页面布局

*<!--题目：假设高度已知，请写出三栏布局，其中左栏、右栏宽度各为300px，中间自适应-->  
<!--在写布局前都要将样式初始化-->*<**style**>  
 **html \***{  
 padding: 0;  
 margin: 0;  
 }  
 .**layout** .**left-center-right div**{  
 height: 100**px**;  
 }  
</**style**>  
*<!--第一种 浮动-->*<**div** class=**"layout float"**>  
 <**style**>  
 .**layout**.**float** .**left**{  
 float: **left**;  
 width: 300**px**;  
 background: **red**;  
 }  
 .**layout**.**float** .**right**{  
 float: **right**;  
 width: 300**px**;  
 background: **blue**;  
 }  
 .**layout**.**float** .**center**{  
 background: **yellow**;  
 }  
 </**style**>  
 <**div** class=**"left-center-right"**>  
 <**div** class=**"left"**></**div**>  
 <**div** class=**"right"**></**div**>  
 <**div** class=**"center"**><**h1**>浮动</**h1**></**div**>  
 </**div**>  
</**div**>

*<!--第二种 绝对定位-->*<**div** class=**"layout absoulte"** style=**"**margin-top: 20**px"**>  
 <**style**>  
 .**layout**.**absoulte** .**left-center-right div**{  
 position: **absolute**;  
 }  
 .**layout**.**absoulte** .**left**{  
 left: 0;  
 width: 300**px**;  
 background: **red**;  
 }  
 .**layout**.**absoulte** .**center**{  
 left: 300**px**;  
 right: 300**px**;  
 background: **yellow**;  
 }  
 .**layout**.**absoulte** .**right**{  
 right: 0;  
 width: 300**px**;  
 background: **blue**;  
 }  
 </**style**>  
 <**div** class=**"left-center-right"**>  
 <**div** class=**"left"**></**div**>  
 <**div** class=**"center"**><**h1**>定位</**h1**></**div**>  
 <**div** class=**"right"**></**div**>  
 </**div**>  
</**div**>

*<!--第三种 flex-->*<**div** class=**"layout flexbox"** style=**"**margin-top: 145**px"**>  
 <**style**>  
 .**layout**.**flexbox** .**left-center-right**{  
 display: **flex**;  
 }  
 .**layout**.**flexbox** .**left**{  
 width: 300**px**;  
 background: **red**;  
 }  
 .**layout**.**flexbox** .**center**{  
 */\*flex:1中间自适应原理\*/* flex: 1;  
 background: **yellow**;  
 }  
 .**layout**.**flexbox** .**right**{  
 width: 300**px**;  
 background: **blue**;  
 }  
 </**style**>  
 <**div** class=**"left-center-right"**>  
 <**div** class=**"left"**></**div**>  
 <**div** class=**"center"**>flex</**div**>  
 <**div** class=**"right"**></**div**>  
 </**div**>  
</**div**>

*<!--第四种 表格布局-->*<**div** class=**"layout table"** style=**"**margin-top: 20**px"**>  
 <**style**>  
 .**layout**.**table** .**left-center-right**{  
 width: 100%;  
 display: **table**;  
 height: 100**px**;  
 }  
 .**layout**.**table** .**left-center-right div**{  
 display: **table-cell**;  
 }  
 .**layout**.**table** .**left**{  
 width: 300**px**;  
 background: **red**;  
 }  
 .**layout**.**table** .**center**{  
 background: **yellow**;  
 }  
 .**layout**.**table** .**right**{  
 width: 300**px**;  
 background: **blue**;  
 }  
 </**style**>  
 <**div** class=**"left-center-right"**>  
 <**div** class=**"left"**></**div**>  
 <**div** class=**"center"**>表格布局</**div**>  
 <**div** class=**"right"**></**div**>  
 </**div**>  
</**div**>

*<!--第五种 网格布局-->*<**div** class=**"layout grid"** style=**"**margin-top: 20**px"**>  
 <**style**>  
 .**layout**.**grid** .**left-center-right**{  
 display: **grid**;  
 width: 100%;  
 */\*行高\*/* grid-template-rows:100**px** ;  
 */\*列\*/* grid-template-columns: 300**px auto** 300**px**;  
 }  
 .**layout**.**grid** .**left**{  
 background: **red**;  
 }  
 .**layout**.**grid** .**center**{  
 background: **yellow**;  
 }  
 .**layout**.**grid** .**right**{  
 background: **blue**;  
 }  
 </**style**>  
 <**div** class=**"left-center-right"**>  
 <**div** class=**"left"**></**div**>  
 <**div** class=**"center"**>grid</**div**>  
 <**div** class=**"right"**></**div**>  
 </**div**>  
</**div**>  
  
*<!--高度位置的情况下 只有flex布局和表格布局可用-->  
<!--同类题目有-->  
<!--上下高度固定，中间自适应-->  
<!--左宽度固定，右自适应-->  
<!--右宽度固定，左自适应-->  
<!--上高度固定，下自适应-->  
<!--下高度固定，上自适应-->*

css盒模型

*<!--css和模型基本概念：分为标准模型和IE模型-->  
<!--css盒模型（有border，margin，padding，content）-->  
<!--标准模型和ie模型区别：宽度和高度的计算方式不同；标准模型的宽度和高度不包括padding和border；ie模型的高度和宽度是包括padding和border-->  
<!--css 如何设置两种模型：标准盒模型是box-sizing:content-box;ie盒模型是box-sizing:border-box-->  
  
<!--js如何设置获取盒模型的宽和高 这里的dom是指获取到的元素-->  
<!--1、dom.style.width/height-->  
<!--2、dom.currentStyle.width/height（只支持ie）-->  
<!--3、window.getComputedStyle(dom).width/height(支持其他的浏览器)-->  
  
<!--根据实例来解释边距重叠-->  
<!--父子之间会存在边距重叠-->  
<!--边距重叠的解决方案BFC；BFC的概念是：块级格式化上下文-->  
<!--BFC的原理：就是BFC的渲染规则：1、BFC这个元素的垂直方向上的边距会发生重叠；2、BFC区域不会与浮动元素的box重叠用来清楚浮动的；  
3、BFC在页面上是一个独立的容器，外面的元素不会影响里面的元素，同理里面的元素也不会影响外面的；4、计算BFC高度的时候浮动元素也会参与计算-->  
<!--如何创建BFC：1、float值不为null；2、position值不为static；3、overflow不为visible;-->  
  
<!--BFC垂直方向边距重叠-->*<**div** id=**"margin"**>  
 <**style**>  
 **#margin**{  
 background: **pink**;  
 */\*创建BFC\*/* overflow: **hidden**;  
 }  
 **#margin p**{  
 margin: 5**px auto** 25**px**;  
 background: **red**;  
 }  
 </**style**>  
 <**p**>1</**p**>  
 <**div** style=**"**overflow: **hidden"**>  
 <**p**>2</**p**>  
 </**div**>  
 <**p**>3</**p**>  
</**div**>  
  
*<!--BFC不与float重叠-->*<**div** id=**"layout"** style=**"**margin-top: 10**px"**>  
 <**style**>  
 **#layout**{  
 background: **red**;  
 }  
 **#layout** .**left**{  
 float: **left**;  
 width: 100**px**;  
 height: 100**px**;  
 background: **pink**;  
 }  
 **#layout** .**right**{  
 height: 110**px**;  
 background: **#ccc**;  
 */\*创建BFC\*/* overflow: **auto**;  
 }  
 </**style**>  
 <**div** class=**"left"**></**div**>  
 <**div** class=**"right"**></**div**>  
</**div**>  
  
*<!--BFC子元素即使是float也会参与高度计算（BFC可以清楚浮动）-->*<**div** id=**"float"** style=**"**margin-top: 10**px"**>  
 <**style**>  
 **#float**{  
 */\*穿件BFC\*/  
 /\*overflow: auto;\*/* float: **left**;  
 background: **red**;  
 }  
 **#float** .**float**{  
 float: **left**;  
 height: 50**px**;  
 }  
 </**style**>  
 <**div** class=**"float"**>我是浮动元素</**div**>  
</**div**>

类型转换

*<!--类型转换分为：显示类型和隐示类型-->  
  
<!--数据类型分为原始类型和对象-->  
<!--原始类型：Boolean，null，Undefined，Number，String，Symbol（这个是es6中的）-->  
<!--对象Object-->  
  
<!--显示类型转换-->  
<!--1、Number函数-->  
  
<!--a、数值：转换后还是原来的值-->*<**script**>  
 **console**.log(Number(324))*//324*</**script**>  
*<!--b、字符串：如果可以被解析为数值，则转换相应的数值；否则得到NaN，空字符串转为0-->*<**script**>  
 **console**.log(Number(**'324abc'**))*//NaN* **console**.log(Number(**''**))*//0*</**script**>  
*<!--c、true转成1，false转成0-->*<**script**>  
 **console**.log(Number(**false**))*//0*</**script**>  
*<!--d、undefined：转成NaN-->*<**script**>  
 **console**.log(Number(**undefined**))*//NaN*</**script**>  
*<!--e、null：转成0-->*<**script**>  
 **console**.log(Number(**null**))*//0*</**script**>  
*<!--f、对象类型转换：先调用对象自身的valueOf方法，如果该方法返回原始类型的值（数值、字符串和布尔值），则直接对该值使用Number方法，不在进行后续步骤-->  
<!--如果valueOf方法返回复合类型的值，再调用对象自身的toString方法，如果toString方法返回原始类型的值，则对该值使用Number方法，不再进行后续步骤-->  
<!--如果toString方法返回的是复合类型的值，则报错-->*<**script**>  
 **console**.log(**'a'**,Number({**a**:1}))*//a:NaN*</**script**>  
  
  
*<!--2、String函数-->  
  
<!--a、数值：转为相应的字符串-->*<**script**>  
 **console**.log(**'a'**,String(123))*//"123"*</**script**>  
*<!--b、字符串：转换后还是原来的值-->*<**script**>  
 **console**.log(**'a'**,String(**'abc'**))*//"abc"*</**script**>  
*<!--c、布尔值：true转为“true”；false转为“false”-->*<**script**>  
 **console**.log(**'a'**,String(**true**))*//"true"*</**script**>  
*<!--d、undefined：转为undefined-->*<**script**>  
 **console**.log(**'a'**,String(**undefined**))*//"undefined"*</**script**>  
*<!--e、null：转为null-->*<**script**>  
 **console**.log(**'a'**,String(**null**))*//"null"*</**script**>  
  
*<!--f、对象类型转换：先调用toString方法，如果toString方法返回的是原始类型的值，则对该值使用String方法  
，不在继续进行；如果toString方法返回的是符合类型的值，再调用valueOf方法，如果valueOf方法返回的是原始类型的值，  
则对该值使用String方法，不再进行以下步骤；如果valueOf方法返回的是复合类型的值，则报错-->*<**script**>  
 **console**.log(**'a'**,String({**a**:1}))*//a [object Object]*</**script**>  
  
*<!--3、Boolean函数-->  
<!--如果遇到undefined、null、-0、+0、NaN、''（空字符串）都转换成false；除了这些都是转换成true-->  
<!--一个例子代表-->*<**script**>  
 **console**.log(Boolean(**undefined**))*//false*</**script**>  
  
*<!--隐式类型转换；发生的情况：四则运算、判断语句、Native调用-->  
<!--一些小题目-->*<**script**>  
 []+[];*//""* []+{};*//"[object Object]"* {}+[];*//0* {}+{};*//"谷歌下是[object Object][object Object]" 火狐下是NaN* **true** + **true**;*//2* 1 + {**a**:1}*//"1[object Object]"*</**script**>

原型链

*<!--创建对象有几种方法-->*<**script**>  
 *//字面量对象* **var *o1*** = {**name**:**'o1'**};  
 *// new操作符创建对象此条就是下面的构造函数  
 // var o11 = new Object({name:'o11'});*</**script**>  
<**script**>  
 *//使用构造函数创建对象* **var** *M1* = **function**(){**this**.**name**=**'o2'**};  
 **var *o2*** = **new** *M1*();  
</**script**>  
<**script**>  
 *//Object.create* **var *P*** = {**name**:**'o3'**};  
 **var *o3*** = Object.create(***P***);  
</**script**>  
  
*<!--理解：首先构造函数可以通过new来生成实例；函数都有一个prototype属性，prototype指的就是原型对象，原型对象中会有个构造器constructor；  
这个构造器会默认指向你声明的那个构造函数；通过new生成的实例的\_proto\_全等于构造函数的prototype；-->  
  
<!--原型链就是说我从我的一个实例对象网上找构造这个实例的相关联的对象，然后这个关联的对象再往上找它又有创造它的上一级的原型对象，以此类推一直到object.prototype终止，这个链条就断了-->*

*<!--例子-->*<**script**>  
 **var** *M1* = **function**(){**this**.**name**=**'o2'**};*//这个构造函数* **var *o2*** = **new** *M1*();*//这个是实例* ***o2* instanceof** *M1//true* ***o2* instanceof** Object*//true* ***o2***.\_\_proto\_\_===*M1*.**prototype***//true  
 M1*.**prototype**.\_proto\_===Object.**prototype***//true  
 //用constructor判断比instanceof更严谨 例子* ***o2***.\_\_proto\_\_.constructor === *M1//true* ***o2***.\_\_proto\_\_.constructor === Object*//false*</**script**>  
  
  
*<!--new运算符-->  
<!--就是一个对象被创建，它继承构造函数的prototype（也就是原型对象）；构造函数被执行，执行的时候，上下文也就是this，  
会被指定为这个新实例，如果构造函数返回了一个对象，那么这个对象会取代整个new出来的结果，如果构造函数没有返回对象，  
那么new出来的结果就是之前创建的对象-->  
<!--下面模拟一个new运算符的例子代码-->*<**script**>  
 **var** *new2* = **function**(func){  
 *// 第一步创建空对象，空对象要关联构造函数的原型对象* **var** o = Object.create(func.**prototype**)  
 *// 第二步执行构造函数* **var** k=func.call(o)  
 *// 第三步判断构造函数的运行结果是不是对象类型，如果是对象就返回构造函数如果不是就返回刚创建的对象o* **if**(**typeof** k === **'object'**){  
 **return** k  
 }**else**{  
 **return** o  
 }  
 }  
</**script**>

DOM事件

*<!--DOM事件的级别：-->  
<!--1、DOM0：element.onclick=function(){}-->  
<!--2、DOM2：element.addEventListener('click',functiong(){},false)-->  
<!--3、DOM3：element.addEventListener('keyup',functiong(){},false) 在2的基础上增加了很多事件类别-->  
  
  
<!--事件模型：就是捕获和冒泡-->  
<!--捕获是从上往下-->  
<!--冒泡是从目标元素往上-->  
  
<!--事件流：浏览器在为当前页面与用户做交互的过程中比如说点击了鼠标然后传到页面上怎么响应的就是事件流-->  
<!--事件流分三个阶段：第一阶段是捕获；第二阶段为目标阶段比如说点击了一下鼠标按钮等；第三阶段是冒泡的过程-->  
<!--也就是事件通过捕获到达目标元素，从目标元素在上传到window对象，也就是冒泡的过程-->  
  
<!--描述DOM事件捕获的具体流程；首先是由上到下，是由window->document->html->body->然后一层一层到达目标元素-->  
<!--冒泡与上面的相反-->  
  
<!--Event对象的常见应用-->  
<!--event.preventDefault() 阻止默认事件-->  
<!--event.stopPropagation() 阻止事件冒泡-->  
<!--event.stoplmmediatePropagation()举个例子就是一个元素绑定两个事件a和b件；但是我想让他执行a不执行b；在a里面加上这个就可以实现-->  
<!--event.currentTarget 当前所绑定的事件的元素-->  
<!--event.target 一般会出现代理题目基础里面有简介-->  
  
<!--自定义事件-->*<**script**>  
 **var *eve*** = **new** Event(**'custome'**);  
 ev.addEventListener(**'custome'**,**function**(){*//ev是获取的元素* **console**.log(**'custome'**);  
 });  
 ev.dispatchEvent(***eve***)*//用dispatchEvent这个来触发自定义事件*</**script**>

js运行机制

*<!--例子-->*<**script**>  
 **console**.log(**"A"**);  
 **while** (**true**){  
  
 }  
 **console**.log(**"B"**)  
 *//这里只会输出A*</**script**>  
  
*<!--例子2-->*<**script**>  
 **for**(**var *i*** = 0;***i***<4;***i***++){  
 setTimeout(**function**(){  
 **console**.log(***i***)  
 },1000)  
 }  
 *//会输出4个4*</**script**>  
*<!--什么时候开启异步任务-->  
<!--setTimeout和setInterval；DOM事件；ES6中的Promise-->*

渲染机制

*<!--什么事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  
。浏览器会使用它来判断文档类型，决定使用何种协议来解析，以及切换浏览器模式-->  
<!--DTD通俗的讲就是告诉浏览器我是什么文档类型，浏览器根据这个来判断用什么引擎来解析它-->  
<!--DOCTYPE是用来声明文档类型和DTD规范的，一个主要的用途便是文件的合法性验证，如果文件代码不合法，那么浏览器解析时便会出现一些错误-->  
  
<!--常见的DOCTYPE有哪些-->  
<!--1、html5的是<!DOCTYPE html>-->  
<!--2、HTML4.01 严格模式(该DTD包含所有HTMl元素和属性，但不包括展示性的和弃用的元素如font)-->  
<!--<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">-->  
  
<!--3、HTML4.01 传统模式(该DTD包含所有HTMl元素和属性，包括展示性的和弃用的元素如font)-->  
<!--<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transition//EN" "http://www.w3.org/TR/html4/loose.dtd">-->  
  
<!--浏览器渲染过程-->  
<!-- 浏览器拿到html 和css之后，将html经过 htmlParser转成domTree，css根据css规则转成cssomTree，最后他两整合形成了RenderTree；  
与RenderTree平行的有个Layout，通过layout就可以精确的计算出要显示的元素的位置和宽高颜色等都在RenderTree呈现出来，最后呈现出来内容-->  
<!--注：domtree就是html节点从上到下进行分析-->  
  
<!--重排Reflow-->  
<!--DOM结构中的各个元素都有自己的（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-->  
<!--增加、删除、修改DOM节点时，会导致Reflow或Repaint-->  
<!--移动DOM的位置，或是搞个动画的时候-->  
<!--修改css样式的时候-->  
<!--Resize窗口的时候（移动端没有这个问题），或者滚动的时候有可能触-->  
<!--修改网页的默认字体时-->  
  
<!--重绘Repaint-->  
<!--当各种盒子的位置、大小以及其他属性，例如颜色、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-->  
  
<!--触发Repaint-->  
<!--DOM改动-->  
<!--CSS改动-->*安全类

*<!--CSRF 跨站请求伪造-->  
<!--场景：用户是网站A的注册用户，然后登陆，网站A核查身份，如果正确就是下发cookie，这样就完成了一次身份认证的过程；  
用户右访问网站B，这里加入有个引诱链接，假设这个链接是指向A的一个接口，点击后，就会攻击，前提一定是在注册的网站登录过-->  
<!--存在攻击的两个原因：第一接口漏洞，第二确实在注册的网站登录过的前提-->  
<!--防御：token验证(只要浏览网友，服务器就会向本地自动存储的)，referer而验证（refer而就是页面来源验证），隐藏令牌-->  
  
<!--XSS 跨域脚本攻击-->  
<!--基础里面有讲解 安全都是作为了解就可以，也可以在下面这个了解下不过没必要细了解-->  
<!--https://www.imooc.com/learn/812-->  
  
<!--XSS和CSRF的区别，前者是向页面里面注入js去运行，后者是利用漏洞去执行那些接口而且前提是要登陆注册的网站-->*

面向对象

*<!--类的声明：一种是构造函数模拟类的方式；另一种es6中对class的声明->*<**script**>  
 *//构造函数声明* **function** *Animal*(){  
 **this**.**name** = **'name'***//this表示构造函数声明* }  
 *//es6中class的声明,class后面就是类名* **class** Animal2{  
 constructor(){  
 *//constructor就是构造函数* **this**.**name** = name;  
 }  
 }  
</**script**>  
*<!--如何通过类来实例化生成一个对象-->  
<!--实例化 直接new就可以了-->*<**script**>  
 **console**.log(**new** *Animal*(),**new** Animal2())  
</**script**>  
  
*<!--js的继承有几种方式有几种形式-->  
<!--1、借助构造函数来实现继承-->  
<!--此方法缺点就是Parent1原型链上的东西并没有被child1所继承，child1指继承了Parent1里面的属性-->*<**script**>  
 **function** *Parent1*(){  
 **this**.**name** = **'parent1'** }  
*// Parent1.prototype.say = function(){}* **function** *Child1*(){  
 *//child1是不会继承Paren1原型对象下的say方法  
 // 将父级元素的构造函数的this指向子构造函数的实例上去  
 Parent1*.call(**this**);*//call和apply 改变函数运行的上下文* **this**.**type** = **'child1'** }  
 **console**.log(**new** *Child1*())  
*// console.log(new Child1().say()) 不会继承say方法*</**script**>  
  
*<!--2、借助原型链实现继承-->  
<!--缺点就是你改一个实例下面的属性，其他实例下的 也会跟着变化-->*<**script**>  
 **function** *Parent2*(){  
 **this**.**name** = **'parent2'** }  
 **function** *Child2*(){  
 **this**.**type** = **'child2'** }  
 *Child2*.**prototype** = **new** *Parent2*();  
 **console**.log(**new** *Child2*())  
</**script**>  
  
*<!--3、组合方式将上面两个结合-->*<**script**>  
 **function** *Parent3*(){  
 **this**.**name** = **'Parent3'**;  
 **this**.**play**=[1,2,3]  
 }  
 **function** *Child3*(){  
 *Parent3*.call(**this**);  
 **this**.**type** = **'child3'**;  
 }  
*// Child3.prototype = new Parent3();  
// 优化1 这个缺点是都指向父类的构造函数，父类和子类没有区分  
// Child3.prototype = Parent3.prototype;  
// 优化2 通过Object.create()实现了父类和子类的隔离  
 Child3*.**prototype** = Object.create(*Parent3*.**prototype**);  
 *Child3*.**prototype**.constructor =*Child3//在指定下即可* **var *s3*** = **new** *Child3*();  
 **var *s4*** = **new** *Child3*();  
 ***s3***.**play**.push(4);  
 **console**.log(***s3***.**play**,***s4***.**play**)*//[1,2,3,4] [1,2,3]*</**script**>

HTTP协议类

*<!--HTTP协议的主要特点：1、简单快速；2、灵活；3、无连接；4、无状态-->  
<!--简单快速是指每个资源是固定的所以http协议中处理起来是比较简单的-->  
<!--灵活是指在每一个http协议中有个头部分会有个数据类型，通过一个http协议就可以完成不同数据类型的传输，所以比较灵活-->  
<!--无连接指的就是连接一次就会断掉；不会保存连接-->  
<!--无状态指的是客户端和服务端是两种身份；例如客户端向服务端请求一个图片，那么http协议帮你建立连接，帮你中间传输，  
这次任务完成后就会断开；下次客户端在过来服务端是没办法区分上一次连接和这次连接是不是同一个身份，因为服务端是没有记住你的状态；  
也就是说但从http协议上不能区分两次连接者的身份的，这就是无状态导致的-->  
  
  
<!--HTTP报文的组成部分：有请求报文和响应报文-->  
<!--请求报文有：请求行；请求头；空行；请求体-->  
<!--响应报文有：状态行；响应头；空行；响应体-->  
  
<!--请求行包括：http方法、页面地址、http协议以及版本-->  
<!--请求头包括：就是一些key value值来告诉服务端我要的内容要注意什么类型-->  
<!--空行就是告诉服务端往下进行-->  
<!--状态行有http协议版本，http状态码-->  
<!--响应头包括：就是一些key value值来响应服务端内容要，类型等-->  
  
<!--Http方法：-->  
<!--Get 获取资源-->  
<!--post 传输资源-->  
<!--put 更新资源-->  
<!--delete 删除资源-->  
<!--head 获得报文首部-->  
  
  
<!--post和get的区别-->  
<!--\*1、get在浏览器回退时基本不会重复提交，而post会再次提交请求-->  
<!--\*2、get产生的url地址可以被收藏。而post不可以-->  
<!--\*3、get请求会被浏览器主动缓存，而post不会，除非手动设置-->  
<!--4、get请求只能进行url编码，而post支持多种编码方式-->  
<!--\*5、get请求参数会被完整保留在浏览器历史记录里，而post中的参数不会被保留-->  
<!--\*6、get请求在url中传送的参数是有长度限制的（一般2kb但是根据具体情况），而psot没有限制-->  
<!--7、对参数的数据类型，get只能接受ASCLL字符，而POST没有限制-->  
<!--8、get比post更不安全，因为参数直接暴漏在url上，所以不能用了传递敏感信息-->  
<!--\*9、get参数通过url传递，post放在Request body中-->  
<!--\*号为必须记住-->  
  
  
<!--持久连接（http持久连接时1.1支持的，1.0不支持）-->  
<!--HTTP协议采用“请求-应答”模式，当使用普通模式，即非Keep-Alive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）-->  
<!--当使用Keep-Alive模式（又称持久连接、连接重用）时，Keep—Alive功能使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-->  
  
<!--管线化-->  
<!--管线化是指通过是持久建立的但是不是说请求一次响应一次；是把请求打包一次请求过去，响应打包一次响应过来-->  
  
<!--持球连接说明-->  
<!--在使用持久连接的情况下，某个连接上消息的传递类似于（说白了就是整个连接没有中断过）-->  
<!--请求1->响应1->请求2->响应2->请求3->响应3-->  
  
<!--管线化说明-->  
<!--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-->  
  
<!--管线化特点-->  
<!--1、管线化机制通过持久连接完成，仅HTTP/1.1支持-->  
<!--2、只有GET和HEAD请求可以进行管线化，而post则有所限制-->  
<!--3、初次创建连接时不应启动管线化机制，只因对方（服务器）不一定支持HTTP/1.1版本的协议-->  
<!--4、管线化不会影响响应到来的顺序-->  
<!--5、HTTP/1.1要求服务器段支持管线化，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-->  
<!--6、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默认未开启管线化支持-->  
<!--挑出几点记住就可以-->  
  
<!--常见的HTTP相应状态码-->  
<!--  
200：请求被正常处理  
204：请求被受理但没有资源可以返回  
206：客户端只是请求资源的一部分，服务器只对请求的部分资源执行GET方法，相应报文中通过Content-Range指定范围的资源。  
301：永久性重定向  
302：临时重定向  
303：与302状态码有相似功能，只是它希望客户端在请求一个URI的时候，能通过GET方法重定向到另一个URI上  
304：发送附带条件的请求时，条件不满足时返回，与重定向无关  
307：临时重定向，与302类似，只是强制要求使用POST方法  
400：请求报文语法有误，服务器无法识别  
401：请求需要认证  
403：请求的对应资源禁止被访问  
404：服务器无法找到对应资源  
500：服务器内部错误  
503：服务器正忙  
-->  
  
<!--HTTP1.1版本新特性  
a、默认持久连接节省通信量，只要客户端服务端任意一端没有明确提出断开TCP连接，就一直保持连接，可以发送多次HTTP请求  
b、管线化，客户端可以同时发出多个HTTP请求，而不用一个个等待响应  
c、断点续传原理-->  
  
<!--常见HTTP首部字段-->  
<!--  
a、通用首部字段（请求报文与响应报文都会使用的首部字段）  
Date：创建报文时间  
Connection：连接的管理  
Cache-Control：缓存的控制  
Transfer-Encoding：报文主体的传输编码方式  
  
b、请求首部字段（请求报文会使用的首部字段）  
Host：请求资源所在服务器  
Accept：可处理的媒体类型  
Accept-Charset：可接收的字符集  
Accept-Encoding：可接受的内容编码  
Accept-Language：可接受的自然语言  
  
c、响应首部字段（响应报文会使用的首部字段）  
Accept-Ranges：可接受的字节范围  
Location：令客户端重新定向到的URI  
Server：HTTP服务器的安装信息  
  
d、实体首部字段（请求报文与响应报文的的实体部分使用的首部字段）  
Allow：资源可支持的HTTP方法  
Content-Type：实体主类的类型  
Content-Encoding：实体主体适用的编码方式  
Content-Language：实体主体的自然语言  
Content-Length：实体主体的的字节数  
Content-Range：实体主体的位置范围，一般用于发出部分请求时使用  
-->  
  
<!--HTTP的缺点与HTTPS-->  
<!--  
a、通信使用明文不加密，内容可能被窃听  
b、不验证通信方身份，可能遭到伪装  
c、无法验证报文完整性，可能被篡改  
HTTPS就是HTTP加上加密处理（一般是SSL安全通信线路）+认证+完整性保护  
-->  
<!--http和https协议有什么区别，重点解释https-->  
<!--http+加密+认证+完整性保护=https-->  
<!--  
http:应用层的无状态，超文本传输协议。端口为80  
HTTPS：只是http通信接口部分用SSL和TLS协议替代。http直接和TCP通信，而HTTPS使用SSL所以是先和SSL通信，再由SSL和TCP通信。端口为443  
-->  
  
<!--HTML5本地存储分为-->  
<!--webStorage(localStorage,sessionStorage)；indexDB-->  
  
<!--DNS解析域名为IP-->  
<!--浏览器缓存中找；系统缓存中找；路由器缓存中找；ISP DNS缓存中找-->  
  
<!--TCP三次握手 了解即可-->  
<!--client->server:SYN(发起一个TCP连接，同步报文)；server->client:SYN+ACK(应答报文，表示已创建连接)；client->server:ACK(应答报文，表示收到已连接)-->  
  
<!--http全过程-->  
<!--输入域名(url)->DNS映射为IP->TCP三次握手->HTTP请求->HTTP响应->(浏览器跟踪重定向地址)->服务器处理请求->服务器返回一个html响应->(视情况决定释放TCP连接)->客户端解析HTML->获取嵌入在HTML中的对象重新发起http请求-->  
<!--同源策略-->  
<!--同源需满足的条件：协议相同；域名相同；端口相同-->  
<!--非同源受限制的行为：cookie localStorage indexDB无法获取；DOM无法获得；ajax请求不能发送-->  
  
  
<!--释放TCP连接-->  
<!--header中的connecton:close：服务器主动关闭TCP连接，客户端被动关闭连接-->  
<!--header中的connecton:keepalive：连接保持一段时间，可以连续发送http请求-->  
  
  
<!--客户端解析HTML-->  
<!--onload ready区别：-->  
<!--ready表示文档加载完毕，不包括图片；onload表示都加载完毕-->*

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。-->  
<!--上面这句话说白了就是不是同一个源的文档没有权利操作另一个源的文档-->  
<!--主要不能操作的有：Cookie、LocalStorage和IndexDB无法读取，Dom无法获得，AJAX请求不能发送-->  
<!--源包括协议、域名、端口-->  
  
<!--前后端如何通信：Ajax；WebSocket；CORS-->  
  
<!--跨域通信的几种方式-->  
<!--JSONP、Hash（特点：改变Hash页面不刷新，注意下search，search改变时刷新页面的不能用作跨域通信）、postMessage、WebSocket、CORS（可以理解为是支持跨域通信的Ajax，  
他的原理是浏览器在识别用ajax发送的一个跨域请求时，他会在http头中加一个origin来允许跨域通信）-->  
  
<!--JSONP 原理：就是利用script标签的异步加载来实现的-->  
<!--具体做法：首先需要给服务端传递一个回调的名例如callback=jsonp，然后利用script发出请求；然后服务器给你下发一个script内容，然后本地一定要有个jsonp的全局函数；后面才能执行服务端下发的内容，当函数去运行-->  
<!--//例子-->*

*<script src="http://www.abx.com/?data=name&callback=jsonp"></script>  
<!--<script>-->  
 <!--jsonp({-->  
 <!--data:{-->  
 <!--... -->  
 <!--}-->  
 <!--})-->  
<!--</script>-->  
  
<!--hash-->  
<!--场景是有一个页面A 通过iframe或frame嵌入了跨域的页面B-->  
<!--具体做法-->  
<!--在A页面里 你要拿到B的src,然后通过hash的方式加一个字符串,然后B在自己的代码中增加一个window.onhashchange事件，监听url是否有改变-->*<**script**>  
 *//在A中* **var *B*** = **document**.getElementsByTagName(**'iframe'**);  
 ***B***.**src**= ***B***.**src** + **'#'** + **'data'**;  
 *//在B中* **window**.onhashchange = **function**(){  
 **var** data = **window**.**location**.**hash** }  
</**script**>  
  
  
*<!--postMessage-->  
<!--场景窗口A（http:A.com）向跨域的窗口B（http://B.com）发送信息-->  
<!--具体做法，首先是在B窗口下调用用window.postMessage('data','http://B.com');在再B窗口下监听message事件-->*<**script**>  
 *//B窗口下调用* **window**.postMessage(**'data'**,**'http://B.com'**);  
 **window**.addEventListener(**'message'**,**function**(event){  
 **console**.log(event.**origin**)*//htttp://A.com* **console**.log(event.**source**)*//Awindow* **console**.log(event.**data**)*//data拿到传过来的数据* })  
</**script**>  
  
  
*<!--WebSocket 不受同源策略的限制 资料扩展可了解http://www.ruanyifeng.com/blog/2017/05/websocket.html-->  
<!--用法首先声明Websocket一个对象，然后做三个事一个是onopen的时候 把请求发出去，onmessage接收对方给的消息，onclose来确定是否关闭-->*<**script**>  
 *//例子* **var *ws*** = **new** WebSocket(**'wss://echo.websocket.org'**)*//这个ws和wss就是加密和非加密后面指向服务器的一个地址* ***ws***.onopen = **function**(evt){  
 **console**.log(**'Connection open...'**)  
 ***ws***.send(**'jizognquan'**)  
 }  
 ***ws***.onmessage = **function**(evt){  
 **console**.log(**"Received Message"**+evt.**data**)  
 ***ws***.close()  
 }  
 ***ws***.onclose = **function**(evt){  
 **console**.log(**'Connection closed'**)  
 }  
</**script**>  
  
*<!--CORS 就是跨域资源共享-->  
<!--为什么说cors支持跨域通信：浏览器会拦截ajax请求，如果他认为ajax请求时跨域的，他会在http头中加一个origin-->  
<!--以下挑流程话进行记忆，过程只是帮助理解-->  
<!--CORS需要浏览器和服务器同时支持。目前，所有浏览器都支持该功能，IE浏览器不能低于IE10。-->  
<!--可以理解为是支持跨域通信的Ajax，他的原理是浏览器在识别用ajax发送的一个跨域请求时，他会在http头中加一个origin来允许跨域通信-->  
<!--这里会用到新的标准fetch，用来实现cors-->*<**script**>  
 fetch(**'/some/url'**,{  
 **method**:**'get'** }).then(**function**(response){  
  
 }).catch(**function**(err){  
  
 })  
</**script**>  
*<!--浏览器将CORS请求分成两类：简单请求（simple request）和非简单请求（not-so-simple request）。-->  
<!--简单请求:-->  
<!--  
（1) 请求方法是以下三种方法之一：HEAD ,GET ,POST  
（2）HTTP的头信息不超出以下几种字段：Accept,Accept-Language,Content-Language,Last-Event-ID,Content-Type：只限于三个值application/x-www-form-urlencoded、multipart/form-data、text/plain-->  
<!--凡是不同时满足上面两个条件，就属于非简单请求。-->  
<!--简单请求流程：对于简单请求，浏览器直接发出CORS请求。具体来说，就是在头信息之中，增加一个Origin字段。Origin字段用来说明，本次请求来自哪个源（协议 + 域名 + 端口）。服务器根据这个值，决定是否同意这次请求。-->  
<!--如果Origin指定的源，不在许可范围内，服务器会返回一个正常的HTTP回应。浏览器发现，这个回应的头信息没有包含Access-Control-Allow-Origin字段（详见下文），就知道出错了，从而抛出一个错误，被XMLHttpRequest的onerror回调函数捕获。注意，这种错误无法通过状态码识别，因为HTTP回应的状态码有可能是200。-->  
<!--如果Origin指定的域名在许可范围内，服务器返回的响应，会多出几个头信息字段。Access-Control-Allow-Origin: http://api.bob.com；Access-Control-Allow-Credentials: true；Access-Control-Expose-Headers: FooBar；Content-Type: text/html; charset=utf-8-->  
<!--  
上面的头信息之中，有三个与CORS请求相关的字段，都以Access-Control-开头。  
  
（1）Access-Control-Allow-Origin；该字段是必须的。它的值要么是请求时Origin字段的值，要么是一个\*，表示接受任意域名的请求。  
  
（2）Access-Control-Allow-Credentials；该字段可选。它的值是一个布尔值，表示是否允许发送Cookie。默认情况下，Cookie不包括在CORS请求之中。设为true，即表示服务器明确许可，Cookie可以包含在请求中，  
 一起发给服务器。这个值也只能设为true，如果服务器不要浏览器发送Cookie，删除该字段即可。  
  
（3）Access-Control-Expose-Headers；该字段可选。CORS请求时，XMLHttpRequest对象的getResponseHeader()方法只能拿到6个基本字段：  
 Cache-Control、Content-Language、Content-Type、Expires、Last-Modified、Pragma。如果想拿到其他字段，就必须在Access-Control-Expose-Headers里面指定。上面的例子指定，getResponseHeader('FooBar')可以返回FooBar字段的值。-->  
  
<!--CORS请求默认不发送Cookie和HTTP认证信息。如果要把Cookie发到服务器，一方面要服务器同意，指定Access-Control-Allow-Credentials字段。Access-Control-Allow-Credentials: true；另一方面，开发者必须在AJAX请求中打开withCredentials属性。  
var xhr = new XMLHttpRequest();  
xhr.withCredentials = true;  
否则，即使服务器同意发送Cookie，浏览器也不会发送。或者，服务器要求设置Cookie，浏览器也不会处理。但是，如果省略withCredentials设置，有的浏览器还是会一起发送Cookie。这时，可以显式关闭withCredentials。  
xhr.withCredentials = false;  
需要注意的是，如果要发送Cookie，Access-Control-Allow-Origin就不能设为星号，必须指定明确的、与请求网页一致的域名。  
同时，Cookie依然遵循同源政策，只有用服务器域名设置的Cookie才会上传，其他域名的Cookie并不会上传，且（跨源）原网页代码中的document.cookie也无法读取服务器域名下的Cookie。-->  
  
<!--非简单请求-->  
<!--  
非简单请求是那种对服务器有特殊要求的请求，比如请求方法是PUT或DELETE，或者Content-Type字段的类型是application/json。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。  
-->*<**script**>  
 *//例子* **var *url*** = **'http://api.alice.com/cors'**;  
 **var *xhr*** = **new** XMLHttpRequest();  
 ***xhr***.open(**'PUT'**, ***url***, **true**);  
 ***xhr***.setRequestHeader(**'X-Custom-Header'**, **'value'**);  
 ***xhr***.send();  
</**script**>  
*<!--上面代码中，HTTP请求的方法是PUT，并且发送一个自定义头信息X-Custom-Header。浏览器发现，这是一个非简单请求，就自动发出一个"预检"请求，要求服务器确认可以这样请求。下面是这个"预检"请求的HTTP头信息。-->  
<!--  
 OPTIONS /cors HTTP/1.1  
 Origin: http://api.bob.com  
 Access-Control-Request-Method: PUT  
 Access-Control-Request-Headers: X-Custom-Header  
 Host: api.alice.com  
 Accept-Language: en-US  
 Connection: keep-alive  
 User-Agent: Mozilla/5.0...  
-->  
<!--"预检"请求用的请求方法是OPTIONS，表示这个请求是用来询问的。头信息里面，关键字段是Origin，表示请求来自哪个源。除了Origin字段，"预检"请求的头信息包括两个特殊字段。  
（1）Access-Control-Request-Method:该字段是必须的，用来列出浏览器的CORS请求会用到哪些HTTP方法，上例是PUT。  
（2）Access-Control-Request-Headers:该字段是一个逗号分隔的字符串，指定浏览器CORS请求会额外发送的头信息字段，上例是X-Custom-Header。-->  
  
<!--服务器收到"预检"请求以后，检查了Origin、Access-Control-Request-Method和Access-Control-Request-Headers字段以后，确认允许跨源请求，就可以做出回应。-->  
  
<!--  
 HTTP/1.1 200 OK  
 Date: Mon, 01 Dec 2008 01:15:39 GMT  
 Server: Apache/2.0.61 (Unix)  
 Access-Control-Allow-Origin: http://api.bob.com  
 Access-Control-Allow-Methods: GET, POST, PUT  
 Access-Control-Allow-Headers: X-Custom-Header  
 Content-Type: text/html; charset=utf-8  
 Content-Encoding: gzip  
 Content-Length: 0  
 Keep-Alive: timeout=2, max=100  
 Connection: Keep-Alive  
 Content-Type: text/plain  
-->  
<!--上面的HTTP回应中，关键的是Access-Control-Allow-Origin字段，表示http://api.bob.com可以请求数据。该字段也可以设为星号，表示同意任意跨源请求。-->  
<!--如果浏览器否定了"预检"请求，会返回一个正常的HTTP回应，但是没有任何CORS相关的头信息字段。这时，浏览器就会认定，服务器不同意预检请求，因此触发一个错误，被XMLHttpRequest对象的onerror回调函数捕获。控制台会打印出如下的报错信息。-->  
<!--XMLHttpRequest cannot load http://api.alice.com.Origin http://api.bob.com is not allowed by Access-Control-Allow-Origin.-->  
  
<!--一旦服务器通过了"预检"请求，以后每次浏览器正常的CORS请求，就都跟简单请求一样，会有一个Origin头信息字段。服务器的回应，也都会有一个Access-Control-Allow-Origin头信息字段。-->  
<!--下面是"预检"请求之后，浏览器的正常CORS请求。-->  
<!--  
 PUT /cors HTTP/1.1  
 Origin: http://api.bob.com  
 Host: api.alice.com  
 X-Custom-Header: value  
 Accept-Language: en-US  
 Connection: keep-alive  
 User-Agent: Mozilla/5.0...  
-->  
<!--上面头信息的Origin字段是浏览器自动添加的。下面是服务器正常的回应。-->  
<!--  
 Access-Control-Allow-Origin: http://api.bob.com  
 Content-Type: text/html; charset=utf-8  
-->  
<!--上面头信息中，Access-Control-Allow-Origin字段是每次回应都必定包含的。-->  
<!--与JSONP的比较:CORS与JSONP的使用目的相同，但是比JSONP更强大。JSONP只支持GET请求，CORS支持所有类型的HTTP请求。JSONP的优势在于支持老式浏览器，以及可以向不支持CORS的网站请求数据。-->*

错误监控

*<!--前端错误的分类-->  
<!--1、即时运行错误：代码错误；2、资源加载错误-->  
  
<!--错误的捕获方式：1、即时运行错误的捕获方式；2、资源加载错误-->  
<!--即时运行错误的捕获方式-->  
<!--1、try...catch 2、window.onerror-->  
<!--资源记载错误-->  
<!--1、object.onerror 2、performance.getEntries()这个是高级浏览器的 3、Error事件捕获-->  
  
<!--跨域的js运行错误可以捕获吗，错误提示什么，应该怎么处理-->  
<!--也是可以捕获的，但是只能拿到一个错误信息叫做sript error，其他拿不到具体信息-->  
<!--处理：1、在客户端做的是在script标签增加crossorigin属性；2、在服务端做的设置js资源响应头Access-Control-Allow-Origin：后面可以是\*也可以是指定的域名-->  
  
  
<!--上报错误的基本原理-->  
<!--1、采用Ajax通信的方式上报-->  
<!--2、利用Image对象上报-->*<**script**>  
 *//例子* (**new Image**()).**src**=**'http://baidu.com'***//src是你想要指定的路径我这里以百度为例*</**script**>

页面性能

*<!--提升页面性能的方法有哪些-->  
<!--1、资源压缩合并，减少HTTP请求-->  
<!--2、非核心代码异步加载-->  
<!--3、利用浏览器缓存-->  
<!--4、使用CDN-->  
<!--5、预解析DNS(下面两种)-->  
<!--<meta http-equiv='x-dns-prefetch-control' content='on'> 这个是强制打开a标签DNS的预解析-->  
<!--<link rel='dns-prefetch' href='//host\_name\_to\_prefetch.com'>-->  
  
<!--异步加载的方式-->  
<!--1、动态脚本加载-->  
<!--2、defer，加在script标签里-->  
<!--3、async 加在script标签里-->  
  
<!--异步加载的区别-->  
<!--1、defer是在HTML解析完之后才会执行，如果是多个，安装加载的顺序依次执行-->  
<!--2、async是在加载完之后立即执行，如果是多个，执行顺序和加载顺序无关-->  
  
<!--浏览器缓存-->  
<!--缓存的分类:强缓存和协商缓存-->  
  
<!--强缓存:直接缓存，如果下面两个同时下发以后者为准-->  
<!--1、Expires翻译过来就是过期时间，这里是绝对时间-->  
<!--2、Cache-Control这里是相对时间-->  
  
<!--协商缓存：指的是和服务器协商-->  
<!--Last-Modified（这个是服务器下发的） if-Modified-Since（这个是请求的时候带着的）这个就是指上次修改的时间然后这两个做对比-->  
<!--Etag（这个是hash值） if-None-Match-->  
  
<!--跟缓存相关的http协议头有哪些-->  
<!--Last-Modified；if-Modified-Since；Etag；Expires；Cache-Control；if-None-Match-->*

算法类

*<!--算法会有：排序；堆栈、队列、链表；递归；波兰式和逆波兰式-->  
  
<!--排序-->  
  
<!--快速排序-->  
<!--算法步骤：  
1、先从数列中取出一个数作为“基准”。  
2、分区过程：将比这个“基准”大的数全放到“基准”的右边，小于或等于“基准”的数全放到“基准”的左边。  
3、再对左右区间重复第二步，直到各区间只有一个数。-->  
<!--快速算法简单说就是 选一个元素作为基准，将所有小于基准的元素放在左边，所有大于基准的 元素放在右边，然后对基准左右两边的子集不断重复这个操作，直到所有子集剩下一个元素，这个过程就可以理解为递归的过程-->*<**script**>  
 **var *arr1*** =[4,8,2,1,3];  
 **var** *quickSort* = **function**(arr) {  
 **if** (arr.**length** <= 1) { **return** arr; }  
 **var** pivotIndex = **Math**.floor(arr.**length** / 2); *//基准位置（理论上可任意选取）* **var** pivot = arr.splice(pivotIndex, 1)[0]; *//基准数* **var** left = [];  
 **var** right = [];  
 **for** (**var** i = 0; i < arr.**length**; i++){  
 **if** (arr[i] < pivot) {  
 left.push(arr[i]);  
 } **else** {  
 right.push(arr[i]);  
 }  
 }  
 **return** *quickSort*(left).concat([pivot], *quickSort*(right)); *//链接左数组、基准数构成的数组、右数组* };  
 **console**.log(quicksort(***arr1***))  
</**script**>  
  
*<!--冒泡排序-->  
<!--比较相邻的两个元素，如果后面的比前面的小，则交换位置。依照这个规则进行多次并且递减的迭代，直到顺序正确。每次将剩下数组里面最大的一个数排到最后面-->*

<**script**>  
 **var *arr*** = [3,6,1,2,5];  
 **var *temp***;  
 **for**(**var *i***= 0;***i***<***arr***.**length**;***i***++){  
 **for**(**var *j***=***i***;***j***<***arr***.**length**;***j***++){  
 **if**(***arr***[***i***] > ***arr***[***j***]){  
 ***temp*** = ***arr***[***i***];  
 ***arr***[***i***] = ***arr***[***j***];  
 ***arr***[***j***] = ***temp***;  
 }  
 }  
 }  
 **console**.log(***arr***);  
</**script**>  
  
  
*<!--选择排序法-->  
<!--在未排序序列中找到最小（大）元素，存放到排序序列的起始位置从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。重复第二步，直到所有元素均排序完毕。-->*<**script**>  
 **function** *selectionSort*(arr) {  
 **var** len = arr.**length**;  
 **var** minIndex, temp;  
 **for** (**var** i = 0; i < len - 1; i++) {  
 minIndex = i;  
 **for** (**var** j = i + 1; j < len; j++) {  
 **if** (arr[j] < arr[minIndex]) { *// 寻找最小的数* minIndex = j; *// 将最小数的索引保存* }  
 }  
 temp = arr[i];  
 arr[i] = arr[minIndex];  
 arr[minIndex] = temp;  
 }  
 **return** arr;  
 }  
</**script**>  
  
  
*<!--希尔排序 选择记忆前三种即可-->  
<!--希尔排序，也称递减增量排序算法;但希尔排序是非稳定排序算法-->*

*<!--选择一个增量序列 t1，t2，……，tk，其中 ti > tj, tk = 1；按增量序列个数 k，对序列进行 k 趟排序；  
每趟排序，根据对应的增量 ti，将待排序列分割成若干长度为 m 的子序列，分别对各子表进行直接插入排序。仅增量因子为 1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。-->*<**script**>  
 **function** *shellSort*(arr) {  
 **var** len = arr.**length**,  
 temp,  
 gap = 1;  
 **while**(gap < len/3) { *//动态定义间隔序列* gap = gap\*3+1;  
 }  
 **for** (gap; gap > 0; gap = **Math**.floor(gap/3)) {  
 **for** (**var** i = gap; i < len; i++) {  
 temp = arr[i];  
 **for** (**var** j = i-gap; j >= 0 && arr[j] > temp; j -= gap) {  
 arr[j+gap] = arr[j];  
 }  
 arr[j+gap] = temp;  
 }  
 }  
 **return** arr;  
 }  
</**script**>  
  
*<!--以下内容 稍作了解即可不用背，但是上面的必须掌握-->  
<!--堆栈-->  
<!--对于一个栈，我们需要实现添加、删除元素、获取栈顶元素、已经是否为空，栈的长度、清除元素等几个基本操作。下面是基本定义。-->*<**script**>  
 **function** *Stack*(){  
 **this**.**items** = [];  
 }  
 *Stack*.**prototype** = {  
 constructor:*Stack*,  
 push:**function**(element){  
 **this**.**items**.push(element);  
 },  
 pop:**function**(){  
 **return this**.**items**.pop();  
 },  
 peek:**function**(){  
 **return this**.**items**[**this**.**items**.**length** - 1];  
 },  
 isEmpty:**function**(){  
 **return this**.**items**.**length** == 0;  
 },  
 clear:**function**(){  
 **this**.**items** = [];  
 },  
 size:**function**(){  
 **return this**.**items**.**length**;  
 },  
 print:**function**(){  
 **console**.log(**this**.**items**.toString());  
 }  
 }  
</**script**>  
*<!--栈的基本操作。-->*<**script**>  
 **var *stack*** = **new** *Stack*();  
 **console**.log(***stack***.isEmpty());*//true* ***stack***.push(5);  
 ***stack***.push(8);  
 **console**.log(***stack***.peek());*//8* ***stack***.push(11);  
 **console**.log(***stack***.size());*//3* **console**.log(***stack***.isEmpty());  
 ***stack***.push(15);  
 ***stack***.pop();  
 ***stack***.pop();  
 **console**.log(***stack***.size());*//2* **console**.log(***stack***.print());*//5,8*</**script**>  
*<!--通过栈实现对正整数的二进制转换。-->*<**script**>  
 **function** *divideBy2*(decNumber){  
 **var** decStack = **new** *Stack*();  
 **var** rem;  
 **var** decString = **''**;  
 **while**(decNumber > 0){  
 rem = decNumber%2;  
 decStack.push(rem);  
 decNumber = **Math**.floor(decNumber/2);  
 }  
 **while**(!decStack.isEmpty()){  
 decString += decStack.pop().toString();  
 }  
 **return** decString;  
 }  
 **console**.log(*divideBy2*(10));*//1010*</**script**>  
  
  
*<!--队列-->  
<!--队列是遵循FIFO(First In First Out,先进先出,也称为先来先服务)原则的一组有序的项。队列在尾部添加新元素,并从顶部移除元素。最新添加的元素必须排在队列的末尾。队列要实现的操作基本和栈一样，只不过栈是FILO(先进后出)。-->*<**script**>  
 **function** *Queue*(){  
 **this**.**items** = [];  
 }  
 *Queue*.**prototype** = {  
 constructor:*Queue*,  
 enqueue:**function**(elements){  
 **this**.**items**.push(elements);  
 },  
 dequeue:**function**(){  
 **return this**.**items**.shift();  
 },  
 front:**function**(){  
 **return this**.**items**[0];  
 },  
 isEmpty:**function**(){  
 **return this**.**items**.**length** == 0;  
 },  
 size:**function**(){  
 **return this**.**items**.**length**;  
 },  
 clear:**function**(){  
 **this**.**items** = [];  
 },  
 print:**function**(){  
 **console**.log(**this**.**items**.toString());  
 }  
 }  
</**script**>  
*<!--队列的基本使用-->*<**script**>  
 **var *queue*** = **new** *Queue*();  
 **console**.log(***queue***.isEmpty());*//true* ***queue***.enqueue(**'huang'**);  
 ***queue***.enqueue(**'cheng'**);  
 **console**.log(***queue***.print());*//huang,cheng* **console**.log(***queue***.size());*//2* **console**.log(***queue***.isEmpty());*//false* ***queue***.enqueue(**'du'**);  
 **console**.log(***queue***.dequeue());*//huang* **console**.log(***queue***.print());*//cheng,du*</**script**>  
*<!--优先队列-->  
<!--元素的添加和移除是基于优先级的。实现一个优先队列,有两种选项:设置优先级,然后在正确的位置添加元素;或者用入列操 作添加元素,然后按照优先级移除它们。 我们在这里实现的优先队列称为最小优先队列,因为优先级的值较小的元素被放置在队列最 前面(1代表更高的优先级)。最大优先队列则与之相反,把优先级的值较大的元素放置在队列最 前面。-->  
<!--优先队列的定义-->  
<!--我们在这里使用组合继承的方式继承自Queue队列。-->*<**script**>  
 **function** *PriorityQueue*(){  
 *Queue*.call(**this**);  
 };  
 *PriorityQueue*.**prototype** = **new** *Queue*();  
 *PriorityQueue*.**prototype**.constructer = *PriorityQueue*;  
 *PriorityQueue*.**prototype**.enqueue = **function**(element,priority){  
 **function** *QueueElement*(tempelement,temppriority){  
 **this**.**element** = tempelement;  
 **this**.**priority** = temppriority;  
 }  
 **var** queueElement = **new** *QueueElement*(element,priority);  
 **if**(**this**.isEmpty()){  
 **this**.**items**.push(queueElement);  
 }**else**{  
 **var** added = **false**;  
 **for**(**var** i = 0; i < **this**.**items**.**length**;i++){  
 **if**(**this**.**items**[i].**priority** > queueElement.**priority**){  
 **this**.**items**.splice(i,0,queueElement);  
 added = **true**;  
 **break**;  
 }  
 }  
 **if**(!added){  
 **this**.**items**.push(queueElement);  
 }  
 }  
  
 }  
 *//这个方法可以用Queue的默认实现  
 PriorityQueue*.**prototype**.print = **function**(){  
 **var** result =**''**;  
 **for**(**var** i = 0; i < **this**.**items**.**length**;i++){  
 result += **JSON**.stringify(**this**.**items**[i]);  
 }  
 **return** result;  
 }  
</**script**>  
*<!--优先队列的基本使用-->*<**script**>  
 **var *priorityQueue*** = **new** *PriorityQueue*();  
 ***priorityQueue***.enqueue(**"cheng"**, 2);  
 ***priorityQueue***.enqueue(**"du"**, 3);  
 ***priorityQueue***.enqueue(**"huang"**, 1);  
 **console**.log(***priorityQueue***.print());*//{"element":"huang","priority":1}{"element":"cheng","priority":2}{"element":"du","priority":3}* **console**.log(***priorityQueue***.size());*//3* **console**.log(***priorityQueue***.dequeue());*//{ element="huang", priority=1}* **console**.log(***priorityQueue***.size());*//2*</**script**>  
  
  
*<!--链表-->  
<!--数组的大小是固定的,从数组的起点或中间插入 或移除项的成本很高,因为需要移动元素(尽管我们已经学过的JavaScript的Array类方法可以帮 我们做这些事,但背后的情况同样是这样)。  
链表存储有序的元素集合,但不同于数组,链表中的元素在内存中并不是连续放置的。每个 元素由一个存储元素本身的节点和一个指向下一个元素的引用(也称指针或链接)组成。  
相对于传统的数组,链表的一个好处在于,添加或移除元素的时候不需要移动其他元素。然 而,链表需要使用指针,因此实现链表时需要额外注意。  
数组的另一个细节是可以直接访问任何 位置的任何元素,而要想访问链表中间的一个元素,需要从起点(表头)开始迭代列表直到找到 所需的元素-->  
<!--链表的创建-->  
<!--我们使用动态原型模式来创建一个链表。列表最后一个节点的下一个元素始终是null。-->*<**script**>  
 **function** *LinkedList*(){  
 **function** *Node*(element){  
 **this**.**element** = element;  
 **this**.**next** = **null**;  
 }  
 **this**.**head** = **null**;  
 **this**.**length** = 0;  
 *//通过对一个方法append判断就可以知道是否设置了prototype* **if**((**typeof this**.append !== **'function'**)&&(**typeof this**.append !== **'string'**)){  
 *//添加元素  
 LinkedList*.**prototype**.append = **function**(element){  
 **var** node = **new** *Node*(element);  
 **var** current;  
 **if**(**this**.**head** === **null**){  
 **this**.**head** = node;  
 }**else**{  
 current = **this**.**head**;  
 **while**(current.**next** !== **null**){  
 current = current.**next**;  
 }  
 current.**next** = node;  
 }  
 **this**.**length**++;  
 };  
 *//插入元素，成功true，失败false  
 LinkedList*.**prototype**.insert = **function**(position,element){  
 **if**(position > -1 && position < **this**.**length**){  
 **var** current = **this**.**head**;  
 **var** previous;  
 **var** index = 0;  
 **var** node = **new** *Node*(element);  
 **if**(position == 0){  
 node.**next** = current;  
 **this**.**head** = node;  
 }**else**{  
 **while**(index++ < position){  
 previous = current;  
 current = current.**next**;  
 }  
 node.**next** = current;  
 previous.**next** = node;  
 }  
 **this**.**length**++;  
 **return true**;  
 }**else**{  
 **return false**;  
 }  
 };  
 *//根据位置删除指定元素，成功 返回元素， 失败 返回null  
 LinkedList*.**prototype**.removeAt = **function**(position){  
 **if**(position > -1 && position < **this**.**length**){  
 **var** current = **this**.**head**;  
 **var** previous = **null**;  
 **var** index = 0;  
 **if**(position == 0){  
 **this**.**head** = current.**next**;  
 }**else**{  
 **while**(index++ < position){  
 previous = current;  
 current = current.**next**;  
 }  
 previous.**next** = current.**next**;  
 }  
 **this**.**length**--;  
 **return** current.**element**;  
 }**else**{  
 **return null**;  
 }  
 };  
 *//根据元素删除指定元素，成功 返回元素， 失败 返回null  
 LinkedList*.**prototype**.remove = **function**(element){  
 **var** index = **this**.indexOf(element);  
 **return this**.removeAt(index);  
 };  
 *//返回给定元素的索引，如果没有则返回-1  
 LinkedList*.**prototype**.indexOf = **function**(element){  
 **var** current = **this**.**head**;  
 **var** index = 0;  
 **while**(current){  
 **if**(current.**element** === element){  
 **return** index;  
 }  
 index++;  
 current = current.**next**;  
 }  
 **return** -1;  
 };  
 *LinkedList*.**prototype**.isEmpty = **function**(){  
 **return this**.**length** === 0;  
 };  
 *LinkedList*.**prototype**.size = **function**(){  
 **return this**.**length**;  
 };  
 *LinkedList*.**prototype**.toString = **function**(){  
 **var** string = **''**;  
 **var** current = **this**.**head**;  
 **while**(current){  
 string += current.**element**;  
 current = current.**next**;  
 }  
 **return** string;  
 };  
 *LinkedList*.**prototype**.getHead = **function**(){  
 **return this**.**head**;  
 };  
 }  
 }  
</**script**>  
*<!--链表的基本使用-->*<**script**>  
 **var *linkedList*** = **new** *LinkedList*();  
 **console**.log(***linkedList***.isEmpty());*//true;* ***linkedList***.append(**'huang'**);  
 ***linkedList***.append(**'du'**)  
 ***linkedList***.insert(1,**'cheng'**);  
 **console**.log(***linkedList***.toString());*//huangchengdu* **console**.log(***linkedList***.indexOf(**'du'**));*//2* **console**.log(***linkedList***.size());*//3* **console**.log(***linkedList***.removeAt(2));*//du* **console**.log(***linkedList***.toString());*//huangcheng*</**script**>  
  
*<!--双向链表的创建-->  
<!--链表有多种不同的类型,这一节介绍双向链表。双向链表和普通链表的区别在于,在链表中, 一个节点只有链向下一个节点的链接,而在双向链表中,链接是双向的:一个链向下一个元素, 另一个链向前一个元素。  
双向链表和链表的区别就是有一个tail属性，所以必须重写insert、append、removeAt方法。每个节点对应的Node也多了一个prev属性。-->*<**script**>  
 *//寄生组合式继承实现，详见javascript高级程序设计第七章* **function** *inheritPrototype*(subType, superType) {  
 **function** *object*(o) {  
 **function** *F*() {}  
 *F*.**prototype** = o;  
 **return new** *F*();  
 }  
 **var** prototype = *object*(superType.**prototype**);  
 prototype.**constructor** = subType;  
 subType.**prototype** = prototype;  
 }  
 **function** *DoublyLinkedList*() {  
 **function** *Node*(element) {  
 **this**.**element** = element;  
 **this**.**next** = **null**;  
 **this**.**prev** = **null**;  
 }  
 **this**.**tail** = **null**;  
 *LinkedList*.call(**this**);  
 *//与LinkedList不同的方法自己实现。* **this**.insert = **function**(position, element) {  
 **if** (position > -1 && position <= **this**.**length**) {  
 **var** node = **new** *Node*(element);  
 **var** current = **this**.**head**;  
 **var** previous;  
 **var** index = 0;  
 **if** (position === 0) {  
 **if** (!**this**.**head**) {  
 **this**.**head** = node;  
 **this**.**tail** = node;  
 } **else** {  
 node.**next** = current;  
 current.**prev** = node;  
 **this**.**head** = node;  
 }  
 } **else if** (position == **this**.**length**) {  
 current = **this**.**tail**;  
 current.**next** = node;  
 node.**prev** = current;  
 **this**.**tail** = node;  
 } **else** {  
 **while** (index++ < position) {  
 previous = current;  
 current = current.**next**;  
 }  
 previous.**next** = node;  
 node.**next** = current;  
 current.**prev** = node;  
 node.**prev** = previous;  
 }  
 **this**.**length**++;  
 **return true**;  
 } **else** {  
 **return false**;  
 }  
 };  
 **this**.append = **function**(element) {  
 **var** node = **new** *Node*(element);  
 **var** current;  
 **if** (**this**.**head** === **null**) {  
 **this**.**head** = node;  
 **this**.**tail** = node;  
 } **else** {  
 current = **this**.**head**;  
 **while** (current.**next** !== **null**) {  
 current = current.**next**;  
 }  
 current.**next** = node;  
 node.**prev** = current;  
 **this**.**tail** = node;  
 }  
 **this**.**length**++;  
 };  
 **this**.removeAt = **function**(position) {  
 **if** (position > -1 && position < **this**.**length**) {  
 **var** current = **this**.**head**;  
 **var** previous;  
 **var** index = 0;  
 **if** (position === 0) {  
 **this**.**head** = current.**next**;  
 **if** (**this**.**length** === 1) {  
 **this**.**tail** = **null**;  
 } **else** {  
 **this**.**head**.**prev** = **null**;  
 }  
 } **else if** (position === (**this**.**length** - 1)) {  
 current = **this**.**tail**;  
 **this**.**tail** = current.**prev**;  
 **this**.**tail**.**next** = **null**;  
 } **else** {  
 **while** (index++ < position) {  
 previous = current;  
 current = current.**next**;  
 }  
 previous.**next** = current.**next**;  
 current.**next**.**prev** = previous;  
 }  
 **this**.**length**--;  
 **return** current.**element**;  
 } **else** {  
 **return false**;  
 }  
 };  
 }  
 *inheritPrototype*(*DoublyLinkedList*, *LinkedList*);  
</**script**>  
*<!--双向链表的基本使用-->*<**script**>  
 **var *doublyList*** = **new** *DoublyLinkedList*();  
 **console**.log(***doublyList***.isEmpty()); *//true;* ***doublyList***.append(**'huang'**);  
 ***doublyList***.append(**'du'**)  
 ***doublyList***.insert(1, **'cheng'**);  
 **console**.log(***doublyList***.toString()); *//huangchengdu* **console**.log(***doublyList***.indexOf(**'du'**)); *//2* **console**.log(***doublyList***.size()); *//3* **console**.log(***doublyList***.removeAt(2)); *//du* **console**.log(***doublyList***.toString()); *//huangcheng*</**script**>  
*<!-- 循环链表-->  
<!--循环链表可以像链表一样只有单向引用,也可以像双向链表一样有双向引用。  
循环链表和链 表之间唯一的区别在于,最后一个元素指向下一个元素的指针(tail.next)不是引用null, 而是指向第一个元素(head)。  
双向循环链表有指向head元素的tail.next,和指向tail元素的head.prev。-->*

mvvm

*<!--首先MVC（Model view constructor）-->  
<!--MVVM就是Model View ViewModel-->  
  
<!--双向绑定原理-->  
<!--view->data;data->view-->*<**script**>  
 *//Object.defineProperty的作用他能监听到data变化，同时它在监听变化的时候有个回调函数，在这个回调函数里面直接写出view和data的关联关系  
 //vue如何实现双向数据绑定的  
 //Object.defineProperty(obj，prop，descriptor)函数* **var *obj*** ={**pwd**:**"123456"**};  
 Object.defineProperty(***obj***,**"userNames"**,{  
 **set**:**function**(val){  
 **console**.log(**"set init"**);  
 **document**.getElementById(**"uName"**).**innerText**=val;  
  
 },  
 **get**:**function**(){  
 **console**.log(**"get init"**);  
 }  
 });  
 **document**.getElementById(**"userName"**).addEventListener(**"keyup"**,**function**(event){  
 ***obj***.**userNames**=event.**target**.**value**;  
 })  
</**script**>  
*<!--属于观察者模式，第一要个监听者，发生变化后会通知所有的观察者列表-->*

页面性能

*<!--提升页面性能的方法有哪些-->  
<!--1、资源压缩合并，减少HTTP请求-->  
<!--2、非核心代码异步加载-->  
<!--3、利用浏览器缓存-->  
<!--4、使用CDN-->  
<!--5、预解析DNS(下面两种)-->  
<!--<meta http-equiv='x-dns-prefetch-control' content='on'> 这个是强制打开a标签DNS的预解析-->  
<!--<link rel='dns-prefetch' href='//host\_name\_to\_prefetch.com'>-->  
  
<!--异步加载的方式-->  
<!--1、动态脚本加载-->  
<!--2、defer，加在script标签里-->  
<!--3、async 加在script标签里-->  
  
<!--异步加载的区别-->  
<!--1、defer是在HTML解析完之后才会执行，如果是多个，安装加载的顺序依次执行-->  
<!--2、async是在加载完之后立即执行，如果是多个，执行顺序和加载顺序无关-->  
  
<!--浏览器缓存-->  
<!--缓存的分类:强缓存和协商缓存-->  
  
<!--强缓存:直接缓存，如果下面两个同时下发以后者为准-->  
<!--1、Expires翻译过来就是过期时间，这里是绝对时间-->  
<!--2、Cache-Control这里是相对时间-->  
  
<!--协商缓存：指的是和服务器协商-->  
<!--Last-Modified（这个是服务器下发的） if-Modified-Since（这个是请求的时候带着的）这个就是指上次修改的时间然后这两个做对比-->  
<!--Etag（这个是hash值） if-None-Match-->  
  
<!--跟缓存相关的http协议头有哪些-->  
<!--Last-Modified；if-Modified-Since；Etag；Expires；Cache-Control；if-None-Match-->*