<https://bailinlin.github.io/>

<https://www.cnblogs.com/fsyz/p/8298921.html>

<https://blog.csdn.net/weixin_37719279/article/details/80961062>

一、dianhua

1. CSS盒模型

## **基本概念**

在 CSS 盒子模型 (Box Model) 规定了元素处理元素的几种方式： **内容、内边距、边框、外边距**。

在 CSS的**标准盒子模型**中，**width 和 height 指的是内容区域**的宽度和高度。增加内边距、边框和外边距不会影响内容区域的尺寸，但是会增加元素框的总尺寸。**IE盒子模型**中，**width 和 height 指的是内容区域+border+padding**的宽度和高度。

## **css如何设置两种模型**

这里用到了CSS3 的属性 box-sizing

/\* 标准模型 \*/

box-sizing:content-box;

/\*IE模型\*/

box-sizing:border-box;

## **JS获取宽高**

通过JS获取盒模型对应的宽和高，有以下几种方法：

为了方便书写，以下用dom来表示获取的HTML的节点。

1.  dom.style.width/height

　　这种方式只能取到dom元素内联样式所设置的宽高，也就是说如果该节点的样式是在style标签中或外联的CSS文件中设置的话，通过这种方法是获取不到dom的宽高的。

 2. dom.currentStyle.width/height

　　这种方式获取的是在页面渲染完成后的结果，就是说不管是哪种方式设置的样式，都能获取到。

　　但这种方式只有IE浏览器支持。

 3. window.getComputedStyle(dom).width/height

　　这种方式的原理和2是一样的，这个可以兼容更多的浏览器，通用性好一些。

 4. dom.getBoundingClientRect().width/height

这种方式是根据元素在视窗中的绝对位置来获取宽高的

空边框盒（译者注：没有内容的边框）会被忽略。如果所有的元素边框都是空边框，那么这个矩形给该元素返回的 width、height 值为0，left、top值为第一个css盒子（按内容顺序）的top-left值。

当计算边界矩形时，会考虑视口区域（或其他可滚动元素）内的滚动操作，也就是说，当滚动位置发生了改变，top和left属性值就会随之立即发生变化（因此，它们的值是相对于视口的，而不是绝对的）。如果你需要获得相对于整个网页左上角定位的属性值，那么只要给top、left属性值加上当前的滚动位置（通过window.scrollX和window.scrollY），这样就可以获取与当前的滚动位置无关的值。

 5.dom.offsetWidth/offsetHeight

　　这个就没什么好说的了，最常用的，也是兼容最好的。

1. box-sizing的应用场

### 1）常规的布局思维是我定好整体的宽高之后，里面元素的整体尺寸会确定下来，后面才是考虑这个元素的内边距以及边框，如果我已经确定好了大体的布局，再去微调边框而导致了整体布局的崩塌，这显然是不科学的，所以这也是我极力推崇前端布局使用box-sizing：border;的原因。

### 2）box-sizing 布局三栏目案例

div{

height:700px;

float:left;

}div.left{

width:25%;

background:red;

}div.cent{

width:50%;

box-sizing:border-box;/\*可以改变元素以使其宽度包含填充\*/

/\* 现在整个元素,包括填充在内,占页面总宽度的50%,所以元素的组合宽度为100%,这全程它们很好地适应于它们的容器.\*/

background:yellow;

padding:0 20px;/\*加了这个会使盒子内容溢出 但是box-sizing很好的自适应了\*/

}div.right{

width:25%;

background:blue;

}

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

<div class="cent"></div>

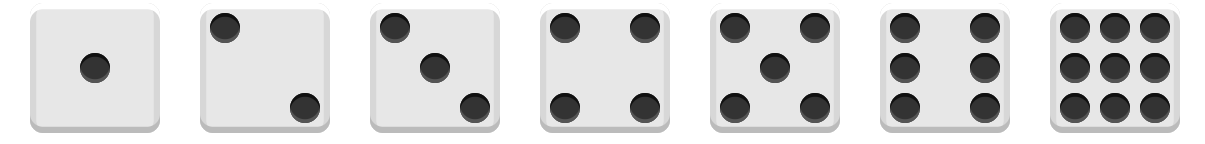
<div class="right"></div>

1. 弹性flex布局

<http://www.ruanyifeng.com/blog/2015/07/flex-grammar.html>

1. 骰子的布局

<https://codepen.io/anon/pen/BPggqE>



CSS：

.first-face {

display: flex;

justify-content: center;

align-items: center;

}

.second-face {

display: flex;

justify-content: space-between;

}

.second-face .pip:nth-of-type(2) {

align-self: flex-end;

}

.third-face {

display: flex;

justify-content: space-between;

}

.third-face .pip:nth-of-type(2) {

align-self: center;

}

.third-face .pip:nth-of-type(3) {

align-self: flex-end;

}

.fourth-face, .sixth-face, .ninth-face {

display: flex;

justify-content: space-between;

}

.fourth-face .column, .sixth-face .column, .ninth-face .column {

display: flex;

flex-direction: column;

justify-content: space-between;

}

.fifth-face {

display: flex;

justify-content: space-between;

}

.fifth-face .column {

display: flex;

flex-direction: column;

justify-content: space-between;

}

.fifth-face .column:nth-of-type(2) {

justify-content: center;

}

/\* OTHER STYLES \*/

\* {

box-sizing: border-box;

}

html, body {

height: 100%;

}

body {

display: flex;

align-items: center;

justify-content: center;

vertical-align: center;

flex-wrap: wrap;

align-content: center;

font-family: 'Open Sans', sans-serif;

background: linear-gradient(top, #222, #333);

}

[class$="face"] {

margin: 16px;

padding: 4px;

background-color: #e7e7e7;

width: 104px;

height: 104px;

object-fit: contain;

box-shadow:

inset 0 5px white,

inset 0 -5px #bbb,

inset 5px 0 #d7d7d7,

inset -5px 0 #d7d7d7;

border-radius: 10%;

}

.pip {

display: block;

width: 24px;

height: 24px;

border-radius: 50%;

margin: 4px;

background-color: #333;

box-shadow: inset 0 3px #111, inset 0 -3px #555;

}

HTML：

<div class="first-face">

<span class="pip"></span>

</div>

<div class="second-face">

<span class="pip"></span>

<span class="pip"></span>

</div>

<div class="third-face">

<span class="pip"></span>

<span class="pip"></span>

<span class="pip"></span>

</div>

<div class="fourth-face">

<div class="column">

<span class="pip"></span>

<span class="pip"></span>

</div>

<div class="column">

<span class="pip"></span>

<span class="pip"></span>

</div>

</div>

<div class="fifth-face">

<div class="column">

<span class="pip"></span>

<span class="pip"></span>

</div>

<div class="column">

<span class="pip"></span>

</div>

<div class="column">

<span class="pip"></span>

<span class="pip"></span>

</div>

</div>

<div class="sixth-face">

<div class="column">

<span class="pip"></span>

<span class="pip"></span>

<span class="pip"></span>

</div>

<div class="column">

<span class="pip"></span>

<span class="pip"></span>

<span class="pip"></span>

</div>

</div>

<div class="ninth-face">

<div class="column">

<span class="pip"></span>

<span class="pip"></span>

<span class="pip"></span>

</div>

<div class="column">

<span class="pip"></span>

<span class="pip"></span>

<span class="pip"></span>

</div>

<div class="column">

<span class="pip"></span>

<span class="pip"></span>

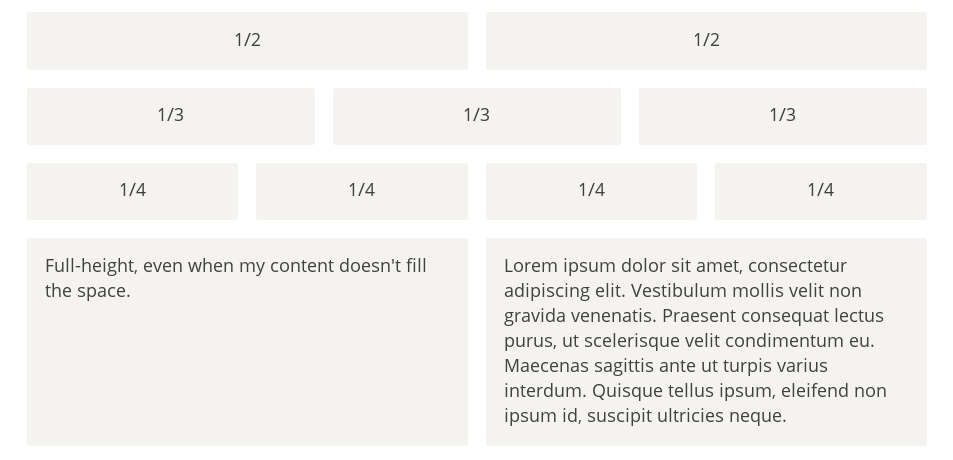
<span class="pip"></span>

</div>

</div>

1. 网格布局
2. 基本网格布局

最简单的网格布局，就是平均分布。在容器里面平均分配空间，跟上面的骰子布局很像，但是需要设置项目的自动缩放。



HTML代码如下。

<div class="Grid">

<div class="Grid-cell">...</div>

<div class="Grid-cell">...</div>

<div class="Grid-cell">...</div>

</div>

CSS代码如下。

.Grid {

display: flex;

}

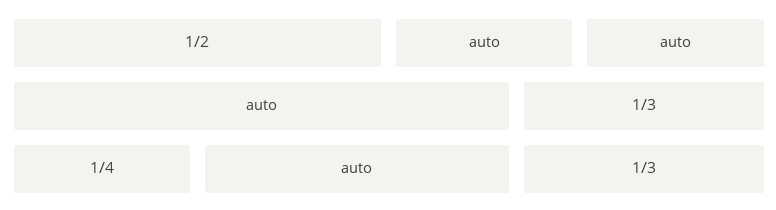
.Grid-cell {

flex: 1;

}

1. 百分比布局

某个网格的宽度为固定的百分比，其余网格平均分配剩余的空间。



HTML代码如下。

<div class="Grid">

<div class="Grid-cell u-1of4">...</div>

<div class="Grid-cell">...</div>

<div class="Grid-cell u-1of3">...</div>

</div>

CSS代码如下。

.Grid {

display: flex;

}

.Grid-cell {

flex: 1;

}

.Grid-cell.u-full {

flex: 0 0 100%;

}

.Grid-cell.u-1of2 {

flex: 0 0 50%;

}

.Grid-cell.u-1of3 {

flex: 0 0 33.3333%;

}

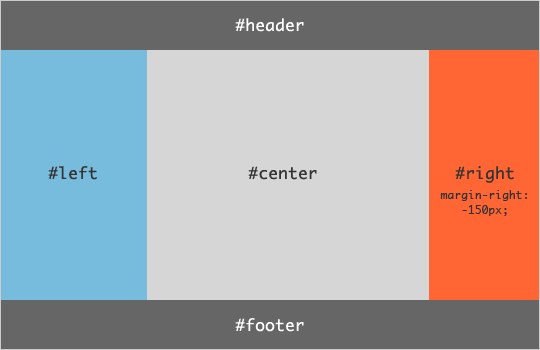
.Grid-cell.u-1of4 {

flex: 0 0 25%;

}

1. 圣杯布局

圣杯布局（Holy Grail Layout）指的是一种最常见的网站布局。页面从上到下，分成三个部分：头部（header），躯干（body），尾部（footer）。其中躯干又水平分成三栏，从左到右为：导航、主栏、副栏。



HTML代码如下。

<body class="HolyGrail">

<header>...</header>

<div class="HolyGrail-body">

<main class="HolyGrail-content">...</main>

<nav class="HolyGrail-nav">...</nav>

<aside class="HolyGrail-ads">...</aside>

</div>

<footer>...</footer></body>

CSS代码如下。

.HolyGrail {

display: flex;

min-height: 100vh;

flex-direction: column;}

header,

footer {

flex: 1;}

.HolyGrail-body {

display: flex;

flex: 1;}

.HolyGrail-content {

flex: 1;}

.HolyGrail-nav, .HolyGrail-ads {

/\* 两个边栏的宽度设为12em \*/

flex: 0 0 12em;}

.HolyGrail-nav {

/\* 导航放到最左边 \*/

order: -1;}

如果是小屏幕，躯干的三栏自动变为垂直叠加。

@media (max-width: 768px) {

.HolyGrail-body {

flex-direction: column;

flex: 1;

}

.HolyGrail-nav,

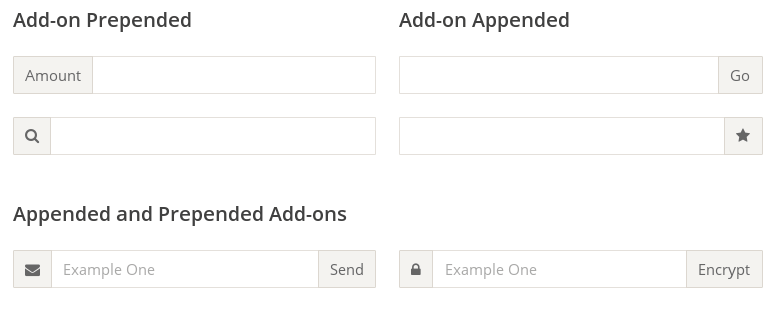
.HolyGrail-ads,

.HolyGrail-content {

flex: auto;

}}

1. 输入框布局



HTML代码如下。

<div class="InputAddOn">

<span class="InputAddOn-item">...</span>

<input class="InputAddOn-field">

<button class="InputAddOn-item">...</button></div>

CSS代码如下。

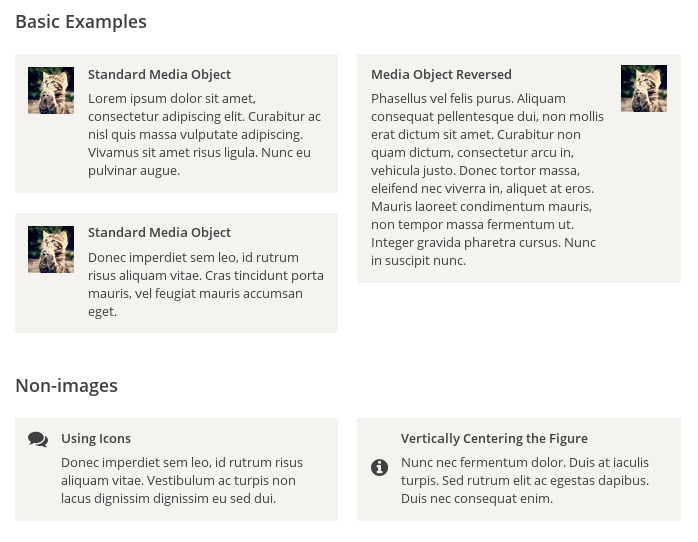
.InputAddOn {

display: flex;}

.InputAddOn-field {

flex: 1;}

1. 悬挂式布局



HTML代码如下。

<div class="Media">

<img class="Media-figure" src="" alt="">

<p class="Media-body">...</p></div>

CSS代码如下。

.Media {

display: flex;

align-items: flex-start;}

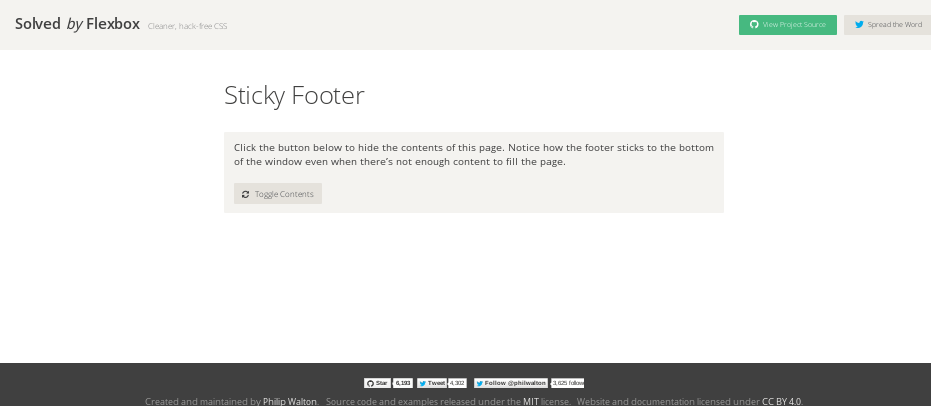
.Media-figure {

margin-right: 1em;}

.Media-body {

flex: 1;}

1. 固定的底栏



HTML代码如下。

<body class="Site">

<header>...</header>

<main class="Site-content">...</main>

<footer>...</footer></body>

CSS代码如下。

.Site {

display: flex;

min-height: 100vh;

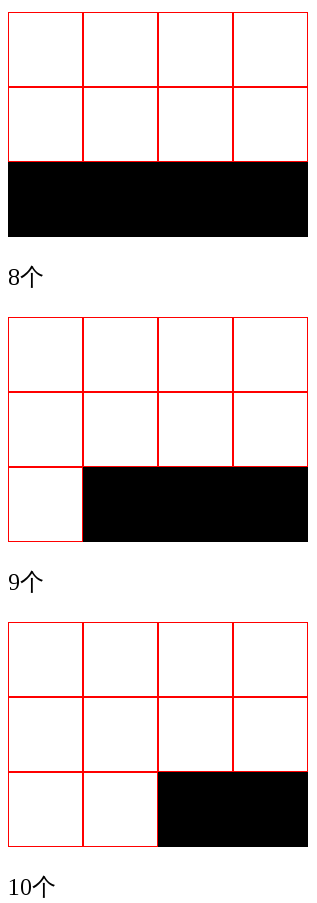
flex-direction: column;}

.Site-content {

flex: 1;}

1. 流式布局

每行的项目数固定，会自动分行



CSS的写法。

.parent {

width: 200px;

height: 150px;

background-color: black;

display: flex;

flex-flow: row wrap;

align-content: flex-start;}

.child {

box-sizing: border-box;

background-color: white;

flex: 0 0 25%;

height: 50px;

border: 1px solid red;}

1. 未知宽高元素上下左右垂直居中

html:

<div class="parent">

<div class="children">实现未知宽高元素垂直居中的方法1：通过定位和transform来实现</div>

</div>

方法1：通过定位和transform属性来实现

思路：子元素绝对定位，距离顶部 50%，左边50%，然后使用css3 transform:translate(-50%; -50%)  
优点：高大上,可以在webkit内核的浏览器中使用  
缺点：IE9以下不支持transform属性

<style>

.parent{

width:100%;

height:400px;

background:#666;

position:relative;

}

.children{

position:absolute;

top:50%;

left:50%;

background:red;

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

-webkit-transform:translate(-50%,-50%);

}

</style>

方法2：通过利用flex布局

思路：使用css3 flex布局  
优点：简单 快捷  
缺点：兼容不好

<style>

.parent{

width:100%;

height:400px;

background:#666;

display:flex;

justify-content:center;

align-items:center;

}

.children{

background:red;

}

</style>

方法3：通过table属性

思路：显示设置父元素为：table，子元素为：cell-table，这样就可以使用vertical-align: middle，实现垂直居中  
优点：父元素（parent）可以动态的改变高度（table元素的特性）  
缺点：IE8以下不支持

<style>

.parent{

width:100%;

height:400px;

background:#666;

display:table;

}

.children{

background:red;

display:table-cell;

text-align:center;

vertical-align:middle;

}

</style>

1. 原型链、对象、构造函数之间的联系
2. 构造函数

<http://www.ruanyifeng.com/blog/2010/05/object-oriented_javascript_encapsulation.html>

为了解决从原型对象生成实例的问题，Javascript提供了一个构造函数（Constructor）模式。

所谓"构造函数"，其实就是一个普通函数，但是内部使用了this变量。对构造函数使用new运算符，就能生成实例，并且this变量会绑定在实例对象上。

Javascript规定，每一个构造函数都有一个prototype属性，指向另一个对象。这个对象的所有属性和方法，都会被构造函数的实例继承。这意味着，我们可以把那些不变的属性和方法，直接定义在prototype对象上

1. 对象
2. 原型链

<http://www.ruanyifeng.com/blog/2010/05/object-oriented_javascript_inheritance.html>

采用prototype对象，实现继承

function extend(Child, Parent) {  
　　　　var F = function(){};

　　　　F.prototype = Parent.prototype;

　　　　Child.prototype = new F();

　　　　Child.prototype.constructor = Child;

　　　　Child.uber = Parent.prototype;

　　 }

拷贝继承

function extend2(Child, Parent) {

　　　　var p = Parent.prototype;

　　　　var c = Child.prototype;

　　　　for (var i in p) {

　　　　　　c[i] = p[i];

　　　　　　}

　　　　c.uber = p;

　　}

1. 非构造函数继承

<http://www.ruanyifeng.com/blog/2010/05/object-oriented_javascript_inheritance_continued.html>

object()方法

function object(o) {

　　　　function F() {}

　　　　F.prototype = o;

　　　　return new F();

}

var Chinese = {

　　nation:'中国'

};

var Doctor = object(Chinese);

Doctor.career = '医生';

alert(Doctor.nation); //中国

浅拷贝

function extendCopy(p) {

　　　var c = {};

　　　for (var i in p) {   
　　　　　　c[i] = p[i];  
　　　　 }

　　　c.uber = p;

　　　return c;  
　　}

深拷贝

function deepCopy(p, c) {

　　　　var c = c || {};

　　　　for (var i in p) {

　　　　　　if (typeof p[i] === 'object') {

　　　　　　　　c[i] = (p[i].constructor === Array) ? [] : {};

　　　　　　　　deepCopy(p[i], c[i]);

　　　　　　} else {

　　　　　　　　　c[i] = p[i];

　　　　　　}  
　　　　 }

　　　 return c;  
　　}

1. DOM事件绑定的几种方式

1. <div id="btn" onclick="clickone()"></div> //直接在DOM里绑定事件

　　　　<script>

　　　　　function clickone(){ alert("hello"); }

　　　　</script>

　2. <div id="btn"></div>

　　　　<script>

　　　　　document.getElementById("btn").onclick = function（）{ alert("hello"); } //脚本里面绑定

　　　　</script>

　3. <div id="btn"></div>

　　　　<script>

　　　　　document.getElementById("btn").addeventlistener("click",clickone,false); //通过侦听事件处理相应的函数

　　　　　function clickone(){ alert("hello"); }

　　　　</script>

那么问题来了，1 和 2 的方式是我们经常用到的，那么既然已经有两种绑定事件的方法为什么还要有第三种呢？答案是这样的：

用 "addeventlistener" 可以绑定多次同一个事件，且都会执行，而在DOM结构如果绑定两个 "onclick" 事件，只会执行第一个；在脚本通过匿名函数的方式绑定的只会执行最后一个事件。

　　1. <div id="btn" onclick="clickone()" onclick="clicktwo()"></div>

　　　　<script>

　　　　　function clickone(){ alert("hello"); } //执行这个

　　　　　function clicktwo(){ alert("world!"); }

　　　　</script>

　　2. <div id="btn"></div>

　　　　<script>

　　　　　document.getElementById("btn").onclick = function（）{ alert("hello"); }

　　　　　document.getElementById("btn").onclick = function（）{ alert("world"); } //执行这个

　　　　</script>

　　3. <div id="btn"></div>

　　　　<script>

　　　　　document.getElementById("btn").addeventlistener("click",clickone,false);

　　　　　function clickone(){ alert("hello"); } //先执行

　　　　　document.getElementById("btn").addeventlistener("click",clicktwo,false);

　　　　　function clicktwo(){ alert("world"); } //后执行

　　　　</script>

以上；可根据场景灵活选择。

1. 项目中用到的技术栈，以及觉得得意和出色的点，以及头疼点，怎么解决的
2. 了解过http2.0,websocket,https？说一下你的理解以及所了解的特性

<https://www.haorooms.com/post/html5_websocket>

<https://www.cnblogs.com/wuyiran/p/6398480.html>

<https://www.cnblogs.com/zlingh/p/5887143.html>

<https://blog.csdn.net/educast/article/details/50474098>

1. Jishu
2. webpack的入口文件怎么配置，多个入口怎么分割
3. Babel插件transform-runtime以及stage-2，它们的作用
4. Webpac.optimize.UglifyJsPlugin这个插件，有没有压缩速度慢？什么办法提升速度
5. http协议，200和304的理解和区别

### **Cache-Control策略**

* Cache-Control策略指明的是当前资源的有效期，控制浏览器缓存取数据还是重新发请求要服务器，其主要参数是max-age，即有效时间，以秒为单位，例如max-age，表明请求有效期为5分钟。

### **Last-Modified/If-Modified-Since**

Last-Modified/If-Modified-Since要配合Cache-Control使用。

* Last-Modified：标示这个响应资源的最后修改时间。web服务器在响应请求时，告诉浏览器资源的最后修改时间。
* If-Modified-Since：当资源过期时（使用Cache-Control标识的max-age），发现资源具有Last-Modified声明，则再次向web服务器请求时带上头 If-Modified-Since，表示请求时间。web服务器收到请求后发现有头If-Modified-Since 则与被请求资源的最后修改时间进行比对。若最后修改时间较新，说明资源又被改动过，则响应整片资源内容（写在响应消息包体内），HTTP 200；若最后修改时间较旧，说明资源无新修改，则响应HTTP 304 (无需包体，节省浏览)，告知浏览器继续使用所保存的cache。
* Http中200、302、304、404和500等响应状态码所表示的意义？
* 200:访问成功（表示一切正常，返回的是正常请求结果）
* 302：临时重定向（指出被请求的文档已被临时移动到别处，此文档的新的URL在Location响应头中给出）
* 304：未修改（表示客户机缓存的版本是最新的，客户机应该继续使用它。）
* 404：访问的文件不存在（服务器上不存在客户机所请求的资源）
* 500：内部服务器错误（服务器端的CGI、ASP、JSP等程序发生错误）

1. DOM事件中target和currentTarget的区别

target：触发事件的元素。

currentTarget：事件绑定的元素。

两者在没有冒泡的情况下，是一样的值，但在用了事件委托的情况下，就不一样了，例如：

<ul id="ulT">

<li class="item1">fsda</li>

<li class="item2">ewre</li>

<li class="item3">qewe</li>

<li class="item4">xvc</li>

<li class="item5">134</li>

</ul>

　　<script type="text/javascript">

document.getElementById("ulT").onclick = function (event) {

console.log(event.target);

console.log(event.currentTarget);

}

</script>

如果点击li，console上的情况如下：

<li class="item5">------target

<ul id="ulT">---------currentTarget

1. 平时怎么解决跨域，以及JSONP的原理和实现，以及cors怎么设置

<https://www.cnblogs.com/banning/p/6250677.html>

1.1、JSONP

　　JSONP是利用浏览器对script的资源引用没有同源限制，通过动态插入一个script标签，当资源加载到页面后会立即执行的原理实现跨域的。JSONP是一种非正式传输协议，该协议的一个要点就是允许用户传递一个callback或者开始就定义一个回调方法，参数给服务端，然后服务端返回数据时会将这个callback参数作为函数名来包裹住JSON数据，这样客户端就可以随意定制自己的函数来自动处理返回数据了。  
　　JSONP只支持GET请求而不支持POST等其它类型的HTTP请求,它只支持跨域HTTP请求这种情况，不能解决不同域的两个页面之间如何进行JavaScript调用的问题，JSONP的优势在于支持老式浏览器，弊端也比较明显：需要客户端和服务端定制进行开发，服务端返回的数据不能是标准的Json数据，而是callback包裹的数据。

1.2、CORS

<http://www.ruanyifeng.com/blog/2016/04/cors.html>

CORS是现代浏览器支持跨域资源请求的一种方式，全称是"跨域资源共享"（Cross-origin resource sharing），当使用XMLHttpRequest发送请求时，浏览器发现该请求不符合同源策略，会给该请求加一个请求头：Origin，后台进行一系列处理，如果确定接受请求则在返回结果中加入一个响应头：Access-Control-Allow-Origin;浏览器判断该相应头中是否包含Origin的值，如果有则浏览器会处理响应，我们就可以拿到响应数据，如果不包含浏览器直接驳回，这时我们无法拿到响应数据。  
　　CORS与JSONP的使用目的相同，但是比JSONP更强大，CORS支持所有的浏览器请求类型，承载的请求数据量更大，开放更简洁，服务端只需要将处理后的数据直接返回，不需要再特殊处理。

二、跨域解决方案距离

2.1、JSONP方案实现跨域

前段AJAX请求

$.ajax({

url: "http://otherdomain.com/manage/role/get",

async: false,

type: "get", 5 dataType: "jsonp",

data:{

"id":1

},

jsonp: "callback",

jsonpCallback:"fn",

success: function(data){

alert(data.code);

},

error: function(){

alert('fail');

}

})

服务端响应数据

@RequestMapping("/manage/role/get")

@ResponseBody

public String get(HttpServletRequest request, HttpServletResponse response) {

BaseOutput outPut = new BaseOutput();

try {

QueryFilter filter = new QueryFilter(request);

logger.info(filter.toString());

String id = filter.getParam().get(MainConst.KEY\_ID);

if(!StringUtil.isEmpty(id)) {

ImRole role = roleService.getByPk(filter);

outPut.setData(role);

}

else {

outPut.setCode(OutputCodeConst.INPUT\_PARAM\_IS\_NOT\_FULL);

outPut.setMsg("The get id is needed.");

}

} catch (Exception e) {

logger.error("获取角色数据异常！", e);

outPut.setCode(OutputCodeConst.UNKNOWN\_ERROR);

outPut.setMsg("获取角色数据异常！ " + e.getMessage());

}

return "fn("+JsonUtil.objectToJson(outPut)+")";

}

注意内容：

1、Ajax请求需要设置请求类型为Jsonp

dataType: "jsonp"

2、Ajax请求需要设置回调函数，当前函数值必须与服务器响应包含的callback名称相同

jsonpCallback:"fn"

3、Ajax请求可以设置jsonp（可选），传递给请求处理程序或页面，用以获得jsonp回调函数名的参数名，默认为:callback

jsonp: "callback"

4、服务端返回Json数据必须使用jsonpCallback设置的值进行包裹

return "fn("+JsonUtil.objectToJson(outPut)+")"

2.2、CORS方案实现跨域

前段AJAX请求

function test() {

$.ajax({

url: "http://localhost:8080/AdsServer/manage/role/get",

type: "get",

async: false,

data:{

"id":1

},

dataType:"json",

withCredentials:true,

success: function(data){

alert(data);

alert(data.code);

},

error: function(){

alert('fail');

}

})

}

服务端响应数据

@RequestMapping("/manage/role/get")

@ResponseBody

public String get(HttpServletRequest request, HttpServletResponse response) {

BaseOutput outPut = new BaseOutput();

try {

QueryFilter filter = new QueryFilter(request);

logger.info(filter.toString());

String id = filter.getParam().get(MainConst.KEY\_ID);

if(!StringUtil.isEmpty(id)) {

ImRole role = roleService.getByPk(filter);

outPut.setData(role);

}

else {

outPut.setCode(OutputCodeConst.INPUT\_PARAM\_IS\_NOT\_FULL);

outPut.setMsg("The get id is needed.");

}

} catch (Exception e) {

logger.error("获取角色数据异常！", e);

outPut.setCode(OutputCodeConst.UNKNOWN\_ERROR);

outPut.setMsg("获取角色数据异常！ " + e.getMessage());

}

return JsonUtil.objectToJson(outPut);

}

注意内容：

1、Ajax请求必须要设置withCredentials属性为true

withCredentials:true

2、服务端需要配置过滤器，讲配置能够进行跨域访问服务器的地址进行配置

response.setHeader("Access-Control-Allow-Origin", origin);

response.setHeader("Access-Control-Allow-Methods", "POST, GET, OPTIONS, DELETE,PATCH");

response.setHeader("Access-Control-Max-Age", "3600");

response.setHeader("Access-Control-Allow-Headers", "x-requested-with");// 是否支持cookie跨域

response.addHeader("Access-Control-Allow-Credentials", "true");

3、withCredentials设置成true时，Access-Control-Allow-Origin不支持通过\*的方式进行统配

4、Access-Control-Allow-Origin不能直接配置多个请求服务器，但是可以通过静态配置多个的方式，然后根据referer匹配，匹配到哪个则设置Access-Control-Allow-Origin为哪个的方式来配置多个

1. 深拷贝的实现原理
2. 项目中觉得可以改进的地方以及做的很优秀的地方
3. jishu2
4. 自己写过webpack的loader？原理？
5. 有没有去研究webpack的原理和机制，怎么实现的
6. Babel把ES6转成ES5或ES3之类的原理？
7. Git大型项目的团队合作，以及持续集成
8. 什么是函数柯里化？JS的API有哪些用到了函数柯里化的实现
9. ES6箭头函数this问题，以及拓展运算符

箭头函数的this指向的是定义时的this对象，而不是执行时的this对象。

<https://blog.csdn.net/yangbingbinga/article/details/61424363>

一针见血式总结:

****普通函数中的this:****

1. this总是代表它的直接调用者(js的this是执行上下文), 例如 obj.func ,那么func中的this就是obj

2.在默认情况(非严格模式下,未使用 'use strict'),没找到直接调用者,则this指的是 window (约定俗成)

3.在严格模式下,没有直接调用者的函数中的this是 undefined

4.使用call,apply,bind(ES5新增)绑定的,this指的是 绑定的对象

****箭头函数中的this****

箭头函数没有自己的this, 它的this是继承而来; 默认指向在定义它时所处的对象(宿主对象),而不是执行时的对象, 定义它的时候,可能环境是window; 箭头函数可以方便地让我们在 setTimeout ,setInterval中方便的使用this。

实例：

   //定义一个对象

   var obj = {

        x:100, //属性x

        show(){

        //延迟500毫秒，输出x的值

           setTimeout(

               //匿名函数

              function(){console.log(this.x);},

               500

           );

        }

   };

   obj.show();//打印结果：undefined

   //定义一个对象

   var obj = {

        x:100,//属性x

        show(){

            //延迟500毫秒，输出x的值

           setTimeout(

               //不同处：箭头函数

              () => { console.log(this.x)},

               500

            );

        }

   };

   obj.show();//打印结果：100

****解构运算符，扩展运算符，和rest运算符****

<https://www.cnblogs.com/chrischjh/p/4848934.html>

解构运算符

解构的作用是可以快速取得数组或对象当中的元素或属性，而无需使用arr[x]或者obj[key]等传统方式进行赋值

数组解构赋值：

var arr = ['this is a string', 2, 3];

//传统方式var a = arr[0],

b = arr[1],

c = arr[2];

//解构赋值，是不是简洁很多？var [a, b, c] = arr;

console.log(a);//this is a string

console.log(b);//2

console.log(c);//3

嵌套数组解构：

var arr = [[1, 2, [3, 4]], 5, 6];var [[d, e, [f, g]], h, i] = arr;

console.log(d);//1

console.log(f);//3

console.log(i);//6

函数传参解构：

var arr = ['this is a string', 2, 3];

function fn1([a, b, c]) {

console.log(a);

console.log(b);

console.log(c);

}

fn1(arr);

//this is a string

//2

//3

for循环解构：

var arr = [[11, 12], [21, 22], [31, 32]];

for (let [a, b] of arr) {

console.log(a);

console.log(b);

}

//11

//12

//21

//22

//31

//32

对象赋值解构：

var obj = {

name: 'chris',

sex: 'male',

age: 26,

son: {

sonname: '大熊',

sonsex: 'male',

sonage: 1

}

};

var {name, sex, age, son} = obj;

console.log(name + ' ' + sex + ' ' + age); //chris male 26

console.log(son); // { sonname: '大熊', sonsex: 'male', sonage: 1 }

对象传参解构：

var obj = {

name: 'chris',

sex: 'male',

age: 26,

son: {

sonname: '大熊',

sonsex: 'male',

sonage: 1

}

};

function fn2({sex, age, name}) {

console.log(name + ' ' + sex + ' ' + age);

}

fn2(obj);

//chris male 26

变量名与对象属性名不一致解构：

var obj = {

name: 'chris',

sex: 'male',

age: 26

};

var {name: nickname, age: howold} = obj;

console.log(nickname + ' ' + howold); //chris 26

嵌套对象解构：

var obj = {

name: 'chris',

sex: 'male',

age: 26,

son: {

sonname: '大熊',

sonsex: 'male',

sonage: 1

}

};

var {name, sex, age, son: {sonname, sonsex, sonage}} = obj;

console.log(sonname + ' ' + sonsex + ' ' + sonage);

//大熊 male 1

//Babel暂不支持这种嵌套解构

obj = {

name: 'chris',

sex: 'male',

age: [1, 2, 3]

}

{name, sex, age: [a, b, c]} = obj;

console.log(c);

嵌套对象属性重名，解构时需要更改变量名：

var obj = {

name: 'chris',

sex: 'male',

age: 26,

son: {

name: '大熊',

sex: 'male',

age: 1

}

};

//赋值解构

var {name: fathername, son: {name, sex, age}} = obj;

console.log(fathername); //chris

console.log(name); //大熊

//传参解构

function fn3({sex, age, name, son: {name: sonname}}) {

console.log(name + ' ' + sex + ' ' + age + ' ' + sonname);

}

fn3(obj);

//chris male 26 大熊

循环解构对象：

var arr = [{name: 'chris', age: 26}, {name: 'jack', age: 27}, {name: 'peter',age: 28}];

for (let {age, name} of arr) {

console.log(name + ' ' + age);

}

//chris 26

//jack 27

//peter 28

解构的特殊应用场景：

//变量互换

var x = 1,

y = 2;

var [x, y] = [y, x];

console.log(x); //2

console.log(y); //1

//字符串解构

var str = 'love';

var [a, b, c, d] = str;

console.log(a);//l

console.log(b);//o

console.log(c);//v

console.log(d);//e

扩展运算符

扩展运算符用三个点号表示，功能是把数组或类数组对象展开成一系列用逗号隔开的值。

var foo = function(a, b, c) {

console.log(a);

console.log(b);

console.log(c);

}

var arr = [1, 2, 3];

//传统写法

foo(arr[0], arr[1], arr[2]);

//使用扩展运算符

foo(...arr);

//1

//2

//3

特殊应用场景：

//数组深拷贝

var arr2 = arr;

var arr3 = [...arr];

console.log(arr===arr2); //true, 说明arr和arr2指向同一个数组

console.log(arr===arr3); //false, 说明arr3和arr指向不同数组

//把一个数组插入另一个数组字面量

var arr4 = [...arr, 4, 5, 6];

console.log(arr4);//[1, 2, 3, 4, 5, 6]

//字符串转数组

var str = 'love';

var arr5 = [...str];

console.log(arr5);//[ 'l', 'o', 'v', 'e' ]

rest运算符

rest运算符也是三个点号，不过其功能与扩展运算符恰好相反，把逗号隔开的值序列组合成一个数组

//主要用于不定参数，所以ES6开始可以不再使用arguments对象

var bar = function(...args) {

for (let el of args) {

console.log(el);

}

}

bar(1, 2, 3, 4);

//1

//2

//3

//4

bar = function(a, ...args) {

console.log(a);

console.log(args);

}

bar(1, 2, 3, 4);

//1

//[ 2, 3, 4 ]

rest运算符配合解构使用：

var [a, ...rest] = [1, 2, 3, 4];

console.log(a);//1

console.log(rest);//[2, 3, 4]

1. JS模块化Commonjs、UMD、CMD规范的了解，以及ES6的模块化跟其他几种的区别，以及出现的意义

<https://www.cnblogs.com/libin-1/p/7127481.html#4046069>

## **CommonJS**

CommonJS是服务器端模块的规范，Node.js采用了这个规范。

根据CommonJS规范，一个单独的文件就是一个模块。加载模块使用require方法，该方法读取一个文件并执行，最后返回文件内部的exports对象。

// foobar.js

//私有变量var test = 123;

//公有方法function foobar () {

this.foo = function () {

// do someing ...

}

this.bar = function () {

//do someing ...

}

}

//exports对象上的方法和变量是公有的

var foobar = new foobar();

exports.foobar = foobar;

//require方法默认读取js文件，所以可以省略js后缀

var test = require('./boobar').foobar;

test.bar();

CommonJS 加载模块是同步的，所以只有加载完成才能执行后面的操作。像Node.js主要用于服务器的编程，加载的模块文件一般都已经存在本地硬盘，所以加载起来比较快，不用考虑异步加载的方式，所以CommonJS规范比较适用。但如果是浏览器环境，要从服务器加载模块，这是就必须采用异步模式。所以就有了 AMD CMD 解决方案。

## **AMD和RequireJS**

AMD

AMD是"Asynchronous Module Definition"的缩写，意思就是"异步模块定义".

AMD设计出一个简洁的写模块API：

define(id?, dependencies?, factory);

第一个参数 id 为字符串类型，表示了模块标识，为可选参数。若不存在则模块标识应该默认定义为在加载器中被请求脚本的标识。如果存在，那么模块标识必须为顶层的或者一个绝对的标识。

第二个参数，dependencies ，是一个当前模块依赖的，已被模块定义的模块标识的数组字面量。

第三个参数，factory，是一个需要进行实例化的函数或者一个对象。

通过参数的排列组合，这个简单的API可以从容应对各种各样的应用场景，如下所述。

定义无依赖的模块

define( {

add : function( x, y ){

return x + y ;

}

} );

定义有依赖的模块

define(["alpha"], function( alpha ){

return {

verb : function(){

return alpha.verb() + 1 ;

}

}

});

定义数据对象模块

define({

users: [],

members: []

});

具名模块

define("alpha", [ "require", "exports", "beta" ], function( require, exports, beta ){

export.verb = function(){

return beta.verb();

// or:

return require("beta").verb();

}

});

包装模块

define(function(require, exports, module) {

var a = require('a'),

b = require('b');

exports.action = function() {};

} );

不考虑多了一层函数外，格式和Node.js是一样的：使用require获取依赖模块，使用exports导出API。

除了define外，AMD还保留一个关键字require。require 作为规范保留的全局标识符，可以实现为 module loader，也可以不实现。

模块加载

require([module], callback)

AMD模块化规范中使用全局或局部的require函数实现加载一个或多个模块，所有模块加载完成之后的回调函数。

其中：

[module]：是一个数组，里面的成员就是要加载的模块；  
callback：是模块加载完成之后的回调函数。

例如：加载一个math模块，然后调用方法 math.add(2, 3);

require(['math'], function(math) {

　math.add(2, 3);

});

## **CMD和SeaJS**

CMD是SeaJS 在推广过程中对模块定义的规范化产出

对于依赖的模块AMD是提前执行，CMD是延迟执行。不过RequireJS从2.0开始，也改成可以延迟执行（根据写法不同，处理方式不通过）。

CMD推崇依赖就近，AMD推崇依赖前置。

//AMD

define(['./a','./b'], function (a, b) {

//依赖一开始就写好

a.test();

b.test();

});

//CMD

define(function (requie, exports, module) {

//依赖可以就近书写

var a = require('./a');

a.test();

...

//软依赖

if (status) {

var b = requie('./b');

b.test();

}

});

虽然 AMD也支持CMD写法，但依赖前置是官方文档的默认模块定义写法。

AMD的API默认是一个当多个用，CMD严格的区分推崇职责单一。例如：AMD里require分全局的和局部的。CMD里面没有全局的 require，提供 seajs.use()来实现模块系统的加载启动。CMD里每个API都简单纯粹。

## **UMD**

UMD是AMD和CommonJS的糅合

AMD模块以浏览器第一的原则发展，异步加载模块。  
CommonJS模块以服务器第一原则发展，选择同步加载，它的模块无需包装(unwrapped modules)。  
这迫使人们又想出另一个更通用的模式UMD （Universal Module Definition）。希望解决跨平台的解决方案。

UMD先判断是否支持Node.js的模块（exports）是否存在，存在则使用Node.js模块模式。  
再判断是否支持AMD（define是否存在），存在则使用AMD方式加载模块。

(function (window, factory) {

if (typeof exports === 'object') {

module.exports = factory();

} else if (typeof define === 'function' && define.amd) {

define(factory);

} else {

window.eventUtil = factory();

}

})(this, function () {

// module ...

});

## ES6

es6通过import、export实现模块的输入输出。其中import命令用于输入其他模块提供的功能，export命令用于规定模块的对外接口。

export

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。如果希望外部文件能够读取该模块的变量，就需要在这个模块内使用export关键字导出变量。如：

// profile.js

export var a = 1;

export var b = 2;

export var c = 3;

下面的写法是等价的，这种方式更加清晰（在底部一眼能看出导出了哪些变量）：

var a = 1;

var b = 2;

var c = 3;

export {a, b, c}

export命令除了输出变量，还可以导出函数或类。

****导出函数****

export function foo(){}

function foo(){

}

function bar(){

}

export {foo, bar as bar2}

其中上面的as表示给导出的变量重命名。

要注意的是，export导出的变量只能位于文件的顶层，如果处于块级作用域内，会报错。如：

function foo() {

export 'bar'; // SyntaxError

}

****导出类****

export default class {} // 关于default下面会说

export语句输出的值是动态绑定，绑定其所在的模块。

// foo.js

export var foo = 'foo';

setTimeout(function() {

foo = 'foo2';

}, 500);

// main.js

import \* as m from './foo';

console.log(m.foo); // foo

setTimeout(() => console.log(m.foo), 500); // foo2

import

import命令可以导入其他模块通过export导出的部分。

// abc.js

var a = 1;

var b = 2;

var c = 3;

export {a, b, c}

//main.js

import {a, b, c} from './abc';

console.log(a, b, c);

如果想为导入的变量重新取一个名字，使用as关键字（也可以在导出中使用）。

import {a as aa, b, c};

console.log(aa, b, c)

如果想在一个模块中先输入后输出一个模块，import语句可以和export语句写在一起。

import {a, b, c} form './abc';

export {a, b, c}

// 使用连写, 可读性不好，不建议

export {a, b, c} from './abc';

模块的整体加载

使用\*关键字。

// abc.js

export var a = 1;

export var b = 2;

export var c = 3;

// main.js

import \* as abc form './abc';

console.log(abc.a, abc.b, abc.c);

export default

在export输出内容时，如果同时输出多个变量，需要使用大括号{}，同时导入也需要大括号。使用export defalut输出时，不需要大括号，而输入（import）export default输出的变量时，不需要大括号。

// abc.js

var a = 1, b = 2, c = 3;

export {a, b};

export default c;

import {a, b} from './abc';

import c from './abc'; // 不需要大括号

console.log(a, b, c) // 1 2 3

本质上，export default输出的是一个叫做default的变量或方法，输入这个default变量时不需要大括号。

// abc.js

export {a as default};

// main.js

import a from './abc'; // 这样也是可以的

// 这样也是可以的

import {default as aa} from './abc';

console.log(aa);

## ES6 模块与 CommonJS 模块的差异

ES6 模块与 CommonJS 模块完全不同。它们有两个重大差异。

* CommonJS 输出是值的拷贝，即原来模块中的值改变不会影响已经加载的该值，ES6静态分析，动态引用，输出的是值的引用，值改变，引用也改变，即原来模块中的值改变则该加载的值也改变。
* CommonJS 模块是运行时加载，ES6 模块是编译时输出接口。
* CommonJS 加载的是整个模块，即将所有的接口全部加载进来，ES6 可以单独加载其中的某个接口（方法），
* CommonJS this 指向当前模块，ES6 this 指向undefined
* ES6 模块的运行机制与 CommonJS 不一样。JS 引擎对脚本静态分析的时候，遇到模块加载命令import，就会生成一个只读引用。等到脚本真正执行时，再根据这个只读引用，到被加载的那个模块里面去取值。ES6 模块不会缓存运行结果，而是动态地去被加载的模块取值，并且变量总是绑定其所在的模块。

CommonJs模块化

// lib.js

var counter = 3;

function incCounter() {

counter++;

}

module.exports = {

counter: counter,

incCounter: incCounter,

};

// main.js

var mod = require('./lib');

console.log(mod.counter); // 3

mod.incCounter();

console.log(mod.counter); // 3

ES6模块化

// lib.js

export let counter = 3;

export function incCounter() {

counter++;

}

// main.js

import { counter, incCounter } from './lib';

console.log(counter); // 3

incCounter();

console.log(counter); // 4

1. Vue实现双向数据绑定的原理，以及vue.js和react.js异同点，如果让你选框架，你怎么权衡这两个框架，分析一下

<https://www.cnblogs.com/libin-1/p/6893712.html>

<https://github.com/canfoo/self-vue>

vue数据双向绑定是通过数据劫持结合发布者-订阅者模式的方式来实现的

vue是通过Object.defineProperty()来实现数据劫持的

1. 写博客，草稿的交互细节以及实现原理
2. jishu3
3. 自我介绍，项目技术栈，项目遇到的问题
4. 一个业务场景，面对产品不断迭代，以及需求变动该怎么应对，具体技术方案实现
5. 代码的可维护性和可扩展性
6. 怎么获取一个元素到视图顶部的距离
7. getBoundingClientRect获取的top和offsetTop获取的top区别
8. 事件委托
9. 百度的服务不想让阿里使用，如何识别到是阿里的请求，然后跳转到404或者拒绝服务之类的
10. http协议头Referer
11. 二分查找的时间复杂度怎么求，是多少
12. XSS是什么，攻击原理，怎么预防
13. 跨站脚本攻击
14. 线性存储结构和链式存储结构有什么区别？以及优缺点
15. 移动端日历、PC端日历以及桌面日历的一些不同和需要注意的地方
16. 用最简单的方式实现数组去重
17. ES6实现

[...new Set([])]

1. ES5实现

[].filter(function(item,index,array){

return index===array.indexOf(item)

})

1. 怎么实现草稿，多终端同步，以及冲突问题