资源的同时，在 Response 头部加上 ETag 唯一标识，这个唯一标识的值是根据当前请求的资源生成的；
- 当浏览器再次请求访问服务器中的该资源时，首先会先检查强制缓存是否过期，如果没有过期，则直接使用本地缓存；如果缓存过期了，会在 Request 头部加上 If-None-Match 字段，该字段的值就是 ETag 唯一标识；
- 服务器再次收到请求后，**会根据请求中的 If-None-Match 值与当前请求的资源生成的唯一标识进行比较**：
- **如果值相等，则返回 304 Not Modified，不会返回资源**；
- 如果不相等，则返回 200 状态码和返回资源，并在 Response 头部加上新的 ETag 唯一标识；
- 如果浏览器收到 304 的请求响应状态码，则会从本地缓存中加载资源，否则更新资源。

> **ETag(响应头)、If-None-Match(请求头)**

强ETab

强ETag值，不论实体发生多么细微的变化都会改变其值。

弱ETag

弱ETag值只用于提示资源是否相同。只有资源发生了根本改变，产生差异时才会改变ETag值。这时，会在字段值最开始处附加W/，'W/'（区分大小写）开头表示使用弱校验。

对于两个资源，强校验要求每个字节都相同才认为是同一个资源；而弱校验则把决定权交给开发者，根据需要区分的要素来生成相应的 ETag 。

为什么要区分强弱呢，因为大多数情况下，确保严格的字节级别的一致性是没有必要的。例如，如果我们开启了 HTTP 压缩，那么对于一个资源的响应，压缩前和压缩后的字节就是不一致的，但通常我们只关心压缩前的 ETag ，这时就需要使用弱校验的 ETag 。

两者对比

如果 HTTP 响应头部同时有 Etag 和 Last-Modified 字段的时候， Etag 的优先级更高，也就是先会判断 Etag 是否变化了，如果 Etag 没有变化，然后再看 Last-Modified。

1. 精确度：**ETag>Last***Modified。ETag按照内容给资源上标识，可准确感知资源的变化。Last*Modified不一样

- 编辑了资源文件，但内容没改变，会造成缓存失效
- Last_Modified可感知的时间单位是s，若在1s内修改了文件，不能体现出来。

2. 性能上：Last*Modified优于Etag。Last*Modified只是记录一个时间点，ETag根据文件具体内容生成哈希值。

协商缓存这两个字段都需要配合强制缓存中 Cache-control 字段来使用，只有在未能命中强制缓存的时候，才能发起带有协商缓存字段的请求。

缓存存储

- 内存缓存：快速读取和实效性
- 硬盘缓存：写入硬盘文件，需要I/O操作，重新解析改缓存内容，读取复杂，速度慢

浏览器中的缓存位置一共有四种，按优先级从高到低排列分别是：

- Service Worker
- Memory Cache
- Disk Cache
- Push Cache

大的JS 、CSS文件直接丢进磁盘，反之丢进内存

内存使用率高时，文件优先进入磁盘

Cookie

调用 Cookie 时，由于可校验 Cookie 的有效期，以及发送方的域、路径、协议等信息，所以正规发布的 Cookie 内的数据不会因来自其他Web 站点和攻击者的攻击而泄露

Set-Cookie

当服务器准备开始管理客户端的状态时，会事先告知各种信息。

> Set-Cookie: status=enable; expires=Tue, 05 Jul 2011 07:26:31 GMT; path

cookie字段

- name = value 如果用于保存用户登录态，赋值 cookie 的名称和值expires=datecookie有效期path=path将服务器上的文件目录作为cookie的使用对象domain=域名cookie适用对象的域名HttpOnly不能通过 JS 访问 Cookie，减少 XSS 攻击secure只能在协议为 HTTPS 的请求中携带same-site规定浏览器不能在跨域请求中携带 Cookie，减少 CSRF 攻击
- **expires**

指定 Cookie 的有效期。省略 expires 属性时，其有效期仅限于维持浏览器会话（Session）时间段内

- **domain**

表示cookie作用域

比如，当指定 .example.com 后，除 .example.com 以外， www.example.com或 www2.example.com等子域名都可以访问Cookie

- **secure**

限制 Web 页面仅在 HTTPS 安全连接时，才可以发送 Cookie

Set-Cookie: name=value; secure

以上例子仅当在https://www.example.com/（HTTPS）安全连接的情况下才会进行Cookie 的回收

当省略 secure 属性时，不论 HTTP还是 HTTPS，都会对 Cookie 进行回收

- **HttpOnly**

禁止JS脚本访问 Cookie

Set-Cookie: name=value; HttpOnly

使用 JavaScript 的 document.cookie 无法读取Cookie的内容。 因此，也就无法在 XSS 中利用 JavaScript 劫持 Cookie

- **SameSite**

是否允许跨域时携带Cookie

strict：任何情况都不允许作为第三方cookie

Lax：宽松模式，只能在请求方法为Get且请求改变了当前页面或打开新的页面时，允许cookie跨域访问

None：默认模式，请求自动携带cookie

Cookie 请求首部字段

告知服务器，当客户端想获得 HTTP状态管理支持时， 就会在请求中包含从服务器接收到的 Cookie。 接收到多个 Cookie 时，同样可以以多个 Cookie形式发送。

Cookie: status=enable

禁止js访问cookie

设置HttpOnly

JS Document.cookieAPI 无法访问cookie

JS设置cookie

document.cookie 属性来创建 、读取、及删除 cookie

创建 cookie：

document.cookie="username=John Doe";

添加一个过期时间（默认cookie 在浏览器关闭时删除：

document.cookie="username=John Doe; expires=Thu, 18 Dec 2043 12:00:00 GMT";

使用 path 参数告诉浏览器 cookie 的路径。默认，cookie 属于当前页面

document.cookie="username=John Doe; expires=Thu, 18 Dec 2043 12:00:00 GMT; path=/";

读取

var x= document.cookie; //以字符串方式返回所有cookie

cookie作用域

domain本身以及domain下的所有子域名

跨域请求访问cookie

1. cookie不能跨根域，但JS可以，JS可以将cookie传给另外的域再保存一次域名cookie，这样可能存在不同的cookie域包含同一个cookie值
2. browser不允许跨根域读写
3. 采用SSO单点登录方式

没有cookie会出现什么问题?

cookie：解决 如何记录客户端用户信息 的问题。保存在本地的一部分数据，再次发送请求被携带传送到服务器，通知server是否请求来自统一状态浏览器，如保持用户登录

使基于无HTTP协议记录的信息状态稳定成为可能

作用：

1. 会话状态
2. 个性化设置
3. 浏览器行为跟踪

浏览器：事件循环

1. Event Loop

> Js引擎在执行代码时候会产生执行栈，当调用异步 API，例如 setTimeout，setInterval，Promise 等回调触发时，就会进入异步任务队列，当同步代码执行完成后，就会去异步队列取出回调函数并执行，这样就形成了一个事件循环。在 Javascript 中有两种任务类型，分别是 宏任务 和 微任务 。

事件循环唯一任务：将队列和调用堆栈连接起来。

事件队列是一个存储执行任务的队列。

执行栈类似于函数调用栈的运行容器，当其为空时，JS检查事件队列，将第一个任务压入栈中执行。若调用栈和微任务队列为空，事件循环检查宏任务队列是否还有任务，从中弹出进入调用栈执行完再弹出。

- setTimeout的时间不是指马上执行，而是最快可以多久后执行，因为它会等待调用 栈为空时执行。
- 浏览器的渲染必须要调用栈为空时才会执行，正常的 forEach 会阻塞渲染。
- 在一个回调出栈，另一个回调进栈的间隙(此时栈空)，渲染得以顺利进行。

单线程JS实现异步

浏览器的内核多线程实现。

script是宏任务

若存在两个script代码块，首先执行第一个 script 中的同步代码，若这个过程中创建了微任务并进入了微任务队列，第一个 script 同步代码执行完之后，首先清空微任务队列，再去开启第二个 script 的执行。

js单线程问题

所有任务都在一个线程上完成，一旦遇到大量任务或遇到一个耗时的任务，网页就可能出现卡死，也无法响应用户的行为。

setTimeout如果在主线程上运行就会阻塞其他活动。

所以，需要做的就是，离开这个线程，同时运行这个任务。

JavaScript的单线程，与它的用途有关。

作为浏览器脚本语言，JavaScript的主要用途是与用户互动，以及操作DOM。这决定了它只能是单线程，否则会带来很复杂的同步问题。

比如，假定JavaScript同时有两个线程，一个线程在某个DOM节点上添加内容，另一个线程删除了这个节点，这时浏览器应该以哪个线程为准?

所以，为了避免复杂性，从一诞生，JavaScript就是单线程，这已经成了这门语言的核心特征，将来也不会改变。

Event Loop

是一个程序结构，用户等待和发送信息的事件。

简单说就是在程序中设置 2 个线程，一个负责程序本身的运行，称为“主线程”；另一个负责主线程和其他进程（主要是各种 I/O 操作）的通信 被称为“Event Loop 线程”（也可以翻译为消息线层）

js 就是采用了这种机制，来解决单线程带来的问题。

浏览器的 Event Loop

microtask在事件循环的macrotask执行完之后执行。

macro（宏任务）有多个，微任务（micro）队列只有一个。

先执行宏任务，再执行微任务，执行宏任务的过程中遇到微任务，依次加入微任务队列。

异步实现

1. 宏观：浏览器多线程（从宏观来看是多线程实现了异步）
2. 微观：Event Loop，事件循环（Event Loop 翻译是事件循环，是实现异步的一种机制）

常见的macrotask有：(一般由浏览器发起)DOM渲染后触发

1. script整体代码
2. setImmediate：node 的方法
3. setTimeout 和 setInterval
4. requestAnimationFrame
5. I/O
6. UI rendering

常见的microtask有：(一般由JS自身创建)DOM渲染前触发

1. process.nextTick (Node环境中)
2. Promise callback(例如 promise.then)
3. Object.observe (基本上已经废弃)
4. MutationObserver

运行过程

线程都有自己的运行数据存储空间，堆的空间比较大，所以存储一些对象；

栈的空间比较小，所以存储一些基础数据类型、对象的引用，函数的调用；

函数调用就入栈，执行完函数体的代码自动从栈中弹出——调用栈。

当栈中的函数出栈时，栈为空的话，我们会调用一些异步函数，这个异步函数会找它们的异步处理模块，异步处理模块包括定时器、promise、Ajax等，异步处理模块会找它们各自对应的线程，线程向任务队列中添加事件，再从任务队列中取出事件，去执行对应的回调。

3个注意点：

1. 整个script代码块属于宏任务
2. 当宏任务执行完，会去执行所有微任务
3. 微任务执行完再去执行下一个宏任务，等调用栈为空时执行一个微任务；调用栈不为空时，任务队列的微任务一直等待；微任务执行完又去取任务队列的宏任务，**依次执行宏任务，执行宏任务时检查当前是否存在微任务，若有微任务就去 执行完所有微任务，然后 再去执行后续的宏任务** 。

注意点：

1. 一个 Event Loop 有一个或多个 task queue（任务队列）
2. 每个 Event Loop 有一个 microtask queue（微任务队列）
3. requestAnimationFrame 不在任务队列也不在为任务队列，**在渲染阶段执行**
4. 任务需要多次事件循环才能执行完，**微任务是一次性执行完**
5. 主程序和 setTimeout 都是宏任务，一个 promise 是微任务，第一个宏任务（主程序）执行完，执行全部的微任务（一个 promise），再执行下一个宏任务（setTimeout）

记住一点，要把本次宏任务下所产生的微任务全部执行完才会执行下一个宏任务，记住是产生的，没有产生的不会执行！

```
//async函数是对一些异步操作的处理方式，一旦调用会立即执行，其中可包含微任务和宏任务
// await语句后面的代码回放进微任务队列执行
async function async1() {

    console.log('async1 start')

    //await是等待，需要把第一轮微任务执行完，再执行下面的内容

    //await后面的内容执行完，又执行宏任务

    await async2()

    console.log('async1 end')
}

async function async2() {

    return Promise.resolve().then(_ => {

        console.log('async2 promise')
    })
}

console.log('start')

setTimeout(function () {

    console.log('setTimeout')
}, 0)

async1()

new Promise(function (resolve) {

    console.log('promise1')

    resolve()

}).then(function () {

    console.log('promise2')
})

//start

//async1 start

//promise1
```

```
//async2 promise

//promise2

//async1 end

//setTimeout
```

所谓一轮事件循环就是第一轮宏任务和微任务结束。当微任务队列清空后,一个事件循环结束。

所以正确的一次 Event loop 顺序是:

1. 执行同步代码，这属于宏任务
2. 执行栈为空，查询是否有微任务需要执行
3. 执行所有微任务
4. 必要的话渲染 UI
5. 然后开始下一轮 Event loop，执行宏任务中的异步代码

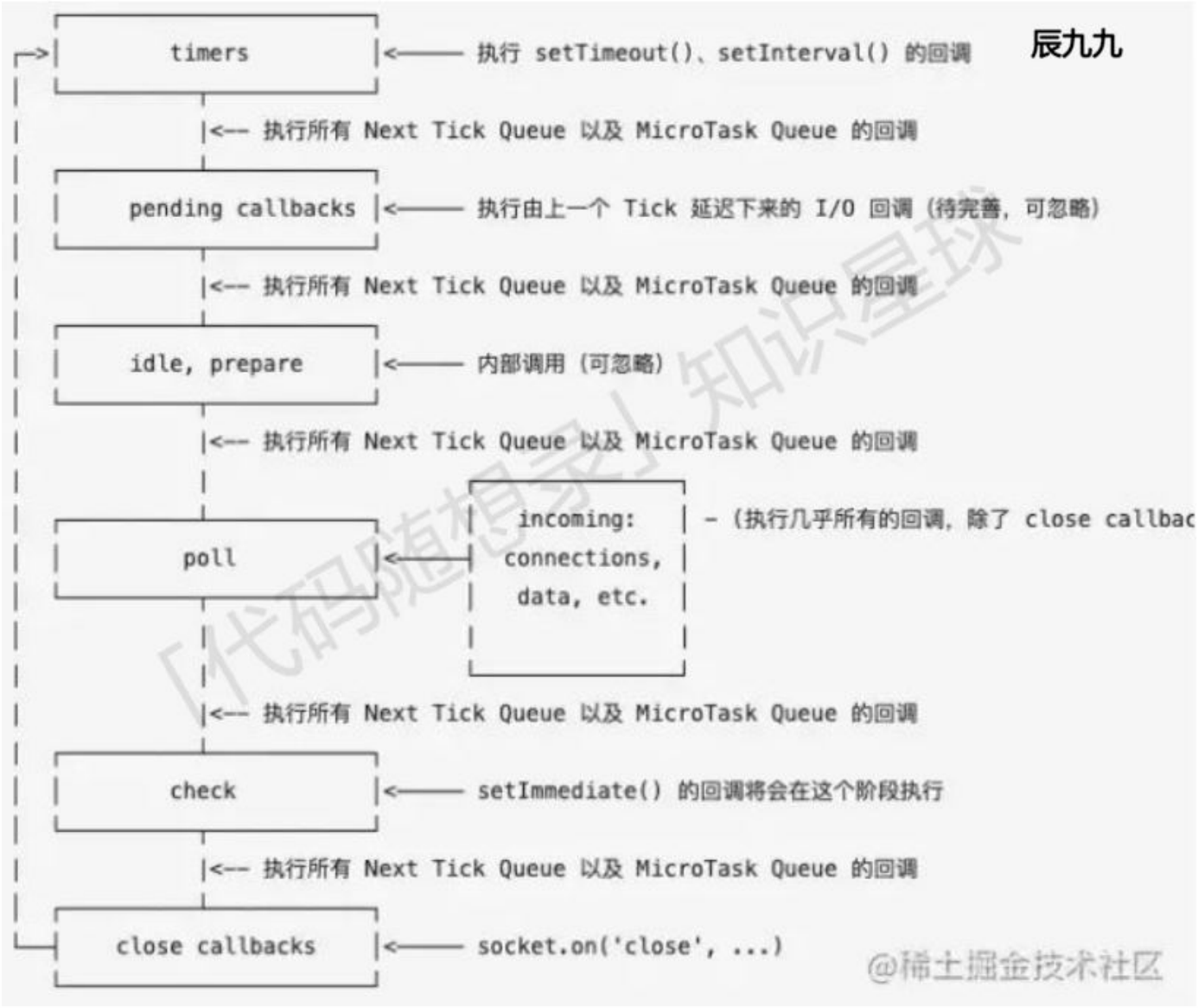
通过上述的 Event loop 顺序可知，如果宏任务中的异步代码有大量的计算并且需要操作 DOM 的话，为了更快的界面响应，我们可以把操作 DOM 放入微任务中。

2. Node事件循环

microtask 在事件循环的各个阶段之间执行（一个阶段执行完毕，就会去执行microtask队列任务）。和浏览器不同，浏览器时每次取一个宏任务执行,执行完后就跑去检查微任务队列.

但nodejs是来都来了,一次全部执行完该阶段的任务好了,那么process.nextTic和微任务啥时候执行?**process.nextTick会优先于微任务.**

事件循环顺序



执行的几个阶段

1. **timers 阶段：执行 timers 的回调，也是执行 setTimeout 和 setInterval 的回调**
2. pending IO callbacks：系统操作的回调，如定时器和setImmediate回调。
3. idle，prepare：内部使用
4. **poll：等待新的 I/O 事件进来,其他所有宏任务都属于poll阶段**，此阶段，系统会做两件事：
   1. 执行到点的定时器
   2. 执行 poll 队列中的事件
5. 且当 poll 中没有定时器的情况下，会发生以下两件事

- 如果 poll 队列不为空，会遍历回调队列并同步执行，直到队列为空或者系统限制
- 如果 poll 队列为空，会有两件事发生
  - 如果有 setImmediate 需要执行，poll 阶段会停止并且进入到 check 阶段执行 setImmediate
  - 如果没有 setImmediate 需要执行，会等待回调被加入到队列中并立即执行回调

6. 如果有别的定时器需要被执行，会回到 timer 阶段执行回调。
7. **check：执行 setImmediate 回调**
8. close callbacks：内部使用

只需关注 1、4、5 阶段

每个阶段都有一个 callbacks 的先进先出的队列需要执行，当 event loop 运行到一个指定阶段时，该阶段的 fifo 队列将会被执行，当队列 callback 执行完或者执行的 callbacks 数量超过该阶段的上限时，event loop 会转入下一个阶段。

setTimeout 和 setImmediate

二者非常相似，区别主要在于调用时机不同。

- setImmediate 设计在 poll 阶段完成时执行，即 check 阶段；
- setTimeout 设计在 poll 阶段为空闲时，且设定时间到达后执行，但它在 timer 阶段执行

区别

- node采用js v8作为解释器，v8解析完代码之后去调用node相关api
- Timer: setTimout
- IO: 进行一些IO事件
- Idle,prepare
- poll: 先看poll队列有没有事件，没有，查看是否有setImmediate的cb或者到期的timer，如果有，就放到timer queue中；如果这两个都空，就会等一个IO事件返回
- check：poll空闲的时候，进入这个阶段setImmediate
- Close callbacks：socket.destory()当socket连接在这个阶段关闭，close回调在这个阶段执行
- 因为在I/O事件的回调中，setImmediate方法的回调永远在timer的回调前执行。
- nextTick在一个阶段执行完成之后优先执行

浏览器内核

一、 浏览器的组成

浏览器的组成，主要分为两部分 外壳+内核。

外壳指菜单、工具栏等，主要是为用户界面操作、参数设置等提供的。它调用内核来实现各种功能。

内核是浏览器的核心，内核是基于标记语言显示内容的程序或模块。（我们可以写插件的，就是能对外壳进行定义，以及调用一些内核API，感觉挺好玩的，还没试过~~~）

二、浏览器作用

向服务器发出请求，在浏览器窗口中展示您选择的网络资源。这里所说的资源一般是指 HTML 文档，也可以是 PDF、图片或其他的类型。资源的位置由用户使用 URI指定，浏览器根据HTML规范进行解释。

三、浏览器内核

浏览器的核心部分是“渲染引擎”也会简称“浏览器内核，负责解释页面语法（HTML、CSS 解析、页面布局）和渲染（显示）页面。但是现在一般我们提到的大部分“浏览器内核”都包含了 JavaScript 引擎，用来处理一些动作，动态效果，所以我们可以一般认为浏览器内核包含渲染引擎和JavaScript引擎。因为浏览器的引擎不同，对我们的网页语法的解析就会产生一些不同。所以我们写CSS的时候，一般会对全局进行一些初始化，以及我们需要对页面做兼容性处理。

四、浏览器使用的内核分类

- **Trident** 内核：IE、MaxThon、TT、The World、360、搜狗浏览器等（当年的大哥了，没落后，IE都被淘汰了，不过国内一些老的机构的页面还是基于IE的，比如教师资格考试就要在IE上报名）
- **Gecko** 内核：Netscape6 及以上、FF、MozillaSuite/SeaMonkey 等（什么鬼东西，要不是搜了，都没听过）
- **Presto** 内核：Opera7 及以上
- **Webkit** 内核：Safari、Chrome 等

浏览器同源策略

一个域下的JS脚本在未经允许的情况下，不能访问另一个域的内容，同源指的是 协议 域名 和 端口均相等的情况，为同一个域。

> 跨域

由于同源策略的限制，请求发送到后端，后端返回数据时被浏览器的跨域报错拦截。

问：跨域的请求在服务端会不会真正执行?

服务端就算是想拦截，也没法判断请求是否跨域，HTTP Reqeust 的所有 Header 都是可以被篡改的，它用什么去判断请求是否跨域呢？很明显服务端心有余而力不足啊！

请求一定是先发出去，在返回来的时候被浏览器拦截了，如果请求是有返回值的，会被浏览器隐藏掉。

options:预检请求有一个很重要的作用就是 询问 服务端是不是允许这次请求，如果当前请求是个跨域的请求，你可以理解为：询问 服务端是不是允许请求在当前域下跨域发送。

当然，它还有其他的作用，比如 询问 服务端支持哪些 HTTP 方法。

预检请求虽然不会真正在服务端执行逻辑，但也是一个请求啊，考虑到服务端的开销，不是所有请求都会发送预检的。

一旦浏览器把请求判定为 简单请求，浏览器就不会发送预检了。

所以，如果你发送的是一个简单请求，这个请求不管是不是会受到跨域的限制，只要发出去了，一定会在服务端被执行，浏览器只是隐藏了返回值而已。

对于前端开发而言，大部分的跨域问题，都是通过代理解决的。

代理使用的场景是：生产环境不发生跨域，但开发环境发生跨域。所以，只需要在开发环境使用代理解决跨域即可——开发代理。

> 解决跨域

JSONP

前后端配合。只支持GET方法且不安全。利用src发送请求，传递一个回调。

不受跨域问题限制：script，link，img，href，src，因为这些操作都不会通过响应结果进行可能出现安全问题的操作。

通过 标签指向一个需要访问的地址并提供一个回调函数来接收数据。

```
//去创建一个script标签
var script = document.createElement("script");
```

```
//script的src属性设置接口地址 并带一个callback回调函数名称
script.src = "http://127.0.0.1:8888/index.php?callback=jsonpCallback";

//插入到页面
document.head.appendChild(script);

//通过定义函数名去接收后台返回数据
function jsonpCallback(data) {

    //注意 jsonp返回的数据是json对象可以直接使用

    //ajax 取得数据是json字符串需要转换成json对象才可以使用。
}
```

postMessage

(传输数据)

这种方式通常用于获取嵌入页面中的第三方页面数据。一个页面发送消息，另一个页面判断来源并接收消息

CORS

跨域资源共享 Cross Origin Resourse-Sharing

一般后端开启。是基于HTTP1.1的一种跨域解决方案。

服务端设置 Access-Control-Allow-Origin 就可以开启 CORS。 该属性表示哪些域名可以访问资源，如果设置通配符则表示所有网站都可以访问资源。

一个请求可以附带很多信息，会对服务器造成不同程度的影响，又得请求只是获取一些新闻，有的请求会改动服务器的数据。

针对不同的请求，CORS规定了3种不同的交互模式

- 简单请求
- 需要预检的请求
- 附带身份凭证的请求

简单请求

当请求**同时满足**以下条件时，浏览器会认为它是一个简单请求：

1.请求方法属于下面的一种：

- get
- post
- head

2.请求头仅包含安全的字段，常见的安全字段如下：

- Accept
- Accept-Language
- Content-Language

- Content-Type
- DPR
- Downlink
- Save-Data
- Viewport-Width
- Width

3.请求头如果包含Content-Type，仅限下面的值之一：

- text/plain
- multipart/form-data
- application/x-www-form-urlencoded

4.请求中的任意 XMLHttpRequest 对象均没有注册任何事件监听器；XMLHttpRequest 对象可以使用 XMLHttpRequest.upload 属性访问。

5.请求中没有使用 ReadableStream 对象。

如果以上条件同时满足，浏览器判定为简单请求。

当浏览器判定某个**ajax 跨域请求是简单请求时**，会发生以下的事情

1. 请求头中会自动添加Origin字段

比如，在页面http://my.com/index.html中有以下代码造成了跨域

// 简单请求

fetch('http://crossdomain.com/api/news');

请求发出后，请求头会是下面的格式：

GET /api/news/ HTTP/1.1

Host: crossdomain.com

Connection: keep-alive

```
Referer: http://my.com/index.html

Origin: http://my.com
```

最后一行，Origin字段会告诉服务器，是哪个源地址在跨域请求

2. 服务器响应头中应包含Access-Control-Allow-Origin

当服务器收到请求后，如果允许该请求跨域访问，需要在响应头中添加Access-Control-Allow-Origin字段

该字段的值可以是：

- *：表示我很开放，什么人我都允许访问
- 具体的源：比如http://my.com，表示我就允许你访问

> 实际上，这两个值对于客户端http://my.com而言，都一样，因为客户端才不会管其他源服务器允不允许，就关心自己是否被允许

> 当然，服务器也可以维护一个可被允许的源列表，如果请求的Origin命中该列表，才响应*或具体的源
>
> **为了避免后续的麻烦，强烈推荐响应具体的源**

假设服务器做出了以下的响应：

HTTP/1.1 200 OK

Date: Tue, 21 Apr 2020 08:03:35 GMT

```
Access-Control-Allow-Origin: http://my.com
```

消息体中的数据

当浏览器看到服务器允许自己访问后，高兴的像一个两百斤的孩子，于是，它就把响应顺利的交给 js，以完成后续的操作

需要预检请求

但是，如果浏览器不认为这是一种简单请求，就会按照下面的流程进行：

1. 浏览器发送预检请求，询问服务器是否允许
2. 服务器允许
3. 浏览器发送真实请求
4. 服务器完成真实的响应

比如，在页面http://my.com/index.html中有以下代码造成了跨域

```
// 需要预检的请求
fetch('http://crossdomain.com/api/user', {

    method: 'POST', // post 请求

    headers: {

        // 设置请求头
        a: 1,

        b: 2,

        'content-type': 'application/json',
    },

    body: JSON.stringify({
        name: '袁小进',
        age: 18
    }), // 设置请求体
});
```

浏览器发现它不是一个简单请求，则会按照下面的流程与服务器交互

1.浏览器发送预检请求，询问服务器是否允许

OPTIONS /api/user HTTP/1.1

Host: crossdomain.com

```
Origin: http://my.com

Access-Control-Request-Method: POST

Access-Control-Request-Headers: a, b, content-type
```

可以看出，这并非我们想要发出的真实请求，请求中不包含我们的请求头，也没有消息体。

这是一个预检请求，它的目的是询问服务器，是否允许后续的真实请求。

预检请求**没有请求体**，它包含了后续真实请求要做的事情

预检请求有以下特征：

- 请求方法为OPTIONS
- 没有请求体
- 请求头中包含
  - Origin：请求的源，和简单请求的含义一致
  - Access-Control-Request-Method：后续的真实请求将使用的请求方法
  - Access-Control-Request-Headers：后续的真实请求会改动的请求头

2.服务器允许

服务器收到预检请求后，可以检查预检请求中包含的信息，如果允许这样的请求，需要响应下面的消息格式

HTTP/1.1 200 OK

Date: Tue, 21 Apr 2020 08:03:35 GMT

```
Access-Control-Allow-Origin: http://my.com

Access-Control-Allow-Methods: POST

Access-Control-Allow-Headers: a, b, content-type

Access-Control-Max-Age: 86400
```

对于预检请求，不需要响应任何的消息体，只需要在响应头中添加：

- Access-Control-Allow-Origin：和简单请求一样，表示允许的源
- Access-Control-Allow-Methods：表示允许的后续真实的请求方法
- Access-Control-Allow-Headers：表示允许改动的请求头
- Access-Control-Max-Age：告诉浏览器，多少秒内，对于同样的请求源、方法、头，都不需要再发送预检请求了

3. 浏览器发送真实请求

预检被服务器允许后，浏览器就会发送真实请求了，上面的代码会发生下面的请求数据

POST /api/user HTTP/1.1

Host: crossdomain.com

Connection: keep-alive

```
Referer: http://my.com/index.html

Origin: http://my.com

{"name": "xiaoming", "age": 18 }
```

4.服务器响应真实请求

HTTP/1.1 200 OK

Date: Tue, 21 Apr 2020 08:03:35 GMT

```
Access-Control-Allow-Origin: http://my.com
```

添加用户成功

可以看出，当完成预检之后，后续的处理与简单请求相同

附带身份凭证请求

默认情况下，ajax 的跨域请求并不会附带 cookie，这样一来，某些需要权限的操作就无法进行

不过可以通过简单的配置就可以实现附带 cookie

```
// xhr
var xhr = new XMLHttpRequest();

xhr.withCredentials = true;

// fetch api
fetch(url, {
  credentials: 'include',
});
```

当一个请求需要附带 cookie 时，无论它是简单请求，还是预检请求，都会在请求头中添加cookie字段

而服务器响应时，需要明确告知客户端：服务器允许这样的凭据

告知的方式也非常的简单，只需要在响应头中添加：Access-Control-Allow-Credentials: true即可

对于一个附带身份凭证的请求，若服务器没有明确告知，浏览器仍然视为跨域被拒绝。

另外要特别注意的是：**对于附带身份凭证的请求，服务器不得设置 Access-Control-Allow-Origin 的值为*。**这就是不推荐使用*的原因

> 额外补充

在跨域访问时，JS 只能拿到一些最基本的响应头，如：Cache-Control、Content-Language、Content-Type、Expires、Last-Modified、Pragma，如果要访问其他头，则需要服务器设置本响应头。

Access-Control-Expose-Headers头让服务器把允许浏览器访问的头放入白名单，例如：

Access-Control-Expose-Headers: authorization, a, b

这样 JS 就能够访问指定的响应头了。

代理

适用场景：生产环境不发生跨域，但开发环境发生跨域。

开发代理：只需要在开发环境使用代理解决跨域。

```
module.exports = {

    devServer: { // 配置开发服务器

        proxy: { // 配置代理

            "/api": { // 若请求路径以 /api 开头

                target: "http://dev.taobao.com", // 将其转发到 http://dev.taobao.com
            },
        },
    },
}
```

nginx 代理

反向代理功能是nginx的三大主要功能之一（静态web服务器、反向代理、负载均衡）。

反向代理：帮服务器拿到数据，然后选择合适的服务器。

和CORS原理同，需要配置请求响应头Access-Control-Allow-Origin等字段。

怎么做反向代理与负载均衡

Nginx作为反向代理服务器，就是把http请求转发到另一个或者一些服务器上。通过把本地一个url前缀映射到要跨域访问的web服务器上，就可以实现跨域访问。

对于浏览器来说，访问的就是同源服务器上的一个url。

而Nginx通过检测url前缀，把http请求转发到后面真实的物理服务器。

并通过rewrite命令把前缀再去掉。这样真实的服务器就可以正确处理请求，并且并不知道这个请求是来自代理服务器的。

> 正向代理就是冒充客户端，反向代理就是冒充服务端。

WebSocket 协议(与HTTP同级)

因为WebSocket请求头信息中有origin字段，表示请求源自哪个域，服务器可以根据这个字段判断是否允许本次通信。

document.domain + iframe

原理：相同主域名不同子域名下的页面，

该方式只能用于二级域名相同的情况下，比如 a.test.com 和 b.test.com 适用于该方式。

只需要给页面添加 document.domain = 'test.com' 表示二级域名都相同就可以实现跨域

location.hash + iframe

通过C页面实现A和B通信

window.name(共享变量) + iframe

使用Apache做转发

逆向代理，让跨域变成同域。

鉴权

Cookie

服务端响应客户端请求时，会返回一个cookie，后续客户端的请求携带这个cookie

特点

1. 存储在客户端，可随意篡改
2. 影响性能，最大为4kb
3. 一个浏览器对于一个网站只能存不超过20个Cookie，而浏览器一般只允许存放300个Cookie
4. 移动端对Cookie支持不友好
5. 一般情况下存储的是纯文本，对象需要序列化之后才可以存储，解析需要反序列化

通过设置正确的domain和path，减少数据传输，节省带宽

Cookie-session

cookie需要的存的东西越来越多，但是cookie大小有限制

所以后端返回sessionId，客户端将sessionid存在cookie中

缓存数据库：所有机器根据sessionId去缓存系统获取用户信息和认证

局限性

1. 依赖Cookie，但Cookie可被禁用
2. 系统不停请求缓存服务器查找信息，内存开销增加
3. 存在单点登录失败的可能性

> 若负责session的机器挂了，整个登录就挂了，但项目中，负责session的机器也是有多台机器的集群进行负载均衡增加可靠性

SSO

（单点登录）三种类型

Single Sign On 在多个应用系统中，只需要登录一次，就可以访问其他相互信任的应用系统

单点登录

1. 同一站点下
2. 相同的顶级域名
3. 不同的顶级域名

相同域名和相同顶级域名下可共享cookie

但是不同域呢?

- CAS（中央认证服务）原理
- 流程和Cookie-session模式相同

Json Web Token

> 最简单的token组成:uid(用户唯一的身份标识)、time(当前时间的时间戳)、sign(签名，由token的前几位+哈希算法压缩成一定长的十六进制字符串，防止恶意三方拼接token请求)

JWT由header（头部）、payload（负载）、signature（签名）这三个部分组成，中间用.来分隔开：Header.Payload.Signature

> jwt:
> "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyaWQiOiJhIiwiaWF0IjoxNTUxOTUxOTk4fQ.2jf3kl_uKWRkwjOP6uQRJFqMlwSABcgqqcJofFH5XCo"

- 弊端

1. JWT的退出是假的登录失效，只要之前的token没过期依然可以用
2. 安全性依赖密钥
3. 加密生成的数据长

- 优点

1. 不依赖Cookie
2. 没有单点登录的cookie-sessionId模式好扩展
3. 服务器保持无状态性

session 和 token 的对比就是「用不用cookie」和「后端存不存」的对比

Webstorage

克服cookie带来的限制，不需要持续将数据返回server。

1. 提供cookie之外的存储会话数据的路径
2. 可以用来跨网站/应用 检测用户的行为而不需要服务端脚本和数据库
3. 拥有在用户即使突然断网的情况下保存部分web应用的能力，不会让你因为网络连接问题受到影响
4. 和cookie一样存在跨域策略

分类

localStorage

(针对同一个域名)

特点:

1. 生命周期：持久化的本地存储，除非手动删除数据，否则数据是永远不会过期的
2. 存储的信息在同一域中共享
3. 大小：5M，和浏览器厂商有关
4. 本质上是对字符串的读取，若存储内容过多会消耗内存空间，导致页面卡顿
5. 受同源策略限制

缺点

1.无法像cookie一样设置过期时间

2.只能存入字符串，无法直接存储对象

sessionStorage

和localStorage相似，**唯一不同就是生命周期，一旦页面关闭，sessionStorage将会删除数据**

相同点

1. **存储大小：一般都是5MB**
2. 存储位置：都存在客户端
3. 存储内容类型：只能存储字符串类型
4. 获取方式：window.localStorage
5. 应用：localStorage用于长期登录，适合长期保存在本地的数据。sessionStorage用于敏感账号一次性登录。
6. 接口封装

优点

1. 存储空间大
2. 节省网络流量
3. 快速显示
4. 安全性
5. 对于那种只需要短暂存储关闭页面就可以丢弃的数据，sessionStorage很好用

方法

```
setItem(key,value) //保存

getItem(key) //获取

key() //获取键名

removeItem(key) //清除

clear() //清除所有数据

key(index) //获取索引的key
```

IndexedDB

Indexed Database API（IndexedDB）

前端数据库有WebSql和IndexDB，其中WebSql被规范废弃，他们都有大约50MB的最大容量，可以理解为localStorage的加强版。

扩展的前端存储方式，是运行在浏览器中的非关系型数据库，理论上容量无上限。

特性

1. 储存量理论上没有上限
2. 所有操作都是异步的，相比 LocalStorage 同步操作性能更高，尤其是数据量较大时
3. 原生支持储存JS的对象
4. 是个正经的数据库，意味着数据库能干的事它都能干
5. 同源策略限制
6. 操作繁琐

Service Worker

> Service workers 本质上充当 Web 应用程序与浏览器之间的代理服务器，也可以在网络可用时作为浏览器和网络间的代理。旨在创建有效的离线体验，拦截网络请求并基于网络是否可用，以及更新的资源是否驻留，在服务器上来采取适当的动作。还允许访问推送通知和后台同步 API。

运行在浏览器背后的独立线程， 通常用来做缓存文件，提高首屏速度 。

不仅仅是cache，还通过worker的方式进一步优化，基于H5的web worker，所以不会阻塞当前JS线程的执行。

SW最重要的是

1. 后台线程：独立于当前网络线程
2. 网络代理：在网页发起请求时代理，缓存文件

使用Service Worker的话，传输协议必须是HTTPS。因为Service Worker中涉及到请求拦截，所以必须使用HTTPS协议保障安全。它可以让我们自由控制缓存哪些文件、如何匹配缓存、如何读取缓存，并且缓存是持续性的。

Service Worker 实现缓存功能一般分为三个步骤：首先需要先注册 Service Worker，然后监听到 install 事件以后就可以缓存需要的文件，那么在下次用户访问的时候就可以通过拦截请求的方式查询是否存在缓存，存在缓存的话就可以直接读取缓存文件，否则就去请求数据。

当 Service Worker 没有命中缓存的时候，我们需要去调用 fetch 函数获取数据。也就是说，如果我们没有在 Service Worker 命中缓存的话，会根据缓存查找优先级去查找数据。但是不管我们是从 Memory Cache 中还是从网络请求中获取的数据，浏览器都会显示我们是从 Service Worker中获取的内容。

Memory Cache

Memory Cache 也就是内存中的缓存，主要包含的是当前中页面中已经抓取到的资源,例如页面上已经下载的样式、脚本、图片等。读取内存中的数据肯定比磁盘快,内存缓存虽然读取高效，可是缓存持续性很短，会随着进程的释放而释放。 一旦我们关闭Tab页面，内存中的缓存也就被释放了。

那么既然内存缓存这么高效，我们是不是能让数据都存放在内存中呢？ 这是不可能的。计算机中的内存一定比硬盘容量小得多，操作系统需要精打细算内存的使用，所以能让我们使用的内存必然不多。

Disk Cache

Disk Cache 也就是存储在硬盘中的缓存，读取速度慢点，但是什么都能存储到磁盘中，比之 Memory Cache 胜在容量和存储时效性上。

Push Cache

Push Cache（推送缓存）是 HTTP/2 中的内容，当以上三种缓存都没有命中时，它才会被使用。它只在会话（Session）中存在，一旦会话结束就被释放，并且缓存时间也很短暂，在Chrome浏览器中只有5分钟左右，同时它也并非严格执行HTTP头中的缓存指令。

1. 所有的资源都能被推送，并且能够被缓存,但是 Edge 和 Safari 浏览器支持相对比较差
2. 可以推送 no-cache 和 no-store 的资源
3. 一旦连接被关闭，Push Cache 就被释放
4. 多个页面可以使用同一个HTTP/2的连接，也就可以使用同一个Push Cache。这主要还是依赖浏览器的实现而定，出于对性能的考虑，有的浏览器会对相同域名但不同的tab标签使用同一个HTTP连接。
5. Push Cache 中的缓存只能被使用一次
6. 浏览器可以拒绝接受已经存在的资源推送
7. 可以给其他域名推送资源

关于一段代码执行前的“编译”

一、编译语言和解释语言

我们写的代码一般不能被电脑直接识别，所以就有了编译器和解释器，也对应不同的语言

编译语言一般是 C / C++， Go等，这类语言首次执行会通过编译器编译出机器能读懂的二进制文件，每次运行的时候会直接运行这个二进制文件。

解释语言一般是Python，JavaScript等，每次执行都需要解释器进行动态解释和执行。

二、编译的过程

那么编译一般来说有六个步骤

1. **分词 / 词法分析**
   a. 把字符组成的字符串分解成有意义的代码块（词法单元）， 如 var a = 2，会分解为 var ， a， =，2；
2. **解析 / 语法分析**
   a. 将上面的词法单元流（数组）转换为由元素逐级嵌套所组成的代表程序语法结构的树，也就是AST
3. **生成抽象语法树（AST）**
4. **词义分析**
5. **生成二进制文件或者字节码**
6. **执行**

编译语言和解释语言的主要区别在于词义分析后生成的类型不同，他们都会生成AST这一步，3,4,5，6可以合并称之为代码生成

我们前端开发用的JavaScript就是解释型语言，其实一开始是没有字节码的，是直接将AST编译成机器码，所以效率是很高的，但是机器码占用的内存过大，所以又有了字节码的出现。这里又涉及到一门新的技术 JIT （即使编译）的出现。

所以我们的V8引擎使用的是 字节码 + JIT 的技术

AST 抽象语法树

首先我们知道我们输入一段 JavaScript 代码，他会先经过词法和语法分析得到 AST 也就是我们的抽象语法树。那么为什么需要转化为 AST 呢？那是我们编写的的的无论是编译型语言还是解释型语言，都是不能被编译器或解释器所理解的。而且转化为 AST 还能有很多的便利，我们的 babel， ESlint 很多工具都是在这一步进行的。

1. 首先第一步是分词，也就是词法分析

分词就是把一行行代码拆分为每一个 token，也就是 语法上不可再分，最小的单位。同时去除空格，对 token 分类等。

```
// 源码
let aaa = '12: 30'

// Tokens数组
[
  { type: { ... }, value: "let", start: 0, end: 3, loc: { ... } },
  { type: { ... }, value: "aaa", start: 4, end: 7, loc: { ... } },
  { type: { ... }, value: "=", start: 8, end: 9, loc: { ... } },
  { type: { ... }, value: "12:30", start: 10, end: 15, loc: { ... } },
]
```

上面的代码就会拆分为，其中关键字“let”、标识符“aaa” 、赋值运算符“=”、字符串“12: 30”四个都是 token，而且它们代表的属性还不一样。添加到 Token 数组中。

2. 词法分析之后就进行解析，也就是语法分析
其作用是将上一步生成的 token 数据，根据语法规则（扫描 token 列表，形成语法二叉树）转为 AST。如果源码符合语法规则，这一步就会顺利完成。但如果源码存在语法错误，这一步就会终止，并抛出一个“语法错误”。

通过这样就构建起 AST 抽象语法树。

LHS 和 RHS

首先变量的赋值操作会执行两个动作，首先编译器会在当前作用域中声明一个变量（之前没有声明），然后运行时引擎会在该作用域中查找该变量，如果能找到就会对他赋值

那么引擎查找中就涉及到我们这里要讲到的 LHS查询 和 RHS查询

这里我们不能简单的认为是左右来判断，而是根据：

LHS：赋值操作的目标是谁
RHS：谁是赋值操作的源头

也可以说查找的目的是对变量进行赋值，那么就会使用LHS 查询；如果目的是获取变量的值，就会使用RHS 查询

考虑以下代码：

> console.log(a);

其中对a 的引用是一个RHS 引用，因为这里a 并没有赋予任何值。相应地，需要查找并取得a 的值，这样才能将值传递给console.log(…)

相比之下，例如：

> a = 2;

这里对a 的引用则是LHS 引用，因为实际上我们并不关心当前的值是什么，只是想要为=2 这个赋值操作找到一个目标。

看个例子

> function foo(a) { //LhS
> console.log(a) //RHS
> }
> foo(2) //RHS

这里面就涉及到两种查询了

同时 LHS 和 RHS 会现在当前的作用域查询，没有找到的话就会到上一层查找，最后到全局作用域

DOM 事件流

一、什么是事件流：

我们点击一个按键，那么他是如何传递的呢，是从文档顶部一层层传入到这个按键，还是从这个按键传出去呢?

- 曾经在 IE 就是从里面往外（确定的逐步到不确定的），叫做事件冒泡
- 而到了 Netscapte 就是从外面往里（不确定的逐步到确定的），叫做事件捕捉
- 最终在 W3C 的定义规范中，就把两者都包含了~~~（哈哈哈，谁都赢了，但谁也没赢）

所以现在在 DOM事件模型中会分为捕获和冒泡。一个事件发生后，会在子元素和父元素之间传播（propagation）。这种传播分成三个阶段。

1. 捕获阶段：事件从window对象自上而下向目标节点传播的阶段；
2. 目标阶段：真正的目标节点正在处理事件的阶段；
3. 冒泡阶段：事件从目标节点自下而上向window对象传播的阶段。

那么为什么会有事件捕获和事件冒泡呢 这就涉及到事件委托

那么什么是事件委托呢，事件委托就是利用事件冒泡，只指定一个事件处理程序，就可以管理某一类型的所有事件

性能优化

1. 网站打开速度
2. 动画流畅
3. 表单提交速度
4. 列表滚动页面切换是否卡顿

优化手法

1. 聚焦用户
2. 尽快响应用户输入
3. 动画执行流畅
4. 最大化主线程空闲时间
5. 网页可交互性

HTML

1. 避免HTML中直接写CSS
2. viewport加速页面渲染
3. 使用语义化标签
4. 减少标签的使用，DOM解析是一个大量遍历的过程
5. 避免src和href空值
6. 减少DNS查询数

CSS

1. 避免后代选择符
2. 避免链式~
3. 避免!important
4. link代替@import

@import会将请求变得 串行化，导致加载增加延迟

1. 减少回流与重绘
2. CSS 放在 head 中
3. 压缩CSS 开启gzip压缩
4. 骨架屏+合理的loading
5. 优化选择器路径，避免 过多嵌套
6. 选择器合并：压缩空间和资源开销
7. 精确样式

比如设置{padding-left:10px}的值，避免{padding:0 0 0 10px}这样的写法

1. 异步加载CSS
2. 避免通配符

.a .b *{} 像这样的选择器，从右到左解析，在解析过程中遇到通配符（**）会去遍历整个dom

1. 少用float：渲染时计算量大
2. 0值不加单位：兼容性
3. 避免使用 昂贵 的属性

因为他们渲染成本挺高，渲染速度慢一些

1. border-radius
2. box-shadow

3. opacity
4. transform
5. filter
6. position: fixed
7. 使用先进布局方式——flex

会造成阻塞吗

1. CSS加载不会阻塞DOM树解析
2. CSS加载会阻塞DOM树渲染
3. CSS加载会阻塞后面JS执行

JS

1. 避免循环操作DOM
2. 事件委托

绑定事件时，不绑定到目标元素上，而是绑定到其祖先元素上

1. 监听事件少
2. 新增节点时，无需增加事件绑定
3. scrip标签放在body后

CSS放在

因为JS阻塞DOM的构建(因为DOM解析遇到JS会停止解析，开始下载脚本并执行)，CSSOM的构建阻塞JS执行

1. 压缩文件
2. 按需加载
3. 避免逐个操作DOM样式，尽可能预留好CSS样式，通过样式名的修改改变DOM样式，集中式操作减少reflow的次数
4. 减少iframe数量

合成

1. 合成层的位图 交由GPU处理，比CPU块
2. repaint本身，不影响其它层
3. transform和opacity不触发重绘

代码问题

1. 频繁使用JSON.parse/JSON.stringify大对象
2. 正则灾难性回溯
3. 内存泄漏

网络相关

DNS预解析

预先获得域名所对应的 IP，href的值是预解析的域名

preload和prefetch

preload强制浏览器立即获取资源， 具有较高优先级

prefetch的资源获取时可选 和 较低 优先级的，是否获取取决于浏览器

缓存

强缓存

协商缓存

选择合适缓存策略

对于大部分的场景都可以使用强缓存配合协商缓存解决

1. 使用 Cache-control: no-store ，表示该资源不需要缓存
2. 使用 Cache-Control: no-cache 并配合 ETag ，表示该资源已被缓存，但是每次都会发送请求询问资源是否更新
3. 使用 Cache-Control: max-age=31536000 并配合策略缓存使用，对文件进行指纹处理，一旦文件名变动就会立刻下载新的文件

使用HTTP2.0

1. 解析速度快
2. 头部压缩
3. 多路复用
4. 服务器推送
5. 浏览器由并发请求限制

预加载

1. 有些资源不需要马上用到，但希望尽早获取
2. 预加载强制请求资源，不会阻塞 onload 事件

预渲染

将下载的文件预先在后台渲染

预渲染可以提高页面的加载速度，但是要确保该页面百分百会被用户，否则白白浪费资源

减少HTTP请求

使用服务端渲染

SSR （service side render）

Gzip压缩

避免重定向

渲染优化

懒执行

将某些逻辑延迟到使用时再计算。该技术可以用于首屏优化，对于某些耗时逻辑并不需要在首屏就使用的，就可以使用懒执行。懒执行需要唤醒，一般可以通过定时器或者事件的调用来唤醒。

懒加载

将不关键的资源延后加载——尽量只加载用户正 浏览 或即将会 浏览的图片

只加载自定义区域（通常是可视区域，但也可以是即将进入可视区域）内需要加载的东西

对于图片来说，先设置图片标签的 src 属性为一张占位图，真实的图片资源放入自定义属性data-src 中，当进入自定义区域时，就将自定义属性替换为 src 属性，这样就会下载图片资源

节流防抖

防抖：单位时间内多次触发，只执行最后的那一次，原理：延迟执行，期间但凡有新的触发就重置定时器

节流：单位时间只触发一次，原理：上锁，只有满足一定间隔时间才能执行

图片

电商类项目，存在大量图片，banner 广告图、菜单导航栏、列表头图等

图片众多以及体积过大影响页面加载速度

为啥?

有些图片请求并发，Chrome最多支持并发请求数有限，其他请求被push进队列中等待或停滞，直到上轮请求完成后才被发出，一部分资源需要排队等待时间，过多的图片影响页面加载展示

合适图片格式

1. WebP 格式具有更好的图像数据压缩算法，更小的图片体积，拥有肉眼识别无差异的图像质量，缺点是兼容性并不好
2. 小图使用 PNG，对于大部分图标，完全可以使用 SVG 代替
3. 照片使用 JPEG

4. 雪碧图(将多个图标文件整合到一张图片中)

可能请求非常多的小图片，会受到浏览器并发 HTTP 请求数的限制

1. 图片压缩
2. 不用图片，用CSS代替
3. 对于移动端来说，屏幕宽度就那么点，完全没有必要去加载原图浪费带宽。一般图片都用 CDN 加载，可以计算出适配屏幕的宽度，然后去请求相应裁剪好的图片

JPEG/JPG

1. **高质量 有损压缩，体积小，不支持透明**
2. 应用于轮播图 大的背景图、banner

PNG

1. **无损压缩，质量好，体积大，支持透明**
2. **应用小的logo**

SVG

1. 体积小，不失真，兼容好
2. 应用于图标

GIF

1. 支持透明

Webp

有损压缩与无损压缩（可逆压缩）的图片文件格式

比PNG/JPEG格式小

支持透明度

体积和效果上都做的不错

```
<picture>
    <source type="image/webp" srcset="/static/img/perf.webp">
    <source type="image/jpeg" srcset="/static/img/perf.jpg">
    <img src="/static/img/perf.jpg">
</picture>
```

webpack压缩

配置 image-webpack-loader

雪碧图

CSS Sprites ，精灵图，图像合成技术，主要用于小图片显示

同原域名请求有最大并发限制，Chrome为6个，如 页面有10个小图，需要10次请求，2次并发

若把10个图合成一个大图，只需1次请求

1. 减少请求次数
2. 减少服务器压力
3. 减少并发
4. 提高加载速度
5. 减少鼠标滑过的一些bug
6. 解决网页设计师在图片命名上的困扰

iconfont

通过字体方式展示图标，用户 图标渲染、简单图形、特殊字体等

1. 轻量，已修改
2. 减少请求次数

内联Base 64

图片转为base64串，解析图片不会请求下载，而是解析字符串

缺点

1. 比使用二进制体积增大 33%
2. 全部内联后，原本可并行加载的图片会串行放入请求

适用于 更新频率低、首屏或骨架图上的小图标

CSS代替图

实现修饰效果，半透明、阴影、圆角、渐变等

CDN图片

图片懒加载

暂时不设置图片的src属性，先卸载data-src中，等图片到了可视区域再将真实src放进src属性

使用background-url，应用到具体元素时，才会下载图片

图片预加载

需要展示大量图，将图提前加载到本地缓存

响应式图加载

在不同分辨率的设备上显示不同尺寸的图

渐进式图片

和骨架屏 原理类似

在图完全加载完前先显示低画质版本，让用户产生图片加载变快的印象，而不是盯着一片空白

其他文件优化

1. 服务端开启文件压缩功能
2. 执行 JS 代码过长会卡住渲染，对于需要很多时间计算的代码可以使用 Webworker

webWorker是运行在后台的JS，另开一个子线程，不会影响性能

CDN

内容分发网络

静态资源使用 CDN 加载，由于浏览器对于单个域名有并发请求上限，可以考虑使用多个 CDN 域名

其他

使用 Webpack 优化

1. 对于 Webpack，打包项目使用 production 模式，会自动开启代码压缩
2. ES6 模块开启 tree shaking，移除没有使用的代码
3. 优化图片，对于小图使用 base64 的方式写入文件
4. 按照路由拆分代码，实现按需加载
5. 给打包出来的文件名添加哈希，实现浏览器缓存文件

监控

采集——>上传——>分析——>报警

渲染几万条数据不卡?

通过 requestAnimationFrame 来每 16 ms 刷新一次。

SPA 首屏优化

浏览器从响应用户输入网址地址，到首屏内容渲染完成时间，整个网页不一定要完全渲染完成，但需要展示当前视窗内容

加载慢的原因

1. 网络延时
2. 资源文件体积过大
3. 资源加载重复发送请求
4. 加载脚本时，渲染内容阻塞

解决

1. 减少入口文件体积
2. 静态资源本地缓存
3. UI框架按需加载
4. 图片资源压缩
5. 组件重复打包
6. 开启GZip压缩
7. 使用SSR

减少入口文件体积

路由懒加载，将不同路由对应组件分割成不同代码块，待路由被请求时单独打包路由，使得入口文件变小

以函数形式动态加载路由，可以把各自的路由文件分别打包，只在解析给定路由时，才会加载路由组件

静态资源本地缓存

HTTP缓存和localStorage

按需加载

对UI库按需引用

图片资源压缩

对icon，使用在线字体图标

组件重复打包

若A.js是一个常用库，多个路由使用它会造成重复下载

在webpack的config中，修改CommonChunkPlugin的配置

minChunks为3表示会把使用3次及以上的包抽离，放进公共依赖文件

使用SSR

服务端渲染，Server Side ，组件或页面通过服务器生成html字符串，发送到浏览器

降低APP首页开屏渲染时间

1. 二次启动时先利用缓存渲染，后台进行异步数据更新
2. 减少不必要的请求 和数据获取
3. 提前请求或减少http请求
4. 优化图片文件尺寸，压缩图片格式，压缩代码
5. 启用gzip 压缩功能
6. 使用CDN
7. 网址后面加上“/”:对服务器而言，不加斜杠服务器会多一次判断的过程，加斜杠就会直接返回网站设置的存放在网站根目录下的默认页面。
8. Ajax采用缓存调用

页面白屏

网络请求，返回状态

组件样式布局，组价未显示

网页卡顿原因

1. 网络请求是否过多，导致数据传输变慢，可通过缓存优化
2. 资源bundle太大，考虑拆分
3. 代码是否有太多循环在主线程上花费太长时间
4. 浏览器某个帧 中 渲染太多东西
5. 页面渲染时，大量回流和重绘
6. 内存泄露

动画性能优化

1. 合理布局
2. transform代替left、top 减少重排
3. 硬件加速
4. 避免不必要的图形层
5. requestAnimationFrame实现动画

动画每一帧都是re-render，显示器刷新频率 60 HZ，意味着每一帧任务耗时不超过 16ms

前端安全

XSS跨站脚本攻击

Cross Site Scripting 为了和 CSS 区别，CSS 指的是层叠样式表 (Cascading Style Sheets)

用户输入或使用其他方式向代码中注入其他JS，然后JS代码被执行。

1. 可能是写一个死循环、获取cookie登录
2. 监听用户行为
3. 修改DOM伪造登录表单
4. 页面生成浮窗广告

反射型

XSS代码通过**URL**注入。

因为恶意脚本通过作为网络请求的参数，经过服务器，然后反射到HTML文档中执行解析，服务器不会存储这些恶意脚本。

反射型 XSS 漏洞常见于通过 URL 传递参数的功能，如网站搜索、跳转等。

存储型

XSS代码发送到**服务器数据库**，前端请求数据时，将XSS代码发送到前端。

场景：留言区提交一段脚本执行，若前后端未做好转义的工作，评论内容存在数据库，页面渲染过程中直接执行，相当于是执行一段未知逻辑的JS代码。

如论坛发帖、商品评论、用户私信等。

文档型

XSS攻击不会经过服务端，作为中间人的角色，数据传输过程劫持到网络数据包，**然后修改里面的HTML文档**。

劫持包括：

1. WIFI路由劫持
2. 本地恶意软件

DOM 型 XSS 跟前两种 XSS 的区别：DOM 型 XSS 攻击中，取出和执行恶意代码由浏览器端完成，属于前端 JavaScript 自身的安全漏洞，而其他两种 XSS 都属于服务端的安全漏洞。

防范：一个信念，两个利用

1.对输入转码过滤

2.利用CSP

浏览器内容安全策略，核心就是服务器决定浏览器加载哪些资源，功能：

1. 限制其他域下的资源加载
2. 禁止向其它域提交数据
3. 提供上报机制

3.HttpOnly

HttpOnly类型的cookie阻止JS对其的访问(标记或授权对话)

阻止 XSS 攻击：服务器对脚本进行过滤或转码，利用 CSP 策略，使用 HttpOnly；

CSRF-跨站伪造请求（钓鱼）

Cross-Site Request Forgery

攻击者诱导受害者进入第三方网站，在第三方网站中，向被攻击网站发送跨站请求。利用受害者在被攻击网站已经获取的注册凭证（比如cookie），绕过后台的用户验证，因此可以冒充用户对被攻击的网站执行某项操作。

利用所在网站的目前登录信息，悄悄提交各种信息，比XSS更恶劣。

> 本质：**利用cookie会在同源请求中携带发送给服务器的特点，以此冒充用户。**

利用服务器的验证漏洞和用户之前的登录状态模拟用户操作。

点击链接后，可能发生3件事

1. 自动发送GET请求。利用src发送请求

2. 自动发送POST请求
3. 诱导点击发送GET请求

防范

1.SameSite

SameSite可以设置为三个值，Strict、Lax和None。

a. 在Strict模式下，浏览器完全禁止第三方请求携带Cookie。比如请求sanyuan.com网站只能在sanyuan.com域名当中请求才能携带 Cookie，在其他网站请求都不能。

b. 在Lax模式，宽松一点，但是只能在 get 方法提交表单况或者a 标签发送 get 请求的情况下可以携带 Cookie，其他情况均不能。

c. 在None模式下，也就是默认模式，请求会自动携带上 Cookie。

2.验证来源站点

请求头中的origin和referer

origin只包含域名信息，referer包含具体的URL路径。

3.CSRF Roken

利用token（后端生成的一个唯一登陆态，传给前端保存）每次前端请求都会带token，后端检验通过才同意请求。

- 敏感操作需要确认
- 敏感信息的cookie只能有较短的生命周期

4.安全框架

如Spring Security。

SQL注入攻击

Sql 注入攻击，将恶意的 Sql查询或添加语句插入到应用的输入参数中，再在后台 Sql服务器上解析执行。

如何让Web服务器执行攻击者的SQL语句？

SQL注入的常规操作就是将有毒的SQL语句放置于Form表单或请求参数之中，然后提交到后端服务器，如果后端服务器没有做输入安全检查，直接将变量取出执行SQL语句，就容易中招。

预防方式如下：

- 严格检查输入变量的类型和格式
- 过滤和转义特殊字符
- 对访问数据库的Web应用程序采用Web应用防火墙

其他种类的攻击：

DDoS全称Distributed Denial of Service：分布式拒绝服务攻击。是拒绝服务攻击的升级版。拒绝攻击服务顾名思义，让服务不可用。常用于攻击对外提供服务的服务器，像常见的：

- Web服务
- 邮件服务
- DNS服务

- 即时通讯服务

DNS劫持

JSON劫持

暴力破解

HTTP报头追踪漏洞

信息泄露

目录遍历漏洞

命令执行漏洞

文件上传漏洞等等

HTTPS中间人攻击

客户端和服务器之间的桥梁、双向获取并且篡改信息

标准回答

攻击者通过与客户端和客户端的目标服务器同时建立连接，作为客户端和服务器的桥梁，处理双方的数据，整个会话期间的内容几乎完全被攻击者控制。攻击者可以拦截双方的会话并且插入新的数据内容

加分回答

中间人攻击的过程：

1. 本地请求被劫持，所有请求均发送到中间人服务器
2. 中间人服务器返回中间人自己的证书
3. 客户端创建随机数，通过中间人证书中的公钥加密 传送给中间人，凭随机数构造对称加密对传输内容加密传输
4. 中间人拥有客户端返回的随机数，可以对内容解密
5. 中间人以客户端的请求内容向真的服务器发送请求
6. 服务器通过建立的通道返回加密后的数据
7. 中间人对加密算法内容解密
8. 中间人对内容加密传输
9. 客户端通过和中间人建立的对称加密算法对返回数据解密

缺少证书的验证，客户端完全不知道自己的网络被拦截，数据被中间人窃取

Vue

Vue的优点和特点

一、优点

1. **首先是双向绑定**
   这也是我们使用框架的一大优势，VUE使用MVVC架构，在 VUE.2X 中使用Object.definProperty () 来劫持绑定数据，在VUE.3X 中使用 Proxy 劫持。如果按照最初的开发，我们前端开发不仅仅是需要完成业务代码的实现，同时还需要对每个 DOM元素进行获取绑定时间和数据。而双向绑定使得我们只需要专注于业务代码的实现上。
2. **简单易学**
   VUE以简单，易上手的特点，在国内很多企业得到使用。同时作为国人开发的框架，中文文档，相关的论坛，生态完善，也便于我们学习和遇到问题的解决方案。
3. **虚拟DOM**
   使用虚拟DOM，结合DIFF 算法能减少性能损耗。他会把我们多次的操作合并为一次，推送到真实的DOM。另外补充，我们说虚拟DOM减少损耗是有条件，是指在频繁操作的情况下，不然肯定简单的获取操作最快的，期间没有很多计算等处理。
4. **组件化的思想**
   实现组件的封装，我们往往是使用组件开发的思想去封装组件，这样不仅便于复用也好维护修改！

二、缺点

1. **生态不够完善**
   相比 angular 和 react 来说，生态环境较为不足，在构建大型的应用方面 ，企业使用 react 的比较多。而中小型企业使用 VUE 比较多。

三、react与VUE的区别和优缺点

VUE 和 React 的共同点是，都是组件化的思想，虚拟DOM，数据绑定，而不同点在于，首先两者的设计思想不同，前者定位降低前端开发的门槛，而后者推崇函数式编程。另外在 React 中，是使用 JSX的写法，把HTML 和 CSS都写入到 JavaScript 中。还有的就是他们的 DIFF算法实现也不太一样。

Vue的生命周期

一、各个生命周期的作用

首先在 vue 生命周期中有 十个阶段:

```
1、beforeCreate（创建前）
```

> 在实例初始化之后,进行数据侦听和事件/侦听器的配置之前同步调用。

因为 data 和 methods 中的数据都还没有初始化，常常在该阶段执行与 vue 数据无关的事件，比如我们的 loading 等待事件。

```
2、created（创建后）
```

> 在实例创建完成后被立即同步调用。在这一步中，实例已完成对选项的处理，意味着以下内容已被配置完毕：数据侦听、计算属性、方法、事件/侦听器的回调函数。然而，挂载阶段还没开始，且 $el property 目前尚不可用。

在该阶段已经完成 data 和 methods 的初始化了，只是页面还未渲染，可以在该阶段来发起请求获取数据等，以及操作 data 和 调用 methods 等

3、beforeMount（挂载前）

> 在挂载开始之前被调用：相关的 render 函数首次被调用。
> 该钩子在服务器端渲染期间不被调用

4、mounted（挂载后）

> 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。
> 注意 mounted 不会保证所有的子组件也都被挂载完成。如果你希望等到整个视图都渲染完毕再执行某些操作，可以在 mounted 内部使用 vm.$nextTick
> 该钩子在服务器端渲染期间不被调用

这时候 vue实例完成初始化，且挂载渲染到页面了，最早我们可以在这个阶段来操作 页面上的 DOM节点。

5、beforeUpdate（更新前）

> 在数据发生改变后，DOM 被更新之前被调用。这里适合在现有 DOM 将要被更新之前访问它，比如移除手动添加的事件监听器。
> 该钩子在服务器端渲染期间不被调用，因为只有初次渲染会在服务器端进行

该阶段此时实例中的数据已经是最新的啦，但是页面的还未更新

6、update（更新后）

> 在数据更改导致的虚拟 DOM 重新渲染和更新完毕之后被调用。
> 当这个钩子被调用时，组件 DOM 已经更新，所以你现在可以执行依赖于 DOM 的操作。然而在大多数情况下，你应该避免在此期间更改状态。如果要相应状态改变，通常最好使用计算属性或 watcher 取而代之。
> 注意，updated 不会保证所有的子组件也都被重新渲染完毕。如果你希望等到整个视图都渲染完毕，可以在 updated 里使用 vm.$nextTick
> 该钩子在服务器端渲染期间不被调用

避免在这个阶段更改状态的，因为这样可能会导致更新无限循环，曾经我做过个获取表格的高度，如果页面大小发送改变就更新表格的大小，在该阶段执行导致了页面的无限循环，结果死机啦~

7、activated（激活前）

> 被 keep-alive 缓存的组件激活时调用。
> 该钩子在服务器端渲染期间不被调用

8、deactivated（激活后）

> 被 keep-alive 缓存的组件失活时调用。
> 该钩子在服务器端渲染期间不被调用

```
9、beforeDestory（销毁前）
```

> 实例销毁之前调用。在这一步，实例仍然完全可用。
> 该钩子在服务器端渲染期间不被调用

```
10、destoryed（销毁后）
```

> 实例销毁后调用。该钩子被调用后，对应 Vue 实例的所有指令都被解绑，所有的事件监听器被移除，所有的子实例也都被销毁。
> 该钩子在服务器端渲染期间不被调用

总结

Vue 的生命周期也就是 Vue实例从创建到销毁的过程，总共分为8个阶段 创建前/后，载入前和后，更新前和后，销毁前和后。

- **创建前和后：**
  - 在 beforeCreated 阶段，Vue 实例刚在内存中被创建出来，data 和 methods 还未初始化，都为 undefined
  - 在 created 阶段，vue实例的 data 和 methods 就已经存在了，但是DOM 还未挂载渲染
- **载入前和后：**
  - 在 beforeMount 阶段，DOM挂载完成，数据也初始化完成，但是数据的双向绑定还是显示{{}}，这是因为Vue采用了Virtual DOM（虚拟Dom）技术。先占住了一个坑
  - 在 mounted 阶段，DOM挂载完成，上一个阶段留的萝卜坑也补上了
- **更新前/后：**
  - 在 beforeUpdate 阶段，此时实例中的 data 以及是最新的了，但是界面上显示的数据还是旧的，此时还没有开始重新渲染DOM节点
  - 在 update 阶段，此时实例中的 data 和界面上显示的数据都是新的了，已经完成了 DOM节点 的渲染
- **销毁前/后：**
  - 在 beforeDestroy 阶段， Vue实例销毁之前调用。在这一步，实例仍然完全可用。
  - 在 beforeDestroy 阶段 ，Vue实例所有的事件监听以及和 dom的绑定都解除了

Vue-router

一、路由常见的属性

路由的两个属性$router和$route。

$router

1. $router.app
2. $router.mode
3. $router.currentRoute
4. router.addRoutes(routes)
5. router.beforeEach(to,from,next)

6. router.afterEach()
7. router.go(n)
8. router.push(location)
9. router.replace(location)
0. router.back()
1. router.forward()
2. router.resolve(location)
3. router.onReady(callback,[errorCallback]){}
4. router.onError(callback)

$route

1. $route.path
2. $route.query
3. $route.params
4. $route.hash
5. $route.fullPath
6. $route.name
7. $route.matched
8. $route.redirectedFrom

Vue-router中hash模式和history模式的区别

- 最明显的是在显示上，hash模式的URL中会夹杂着#号，而history没有。
- Vue底层对它们的实现方式不同。hash模式是依靠onhashchange事件(监听location.hash的改变)，而history模式是主要是依靠的HTML5 history中新增的两个方法，pushState()可以改变url地址且不会发送请求，replaceState()可以读取历史记录栈,还可以对浏览器记录进行修改。
- 当真正需要通过URL向后端发送HTTP请求的时候，比如常见的用户手动输入URL后回车，或者是刷新(重启)浏览器，这时候history模式需要后端的支持。因为history模式下，前端的URL必须和实际向后端发送请求的URL一致，例如有一个URL是带有-路径path的(例如www.lindaidai.wang/blogs/id)，如果后端没有对这个路径做处理的话，就会返回404错误。所以需要后端增加一个覆盖所有情况的候选资源，一般会配合前端给出的一个404页面

Vue路由传参

路由传参方式可划分为 params 传参和 query 传参，而 params 传参又可分为在 url 中显示参数和不显示参数两种方式

方式A：这种需要在路由配置好可以传递参数XXX的，不是最方便的

```
路由配置
{
 path: '/child/:XXX',
 component: Child
}
父组件
<router-link to = "/child/XXX"></router-link>
子组件读取
this.num = this.$route.params.XXX
```

方式B：这种同样需要在路由配置好可以传递参数XXX的，不过是用到push方法的

```
路由配置
{
 path: '/child/:XXX',
 component: Child
}
父组件
this.$router.push({
        path: `/child/${XXX}`
      })
子组件读取
this.num = this.$route.params.XXX
```

上面两种方式都会在地址显示出传递的参数，类似get请求

方式C：这种不需要在路由配置好根据路由的名称，需要保持一致

```
路由配置
不需要配置，但是子组件的name必须与父组件传递的路由一致
父组件
this.$router.push({
        name: 'B',
        params: {
          XXX: '妈呀'
        }
      })
子组件读取
this.num = this.$route.params.XXX  //妈呀
```

方式D：这种不需要在路由配置好根据路由的名称，通过query来传递

```
路由配置
不需要配置，但是子组件的name必须与父组件传递的路由一致{
父组件
this.$router.push({
        path: '/child',
        query: {
            XXX: '妈呀'
        }
      })
子组件读取
this.num = this.$route.query.XXX
```

总的来说使用方式C和D最为多，毕竟不需要对路由配置做修改

Vue过滤器

定义过滤器有两种常见的方式: **双花括号插值和 v-bind 表达式**

```
<!-- 在双花括号中 -->
{{ message | capitalize }}

<!-- 在 `v-bind` 中 -->
<div v-bind:id="rawId | formatId"></div>
```

其实一般用来格式文本，比如我们往往拿到的时间是时间戳，那么就可以很简单的通过过滤器来格式成我们常见的格式。

使用上的我们先要定义过滤的方法，这个可以全局定义或者组件内定义

```
//组件内定义
filters: {
  capitalize: function (value) {
    if (!value) return ''
    value = value.toString()
    return value.charAt(0).toUpperCase() + value.slice(1)
  }
}
//全局定义
Vue.filter('capitalize', function (value) {
  if (!value) return ''
  value = value.toString()
  return value.charAt(0).toUpperCase() + value.slice(1)
})

new Vue({
  // ...
})
```

Vue中实现组件通信的方式

在VUE中实现通信有很多种的方式，每一种都有其对应的使用情况。

首先我们看看有哪些方式：

- props
- emit
- v-model
- refs
- provide/inject
- eventBus
- vuex/pinia(状态管理工具)

常见的是上面的这几种，少见的其实还可以算上插槽 slot， 混入，路由携带参数，localStorage， $parent / $children等

1. props
常常使用在父组件传递給子组件通信的过程中，首先在子组件中使用 props，来接收对应的属性，而在父组件中使用子组件的地方，添加上面定义的属性。

2. emit
这个就和上面的相反，是使用在子组件给父组件传递值的中。子组件中声明对应的事件，当子组件触发事件，就会通过 this.$emit('事件','数据')传递到，而在父组件中使用子组件的地方，添加上面定义的事件，这个可以获取子组件传来的值。

3. provide/inject
这对组合往往使用在层级比较深的时候，比如A组件下可能还有 B组件 ，B组件下有C组件…E组件.而使用这对API，就能无论层级有多深都能获取到

4. eventBus

也就是事件总线，简单粗暴，可以到处飞。可以不管你是不是父子关系，通过新建一个Vue事件bus对象，然后通过bus.emit 触发事件，bus.on 监听触发的事件。但不建议乱用，不好维护。

5. vuex
对于大型的项目来说往往是很必要的，尤其单页面应用，很多页面嵌套页面，关系很多。而使用 VUEX 就能便捷的统一管理。

总结

常见的组件通信方式有通过 props / emit / provide 和 inject / eventBus / vuex 等，一般根据不同的场景来决定使用的方式。比如父子组件通信使用 props ，反过来使用 emit 。而当层级很多的时候使用 provide ，全局的状态管理使用 vuex等。

Vue事件总线（EventBus）

简单来说，我们知道Vue中有多种的方式来父子组件传值

- 父传子：props
- 子传父：this.$emit('事件','数据')

- 兄弟组件：中间事件总线思想（也叫事件巴士）
- 多层级：父组件中使用 provide ，在子组件中使用 inject

今天我们要来讲的就是事件总线这个，简单来说，它的使用就是首先创建事件总线

```
//两种方式，一种是创建一个单独的文件如下面的的bus.js在用到的地方导入，
//还有中就是在项目的main中实例化

// bus.js
import Vue from 'vue'
export const EventBus = new Vue()

// main.js
Vue.prototype.$EventBus = new Vue()
```

创建好之后我们就很简单啦

```
//在需要发送事件的地方用通过 this.$emit('事件' , '数据')来发送
//在需要接收的地方用xxx.$on('事件',(数据)=>{ })来接收
//下面是一个完整的例子

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
<div id="app">
    <Parents></Parents>
</div>
<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
<script>
    //创建一个媒介来连通两个是组件
    const medium = new  Vue()

    const ABorder = {
        template: `<div>我是儿子A
        <div>我的名字叫：{{name}}</div>
        <button @click="newName">给自己命名</button>
        </div>`,
        created () {
            medium.$on('letDiscussionName',(val)=>{
              alert(val)
            })
        },
        //用props来接收父组件传来的值，同时父组件要绑定属性
        props: {
            name:{
                //设置类型，符合才接收
```

```
            type: String,
            //不传默认为这个
            default: '我也不知道呀'
        }
    },
    methods: {
      newName() {
          this.$emit('newName','大黄狗')
      }
    }
};
const BBorder = {
    template: `<div>我是儿子B
                <button @click="discussionName">讨论</button>
               </div>`,
    methods: {
        discussionName() {
            medium.$emit('letDiscussionName','儿子A来讨论')
        }
    }
}

const Parents = {
    data() {
        return {
            msg: '孩儿们好呀！',
            aName: '大花猫',
            bName: '大黄狗'
        }
    },
    methods: {
        chang(value) {
            this.aName = value
        }
    },
    //可以有下面两种形式来调用组件
    template: `<div>我是父组件:{{msg}}
    <hr/>
    <a-border :name= 'this.aName'  @newName = 'chang'></a-border>
    <hr/>
    <BBorder></BBorder></div>`,
    components: {
        ABorder,
        BBorder
    }
}
//实例化一个vue，并且绑定到app这个元素
const vm = new Vue({
    el: '#app',
    components: {
        Parents,
    }
```

```
    })
</script>
</body>
</html>
```

Vue中Scoped原理

Vue的作用域样式 Scoped CSS 的实现思路如下：

1. 为每个组件实例生成一个能唯一标识组件实例的标识符，记作 InstanceID；
2. 给组件模板中的每一个标签对应的 Dom 元素添加一个标签属性，格式为 data-v-实例标识（示例：< div data-v-e0f690c0=“” >）；
3. 给组件的作用域样式 < style scoped> 的每一个选择器的最后一个选择器单元增加一个属性选择器

特点

1、 将组件的样式的作用范围限制在了组件自身的标签

即：组件内部，包含子组件的根标签，但不包含子组件的除根标签之外的其它标签；所以 组件的 css 选择器也不能选择到子组件及后代组件的中的元素（子组件的根元素除外）；

> 因为它给选择器的最后一个选择器单元增加了属性选择器 [data-v-实例标识] ，而该属性选择器只能选中当前组件模板中的标签；而对于子组件，只有根元素 即有 能代表子组件的标签属性 data-v-子实例标识，又有能代表当前组件（父组件）的 签属性 data-v-父实例标识，子组件的其它非根元素，仅有能代表子组件的标签属性 data-v-子实例标识；

2、 如果递归组件有后代选择器，则该选择器会打破特性1中所说的子组件限制，从而选中递归子组件的中元素；

> 原因：假设递归组件A的作用域样式中有选择器有后代选择器 div p ，则在每次递归中都会为本次递归创建新的组件实例，同时也会为该实例生成对应的选择器 div p[data-v-当前递归组件实例的实例标识]，对于递归组件的除了第一个递归实例之外的所有递归实例来说，虽然 div p[data-v-当前递归组件实例的实例标识] 不会选中子组件实例（递归子组件的实例）中的 p 元素（具体原因已在特性1中讲解），但是它会选中当前组件实例中所有的 p 元素，因为 父组件实例（递归父组件的实例）中有匹配的 div 元素；

Vue中的ref

ref

常用来辅助开发者，获取 DOM 元素或组件的引用，以及用于在父子组件中获取对方的某个元素进行取值，调用方法等。

在每个 Vue 的组件实例上， 都包含一个 $refs 对象， 里面存储着对应的 DOM 元素或组件的引用

- 默认情况下，组件的 $refs 指向一个空对象
- 如果想要使用 ref 引用页面上的组件实例，则可以按照如下方式：
- 使用ref属性，为对应的组件添加引用名称

```
<my-counter ref="counterRef"></my-counter>
<button @click="getRef">获取 $refs 引用</button>
methods: {
  getRef(){
    // 通过 this.$refs. 引用的名称，可以引用组件的实例
    console.log(this.$refs.counterRef)
    // 引用到组件的实例之后，就可以调用组件上的methods方法
    this.$refs.counterRef.add()
  }
}
```

这个方法可以说很便利，但是不要太依赖了，往往在不能通过其他方法获取的时候回才比较建议使用，毕竟我们因该尽量

减少添加，而是复用可以复用的部分。

> $refs 只会在组件渲染完成之后生效，并且不是响应式的。这仅作为一个用于直接操作子组件的“逃生舱”——你应该避免在模板或计算属性中访问
> tip：如果获取不到的时候，可以试一试使用nextTick

Vue中$nextTick的使用

$nextTick

官方解释：在下次 DOM 更新循环结束之后执行延迟回调。在修改数据之后立即使用这个方法，获取更新后的 DOM。

因为 vue 中不是数据一发生改变就马上更新视图层的，如果这样会带来比较差的性能优化，而是在 vue 中会添加到任务队列中，待执行栈的任务处理完才执行。所以 vue 中我们改变数据时不会立即触发视图，但是如果我们需要实时获取到最新的DOM，这个时候可以手动调用 nextTick。

Vue中的keep-alive

用 keep-alive 包裹组件时，会缓存不活动的组件实例，而不是销毁，使得我们返回的时候能重新激活。keep-alive 主要用于保存组件状态或避免重复创建。避免重复渲染导致的性能问题。

常见场景
页面的缓存，如上面的，保存浏览商品页的滚动条位置，筛选信息等

> 注意
> 这个 keep-alive 和 这个 Connection: Keep-Alive 是不一样的，这个属性的作用是，往往我们三次握手后，传输完数据就会断开连接进行四次挥手，关闭TCP连接，但是当我们在头信息中加入了该属性，那么TCP会在发送后仍然保持打开状态，这样浏览器就可以继续通过同一个TCP连接发送请求，保持TCP连接可以省去下次请求时需要建立连接的时间，提升资源加载速度。

Vue中的key

关于 key，我们常常在 v-for 中会接触到，当我们需要进行列表循环的时候，如果没有使用 key，就会有警报提示。

```
//场景一：循环列表
<div v-for="num in numbers" :key="index">
  {{num}}
</div>
//场景二：通过时间戳强制重新渲染
<div :key="+new Date()" >+new Date()</div>
```

那么为什么需要 key呢?

因为使用 key 相当于给 vnode 添加上一个唯一的 id，主要是在 DOM DIFF算法中使用的下面我们看个例子：

1. 如果上面的场景一
   items 的值为 [1,2,3,4,5,6,7,8,9,10] 那么就是渲染出十个div 是吧。如果没有 key，那么当我们现在 items 的值变为 [0,1,2,3,4,5,6,7,8,9] 呢，那么机会第一个原来是 “1” 的 div内容变为“0”，而原来是“2”的div内容变为“1”…以此类推。（因为没有key属性，Vue无法跟踪每个节点，使用的是“就地复用” 得策略，通过这样的方法来完成变更）
   但是如果key的情况下，Vue能跟踪每个节点，就会直接新增一个 div 添加到内容是“1”的div前面。根据key，更准确， 更快的找到对应的vnode节点。
   设置key能够大大减少对页面的DOM操作，提高了diff效率，更加的高效更新虚拟DOM
2. 还有的用途就是我们的场景二
   用来强制替换元素，当key改变时，Vue认为一个新的元素产生了，从而会新插入一个元素来替换掉原有的元素。所以上面的元素会被删除而而重新添加。但是如果没有添加 key的话，就会直接替换div 里面的内容。而不是删除添加元素。

注意点

尽量不要使用索引 index 做 key 的值，要使用唯一的标识值，比如 id 之类的，因为如果使用数组索引 index 为 key，当向数组中指定位置插入一个新元素后，因为这时候会重新更新 index索引，对应着后面的虚拟DOM的 key 值全部更新了，这时候做的更新是没有必要的，就像没有加key一样，因此index虽然能够解决key不冲突的问题，但是并不能解决复用的情况。如果没有唯一ID的情况下，可以使用用组件的uid拼接index的形式，因为uid是唯一的 。或者生成一个时间戳~。

总结

关于在 v-for 中使用 key 是因为使用 key 相当于给节点添加上一个唯一的 id，在 DOM DIFF算法中能更高效的更新虚拟DOM。

当没有使用的时候，如果我们需要对一个循环渲染的列表插入一个，那么实际上会 “就地复用” 的策略，比如我们需要在最前面插入一个，那么就会后面的全部往后变更，发生多次DOM操作，这是因为 Vue无法跟踪每个节点，而使用 key 后，节点标识，定位到最前面，然后插入修改，只发生一次DOM操作，大大优化了性能。

v-show 和 v-if 的区别

1、本质上

- v-show 是把标签里的 display 设置为 none，所以页面上是可见的
- v-if 是动态的操作DOM元素，页面上不可见的

2、性能上

- 要是需要频繁的操作的话，肯定是 v-show ，因为他只是操作css的值，不会频繁触发重排。但是 v-if 是不断的向DOM树添加或删除元素，在比较少改变的时候比较合适。
- v-show 无论任何条件，初始都会渲染，v-if是惰性的，如果初始条件为 false，初始不会渲染 DOM ，为 true 才会渲染。因此 v-if 有更高的切换开销，而 v-show 有更高的初始渲染开销

v-if 和 v-for 为什么不建议一起使用呢

`v-if`：根据表达式的真值来有条件的渲染元素，在切换时元素及它的数据绑定 / 组件被销毁并重建。

`v-for`：基于源数据多次渲染元素或模板块。此指令之值，必须使用特定语法 alias in expression，为当前遍历的元素提供别名，同时设置 独一无二的 key 。

我们有时候会写出下面的例子：

```
<div id="app">
    <p v-if="isShow" v-for="item in items">
        {{ item.title }}
    </p>
</div>
```

而实际上这段代码会先循环然后再进行判断，每次v-for都会执行v-if，造成不必要的计算，这会带来性能上的浪费。避免这种使用在同一个元素的情况，我们可以在外层嵌套template（页面渲染不生成dom节点），在这一层进行v-if判断，然后在内部进行v-for循环

```
<template v-if="isShow">
    <p v-for="item in items">
</template>
```

如果条件出现在循环内部，可通过计算属性computed提前过滤掉那些不需要显示的项。

```
<div>
  <div v-for="(user,index) in activeUsers" :key="user.index" >
    {{ user.name }}
  </div>
</div>

data () {
    return {
      users,: [{
        name: '1',
        isShow: true
      }, {
        name: '2',
        isShow: true
```

```
      },{
        name: '3',
        isShow: false
      }]
    }
  }

computed: {
  activeUsers: function () {
    return this.users.filter(function (user) {
      return user.isShow;//返回isShow=true的项，添加到activeUsers数组
    })
  }
}
```

总结

首先，两者的作用分别是 v-if 是根据真值来判断是否渲染组件，而 v-for 是循环渲染组件列表。之所以不能一起使用是因为 v-for 的优先级 比 v-if 高。

这会导致的情况就是，当我们在一个循环列表中做判断时，他会先全部列表渲染出来，然后再根据判断做销毁，这会造成不必要的浪费，就是做好了，多出来的砸坏。一般我们可通过计算属性对数据做过滤，把不必要的数据过滤掉，就不必使用 v-if 。

如果使用v-show就不一样了，v-show 和 v-if 的区别在于v-show 是通过display属性设置为 none 来隐藏属性的，而 v- if 是销毁 DOM上的元素的，所以如果同时使用 v-show 和 v-for 是没有影响的。

computed 和 watch 的区别

一、这两者解决的问题

当我们的数据发生改变的时候，所有和该数据有关的数据都自动更新，执行我们定义的方法。不同于 methods 是需要相关的方法和交互来实现调用执行的。

二、computed 和 watch 以及 methods 的区别

1、性质上
methods：定义函数，**手动调用**
computed：计算属性，return 返回结果，**自动调用**
watch： 观察，**监听，发生改变就调用**

2、使用场景
methods：一般不处理数据的逻辑，用于获取数据，和改变状态等情况
computed：多用于一个数据受多个数据影响的情况，具有 缓存的特性，避免每次重新计算
watch： 多用于一个数据影响多个数据的情况，且类似异步等情况的时候建议使用

3、执行时间
computed 和 methods 的初始化是在 beforeCreated 和 created 之间完成的。（以及Props, data都是）

4、缓存

computed：有缓存，重新渲染时，要是值没有改变，会直接返回之前的

watch：无缓存，重新渲染时，要是值没有改变，也会执行

5、是否异步

computed：不支持异步，但异步监听的时候，无法监听数据变化

watch：支持异步监听

三、computed 和 methods 的区别

无论是定义为一个计算属性还是一个函数方法，对于结果来说都是一样的，不同点在于，计算属性是根据依赖值变化才发生改变，而函数方法需要调用执行。

Vue中的插槽使用

在开发中，往往会把一个复杂的页面分为多个组件去组合，或者当复用性比较高的也会提取为一个组件，但是当我们设计的组件还不能在某些特殊的情况满足我们的需求的时候，我们还需要添加额外的时候，那么该怎么办呢？插槽就出现了。

当实际使用的组件不能完全的满足我们的需求的时候，我们就可以用插槽来分发内容，往我们的这个组件中添加一点东西。

基本的使用就是在子组件里面我们需要添加内容的地方加上

```
<slot></slot>
```

那么我们在父组件中使用该子组件的时候，就能直接在这个子组件中添加进入内容。这就是最基本的使用。

那么要是我们想在父组件中传递多个内容呢？这时候就有我们的对应使用的具名插槽了，我们可以对插槽命名狮子对应在子组件中的位置。

```
//我们可以在子组件的插槽中写入内容，也就是默认值，当我们在父组件中没有传递值的时候，就会使用默认值
<slot>我是默认的插槽值</slot>

//具名插槽
如下，我们在子组件中的插槽可以命名，这样我们在父组件中可以通过名字来确定位置

 //子组件B
    const BBorder = {
        template: `<div>我是儿子B
        <slot name= 'header'> 这里是头部</slot>
        <slot> 这里是默认 </slot>
        <slot name='footer'> 这里是尾部 </slot></div>`,
    }
    const Parents = {
        methods: {
        },
        //实例化BBorder这个子组件
        template: `<div>我是父组件:{{msg}}
            <BBorder>
```

```
                <template v-slot:header> 头部 </template>
                <p>A 9999999999999</p>
                <template v-slot:footer> 尾部 </template>
            </BBorder>
            </div>`,
        //注册两个子组件
        components: {
            BBorder
        }
    }
```

同样的，也能出传递值给到父组件中，但是我么要注意，父子组件的编译作用域是不同的，父级模板里的所有内容都是在父级作用域中编译的；子模板里的所有内容都是在子作用域中编译的。

从子组件中传递值到父组件

```
//子组件中
  <slot :user="user">
    {{ user.lastName }}
  </slot>

//父组件中
<current-user>
  <template v-slot:default="slotProps">
    {{ slotProps.user.firstName }}
  </template>
</current-user>
```

Vuex

一、Vuex是什么

Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式。它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。

也就是说，vuex是基于 vue 框架的一个状态管理库，那么什么是状态管理呢?

> 状态管理模式：简单来说就是集中式的存储管理组件的状态思想

二、Vuex 解决了什么

我们思考一个场景：

比如我们有一个在线多人聊天的，类似微信的，我们有一个通知栏来告诉用户是否有未读信息，但是出现了通知栏偶尔会给出错误的通知，也就是用户收到一条新的未读消息的通知，但是当他们检查它是什么时，这只是他们已经看到的消息。

这是为什么呢？因为这里面

- 多个组件 / 实例依赖于同一状态，兄弟组件之间的状态是无法传递。

- 并且来自不同视图的行为当需要变更同一状态的时候，经常需要通过事件等来变更和同步状态，当遗漏的时候就会出现有些地方状态更新错误。

```
var Vo= {}

var A = new Vue({
  data: Vo
})

var B = new Vue({
  data: Vo
})
```

比如某个数据，在实例A和实例B中使用了，而这个数据在实例A中修改了，如果我们不做处理，这个状态是不会更新到实例B中的，对于实例B来说，这个数据是没有发生改变的。小型的项目我们还比较好处理，对实例B更新就好了。

但是，当我们在做大型的开发的时候的话，往往会有多个实例 / 组件（A，B，C，D…）共享 **一个数据，这种情况它们之间互相关联，导致数据的复杂性大大增加，数据的状态变得不可预测或难以理解。结果就是应用程序变得无法扩展或维护，变得臃肿难以维护及复用**

所以我们需要使用 vuex 统一的管理（思想是实现 Flux 模式），不必考虑各处的处理，只需要考虑一个，且具有下面的优点：

1. 全局管理
2. 状态变更跟踪
3. 规范化
4. 便于调试

> **Flux 模式的基本思想:**
>
> - 单一数据来源
> - 数据是只读的
> - 更新是同步的

二、Vuex 的好处

1. 首先多层嵌套的组件、兄弟组件间的状态会更好管理维护，且使用 vuex 也是组件通信中的一种
2. 减轻服务端压力，对于同一份数据集，请求会缓存，所有共用
3. 当一个项目比较大型的时候，这是便于团队之间开发和维护的，和便于调试

二、五个核心的概念 state / getters / mutations / action / modules

在 Vuex 中，核心就是 一个 “store” 仓库，包含我们应用中的大部分状态，同时提供包含 state / getters / mutations / action / modules等在内的管理调用方法。

1、state

state 的存储状态类似于就组件中data，是我们所有的集中数据的地方，访问 “store” 的状态可以使用两种方式，通过 this.$store.state属性 的方式或者 mapState 辅助函数。

```
//存储
 const store = new Vuex.Store({
  // 存储状态数据
```

```
  state: {
      count: 0,
      info:{
          name:"wenmo",
          age:18
      }
  },
)
//读取方式
 computed: {
    count () {
      return store.state.count
    }
  }

//下面这种是全局注册了store的
 computed: {
    count () {
      return this.$store.state.count
    }
  }
```

每当 store.state.count 变化的时候, 都会重新求取计算属性，并且触发更新相关联的 DOM。

辅助函数 mapState：当一个组件需要获取多个状态的时候，将这些状态都声明为计算属性会有些重复和冗余

```
computed: {
  // 使用对象展开运算符将此对象混入到外部对象中
  ...mapState({
    // ...
  })
}
```

2、getters

getters 就是类似组件中的计算属性，可以简单理解为 state 的计算属性，依赖于 state 的值，state 发生改变就会同时改变

```
const store = new Vuex.Store({
  state: {
    todos: [
      { id: 1, text: '...', done: true },
      { id: 2, text: '...', done: false }
    ]
  },
  getters: {
    doneTodos: state => {
      return state.todos.filter(todo => todo.done)
    }
  }
})
```

3、mutations

mutations的作用是用来修改 “store” 里面的值，并且是唯一的方式。

我们不能直接修改 store 容器中的状态，需要先在容器中注册一个事件函数，当需要更新状态时候要通过触发该事件来完成。

```
const store = new Vuex.Store({
  state: {
    name: "wenmo"
  },
  mutations: {
  // 注册事件
  addCount(state){
  state.name = "WENMO"
  }
}
})
//在实例中显式的调用执行
new Vue({
   el: '#app',
    store,
    methods:{
        // 显示的调用
        handleAdd(){
            this.$store.commit('addCount')
        }
    }
})
```

4. action

Action类似于Mutation, 但是是用来代替Mutation进行异步操作的.action用于异步的修改state，它是通过muation间接修改state的。

若需要异步操作来修改state中的状态，首先需要action来注册事件，组件视图在通过 dispatch 分发方式调用该事件，该事件内部提交mutation中的事件完成改变状态操作，总之，通过action这个中介来提交mutation中的事件函数.

```
const store = new Vuex.Store({
  state: {
    name: "wenmo"
  },
  mutations: {
  // 注册事件
  setName(state){
  state.name = "WENMO"
  }
};
    // 注册事件,提交给 mutation
  actions: {
      setAction(context){
```

```
            setTimeout(() => {
                context.commit('setName')
            }, 1000)
        },
    }
})
//在实例中显式的调用执行
new Vue({
    el: '#app',
     store,
     methods:{
         // 显示的调用
         handleAdd(){
              this.$store.dispatch('setAction')
         }
     }
})
```

5. modules

```
const moduleA = {
  state: () => ({ ... }),
  mutations: { ... },
  actions: { ... },
  getters: { ... }
}

const moduleB = {
  state: () => ({ ... }),
  mutations: { ... },
  actions: { ... }
}

const store = createStore({
  modules: {
    a: moduleA,
    b: moduleB
  }
})

store.state.a // -> moduleA 的状态
store.state.b // -> moduleB 的状态
```

6、关于这五个方法的总结

- state 对应的是在 store 中存储数据，类似组件中的 data
- getter 对应的是 store 中访问数据，类似组件中的 计算属性
- mutation 对应的是 store 中修改数据，并且是唯一的更新方式
- action 类似于 mutation，用来代替 mutation进行异步操作的
- modules 的作用是用来分割 store 的，每个 module 拥有自己的state、mutation、action、getters等

总结

Vuex 是一个专为 Vue开发的状态管理模式，使用集中式存储，管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。内部实现是 FLUX 模式，也就是约定数据的单向流动。

主要有五个属性，state / getter / mutation / action / modules ，分别是用来存储数据，访问数据，修改数据，进行异步操作的，和划分模块的。

Flux 模式

一、解决什么问题
当应用程序中有多个组件共享数据时，它们互连的复杂性会增加到数据状态不再可预测或不可理解的程度。因此，应用程序变得无法扩展或维护。

举个例子就是，比如我们一个在线多人聊天，类似微信的，我们有一个通知栏来告诉用户未读信息，但是通知栏偶尔会给出错误的通知。用户将收到一条新的未读消息的通知，但是当他们检查它是什么时，这只是他们已经看到的消息。

这就是多组件之间状态管理的混乱，因为 MVC架构中没法保证数据的单向流动。

二、 特点

- **单一事实来源**，任何要在组件之间共享的数据，都需要保存在一个地方，与使用它的组件分开。这个单一的位置被称为“商店”。组件必须从该位置读取应用程序数据，而不是保留自己的副本以防止冲突或分歧。
- **数据只可读**，组件可以自由地从存储中读取数据。但他们不能更改存储中的数据，至少不能直接更改。相反，他们必须通知存储他们更改数据的意图，并且存储将负责通过已定义“更新”的函数进行这些更改。
- **更新是同步的**，我们根据上面的两个原则，就能很便利的追踪数据的状态变化和调试，但是如果是异步的话，我们就不能很好的判断执行的顺序。

三、具体使用

具体的应用就是我们常见的 VUEX 和 React 中。

虚拟DOM

一、什么是虚拟 **DOM**呢?

再过去或者我们原生JavaScript的时候，我们可以发现，当我们需要改变视图的数据的时候，我们往往需要先获取到这个 DOM 元素，然后对其进行更新。也就是:

数据改变-->操作DOM--> 视图更新

但是我们在 VUE或者 React 中是直接改变 Data 就能实现视图的更新了。

数据改变-->视图更新

那么要是我们每一次数据改变都需要操作 DOM，那就非常麻烦，而且慢。因为 JavaScript执行时很快的，但是操作 DOM就不是了。所以就有了虚拟 DOM了

数据改变-->操作虚拟 DOM（计算变更）-->操作真实的 DOM--> 视图更新

那么什么虚拟 DOM 呢?

可以说虚拟 DOM 本质上是 JS 和 DOM 之间的映射，表现为是一个能描述 DOM结构 及其属性信息的 JS对象。那么下面我们看一个例子为什么这么说吧：

```
//我们定义的 DOM
<div id="app">
  <p class="text">hello</p>
  <h1 class="text">hello world!!!</h1>
</div>

//转换为虚拟 DOM
{
  tag: 'div',
  props: {
    id: 'app'
  },
  chidren: [
    {
      tag: 'p',
      props: {
        className: 'text'
      },
      chidren: [
        'hello'
      ]
    },
     {
      tag: 'h1',
      props: {
        className: 'text'
      },
      chidren: [
        'hello world!!!'
      ]
    }
  ]
}
```

实际上就是通过 JavaScript 对象来作为基础的树，用对象的属性来描述节点，然后通过映射成真实的 DOM 结构。

二、那么这么做有什么好处呢

1、 优化性能

我们常说要减少重排的发生，那是为什么呢，因为那会涉及到 DOM 树的重新渲染。而操作 DOM 树是比较慢的，且 DOM 元素的数量谁很庞大的，当操作 DOM 很容易带来页面的性能问题。很容易给用户带来不好的体验，而这点是很重要的，可以说我们前端页面是与用户的第一个窗口。

比如，正常情况下，当我们要更新10个节点的时候，浏览器就会计算 10次 ，一次一次的进行更新。而在使用虚拟 DOM ，他相对就好像多了一个缓存区，当 DOM 操作（渲染更新）比较频繁时，它会先将前后两次的虚拟 DOM 树进行对比，定位出具体需要更新的部分，生成一个“补丁集”（也就是一个 js对象），最后只把“补丁”一次性打在需要更新的那部分真实的 DOM 树上，避免了很多没必要的计算，提高了性能。

2、抽象化渲染过程

很多人认为虚拟 DOM 最大的优势是 diff 算法，减少 JavaScript 操作真实 DOM 的带来的性能消耗。虽然这一个虚拟 DOM 带来的一个优势，但并不是全部。虚拟 DOM 最大的优势在于抽象了原本的渲染过程，实现了跨平台的能力，而不仅仅局限于浏览器的 DOM，可以是安卓和 IOS 的原生组件，可以是近期很火热的小程序，也可以是各种 GUI。
解耦了视图层和渲染平台，带来了更多的可能性

> 有点像我们语言中，都会编译成 AST ，而在这个阶段，可以实现多种转化，就像 Babel ，和 ESList 等。

3、缺点也是有的：在初次渲染的时候，会多出一层 虚拟DOM的计算，而且使用虚拟DOM也不一定是高效的，往往当我们每一次改动不大的时候，才是相对高效的，但是如果我们改动大的话，相比还多出了更多的计算，是不利的。

总结

虚拟 DOM 本质上是 JS 和 DOM 之间的映射，表现为一个能描述 DOM结构 及其属性信息的 JS对象。没有使用 虚拟DOM 时，当我们改变数据–>就需要操作DOM–>然后视图更新，当是当我们使用虚拟DOM，就是我们改变数据–>框架就会操作虚拟 DOM进行计算变更–>之后映射到真实的 DOM上–> 完成视图更新。

当我们说虚拟DOM的优点，我们常说 虚拟DOM 提高效率，其实这需要根据场景而言的！当操作 DOM 的数量比较少的时候，使用 虚拟DOM 反而效率低，因为添加了更多的计算等操作。而当操作大量 DOM 的时候，使用 虚拟DOM 会将多次操作合并为一次更新，减少 JavaScript 操作真实 DOM 的带来的性能消耗。另外一个重要的优点是 虚拟 DOM 抽象了原本的渲染过程，实现了跨平台，跨端的能力！不再局限于浏览器！

DIFF 算法

DIFF算法的作用：**同层树节点比较的算法**

那么 DIFF算法是如何工作的?

一、首先是先计算新老DOM的最小变化

该算法会先遍历一遍老的DOM，然后在遍历新的DOM，最后会判断是改变/新增/删除来重新排序。
这样无疑是非常耗费计算的，我们看一看出总共遍历了三回，如果有一千个节点，那么就湖发生了十亿次的计算。

二、diff算法的优化
diff 算法的优化，也就是这个算法的核心部分了，简单来说就是针对具有相同父节点的同层新旧子节点进行比较，不相同的话就会新增或者删除，而不是使用逐层搜索递归遍历的方式。时间复杂度为O(n)。

1. **只比较同一层级，不跨级比较**
2. **标签不同，直接删除，不继续比较**
3. **标签名相同，key相同，就认为是相同节点，不继续深度比较**

React

React性能优化

思路

- 减少render执行次数
- 减少渲染的节点
- 降低渲染计算量
- VDOM
- 使用工具分析性能
- 使用不可突变数据结构，数组使用concat，对象使用Object.assign()
- 组件尽可能拆分
- 列表类组件优化
- bind函数优化
- 不滥用props
- reactDOMServer服务端渲染组件

React.memo

缓存组件

记忆组件渲染结果，提高组件性能

只检查props是否变化

做浅比较

第二个参数可传入自定义比较函数

areEqual方法和shouldComponentUpdate返回值相反

```
const App = React.memo(

        function myApp(props) {
        //使用props渲染
    }

        function areEqual(prevProps, nextProps) {
        //如果把prevProps传入render方法的返回结果和将nextProps传入render的返回结果一样，则返回true，否则返回false
    });
```

React.useMemo

缓存大量计算

useMemo 的第一个参数就是一个函数，这个函数返回的值会被缓存起来，同时这个值会作为 useMemo 的返回值，第二个参数是一个数组依赖，如果数组里面的值有变化，那么就会重新去执行第一个参数里面的函数，并将函数返回的值缓存起来并作为 useMemo 的返回值。

```
// 避免这样做
function Component(props) {

    const someProp = heavyCalculation(props.item);

    return  < AnotherComponent someProp = {
        someProp
    }
    />
    }

    // 只有 `[props.item](http://props.item)` 改变时someProp的值才会被重新计算
    function Component(props) {

        const someProp = useMemo(() => heavyCalculation(props.item), [props.item]);
        return  < AnotherComponent someProp = {
            someProp
        }
        />
        }
```

PureComponent

避免重复渲染

React.Component并未实现 shouldComponentUpdate()，而 React.PureComponent中以浅层对比 Prop 和 State 的方式来实现了该函数。

shouldComponentUpdate函数中做的是“浅层比较”。若是“深层比较”，那时某个特定组件的行为，需要我们自己编写。

父组件状态的每次更新，都会导致子组件的重新渲染，即使是传入相同props。但是这里的重新渲染不是说会更新DOM,而是每次都会调用diif算法来判断是否需要更新DOM。这对于大型组件例如组件树来说是非常消耗性能的。

shouldComponentUpdate生命周期来确保只有当组件props状态改变时才会重新渲染.

PureComponent会进行浅比较来判断组件是否应该重新渲染，对于传入的基本类型props，只要值相同，浅比较就会认为相同，对于传入的引用类型props，**浅比较只会认为传入的props是不是同一个引用**，如果不是，哪怕这两个对象中的内容完全一样，也会被认为是不同的props。

PureComponent，因为进行浅比较也会花费时间，这种优化更适用于大型的展示组件上。大型组件也可以拆分成多个小组件，并使用memo来包裹小组件，也可以提升性能。

- 确保数据类型是值类型
- 如果是引用类型，不应当有深层次的数据变化(解构)

避免使用内联对象

使用内联对象时，react会在每次渲染时重新创建对此对象的引用，这会导致接收此对象的组件将其视为不同的对象,因此，该组件对于prop的浅层比较始终返回false,导致组件一直重新渲染。

```
// Don't do this!
```

```
function Component(props) {
const aProp = { someProp: 'someValue' }
return <AnotherComponent style={{ margin: 0 }} aProp={aProp} />
}

// Do this instead :)
const styles = { margin: 0 };
function Component(props) {
const aProp = { someProp: 'someValue' }
return <AnotherComponent style={styles} {...aProp} />
}
```

避免使用匿名函数

```
// 避免这样做
function Component(props) {
return <AnotherComponent onChange={() => props.callback(props.id)} />
}
// 优化方法一
function Component(props) {
const handleChange = useCallback(() => props.callback(props.id), [props.id]);
return <AnotherComponent onChange={handleChange} />
}
// 优化方法二
class Component extends React.Component {
handleChange = () => {
this.props.callback(this.props.id)
}
```

```
render() {

return <AnotherComponent onChange={this.handleChange} />

}

}
```

组件懒加载

可以使用新的React.Lazy和React.Suspense轻松完成。

React.lazy

定义一个动态加载的组件，这可以直接缩减打包后 bundle 的体积，延迟加载在初次渲染时不需要渲染的组件

React.Suspense

悬挂 终止 暂停

配合渲染 lazy 组件，在等待加载 lazy组件时展示 loading 元素，不至于直接空白，提升用户体验；

```
<Suspense fallback={<PageLoading />}>{this.renderSettingDrawer()}</Suspense>

const SettingDrawer = React.lazy(() => import('@/components/SettingDrawer'));
```

用 CSS

而不是强制加/卸载组件

渲染成本很高，尤其是在需要更改DOM时。此操作可能非常消耗性能并可能导致延迟

```
// 避免对大型的组件频繁对加载和卸载

function Component(props) {

const [view, setView] = useState('view1');

return view === 'view1' ? <SomeComponent /> : <AnotherComponent />

}

// 使用该方式提升性能和速度

const visibleStyles = { opacity: 1 };

const hiddenStyles = { opacity: 0 };

function Component(props) {

const [view, setView] = useState('view1');

return (
```

```
<React.Fragment>
<SomeComponent style={view === 'view1' ? visibleStyles : hiddenStyles}>
<AnotherComponent style={view !== 'view1' ? visibleStyles : hiddenStyles}>
</React.Fragment>
)
}
```

React.Fragment

聚合子元素列表，不在DOM中添加额外标签

```
function Component() {
return (
<React.Fragment>
    <h1>Hello world!</h1>
    <h1>Hello there!</h1>
    <h1>Hello there again!</h1>
</React.Fragment>
)
}
```

key

- 保证key具有唯一性
- DIff算法根据key判断元素时新创建还是被移动的元素，从而减少不必要渲染

虽然key是一个prop，但是接受key的组件并不能读取到key的值，因为key和ref是React保留的两个特殊prop

合理使用Context

无需为每层组件手动添加 Props，通过provider接口在组件树间进行数据传递

原则

Context 中只定义被大多数组件所共用的属性

- 使用createContext创建一个上下文
- 设置provider并通过value接口传递state数据
- 局部组件从value接口中传递的数据对象中获取读写接口

toggle——切换

```
import { createContext } from 'react';

const LoginContext = createContext();

export default LoginContext;
```

虚拟列表

合理设计组件

简化Props

简化State

减少组件嵌套

React生命周期

React15

```
constructor() // 构造函数

componentWillReceiveProps() // 父组件状态属性更新触发

shouldComponentUpdate() // 组件更新时调用，在此可拦截更新

componentWillMount() // 初始化渲染时调用（挂载前调用）

componentWillUpdate() // 组件更新时调用

componentDidUpdate() // 组件更新后调用

componentDidMount() // 初始化渲染时调用（挂载后调用）

render() // 生成组件虚拟Dom

componentWillUnmount() // 组件卸载时调用
```

React16

```
constructor() // 构造函数

getDerivedStateFromProps() // 组件初始化和更新时调用

shouldComponentUpdate() // 组件更新时调用，在此可拦截更新

render() // 生成虚拟Dom
```

```
getSnapshotBeforeUpdate() // 组件更新时调用

componentDidMount() // 组件初始化时调用（挂载后调用）

componentDidUpdate(prevProps, prevState) // 组件更新后调用

componentWillUnmount() // 组件卸载时调用
```

constructor

实例过程中自动调用，方法内部通过super关键字获取父组件的props

通常初始化state或者this上挂载方法

16新增2个

static getDerivedStateFromProps(nextProps, prevState)

静态方法(纯函数)

执行时机：组件创建和更新阶段，不论是props变化还是state变化，都会调用

在每次render方法前调用，第一个参数为即将更新的props，第二个参数为上一个状态的state，比较props 和 state来加一些限制条件，防止无用的state更新

该方法返回一个新对象作为新的state或者返回null表示state状态不需要更新

getDerivedStateFromProps() contains the following legacy lifecycles:

componentWillMount

componentWillReceiveProps

componentWillUpdate

> React 16.4后对getDerivedStateFromProps做了微调。在>=16.4以后的版本中，组件任何的更新流程都会触发getDerivedStateFromProps，而在16.4以前，只有父组件的更新会触发该生命周期

getSnapshotBeforeUpdate(prevProps, prevState)

该周期函数在render后执行，执行之时DOM元素还没有被更新

该方法返回一个Snapshot值，作为componentDidUpdate第三个参数传入

```
getSnapshotBeforeUpdate(prevProps, prevState) {

console.log('#enter getSnapshotBeforeUpdate');

return 'foo';

}

componentDidUpdate(prevProps, prevState, snapshot) {

console.log('#enter componentDidUpdate snapshot = ', snapshot);

}
```

目的在于获取组件更新前的信息，比如组件的滚动位置之类的，在组件更新后可以根据这些信息恢复UI视觉上的状态

componentDidMount

组件挂载到ADOM节点后执行，在render之后执行，执行一些数据获取，事件监听等操作

shouldComponentDidMount

告知组件本身基于当前的props和state是否需要重新渲染组件，默认情况返回true。

执行时机：到新的props或者state时都会调用，通过返回true或者false告知组件更新与否

一般情况，不建议在该周期方法中进行深层比较，会影响效率

同时也不能调用setState，会导致无限循环

componentDidUpdate

执行时机：组件更新结束后触发

在该方法中，可以根据前后的props和state的变化做相应的操作，如获取数据，修改DOM样式等

componentWillUnmount

组件卸载前，清理一些注册或监听事件，或者取消订阅的网络请求

一旦一个组件实例被卸载，其不会被再次挂载，只可能被重新创建

准备废弃的生命周期

componentWillMount

风险很高，鸡肋

componentWillReceiveProps(nextProps)

一个API并非越复杂才越优秀。从根源帮助开发者避免不合理的编程方式和对生命周期的滥用

判断另个props是否相同，若不同将新的props更新到相应的state上

- 破坏state数据的单一数据源，导致组件状态不可预测

- 增加组件的重绘次数

componentWillUpdate(nextProps, nextState)

挡了Fiber架构的路

组件更新前，读取某个DOM元素状态，并在componentDidUpdate中处理

由getSnapshotBeforeUpdate(prevProps,prevState) 代替.在最终的render之前调用

> 为何要废弃它们?

在Fiber机制下，render阶段允许被暂停、终止和重启

导致render阶段的生命周期都可能重复执行。这几个方法常年被滥用，执行过程中存在风险。比如setState fetch发起请求 操作真实Dom等

这些操作完全可以转移到其他方法中做

即使没有开启异步渲染，Recat15中也可能导致一些严重的问题，比如componentWillReceiveProps和componentWillUpdate里滥用setState导致重复渲染

首先确保了Fiber机制下数据安全性，同时也确保生命周期方法的行为更纯粹

React Router

原理

客户端路由 实现原理

- 基于hash 路由，监听hashchange 事件，location.hash=xxxx 改变路由
- 基于H5的 history，通过history，pushState 和 replaceState 修改URL，能应用 history.go() 等 API，允许通过 自定义事件 触发实现

react-router实现原理：

- 基于 history 库实现，可 保存 浏览器 历史记录
- 维护列表，每次回收URL发生变化的回收，通过配置的路径，匹配对应的 Component 并 render

路由切换

- 组件 ，通过 比较 的path属性和当前地址的 pathname 实现，匹配成功则 render，否则渲染null

```
// when location = { pathname: '/about' }

 < Route path = '/about' component =
{
    About
}
/> / / renders < About /  >

 < Route path = '/contact' component =
{
```

```
    Contact
}
/> / / renders null

 < Route component =
{
    Always
}
/> / / renders < Always /  >
```

- +，对进行分组，遍历所有的子，渲染匹配的第一个元素

```
 < Switch >

 < Route exact path = "/" component =
{
    Home
}
/>

      <Route path="/about" component={About} />

<Route path=" / contact " component={Contact} />

</Switch>
```

- ，组件创建链接，会在HTML处 渲染锚()

```
 < Link to = "/" > Home <  / Link >

    // <a href='/'>Home</a>

    当 < NavLink > 的 to 属性与当前地址匹配时，可将其定义为活跃的

    // location = { pathname: '/react' }

     < NavLink to = "/react" activeClassName = "hurray" >

    React

     <  / NavLink >

    // <a href='/react' className='hurray'>React</a>
```

渲染强制导航

重定向

实现路由重定向

请求 /users/:id 被重定向去 '/users/profile/:id':

- 属性 from: string：匹配的将要被重定向路径
- 属性 to: string：重定向的 URL 字符串
- 属性 to: object：重定向的 location 对象
- 属性 push: bool：若为真，重定向将会把新地址加入到访问历史记录里，且无法回退到前面的页面

Link和a

都是链接，都是标签

是 react-router 中实现路由跳转的链接，一般配合 使用，跳转 只会触发相匹配的 对应页面内容更新，不会刷新all page

行为

1. 有onClick 就 onClick
2. click 时阻止a 默认事件
3. 根据to属性，使用histor 或 hash 跳转，跳转只是链接变了，没有all page 刷新，是普通的超链接

禁用后咋实现跳转?

手动赋值给标签 location 属性

```
let domArr = document.getElementsByTagName('a')

    [...domArr].forEach(item =>
    {

        item.addEventListener('click', function ()
        {

            location.href = this.href

        }
        )

    }
    )
```

获取URL参数

- get 方法

this.props.location.search 获取 URL 得到字符串，解析参数，或者自己封装方法获取参数

- 动态路由传值

如 path='/admin/:id'，this.props.match.params.id 获取 URL中动态部分路由 id值，或者使用hooks的 API获取

- query 或 state传值

组件 的 to属性可以传递对象 {pathname:'/admin',query:'111',state:'111'}，this.props.location.state或this.props.location.query 获取，一旦刷新页面 数据丢失

获取历史对象

- React>=16.8 ， 使用 React Router的Hooks

```
import
{
    useHistory
}
from "react-router-dom";

let history = useHistory();
```

- this.props.history

路由模式

- BrowserRouter

H5的history API控制路由跳转

- HashRouter

URl的hash 属性控制路由挑转

React组件通信

- 父组件向子组件传递
- 子组件向父组件传递
- 兄弟组件之间的通信
- 父组件向后代组件传递
- 非关系组件传递

父组件向子组件传递

React的数据流是单向的，这是最常见的方式，props

子组件向父组件传递

父组件向子组件传一个函数，然后通过这个函数的回调拿到子组件传递的值

兄弟组件之间的通信

父组件作为中间层实现数据互通

父组件向后代组件传递

最普通的事情，像全局数据一样。

使用context可共享数据，其他数据都能读取对应的数据。

非关系组件传递

组件间关系类型复杂，可以将数据进行一个全局资源管理，从而实现通信，例如redux dva

Hooks

React16.8之后，退出新功能：Hooks。这样可以在函数定义的组件中使用类组件的特性

Hooks好处

- 跨组件复用，轻量，改造成本小，不影响原来组件层级结构和嵌套地狱
- hook没生命周期，class有（hook可以使用钩子函数模仿生命周期）
- 状态与UI隔离，状态逻辑粒度更小，可以把业务逻辑抽离出来做自定义hook

hook缺点

- 状态不同步，容易产生闭包，需要使用useRef去记录

注意

- React函数组件中使用Hooks
- 避免循环/条件中调用hooks，保证调用顺序的稳定
- 不能在useEffect中使用useState，React会报错

hooks前，类组件边界强于函数组件

> UI=render(data)
> 或
> UI=func(date)

为啥useState使用数组

解构赋值！！

如果使用数组，我们可以对数组中元素命名，代码比较干净

如果使用对象，解构时必须要和useState内部实现返回对象同名，多次使用只能设置别名

```
// 第一次使用
const
{
    state,
    setState
} = useState(false);

// 第二次使用
const
{
    state: counter,
    setState: setCounter
} = useState(0)
```

解决问题

- 组件间逻辑复用
- 复杂组件理解
- 难以理解的class

使用限制

- 不再循环、条件和嵌套函数中调用(链表 实现Hook，可能导致 数组取值错位)
- 仅在函数组件中调用hook(因为没有 this)

useLayoutEffect和useEffect

- 共同点

都是处理副作用，包括 DOM改变，订阅设置，定时器操作

- 不同点

useEffect异步调用，按顺序执行，屏幕像素改变后执行，可能会闪烁

useLayoutEffect在所有DOM变更后同步调用，处理 DOM操作，样式调整；改变屏幕像素之前执行(会推迟页面显示的时间，先改变DOM后渲染)，不会闪烁，**总是比 useEffect先执行!**

useLayoutEffect 和 componentDidMount，componentDidUpdate 执行时机一样，在浏览器将所有变化渲染到屏幕之前执行

建议使用 useEffect！建议使用 useEffect！建议使用 useEffect！ 避免阻塞视觉更新

页面有异常就再替换为 useLayoutEffect

常见 Hooks

useState，useEffect，useContext，useReducer，useRef，useMemo，useCallback，useLayoutEffect，useImperativeHandle，useDebugValue

useMemo

当父组件中调用子组件时，父组件的state变化，会导致父组件更新，子组件即使没有变化，也会更新

函数式组件从头更新到尾，只要一处改变，所有模块都会刷新——没必要！

理想状态是 各个模块只进行 自己的更新，不相互影响

useMemo为了防止一种情况——父组件更新，无论是否最子组件操作，子组件都会更新

memo 结合了 PureComponent 纯组件 和 componentShouldUpdate 功能，会对传入的props进行一次对比，根据第二个函数返回值判断哪些props需要更新

useMemo和memo类似，都是 判断 是否满足当前限定条件决定是否执行 callback ，useMemo的第二个参数是一个数组，通过这个数组判断是否更新回调函数

好处

1. 减少不必要 循环 和 渲染
2. 减少 子组件 渲染次数
3. 避免不必要 的开销

useCallback

和useMemo 可以说是一模一样，唯一不同的是useMemo返回函数运行结果，而useCallback返回函数

这个函数是父组件传递子组件的 一个函数，防止无关 刷新，这个组件必须配合 memo，否则可能 还会降低 性能

useRef

获取当前元素的所有属性，返回一个可变的ref对象，且这个对象 只有 current 属性，可设置 initialValue

获取对应属性值

缓存数据

react-redux 源码中，hooks推出后，react-redux用大量的useMemo重做了 Provide 等核心模块，运用了useRef 缓存数据，所运用的useRef() 没有一个是绑定在dom元素上的，都是用于数据缓存

react-redux 利用 重新赋值，改变了缓存的数据，减少不必要更新，如果使用userState势必会re-render

总结

1. 一个优秀的hooks一定具备 useMemo useCallback 等api 优化
2. 制作自定义hooks遇到传递过来的值，优先考虑使用useRef，再考虑使用useState
3. 封装时，应该将存放的值放入useRef中

ref

ref --> reference：引用

React中 引用简写，是一个 属性，助于 存储 对React元素 或 组件的引用，引用由组件渲染配置函数返回

ref使用有三种方式：字符串(不推荐使用)，对象，函数

1、字符串 React16 之前用的最多的，如

```
<p ref="info">text</p>
```

2、函数格式，ref对应一个方法，该方法有一个参数，即，对应的节点实例

```
<p ref={ele=>this.info=ele}>test</p>
```

3、createRef方法，React16 提供的API，使用React.createRef()实现

使用场景

- input标签聚焦
- 希望直接使用DOM元素中的某个方法，或者直接使用自定义组件中的某一个方法
- ref作用于内置的HTML组件，得到的将是真实的DOM元素

- ref作用于类组件，得到的将是类的实例
- ref不能作用于函数组件，但是函数组件中可以传递ref

如何使用

1. 在 class 组件用 React.createRef() 来声明，例如

```
class MyComponent extends React.Component
{
    constructor(props)
    {

        super(props);

        this.myRef = React.createRef();

    }

    render()
    {

        return  < div ref =
        {
            this.myRef
        }
        />;

            }

        }
```

2. 在函数组件使用 React.forwardRef() 来暴露函数组件的 DOM 元素

```
const Input = forwardRef((props, ref) =>
    {

        return (

             < input
        {
            ...props
        }
            ref =
            {
                ref
            }
             > )

    }
    )

    class MyComponent extends React.Component
```

```
{

    constructor()
    {

        this.ref = React.createRef(null);

    }

    clickHandle = () =>
    {

        // 操作Input组件聚焦

        this.ref.current.focus();

    }

    render()
    {

        return (

             < div >

             < button onClick =
            {
                this.clickHandle
            }
             >

            点击

             <  / button >

             < Input ref =
            {
                this.ref
            }
             >

             <  / div > )

    }

}
```

render不能访问refs

因为 render 阶段 DOM还未生成，无法获取 DOM，DOM的获取需要在 pre-commit 和 commit 阶段

JSX


```
const ele=<h1>hello world</h1>
```

既不是字符串也不是HTML，是jsx，是JS的语法扩展，可以生成React元素

React认为渲染逻辑本质与其他UI逻辑内在耦合，将标签和逻辑共同存放在组件中，实现关注点分离

编译之后，JSX被转换为JS普通函数调用，取值后得到JS对象

允许在条件或循环语句中使用JSX，将其赋值给变量，传入JSX作为参数，以及从函数中返回JSX

防止注入攻击

1. 可以安全地在JSX中插入用户内容
2. React的DOM在渲染所有内容后，会进行转义，确保应用中永不会注入那些不是自己写的内容

JSX表示对象

Babel将JSX转译为一个名为React.createElement()函数调用

React读取这些React元素对象并使用它们构建DOM以保持随时更新

Fiber

React 15 ，渲染时 会递归对比 VDOM，找出需要变动的节点，再同步更新，一气呵成！！此过程中，React会占据浏览器资源，导致用户触发的事件得不到响应，导致 掉帧，用户觉得卡顿！！

将浏览器 渲染 布局 绘制 资源加载 事件响应 脚本执行 看成OS的process，通过一些调度策略合理分配 CPU 资源，提高浏览器 的用户响应速率，兼顾 执行效率

React16起，引入 Fiber 架构

让执行过程 变得 可中断！！

- 分批延时 操作 DOM，避免一次性操作大量 DOM 节点，用户体验更好
- 给浏览器一点喘息的机会，对代码进行 编译 优化 及热代码优化

Fiber也称协程或纤程，和线程并不一样，协程本身没有并发或者 并行能力，它只是一种控制流程的让出机制。让出 CPU 的执行权，让 CPU 能在这段时间执行其他的操作。

Redux

Redux名字的含义是Reducer+Flux，——视图层框架

context“上下文环境”，让一个树状组件上所有的组件都能访问一个共同的对象，context 由其父节点链上所有组件通过 getChildContext() 返回context对象组合而成，因此，通过context 可以访问其父组件链上context

创建一个特殊的react组件，它将是通用的一个context提供者，可以应用在任何一个应用中——Provider!

Provider只是把渲染工作完全交给子组件，只是提供Context包住最顶层组件，context覆盖了整个应用

Context=提供一个全局可访问的对象，但是全局对象应该避免，只要有一个地方做了修改，其他地方会受影响

context 只有对个别组件适用，不要滥用!

Redux的store封装得好，没有提供直接修改状态功能，克服了全局对象的缺点。且一个应用只有一个Store，store是context唯一需要，不算是滥用

工作原理

1. 用户(View)发出 action，使用dispatch
2. store 自动调用 reducer，传入参数：state 和收到的action，reducer返回新的state
3. state变化，store调用监听函数，更新 View

异步请求处理

借助 redux 的中间件 异步处理

主要有 redux-thunk，redux-saga(常用)

此处主要介绍 redux-saga

优点：

- 异步解耦，不会掺杂在 action 或 component 中，代码简洁度提高
- 异常处理，可使用 try/catch 捕获异常
- 功能强大，无需封装或简单封装 即可使用
- 灵活，可将saga 串行 起来，形成异步 flow
- 易测性

缺陷：

- 学习成本高

- 体积略大
- 功能过剩
- yield无法返回 TS 类型

@connect

连接React和Redux

1. connect通过context获取Provider中的state，通过store.getState()获取store tree上所有state
2. 将state和action通过 props 的方式传入组件内部，wrapWithConnect 返回一个Component对象connect，connect重新render 外部传入的元组件 wrapWithConnect ，将connect中传入的mapStateToProps ，mapDispatchToProps 与组件上原有的props 合并，通过属性的方式传给 WrappedComponent
3. 监听 state tree 变化。connect缓存了store tree中state的状态，根据state当前和变更前状态比较，确定是否setState触发connect及re-render

connect的工作

1. 把store上的状态转化为内层傻瓜组件的prop
2. 把内层傻瓜组件的用户动作 转化为派送给store的动作

@有时候使用Symbol代替字符串表示枚举值更合适，但是有的浏览器不支持

action构造函数返回一个action对象

Redux和Vuex

区别

- vuex改进了redux中的action和reducer，以mutations 变化函数取代 reducer，无需 switch，只需在mutations中修改对应state即可
- vuex无需订阅重新渲染，只需要生成新的state
- vuex数据流，view调用store，commit提交对应请求到store中对应mutations，store修改，vue检测state修改自动渲染

vuex弱化dispatch和reducer，更简易

共同点

- 单一数据流
- 可预测变化

都是对MVVM思想服务

中间件如何拿到store和action

redux 中间件本质是一个函数柯里化，

涉及 源码

有状态组件和无状态组件

有状态组件

特点

- 是类组件
- 有继承
- 有this
- 有生命周期
- 使用较多，易触发生命周期钩子函数
- 内部使用state，根据外部组件传入的props和自身state渲染

使用场景

- 需要使用状态的
- 需要状态操作组成的

总结

可维护自己的state，可以对组件做更多的控制

无状态组件

特点

- 不依赖自身state
- 可以是类组件或函数组件
- 可避免使用this
- 组件内部不维护state，props改变，组件re-render

使用场景

组件不需要管理state

优点

- 简化代码 专注render
- 组件不需要实例化，无生命周期
- 视图和数据解耦

缺点

- 无法使用ref
- 无生命周期
- 无法控制组件re-render

> 当一个组件不需要管理自身状态时，就是无状组件，应该优先设计为函数组件，比如定义的

受控组件和非受控组件

受控组件

表单状态变化，触发onChange事件，更新组件state

受控组件中，组件渲染出的状态和它的value或checked属性相对应，react通过这种方式消除了组件的局部状态，使组件变得可控

缺点

多个输入框需要获取到全部值时，需要每个都编写事件处理函数，代码变得臃肿

后来，出现了非受控组件

非受控组件

表单组件没有value props

可使用ref 从 DOM 中获取表单值，而不是编写事件处理函数

使用非受控组件可以减少代码量

合成事件

充当浏览器原生事件的跨浏览器包装器 的对象

将不同浏览器行为组合到一个 API中，确保事件显示一致的 属性

React 事件机制

```
 < div onClick =
{
    this.handleClick.bind(this)
}
> 点我 <  / div >
```

React不是将click事件绑定到了真实DOM上，而是在document处监听了所有的事件，当事件发生并且冒泡到document处的时候，React将事件内容封装并交由真正的处理函数运行。

这样可以 减少内存消耗，还能在组件挂在销毁时统一订阅和移除事件

冒泡到document上的事件也不是原生浏览器事件，而是由react自己实现的合成事件（SyntheticEvent）。

如果不想要事件冒泡的话应调用event.preventDefault()方法，而不是调用event.stopProppagation()方法

合成事件的目的如下：

- 合成事件抹平了浏览器间的兼容问题，这是一个跨浏览器原生事件包装器，赋予了跨浏览器开发的能力
- 对于原生浏览器事件来说，浏览器会给监听器创建一个事件对象。如果你有很多的事件监听，就需要分配很多的事件对象，造成高额内存分配问题。但是对于合成事件来说，有一个事件池专门来管理它们的创建和销毁，当事件需要被使用时，会从池子中复用对象，事件回调结束后，销毁事件对象上的属性，便于复用

setState 同步还是异步

API层面上，setState是普通的调用执行的函数，自然是同步API

此 同步 非 彼 同步

同步还是异步 指的是调用API后更新DOM是异步还是同步？——取决于 被调用的环境

- React能控制的范围调用，如 合成事件处理函数、生命周期函数，会批量更新，将状态合并后再进行DOM更新，为异步
- 原生JS控制的范围调用，如 原生事件处理函数，定时器回调函数，Ajax回调函数，此时setState被调用后立即更新DOM，为同步

所谓 异步 就是 批量更新，减少ADOM渲染次数，在React能控制范围内，一定是批量更新(为了性能着想)，先合并状态，再一次性 更新DOM

setState意味着一个完整的渲染流程，包括

> shouldComponentUpdate->componentWillUpdate->render->componentDidUpdate

setState 批量更新(异步) 就是为了避免 频繁 re-render，消耗性能非常恼火

批量更新 过程 和 事件循环 类似，来一个setState就将其加入 一个队列，待时机成熟， 将队列中 state 结果合并，最后只会对最新的state 进行一个更新流程

在 react17 中，setState 批量执行，但如果在 setTimeout、事件监听器等函数里，setState 会同步执行。可以在外面包一层 batchUpdates 函数，手动设置 excutionContext 切换成批量执行

react18 中所有的 setState 都是异步批量执行

dangerouslySetInnerHTML

是REact中innerHTML的替代品

```
<div dangerouslySetInnerHTML={{ __html: newsTexts }} />
```

React的HTML元素上的一个属性，它可能是危险的，因为我们容易收到XSS(跨站脚本攻击)——从第三方获取数据或用户提交内容时

React会识别HTML标签，然后渲染

HTML元素可能会执行脚本，当JS代码附加到HTML元素上

使用HTML净化工具DOMPurify检测HTML中潜在的恶意部分

PureComponent&Component

Component需要手动实现shouldComponentDidMount，而PureComponent通过浅对比默认实现了shouldComponentDidMount

浅比较，只比较第一层

PureComponent不仅影响本身，而且会影响子组件，所以其最佳情况是展示组件

VDom

一个对象，将真实的DOM树转换为JS对象树

```
 < button class = "myButton" >

     < span > this is button <  / span >

     <  / button >

    //转换

{

    type: 'button',

    props:
    {

        className: 'myButton',

        children: [
            {

                type: 'span',

                props:
                {

                    type: 'text'

                    children: 'this is button'

                }

            }
        ]

    }

}
```

将多次DOM修改的结果一次性更新到页面上，有效减少页面渲染次数，减少修改DOM的重排重绘次数，提高渲染性能

优越之处在于，提供更爽的、更高效的研发模式(函数式的UI编程模式)的同时，仍然保持还不错的性能

- 保证性能下限，不手动优化时，提供过得去的性能
- 跨平台

VDOM生成真实对象render方法

1.构建虚拟 DOM

2.通过虚拟 DOM 构建真正的 DOM

3.生成新的 VDOM

4.比较两棵 VDOM 异同

5.应用变更于 DOM

diff算法

1. ADOM映射为VDOM
2. VDOM变化后，根据差距生成patch，这个patch是结构化数据，包括增加 更新 和移除
3. 根据patch更新 ADOM

diff 从 树 组件 及 元素 3个层面进行复杂度的优化

1、忽略节点跨层级操作

若节点不存在了，则该节点及其子节点会被删除掉，不会进一步比较

2、若组件 是同一类型就进行树 对比，如果不是就直接放入 patch中，只要父组件类型不一致，就会re-render

3、同一层级子节点，通过key 进行列表对比

标记key，React 可以直接移动 DOM节点，降低消耗

VDom一定更快吗

VDOM是JS对象，模拟DOM节点，是通过特定算法计算出一次操作所带来的DOM变化。react和vue中都使用了VDOM

react中涉及到VDOM的代码主要分为三部分，第二步domDiff是核心：

- 把render中的JSX(或者createElement这个API)转化成VDOM
- 状态或属性改变后重新计算VDOM并生成补丁对象(domDiff)
- 通过补丁对象更新视图中的DOM节点

> 不一定更快

DOM操作是性能杀手，因为操作DOM会引起页面回流或重绘。相比，通过预先计算减少DOM的操作更划算

但是，“使用VDOM会更快”不一定适用所有场景。例如：一个页面就有一个按钮，点击一下，数字加一，肯定是直接操作DOM更快。使用VDOM无非白白增加计算量和代码量。即使是复杂情况，浏览器也会对我们的DOM操作进行优化，大部分浏览器会根据我们操作的时间和次数进行批量处理，所以直接操作DOM也未必很慢

那为什么现在的框架都使用VDOM？因为VDOM可以提高代码的性能下限，并极大优化大量操作DOM时产生的性能损耗。同时也保证了，即使在少数VDOM不太给力的场景下，性能也在我们接受的范围内