1、左两行右一行的布局（仅限三个div）

<div class=”d1”></div>

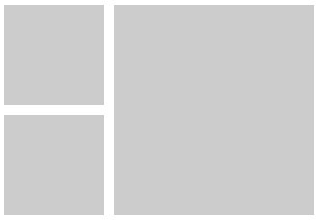
<div class=”d2”></div>

<div class=”d3”></div>

.div{background:#ccc;}

.d1, .d2{width:100px; height:100px; float: left; }

.d2{clear:both; margin-top:10px;}

.d3{width:200px;height:210px; margin-left: 110px;}

2、js的2种继承方式详解

*//对象冒充包括三种  
//通过对象冒充的方式，无法继承通过prototype方式定义的变量和方法***function** *Person*(name) {  
 **this**.**name** = name;  
 **this**.say = **function**(){  
 **console**.log(**"My name is "**+**this**.**name**);  
 }

}

**function** *F2E*(name, id) {  
 *//1.临时属性方式  
 //this.temp = Person;  
 //this.temp(name);  
 //delete this.temp;  
 //2 call()、3apply()方式  
 Person*.call(**this**,name);*//apply()方式 -> Person.call(name,new Array(name));* **this**.**id** = id;  
 **this**.showId = **function**(){  
 **console**.log(**'Good morning, Sir, My work number is '**+**this**.**id**)  
 }

}

**var *simon*** = **new** *F2E*(**'Simon'**,998);  
***simon***.say();  
***simon***.showId();

在OO概念中，new实例化后，对象就在堆内存中形成了自己的空间，值得注意的是，这个代码段。而成员方法就是存在这个代码段的，并且方法是共用的。问题就在这里，通过对象冒充方式继承时，所有的成员方法都是指向this的，也就是说new之后，每个实例将都会拥有这个成员方法，并不是共用的，这就造成了大量的内存浪费。并且通过对象冒充的方式，无法继承通过prototype方式定义的变量和方法。

*//原型链方式：***function** *Person2*(){  
 **this**.**name** = **'Simon'**;  
}  
*Person2*.**prototype**.say = **function** () {  
 **console**.log(**"My name is "**+**this**.**name**)  
}  
**function** *F2E2*(id) {  
 **this**.**id** = id;  
 **this**.showId = **function** () {  
 **console**.log(**'Good morning,Sir,My work number is '**+**this**.**id**);  
 }

}

*F2E2*.**prototype** = **new** *Person2*();  
**var *simon2*** = **new** *F2E2*(998);  
***simon2***.say();  
***simon2***.showId();  
**console**.log(***simon2***.hasOwnProperty(**'id'**));  
  
*//自认为较好的继承实现方式，成员变量采用对象冒充方式，成员方法采用原型链方式，代码如下：***function** *Person3*(name){  
 **this**.**name** = name;  
}  
*Person3*.**prototype**.say = **function** () {  
 **console**.log(**"My name is "**+**this**.**name**);  
}  
**function** *F2E3*(name,id) {  
 *Person3*.call(**this**,name);  
 **this**.**id** = id;  
}

3、FF浏览器下文本无法撑开容器的高度解决实例与CSS代码

标准浏览器中固定高度值的容器是不会象IE6里那样被撑开的,那我又想固定高度，又想能被撑开需要怎样设置呢？办法就是去掉height设置min-height:200px; 这里为了照顾不认识min-height的IE6 可以这样定义：

4、浏览器内核：

IE Trident ,chrome-safria – webkit , FF gecko, opera presto;

此外，由于IE浏览器在国内的普及率非常高，所以造成了很多网上银行和支付系统只支持IE的Trident内核，其他浏览器访问根本无法进行正常支付和转账等业务。这就造成了经常在网上购物的人离不开Trident内核的IE浏览器。但是Trident内核的速度和兼容性上又是在比不上Chrome浏览器的WebKit内核和Firefox浏览器的Gecko内核。于是双核浏览器应运而生，比如傲游、枫树浏览器（ChromePlus）、搜狗高速浏览器

5、<!DOCTYPE html >

1作用：声明文档的解析类型(document.compatMode)，避免浏览器的怪异模式。

浏览器按照自己的方式解析渲染页面，那么，在不同的浏览器就会显示不同的样式。

如果你的页面添加了<!DOCTYPE html>那么，那么就等同于开启了标准模式，那么浏览器就得老老实实的按照W3C的

标准解析渲染页面，这样一来，你的页面在所有的浏览器里显示的就都是一个样子了。

这就是<!DOCTYPE html>的作用。

2 使用：<!DOCTYPE html>

2.1 使用也很简单，就是在你的html页面的第一行添加"<!DOCTYPE html>"一行代码就可以了

2.2 jsp的话，添加在<%@ page %>的下一行。

3 <!DOCTYPE html>是

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">的简写

六、

1. css

1.1 简要手绘盒子模型，并对边框和边距进行标注

1.2 界面div三平分除了flex布局外，简写其它两种布局代码?—relative -absolute

2. javascrpt

2.1 es6中let const var的区别是什么?

let是修复了var的作用域的一些bug，变的更加好用。let是更好的var。var的作用于是函数作用于，而let是块级别（大括号括起来的内容）  
const声明的变量只可以在声明时赋值，不可随意修改，这是最大的特点。

2.2 对数组[5,6,6,7,8,8]进行去重，es5或者es6方法

**var *aa*** = [5,6,6,7,8,8];

**function** *fun*(arr){

**var** bb = [];

**for**(**var** i = 0;i<arr.**length**;i++){

**if**(bb.indexOf(arr[i]) <0){

bb.push(arr[i]);  
 }  
 }  
 **return** bb;  
}  
**console**.log(*fun*(***aa***))

**console**.log(*new Set(****aa****)*)

2.3 es6的新特性:

ES6是即将到来的新版本JavaScript语言的标准，使得语言更容易理解和更具有可读性，，如箭头函数（=>）、class等等。用一句话来说就是：

ES6给我们提供了许多的新语法和代码特性来提高javascript的体验

不过遗憾的是，现在还没有浏览器能够很好的支持es6语法，点这里查看浏览器支持情况，所以我们在开发中还需要用babel进行转换为CommonJS这种模块化标准的语法。

因为下面我会讲到一些es6新特性的例子，如果想要运行试试效果的话，可以点这里去测试es6的代码。

常用es6特性

然后我下面简单的介绍一些很常用的语法特性，如果想完整的了解ES6，我推荐大家点这里

**定义函数**

我们先来看一个基本的新特性，在javascript中，定义函数需要关键字function，但是在es6中，还有更先进的写法，我们来看：

es6写法：

var human = {

breathe(name) { //不需要function也能定义breathe函数。

console.log(name + ' is breathing...');

}

}

human.breathe('jarson'); //输出 ‘jarson is breathing...’

转成js代码：

var human = {

breathe: function(name) {

console.log(name + 'is breathing...');

}

};

human.breathe('jarson');

很神奇对不对？这样一对比，就可以看出es6的写法让人简单易懂。别着急，下面还有更神奇的。

**创建类**

我们知道，javascript不像java是面向对象编程的语言，而只可以说是基于对象编程的语言。所以在js中，我们通常都是用function和prototype来模拟 类 这个概念。

但是现在有了es6，我们可以像java那样‘明目张胆’的创建一个类了：

class Human {

constructor(name) {

this.name = name;

}

breathe() {

console.log(this.name + " is breathing");

}

}

var man = new Human("jarson");

man.breathe(); //jarson is breathing

上面代码转为js格式：

function Human(name) {

this.name = name;

this.breathe = function() {

console.log(this.name + ' is breathing');

}

}

var man = new Human('jarson');

man.breathe(); //jarson is breathing

所以我们看到，我们可以像java那样语义化的去创建一个类。另外，js中的继承父类，需要用prototype来实现。那么在es6中，又有什么新的方法来实现类的继承呢？继续看：

假如我们要创建一个Man类继承上面的Human类，es6代码：

class Man extends Human {

constructor(name, sex) {

super(name);

this.sex = sex;

}

info(){

console.log(this.name + 'is ' + this.sex);

}

}

var xx = new Man('jarson', 'boy');

xx.breathe(); //jarson is breathing

xx.info(); //arsonis boy

代码很简单，不作赘述，可以使用文章里提到的在线工具去试试效果就能明白了。需要注意的是： super() 是父类的构造函数。

**模块**

在ES6标准中，javascript原生支持module了。将不同功能的代码分别写在不同文件中，各模块只需 导出(export) 公共接口部分，然后在需要使用的地方通过模块的 导入(import) 就可以了。下面继续看例子：

内联导出

ES6模块里的对象可在创建它们的声明中直接导出，一个模块中可无数次使用export。

先看模块文件 app.js ：

export class Human{

constructor(name) {

this.name = name;

}

breathe() {

console.log(this.name + " is breathing");

}

}

export function run(){

console.log('i am runing');

}

function eat() {

console.log('i am eating');

}

例子中的模块导出了两个对象：Human类和run函数，eat函数没有导出，则仍为此模块私有，不能被其他文件使用。

导出一组对象

另外，其实如果需要导出的对象很多的时候，我们可以在最后统一导出一组对象。

更改 app.js 文件：

class Human{

constructor(name) {

this.name = name;

}

breathe() {

console.log(this.name + " is breathing");

}

}

function run(){

console.log('i am runing');

}

function eat() {

console.log('i am eating');

}

export {Human, run};

这样的写法功能和上面一样，而且也很明显，在最后可以清晰的看到导出了哪些对象。

Default导出

导出时使用关键字default，可将对象标注为default对象导出。default关键字在每一个模块中只能使用一次。它既可以用于内联导出，也可以用于一组对象导出声明中。

查看导出default对象的语法：

... //创建类、函数等等

export default { //把Human类和run函数标注为default对象导出。

Human,

run

};

无对象导入

如果模块包含一些逻辑要执行，且不会导出任何对象，此类对象也可以被导入到另一模块中，导入之后只执行逻辑。如：

import './module1.js';

导入默认对象

使用Default导出方式导出对象，该对象在import声明中将直接被分配给某个引用，如下例中的“app”。

import app from './module1.js';

上面例子中，默认 ./module1.js 文件只导出了一个对象；若导出了一组对象，则应该在导入声明中一一列出这些对象，如：

import {Human, run} from './app.js'

let与const

在我看来，在es6新特性中，在定义变量的时候统统使用 let 来代替 var 就好了， const 则很直观，用来定义常量，即无法被更改值的变量。

for (let i=0;i<2;i++) {

console.log(i); //输出: 0,1

}

箭头函数

ES6中新增的箭头操作符 => 简化了函数的书写。操作符左边为输入的参数，而右边则是进行的操作以及返回的值，这样的写法可以为我们减少大量的代码，看下面的实例：

let arr = [6, 8, 10, 20, 15, 9];

arr.forEach((item, i) => console.log(item, i));

let newArr = arr.filter((item) => (item<10));

console.log(newArr); //[6, 8, 9];

上面的 (item, i) 就是参数，后面的 console.log(item, i) 就是回到函数要执行的操作逻辑。

上面代码转为js格式：

var arr = [6, 8, 10, 20, 15, 9];

arr.forEach(function(item, i) {

return console.log(item, i);

});

var newArr = arr.filter(function(item) {

return (item < 10);

});

console.log(newArr);

**字符串模版**

ES6中允许使用反引号 ` 来创建字符串，此种方法创建的字符串里面可以包含由美元符号加花括号包裹的变量${vraible}。看一下实例就会明白了：

//产生一个随机数

let num = Math.random();

//将这个数字输出到console

console.log(`your num is ${num}`);

解构

若一个函数要返回多个值，常规的做法是返回一个对象，将每个值做为这个对象的属性返回。在ES6中，利用解构这一特性，可以直接返回一个数组，然后数组中的值会自动被解析到对应接收该值的变量中。我们来看例子：

function getVal() {

return [1, 2];

}

var [x,y] = getVal(); //函数返回值的解构

console.log('x:'+x+', y:'+y); //输出：x:1, y:2

默认参数

现在可以在定义函数的时候指定参数的默认值了，而不用像以前那样通过逻辑或操作符来达到目的了。

function sayHello(name){

var name=name||'tom'; //传统的指定默认参数的方式

console.log('Hello '+name);

}

//运用ES6的默认参数

function sayHello2(name='tom'){ //如果没有传这个参数，才会有默认值，

console.log(`Hello ${name}`);

}

sayHello();//输出：Hello tom

sayHello('jarson');//输出：Hello jarson

sayHello2();//输出：Hello tom

sayHello2('jarson');//输出：Hello jarson

注意： sayHello2(name='tom') 这里的等号，意思是没有传这个参数，则设置默认值，而不是给参数赋值的意思。

Proxy

Proxy可以监听对象身上发生了什么事情，并在这些事情发生后执行一些相应的操作。一下子让我们对一个对象有了很强的追踪能力，同时在数据绑定方面也很有用处。

//定义被监听的目标对象

let engineer = { name: 'Joe Sixpack', salary: 50 };

//定义处理程序

let interceptor = {

set(receiver, property, value) {

console.log(property, 'is changed to', value);

receiver[property] = value;

}

};

//创建代理以进行侦听

engineer = new Proxy(engineer, interceptor);

//做一些改动来触发代理

engineer.salary = 70;//控制台输出：salary is changed to 70

对于处理程序，是在被监听的对象身上发生了相应事件之后，处理程序里面的方法就会被调用。

结语

总的来说，虽然支持es6的情况到目前还不是很乐观，但es6的新语法特性让前端和后端的差异越来越小了，这是一个新时代的开始，我们必须要了解这些新的前沿知识，才能跟上时代的步伐。

3. html5&css3

3.1 **html5和css3新增了哪些标签和属性**?

Video表示一段视频并提供播放的用户界面

audio表示音频

canvas表示位图区域

source为video和audio提供数据源

track为video和audio指定字母

code 代码段

footer、header、nav

css3选择器、框模型、文本效果、文本效果（常用）text-shadow：设置文字阴影 、word-wrap：强制换行word-break、背景和边框border-radius、2/3D转换、transform：向元素应用2/3D转换、transition：过渡

3.2 用css3写一个开关按钮，按下时置灰变色，并改变按钮文字(如按下之前是on,按下之后是off)

4. http；

post请求和get请求的区别是什么?

1、GET请求，请求的数据会附加在URL之后，以?分割URL和传输数据，多个参数用&连接。URL的编码格式采用的是ASCII编码，而不是uniclde，即是说所有的非ASCII字符都要编码之后再传输。

2、POST请求：POST请求会把请求的数据放置在HTTP请求包的包体中。上面的item=bandsaw就是实际的传输数据。

因此，GET请求的数据会暴露在地址栏中，而POST请求则不会

5. 前端框架和模板

5.1 使用过的前端框架有哪些?juery

5.5 如何理解、前端MVC架构，简要画出流程图

**bind方法的原理**

bind()最简单的用法就是创建一个函数，是这个函数不论怎么调用都有同样的this值。常见的错误就是将方法从对象中拿出来，然后调用，并且希望this指向原来的对象，。如果不做特殊处理，一般会丢失原来的对象。使用bind()方法能够解决这个问题。

bind()方法会创建一个新函数，称为绑定函数，当调用这个绑定函数时，绑定函数会以创建它时传入 bind()方法的第一个参数作为 this，传入 bind() 方法的第二个以及以后的参数加上绑定函数运行时本身的参数按照顺序作为原函数的参数来调用原函数。

var foo = {

    bar : 1,

    eventBind: function(){

        var \_this = this;

        $('.someClass').on('click',function(event) {

            /\* Act on the event \*/

            console.log(\_this.bar);     //1

        });

    }

}

var foo = {

    bar : 1,

    eventBind: function(){

        $('.someClass').on('click',function(event) {

            /\* Act on the event \*/

            console.log(this.bar);      //1

        }.bind(this));

    }

在Javascript中，多次 bind() 是无效的

**实现forEach；**

首先，虽然叫Foreach语法但关键字还是用for哦，这个语法只是对平时用开的for语法的一个简化而已。

这个语法平时最多还是用来遍历数组，这时候每一个循环得到的是数组的索引(一个整形数字),然后通过数组名[整形索引]获得数组中的对象。

但是这个语法还可以用来遍历对象，拿到的是对象的属性名称(一个字符串).然后通过对象名[属性名称]就可以拿到对象。

所以理解该语法的关键是理解每次循环得到的到底是什么。

其实，这一功能的实现得益于javascript的数组索引可以是字符串，如果不能（想想java）就没有这出戏唱了。

<!DOCTYPE html>  
<**html** lang=**"en"**>  
<**heap**>  
 <**script** type=**"text/javascript"**>  
 **var *mycolors*** = **new** Array(**'blue'**,**'red'**,**'yellow'**);  
 **function** *f1*(){  
 **var** content=**""**;  
 **for**(**var** key **in *mycolors***){  
 **console**.log(key)

content += key+**": "**+***mycolors***[key]+**"<br/>"**;  
 }  
 **document**.getElementById(**"content"**).**innerHTML** = content;  
 }  
 **function** *User*(){}  
 **function** *f2*(){  
 **var** u1=**new** *User*();  
 u1.**uname**=**"张三"**;  
 u1.**age**=**"18"**;  
 **var** content=**""**;  
 **for**(**var** key **in** u1){  
 **console**.log(key)  
 content += key+**": "**+u1[key]+**"<br/>"**;  
 }  
 **document**.getElementById(**"content"**).**innerHTML** = content;  
 }  
 </**script**>  
</**heap**>  
<**body**>  
<**input** type=**"button"** id=**"c1"** name=**"c1"** onclick=**"***f1*();**"** value=**"click one"**/>  
<**input** type=**"button"** id=**"c2"** name=**"c2"** onclick=**"***f2*();**"** value=**"click two"**/>  
<**div** id=**"content"**></**div**>  
</**body**>  
</**html**>

**移动端/PC端的区别**

1. PC端在开发过程中考虑的是浏览器的兼容性，移动端考虑的是手机兼容性问题，做移动端开发，更多的是手机分辨率的自适应和不同手机操作系统的略微差异化；
2. 在部分时间的处理上，移动端自然偏向于触屏的，意外包括移动端弹出的手机键盘应该如何处理，这样的问题在PC上肯定不会遇到，但在移动端，如果没有经验，处理起来相当麻烦；
3. 在布局上，移动端开发室要做到页面布局自适应的，而PC端页面布局的比例会相对固定；
4. 在动画处理上，PC端开考虑的是IE的兼容性，通常JS做动画会通用性好一些，但相比CSS3，缺牺牲了较大的性能，而在手机端，如果要做一些动画，特效，第一选择肯定是CSS3，既简单又高效。

Bootstrap

响应式设计

**Webpack**

webpack介绍：在开发过程中，我们可以利用模块化开发（CMD模式），将一个大型的功能按不同的小功能，拆分成不同的模块（文件），这样在以后的维护中，我们可以针对不同的功能进行维护，不至于在一个上万行的文件中，找来找去，而且当一个文件很大的时候，电脑不好的在查看，修改的时候会很卡，也不利于工作效率，因此产生了模块化开发这个概念；而webpack正是将模块化进行打包发布处理的一个工具

常见模块化开发规范：CMD（seaJS）  AMD（requireJS...）

npm install webpack -g

webpack打包：

**app/runoob1.js 文件**

内容：document.write("It works.");

打包：webpack runoob1.js bundle.js 编译 runoob1.js 文件并生成bundle.js 文件

**app/index.html 文件**

<html> <head> <meta charset="utf-8"> </head> <body> <script type="text/javascript" src="bundle.js" charset="utf-8"></script> </body> </html>

**创建第二个 JS 文件**

接下来我们创建另外一个 js 文件 runoob2.js，代码如下所示：

**app/runoob2.js 文件**

module.exports = "It works from runoob2.js.";

更新 runoob1.js 文件，代码如下：

**app/runoob1.js 文件**

document.write(require("./runoob2.js"));

**webpack 根据模块的依赖关系进行静态分析，这些文件(模块)会被包含到 bundle.js 文件中。Webpack 会给每个模块分配一个唯一的 id 并通过这个 id 索引和访问模块。 在页面启动时，会先执行 runoob1.js 中的代码，其它模块会在运行 require 的时候再执行。**

**LOADER**

Webpack 本身只能处理 JavaScript 模块，如果要处理其他类型的文件，就需要使用 loader 进行转换。

所以如果我们需要在应用中添加 css 文件，就需要使用到 css-loader 和 style-loader，他们做两件不同的事情，css-loader 会遍历 CSS 文件，然后找到 url() 表达式然后处理他们，style-loader 会把原来的 CSS 代码插入页面中的一个 style 标签中。

接下来我们使用以下命令来安装 css-loader 和 style-loader(全局安装需要参数 -g)。

cnpm install css-loader style-loader

执行以上命令后，会再当前目录生成 node\_modules 目录，它就是 css-loader 和 style-loader 的安装目录。

接下来创建一个 style.css 文件，代码如下：

**app/style.css 文件**

body { background: yellow; }

修改 runoob1.js 文件，代码如下：

**app/runoob1.js 文件**

require("!style-loader!css-loader!./style.css");

document.write(require("./runoob2.js"));

接下来我们使用 webpack 命令来打包：

webpack runoob1.js bundle.js

**配置文件**

我们可以将一些编译选项放在配置文件中，以便于统一管理：

创建 webpack.config.js 文件，代码如下所示：

**app/webpack.config.js 文件**

module.exports = {

entry: "./runoob1.js",

output: { path: \_\_dirname, filename: "bundle.js" },

module: {

loaders: [ { test: /\.css$/, loader: "style-loader!css-loader" }]

}

};

接下来我们只需要执行 webpack 命令即可生成 bundle.js 文件：

**插件**

插件在 webpack 的配置信息 plugins 选项中指定，用于完成一些 loader 不能完成的工。

webpack 自带一些插件，你可以通过 cnpm 安装一些插件。

使用内置插件需要通过以下命令来安装：

cnpm install webpack --save-dev

比如我们可以安装内置的 BannerPlugin 插件，用于在文件头部输出一些注释信息。

修改 webpack.config.js，代码如下：

**app/webpack.config.js 文件**

var webpack=require('webpack');

module.exports = {

entry: "./runoob1.js",

output: { path: \_\_dirname, filename: "bundle.js" },

module: { loaders: [ { test: /\.css$/, loader: "style-loader!css-loader" } ] },

plugins:[ new webpack.BannerPlugin('菜鸟教程 webpack 实例') ]

};

然后运行: webpack

打开 bundle.js，可以看到文件头部出现了我们指定的注释信息。

**开发环境**

当项目逐渐变大，webpack 的编译时间会变长，可以通过参数让编译的输出内容带有进度和颜色。

webpack --progress --colors

如果不想每次修改模块后都重新编译，那么可以启动监听模式。开启监听模式后，没有变化的模块会在编译后缓存到内存中，而不会每次都被重新编译，所以监听模式的整体速度是很快的。

webpack --progress --colors --watch

当然，我们可以使用 webpack-dev-server 开发服务，这样我们就能通过 localhost:8080 启动一个 express 静态资源 web 服务器，并且会以监听模式自动运行 webpack，在浏览器打开 http://localhost:8080/ 或 http://localhost:8080/webpack-dev-server/ 可以浏览项目中的页面和编译后的资源输出，并且通过一个 socket.io 服务实时监听它们的变化并自动刷新页面。

# 安装 cnpm install webpack-dev-server -g # 运行 webpack-dev-server --progress --colors