前端几大类题型

（主要包含基础知识难点、重点，由浅入深拓展）

●页面布局

例如，高度固定，实现三列布局，左右宽度 300px,中间自适应（或者其他布局，上下三列布局、左右两列布局、上下两列布局等，某个固定，其他自适应等问题）

方案：浮动布局、inline-block 布局、定位布局、flex 布局、表格布局、网格布局.....

由此考察基本功以及知识面广泛。可由此深入：

比如换位思考，对比几种布局的不同和优缺点，

内容过多后，或者去除高度，哪种布局会出问题，为什么会出问题，以及解决方案是啥，

如何实现一个高度改变，其他自适应高度，或者一个高度改变，其他高度不受影响。

比如浮动，解决浮动的方法有哪些，为什么会出现浮动的问题，浮动的原理等。

表格布局或 css 的 display:table-cell 或者网格布局等 css 代码掌握多少.....

参考网站，某些动画效果怎么实现？css3 动画掌握如何？

●css 盒模型

基本概念：标准盒模型（w3c 盒模型）和 IE 盒模型

标准盒模型和 IE 盒模型区别

标准盒模型：width+padding+border+margin

width = content;

IE 盒模型：width+margin;

width = content+padding+border;

Css 如何设置两种盒模型

display:content-box; 标准盒模型

display:border-box; IE 盒模型

Js 如何设置获取盒模型的宽高（此内容一块复习下 js 获取 dom 元素宽高左右距离等属性）

dom.style.width/height //只能用于行内样式的获取，其他样式获取不到

dom.currentStyle.width/height ; //渲染后的宽高，只有 ie 支持，兼容性差

window.getComputedStyle(dom).width/height; //同上面一样，但是兼容性强，兼容 ie、火狐、谷歌

dom.getBoundingClientRect().width/height; //此方法拿到四个属性，left,top,width,height 计算绝对视图的位置，当前距离视图（浏览器）的左右间距，以及 dom 的宽高

扩充：js clientHeight、offsetHeight、scrollHeight 等以及宽度的区别

实例题（根据盒模型解释边距重叠）

常见的边距重叠问题：

第一个子元素设置 margin-top;上边距会跑到父元素起作用，

兄弟元素，上面元素设置下边距 30px;下面元素设置上边距 10px；两个兄弟元素真正的边距为 30px，取最大值为两者边距

空元素：设置空元素上下边距，取最大值作为一个边距

由边距重叠问题引出 css BFC（边距重叠解问题决方案）或者 IFC

BFC 基本概念

“块级格式化上下文（Block Formatting Contexts）”。它是一个独立的渲染区域，只有 Block-level box（块级盒）参与， 它规定了内部的 Block-level Box 如何布局，并且与这个区域外部毫不相干

BFC 原理（布局、渲染规则）

Box 垂直方向的兄弟元素 margin 边距会发生重叠。

BFC 的区域不会与浮动元素 box 重叠（可用于清除浮动）。

计算 BFC 的高度时，浮动元素也参与计算。

BFC 就是页面上的一个独立的容器，容器里面的子元素不会影响到外面的元素，外面的元素也不会影响里面的元素

怎么创建 BFC

float 的值不是 none。（给重叠元素添加浮动，注意浮动的影响）

position 的值不是 static 或者 relative。（添加定位）

display 的值是 inline-block、table-cell、flex、table-caption 或者 inline-flex（添加 display 属性）

overflow 的值不是 visible。 auto，hidden 都可以（为发生重叠的元素添加一个父元素，给父元素设置 overflow,或者添加 border）

BFC 使用场景哪些

结合上个问题中怎么创建 bfc

IFC

IFC 基本概念

“内联格式化上下文（Inline Formatting Contexts）”。 IFC 的 line box（线框）高度由其包含行内元素中最高的实际高度计算而来（不受到竖直方向的 padding/margin 影响)。

形成条件

块级元素中仅包含内联级别元素

形成条件非常简单，需要注意的是当 IFC 中有块级元素插入时，会产生两个匿名块将父元素分割开来，产生两个 IFC，这里不做过多介绍。

IFC 特性

IFC 中的 line box 一般左右都贴紧整个 IFC,但是会因为 float 元素而扰乱。float 元素会位于 IFC 与与 line box 之间，使得 line box 宽度缩短。

IFC 中时不可能有块级元素的，当插入块级元素时（如 p 中插入 div）会产生两个匿名块与 div 分隔开，即产生两个 IFC，每个 IFC 对外表现为块级元素，与 div 垂直排列。

IFC 布局规则

子元素水平方向横向排列，并且垂直方向起点为元素顶部。

子元素只会计算横向样式空间,【padding、border、margin】,垂直方向样式空间不会被计算,【padding、border、margin】。

在垂直方向上，子元素会以不同形式来对齐（vertical-align）

能把在一行上的框都完全包含进去的一个矩形区域，被称为该行的行框（line box）。行框的宽度是由包含块（containing box）和与其中的浮动来决定。

IFC 中的“line box”一般左右边贴紧其包含块，但 float 元素会优先排列。

IFC 中的“line box”高度由 CSS 行高计算规则来确定，同个 IFC 下的多个 line box 高度可能会不同。

当 inline-level boxes 的总宽度少于包含它们的 line box 时，其水平渲染规则由 text-align 属性值来决定。

当一个“inline box”超过父元素的宽度时，它会被分割成多个 boxes，这些 oxes 分布在多个“line box”中。如果子元素未设置强制换行的情况下，“inline box”将不可被分割，将会溢出父元素。

举例

很多时候，上下间距不生效可以使用 IFC 来解释。左右 margin 撑开，上下 margin 并未撑开，符合 IFC 规范，只计算横向样式空间，不计算纵向样式空间。

```
.warp { border: 1px solid red; display: inline-block; }
.text { margin: 20px; background: green; }
<div class="warp">
    <span class="text">文本一</span>
    <span class="text">文本二</span>
</div>
```

多个元素水平居中

```
.warp { border: 1px solid red; width: 200px; text-align: center; }
.text { background: green; }
<div class="warp">
    <span class="text">文本一</span>
    <span class="text">文本二</span>
</div>
```

float 元素优先排列

```
.warp { border: 1px solid red; width: 400px; }
.text { background: green; }
.f-l { float: left; }
<div class="warp">
    <span class="text">这是文本 1</span>
    <span class="text">这是文本 2</span>
    <span class="text f-l">这是文本 3</span>
    <span class="text">这是文本 4</span>
</div>
```

IFC 常用应用

水平居中：当一个块要在环境中水平居中时，设置其为 inline-block 则会在外层产生 IFC，通过 text-align 则可以使其水平居中。

垂直居中：创建一个 IFC，用其中一个元素撑开父元素的高度，然后设置其 vertical-align:middle，其他行内元素则可以在此父元素下垂直居中。

扩充：css3 新增两种：

GFC（"网格布局格式化上下文 GridLayout Formatting Contexts"）

FFC（"自适应格式化上下文 Flex Formatting Contexts"）

●DOM 事件

基本概念：dom 事件的级别

Dom0：element.onclick = function(){};

Dom2：element.addEventListener（'click'，function(){},false/true);(第三个参数 true 为捕获阶段执行，false 为冒泡阶段执行)

Dom3：element.addEventListener（'keyup'，function(){},false/true);（dom3 和 dom2 定义类型一致，只不过增加了鼠标键盘事件等等新东西）

（dom1 制定时，没有涉及制定事件有关的东西，所以没有，跳过，但是不代表 dom1 标准不存在）

Dom 事件模型

事件模型即事件捕获阶段、事件目标阶段、事件冒泡阶段

Dom 事件流

事件流，即对上述 dom 事件模型的具体描述，说白了就是浏览器当前页面与用户交互的过程。

举个例子：

用户点了鼠标左键，左键怎么传到页面上，又是怎么响应的。一个完整的事件流分三个阶段，即捕获阶段，目标阶段，冒泡阶段。捕获阶段即由上向下执行，父元素向子元素一直到目标元素，目标阶段即获取到触发事件的目标元素，冒泡阶段即目标元素由下向上，子元素向父元素执行。

注：事件的捕获阶段是从 window 开始触发的，由 window 向 document，再向 HTML，body 一直到目标元素

Js 获取 html 标签为 document. documentElement;

Event 对象的常见应用

键盘事件，根据键盘码判断点击的哪个键盘或者鼠标

event.preventDefault();组织默认行为，如表单提交，超链接跳转

event.stopPropagation();阻止冒泡

event.currentTarget();获取当前绑定事件的元素，即事件在哪个 dom 元素上的元素

event.target();获取真正触发的元素，比如真正点击了的 dom 元素，非事件所绑定的元素

event.stopImmediatePropagation();将事件就地停止,在当前事件之后注册的其他事件,都不会执行。（举例，如果一个按钮绑定了多个 click 事件，默认会多个依次执行。假设我想按顺序触发事件，让一个事件执行后，后面的不在执行，那么给第一个事件后面添加此方法即可阻止后面的事件触发。）

事件代理/事件委托（不同资料不同叫法）

用于解决事件处理程序过多问题（减少占用内存，性能优化）

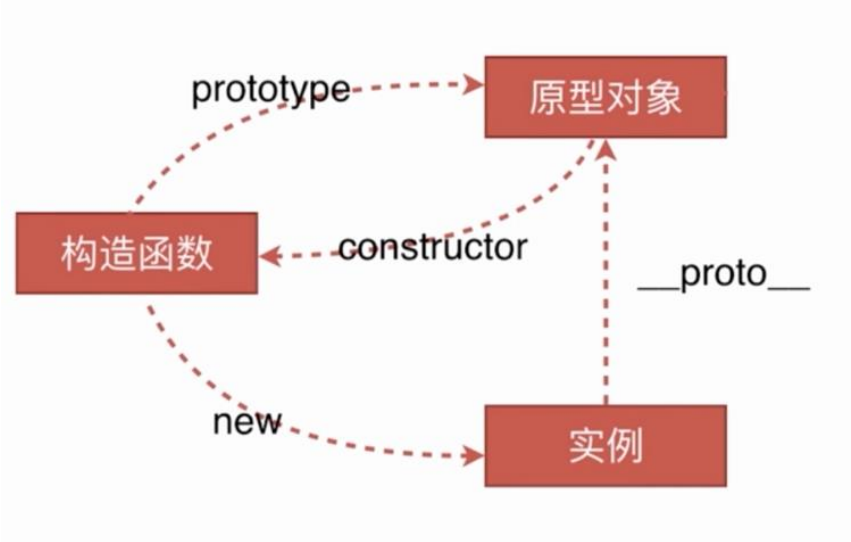
子元素的事件添加到父元素，如点击 li，当多有多个 li 时，如果给没个 li 添加点击事件，把点击事件放到父元素 ul，从而触发冒泡执行点击事件，可以通过 event.currentTarget()或者 event.target()判断哪个点击。本来是获得多个 li，没个 li 添加点击事件，改为获得一个 li 添加一个事件，减少 dom 获取，减少内存占用，性能优化，

自定义事件

```
var eve = new event( 'custom' );
dom.addEventListener( 'custom' ,function(){
    ........
});
dom.dispatchEvent(eve);
```

●原型（链）

原型对象、构造函数、实例关系：



例子：

```
function Person(){
    this.name  =  'a' ;
}
var p = new Person();
```

上面创建了一个构造函数 Person();并且创建了一个实例 p；构造函数可以通过 new 运算符创建一个实例，构造函数就是一个函数，每个函数默认都有一个 prototype 属性指向了一个对象，这个对象就是函数的原型对象，函数的原型对象有一个 constructor 构造器，构造器指向当前原型对象的函数。实例有一个__proto__属性，指向创建自己函数的原型对象。再者，由于函数也是一个对象，所以函数也有一个__proto__属性，但是，函数的__proto__属性指向了 Function 的原型对象。由此可知，函数是 Function 构造函数的一个实例。

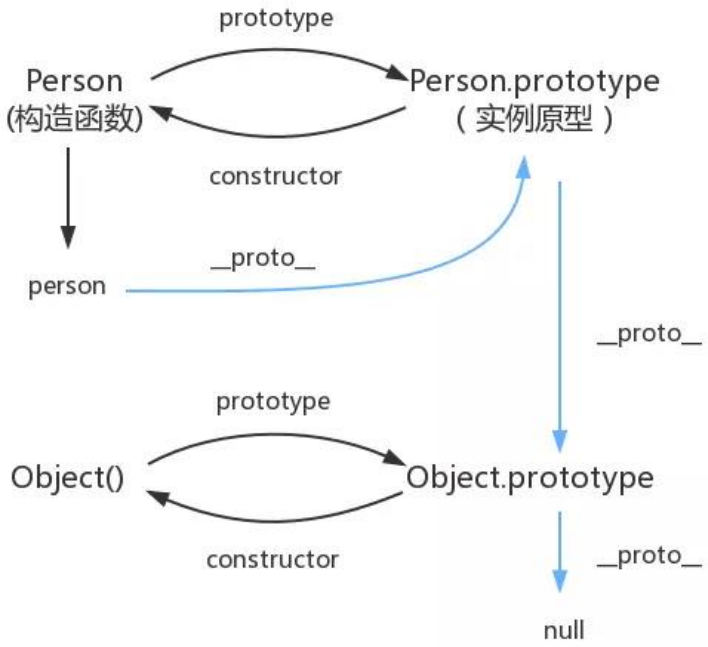
其中

Person.prototype.constructor === Person 返回 true；都是指向了同一个函数

p.__proto__ === Person.prototype;也返回 true；都指向了同一个原型对象

Person.__proto__===Function.prototype;返回 true。函数的__proto__指向了 Function 的原型对象

原型链



简单回顾一下构造函数、原型和实例的关系：每个构造函数都有一个原型对象，原型对象都包含一个指向构造函数的指针，而实例都包含一个指向原型对象的内部指针。那么，假如我们让原型对象等于另一个类型的实例，结果会怎么样呢？显然，此时的原型对象将包含一个指向另一个原型的指针，相应地，另一个原型中也包含着一个指向另一个构造函数的指针。假如另一个原型又是另一个类型的实例，那么上述关系依然成立，如此层层递进，就构成了实例与原型的链条。这就是所谓原型链的基本概念。(来自 js 高级程序设计 3)

简单讲，原型链就是一个原型对象访问另一个原型对象，另一个原型对象再去访问其他原型对象；最终实现多个原型对象之间的相互访问，串成了原型对象之间相互访问的链条，组成了原型对象与原型对象之间的链接，构成了原型链。

上述例子中，Person.prototype 是 Person 的原型对象，由于原型对象也是一个对象，所以原型对象默认也有一个__proto__属性，这个属性指向谁？Object 的原型。所有函数顶层都继承自 Object，Object 也是个构造函数，所以有了以下结果：

Person.prototype.__proto__ === Object.prototype;返回 true

而 Object 的原型对象.__proto__最终访问的为 Null。就此终止。

原型链的概念，构建了 js 继承的实现原理。

（原型链继承：其实就是通过原型对象与另一个函数的构造函数相互关联，实现这个函数的原型对象可以访问另一个构造函数，访问了另一个构造函数，另一个构造函数通过自身__proto__访问了自己的原型对象，所以两个函数之间的原型对象就相互关联了，即我的原型对象访问了你的构造函数，你的构造函数访问了你的原型对象，通过连接，也就是所构成的原型链，我的原型对象就访问了你的原型对象。从而实现了继承。）

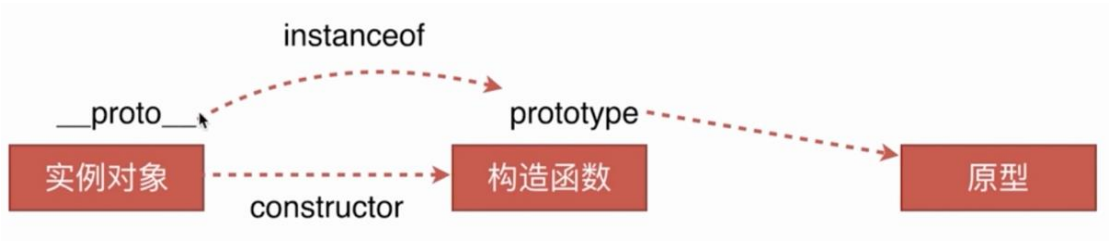
所以有了以下结论：

由此得出，只要在同一条原型链上，所有的实例 instanceof 此原型链上的构造函数都返回 true，所以如果 a instanceof B 反回了 true，a 不一定就是 B 创建的一个实例，可能同属于一个原型链。

那如何去判断当前是否是某个构造函数创建的实例呢?

p.__proto__.constructor === Person 即可，用当前实例指向的原型对象指向的构造函数与创建实例的构造函数进行判断，看是否为同一个构造函数即可。(如果 p 是 Person 创建的实例，则 p 的__proto__会指向 Person 的原型对象，同样 Person 的原型对象中的 constructor 则指向了自己的函数，通过此判断来判断 p 是否是 Person 创建的实例。)

instanceof 原理



instanceof 其实比较的是实例的__proto__和构造函数的 prototype 是否是引用的同一个地址，这里都引用了同一个原型对象，原型链也符合这个特点，所以导致上述例子的
p instanceof Person 返回 true，由于原型链作用，顶层都是 Object 对象，所以
p instanceof Object 也返回 true。

new 运算符原理

找资料慢慢研究，有点抽象

●面向对象、继承、闭包

面向对象

创建对象的几种方式：

new 操作符创建对象
字面量创建对象
Object.create();
工厂模式
构造函数模式
原型模式
组合使用构造函数模式和原型模式
动态原型模式
寄生构造函数
稳妥构造函数

继承

继承的几种方式

原型链继承
借用构造函数继承
组合继承（原型链与构造函数结合）
原型式继承
寄生式继承
寄生组合式继承
（Es6 class 继承）

闭包

概念

闭包是指有权访问另一个函数作用域中变量的函数，创建闭包的最常见的方式就是在一个函数内创建另一个函数，通过另一个函数访问这个函数的局部变量,利用闭包可以突破作用链域，将函数内部的变量和方法传递到外部。

闭包的特性：

封闭性：外界无法访问闭包内部的数据，如果在闭包内声明变量，外界是无法访问的，除非闭包主动向外界提供访问接口；

持久性：一般的函数，调用完毕之后，系统自动注销函数，而对于闭包来说，在外部函数被调用之后，闭包结

构依然保存在。

闭包的影响：

使用闭包会占有内存资源，过多的使用闭包会导致内存溢出等。可以用闭包名=null 来释放闭包

闭包常见面试题

1、 点击 li 实现打印当前的 li 索引以及每隔三秒打印 li 内容

```
<ul id="ul">
    <li>内容 1</li>
    <li>内容 2</li>
    <li>内容 3</li>
    <li>内容 4</li>
</ul>

<script type="text/javascript">
var lis = document.querySelectorAll('#ul li');
for(var i=0; i<lis.length; i++){
    (function(i){
        lis[i].onclick=function(){
            console.log(i)
        }
    })(i)
}

var lis = document.querySelectorAll('#ul li');
for(var i=0; i<lis.length; i++){
    (function(i){
        setTimeout(function(){
            console.log(lis[i].innerHTML)
        },3000);
    })(i)
}

</script>
```

●http 协议（考察理论居多）

http 协议概念

http 是一个简单的请求-响应协议，它通常运行在 TCP 之上。它指定了客户端可能发送给服务器什么样的消息以及得到什么样的响应。请求和响应消息的头以 ASCII 码形式给出；而消息内容则具有一个类似 MIME 的格式。这个简单模型是早期 Web 成功的有功之臣，因为它使得开发和部署是那么的直截了当。

https

HTTPS （全称：Hyper Text Transfer Protocol over SecureSocket Layer），是以安全为目标的 HTTP 通道，在 HTTP 的基础上通过传输加密和身份认证保证了传输过程的安全性 [1] 。HTTPS 在 HTTP 的基础下加入 SSL 层，HTTPS 的安全基础是 SSL，因此加密的详细内容就需要 SSL。 HTTPS 存在不同于 HTTP 的默认端口及一个加密/身份验证层（在 HTTP 与 TCP 之间）。这个系统提供了身份验证与加密通讯方法。它被广泛用于万维网上安全敏感的通讯，例如交易支付等方面

http 协议的主要特点

简单快速

每个资源，也就是 URI 都是固定的，比如一张图片和一个页面地址，即统一资源符，这是固定的，处理也是比较简单的，想访问某个资源，输入 URI 就可以了。

灵活

通过一个 http 协议完成不同数据类型传输

无连接（重点记住）

连接一次就会断掉，不会保持连接

无状态（重点记住）

客户端和服务端，可以认为是两种身份，比如请求一张图，第一次请求完后，第二次再请求，服务端无法区分这两次连接的身份。

http 报文的主要组成部分

（http 就是建立在 tcp 的应用上的）

1、**请求报文：**

请求行

包含 http 方法、页面地址、http 协议、http 协议版本

请求头

是一些 key、value 值，告诉服务端我要什么内容，注意什么类型

空行

告诉服务端下一个不是请求头部分了，当做请求体解析

请求体

是报文体，它将一个页面表单中的组件值通过 param1=value1&param2=value2 的键值对形式编码成一个格式化串，它承载多个请求参数的数据。不但报文体可以传递请求参数，请求 URL 也可以通过类似于“/chapter15/user.html? param1=value1&param2=value2”的方式传递请求参数。

2、**响应报文：**

状态行

http 协议、http 协议版本、响应状态码、

响应头

同请求头类似，也是一些 key、value 值，响应的内容，数据类型等

空行

响应体

响应的数据。

http 方法

get：获取资源

post：传输资源（向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST 请求可能会导致新的资源的建立和/或已有资源的修改。）

put：更新资源

delete：删除资源

head：获取报文头。（类似于 GET 请求，只不过返回的响应中没有具体的内容）

post 和 get 的区别（区别很多，记住三四个就可以了，下面写的也不一定全）

get 请求的 URL 中的参数长度是有限的，post 没有限制（get 请求的 URL 参数基本大概是 2kb，浏览器不同，限制也不一样。Get 请求的 URL 参数如果过长，会被浏览器截断，服务端得不到正确信息，请求报错）（记住，很重要）

get 比 post 更不安全，参数直接暴露在了 URL 上，不能传递敏感信息。Post 相对安全，参数放在请求体中。（记住）

get 请求会被浏览器主动缓存，post 不会，需手动设置（记住）

get 请求参数会被完整的保留在浏览器历史记录，post 参数不会保留（记住）

get 在浏览器回退是无害的，post 会再次提交请求（记住）

get 之接收 ASCII 字符，post 没有限制

get 产生的 url 地址浏览器可以收藏，post 不能收藏

get 请求只支持 URL 编码，post 支持多种编码方式

http 状态码（具体的状态码自己百度吧）

1xx：指示信息——表示请求已接收，继续处理

2xx：成功——表示请求成功接收

200：OK，客户端请求成功

3xx：重定向——完成请求必须进行更进一步操作

301：永久重定向。请求的资源已被永久的移动到新 URI，返回信息会包括新的 URI，浏览器会自动定向到新 URI。今后任何新的请求都应使用新的 URI 代替

302：临时重定向。与 301 类似。但资源只是临时被移动。客户端应继续使用原有 URI

4xx：客户端错误——请求有语法错误或者请求无法实现

400：一般是参数错误，比如没用使用 json 类型，或者参数个数，数据类型不对等。

401：一般是没有权限，身份认证过期等，比如没有缺少 token

403：资源禁止访问

404：请求资源地址不存在

405：大部分是请求方法不对，比如 get 请求用成了 post 请求等。

415：一般是参数类型错误，contentType: "application/json;charset=UTF-8",可能请求类型与设置的不一致

5xx：服务器错误——服务器未实现合法请求

500：服务器错误，无法完成请求

503：由于超载或系统维护，服务器暂时无法处理客户端的请求。一段时间后可能恢复正常。

缓存

缓存分类（详细自己查资料）

以下六个都是 http 协议头

1、 强缓存（不请求，直接拿过来就用）

Expires　Expires:Thu,21 Jan 2017 23:39:02 GMT（key,value 值，日期为绝对时间）

Cache-Control　Cache-Control:max-age=3600（key,value 值，相对时间，3600 单位秒）

两个缓存都下发，以后者为准

2、 协商缓存（浏览器发现本地有副本（也就是缓存），不确定用不用，问下服务器用不用，是不是过期了）

Last-Modified（上次修改时间） 配合 If-Modified-Since 或 if-Unmodified-Sindce 使用

Last-Modified：Web, 26 Jan 2017:35:11 GMT

Etag（数据签名） 配合 If-Match 或 If-None-Match 使用

通过对比资源的签名判断是否需要缓存。

什么是持久链接（keep-alive/persistent）

首先 http 持久连接是 http1.1 才支持，1.0 不支持。

设置长连接，在 http 头部设置 connection:keep-alive

http 协议采用“请求-应答”模式，当时用普通模式即非持久连接模式时，每个请求/应答客户和服务器都要新建一个连接，完成之后立即断开连接（http 协议为无连接的协议）。

当时用持久化连接模式（又称持久连接、连接重用）时，keep-alive 功能是客户端到服务器端的连接持续有效，当出现对服务器的后继请求时，keep-alive 功能避免了建立或者重新建立连接

HTTP/1.1 允许 HTTP 设备在事务处理结束之后将 TCP 连接保持在打开状态，以便为未来的 HTTP 请求重用现存的连接。在事务处理结束后仍然保持在打开状态的 TCP 连接被称为持久连接。非持久连接会在每个事务结束之后关闭。持久连接会在不同事务之间保持打开状态，直到客户端或服务器决定将其关闭为止。

持久连接降低时延和连接建立的开销，将连接保持在已调谐状态，而且减少了打开连接的潜在数量。

持久连接与并行连接配合使用可能是最高效的方式。持久连接有两种类型：比较老的 HTTP/1.0+ "keep-alive" 连接，以及现代的 HTTP/1.1 "persistent" 连接。

（持久连接可以概括为一个连接请求多个资源）

什么是管线化

在使用持久连接的情况下，某个连接上消息的传递类似于：

请求 1->响应 1->请求 2->响应 2->请求 3->响应 3

（通信一次，资源传输完也不要断开。）

管线化，某个连接上的消息变成了类似这样，

请求 1->请求 2->请求 3->响应 1->响应 2->响应 3

（通道是持久建立的，不是请求一次响应一次了，把请求打包，一次发送，响应打包，一次全部响应）

管线化特点：（了解即可）

管线化机制通过持久连接完成，仅 http1.1 支持。

只有 get 和 head 请求可以管线化，post 有所限制。

管线化不会影响响应到来的顺序

除此链接时，不应启动管线机制，因为对方服务器不一定支持 http/1.1 协议。

http1.1 要求服务器端支持管线化，但并不要求服务器端也对响应进行管线化处理，只是要求对 于 管 线化的请求不失败即可。

由于上面提到的服务器端问题，开启管线化很可能并不会带来大幅度的性能提升，而且很多服 务 器 端和代理程序对管线化的支持并不好，因此现代浏览器如 chrome 和 firefox 默认并未开启管线 化支持。

管线化实现原理等深究问题，自己研究吧（一般了解即可）

●通信类

什么是同源策略及限制

同源策略限制从一个源加载的文档或脚本如何与来自另一个源的资源进行交互。这是一个用于隔离潜在恶意文件的关键的安全机制。

解释：

源：协议+域名+端口号组成，这三者中有一个不一样，就是不同源

限制就是不是一个源的文档无法操作另一个源的文档，有几下几个限制：

Cookie、localstorage、indexDB 无法读取

DOM 不能获得，

Ajax 请求不能发送

前后端如何通信

Ajax 只能同源
WebSocket 不受同源限制
CORS 支持同源和跨域，是一个新的通信标准

如何创建 ajax

XMLHttpRequest 对象的工作流程
兼容性处理
事件的触发条件
事件的触发顺序

```
var xhr = XMLHttpRequest ? new  XMLHttpRequest : new  window.
          ActiveXObject( 'Microsoft.XMLHTTP' );
xhr.open( 'get/post' ,' url  ' );
xhr.send();
xhr.onreadystatechange = function(){
    if(xhr.readyState == 4 && xhr.status == 200){
        console.log(xhr.responseText);
    }
}
```

跨域的几种方式

JSONP（重点记忆）

原理，怎么实现的

利用 script 标签异步加载实现的，script 标签的 src 可以访问不同源地址，客户端通过 script 标签请求服务端，给服务端传递一个回调函数的名字，服务端通过路径获取地址的回调参数，通过一系列操作，请求数据，最后得到数据，把得到的数据，通过最后返回一个可执行的 js 脚本的方式传入获取客户端传来的函数中执行，把数据返回给客户端。

模拟客户端：

```
<script type="text/javascript">
var funcName = 'jsonpfunc';//随机生成函数名，可动态创建函数名，防止重复
window[funcName] = function(data){
    console.log(data);
}
</script>
<!-- 下面的 script 标签请求的为服务端地址，无论请求什么地址，只要是返回一个可执行的 js 脚本  即可，这里用 js 模拟-->
<!-- 通过路径传参的形式把自己定义的回调函数名字传给服务端 -->
<script type="text/javascript" src="./data.js?callback=funcName"></script>
```

模拟服务端：

```
//.....服务端执行某些操作获得 data 数据
 var data = {
      "name":"xiaoming",
 }
 /*
    以上为服务端操作的数据，通过路径获取到客户端传递的回调函数，
    最后通过执行获取到的函数，把数据传入，返回一个可之乡的 js 代码，从而把数据返回给客户端
 */
 jsonpfunc(data);
```

Hash (URL 地址中#后面的东西，hash 的变动页面不会刷新，serach：URL 中？后面的叫 search
search 的改变会刷新页面，所以不能做跨域通信)

获取 URL 地址，在 URL 后追加#+传送的数据，数据先转换为字符串，然后通过 hash 改变监听函 数 获取 hash 的值，根据需要做处理。

例子：

```
var url = document.getElementsByTagName('iframe').src;
url = url+'#'+data;
window.onhashchange = function(){
    var hashdata = window.location.hash;//此处得到的是 hash 的所有内容，#后的东西，
                                          需要啥自己处理
}
```

postMessage（重点记忆。html5 新增加的,通过调用提供 api 即可，只能两个窗口之间跨域，不能和服务器跨域）

两个窗口能通信的前提是，一个窗口以 iframe 的形式存在于另一个窗口，或者一个窗口是从另一个窗口通过 window.open()或者超链接的形式打开的（同样可以用 window.opener 获取源窗口）

例子：两个窗口，窗口 A(http:A.com)，B 窗口（http:B.com）A 窗口向 B 窗口发送消息

Awindow:A 窗口的 window 对象　　Bwindow:B 窗口的 window 对象

在 A 窗口向 B 窗口发送信息，向哪个窗口发送信息，选中哪个窗口进行 postMessage，发送数据推荐字符串格式

```
//在 A 窗口向 B 窗口发送数据:
Bwindow.postMessage('data','http://B.com');
//在 B 窗口监听
Window.addEventListener('message',function(event){//接收 message 事件
    event.origin;//判断发送者的源，这里是 http:A.com
    event.source;//Awindow 的引用，Awindow
    event.data;//拿到发送的数据
},false);
```

WebSocket

```
var ws = new WebSocket('wss://服务器地址');ws 和 wss 加密和非加密
//发送消息
ws.onopen = function(evt){
    ws.send('hellow websockets');
}
//接收消息
ws.onmessage = function(evt){
    evt.data;拿到响应数据
```

```
        ws.close();//关闭通信
    }
    //下面是监听关闭的
    ws.onclose = function(evt){
        console.log('连接关闭');
    }
```

CORS（全称是"跨域资源共享"（Cross-origin resource sharing）。可以理解为支持跨域通信的 ajax，他在 http 头上加个 origin，允许跨域通信）

可以用新型的 api fetch 实现，fecth 就是用 CORS 通信的，在第二个参数的对象中添加配置 就行了，怎么配置查资料吧，推荐资料：

（http://www.ruanyifeng.com/blog/2016/04/cors.html）

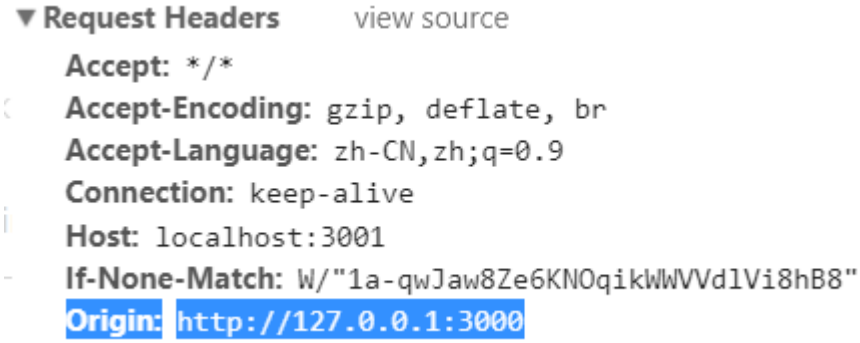
```
    fetch('url',{
        method:'get',
    }).then(function(response){

    }).catch(function(err){
        //错误信息
    })
```

cors 为什么支持跨域?

浏览器会拦截 ajax 请求，如果浏览器觉得这个请求是跨域的，他会在 http 请求前面添加 origin，同样响应头中也会有响应字段 Access-Control-Allow-Origin

简单请求：

对于简单请求，浏览器会直接发起跨域请求，会自动在 HTTP 头部加入 Origin 字段，如下：



跨域请求的服务器会根据这个 Origin 进行判断是否同意该域名的跨域请求。

如果服务器不同意，服务器会回应一个 HTTP（状态为 200），浏览器会在返回的 HTTP 头部检查 Access-Control-Allow-Origin 字段，如果没有该字段，就会自动报错。

如果服务器同意，浏览器的 HTTP 头部字段可能会多出以下字段：

Access-Control-Allow-Origin

Access-Control-Allow-Credentials

Access-Control-Expose-Header

字段解释，上述的三个头部字段都是在服务器设置:

Access-Control-Allow-Origin: 必写，请求域的 domin 或者（*） ,*代表着允许所有域名的跨域请求，

Access-Control-Allow-Credentials: 可写，该值是一个布尔值，表示是否允许发送 Cookie,默认情况下，Cookie 不包括 CORS 请求中。

Access-Control-Expose-Header：在 CORS 请求时浏览器的 XMLHttpRequest 对象的 getResponseHeader() 方法只可以拿到六个基本字段，如果还想拿到其他字段就必须在 Access-Control-Expose-Header 中指定。

非简单请求：

当发出的的请求为非简单的请求时，比如请求方法是 PUT 或者 DELETE,或者 Content-Type 字段是 application/json。此时浏览器会进行一次预检。

简述：在非简单请求的 CORS 中，在正式进行通信前，浏览器会增加 HTTP 请求，称为预检。

预检内容：浏览器会向服务器询问，当前网页的域名是否在许可名单中，以及可以使用哪些字段、哪些方法，只有得到肯定的回复，浏览器才可以进行 Ajax 通信，否则报错。

如果非简单请求（预检请求）发送成功，则会在头部多返回以下字段

Access-Control-Allow-Origin: http://localhost:3001 //该字段表明可供那个源跨域

Access-Control-Allow-Methods: GET, POST, PUT // 该字段表明服务端支持的请求方法

Access-Control-Allow-Headers: X-Custom-Header // 实际请求将携带的自定义请求首部字段

（一）浏览器设置预检字段：

Access-Control-Request-Method:必写，用来列出浏览器 CORS 请求可以使用的 HTTP 方法

Access-Control-Request-Headers：用来指定浏览器可额外发送的请求头

（二）服务器通过预检后比简单请求多出的返回字段

Access-Control-Allow-Methods: 必须设置，表明服务器支持的跨域方法，值为逗号隔开的字符串

Access-Control-Allow-Headers: 如果浏览器设置了 Access-Control-Request-Headers，那么此字段也必须设置，值为逗号隔开的字符串

Access-Control-Max-Age: 该字段可选，用来指定本次预检请求的有效期，单位为秒

Cors 跨域案例（来自头条）

跨越案例：

页面地址 http://client.cors.com:8000/greeter.html，代码如下：

```
<!DOCTYPE html>
<html>

<head>
    <title>cors</title>
</head>

<body>
    <script>
        var url = 'http://server.cors.com:3000/data';
        var xhr = new XMLHttpRequest();
        xhr.open('GET', url, true);
        xhr.send();
        xhr.onreadystatechange = function() {
            console.info(xhr.responseText);
        }
    </script>
</body>

</html>
```


图 1

服务器接口地址：http://server.cors.com:3000/data，服务器代码如下：

```
JS server.js > ...
1   const Koa = require('koa');
2   const app = new Koa();
3
4   app.use(async ctx => {
5       ctx.body = {
6           errno: 0,
7           data: [1, 2, 3]
8       };
9   });
10
11  app.listen(3000);
```


图 2

很明显，当页面在请求服务器接口时会发生跨域现象，如下：

图 3

我们去浏览器 Network 中看一下请求信息，

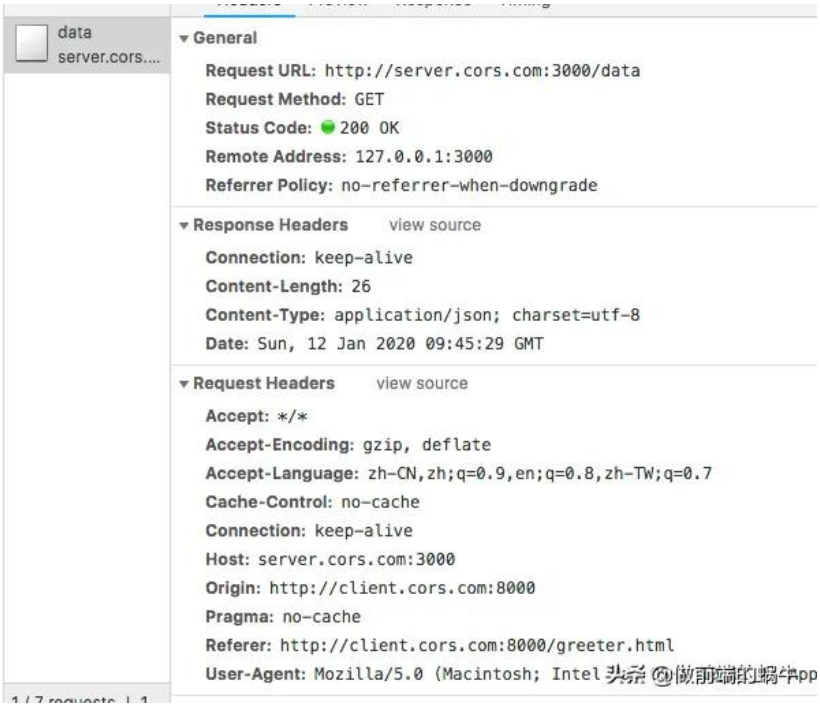


图 4

如图 4 所示，响应为 200，response Headers 信息也很正常，**这说明在跨域的情况下请求依然可以到达服务器，并且服务器能够正常响应，但是由于浏览器的同源策略并没有把返回的数据给到页面。**

那么如何让页面在跨域的情况下获取到数据呢？我们回看图 3，似乎在说少了一个 Access-Control-Allow-Origin 头，那么我们在服务器代码中加一下。

```
JS server.js > ...
1   const Koa = require('koa');
2   const app = new Koa();
3
4   app.use(async ctx => {
5       ctx.set({
6           'Access-Control-Allow-Origin': 'http://client.cors.com:8000'
7       });
8       ctx.body = {
9           errno: 0,
10          data: [1, 2, 3]
11      };
12  });
13
14  app.listen(3000);
```


图 5

现在刷新页面，服务器返回的数据就能正常打印出来了。（Access-Control-Allow-Origin': '*'表示接受任意域名的请求）

携带凭证

在跨域的情况，服务器有时依然需要鉴权。通常服务器鉴权都是从 cookie 中获取信息来判断客户端的身份，那么跨域的情况下请求还能传递 cookie 吗？当然能，但是 cookie 会遵守同源策略！

1）服务器设置 cookie

```
JS server.js > app.use() callback
1   const Koa = require('koa');
2   const app = new Koa();
3
4   app.use(async ctx => {
5       ctx.set({
6           'Access-Control-Allow-Origin': 'http://client.cors.com:8000',
7           'Access-Control-Allow-Credentials': true
8       });
9
10      ctx.cookies.set('server', 123, {
11          httpOnly: false
12      });
13
14      ctx.body = {
15          errno: 0,
16          data: [1, 2, 3]
17      };
18  });
19
20  app.listen(3000);
```


图 6

```
<body>
    <script>
        var url = 'http://server.cors.com:3000/data';
        var xhr = new XMLHttpRequest();
        xhr.withCredentials = true;
        xhr.open('GET', url, true);
        xhr.send();
        xhr.onreadystatechange = function() {
            console.info(xhr.responseText);
        }
    </script>
</body>
```


图 7

如果需要服务器设置 cookie，必须设置 Access-Control-Allow-Credentials: true，否则会出现如下错误。

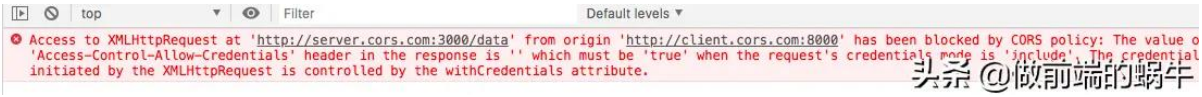


图 8

页面中的 xhr 对象也必须设置属性 withCredentials=true。

此时刷新页面，在页面控制台中通过 document.cookie 查看 server=123，你会发现 server 端设置的 cookie 并不存在。上面已经说了 cookie 会遵循同源策略，服务器的域名是 server.cors.com，所以服务器设置的 cookie 应该是在这个域名下，并不会在页面的域名（client.cors.com）下，那如何验证服务器设置 cookie 成功呢?

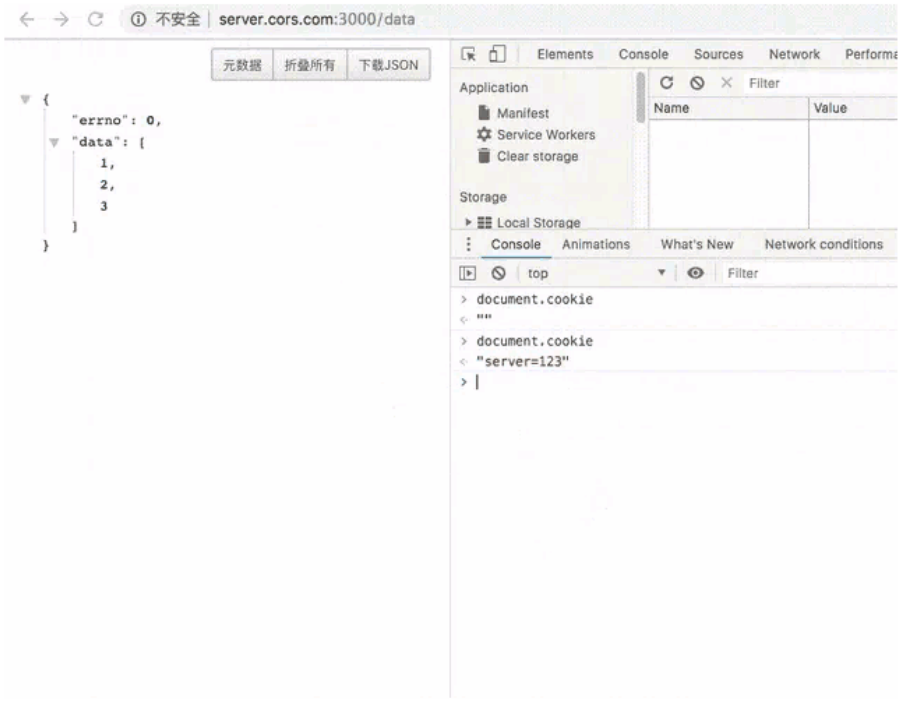


图 9

先打开接口页面，这个页面是同源的；

回到请求页面，刷新；

再回到接口页面，在控制台通过 document.cookie 就可以拿到服务器设置的 cookie。

2）页面设置 cookie

如果主域名相同，比如现在的例子，主域名都是 cors.com，那么就可以把 cookie 设置在这个主域名下。document.cookie="client=1;domain=cors.com;" 这样服务器就能获取到页面设置的 cookie。

如果连主域名都不一样，那就不要妄想在页面上设置 cookie 让服务器获取到。这种情况下，服务器该如何鉴权呢?

第一种方式是让后端把跨域的接口代理成同域的，这样我们的后端可以拿到 cookie，在他那把 cookie 转发给跨域服务。

图 10

第二种方式是页面发送请求时在 header 中附加一个 token，用于鉴权，

```
<body>
    <script>
        var url = 'http://server.cors.com:3000/data';
        var xhr = new XMLHttpRequest();
        xhr.withCredentials = true;
        xhr.open('GET', url, true);

        xhr.setRequestHeader('token', '1234567890');

        xhr.send();
        xhr.onreadystatechange = function() {
            console.info(xhr.responseText);
        }
    </script>
```


图 11

当刷新页面时，页面控制台又报错了。

图 12

提示设置 Access-Control-Allow-Headers，那我们就设置一下。

```
const Koa = require('koa');
const app = new Koa();

app.use(async ctx => {
    ctx.set({
        'Access-Control-Allow-Origin': 'http://client.cors.com:8000',
        'Access-Control-Allow-Credentials': true,
        'Access-Control-Allow-Headers': 'token'
    });

    console.info(ctx.get('token'));

    ctx.cookies.set('server', 123, {
        httpOnly: false
    });

    ctx.body = {
        errno: 0,
        data: [1, 2, 3]
    };
});

app.listen(3000);
```


图 13

再刷新页面，请求正常了，服务端也能拿到 token 的值了。

简单请求与非简单请求

图 13 中我们拿到了 token 的值，此时我们再去瞧瞧浏览器中的 Network，会发现页面发送了两个请求，第一个请求的 method 是 OPTIONS，这是怎么回事呢?

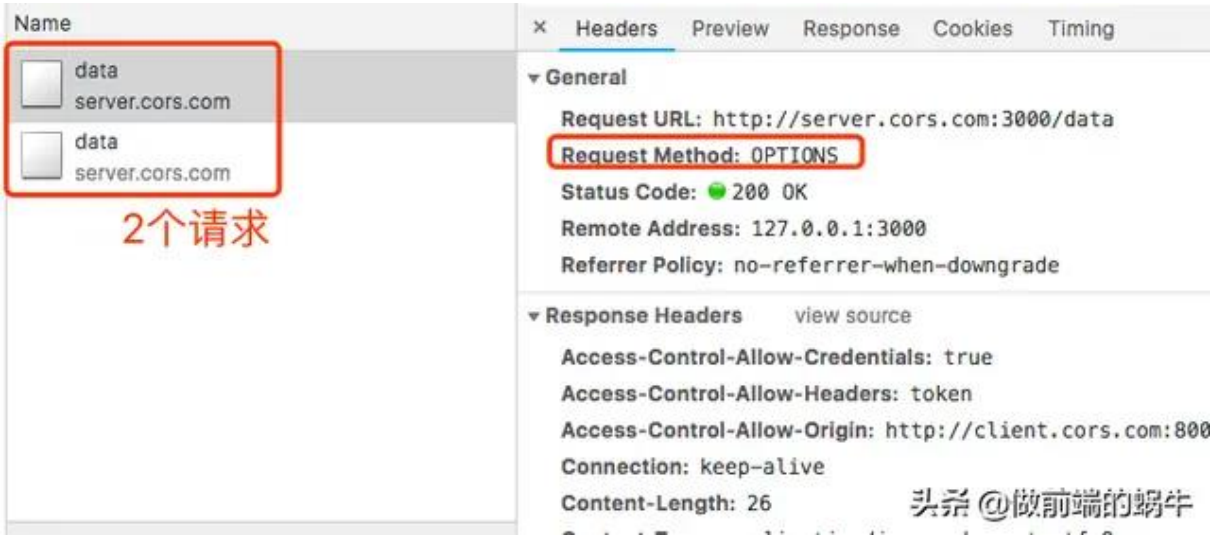


图 14

原来 cors 跨域也分简单请求和非简单请求。

简单请求条件如下：

请求方法是必须是 HEAD/GET/POST 三种方法之一；

HTTP 的头信息不超出这几种字段：Accept/Accept-Language/Content-Language/Content-Language/Last-Event-ID/Content-Type，Content-Type 只限于三个值 application/x-www-form-urlencoded、multipart/form-data、text/plain。

图 11 中我们设置了 token 请求头，显然不满足以上条件，所以是非简单请求。非简单请求的 CORS 请求会在正式通信之前增加一次 HTTP 查询请求，称为预检请求（preflight）。浏览器先询问服务器，当前网页所在的域名是否在服务器的许可名单之中，以及可以使用哪些 HTTP 动词和头信息字段。只有得到肯定答复，浏览器才会发出正式的 XMLHttpRequest 请求，否则就报错。预检请求用的请求方法是 OPTIONS，表示这个请求是用来询问的。

我们现在尝试发送一次 PUT 请求，看看会有什么现象？

```
<body>
    <script>
        var url = 'http://server.cors.com:3000/data';
        var xhr = new XMLHttpRequest();
        xhr.withCredentials = true;

        xhr.open('PUT', url, true);

        xhr.setRequestHeader('token', '1234567890');

        xhr.send();
        xhr.onreadystatechange = function() {
            console.info(xhr.responseText);
        }
    </script>
</body>
```


图 15

不出所料，浏览器再次报错！

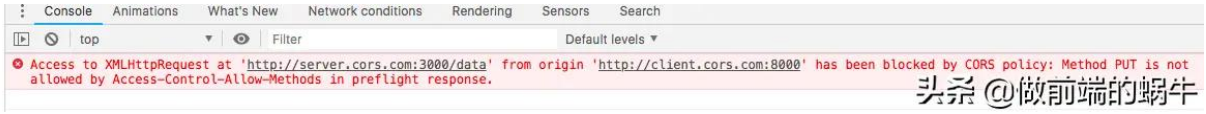


图 16

提示我们设置 Access-Control-Allow-Methods，那就只能设置了！

```
const Koa = require('koa');
const app = new Koa();

app.use(async ctx => {
    ctx.set({
        'Access-Control-Allow-Origin': 'http://client.cors.com:8000',
        'Access-Control-Allow-Credentials': true,
        'Access-Control-Allow-Headers': 'token',
        'Access-Control-Allow-Methods': 'PUT'
    });

    console.info(ctx.get('token'));

    ctx.cookies.set('server', 123, {
        httpOnly: false
    });

    ctx.body = {
        errno: 0,
        data: [1, 2, 3]
    };
});

app.listen(3000);
```

图 17

再次刷新页面，现在请求正常了！我们回头看一下预检请求，

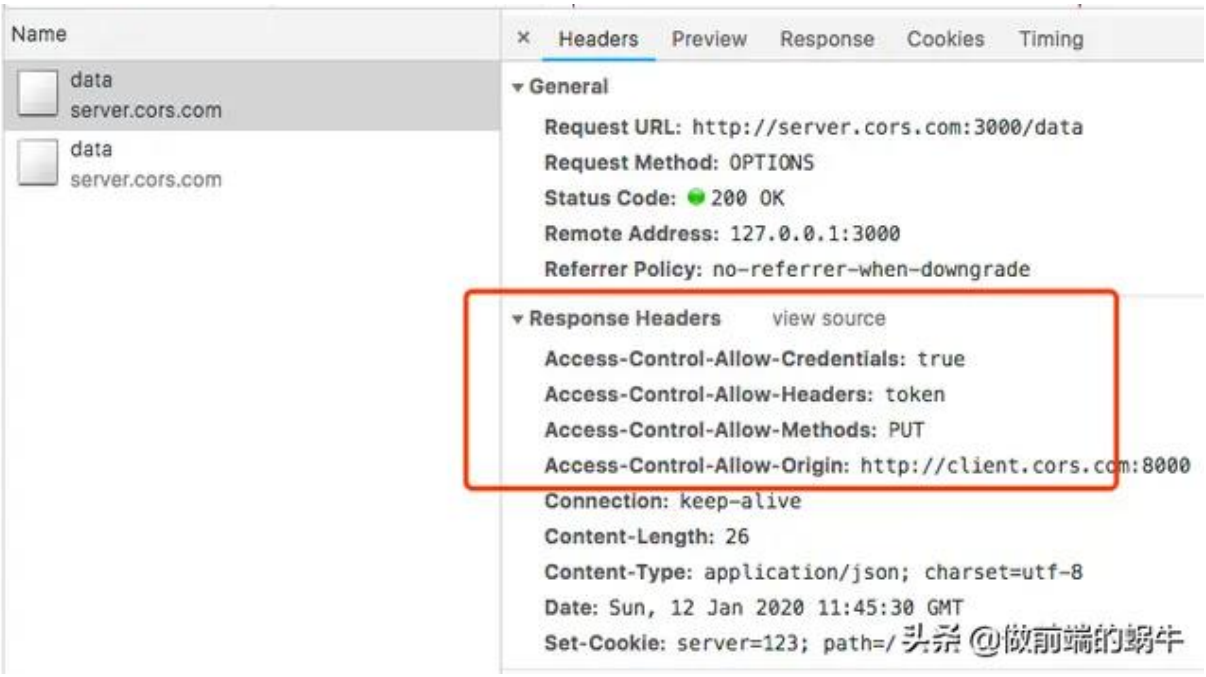


图 18

不得不说，浏览器在访问跨域接口时真的是非常的小心，当然这一切都是为了安全考虑。即使这样，现在网络中也不乏 XSS、CSRF 等攻击。

●安全类（具体找资料深入了解吧）

前端安全分类，主要有以下两大类：

1、CSRF

概念：

CSRF（Cross-site request forgery）通常称为跨站请求伪造

原理：

防御措施：

Token 验证

Referer 验证

隐藏令牌

2、XSS

概念：

跨域脚本攻击（cross-site scripting）

原理：

http://www.imooc.com/learn/812

防御措施：

http://www.imooc.com/learn/812

●算法类（无标准，非所有公司都考察，难度最大）

首先要必备数据结构与算法知识，然后慢慢刷题研究，非一朝一夕能完成的，这部分自己找资料慢慢研究摸索吧。

几个基本功类题型：

排序

快速排序：https://segmentfault.com/a/1190000009426421

选择排序：https://segmentfault.com/a/1190000009366805

希尔排序：https://segmentfault.com/a/1190000009461832

堆栈、队列、链表

堆栈：http://juejin.im/entry/58759e79128fe1006b48cdfd

队列：http://juejin.im/entry/58759e79128fe1006b48cdfd

链表：http://juejin.im/entry/58759e79128fe1006b48cdfd

递归

https://segmentfault.com/a/1190000009857470

波兰式和逆波兰式

理论：http://www.cnblogs.com/chenying99/p/3675876.html

源码：https://github.com/Tairraos/rpn.js/blob/master/rpn.js

.....

基础能力提升部分

（主要考察知识面广、理解深刻、回答灵活）

●渲染机制类

什么是 DOCTYPE 及作用

DTD（document type definition，文档类型定义）是一系列的语法规则，用来定义 XML 或（X）HTML 的文件类型。浏览器会使用他来判断文档类型，决定使用何种协议来解析，以及切换浏览器模式。

（DTD 简单解释就是说告诉浏览器我是什么文档类型，浏览器根据这个去判断用什么解析他,渲染他）

DOCTYPE 是用来声明文档类型和 DTD 规范的。一个最主要的用途就是用来验证文件的合法性验证，如果那文件代码不合法，那么浏览器解析时会出一些差错

（DOCTYPE 就是告诉浏览器当前文件是什么 DTD 的，也就是什么文档类型）

综上所述，简单来说就是 DOCTYPE 用于声明是哪种 DTD，也就是什么文档类型，浏览器根据当前的文档类型去判断用什么去解析、渲染当前文档，DOCTYPE 不存在或格式不正确会导致文档以兼容模式呈现。如果不声明，浏览器解析的文档与当前不一致，就会导致错误。

常见的 DOCTYPE 文档类型有（记住 HTML5）：

HTML5： <!DOCTYPE html>

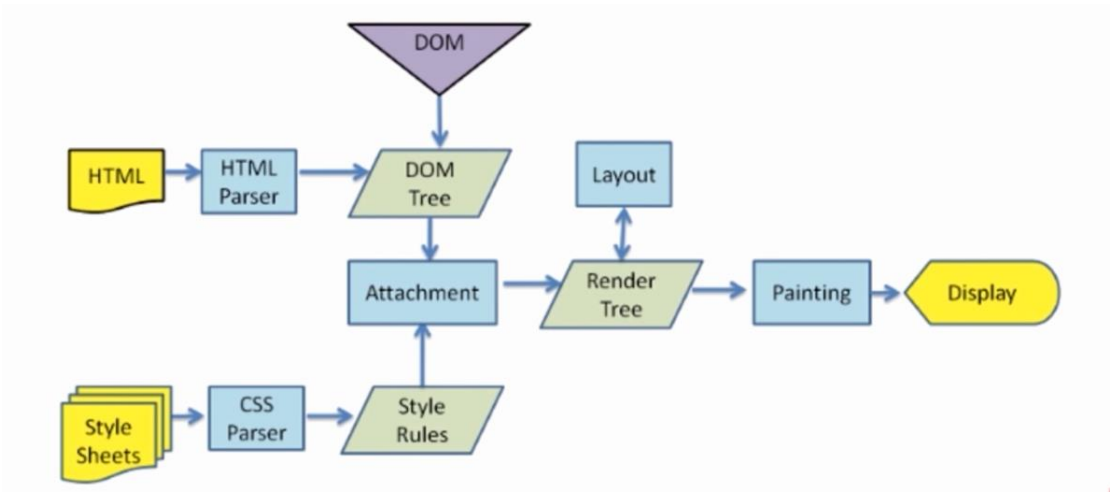
其他的，比如 html4.01 等写法挺长的，查查了解下吧，没必要刻意记住，知道 HTML4.01 分为严格模式和传统模式两种写法就行了

HTML5 为什么只需要写<!DOCTYPE html> ？

HTML5 不基于 SGML，因此不需要对 DTD 进行引用，但是需要 doctype 来规范浏览器的行为（让浏览器按照它们应该的方式来运行）。

而 HTML4.01 基于 SGML,所以需要对 DTD 进行引用，才能告知浏览器文档所使用的文档类型。

浏览器渲染过程（主要是 html、css、js）



简单描述：

通过上图结合记忆

1、Html 通过 HTML 解析器（HTML parser）解析成 HTML DOM 树（html dom tree）

2、Css 通过 css 解析器（css parser）解析成 CSS OM 树（css om tree）

3、将上述两者一结合，形成了渲染树（render tree），(此时，HTML 结构已经出来)

4、布局（layout）， 根据 Render Tree 计算每个节点的位置大小颜色等信息

5、绘制（Painting ）根据计算好的信息绘制整个页面，最后的 display 就是浏览器展示的页面。

由渲染机制引发了一个例题：

从浏览器 URL 输入一个地址（比如百度地址），敲回车，浏览器做了啥?

1. **根据域名到 DNS 中找到 IP（有浏览器缓存——>系统缓存——>路由缓存——>服务器）**

   正常情况下，浏览器会缓存 DNS 一段时间，一般 2 分钟到 30 分钟不等。如果有缓存，直接返回 IP。

   缓存中如果没有查到 IP，浏览器会做系统调用，读取主机的 hosts 文件，也就是 hosts 文件中的域名与 ip 的映射关系，(即查找的系统缓存是否有 ip）如果找到，直接返回 IP。

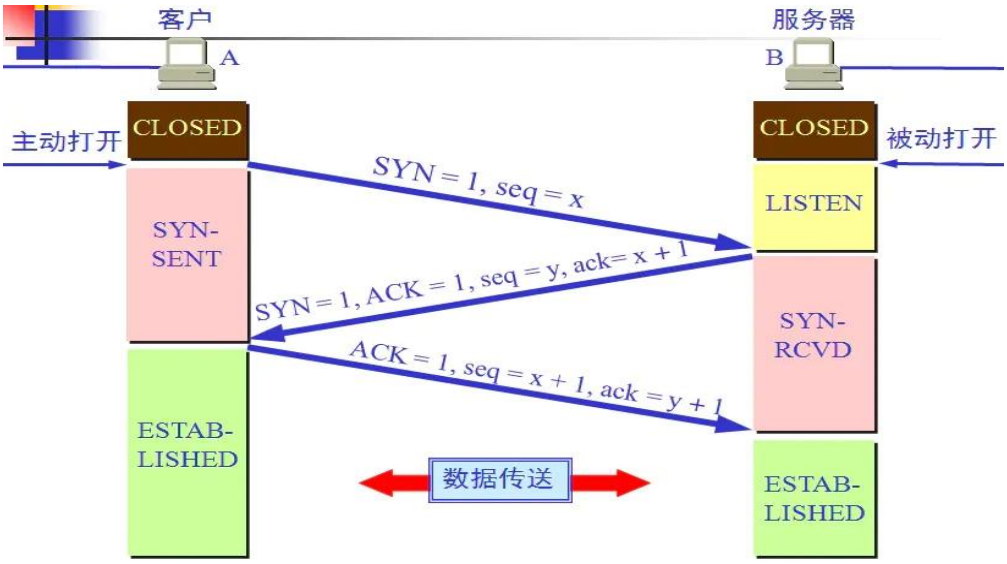
   hosts 文件里面还是没有找到，则直接去路由器中寻找 DNS 缓存，一般这个时候都能找到对应的 IP。

   如果还是没有找到，ISP 的 DNS 服务器就开始从根域名服务器开始递归搜索，从.com 顶级域名服务器开始，一直到 baidu 的域名服务器。

   这个时候，浏览器就获取到了对应的 IP。在解析的过程中，常常会解析出不同的 IP，这是根据不同的用户，不同的网络供应商，所在的地域，等等等等进行计算给出的最优的 IP 地址。

   劫持 DNS，可以屏蔽掉很多网点的访问

2. **根据 IP 建立 TCP 连接(三次握手)**



（1）浏览器发出 TCP 连接请求，请求报文段：同步位 SYN = 1，选择序号 seq = x，然后等待服务器确认

（2）服务器收到连接请求后，返回应答报文：SYN = 1， ACK = 1，确认号 ack = x +1，自己的序号 seq = y

（3）浏览器收到应答报文后，发现 ACK 标志位为 1，表示连接请求确认。浏览器返回确认报文：ACK = 1，确认号 ack = y+1 服务器收到确认报文后建立 TCP 连接。

3. 连接建立成功发起 http 请求

浏览器向主机发起一个 HTTP-GET/POST 方法报文请求。请求中包含访问的 URL，还有 User-Agent 用户浏览器操作系统信息，编码等。Accept-Encoding 一般采用 gzip，压缩之后传输 html 文件。Cookies 如果是首次访问，会提示服务器建立用户缓存信息，如果不是，可以利用 Cookies 对应键值，找到相应缓存，缓存里面存放着用户名，密码和一些用户设置项。

4. 服务器响应 http 请求（这里可能会发生重定向），最终发回一个 html 响应

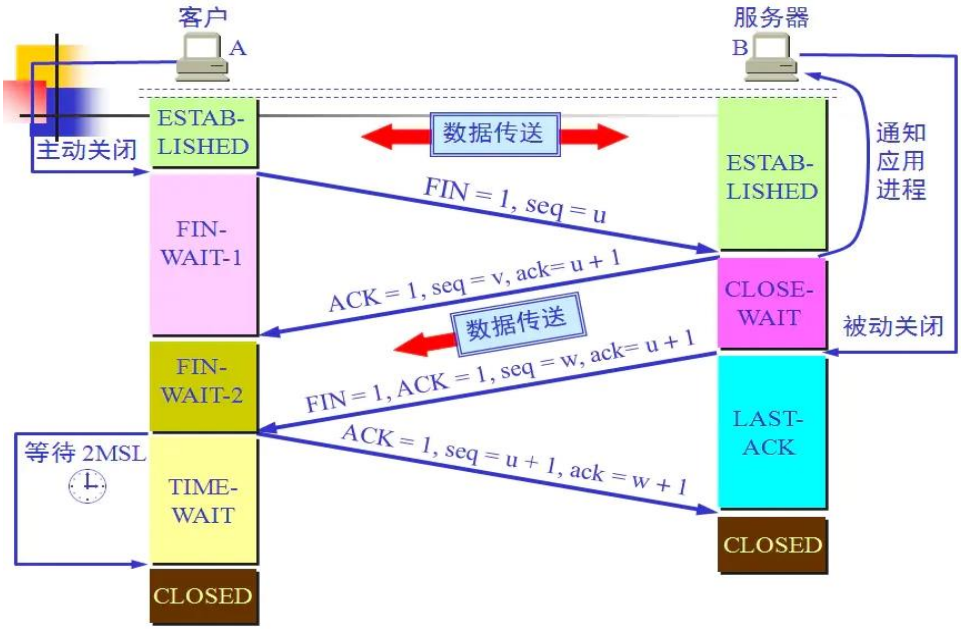
服务器接收到获取请求，然后处理并返回一个响应，返回状态码 200 表示 OK，表示服务器可以响应请求，返回报文，如果在报头中 Content-type 为 “text/html”，浏览器以 HTML 形式呈现。

但是，对于大型网站存在多个主机站点，往往不会直接返回请求页面，而是重定向。返回的状态码就不是 200 而是 301,302 等以 3 开头的重定向码，浏览器在获取了重定向响应后，在响应报文中 Location 项找到重定向地址，浏览器重新第一步访问即可。

重定向是为了负载均衡或者导入流量，提高 SEO 排名。利用一个前端服务器接受请求，然 后负载到不同的主机上，可以大大提高站点的业务并发处理能力；重定向也可将多个域名的访问，集中到一个站点；由于 baidu.com，www.baidu.com 会被搜索引擎认为是两个网站，照成每个的链接数都会减少从而降低排名，永久重定向会将两个地址关联起来，搜索引擎会认为是同一个网站，从而提高排名。

5. 关闭 TCP 连接（四次挥手）

断开一个 TCP 连接时，需要客户端和服务端总共发送 4 个包以确认连接的断开。在 socket 编程中，这一过程由客户端或服务端任一方执行 close 来触发。由于 TCP 连接时全双工的，因此，每个方向都必须要单独进行关闭，这一原则是当一方完成数据发送任务后，发送一个 FIN 来终止这一方向的连接，收到一个 FIN 只是意味着这一方向上没有数据流动了，即不会再收到数据了，但是在这个 TCP 连接上仍然能够发送数据，直到这一方向也发送了 FIN。首先进行关闭的一方将执行主动关闭，而另一方则执行被动关闭。整个流程如下：



6. 渲染页面

执行重排重绘的过程

（1）.Html 通过 HTML 解析器（HTML parser）解析成 HTML DOM 树（html dom tree）

（2）.Css 通过 css 解析器（css parser）解析成 CSS OM 树（css　om tree）

（3）.将上述两者一结合，形成了渲染树（render tree），(此时，HTML 结构已经出来)

（4）.布局（layout），根据 Render Tree 计算每个节点的位置大小颜色等信息

（5）.绘制（Painting ）根据计算好的信息绘制整个页面，最后的 display 就是浏览器展示的页面。

可参考以下地址：

https://blog.csdn.net/weixin_38497513/article/details/80918425

https://blog.csdn.net/MiemieWan/article/details/85708052

https://www.jianshu.com/p/8446d0ce9782

重排 reflow

定义：

DOM 元素中的各个元素都有自己的盒子（盒模型），这些都需要浏览器根据各种样式来计算，并根据计算结果将元素放到它该出现的位置，这个过程称之为 reflow。

触发 reflow:

对 dom 进行增删改的操作，会导致 reflow 或 repaint

移动 dom 的位置，或者说添加个动画

修改 css 样式（一般是改变宽高显示隐藏等）

Resize 窗口（移动端没这个问题）或滚动

修改网页默认字体时

重绘 repaint

一般只要 reflow 就会触发 repaint

定义：

当盒子的位置大小以及其他属性，如颜色字体大小等都确定下来后，浏览器于是便把这些元素都按照各自的特性绘制了一遍，于是页面的内容出现了，这个过程称之为 repaint。

（只要显示的内容样式发生了改变，页面显示的内容不一样了，就会出发 repaint）

触发 repaint：

DOM 改动

CSS 改动（颜色字体等）

补充：

页面渲染的一般过程为 JS > CSS > 计算样式 > 布局 > 绘制 > 渲染层合并。

页面的重绘与重排是无法避免的，怎么优化呢?

优化方案：

1).DOM 的多个读操作（或多个写操作），应该放在一起。不要两个读操作之间，加入一个写操作

2).如果某个样式是通过重排得到的，那么最好缓存结果。避免下一次用到的时候，浏览器又要重排。

3). 不要一条条地改变样式，而要通过改变 class，或者 csstext 属性，一次性地改变样式。

4).尽量使用离线 DOM，而不是真实的网面 DOM，来改变元素样式。比如，操作 Document Fragment 对象，完成后再把这个对象加入 DOM。再比如，使用 cloneNode() 方法，在克隆的节点上进行操作，然后再用克隆的节点替换原始节点。

5). 先将元素设为 display: none（需要 1 次重排和重绘），然后对这个节点进行 100 次操作，最后再恢复显示（需要 1 次重排和重绘）。这样一来，你就用两次重新渲染，取代了可能高达 100 次的重新渲染。

6). position 属性为 absolute 或 fixed 的元素，重排的开销会比较小，因为不用考虑它对其他元素的影响。

7). 只在必要的时候，才将元素的 display 属性为可见，因为不可见的元素不影响重排和重绘。另外，visibility: hidden 的元素只对重绘有影响，不影响重排。

8).使用虚拟 DOM 的脚本库，比如 React 等。

●JS 运行机制

例题：

```
console.log(1);
setTimeout(function(){
    console.log(2);
},0);
console.log(3);
```

以上打印顺序为 1　3　2

Js 为单线程，一次只能做一件事。

任务队列顺序：

同步任务

console.log()为同步任务

异步任务

setTimeout 就是异步任务，异步任务先挂载，先执行完同步任务再执行异步任务

setInterval、DOM 事件和 es6 Promise 也是异步任务

●页面性能（页面流不流畅）

常见题目：提升页面性能方式有哪些？（具体请查看后面的前端性能优化）

1、 资源压缩合并，减少 http 请求。

2、 js 代码异步加载（——>异步加载的方式——>异步加载的区别）

3、 利用浏览器缓存（——>缓存的分类——>缓存的原理）

4、 使用 CDN

5、 预解析 DNS

```
<link rel="dns-prefetch"   href="//host_name_to_prefetch.com">

<meta http-equiv="x-dns-prefetch-control" content="on">
```

高级的浏览器比如 a 标签之类默认会开启 dns 预解析，但是在 https 协议请求下，很多浏览器默认关闭 dns 预解析，上面的 meta 就是强制打开 dns 预解析的。

js 异步加载的方式：

动态脚本加载、async、defer

async 和 defer 区别

async 是加载和渲染后续文档元素的过程将和 js 的加载与执行并行进行（异步）。如果是多个，加载顺序和执行顺序无关。async 的脚本并不保证按照指定它们的先后顺序执行。对它来说脚本的加载和执行是紧紧挨着的，所以不管你声明的顺序如何，只要它加载完了就会立刻执行。

async 浏览器立即异步下载文件，不同于 defer 得是，下载完成会立即执行，此时会阻塞 dom 渲染，所以 async 的 script 最好不要操作 dom。因为是下载完立即执行，不能保证多个加载时的先后顺序。

defer 加载后续文档元素的过程将和 js 的加载并行进行（异步），但是 js 的执行要在所有元素解析完成之后，DOMContentLoaded 事件触发之前完成。也就是在 HTML 解析完之后 js 才会执行，如果多个，按加载顺序依次执行

defer 是表明脚本在执行时不会影响页面的构造。也就是说，脚本会被延迟到整个页面都解析完毕后再运行。浏览器渲染页面，读取到包含 defer 属性的外部<script>标签时不会停止 DOM 渲染，而是异步下载，加载完整个页面再运行 js。有多个 defer 的标签时，会按照顺序下载执行。

缓存分类（见上面的 http 协议中的缓存）

●错误监控

前端错误的分类

即时运行错误：代码错误

资源加载错误：图片加载失败、css、js 加载失败.....

错误的捕获方式

即时运行错误捕获方式：

try...catch...

window.onerror

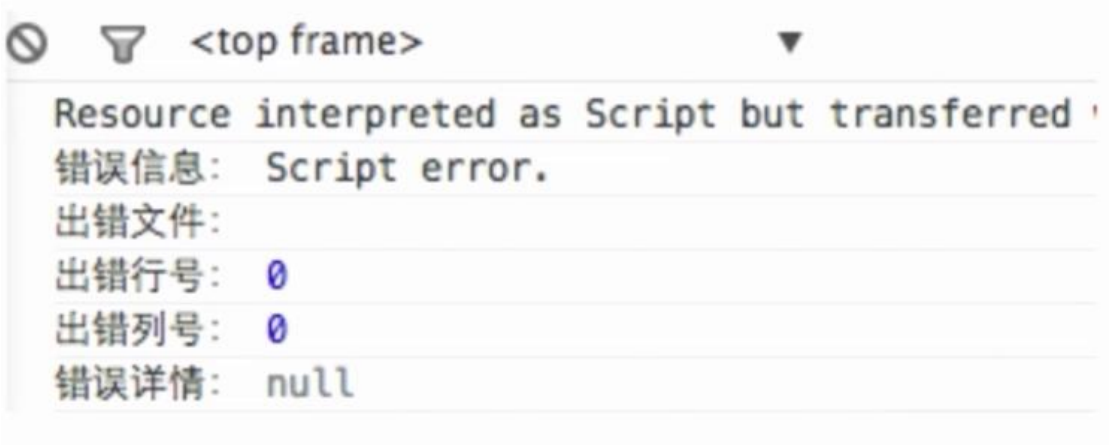
资源加载错误捕获方式（资源加载错误不会冒泡）：

object.onerror（object 为 img、script 等，添加 onerror 事件）

performance.getEntries()（高级浏览器提供的对象，返回一个数组）

Error 事件捕获（给 window 对象添加 error 事件在捕获阶段捕获错误）

延伸：跨域的 js 运行错误可以捕获吗，什么提示，怎么处理？



1、在 css 标签增加 crossorigin 属性

2、设置 js 资源响应头 Access-Control-Allow-Origin:*（可以是*也可以是域名）

上报错误的基本原理

1、 采用 ajax 通信的方式上报（可以实现，但是所有的错误都不是这么上报的）

2、 利用 image 对象上报（很多都用的这个）

(new Image()).src = ‘上报的路径’ 即可实现，路径也可以加任何参数信息

琐碎知识整理

●介绍下主流浏览器及内核

世界上一共有五大主流浏览器，都是在国外，就算不知道也千万别回答 360、QQ 等国内浏览器，否则你会显得很 **low**，很没有知识。

IE 浏览器：Trident 内核，俗称 IE 内核

（现在 win10 的 edge 浏览器用的 EdgeHTML 内核，属于 Trident 的分支）

谷歌浏览器：以前为 Webkit 内核，现在为 Blink 内核（属于 Webkit 内核的分支）

苹果的 safari 浏览器：Webkit 内核

火狐浏览器：Gecko 内核

欧朋浏览器：以前用的自己的独立内核 Presto，后来改为 Webkit，现在也改为谷歌的 Blink 内核

国内的浏览器，比如 360、QQ、猎豹等大多采用的是双内核，也就是 IE+chrome 内核，所以现在的浏览器大多有兼容模式（IE 内核）和极速模式（谷歌内核），其实就是两种内核的互相切换。

●Html 中的几种元素

行内元素（内联元素）

特点：

所有元素都默认在一行，大小随内容大小自动撑开，不能设置宽高，可以设置左右的 margin，不能设置上下的 margin，可以设置左右的 padding，不能设置上下的 padding（请看下面的括号内容）。行内元素中只能包含行内元素，不能有块级元素。

（有的文档说只能设置左右 padding，包括规范也是这么定义的。但是通过实验发现上下左右 padding 都可以，页面中可以看到设置上下的 padding 有效果。虽然页面效果是上下的 padding 都有作用，但是实际中会发现行内元素的上下 padding 真的不起作用，首先规范是这么定的，所谓的不起作用指的是，比如加的上下 padding 不会撑开父元素，padding 会超出，而且底下的元素不会按照正常的文档流显示，会显示在行内元素的 padding 内侧，不会被撑开，要解决需要转换为块级元素或行内块元素才可）。

问题如下展示：

div 里放了两个行内元素，一个 span,一个 a，设置了上下 padding，但是没有把父元素也就是绿色背景的元素撑开，div 下面的 div（黄色背景）跑到了 padding 里面，没有被行内元素的 padding 撑开。

即上下 padding 效果看似有了，但是实际不占据空间，有也像是没有效果，看似有，实际没有。

举例：

span、a、input、img、label、textarea、select、br、strong、sub、sup

HTML5：main 等等

问题：

1、 行内元素左右会有默认间距，可以通过设置父元素添加 font-size:0px;解决

2、 行内元素默认会通过基线（baseline）对齐，如果给一个行内元素转换为行内块元素的时候并设置了 margin-top，会发现这一行的所有行内元素都会具有 margin-top 属性，都有了上边距，这是行内元素默认的基线对齐导致的，解决办法为当前所在行内元素的所有行内元素添加 vertical-align:top/middle/bottom 可以解决。

转换块级元素：display:block;

注意：

img、input、select 可以设置宽高，因为它既是行内元素也是替换元素。

a 标签里面不能嵌套 a 标签

块级元素

特点：

一个元素在一行，行行排列，宽度默认百分百，可以设置宽高 margin 和 padding，块级元素中既可以包含块级元素也可以包含行内元素。

举例：

div、ul、li、ol、dl、dt、dd、table、th、tr、td、h1 到 h6、hr

HTML5：header、section、article、footer 等等

转换行内元素：display:inline;

注意：有些款元素里面也不能嵌套块元素 p，h1-h6，td 其中不能嵌套其他块元素

行内块元素

特点：

由于 display 有个属性 inline-block，所以有了行内块元素。它具有行内元素和块级元素的特性，默认水平排

列，可以设置宽高，margin 和 padding。

设置：display:inline-block;

空元素

特点：

没有内容和子节点，并且闭合在开始标签内，也就是单个标签的元素成为空元素。

举例：

img、br、hr、input、link、meta、source、keygen、command、embed、base 等

替换元素

特点：

替换元素是指元素内容的展现不是由 CSS 来控制的，而是外观渲染独立于 CSS 的外部对象。简而言之，替换元素就是浏览器根据元素的标签和属性，来决定元素的具体显示内容。这些元素都有默认尺寸。<img><input><textarea><select><object>等。替换一般有内在尺寸如 img 默认的 src 属性引用的图片的宽高，表单元素如 input 也有默认的尺寸。img 和 input 的宽高可以设定。

举例：

```
<img src=”a.png” />
<input type=”button” value=”按钮”/>
<select>
    <option value=”aaa”>aaa</option>
</select>
```

不可替换元素

特点：

不可替换元素则是将内容直接展现出来，不受元素标签和属性约束。Html 大多都是不可替换元素

举例：

```
<div>这是 div</div>
<p>这是段落</p>
<span>这是 span</span>
```

●如何清除浮动

1、 父元素设置高度
2、 父元素设置 overflow：hidden；
3、 父元素后面添加 div，设置 div 清除浮动
4、 用伪元素

```
父元素::after{
        content:”“;
        display:block;
        width:0;
        height:0;
        clear:both;
        visibility:hidden;     /*元素隐藏，可不加，本身设置宽高都是 0，加不加没啥意义*/
}
 父元素{
        zoom:1；/*网上说为了兼容 IE*，本人测试，加不加一样，没啥区别，IE9 以下一样不兼容/
}
```

5、 创建 BFC

●前端页面由哪三层构成，分别是什么，作用是什么

Html (结构)：

超文本标记语言，由 HTML 或 xhtml 之类的标记语言负责创建。标签，也就是那些出现在尖括号里的单词，对网页内容的语义含义做出了描述，但这些标签不包含任何关于如何显示有关内容的信息。例如，P 标签表达了这样一种语义：“这是一个文本段。”

Css (表现)：

层叠样式表 ，由 css 负责创建。css 对“如何显示有关内容”的问题做出了回答。

Js (行为)：

客户端脚本语言 ，内容应该如何对事件做出反应

●CSS 优先级

从样式上来讲：

行内样式>内部样式>外部样式

外部样式按照从上到下的加载顺序优先级依次变大，下面的样式会替换上面的样式

选择器：

!important > id 选择器 > class 选择器 > 标签选择器 > 通配符选择器 > 继承 > 浏览器默认属性

总体结合：

!important > 行内样式 > ID 选择器 > 类选择器 > 标签 > 通配符 > 继承 > 浏览器默认属性

总之无论如何，!important 选择器是最大的，永远排老大的位置，一旦使用，样式很难替换，使用要慎重，不到万不得已，不推荐使用。

●Js 基本数据类型

Es5 五种：String，Number，Boolean，Null，undefined　　es6 新增 Symbol

一种复杂类型：object

Type of 返回几种数据类型?

Es5 六种，加上 es6 的 symbol 共七种：

String，Number，Boolean，undefined，object(Null 返回对象)，function，symbol（es6 新增）

1、number

```
typeof(10);
typeof(NaN);
//NaN 在 JavaScript 中代表的是特殊非数字值,它本身是一个数字类型。
typeof(Infinity);
```

2、boolean

```
typeof(true);
typeof(false);
```

3、string

```
typeof("abc");
```

4、undefined
typeof(undefined);
typeof(a);//不存在的变量

5、object
对象，数组，null 返回 object
typeof(null);
typeof(window);

6、function
typeof (Array)
typeof(Date)

7、symbol
typeof Symbol() // ES6 提供的新的类型

补充：Undefined 和 Null 区别

Undefined 类型只有一个值，即 undefined。当声明的变量还未被初始化时，变量的默认值为 undefined。

Null 类型也只有一个值，即 null。null 用来表示尚未存在的对象，常用来表示函数企图返回一个不存在的对象。null 表示一个值被定义了，但是这个是是个空值

●垂直居中的解决方案

1、定位居中

绝对定位，左右百分之五十，上下外边距设置负值，为当前盒子宽高一半

绝对定位，左右百分之五十，用 transform:translate(-50%,-50%);自动左右减少盒子一半。

2、设置行高

父元素设置高度，添加 line-height；为高度高度

3、设置 display:table-cell

父元素添加 display:table-cell; vertical-align:middle;

4、伸缩盒子 display:flex;

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

●Link 与@import 区别

本质上，这两种方式都是为了加载 css 文件，但还是存在细微的差别。

差别 1：

link 属于 XHTML 标签，而@import 完全是 css 提供的一种方式。link 标签除了可以加载 css 外，还可以做很多其他的事情，比如定义 RSS，定义 rel 连接属性等，@import 只能加载 CSS。

差别 2：

加载顺序的差别：当一个页面被加载的时候（就是被浏览者浏览的时候），link 引用的 CSS 会同时被加载，而@import 引用的 CSS 会等到页面全部被下载完再加载。所以有时候浏览@import 加载 CSS 的页面时会没有样式(就

是闪烁），网速慢的时候还挺明显。

差别 3：

兼容性的差别。由于@import 是 CSS2.1 提出的所以老的浏览器不支持，@import 只有在 IE5 以上的才能识别，而 link 标签无此问题，完全兼容。

差别 4：

使用 dom 控制样式时的差别。当时用 JavaScript 控制 dom 去改变样式的时候，只能使用 link 标签，因为@import 不是 dom 可以控制的（不支持）。

差别 5（不推荐）：

@import 可以在 css 中再次引入其他样式表，比如创建一个主样式表，在主样式表中再引入其他的样式表，如：

```
@import “sub1.css”；
@import “sub2.css”；
sub1.css
p {color:red;}
sub2.css
.myclass {color:blue}
```

这样有利于修改和扩展。

但是：这样做有一个缺点，会对网站服务器产生过多的 HTTP 请求，以前是一个文件，而现在确实两个或更多的文件了，服务器压力增大，浏览量大的网站还是谨慎使用。

@import 的书写方式

```
@import 'style.css' //Windows IE4/ NS4, Mac OS X IE5, Macintosh IE4/IE5/NS4 不识别
@import "style.css" //Windows IE4/ NS4, Macintosh IE4/NS4 不识别
@import url(style.css) //Windows NS4, Macintosh NS4 不识别
@import url('style.css') //Windows NS4, Mac OS X IE5, Macintosh IE4/IE5/NS4 不识别
@import url("style.css") //Windows NS4, Macintosh NS4 不识别
```

由上分析知道，@import url(style.css)和@import url（"style.css"）是@import 写法的最优选择，兼容的浏览器最多。从字节优化的角度来看@import url(style.css)最值得推荐。

●css 单位及区别

px： 绝对单位，页面按精确像素展示。代表物理屏幕上能显示出的最小的一个点，也就是像素值

em： 相对单位，基准点为父节点字体的大小，继承父类字体的大小，相当于“倍”。浏览器默认字体大小为 16px=1em，始终按照 div 继承来的字体大小显示，如果自身定义了 font-size 按自身来计算，整个页面内 1em 不是一个固定的值，1em 相当于当前一个字体大小。

rem： 相对单位，可理解为”root em”，相对根节点 html 的字体大小来计算，继承根节点的属性（即标签）。1rem 相当于根节点一个字体大小。

注： chrome 强制最小字体为 12 号，即使设置成 10px 最终都会显示成 12px，当把 html 的 font-size 设置成 10px,子节点 rem 的计算还是以 12px 为基准。

vw： viewpoint width，视窗宽度，1vw 等于视窗宽度的 1%。

即当前可见宽度的 1%=1vw，区别是：当 div 中没有内容时，width=100%，则宽度不显示出来；当 div 中没有内容时，width=100vh，则宽度依然能显示出来。

vh： viewpoint height，视窗高度，1vh 等于视窗高度的 1%。即当前可见高度的 1%=1vh，区别是：当 div 中没有内容时，height=100%，则高度不显示出来；当 div 中没有内容时，height=100vh，则高度依然能显示出来。

vmin： vw 和 vh 中较小的那个。

vmax： vw 和 vh 中较大的那个。

注： vw, vh, vmin, vmax：IE9+局部支持，chrome/firefox/safari/opera 支持，iOS safari 8+支持，Android browser4.4+支持，chrome for android39 支持。

●数组去重

以下仅供参考，有些可能类似，方法多种多样。

一、利用 ES6 Set 去重（ES6 中最常用）

```
function unique (arr) {
  return Array.from(new Set(arr))
}
var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];
console.log(unique(arr))
 //[1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {}, {}]
```

不考虑兼容性，这种去重的方法代码最少。这种方法还无法去掉“{}”空对象，后面的高阶方法会添加去掉重复“{}”的方法。

二、利用 for 嵌套 for，然后 splice 去重（ES5 中最常用）

```
function unique(arr){
        for(var i=0; i<arr.length; i++){
            for(var j=i+1; j<arr.length; j++){
                if(arr[i]==arr[j]){            //第一个等同于第二个，splice 方法删除第二个
                    arr.splice(j,1);
                    j--;
                }
            }
        }
return arr;
}
var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];
    console.log(unique(arr))
    //[1, "true", 15, false, undefined, NaN, NaN, "NaN", "a", {...}, {...}]        //NaN 和{}没有去重，两个 null 直接消失了
```

双层循环，外层循环元素，内层循环时比较值。值相同时，则删去这个值。

三、利用 indexOf 去重

```
function unique(arr) {
    if (!Array.isArray(arr)) {
        console.log('type error!')
        return
    }
```

```
    var array = [];
    for (var i = 0; i < arr.length; i++) {
        if (array .indexOf(arr[i]) === -1) {
            array .push(arr[i])
        }
    }
    return array;
}
var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];
console.log(unique(arr))
   // [1, "true", true, 15, false, undefined, null, NaN, NaN, "NaN", 0, "a", {...}, {...}]   //NaN、{}没有去重
```

新建一个空的结果数组，for 循环原数组，判断结果数组是否存在当前元素，如果有相同的值则跳过，不相同则 push 进数组。

四、利用 sort()

```
function unique(arr) {
    if (!Array.isArray(arr)) {
        console.log('type error!')
        return;
    }
    arr = arr.sort()
    var arrry= [arr[0]];
    for (var i = 1; i < arr.length; i++) {
        if (arr[i] !== arr[i-1]) {
            arrry.push(arr[i]);
        }
    }
    return arrry;
}
     var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];
        console.log(unique(arr))
// [0, 1, 15, "NaN", NaN, NaN, {...}, {...}, "a", false, null, true, "true", undefined]          //NaN、{}没有去重
```

利用 sort()排序方法，然后根据排序后的结果进行遍历及相邻元素比对。

五、利用对象的属性不能相同的特点进行去重（这种数组去重的方法有问题，不建议用，有待改进）

```
function unique(arr) {
    if (!Array.isArray(arr)) {
        console.log('type error!')
        return
    }
    var arrry= [];
     var   obj = {};
    for (var i = 0; i < arr.length; i++) {
        if (!obj[arr[i]]) {
```

```
                arrry.push(arr[i])
                obj[arr[i]] = 1
            } else {
                obj[arr[i]]++
            }
        }
        return arrry;
}
    var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];
        console.log(unique(arr))
//[1, "true", 15, false, undefined, null, NaN, 0, "a", {...}]      //两个 true 直接去掉了，NaN 和{}去重
```

六、利用 includes

```
function unique(arr) {
    if (!Array.isArray(arr)) {
        console.log('type error!')
        return
    }
    var array =[];
    for(var i = 0; i < arr.length; i++) {
            if( !array.includes( arr[i]) ) {//includes 检测数组是否有某个值
                    array.push(arr[i]);
                }
    }
    return array
}
var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];
    console.log(unique(arr))
    //[1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {...}, {...}]        //{}没有去重
```

七、利用 hasOwnProperty

```
function unique(arr) {
    var obj = {};
    return arr.filter(function(item, index, arr){
        return obj.hasOwnProperty(typeof item + item) ? false : (obj[typeof item + item] = true)
    })
}
    var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];
        console.log(unique(arr))
//[1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {...}]     //所有的都去重了
```

利用 hasOwnProperty 判断是否存在对象属性

八、利用 filter

```
function unique(arr) {
   return arr.filter(function(item, index, arr) {
      //当前元素，在原始数组中的第一个索引==当前索引值，否则返回当前元素
      return arr.indexOf(item, 0) === index;
   });
}
      var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];
            console.log(unique(arr))
//[1, "true", true, 15, false, undefined, null, "NaN", 0, "a", {...}, {...}]
```

九、利用递归去重

```
function unique(arr) {
            var array= arr;
            var len = array.length;

      array.sort(function(a,b){     //排序后更加方便去重
            return a - b;
      })

      function loop(index){
            if(index >= 1){
                  if(array[index] === array[index-1]){
                        array.splice(index,1);
                  }
                  loop(index - 1);      //递归 loop，然后数组去重
            }
      }
      loop(len-1);
      return array;
}
 var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];
console.log(unique(arr))
//[1, "a", "true", true, 15, false, 1, {...}, null, NaN, NaN, "NaN", 0, "a", {...}, undefined]
```

十、利用 Map 数据结构去重

```
function arrayNonRepeatfy(arr) {
   let map = new Map();
   let array = new Array();   // 数组用于返回结果
   for (let i = 0; i < arr.length; i++) {
      if(map .has(arr[i])) {   // 如果有该 key 值
         map .set(arr[i], true);
      } else {
         map .set(arr[i], false);     // 如果没有该 key 值
         array .push(arr[i]);
```

```
      }
    }
    return array ;
}
 var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];
      console.log(unique(arr))
//[1, "a", "true", true, 15, false, 1, {...}, null, NaN, NaN, "NaN", 0, "a", {...}, undefined]
```

创建一个空 Map 数据结构，遍历需要去重的数组，把数组的每一个元素作为 key 存到 Map 中。由于 Map 中不会出现相同的 key 值，所以最终得到的就是去重后的结果。

十一、利用 reduce+includes

```
function unique(arr){
      return arr.reduce((prev,cur) => prev.includes(cur) ? prev : [...prev,cur],[]);
}
var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];
console.log(unique(arr));
// [1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {...}, {...}]
```

十二、[...new Set(arr)]

```
[...new Set(arr)]
//代码就是这么少----（其实，严格来说并不算是一种，相对于第一种方法来说只是简化了代码）
```

●==与===区别

==是用来比较值是否相等，如果数据类型不同，往往会先转换为同一数据类型在比较，值相等即为相等。
===为全等与，即看值也看数据类型，有一个不同即为不等。值相同，数据类型不同，也为不等。

●js 中 this 指向问题

在 ES5 中，其实 this 的指向，始终坚持一个原理：this 永远指向最后调用它的那个对象，来，跟着我朗读三遍：this 永远指向最后调用它的那个对象，this 永远指向最后调用它的那个对象，this 永远指向最后调用它的那个对象。记住这句话，this 你已经了解一半了。

下面我们来看一个最简单的例子：

例 1：

```
var name = "windowsName";
function a() {
 var name = "Cherry";
 console.log(this.name);     // windowsName
 console.log("inner:" + this); // inner: Window
}
a();
console.log("outer:" + this)     // outer: Window
```

这个相信大家都知道为什么 log 的是 windowsName，因为根据刚刚的那句话“this 永远指向最后调用它的那个对象”，我们看最后调用 a 的地方 a();，前面没有调用的对象那么就是全局对象 window，这就相当于是 window.a()；注意，这里我们没有使用严格模式，如果使用严格模式的话，全局对象就是 undefined，那么就会报错 Uncaught TypeError: Cannot read property 'name' of undefined。

再看下这个例子：

例 2：

```
var name = "windowsName";
 var a = {
  name: "Cherry",
  fn : function () {
   console.log(this.name);   // Cherry
  }
 }
 a.fn();
```

在这个例子中，函数 fn 是对象 a 调用的，所以打印的值就是 a 中的 name 的值。是不是有一点清晰了呢~我们做一个小小的改动：

例 3：

```
var name = "windowsName";
 var a = {
  name: "Cherry",
  fn : function () {
   console.log(this.name);   // Cherry
  }
 }
 window.a.fn();
```

这里打印 Cherry 的原因也是因为刚刚那句话“this 永远指向最后调用它的那个对象”，最后调用它的对象仍然是对象 a。

我们再来看一下这个例子：

例 4：

```
var name = "windowsName";
var a = {
 // name: "Cherry",
 fn : function () {
  console.log(this.name);   // undefined
 }
}
window.a.fn();
```

这里为什么会打印 undefined 呢？这是因为正如刚刚所描述的那样，调用 fn 的是 a 对象，也就是说 fn 的内部的 this 是对象 a，而对象 a 中并没有对 name 进行定义，所以 log 的 this.name 的值是 undefined。

这个例子还是说明了：this 永远指向最后调用它的那个对象，因为最后调用 fn 的对象是 a，所以就算 a 中没有 name 这个属性，也不会继续向上一个对象寻找 this.name，而是直接输出 undefined。

再来看一个比较坑的例子：

例 5：

```
var name = "windowsName";
var a = {
 name : null,
 // name: "Cherry",
 fn : function () {
  console.log(this.name);   // windowsName
 }
}
var f = a.fn;
f();
```

这里你可能会有疑问，为什么不是 Cherry，这是因为虽然将 a 对象的 fn 方法赋值给变量 f 了，但是没有调用，再接着跟我念这一句话：“this 永远指向最后调用它的那个对象”，由于刚刚的 f 并没有调用，所以 fn() 最后仍然是被 window 调用的。所以 this 指向的也就是 window。

由以上五个例子我们可以看出，this 的指向并不是在创建的时候就可以确定的，在 es5 中，永远是 this 永远指向最后调用它的那个对象。

再来看一个例子：

例 6：

```
var name = "windowsName";
function fn() {
 var name = 'Cherry';
 innerFunction();
 function innerFunction() {
  console.log(this.name);   // windowsName
 }
}
fn()
```

读到现在了应该能够理解这是为什么了吧(o ˘ ▿ ˘)o。

怎么改变 this 的指向？改变 this 的指向我总结有以下几种方法：

使用 ES6 的箭头函数

在函数内部使用 _this = this

使用 apply、call、bind

new 实例化一个对象

例 7：

```
var name = "windowsName";
 var a = {
  name : "Cherry",
  func1: function () {
   console.log(this.name)
  },
  func2: function () {
   setTimeout( function () {
    this.func1()
```

```
    },100);
  }
 };
 a.func2()  // this.func1 is not a function
```

在不使用箭头函数的情况下，是会报错的，因为最后调用 setTimeout 的对象是 window，但是在 window 中并没有 func1 函数。

我们在改变 this 指向这一节将把这个例子作为 demo 进行改造。

箭头函数

众所周知，ES6 的箭头函数是可以避免 ES5 中使用 this 的坑的。箭头函数的 this 始终指向函数定义时的 this，而非执行时。，箭头函数需要记着这句话："箭头函数中没有 this 绑定，必须通过查找作用域链来决定其值，如果箭头函数被非箭头函数包含，则 this 绑定的是最近一层非箭头函数的 this，否则，this 为 undefined"。

例 8 ：

```
var name = "windowsName";
var a = {
 name : "Cherry",
 func1: function () {
  console.log(this.name)
 },
 func2: function () {
  setTimeout( () => {
   this.func1()
  },100);
 }
};
a.func2()  // Cherry
```

在函数内部使用 _this = this

如果不使用 ES6，那么这种方式应该是最简单的不会出错的方式了，我们是先将调用这个函数的对象保存在变量 _this 中，然后在函数中都使用这个 _this，这样 _this 就不会改变了。

例 9:

```
var name = "windowsName";
var a = {
 name : "Cherry",
 func1: function () {
  console.log(this.name)
 },
 func2: function () {
  var _this = this;
  setTimeout( function() {
   _this.func1()
  },100);
 }
};
a.func2()  // Cherry
```

这个例子中，在 func2 中，首先设置 var _this = this;，这里的 this 是调用 func2 的对象 a，为了防止在 func2 中的 setTimeout 被 window 调用而导致的在 setTimeout 中的 this 为 window。我们将 this(指向变量 a) 赋值给一个变量 _this，这样，在 func2 中我们使用 _this 就是指向对象 a 了。

使用 apply、call、bind

使用 apply、call、bind 函数也是可以改变 this 的指向的，原理稍后再讲，我们先来看一下是怎么实现的：

使用 apply

例 10：

```
var a = {
 name : "Cherry",
 func1: function () {
  console.log(this.name)
 },
 func2: function () {
  setTimeout( function () {
   this.func1()
  }.apply(a),100);
 }
};
a.func2()     // Cherry
```

使用 call

例 11：

```
var a = {
 name : "Cherry",
 func1: function () {
  console.log(this.name)
 },
 func2: function () {
  setTimeout( function () {
   this.func1()
  }.call(a),100);
 }
};
a.func2()     // Cherry
```

使用 bind

例 12：

```
var a = {
  name : "Cherry",
  func1: function () {
   console.log(this.name)
  },
  func2: function () {
   setTimeout( function () {
    this.func1()
```

```
    }.bind(a)(),100);
  }
 };
 a.func2()     // Cherry
```

apply、call、bind 区别

刚刚我们已经介绍了 apply、call、bind 都是可以改变 this 的指向的，但是这三个函数稍有不同。

在 MDN 中定义 apply 如下；

apply() 方法调用一个函数，其具有一个指定的 this 值，以及作为一个数组（或类似数组的对象）提供的参数

语法：

fun.apply(thisArg, [argsArray])

thisArg：在 fun 函数运行时指定的 this 值。需要注意的是，指定的 this 值并不一定是该函数执行时真正的 this 值，如果这个函数处于非严格模式下，则指定为 null 或 undefined 时会自动指向全局对象（浏览器中就是 window 对象），同时值为原始值（数字，字符串，布尔值）的 this 会指向该原始值的自动包装对象。

argsArray：一个数组或者类数组对象，其中的数组元素将作为单独的参数传给 fun 函数。如果该参数的值为 null 或 undefined，则表示不需要传入任何参数。从 ECMAScript 5 开始可以使用类数组对象。浏览器兼容性请参阅本文底部内容。

apply 和 call 的区别

其实 apply 和 call 基本类似，他们的区别只是传入的参数不同。

call 的语法为：

fun.call(thisArg[, arg1[, arg2[, ...]]])

所以 apply 和 call 的区别是 call 方法接受的是若干个参数列表，而 apply 接收的是一个包含多个参数的数组。

例 13：

```
var a ={
   name : "Cherry",
   fn : function (a,b) {
     console.log( a + b)
   }
 }
 var b = a.fn;
 b.apply(a,[1,2])   // 3
```

例 14：

```
var a ={
   name : "Cherry",
   fn : function (a,b) {
     console.log( a + b)
   }
 }
 var b = a.fn;
 b.call(a,1,2)   // 3
```

bind 和 apply、call 区别

我们先来将刚刚的例子使用 bind 试一下

```
var a ={
 name : "Cherry",
 fn : function (a,b) {
  console.log( a + b)
 }
}
var b = a.fn;
b.bind(a,1,2)
```

我们会发现并没有输出，这是为什么呢，我们来看一下 MDN 上的文档说明：

bind()方法创建一个新的函数，当被调用时，将其 this 关键字设置为提供的值，在调用新函数时，在任何提供之前提供一个给定的参数序列。

所以我们可以看出，bind 是创建一个新的函数，我们必须要手动去调用：

```
var a ={
  name : "Cherry",
  fn : function (a,b) {
   console.log( a + b)
  }
 }
 var b = a.fn;
 b.bind(a,1,2)()     // 3
```

●js 深拷贝与浅拷贝

前提：在 js 中，基本数据类型拷贝都是直接赋值，赋值一个新的数值，保存给新的变量，原来的值改变不会影响新的值得变化。而引用类型，如对象和数组，拷贝的则是引用的地址，指针的指向问题，原来值得变化还会反应到新的数值中。

浅拷贝：

浅拷贝只拷贝最外层的东西，更深层的东西拷贝的是引用。

深拷贝：

深拷贝拷贝的是所有的东西，拷贝的是多层，每一层的东西都会拷贝。

浅拷贝例子：

比如有一个对象，拷贝对象内容到另一个对象，先创建一个空对象，遍历循环原对象，把原对象的每一项值都赋值给新对象，如果原对象里面的数据中还有一个对象，那么新对象就是拷贝的这个对象的引用，原对象的值改变，所拷贝对象的值会跟着改变，这就是浅拷贝，当满足所有的全部拷贝，与原对象不受影响，那就是深拷贝。

其中 es6 提供了 object.assign(target,{})就是浅拷贝。

```
var obj = {
    name:’小明’ ,
    age:20,
    family:{//此对象就是拷贝的引用。
        parent: ‘大明’
    }
}
Var o = {}
Object.assign(o.obj);
Console.log(o);
```

以上就是浅拷贝，family 拷贝的是后面对象的引用，obj 如果更改，o 对象也会被更改。

深拷贝例子：

深拷贝可以通过函数递归来实现，如果是对象，递归调用，继续拷贝

```
var obj = {
    name:’小明’ ,
    age:20,
    family:{//此对象就是拷贝的引用。
        parent：‘大明’
    },
    color:[1,2],
}
Var o = {}
//数组和对象都会发生引用地址的拷贝，只需要判断这些就可，其他直接赋值。
function  deepCopy(newobj,oldobj){
    for(var k in oldobj){
        var item = oldobj[k];
        if(item instanceof Array){//这里先判断数组，后判断对象，因为数组也是对象，防止下面赋值为空对象。
            newobj[k] = [];
            deepCopy(newobj[k],item);
        }else if(item  instanceof  Object){
            newobj[k] = {};
            deepCopy(newobj[k],item);
        }else{
            newobj[k]=item;
        }
    }
}
deepCopy(o.obj);
console.log(o);
```

补充：数组 instanceof object 也是 true，所以上面先判断数组，在判断对象。

深拷贝和浅拷贝也有很多其他方法，可以自己搜下研究研究。

●js 网络请求相关（防抖、节流）

概念：

函数防抖(debounce)：

触发高频事件后 n 秒内函数只会执行一次，如果 n 秒内高频事件再次被触发，则重新计算时间。

函数节流(throttle)：

高频事件触发，但在 n 秒内只会执行一次，所以节流会稀释函数的执行频率。

函数节流（throttle）与 函数防抖（debounce）都是为了限制函数的执行频次，以优化函数触发频率过高导致的响应速度跟不上触发频率，出现延迟，假死或卡顿的现象。

防抖（debounce）

概念：

触发高频事件后 n 秒内函数只会执行一次，如果 n 秒内高频事件再次被触发，则重新计算时间。

缺点：

如果事件在规定的时间间隔内被不断的触发，则调用方法会被不断的延迟

实现方式：

每次触发事件时设置一个延迟调用方法，并且取消之前的延时调用方法

```
//防抖debounce代码：
function debounce(fn,delay) {
    var timeout = null; // 创建一个标记用来存放定时器的返回值
    return function (e) {
        // 每当用户输入的时候把前一个 setTimeout clear 掉
        clearTimeout(timeout);
        // 然后又创建一个新的 setTimeout，这样就能保证interval 间隔内如果时间持续触发，就不会执行 fn 函数
        timeout = setTimeout(() => {
            fn.apply(this, arguments);
        }, delay);
    };
}
// 处理函数
function handle() {
    console.log('防抖：', Math.random());
}

//滚动事件
window.addEventListener('scroll', debounce(handle,500));
```

```
//防抖 debounce 代码：
function debounce(fn,delay) {
    var timeout = null; // 创建一个标记用来存放定时器的返回值
    return function (e) {
        // 每当用户输入的时候把前一个 setTimeout clear 掉
        clearTimeout(timeout);
        // 然后又创建一个新的 setTimeout，这样就能保证 interval 间隔内如果时间持续触发，就不会执行 fn 函数
        timeout = setTimeout(() => {
            fn.apply(this, arguments);
        }, delay);
    };
}
// 处理函数
function handle() {
    console.log('防抖：', Math.random());
}
//滚动事件
window.addEventListener('scroll', debounce(handle,500));
```

应用场景

(1) 用户在输入框中连续输入一串字符后，只会在输入完后去执行最后一次的查询 ajax 请求，这样可以有效减少请求次数，节约请求资源；

(2) window 的 resize、scroll 事件，不断地调整浏览器的窗口大小、或者滚动时会触发对应事件，防抖让其只触发一次；

```
<head>
    <meta charset="UTF-8">
    <title>加入防抖</title>
    <style type="text/css"></style>
    <script type="text/javascript">
        window.onload = function () {
            //模拟ajax请求
            function ajax(content) {
                console.log('ajax request ' + content)
            }
            function debounce(fun, delay) {
                return function (args) {
                    //获取函数的作用域和变量
                    let that = this
                    let _args = args
                    //每次事件被触发，都会清除当前的timeer，然后重写设置超时调用
                    clearTimeout(fun.id)
                    fun.id = setTimeout(function () {
                        fun.call(that, _args)
                    }, delay)
                }
            }
            let inputDebounce = document.getElementById('debounce')
            let debounceAjax = debounce(ajax, 500)
            inputDebounce.addEventListener('keyup', function (e) {
                debounceAjax(e.target.value)
            })
        }
    </script>
</head>

<body>
    <div>
        2.加入防抖后的输入:
        <input type="text" name="debounce" id="debounce">
    </div>
</body>

</html>
```

代码说明:

1.每一次事件被触发，都会清除当前的 timer 然后重新设置超时调用，即重新计时。 这就会导致每一次高频事件都会取消前一次的超时调用，导致事件处理程序不能被触发;

2.只有当高频事件停止，最后一次事件触发的超时调用才能在 delay 时间后执行;

加入防抖后，当持续在输入框里输入时，并不会发送请求，只有当在指定时间间隔内没有再输入时，才会发送请求。如果先停止输入，但是在指定间隔内又输入，会重新触发计时。

节流（throttle）

概念：

高频事件触发，但在 n 秒内只会执行一次，所以节流会稀释函数的执行频率。

实现方式：

每次触发事件时，如果当前有等待执行的延时函数，则直接 return

```
//节流throttle代码：
function throttle(fn,delay) {
    let canRun = true; // 通过闭包保存一个标记
    return function () {
         // 在函数开头判断标记是否为true，不为true则return
        if (!canRun) return;
         // 立即设置为false
        canRun = false;
        // 将外部传入的函数的执行放在setTimeout中
        setTimeout(() => {
        // 最后在setTimeout执行完毕后再把标记设置为true(关键)表示可以执行下一次循环了。
        // 当定时器没有执行的时候标记永远是false，在开头被return掉
            fn.apply(this, arguments);
            canRun = true;
        }, delay);
    };
}

function sayHi(e) {
    console.log('节流: ', e.target.innerWidth, e.target.innerHeight);
}
window.addEventListener('resize', throttle(sayHi,500));
```


//节流 throttle 代码：
function throttle(fn,delay) {
    let canRun = true; // 通过闭包保存一个标记
    return function () {
         // 在函数开头判断标记是否为 true，不为 true 则 return
        if (!canRun) return;
         // 立即设置为 false
        canRun = false;
        // 将外部传入的函数的执行放在 setTimeout 中
        setTimeout(() => {
        // 最后在 setTimeout 执行完毕后再把标记设置为 true(关键)表示可以执行下一次循环了。
        // 当定时器没有执行的时候标记永远是 false，在开头被 return 掉
            fn.apply(this, arguments);
            canRun = true;
        }, delay);
    };
}

function sayHi(e) {

```
    console.log('节流：', e.target.innerWidth, e.target.innerHeight);
}
window.addEventListener('resize', throttle(sayHi,500));
```

应用场景

(1)鼠标连续不断地触发某事件（如点击），只在单位时间内只触发一次；

(2)在页面的无限加载场景下，需要用户在滚动页面时，每隔一段时间发一次 ajax 请求，而不是在用户停下滚动页面操作时才去请求数据；

(3)监听滚动事件，比如是否滑到底部自动加载更多，用 throttle 来判断；

```
    <script type="text/javascript">
        window.onload = function () {
            //模拟ajax请求
            function ajax(content) {
                console.log('ajax request ' + content)
            }

            function throttle(fun, delay) {
                let last, deferTimer
                return function (args) {
                    let that = this;
                    let _args = arguments;

                    let now = +new Date();
                    if (last && now < last + delay) {
                        clearTimeout(deferTimer);
                        deferTimer = setTimeout(function () {
                            last = now;
                            fun.apply(that, _args);
                        }, delay)
                    } else {
                        last = now;
                        fun.apply(that, _args);
                    }
                }
            }
            let throttleAjax = throttle(ajax, 1000)
            let inputThrottle = document.getElementById('throttle')
            inputThrottle.addEventListener('keyup', function (e) {
                throttleAjax(e.target.value)
            })
        }
    </script>
</head>

<body>
    <div>
        3.加入节流后的输入:
        <input type="text" name="throttle" id="throttle">
    </div>
</body>
```

实验可发现在持续输入时，会安装代码中的设定，每 1 秒执行一次 ajax 请求

总结:

函数防抖是某一段时间内只执行一次；而函数节流是间隔时间执行，不管事件触发有多频繁，都会保证在规定时间内一定会执行一次真正的事件处理函数。

原理：

防抖是维护一个计时器，规定在 delay 时间后触发函数，但是在 delay 时间内再次触发的话，都会清除当前的 timer 然后重新设置超时调用，即重新计时。这样一来，只有最后一次操作能被触发。

节流是通过判断是否到达一定时间来触发函数，若没到规定时间则使用计时器延后，而下一次事件则会重新设定计时器。

●Cookie、session、web storage

这里只做简单说明，具体怎么使用自行查找文档详细学习。

cookie 和 session

会话（Session）跟踪是 Web 程序中常用的技术，用来跟踪用户的整个会话。常用的会话跟踪技术是 Cookie 与 Session。Cookie 通过在客户端记录信息确定用户身份，Session 通过在服务器端记录信息确定用户身份。

详情可参考：https://www.cnblogs.com/shoshana-kong/p/10669900.html

webstorage

html5 中的 Web Storage 包括了两种存储方式：sessionStorage 和 localStorage。

sessionStorage 用于本地存储一个会话（session）中的数据，这些数据只有在同一个会话中的页面才能访问并且当会话结束后数据也随之销毁。因此 sessionStorage 不是一种持久化的本地存储，仅仅是会话级别的存储。而 localStorage 用于持久化的本地存储，除非主动删除数据，否则数据是永远不会过期的；

三者区别：

1、 cookie 数据始终在同源的 http 请求中携带（即使不需要），即 cookie 在浏览器和服务器间来回传递。而 sessionStorage 和 localStorage 不会自动把数据发给服务器，仅在本地保存。cookie 数据还有路径（path）的概念，可以限制 cookie 只属于某个路径下。

2、 存储大小限制也不同，cookie 数据不能超过 4k，同时因为每次 http 请求都会携带 cookie，所以 cookie 只适合保存很小的数据，如会话标识。sessionStorage 和 localStorage 虽然也有存储大小的限制，但比 cookie 大得多，可以达到 5M 或更大。

3、 数据有效期不同，sessionStorage：仅在当前浏览器窗口关闭前有效，自然也就不可能持久保持；localStorage：始终有效，窗口或浏览器关闭也一直保存，因此用作持久数据；cookie 只在设置的 cookie 过期时间之前一直有效，即使窗口或浏览器关闭。

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

●渐进增强、优雅降级

优雅降级：

优雅降级指的是一开始针对一个高版本的浏览器构建页面，先完善所有的功能。然后针对各个不同的浏览器进行测试，修复，保证低级浏览器也有基本功能 就好，低级浏览器被认为 “简陋却无妨 (poor, but passable)” 可以做一些小的调整来适应某个特定的浏览器。但由于它们并非我们所关注的焦点，因此除了修复较大的错误之外，其它的差异将被直接忽略。也就是以高要求，高版本为基准，向下兼容

（一开始就构建完整的功能，然后再针对低版本浏览器进行兼容。）

渐进增强：

渐进增强认为应该专注于内容本身。一开始针对低版本的浏览器构建页面，满足最基本的功能，再针对高级浏览器进行效果，交互，追加各种功能以达到更好用户体验,换句话说，就是以最低要求，实现最基础功能为基本，向上兼容。

（从被所有浏览器支持的基本功能开始，逐步地添加那些只有新式浏览器才支持的功能，向页面添加无害于基础浏览器的额外样式和功能。当浏览器支持时，它们会自动地呈现出来并发挥作用。）

二者区别

1.）如果你采用渐进增强的开发流程，先做一个基本功能版，然后针对各个浏览器进行渐进增加，增加各种功能。相对于优雅降级来说，开发周期长，初期投入资金大。 你想一下不可能拿个基本功能版给客户看呀，多寒酸，搞不好一气之下就不找你做了，然后就炸了。但是呢，也有好处，就是提供了较好的平台稳定性，维护起来资金小， 长期来说降低开发成本。
2.）那采用优雅降级呢，这样可以在较短时间内开发出一个只用于一个浏览器的完整功能版，然后就可以拿给 PM 找客户谈呀，可以拿去测试，市场试水呀，对于功能尚未确定的 产品，优雅降级不失为一种节约成本的方法。

从技术出发

前缀 CSS3（-webkit- / -moz- / -o-*）和正常 CSS3 在浏览器中的支持情况是这样的：

1.很久以前：浏览器前缀 CSS3 和正常 CSS3 都不支持；

2.不久之前：浏览器只支持前缀 CSS3，不支持正常 CSS3；

3.现在：浏览器既支持前缀 CSS3，又支持正常 CSS3；

4.未来：浏览器不支持前缀 CSS3，仅支持正常 CSS3.

```
.transition { /*渐进增强写法*/
-webkit-border-radius:30px 10px;
-moz-border-radius:30px 10px;
border-radius:30px 10px;
}
.transition { /*优雅降级写法*/
border-radius:30px 10px;
-moz-border-radius:30px 10px;
-webkit-border-radius:30px 10px;
}
```

按理说这两种写法效果应该是一样的，但是我们现在浏览器停留在操蛋的第三阶段，也就是现在，既支持前缀写法，又支持正常写法，这样就要出问题了。

当属性超过一个参数值的时候，不同属性产生的作用是不一样的!

点击链接查看效果（https://www.zhangxinxu.com/study/201009/css3-properties-order.html）

可以看到，采用优雅降级的写法，如果一个浏览器同时支持前缀写法和正常写法，后面的旧版浏览器样式就覆盖了新版样式，出现一些奇怪的问题 ，但是用渐进增强的写法就不存在这个问题。这种属性不止 border-radius 一个，所以为了避免这个不必要的错误，建议大家都采用渐进增强的写法。

原文链接：https://blog.csdn.net/jnshu_it/article/details/77016996

●浏览器兼容问题:

不同浏览器的标签默认的外补丁和内补丁不同.

CCS 里: *{margin:0; padding:0}

块属性标签 float 后，又有横行的 margin 情况下，在 IE6 显示 margin 比设置的大会使得 ie6 后面的一块被顶到下一行.

在 float 的标签样式中加入 display: inline; 将其转化为行内属性

设置较小高度标签（一般小于 10px），在 IE6，IE7，中高度超出自己设置高度

给超出高度的标签设置 overflow:hidden;或者设置行高 line-height 小于你设置的高度

原因: ie8 之前的浏览器都会给标签一个最小默认的行高的高度. 即使标签是空的,这个标签的高度还是会达到默认的行高.

图片默认有间距。几个 img 标签放在一起的时候，有些浏览器会有默认的间距，加了问题一中提到的通配符也不起作用。

使用 float 属性为 img 布局

标签最低高度设置 min-height 不兼容

因为 min-height 本身就是一个不兼容的 CSS 属性，所以设置 min-height 时不能很好的被各个浏览器兼容

如果我们要设置一个标签的最小高度 200px，需要进行的设置为：{min-height:200px; height:auto !important; height:200px; overflow:visible;}

IE ol 的序号全为 1，不递增

li 设置样式{display: list-item}

ie6,7 不支持 display:inline-block

设置 inline 并触法 haslayout

display:inline-block; *display:inline; *zoom:1

IE9 以下浏览器不能使用 opacity

Opacity:0.5;

Filter:alfha(opacity=50);

filter：progid：DXImageTransform.Microsoft.Alfha(style=0,opacity=50);

cursor：hand 显示手型在 safari 上不支持

统一使用 cursor：pointer

event.srcElement 问题

问题说明：IE 下，event 对象有 srcElement 属性，但是没有 target 属性；Firefox 下，event 对象有 target 属性，但是没有 srcElement 属性。

使用 srcObj = event.srcElement?event.srcElement:event.target;

事件绑定

IE:dom.attachEvent();

其他浏览器：dom.addEventListener();

操作 tr 的 html

在 ie9 以下，不能操作 tr 的 innerHTML

ajax 略有不同

IE：ActiveXObject

其他：xmlHttpReuest

更多可以参考 https://www.jianshu.com/p/6afd596440bb

●前端性能优化

先简单概括几条，下面比较详细，有空可以看看研究研究。

静态资源压缩有关：

压缩 css、js 代码，压缩图片等资源，减少 http 请求

网络加速连接有关：

使用 cdn 加速

开启 dns 域解析

http 使用 keep-alive 或 presistent 建立持久连接，减少 http 请求次数

http 开启管线化

http 开启 GZIP 压缩技术

优化资源加载：

合理放置资源加载位置。比如：

1、CSS 文件放在 head 中，先外链，后本页

2、JS 文件放在 body 底部，先外链，后本页

3、body 中间尽量不写 style 标签和 script 标签

Js 异步加载 async 或者 defer

构建模块化，按需加载

页面渲染相关：

减少 dom 操作，减少页面重绘重排，避免回流

使用缓存

Js 使用事件代理

及时清理环境，垃圾回收

其他

使用 webpack、gulp 或 grunt 技术

以下为较为详细介绍：

减少请求数量

【合并】

如果不进行文件合并，有如下 3 个隐患

1、文件与文件之间有插入的上行请求，增加了 N-1 个网络延迟

2、受丢包问题影响更严重

3、经过代理服务器时可能会被断

但是，文件合并本身也有自己的问题

1、首屏渲染问题

2、缓存失效问题

所以，对于文件合并，有如下改进建议

1、公共库合并

2、不同页面单独合并

【图片处理】

1、雪碧图

CSS 雪碧图是以前非常流行的技术，把网站上的一些图片整合到一张单独的图片中，可以减少网站的 HTTP 请求数量，但是当整合图片比较大时，一次加载比较慢。随着字体图片、SVG 图片的流行，该技术渐渐退出了历史舞台

2、Base64

将图片的内容以 Base64 格式内嵌到 HTML 中，可以减少 HTTP 请求数量。但是，由于 Base64 编码用 8 位字符表示信息中的 6 个位，所以编码后大小大约比原始值扩大了 33%

3、 使用字体图标来代替图片

【减少重定向】

尽量避免使用重定向，当页面发生了重定向，就会延迟整个 HTML 文档的传输。在 HTML 文档到达之前，页面中不会呈现任何东西，也没有任何组件会被下载，降低了用户体验

如果一定要使用重定向，如 http 重定向到 https，要使用 301 永久重定向，而不是 302 临时重定向。因为，如果使用 302，则每一次访问 http，都会被重定向到 https 的页面。而永久重定向，在第一次从 http 重定向到 https 之后 ，每次访问 http，会直接返回 https 的页面

【使用缓存】

使用 cach-control 或 expires 这类强缓存时，缓存不过期的情况下，不向服务器发送请求。强缓存过期时，会使用 last-modified 或 etag 这类协商缓存，向服务器发送请求，如果资源没有变化，则服务器返回 304 响应，浏览器继续从本地缓存加载资源；如果资源更新了，则服务器将更新后的资源发送到浏览器，并返回 200 响应

【不使用 CSS @import】

CSS 的@import 会造成额外的请求

【避免使用空的 src 和 href】

a 标签设置空的 href，会重定向到当前的页面地

form 设置空的 method，会提交表单到当前的页面地址

减小资源大小

【压缩】

1、HTML 压缩

HTML 代码压缩就是压缩在文本文件中有意义，但是在 HTML 中不显示的字符，包括空格，制表符，换行符等

2、CSS 压缩

CSS 压缩包括无效代码删除与 CSS 语义合并

3、JS 压缩与混乱

JS 压缩与混乱包括无效字符及注释的删除、代码语义的缩减和优化、降低代码可读性，实现代码保护

4、图片压缩

针对真实图片情况，舍弃一些相对无关紧要的色彩信息

【webp】

在安卓下可以使用 webp 格式的图片，它具有更优的图像数据压缩算法，能带来更小的图片体积，同等画面质量下，体积比 jpg、png 少了 25%以上，而且同时具备了无损和有损的压缩模式、Alpha 透明以及动画的特性

【开启 gzip】

HTTP 协议上的 GZIP 编码是一种用来改进 WEB 应用程序性能的技术。大流量的 WEB 站点常常使用 GZIP 压缩技术来让用户感受更快的速度。这一般是指 WWW 服务器中安装的一个功能，当有人来访问这个服务器中的网站时，服务器中的这个功能就将网页内容压缩后传输到来访的电脑浏览器中显示出来。一般对纯文本内容可压缩到原大小的 40%

优化网络连接

Web 前端性能优化之 CDN 加速

1、什么是 CDN?

在介绍 CDN 加速之前，我们先来简单的了解一下什么是 CDN? CDN（Content Delivery Network）即内容分发网络，其基本思路是尽可能的避开互联网上有可能影响数据传输速度和稳定性的瓶颈和环节，使内容传输更快、更稳定。通过在网络各处放置节点服务器所构成的现有互联网基础之上的一层虚拟网络，CDN 系统能够实时的根据网络流量和各节点的连接，负载状况以及用户的距离和响应时间等综合信息将用户的请求重新导向离用户最近的服务节点上。其目的就是使用户能够就近的获取请求数据，解决网络访问拥挤状况，提高用户访问系统的响应时间。

2、CDN 加速

CDN 的本质仍然是一个缓存，而且将数据缓存在里用户最近的地方，使用户以最快的速度获取数据，即网络访问第一跳。

由于 CDN 是部署在网络运营商的机房，这些运营商又是终端用户网络的提供商，因此用户请求的第一跳就到达 CDN 服务器，当 CDN 服务器中缓存有用户请求的数据时，就可以从 CDN 直接返回给客户端浏览器，最短路径的

返回响应，加快用户的访问速度，减少数据中心的负载压力。

CDN 能够缓存一般的 CSS，js 图片等静态资源文件，而且这些文件的访问频率很高，将其缓存在 CDN 可以极大的提高网站的访问速度。

但是由于 CDN 是部署在网络运营商的机房，所以在一般的网站中都很少用 CDN 加速。

【使用 CDN】

CDN 全称是 Content Delivery Network，即内容分发网络，它能够实时地根据网络流量和各节点的连接、负载状况以及到用户的距离和响应时间等综合信息将用户的请求重新导向离用户最近的服务节点上。其目的是使用户可就近取得所需内容，解决 Internet 网络拥挤的状况，提高用户访问网站的响应速度

【使用 DNS 预解析】

当浏览器访问一个域名的时候，需要解析一次 DNS，获得对应域名的 ip 地址。在解析过程中，按照浏览器缓存、系统缓存、路由器缓存、ISP(运营商)DNS 缓存、根域名服务器、顶级域名服务器、主域名服务器的顺序，逐步读取缓存，直到拿到 IP 地址

DNS Prefetch，即 DNS 预解析就是根据浏览器定义的规则，提前解析之后可能会用到的域名，使解析结果缓存到系统缓存中，缩短 DNS 解析时间，来提高网站的访问速度

方法是在 head 标签里面写上几个 link 标签

```
<link rel="dns-prefecth" href="https://www.google.com">
<link rel="dns-prefecth" href="https://www.google-analytics.com">
```

对以上几个网站提前解析 DNS，由于它是并行的，不会堵塞页面渲染，这样可以缩短资源加载的时间

【并行连接】

由于在 HTTP1.1 协议下，chrome 每个域名的最大并发数是 6 个。使用多个域名，可以增加并发数

【持久连接】

使用 keep-alive 或 presistent 来建立持久连接，持久连接降低了时延和连接建立的开销，将连接保持在已调谐状态，而且减少了打开连接的潜在数量

【管道化连接】

在 HTTP2 协议中，可以开启管道化连接，即单条连接的多路复用，每条连接中并发传输多个资源，这里就不需要添加域名来增加并发数了

优化资源加载

【资源加载位置】

通过优化资源加载位置，更改资源加载时机，使尽可能快地展示出页面内容，尽可能快地使功能可用

1、CSS 文件放在 head 中，先外链，后本页

2、JS 文件放在 body 底部，先外链，后本页

3、处理页面、处理页面布局的 JS 文件放在 head 中，如 babel-polyfill.js 文件、flexible.js 文件

4、body 中间尽量不写 style 标签和 script 标签

【资源加载时机】

1、异步 script 标签

defer: 异步加载，在 HTML 解析完成后执行。defer 的实际效果与将代码放在 body 底部类似

async: 异步加载，加载完成后立即执行

2、模块按需加载

在 SPA 等业务逻辑比较复杂的系统中，需要根据路由来加载当前页面需要的业务模块

按需加载，是一种很好的优化网页或应用的方式。这种方式实际上是先把代码在一些逻辑断点处分离开，然后在一些代码块中完成某些操作后，立即引用或即将引用另外一些新的代码块。这样加快了应用的初始加载速度，减轻了它的总体体积，因为某些代码块可能永远不会被加载

webpack 提供了两个类似的技术，优先选择的方式是使用符合 ECMAScript 提案 的 import() 语法。第二种则是使用 webpack 特定的 require.ensure

3、使用资源预加载 preload 和资源预读取 prefetch

preload 让浏览器提前加载指定资源，需要执行时再执行，可以加速本页面的加载速度

prefetch 告诉浏览器加载下一页面可能会用到的资源，可以加速下一个页面的加载速度

4、资源懒加载与资源预加载

资源延迟加载也称为懒加载，延迟加载资源或符合某些条件时才加载某些资源

资源预加载是提前加载用户所需的资源，保证良好的用户体验

资源懒加载和资源预加载都是一种错峰操作，在浏览器忙碌的时候不做操作，浏览器空间时，再加载资源，优化了网络性能

减少重绘回流

【样式设置】

1、避免使用层级较深的选择器，或其他一些复杂的选择器，以提高 CSS 渲染效率

2、避免使用 CSS 表达式，CSS 表达式是动态设置 CSS 属性的强大但危险方法，它的问题就在于计算频率很快。不仅仅是在页面显示和缩放时，就是在页面滚动、乃至移动鼠标时都会要重新计算一次

3、元素适当地定义高度或最小高度，否则元素的动态内容载入时，会出现页面元素的晃动或位置，造成回流

4、给图片设置尺寸。如果图片不设置尺寸，首次载入时，占据空间会从 0 到完全出现，上下左右都可能位移，发生回流

5、不要使用 table 布局，因为一个小改动可能会造成整个 table 重新布局。而且 table 渲染通常要 3 倍于同等元素时间

6、能够使用 CSS 实现的效果，尽量使用 CSS 而不使用 JS 实现

【渲染层】

1、此外，将需要多次重绘的元素独立为 render layer 渲染层，如设置 absolute，可以减少重绘范围

2、对于一些进行动画的元素，使用硬件渲染，从而避免重绘和回流

【DOM 优化】

1、缓存 DOM

const div = document.getElementById('div')

由于查询 DOM 比较耗时，在同一个节点无需多次查询的情况下，可以缓存 DOM

2、减少 DOM 深度及 DOM 数量

HTML 中标签元素越多，标签的层级越深，浏览器解析 DOM 并绘制到浏览器中所花的时间就越长，所以应尽可能保持 DOM 元素简洁和层级较少。

3、批量操作 DOM

由于 DOM 操作比较耗时，且可能会造成回流，因此要避免频繁操作 DOM，可以批量操作 DOM，先用字符串拼接完毕，再用 innerHTML 更新 DOM

4、批量操作 CSS 样式

通过切换 class 或者使用元素的 style.csstext 属性去批量操作元素样式

5、在内存中操作 DOM

使用 DocumentFragment 对象，让 DOM 操作发生在内存中，而不是页面上

6、DOM 元素离线更新

对 DOM 进行相关操作时，例、appendChild 等都可以使用 Document Fragment 对象进行离线操作，带元素“组装”完成后再一次插入页面，或者使用 display:none 对元素隐藏，在元素“消失”后进行相关操作

7、DOM 读写分离

浏览器具有惰性渲染机制，连接多次修改 DOM 可能只触发浏览器的一次渲染。而如果修改 DOM 后，立即读取 DOM。为了保证读取到正确的 DOM 值，会触发浏览器的一次渲染。因此，修改 DOM 的操作要与访问 DOM 分开进行

8、事件代理

事件代理是指将事件监听器注册在父级元素上，由于子元素的事件会通过事件冒泡的方式向上传播到父节点，因此，可以由父节点的监听函数统一处理多个子元素的事件

利用事件代理，可以减少内存使用，提高性能及降低代码复杂度

9、防抖和节流

使用函数节流（throttle）或函数去抖（debounce），限制某一个方法的频繁触发

10、及时清理环境

及时消除对象引用，清除定时器，清除事件监听器，创建最小作用域变量，可以及时回收内存

性能更好的 API

1、用对选择器

选择器的性能排序如下所示，尽量选择性能更好的选择器

复制代码

复制代码

id 选择器（#myid）

类选择器（.myclassname）

标签选择器（div,h1,p）

相邻选择器（h1+p）

子选择器（ul > li）

后代选择器（li a）

通配符选择器（*）

属性选择器（a[rel="external"]）

伪类选择器（a:hover,li:nth-child）

复制代码

复制代码

2、使用 requestAnimationFrame 来替代 setTimeout 和 setInterval

希望在每一帧刚开始的时候对页面进行更改，目前只有使用 requestAnimationFrame 能够保证这一点。使用 setTimeout 或者 setInterval 来触发更新页面的函数，该函数可能在一帧的中间或者结束的时间点上调

用，进而导致该帧后面需要进行的事情没有完成，引发丢帧

3、使用 IntersectionObserver 来实现图片可视区域的懒加载

传统的做法中，需要使用 scroll 事件，并调用 getBoundingClientRect 方法，来实现可视区域的判断，即使使用了函数节流，也会造成页面回流。使用 IntersectionObserver，则没有上述问题

4、使用 web worker

客户端 javascript 一个基本的特性是单线程：比如，浏览器无法同时运行两个事件处理程序，它也无法在一个事件处理程序运行的时候触发一个计时器。Web Worker 是 HTML5 提供的一个 javascript 多线程解决方案，可以将一些大计算量的代码交由 web Worker 运行，从而避免阻塞用户界面，在执行复杂计算和数据处理时，这个 API 非常有用

但是，使用一些新的 API 的同时，也要注意其浏览器兼容性

webpack 优化

【打包公共代码】

使用 CommonsChunkPlugin 插件，将公共模块拆出来，最终合成的文件能够在最开始的时候加载一次，便存到缓存中供后续使用。这会带来速度上的提升，因为浏览器会迅速将公共的代码从缓存中取出来，而不是每次访问一个新页面时，再去加载一个更大的文件

webpack 4 将移除 CommonsChunkPlugin，取而代之的是两个新的配置项 optimization.splitChunks 和 optimization.runtimeChunk

通过设置 optimization.splitChunks.chunks: "all" 来启动默认的代码分割配置项

【动态导入和按需加载】

webpack 提供了两种技术通过模块的内联函数调用来分离代码，优先选择的方式是，使用符合 ECMAScript 提案 的 import() 语法。第二种，则是使用 webpack 特定的 require.ensure

【剔除无用代码】

tree shaking 是一个术语，通常用于描述移除 JavaScript 上下文中的未引用代码(dead-code)。它依赖于 ES2015 模块系统中的静态结构特性，例如 import 和 export。这个术语和概念实际上是兴起于 ES2015 模块打包工具 rollup

JS 的 tree shaking 主要通过 uglifyjs 插件来完成，CSS 的 tree shaking 主要通过 purify CSS 来实现的

【长缓存优化】

1、将 hash 替换为 chunkhash，这样当 chunk 不变时，缓存依然有效

2、使用 Name 而不是 id

每个 module.id 会基于默认的解析顺序(resolve order)进行增量。也就是说，当解析顺序发生变化，ID 也会随之改变

下面来使用两个插件解决这个问题。第一个插件是 NamedModulesPlugin，将使用模块的路径，而不是数字标识符。虽然此插件有助于在开发过程中输出结果的可读性，然而执行时间会长一些。第二个选择是使用 HashedModuleIdsPlugin，推荐用于生产环境构建

【公用代码内联】

使用 html-webpack-inline-chunk-plugin 插件将 mainfest.js 内联到 html 文件中

●浏览器加载文件(repaint/reflow)

文件加载顺序

浏览器加载页面上引用的 CSS、JS 文件、图片时，是按顺序从上到下加载的，每个浏览器可以同时下载文件的个数不同，因此经常有网站将静态文件放在不同的域名下，这样可以加快网站打开的速度。

ES6（ECMASciprt 6）

（ES6 推荐学习下阮一峰的 ES6 入门教程 http://es6.ruanyifeng.com）

●Var、let、const

Var 有变量声明提升这一概念。作用域只有全局作用域和函数作用域，没有块级作用域，var 可重复声明变量。

Let 在 es6 中没有变量提升这一概念，必须先声明在调用，否则报错。let 在 es6 中有块级作用域，出了作用域无法访问。Let 不可重复声明变量，存在暂时性死区。

Const 是 es6 用于声明只读常量，一旦声明不能改变。必须直接声明并且立即初始化，不能先声明，再初始化。同 let 一样，没有变量提升这一概念，只能在块级作用域内有效，存在暂时性死区；const 不可重复声明变量。

正确示例：Const PI = 3.14;　错误示例：const PI; PI = 3.14;

●解构赋值

数组的解构赋值

对象的解构赋值

●字符串的扩展

模板字符串

反斜杠``使用，${变量}使用等

字符创新增方法

比如 includes()、startsWidth()、endsWidth()、padStart()、padEnd()、repeat()、trimStart()、trimEnd()等

●正则扩展

●数值扩展

IsFinite()、isNaN()、
parseInt()、parseFloat()移值到 Number 对象身上。
Number.isInteger() 判断一个值是否为整数

Math 对象扩展：

Math.trunc()去除小数部分，取整数
Math.sign()判断一个数到底是正数、负数、还是零。对于非数值，会先将其转换为数值。

参数为正数，返回+1；
参数为负数，返回-1；
参数为 0，返回 0；
参数为-0，返回-0;
其他值，返回 NaN。

.....

●函数扩展

函数参数默认值

ES5 函数参数不允许有默认值，ES6 允许默认赋值

箭头函数

●数组扩展

扩展运算符

Array.from() 将两类对象转换为数组（即类似数组的对象（array-like object）和可遍历（iterable）的对象（包括 ES6 新增的数据结构 Set 和 Map））

Array.of() 将一组值转换为数组

Array.of(1,2,3,4);// [1,2,3,4]

entries()、keys()、values()用于遍历数组。可以用 for...of 循环进行遍历，唯一的区别是 keys()是对键名的遍历、values()是对键值的遍历，entries()是对键值对的遍历。

Includes() 返回一个布尔值，表示某个数组是否包含给定的值，与字符串的 includes 方法类似

●对象扩展

属性的简洁表达式

键值一样时，可以缩写为一个。

```
Let obj = {pro}等同于
Let obj = { pro:” pro”,}
```

属性名表达式

允许属性名放在中括号内

```
Let obj={
    [name]:” xiaoming”,
}
```

属性的遍历

ES6 一共有 5 种方法可以遍历对象的属性。

（1）for...in

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）。

（2）Object.keys(obj)

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

（3）Object.getOwnPropertyNames(obj)

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

（4）Object.getOwnPropertySymbols(obj)

Object.getOwnPropertySymbols 返回一个数组，包含对象自身的所有 Symbol 属性的键名。

（5）Reflect.ownKeys(obj)

Reflect.ownKeys 返回一个数组，包含对象自身的所有键名，不管键名是 Symbol 或字符串，也不管是否可枚举。

以上的 5 种方法遍历对象的键名，都遵守同样的属性遍历的次序规则。

首先遍历所有数值键，按照数值升序排列。
其次遍历所有字符串键，按照加入时间升序排列。
最后遍历所有 Symbol 键，按照加入时间升序排列。

Super 关键字

this 关键字总是指向函数所在的当前对象，ES6 又新增了另一个类似的关键字 super，指向当前对象的原型对象。

对象新增方法：

Object.is()
Object.assign()
Object.getOwnPropertyDescriptors()
__proto__属性，Object.setPrototypeOf()，Object.getPrototypeOf()
Object.keys()，Object.values()，Object.entries()
Object.fromEntries()

●Symbol

ES6 引入了一种新的原始数据类型 Symbol，表示独一无二的值。它是 JavaScript 语言的第七种数据类型，第六种基本数据类型

- Set 和 Map
- Proxy
- Reflect
- Promise
- Iterator 和 for...of
- Generator
- Async
- Class
- Module

TS（TypeScript）

前端三大框架（react、vue、angular）

（这部分根据自身技术找资料学吧，能力强的最好研究下源码）

VUE

●Vue 介绍（了解）

Vue 简介：

Vue.js 是一个轻巧、高性能、可组件化的 MVVM 库，同时拥有非常容易上手的 API；

Vue.js 是一个构建数据驱动的 Web 界面的库。Vue.js 是一套构建用户界面的 渐进式框架。与其他重量级框架不同的是，Vue 采用自底向上增量开发的设计。Vue 的核心库只关注视图层，并且非常容易学习，非常容易与其它库或已有项目整合。另一方面，Vue 完全有能力驱动采用单文件组件和 Vue 生态系统支持的库开发的复杂单页应用。数据驱动+组件化的前端开发。简而言之：Vue.js 是一个构建数据驱动的 web 界面的渐进式框架。Vue.js 的目标是通过尽可能简单的 API 实现响应的数据绑定和组合的视图组件。核心是一个响应的数据绑定系统。

Vue 特点：

简洁：页面由 HTML 模板+Json 数据+Vue 实例组成

数据驱动：自动计算属性和追踪依赖的模板表达式

组件化：用可复用、解耦的组件来构造页面

轻量：代码量小，不依赖其他库

快速：精确有效批量 DOM 更新

模板友好：可通过 npm，yarn，bower 等多种方式安装，很容易融入

●请说出 vue.cli 项目中 src 目录每个文件夹和文件的用法?

assets 文件夹是放静态资源；components 是放组件；router 是定义路由相关的配置; app.vue 是一个应用主组件；main.js 是入口文件。

●什么是 MVVM

MVVM 是 Model-View-ViewModel 的缩写。

Model 代表数据模型，主要负责数据的提供。提供业务逻辑的数据结构（比如，实体类），提供数据的获取，（比如，从本地数据库或远程网络获取数据），提供数据存储

View 代表 ui 组件，负责界面的展示，不涉及任何业务逻辑处理，他持有 viewModel 层的引用，当需要进行业务逻辑处理事通知 viewModel 层。

ViewModel 监听模型数据的改变和控制视图行为、处理用户交互，简单理解就是一个同步 View 和 Model 的对象，连接 Model 和 View。在 MVVM 架构下，View 和 Model 之间并没有直接的联系，而是通过 ViewModel 进行交互，Model 和 ViewModel 之间的交互是双向的， 因此 View 数据的变化会同步到 Model 中，而 Model 数据的变化也会立即反应到 View 上。ViewModel 通过双向数据绑定把 View 层和 Model 层连接了起来，而 View 和 Model 之间的同步工作完全是自动的，无需人为干涉，因此开发者只需关注业务逻辑，不需要手动操作 DOM，不需要关注数据状态的同步问题，复杂的数据状态维护完全由 MVVM 来统一管理。

●什么是虚拟 DOM 和 DIFF 算法

虚拟 dom

DOM 的本质是什么：

浏览器中的概念，用 JS 对象来表示 页面上的元素，并提供了操作 DOM 对象的 API；

什么是虚拟 DOM：是框架中的概念，是程序员 用 JS 对象来模拟页面上的 DOM 和 DOM 嵌套；

为什么要实现虚拟 DOM（虚拟 DOM 的目的）：为了实现页面中， DOM 元素的高效更新

DOM 和虚拟 DOM 的区别：

DOM：浏览器中，提供的概念；用 JS 对象，表示页面上的元素，并提供了操作元素的 API；

虚拟 DOM：是框架中的概念；而是开发框架的程序员，手动用 JS 对象来模拟 DOM 元素和嵌套关系；

本质：用 JS 对象，来模拟 DOM 元素和嵌套关系；

目的：就是为了实现页面元素的高效更新；

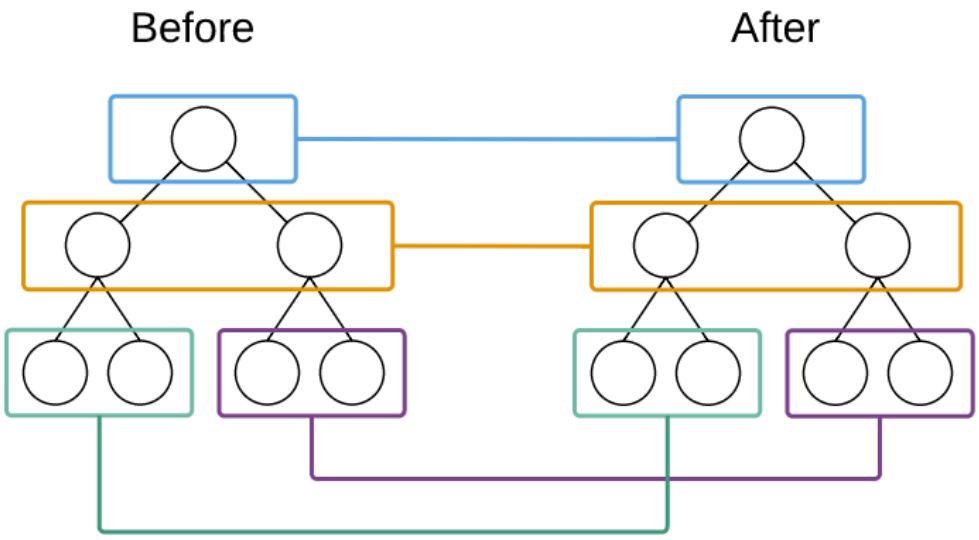
Diff 算法

tree diff: 新旧 DOM 树，逐层对比的方式，就叫做 tree diff,每当我们从前到后，把所有层的节点对比完后，必然能够找到那些 需要被更新的元素；

component diff: 在对比每一层的时候，组件之间的对比，叫做 component diff;当对比组件的时候，如果两个组件的类型相同，则暂时认为这个组件不需要被更新，如果组件的类型不同，则立即将旧组件移除，新建一个组件，替换到被移除的位置；

element diff: 在组件中，每个元素之间也要进行对比，那么，元素级别的对比，叫做 element diff；

key: key 这个属性，可以把 页面上的 DOM 节点 和 虚拟 DOM 中的对象，做一层关联关系；



虚拟 dom 与 diff 算法总体概述：

框架通过 js 模拟原有 dom 对象来创建一个 dom 结构，也就是虚拟 dom，把虚拟 dom 转换成真正 dom 插入页面中，当数据发生变化时，原有的 dom 不会发生修改，全都作用到虚拟 dom 身上，通过比较新旧两个 dom 树的差异，比较出差异对象，把差异对象一次性直接全部应用到真正 dom 身上，从而实现只更新变化的东西，一次性完成更新，减少 dom 操作，减少回流和重绘，提升页面性能。

●Vue 生命周期函数及应用场景

beforeCreate（创建前）：

实例初始化之前，组件被创建时期，这个时候的 el，data，message 都是 underfined。

应用场景：可以在此时加一些 loading 效果，在 created 时进行移除

created（创建后）：

实例创建完成后，data、methods 被初始化。

应用场景：结束 loading 事件；需要异步请求数据的方法可以在此时执行。推荐这个时候发送请求数据，尤其是返回的数据与绑定事件有关时。此时的数据不会在 dom 元素渲染。

beforeMount（载入前）：

在挂载开始之前被调用，相关的 render 函数首次被调用。编译模板，把 data 里面的数据和模板生成 html，模板已经在内存中编辑完成了，但是尚未把 模板渲染到页面中

应用场景：也可以发送数据请求

mounted（载入后）：

内存中的模板，完成模板中的 html 渲染到 html 页面中，用户可以看到渲染完后的页面了。此过程中进行 ajax 交互。

应用场景：获取 el 中 DOM 元素，进行 DOM 操作；如果返回的数据操作依赖 DOM 完成，推荐这个时候发送数据请求

beforeUpdate（更新前）：

页面中的显示的数据，还是旧的，此时 data 数据是最新的，页面尚未和 最新的数据保持同步

应运用场景：挂载完成之前访问现有 DOM ，比如手动移除已添加的事件监听器；也可以进一步修改数据

updated（更新后）：

页面和 data 数据已经保持同步了，都是最新的。然而在大多数情况下，应该避免在此期间更改状态，因为这可能会导致更新无限循环。该钩子在服务器端渲染期间不被调用。

应用场景：任何数据的更新，如果要做统一的业务逻辑处理

beforeDestroy（销毁前）：

在实例销毁之前调用。实例仍然完全可用。

应用场景：可以做一个确认停止事件或删除的确认框

destroyed（销毁后）：

在实例销毁之后调用。调用后，所有的事件监听器会被移除，所有的子实例也会被销毁。该钩子在服务器端渲染期间不被调用。

应用场景：提示已删除

参考网站（详解）：https://segmentfault.com/a/1190000019743049

1.什么是 vue 生命周期?

Vue 实例从创建到销毁的过程，就是生命周期。从开始创建、初始化数据、编译模板、挂载 Dom→渲染、更新→渲染、销毁等一系列过程，称之为 Vue 的生命周期。

2.vue 生命周期的作用是什么?

它的生命周期中有多个事件钩子，让我们在控制整个 Vue 实例的过程时更容易形成好的逻辑。

3.vue 生命周期总共有几个阶段?

它可以总共分为 8 个阶段：创建前/后，载入前/后,更新前/后,销毁前/销毁后。

4.第一次页面加载会触发哪几个钩子?

会触发 下面这几个 beforeCreate, created, beforeMount, mounted 。

5.DOM 渲染在 哪个周期中就已经完成?

在 mounted 中就已经完成了。

6.created 和 mouted 有什么不同?

created:在模板渲染成 html 前调用，即通常初始化某些属性值，然后再渲染成视图。

mounted:在模板渲染成 html 后调用,通常是初始化页面完成后,再对 html 的 dom 节点进行一些需要的操作。

●为什么 vue 中 data 必须是个函数

对象（object）为引用类型，当重用组件时，由于数据对象都指向同一个 data 对象，当在一个组件中修改 data 时，其他重用的组件中的 data 会同时被修改；而使用返回对象的函数，由于每次返回的都是一个新对象（Object 的实例），引用地址不同，则不会出现这个问题。

●v-for 为什么用 key

当 Vue 正在更新使用 v-for 渲染的元素列表时，它默认使用“就地更新”的策略。如果数据项的顺序被改变，Vue 将不会移动 DOM 元素来匹配数据项的顺序，而是就地更新每个元素，并且确保它们在每个索引位置正确渲染。

这个默认的模式是高效的，但是只适用于不依赖子组件状态或临时 DOM 状态（例如：表单输入值）的列表渲染输出。

为了给 Vue 一个提示，以便它能跟踪每个节点的身份，从而重用和重新排序现有元素，你需要为每项提供一个唯一 key 属性。所以 key 的作用可以理解为高效更新虚拟 dom。

●计算属性 computed 和 watch 区别

computed 计算属性是用来声明式的描述一个值依赖了其它的值。当你在模板里把数据绑定到一个计算属性上时，Vue 会在其依赖的任何值导致该计算属性改变时更新 DOM。这个功能非常强大，它可以让你的代码更加声明式、数据驱动并且易于维护。

conputed 是一个计算属性,类似于过滤器,对绑定到 view 的数据进行处理。

watch 是属性监听器，一般用来监听属性的变化（也可以用来监听计算属性函数），并做一些逻辑。监听的是你定义的变量，当你定义的变量的值发生变化时，调用对应的方法。

Watch 是一个观察的动作。

computed:

如果一个数据需要经过复杂计算就用 computed；当一个属性受多个属性影响的时候就需要用到 computed

例子： 购物车商品结算的时候

watch:

当一条数据影响多条数据的时候就需要用 watch；如果一个数据需要被监听并且对数据做一些操作就用 watch

例子：搜索数据

```
<div>{{Name}}</div>
data(){
    return {
        num:0,
        lastname:'',
        firstname:'',
    }
}
//当 num 的值发生变化时，就会调用 num 的方法，方法里面的形参对应的是 num 的新值和旧值
```

```
watch:{
        num:function(val,oldval){
              console.log(val,oldval);
        }
},
//计算属性 computed,计算的是 Name 依赖的值,它不能计算在 data 中已经定义过的变量。
computed:{
      Name:function(){
            return this.firstname+this.lastname;
      }
}
```

computed 和 watch 的区别

computed 特性

1.是计算值，

2.应用：就是简化 tempalte 里面{{}}计算和处理 props 或$emit 的传值

3.具有缓存性，页面重新渲染值不变化,计算属性会立即返回之前的计算结果，而不必再次执行函数

watch 特性

1.是观察的动作，

2.应用：监听 props，$emit 或本组件的值执行异步操作

3.无缓存性，页面重新渲染时值不变化也会执行

大概总结一下，computed 和 watch 的使用场景并不一样，computed 的话是通过几个数据的变化，来影响一个数据，而 watch，则是可以一个数据的变化，去影响多个数据。

●Props 验证和默认值

我们在父组件给子组件传值得时候，为了避免不必要的错误，可以给 prop 的值进行类型设定，让父组件给子组件传值得时候，更加准确，prop 可以传一个数字，一个布尔值，一个数组，一个对象，以及一个对象的所有属性。组件可以为 props 指定验证要求。如果未指定验证要求，Vue 会发出警告比如传一个 number 类型的数据，用 defalt 设置它的默认值，如果验证失败的话就会发出警告。

```
props: {
    visible: {
        default: true,
        type: Boolean,
        required: true
    },
},
```

●Vue 通信：父子组件传值，传递方法

父子组件传值

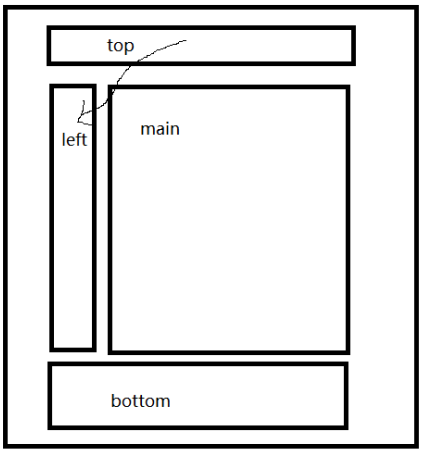
父子组件传值是通过 props 实现的，给父组件添加 v-bind 命令，在 v-bind:parentmsg=”父组件的值” v-bind 后添加自定义的变量，与父组件的值关联，子组件通过添加 props:[parentmsg],属性，后面传入父组件定义的变量即可。

例子：

```
<body>
    <div id="app">
        <com1 v-bind:parentmsg="msg"></com1>
    </div>
  <script>
    var vm = new Vue({
      el: '#app',
      data: {
        msg: '123  啊-父组件中的数据'
      },
      components: {
        com1: {
          data() {
            return {
            }
          },
          template: '<h1 @click="change">这是子组件  --- {{ parentmsg }}</h1>',
          props: ['parentmsg'],
          methods: {
            change() {
              this.parentmsg = '被修改了'
            }
          }
        }
      }
    });
  </script>
</body>
```

兄弟组件传值

首先我的需求是这样的，页面上由 top，left，main，bottom 四个组件构成。需要将 top 中的值传到 left 中。



第一步：创建一个 js 文件，eventBus.js，位置随便放，我是放在了 src 目录下

```
import Vue from 'vue'
```

```
export default new Vue()
```

第二步：在 top 组件中，引入刚才的 js

```
Import  eventBus  from  '@/eventBus.js'
```

然后在 methods 里边定义一个函数

```
methods:{
    changesize(){
        eventBus.$emit('add',this.arg)
    }
}
```

我测试用的是 button 点击触发 changesize 函数，然后将 arg 传出去

第三步：在 left 组件中也先让引入 eventBus.js，然后使用 created 生命周期函数

```
Import  eventBus  from  '@/eventBus.js'
created(){
    eventBus.$on('add',(message)=>{
        //一些操作，message 就是从 top 组件传过来的值
        console.log(message)
    })
},
```

传递方法（父组件传递方法给子组件，子组件通过方法像父组件传值）：

在父组件身上定义@func=’父组件方法’，@后名字自定义，子组件方法内通过调用 this.$emit(‘func’，需要传递的值)，来调用父组件方法，后面的参数传参可以向父组件传值，父组件方法通过形参接受传的值即可。

例子：

```
<body>
  <div id="app">
    <com2 @func="show"></com2>
  </div>

  <template id="tmpl">
    <div>
      <h1>这是子组件</h1>
      <input type="button" value="这是子组件中的按钮 - 点击它，触发 父组件传递过来的 func 方法" @click="myclick">
    </div>
  </template>
  <script>
    var com2 = {
      data() {
        return {
          sonmsg: { name: '小头儿子', age: 6 }
        }
      },
      methods: {
        myclick() {
          this.$emit('func', this.sonmsg)
        }
```

```
    },
  }
  var vm = new Vue({
    el: '#app',
    data: {
      datamsgFormSon:null
    },
    methods: {
      show(data) {
        this.datamsgFormSon = data;
      }
    },
    components: {
      com2,
    },
  });
</script>
</body>
```

●Vue 双向绑定原理（重点）

vue 双向数据绑定是通过 **数据劫持**结合**订阅发布模式**的方式来实现的， 也就是说数据和视图同步，数据发生变化，视图跟着变化，视图变化，数据也随之发生改变；

核心：关于 vue 双向数据绑定，其核心是 Object.defineProperty()方法。

指的是在访问或者修改对象的某个属性时，通过一段代码拦截这个行为，进行额外的操作或者修改返回结果

observer，创建数据监听，并为每个属性建立一个发布类。

Dep 是发布类，维护与该属性相关的订阅实例，当数据发生更新时，会通知所有的订阅实例。

Watcher 是订阅类，注册到所有相关属性的 Dep 发布类中，接受发布类的数据变更通知，通过回调，实现视图的更新。

1.实现一个监听者 Oberver 来劫持并监听所有的属性，一旦有属性发生变化就通知订阅者

2.实现一个订阅者 watcher 来接受属性变化的通知并执行相应的方法，从而更新视图

3.实现一个解析器 compile，可以扫描和解析每个节点的相关指令，并根据初始化模板数据以及初始化相对应的订阅者

vue2.x 使用 Object.defineProperty();

vue3.x 使用 Proxy;

Object.defineProperty():

Object.defineProperty（）来劫持各个属性的 setter，getter，在数据变动时发布消息给订阅者，触发相应监

听回调。当把一个普通 Javascript 对象传给 Vue 实例来作为它的 data 选项时，Vue 将遍历它的属性，用 Object.defineProperty 将它们转为 getter/setter。用户看不到 getter/setter，但是在内部它们让 Vue 追踪依赖，在属性被访问和修改时通知变化。每个组件实例都有相应的 watcher 程序实例，它会在组件渲染的过程中把属性记录为依赖，之后当依赖项的 setter 被调用时，会通知 watcher 重新计算，从而致使它关联的组件得以更新。

vue 的数据双向绑定 将 MVVM 作为数据绑定的入口，整合 Observer，Compile 和 Watcher 三者，通过 Observer 来监听自己的 model 的数据变化，通过 Compile 来解析编译模板指令（vue 中是用来解析 {{}}），最终利用 watcher 搭起 observer 和 Compile 之间的通信桥梁，达到数据变化 —>视图更新；视图交互变化（input）—>数据 model 变更双向绑定效果。

例 1：

```
<body>
    <div id="app">
    <input type="text" id="txt">
    <p id="show"></p>
</div>
</body>
<script type="text/javascript">
    var obj = {}
    Object.defineProperty(obj, 'txt', {
        get: function () {
            return obj
        },
        set: function (newValue) {
            document.getElementById('txt').value = newValue
            document.getElementById('show').innerHTML = newValue
        }
    })
    document.addEventListener('keyup', function (e) {
        obj.txt = e.target.value
    })
</script>
```

例 2：

```
 <input id="in" type="text" onchange="change(this.value)">
<script type="text/javascript">
window.onload = function(){
    document.getElementById('in').value = data.name;
}
var input = document.getElementById('in');
var data = {
};
Object.defineProperty(data,'name',{
    configurable:true,
    get(){
        console.log("数据取值操作");
        return name;
```

```
        },
        set(newVal){
            console.log("数据赋值操作");
            name = newVal;
        },
    });
    function change(value){
        data.name = value;
        console.log(data.name)
    }
    console.log(data.name)
    </script>
```

详细可参考 https://www.cnblogs.com/libin-1/p/6893712.html

●Vue-router 有几种守卫（导航钩子）

“导航”表示路由正在发生改变。正如其名，vue-router 提供的导航守卫主要用来通过跳转或取消的方式守卫导航。有多种机会植入路由导航过程中：全局的，单个路由独享的，或者组件级的。

三种

第一种：全局导航钩子：

全局前置守卫

router.beforeEach(to,from,next)，作用：跳转前进行判断拦截。

全局后置钩子

```
router.afterEach((to, from) => {
  // ...
})
```

全局解析守卫

router.beforeResolve 注册一个全局守卫。这和 router.beforeEach 类似，区别是在导航被确认之前，同时在所有组件内守卫和异步路由组件被解析之后，解析守卫就被调用。

第二种：单个路由独享守卫

```
beforeEnter: (to, from, next) => {
        // ...
}
```

第三种：组件内的守卫；

```
beforeRouteEnter (to, from, next) {
    // 在渲染该组件的对应路由被 confirm 前调用
    // 不！能！获取组件实例 `this`
    // 因为当守卫执行前，组件实例还没被创建
},
beforeRouteUpdate (to, from, next) {
    // 在当前路由改变，但是该组件被复用时调用
    // 举例来说，对于一个带有动态参数的路径 /foo/:id，在 /foo/1 和 /foo/2 之间跳转的时候，
    // 由于会渲染同样的 Foo 组件，因此组件实例会被复用。而这个钩子就会在这个情况下被调用。
    // 可以访问组件实例 `this`
},
beforeRouteLeave (to, from, next) {
```

```
    // 导航离开该组件的对应路由时调用
    // 可以访问组件实例 `this`
}
```

完整的导航解析流程

1、导航被触发。
2、在失活的组件里调用离开守卫。
3、调用全局的 beforeEach 守卫。
4、在重用的组件里调用 beforeRouteUpdate 守卫 (2.2+)。
5、在路由配置里调用 beforeEnter。
6、解析异步路由组件。
7、在被激活的组件里调用 beforeRouteEnter。
8、调用全局的 beforeResolve 守卫 (2.5+)。
9、导航被确认。
10、调用全局的 afterEach 钩子。
11、触发 DOM 更新。
12、用创建好的实例调用 beforeRouteEnter 守卫中传给 next 的回调函数。

可参考：(https://router.vuejs.org/zh/guide/advanced/navigation-guards.html)

●Vue 路由传参

1.使用 query 方法传入的参数使用 this.$route.query 接收
2.使用 params 方式传入的参数使用 this.$route.params 接收

●Vue 的路由模式，实现方式

hash 模式 和 history 模式

hash 模式：在浏览器中符号“#”，#以及#后面的字符称之为 hash，用 window.location.hash 读取；
特点：hash 虽然在 URL 中，但不被包括在 HTTP 请求中；用来指导浏览器动作，对服务端安全无用，hash 不会重加载页面。

hash 模式下:仅 hash 符号之前的内容会被包含在请求中，如 http://www.xxx.com，因此对于后端来说，即使没有做到对路由的全覆盖，也不会返回 404 错误。

history 模式：history 采用 HTML5 的新特性；且提供了两个新方法：pushState（），replaceState（）可以对浏览器历史记录栈进行修改，以及 popState 事件的监听到状态变更。

history 模式:前端的 URL 必须和实际向后端发起请求的 URL 一致，如 http://www.xxx.com/items/id。后端如果缺少对 /items/id 的路由处理，将返回 404 错误。Vue-Router 官网里如此描述：“不过这种模式要玩好，还需要后台配置支持……所以呢，你要在服务端增加一个覆盖所有情况的候选资源：如果 URL 匹配不到任何静态资源，则应该返回同一个 index.html 页面，这个页面就是你 app 依赖的页面。”

●Vuex 是什么，有几种属性

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。

（可以理解为 vue 集中管理数据的一个容器，供所有的组件直接访问调用数据）

State（可以理解为组件的 data，存数据的）

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

在组件中访问 Vuex 中的数据，只能通过 this.$store.state.名称访问

Mutations（可以理解为组件的 methods，存方法，用于操作 state 中数据）

mutations 定义的方法动态修改 Vuex 的 store 中的状态或数据。所有的组件都需要经过 mutations 去操作 vuex 中的数据，不能直接越过 mutations 操作 vuex 数据，否则容易造成数据的紊乱。mutations 相当于 vuex 的一个库管理员，用于对 state 中的数据进行统一操作。

Mutations 内方法的参数列表最多支持两个参数，第一个是 state，用于操作 state 中的数据（state.数据），第二个是组件中通过调用 mutations 方法传递的参数。如果传多个参数，可传递数组，对象。

组件想调用 mutations 的方法，需要用 this.$store.commit（‘方法名’）

getters

类似 vue 的计算属性，主要用来过滤一些数据，可在内部定义方法进行数据的操作。只负对外提供数据。外面接收数据，不能修改。类似组件中的过滤器，把元数据进行包装提供给调用者。也与组件的 computed 比较像，只要 state 中的数据发生变化，那么如果 getters 正好也引用了这个数据，会立即触发 getters 的重新求值。

一般情况下用于对 state 数据进行进一步包装

通过 this.$store.getters.名称调用。

action

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

modules

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

●v-show 和 v-if 相同及不同点

都是用于显示隐藏 dom 元素的

v-show 指令是通过修改元素的 css 的 display 属性让其显示或者隐藏

v-if 指令是通过条件判断直接销毁和创建 DOM 达到让元素显示和隐藏的效果

●keep-alive

keep-alive 是 Vue 内置的一个组件，可以使被包含的组件保留状态，避免重新渲染；或者说是当前组件应用缓存。

在 vue 2.1.0 版本之后，keep-alive 新加入了两个属性: include(包含的组件缓存) 与 exclude(排除的组件不缓存，优先级大于 include）。

举例应用场景:

比如有一个列表组件和一个详情组件，那么用户就会经常执行打开详情=>返回列表=>打开详情...这样的话列表和详情都是一个频率很高的页面，那么就可以对列表组件使用<keep-alive></keep-alive>进行缓存、保留状态，这样用户每次返回列表的时候，都能从缓存中快速渲染，而不是重新渲染。

使用方法

```
<keep-alive include='include_components' exclude='exclude_components'>
  <component>
    <!-- 该组件是否缓存取决于 include 和 exclude 属性 -->
  </component>
</keep-alive>
```

参数解释

include - 字符串或正则表达式，只有名称匹配的组件会被缓存

exclude - 字符串或正则表达式，任何名称匹配的组件都不会被缓存

include 和 exclude 的属性允许组件有条件地缓存。二者都可以用“,”分隔字符串、正则表达式、数组。当使用正则或者是数组时，要记得使用 v-bind 。

使用示例

```
<!-- 逗号分隔字符串，只有组件 a 与 b 被缓存。 -->
<keep-alive include="a,b">
  <component></component>
</keep-alive>

<!-- 正则表达式 (需要使用 v-bind，符合匹配规则的都会被缓存) -->
<keep-alive  :include="/a|b/">
  <component></component>
</keep-alive>

<!-- Array (需要使用 v-bind，被包含的都会被缓存) -->
<keep-alive  :include="['a', 'b']">
  <component></component>
</keep-alive>
```

问题：

Keep-alive 生命周期：

activated： 页面第一次进入的时候，钩子触发的顺序是 created->mounted->activated

deactivated： 页面退出的时候会触发 deactivated，当再次前进或者后退的时候只触发 activated

●v-for 与 v-if 优先级

v-for 比 v-if 优先，如果每一次都需要遍历整个数组，将会影响速度，尤其是当只需要渲染很小一部分的时候。

●axios 和 vue-resource

Vue-resource 和 axios 都是 vue 推荐的请求后端的类似于 ajax 的通信工具，具体怎么使用，自己查找相关文档吧。具体用哪一个都可以，看公司了，推荐 axios 吧，毕竟是新东西，vue-resource 新的 vue 不推荐使用了，已不再更新。

vue-resource（vue2.0 前推荐，2.0 后不再更新）

vue-resource 插件具有以下特点：

1，体积小

vue-resource 非常小巧，在压缩以后只有大约 12KB，服务端启用 gzip 压缩后只有 4.5KB 大小，这远比 jQuery 的体积要小得多

2，支持主流的浏览器

和 Vue.js 一样，vue-resource 除了不支持 IE 9 以下的浏览器，其他主流的浏览器都支持。

3，支持 Promise API 和 URI Templates

Promise 是 ES6 的特性，Promise 的中文含义为“先知”，Promise 对象用于异步计算。

URI Templates 表示 URI 模板，有些类似于 ASP.NET MVC 的路由模板。

4，支持拦截器

拦截器是全局的，拦截器可以在请求发送前和发送请求后做一些处理。

拦截器在一些场景下会非常有用，比如请求发送前在 headers 中设置 access_token，或者在请求失败时，提供共通的处理方式。

axios（vue2.0 后推荐使用）

1、 axios 是一个基于 promise 的 HTTP 库，支持 promise 的所有 API；
2、 可以在浏览器中发送 XMLHttpRequest 请求
3、 可以在 node.js 发送 http 请求
4、 它可以拦截请求和响应；
5、 能够取消请求
6、 它可以转换请求数据和响应数据，并对响应回来的内容自动转换为 json 类型的数据；
7、 客户端支持保护安全免受 CSRF/XSRF 攻击

●$route 和$router 区别

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

$router 是'路由实例'对象包括了路由的跳转方法，钩子函数等。

●params 和 query 的区别

query 要用 path 来引入，params 要用 name 来引入，接收参数都是类似的，分别是 this.$route.query.name 和 this.$route.params.name。

url 地址显示：query 更加类似于我们 ajax 中 get 传参，params 则类似于 post，说的再简单一点，前者在浏览器地址栏中显示参数，后者则不显示

注意点：query 刷新不会丢失 query 里面的数据；params 刷新 会 丢失 params 里面的数据。

●$nextTick 的使用

当你修改了 data 的值然后马上获取这个 dom 元素的值，是不能获取到更新后的值，你需要使用$nextTick 这个回调，让修改后的 data 值渲染更新到 dom 元素之后在获取，才能成功。

●Vue 如何兼容 IE 问题

Babel-polyfill 插件

●Vue 初始化页面闪动问题

使用 vue 开发时，在 vue 初始化之前，由于 div 是不归 vue 管的，所以我们写的代码在还没有解析的情况下会容易出现花屏现象，看到类似于{{message}}的字样，虽然一般情况下这个时间很短暂，但是我们还是有必要让解决这个问题的。

v-cloak 指令：

v-cloak 指令和 css 规则如[v-cloak]{display:none}一起用时，这个指令可以隐藏未编译的 Mustache 标签直到实例准备完毕。

v-cloak 指令可以像 css 选择器一样绑定一套 css 样式然后这套 css 会一直生效到实例编译结束。

css 加上如下：

```
[v-cloak]{
        display:none;
}
<div v-cloak>{{ message }}</div>
```

如果没有彻底解决问题，则在根元素加上 style="display: none;" :style="{display: 'block'}"

```
<div class="app" style="display: none;" :style="{display: 'block'}">
      {{message}}
</div>
```

●Router-link 在电脑上有用，在安卓上没反应怎么解决?

Vue 路由在 Android 机上有问题，babel 问题，安装 babel polypill 插件解决。

●页面刷新 vuex 被清空解决办法

1.localStorage 存储到本地再回去

2.重新获取接口获取数据

●如何优化 SPA 应用的首屏加载速度慢的问题

1、将公用的 JS 库通过 script 标签外部引入，减小 app.bundel 的大小，让浏览器并行下载资源文件，提高下载速度；

2、在配置 路由时，页面和组件使用懒加载的方式引入，进一步缩小 app.bundel 的体积，在调用某个组件时再加载对应的 js 文件；

3、加一个首屏 loading 图，提升用户体验；

●vue 等单页面应用及其优缺点

优点：

Vue 的目标是通过尽可能简单的 API 实现响应的数据绑定和组合的视图组件，核心是一个响应的数据绑定系统。MVVM、数据驱动、组件化、轻量、简洁、高效、快速、模块友好。

缺点：

不支持低版本的浏览器，最低只支持到 IE9；不利于 SEO 的优化（如果要支持 SEO，建议通过服务端来进行渲

染组件)；第一次加载首页耗时相对长一些；不可以使用浏览器的导航按钮需要自行实现前进、后退。

●单页面应用和多页面应用区别及优缺点

单页面应用（SPA)：

通俗一点说就是指只有一个主页面的应用，浏览器一开始要加载所有必须的 html, js, css。所有的页面内容都包含在这个所谓的主页面中。但在写的时候，还是会分开写（页面片段），然后在交互的时候由路由程序动态载入，单页面的页面跳转，仅刷新局部资源。多应用于 pc 端。

多页面（MPA)：就是指一个应用中有多个页面，页面跳转时是整页刷新

单页面的优点：

用户体验好，快，内容的改变不需要重新加载整个页面，基于这一点 spa 对服务器压力较小；前后端分离；页面效果会比较炫酷（比如切换页面内容时的专场动画)。

单页面缺点：

不利于 seo；导航不可用，如果一定要导航需要自行实现前进、后退。(由于是单页面不能用浏览器的前进后退功能，所以需要自己建立堆栈管理)；初次加载时耗时多；页面复杂度提高很多。

●vue 怎么实现权限控制

●webpack 结合 vue 修改配置文件

●webpack 和 gulp 和 grunt 三者区别

学过的了解下，没学过的就算了。

总结前端面试几大题型（仅供参考）

1、盒模型加 BFC

2、Js 基础知识，社超特别重视结合实际业务场景出题

3、http 协议及通信

4、浏览器渲染原理以及前端页面加载的流程

5、前端框架及对比

6、性能优化方案（建议从 tcp 请求到接口请求，到页面绘制的整个过程)

7、算法题（一般是加分题，手写代码……，多刷题，多刷题，多刷题)

面试=实力+运气+技巧

多总结，多思考，多沉淀东西。