1.什么是前端？

前端对于网站来说，通常是指，网站的前台部分包括网站的表现层和结构层。因此前端技术一般分为前端设计和[前端开发](http://baike.baidu.com/view/1319256.htm)，前端设计一般可以理解为网站的视觉设计，前端开发则是网站的前台代码实现，包括基本的HTML和CSS以及JavaScript/ajax，现在最新的高级版本HTML5、CSS3，以及SVG等。

2.js全称是什么？

JavaScript

3.java与js是同一种语言么，有什么区别？

Java，全称应该是 Java Applet，是嵌在网页中，而又有自己独立的运行窗口的小程序。  
Java Applet 是预先编译好的，一个Applet 文件（.class）用 Notepad 打开阅读，根本不能理解。Java Applet 的功能很强大，可以访问 http、ftp等协议，甚至可以在电脑上种病毒（已有先例了）。  
  
相比之下，JavaScript 的能力就比较小了。JavaScript 是一种“脚本”（“Script”），它直接把代码写到 HTML文档中，浏览器读取它们的时候才进行编译、执行，所以能查看 HTML 源文件就能查看JavaScript 源代码。JavaScript 没有独立的运行窗口，浏览器当前窗口就是它的运行窗口。  
它们的相同点，我想只有同是以 Java 作编程语言一点了。

4.什么是瀑布流？什么是响应式？

瀑布流，又称瀑布流式布局。是比较流行的一种网站页面布局，视觉表现为参差不齐的多栏布局，随着页面滚动条向下滚动，这种布局还会不断加载数据块并附加至当前尾部。最早采用此布局的网站是Pinterest，逐渐在国内流行开来。国内大多数清新站基本为这类风格，像花瓣网、蘑菇街、美丽说等。

响应式布局是Ethan Marcotte在2010年5月份提出的一个概念，简而言之，就是一个网站能够兼容多个终端——而不是为每个终端做一个特定的版本。

5.处理浏览器的兼容性有哪些方法？

所谓的浏览器兼容性问题，是指因为不同的浏览器对同一段代码有不同的解析，造成页面显示效果不统一的情况。在大多数情况下，我们的需求是，无论用户用什么浏览器来查看我们的网站或者登陆我们的系统，都应该是统一的显示效果。所以浏览器的兼容性问题是前端开发人员经常会碰到和必须要解决的问题。

在学习浏览器兼容性之前，我想把前端开发人员划分为两类：

第一类是精确按照设计图开发的前端开发人员，可以说是精确到1px的，他们很容易就会发现设计图的不足，并且在很少的情况下会碰到浏览器的兼容性问题，而这些问题往往都死浏览器的bug，并且他们制作的页面后期易维护，代码重用问题少，可以说是比较牢固放心的代码。

第二类是基本按照设计图来开发的前端开发人员，很多细枝末节差距很大，不如间距，行高，图片位置等等经常会差几px。某种效果的实现也是反复调试得到，具体为什么出现这种效果还模模糊糊，整体布局十分脆弱。稍有改动就乱七八糟。代码为什么这么写还不知所以然。这类开发人员往往经常为兼容性问题所困。修改好了这个浏览器又乱了另一个浏览器。改来改去也毫无头绪。其实他们碰到的兼容性问题大部分不应该归咎于浏览器，而是他们的技术本身了。

文章主要针对的是第一类，严谨型的开发人员，因此这里主要从浏览器解析差异的角度来分析兼容性问题。（相关文章推荐：[主流浏览器CSS 3和HTML 5兼容清单](http://developer.51cto.com/art/201004/192415.htm)）

**浏览器兼容问题一：不同浏览器的标签默认的外补丁和内补丁不同**

问题症状：随便写几个标签，不加样式控制的情况下，各自的margin 和padding差异较大。

碰到频率:100%

解决方案：CSS里    \*{margin:0;padding:0;}

备注：这个是最常见的也是最易解决的一个浏览器兼容性问题，几乎所有的CSS文件开头都会用通配符\*来设置各个标签的内外补丁是0。

**浏览器兼容问题二：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**

问题症状:常见症状是IE6中后面的一块被顶到下一行

碰到频率：90%（稍微复杂点的页面都会碰到，float布局最常见的浏览器兼容问题）

解决方案：在float的标签样式控制中加入 display:inline;将其转化为行内属性

备注：我们最常用的就是div+CSS布局了，而div就是一个典型的块属性标签，横向布局的时候我们通常都是用div float实现的，横向的间距设置如果用margin实现，这就是一个必然会碰到的兼容性问题。

**浏览器兼容问题三：设置较小高度标签（一般小于10px），在IE6，IE7，遨游中高度超出自己设置高度**

问题症状：IE6、7和遨游里这个标签的高度不受控制，超出自己设置的高度

碰到频率：60%

解决方案：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。

备注：这种情况一般出现在我们设置小圆角背景的标签里。出现这个问题的原因是IE8之前的浏览器都会给标签一个最小默认的行高的高度。即使你的标签是空的，这个标签的高度还是会达到默认的行高。

**浏览器兼容问题四：行内属性标签，设置display:block后采用float布局，又有横行的margin的情况，IE6间距bug**

问题症状：IE6里的间距比超过设置的间距

碰到几率：20%

解决方案：在display:block;后面加入display:inline;display:table;

备注：行内属性标签，为了设置宽高，我们需要设置display:block;(除了input标签比较特殊)。在用float布局并有横向的margin后，在IE6下，他就具有了块属性float后的横向margin的bug。不过因为它本身就是行内属性标签，所以我们再加上display:inline的话，它的高宽就不可设了。这时候我们还需要在display:inline后面加入display:talbe。

**浏览器兼容问题五：图片默认有间距**

问题症状：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。

碰到几率：20%

解决方案：使用float属性为img布局

备注：因为img标签是行内属性标签，所以只要不超出容器宽度，img标签都会排在一行里，但是部分浏览器的img标签之间会有个间距。去掉这个间距使用float是正道。（我的一个学生使用负margin，虽然能解决，但负margin本身就是容易引起浏览器兼容问题的用法，所以我禁止他们使用）

**浏览器兼容问题六：标签最低高度设置min-height不兼容**

问题症状：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

碰到几率：5%

解决方案：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;}

备注：在B/S系统前端开时，有很多情况下我们又这种需求。当内容小于一个值（如300px）时。容器的高度为300px；当内容高度大于这个值时，容器高度被撑高，而不是出现滚动条。这时候我们就会面临这个兼容性问题。

**浏览器兼容问题七：透明度的兼容CSS设置**

做兼容页面的方法是：每写一小段代码（布局中的一行或者一块）我们都要在不同的浏览器中看是否兼容，当然熟练到一定的程度就没这么麻烦了。建议经常会碰到兼容性问题的新手使用。很多兼容性问题都是因为浏览器对标签的默认属性解析不同造成的，只要我们稍加设置都能轻松地解决这些兼容问题。如果我们熟悉标签的默认属性的话，就能很好的理解为什么会出现兼容问题以及怎么去解决这些兼容问题。

1. /\* CSS hack\*/

我很少使用hacker的，可能是个人习惯吧，我不喜欢写的代码IE不兼容，然后用hack来解决。不过hacker还是非常好用的。使用hacker我可以把浏览器分为3类：IE6 ；IE7和遨游；其他（IE8 chrome ff safari opera等）

◆IE6认识的hacker 是下划线\_ 和星号 \*

◆IE7 遨游认识的hacker是星号 \*

比如这样一个CSS设置：

1. height:300px;\*height:200px;\_height:100px;

IE6浏览器在读到height:300px的时候会认为高时300px；继续往下读，他也认识\*heihgt， 所以当IE6读到\*height:200px的时候会覆盖掉前一条的相冲突设置，认为高度是200px。继续往下读，IE6还认识\_height,所以他又会覆盖掉200px高的设置，把高度设置为100px；

IE7和遨游也是一样的从高度300px的设置往下读。当它们读到\*height200px的时候就停下了，因为它们不认识\_height。所以它们会把高度解析为200px，剩下的浏览器只认识第一个height:300px;所以他们会把高度解析为300px。因为优先级相同且想冲突的属性设置后一个会覆盖掉前一个，所以书写的次序是很重要的。

一旦为页面设置了恰当的 DTD，大多数浏览器都会按照上面的图示来呈现内容。然而 IE 5 和 6 的呈现却是不正确的。根据 W3C 的规范，元素内容占据的空间是由 width 属性设置的，而内容周围的 padding 和 border 值是另外计算的。不幸的是，IE5.X 和 6 在怪异模式中使用自己的非标准模型。这些浏览器的 width 属性不是内容的宽度，而是内容、内边距和边框的宽度的总和。

虽然有方法解决这个问题。但是目前最好的解决方案是回避这个问题。也就是，不要给元素添加具有指定宽度的内边距，而是尝试将内边距或外边距添加到元素的父元素和子元素。

**html5浏览器支持**

最新版本的 Safari、Chrome、Firefox 以及 Opera 支持某些 HTML5 特性。Internet Explorer 9 将支持某些 HTML5 特性。

什么是事件冒泡？

在页面上可以有多个事件，也可以有多个元素响应同一件事情。假设网页上有两个元素，其中一个元素嵌套在另一个元素里，并且都被绑定了click事件，同事<body>元素上也绑定了click事件。当点击内部<span>元素，即触发<span>元素的click事件时，会输出3条记录。

在点击<span>元素的同时，也点击了包含<span>元素的元素<div>和包含<div>元素的元素<body>，并且每个元素都会按照特定的顺序响应click事件。

元素的click事件会按照以下顺序“冒泡”。

（1）<span>

（2）<div>

（3）<body>

之所以称之为冒泡，是因为事件会按照DOM的层次结构像水泡一样不多向上直至顶端。

什么是开包闭包？

1、什么是闭包

闭包，官方对闭包的解释是：一个拥有许多变量和绑定了这些变量的环境的表达式（通常是一个函数），因而这些变量也是该表达式的一部分。闭包的特点：

　　1. 作为一个函数变量的一个引用，当函数返回时，其处于激活状态。

　　2. 一个闭包就是当一个函数返回时，一个没有释放资源的栈区。

　　简单的说，[Java](http://www.2cto.com/kf/ware/Java/)script允许使用内部函数---即函数定义和函数表达式位于另一个函数的函数体内。而且，这些内部函数可以访问它们所在的外部函数中声明的所有局部变量、参数和声明的其他内部函数。当其中一个这样的内部函数在包含它们的外部函数之外被调用时，就会形成闭包。

2、闭包的几种写法和用法

首先要明白，在JS中一切都是对象，函数是对象的一种。下面先来看一下闭包的5种写法，简单理解一下什么是闭包。后面会具体解释。

复制代码

//第1种写法

function Circle(r) {

      this.r = r;

}

Circle.PI = 3.14159;

Circle.prototype.area = function() {

  return Circle.PI \* this.r \* this.r;

}

var c = new Circle(1.0);

alert(c.area());

复制代码

这种写法没什么特别的，只是给函数添加一些属性。

复制代码

//第2种写法

var Circle = function() {

   var obj = new Object();

   obj.PI = 3.14159;

   obj.area = function( r ) {

       return this.PI \* r \* r;

   }

   return obj;

}

var c = new Circle();

alert( c.area( 1.0 ) );

复制代码

这种写法是声明一个变量，将一个函数当作值赋给变量。

复制代码

//第3种写法

var Circle = new Object();

Circle.PI = 3.14159;

Circle.Area = function( r ) {

       return this.PI \* r \* r;

}

alert( Circle.Area( 1.0 ) );

复制代码

这种方法最好理解，就是new 一个对象，然后给对象添加属性和方法。

复制代码

//第4种写法

var Circle={

   "PI":3.14159,

 "area":function(r){

          return this.PI \* r \* r;

        }

};

alert( Circle.area(1.0) );

复制代码

这种方法使用较多，也最为方便。var obj = {}就是声明一个空的对象。

//第5种写法

var Circle = new Function("this.PI = 3.14159;this.area = function( r ) {return r\*r\*this.PI;}");

alert( (new Circle()).area(1.0) );

说实话，这种写法我是没用过，大家可以参考一下。

总的来说，上面几种方法，第2中和第4中较为常见，大家可以根据习惯选择。

上面代码中出现了JS中常用的Prototype，那么Prototype有什么用呢？下面我们来看一下：

复制代码

    var dom = function(){

    };

    dom.Show = function(){

        alert("Show Message");

    };

    dom.prototype.Display = function(){

        alert("Property Message");

    };

    dom.Display(); //error

    dom.Show();

    var d = new dom();

    d.Display();

    d.Show(); //error

复制代码

我们首先声明一个变量，将一个函数赋给他，因为在Javascript中每个函数都有一个Portotype属性，而对象没有。添加两个方法，分别直接添加和添加打破Prototype上面，来看下调用情况。分析结果如下：

 　　1、不使用prototype属性定义的对象方法，是静态方法，只能直接用类名进行调用！另外，此静态方法中无法使用this变量来调用对象其他的属性！

　　 2、使用prototype属性定义的对象方法，是非静态方法，只有在实例化后才能使用！其方法内部可以this来引用对象自身中的其他属性！

下面我们再来看一段代码：

复制代码

var dom = function(){

        var Name = "Default";

        this.Sex = "Boy";

        this.success = function(){

            alert("Success");

        };

    };

    alert(dom.Name);

    alert(dom.Sex);

复制代码

大家先看看，会显示什么呢？ 答案是两个都显示Undefined,为什么呢？这是由于在Javascript中每个function都会形成一个作用域，而这些变量声明在函数中，所以就处于这个函数的作用域中，外部是无法访问的。要想访问变量，就必须new一个实例出来。

复制代码

var html = {

        Name:'Object',

        Success:function(){

            this.Say = function(){

                    alert("Hello,world");

            };

            alert("Obj Success");

        }

    };

复制代码

再来看看这种写法，其实这是Javascript的一个"语法糖"，这种写法相当于：

复制代码

    var html = new Object();

    html.Name = 'Object';

    html.Success = function(){

            this.Say = function(){

                    alert("Hello,world");

            };

            alert("Obj Success");

复制代码

变量html是一个对象，不是函数，所以没有Prototype属性，其方法也都是公有方法，html不能被实例化。否则会出现如下错误：

但是他可以作为值赋给其它变量，如var o = html; 我们可以这样使用它：

    alert(html.Name);

    html.Success();

说到这里，完了吗？细心的人会问，怎么访问Success方法中的Say方法呢？是html.Success.Say()吗？

当然不是，上面刚说过由于作用域的限制，是访问不到的。所以要用下面的方法访问：

复制代码

var s = new html.Success();

s.Say();

//还可以写到外面

html.Success.prototype.Show = function(){

    alert("HaHa");

};

var s = new html.Success();

s.Show();

复制代码

关于Javascript作用域的问题，不是一两句能说清楚的，有兴趣的大家可以网上找些资料看看。

二、Javascript闭包的用途

事实上，通过使用闭包，我们可以做很多事情。比如模拟面向对象的代码风格；更优雅，更简洁的表达出代码；在某些方面提升代码的执行效率。

1、匿名自执行函数

　　我们知道所有的变量，如果不加上var关键字，则默认的会添加到全局对象的属性上去，这样的临时变量加入全局对象有很多坏处，

比如：别的函数可能误用这些变量；造成全局对象过于庞大，影响访问速度(因为变量的取值是需要从原型链上遍历的)。

除了每次使用变量都是用var关键字外，我们在实际情况下经常遇到这样一种情况，即有的函数只需要执行一次，其内部变量无需维护，

比如UI的初始化，那么我们可以使用闭包：

复制代码

var data= {

    table : [],

    tree : {}

};

(function(dm){

    for(var i = 0; i < dm.table.rows; i++){

       var row = dm.table.rows[i];

       for(var j = 0; j < row.cells; i++){

           drawCell(i, j);

       }

    }

})(data);

复制代码

我们创建了一个匿名的函数，并立即执行它，由于外部无法引用它内部的变量，因此在函数执行完后会立刻释放资源，关键是不污染全局对象。

2、结果缓存

我们开发中会碰到很多情况，设想我们有一个处理过程很耗时的函数对象，每次调用都会花费很长时间，

那么我们就需要将计算出来的值存储起来，当调用这个函数的时候，首先在缓存中查找，如果找不到，则进行计算，然后更新缓存并返回值，如果找到了，直接返回查找到的值即可。闭包正是可以做到这一点，因为它不会释放外部的引用，从而函数内部的值可以得以保留。

复制代码

var CachedSearchBox = (function(){

    var cache = {},

       count = [];

    return {

       attachSearchBox : function(dsid){

           if(dsid in cache){//如果结果在缓存中

              return cache[dsid];//直接返回缓存中的对象

           }

           var fsb = new uikit.webctrl.SearchBox(dsid);//新建

           cache[dsid] = fsb;//更新缓存

           if(count.length > 100){//保正缓存的大小<=100

              delete cache[count.shift()];

           }

           return fsb;

       },

       clearSearchBox : function(dsid){

           if(dsid in cache){

              cache[dsid].clearSelection();

           }

       }

    };

})();

CachedSearchBox.attachSearchBox("input");

复制代码

这样我们在第二次调用的时候，就会从缓存中读取到该对象。

3、封装

复制代码

var person = function(){

    //变量作用域为函数内部，外部无法访问

    var name = "default";

    return {

       getName : function(){

           return name;

       },

       setName : function(newName){

           name = newName;

       }

    }

}();

print(person.name);//直接访问，结果为undefined

print(person.getName());

person.setName("abruzzi");

print(person.getName());

得到结果如下：

undefined

default

abruzzi

复制代码

4、实现类和继承

复制代码

function Person(){

    var name = "default";

    return {

       getName : function(){

           return name;

       },

       setName : function(newName){

           name = newName;

       }

    }

    };

    var p = new Person();

    p.setName("Tom");

    alert(p.getName());

    var Jack = function(){};

    //继承自Person

    Jack.prototype = new Person();

    //添加私有方法

    Jack.prototype.Say = function(){

        alert("Hello,my name is Jack");

    };

    var j = new Jack();

    j.setName("Jack");

    j.Say();

    alert(j.getName());

复制代码

我们定义了Person，它就像一个类，我们new一个Person对象，访问它的方法。

下面我们定义了Jack，继承Person，并添加自己的方法。