**1.DOM事件**

HTML事件的例子：

当用户点击鼠标时onmousedown、onmouseup 以及 onclick 事件

当网页已加载时

当图片已加载时

当鼠标移动到元素上时onmouseover 和 onmouseout 事件

当输入字段被改变时：onchange

当 HTML 表单被提交时

当用户触发按键时

当用户进入或离开页面时，会触发 onload 和 onunload 事件。

**2.事件监听**

element.addEventListener(<event-name>,<callback>,<use-capture>)

其中第三个参数决定回调函数是否在捕获阶段被触发。

**3.移除事件监听**

element.removeEventListener(<event-name>, <callback>, <use-capture>);

**4.维护回调上下文**

看一个简单的例子：

var element = document.getElementById('element');

var user = {

firstname: 'Wilson',

greeting: function(){

alert('My name is ' + this.firstname);

}

};

// Attach user.greeting as a callback

element.addEventListener('click', user.greeting);

// alert => 'My name is undefined'

这里我们期待输出My name is Wilson，但实际输出的却是My name is undefined。因为我们传递greeting函数的时候，传递的是一个函数的引用，user相应的上下文并没有传递过去。运行的时候，这个回调函数实际上是在element的上下文中执行了，this指向的是element而不是user。

两种方式避免这种问题：

第一种方法，我们可以在一个匿名函数内部调用user.greeting()方法，从而获得正确的函数执行上下文(user)。

element.addEventListener('click', function() {

user.greeting();

// alert => 'My name is Wilson'

});

上一种方式并不是非常好，因为我们不能获得回调函数的句柄以便后面通过.removeEventListener()移除事件监听。另外，这种方式也比较丑陋。

另一种方法是使用.bind()方法(做为ECMAScript 5的标准内建在所有的函数对象中）来生成一个新的函数（被绑定过的函数），这个函数会在指定的上下文中被执行。然后我们将这个被绑定过的函数作为参数传给.addEventListener()的回调函数。

// Overwrite the original function with

// one bound to the context of 'user'

user.greeting = user.greeting.bind(user);

// Attach the bound user.greeting as a callback

button.addEventListener('click', user.greeting);

与此同时，我们获得了回调函数的句柄，从而可以随时从元素上移除相应的事件监听。

button.removeEventListener('click', user.greeting);

**5.Event对象**

Event对象在event第一次触发的时候被创建出来，并且一直伴随着事件在DOM结构中流转的整个生命周期。event对象会被作为第一个参数传递给事件监听的回调函数。我们可以通过这个event对象来获取到大量当前事件相关的信息：

type (String) — 事件的名称

target (node) — 事件起源的DOM节点

currentTarget?(node) — 当前回调函数被触发的DOM节点（后面会做比较详细的介绍）

bubbles (boolean) — 指明这个事件是否是一个冒泡事件（接下来会做解释）

preventDefault(function) — 这个方法将阻止浏览器中用户代理对当前事件的相关默认行为被触发。比如阻止<a>元素的click事件加载一个新的页面

stopPropagation (function) — 这个方法将阻止当前事件链上后面的元素的回调函数被触发，当前节点上针对此事件的其他回调函数依然会被触发。（我们稍后会详细介绍。）

stopImmediatePropagation (function) — 这个方法将阻止当前事件链上所有的回调函数被触发，也包括当前节点上针对此事件已绑定的其他回调函数。

cancelable (boolean) — 这个变量指明这个事件的默认行为是否可以通过调用event.preventDefault来阻止。也就是说，只有cancelable为true的时候，调用event.preventDefault才能生效。

defaultPrevented (boolean) — 这个状态变量表明当前事件对象的preventDefault方法是否被调用过

isTrusted (boolean) — 如果一个事件是由设备本身（如浏览器）触发的，而不是通过JavaScript模拟合成的，那个这个事件被称为可信任的(trusted)

eventPhase (number) — 这个数字变量表示当前这个事件所处的阶段(phase):none(0), capture(1),target(2),bubbling(3)。我们会在下一个部分介绍事件的各个阶段

timestamp (number) — 事件发生的时间

此外事件对象还可能拥有很多其他的属性，但是他们都是针对特定的event的。比如，鼠标事件包含clientX和clientY属性来表明鼠标在当前视窗的位置。

**6.事件阶段**

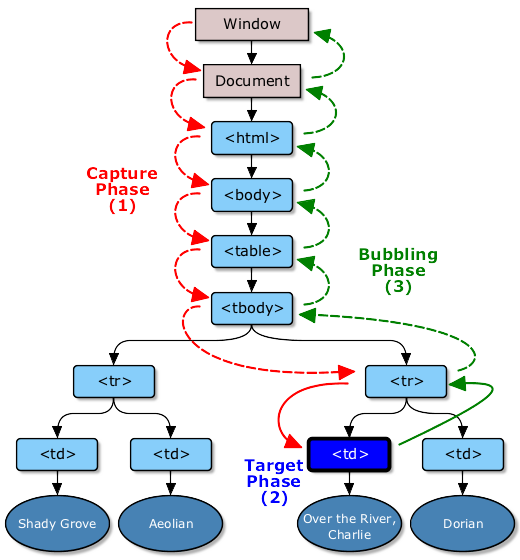
当一个DOM事件被触发的时候，它并不只是在它的起源对象上触发一次，而是会经历三个不同的阶段。简而言之：事件一开始从文档的根节点流向目标对象（捕获阶段），然后在目标对向上被触发（目标阶段），之后再回溯到文档的根节点（冒泡阶段）。

**事件捕获阶段（Capture Phase）**

事件的第一个阶段是捕获阶段。事件从文档的根节点出发，随着DOM树的结构向事件的目标节点流去。途中经过各个层次的DOM节点，并在各节点上触发捕获事件，直到到达事件的目标节点。捕获阶段的主要任务是建立传播路径，在冒泡阶段，事件会通过这个路径回溯到文档跟节点。

正如文章一开始的地方提到，我们可以通过将addEventListener的第三个参数设置成true来为事件的捕获阶段添加监听回调函数。在实际应用中，我们并没有太多使用捕获阶段监听的用例，但是通过在捕获阶段对事件的处理，我们可以阻止类似clicks事件在某个特定元素上被触发。

如果你对这种用法不是很了解的话，最好还是将useCapture设置为false或者undefined，从而在冒泡阶段对事件进行监听。



**目标阶段（Target Phase）**

当事件到达目标节点的，事件就进入了目标阶段。事件在目标节点上被触发，然后会逆向回流，直到传播至最外层的文档节点。

对于多层嵌套的节点，鼠标和指针事件经常会被定位到最里层的元素上。假设，你在一个<div>元素上设置了click事件的监听函数，而用户点击在了这个<div>元素内部的<p>元素上，那么<p>元素就是这个事件的目标元素。事件冒泡让我们可以在这个<div>（或者更上层的）元素上监听click事件，并且事件传播过程中触发回调函数。

**冒泡阶段（Bubble Phase）**

事件在目标元素上触发后，并不在这个元素上终止。它会随着DOM树一层层向上冒泡，直到到达最外层的根节点。也就是说，同一个事件会依次在目标节点的父节点，父节点的父节点...直到最外层的节点上被触发。

将DOM结构想象成一个洋葱，事件目标是这个洋葱的中心。在捕获阶段，事件从最外层钻入洋葱，穿过途径的每一层。在到达中心后，事件被触发（目标阶段）。然后事件开始回溯，再次经过每一层返回（冒泡阶段）。当到达洋葱表面的时候，这次旅程就结束了。

冒泡过程非常有用。它将我们从对特定元素的事件监听中释放出来，相反，我们可以监听DOM树上更上层的元素，等待事件冒泡的到达。如果没有事件冒泡，在某些情况下，我们需要监听很多不同的元素来确保捕获到想要的事件。

**7.停止传播（stop propagation）**

可以通过调用事件对象的stopPropagation方法在任何阶段（捕获或冒泡）中断事件的传播。此后，事件不会在后面传播过程中的经过的节点上调用任何的监听函数。

<div class='overlay'>Click outside to close.</div>

var overlay = document.querySelector('.overlay');

overlay.addEventListener('click', function(event) {

event.stopPropagation();

});

// Remove the overlay when a 'click'

// is heard on the document (<html>) element

document.addEventListener('click', function(event) {

overlay.parentNode.removeChild(overlay);

});

上面的代码点击document其他地方overlay会被隐藏，点击overlay本身时不会隐藏，因为event.stopPropagation();阻止了事件冒泡向上传播，document的事件不会执行。

<div class='overlay'>Click outside to close.</div>

var overlay = document.querySelector('.overlay');

overlay.addEventListener('click', function(event) {

event.stopPropagation();

});

// Remove the overlay when a 'click'

// is heard on the document (<html>) element

overlay.addEventListener('click', function(event) {

overlay.parentNode.removeChild(overlay);

});

event.stopPropagation()不会阻止当前节点上其他监听函数被调用。例如上面的代码，点击overlay时它会被移除。如果希望阻止当前 节点上其他回调函数被调用的话，可以使用更加激进的event.stopImmediatePropagation()

**8.阻止浏览器默认行为**

调用event.stopPropagation()只会阻止传播链中后续的回调函数被触发。它不会阻止浏览器的自身的行为。

function stopDefault(e){

//防止浏览器默认行为(W3C)

if(e && e.preventDefault){

e.preventDefault();

}

//IE中组织浏览器行为

else{

window.event.returnValue=fale;

}

return false;

}

**9.代理事件监听**

如果有一个列表<ul>包含了100个子元素<li>，它们都需要对click事件做出相似的响应，那么我们可能需要查询这100个子元素，并分别为他们添加上事件监听器。这样的话，我们就会产生100个独立的事件监听器。如果有一个新的元素被添加进去，我们也需要为它添加同样的监听器。这种方式不但代价比较大，维护起来也比较麻烦。

代理事件监听可以让我们更简单的处理这种情况。我们不去监听所有的子元素的click事件，相反，我们监听他们的父元素<ul>。当一个<li>元素被点击的时候，这个事件会向上冒泡至<ul>，触发回调函数。我们可以通过检查事件的event.target属性来判断具体是哪一个<li>被点击了：

var list = document.querySelector('ul');

list.addEventListener('click', function(event) {

var target = event.target;

while (target.tagName !== 'LI') {

target = target.parentNode;

if (target === list) return;

}

// Do stuff here

});

不建议在项目中使用这个粗糙的实现。可以使用封装好的方法，如在jQuery中可以在调用.on()方法的时候，将一个选择器作为第二个参数的方式来轻松的实现事件代理。

// Not using event delegation

$('li').on('click', function(){});

// Using event delegation

$('ul').on('click', 'li', function(){});

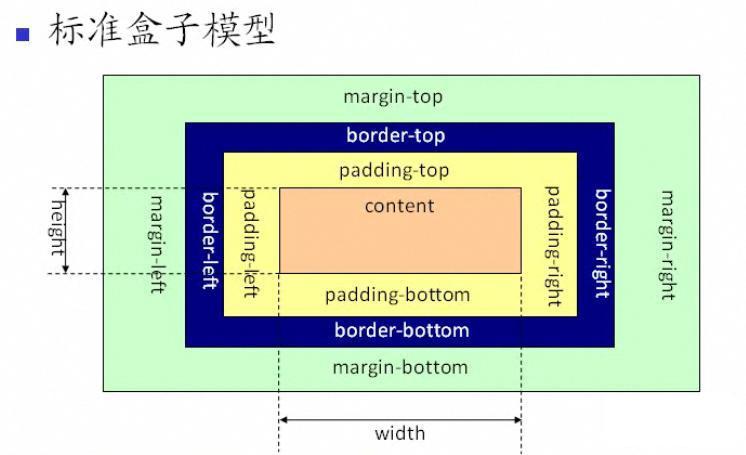
例如一个ul下有8个li，现要求点击li时输出li的id：

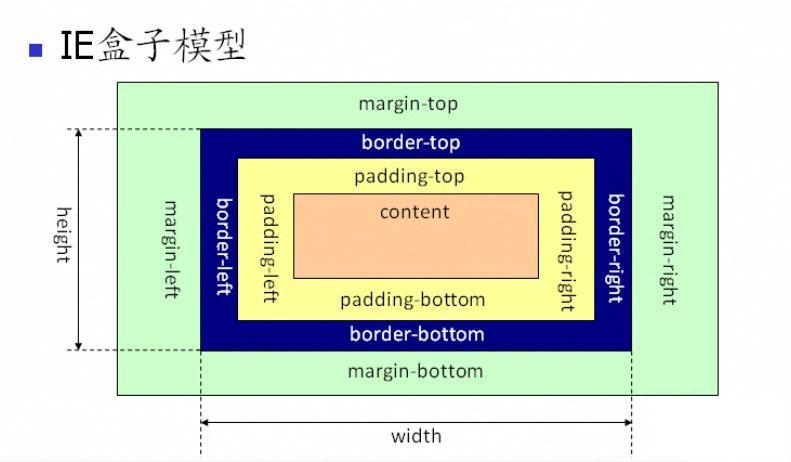
$("ul").on('click','li',function(){

console.log($(this).attr("id"));

})

**盒模型**





标准 w3c 盒子模型的范围包括 margin、border、padding、content，并且 content 部分不包含其他部分。

ie 盒子模型的范围也包括 margin、border、padding、content，和标准 w3c 盒子模型不同的是：ie 盒子模型的 content 部分包含了 border 和 padding。

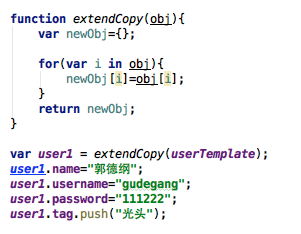
例：一个盒子的 margin 为 20px，border 为 1px，padding 为 10px，content 的宽为 200px、高为 50px，假如用标准 w3c 盒子模型解释，那么这个盒子需要占据的位置为：宽 20\*2+1\*2+10\*2+200=262px、高 20\*2+1\*2\*10\*2+50=112px，盒子的实际大小为：宽 1\*2+10\*2+200=222px、高 1\*2+10\*2+50=72px；假如用ie 盒子模型，那么这个盒子需要占据的位置为：宽 20\*2+200=240px、高 20\*2+50=70px，盒子的实际大小为：宽 200px、高 50px。（盒子实际大小不包括margin）

**对象浅克隆**

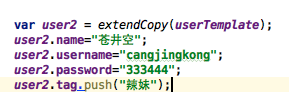
想象一个场景：我们希望给每一个注册用户建立一个数据模型，在此之前，我们要把这个用户的数据模型模板给创建出来：



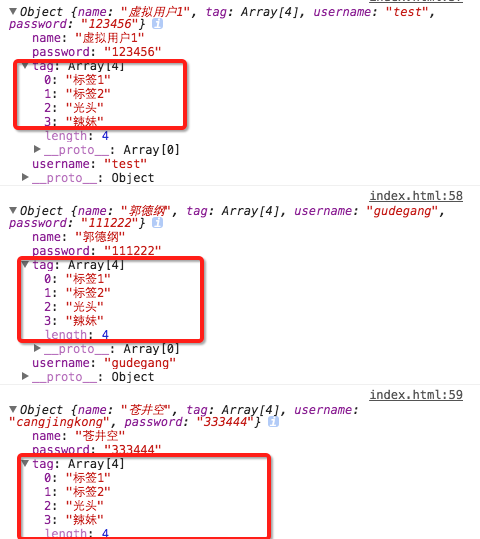
现在来了一个新用户来注册，这个用户就是我非常喜欢的人民艺术家郭德纲。哈哈。我用这个模板对象复制一个新对象来,就像刚才说的，我写一个函数，把userTemplate对象的每一个属性都复制到这个新对象里，代码是这样的：



然后我们又来了一位用户，是位美女，于是我们用刚才写的函数新复制一个对象：

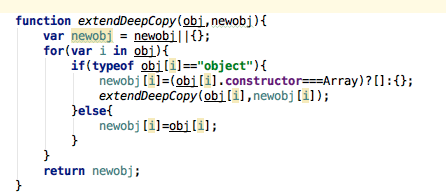


下面我在把模型插入数据库之前，我要打印出来三个对象看看内容是不是正确，打印的结果是这样的：



这就是**浅克隆**。因为简单的复制对象，如果对象其中一个属性是引用型变量，就会出现这种情况，因为引用型变量保存的是内存地址，所以其实后来操作的都是同一块内存，导致了数组内容都一样。

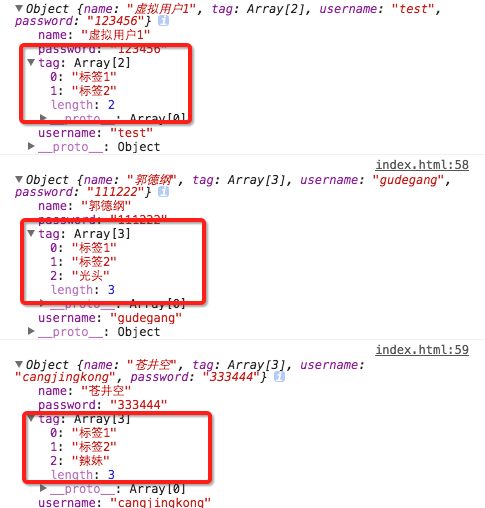
**对象深克隆**



然后用这个新写的函数去复制对象，重新让郭德纲和苍井空两位老师注册一下。



这样的话再打印出来看看：



写一个traverse函数，输出所有页面宽度和高度大于50像素的节点。

function traverse(oNode) {

    var aResult = [];

    oNode = oNode || document.body;

    if (oNode.style) {

        var nWidth = window.parseInt(oNode.style.width, 10) || 0;

        var nHeight = window.parseInt(oNode.style.height, 10) || 0;

        if (nWidth > 50 && nHeight > 50) {

            aResult.push(oNode);

        }

    }

    var aChildNodes = oNode.childNodes;

    if (aChildNodes.length > 0) {

        for (var i = 0, l = aChildNodes.length; i < l; i++) {

            var oTmp = aChildNodes[i];

            aResult = aResult.concat(traverse(oTmp));

        }

    }

    return aResult;

}

排序：

冒泡、选择、插入、归并、希尔、堆、快速

查找

二分查找法

二叉查找树

红黑树

哈希表

跳表

构建一个hash set的步骤：

1.底层数据结构：hash表，数组+链表

2.节点的抽象：Entry（key,value,hash,next）

3.hash函数

进阶

4.扩容（每当使用容量超过负载因子时，容量自动扩大1倍）、负载因子（默认75%，超出后性能急剧下降）

5.hashcode，equals

6.线程安全

理解：hash set为何可以去重（equals）

23456树→红黑树

1.插入节点一定为红

2.按照普通的二叉排序树插入，若为右子树，则通过一定规则变为左子树

3.变化规则：只可能有三种情况，分别对应右旋，左旋，flip color。

递归

遍历树：前序，中序，后序，层序

[JavaScript 命名空间](http://www.cnblogs.com/dolphinX/p/3269145.html)

使用过Java、C#的同学对命名空间非常的熟悉，在复杂的系统中会有N多的函数、对象，语言提供的、架构预定义的，这么多的函数和对象，由于编程规范要求起有实际意义的名字，难免会重名发生错误调用，而有了命名空间烦恼就没有了，不但可以分类组织函数与对象，还可以形成隔离，解决重名问题。

使用JavaScript就没有这么舒服了，Javascript只有函数作用域，什么块儿啊、神马文件啊统统都认为是一个命名空间的，有时候因为一些重名问题导致的错误让人莫名其妙，难以调试解决。

### 一个简单的例子

<input type="button" value="test" onclick="alert();"/>

<script type="text/javascript">

function alert(){

//.......

test2();

//.......

}

function test2(){

alert('test2')

}

在个例子在不同的浏览器下有不同表现，IE会报Stack over flow, Firefox会死掉。。。反正都会报错，很简单的错误，代码中自定义了一个alert函数，在alert函数中调用了test2函数，test2函数中意图调用window的alert方法，这样循环调用了，也许看了你会说这么明显的错误谁会犯，但是如果自定义的方法叫close（这个经常会出现吧），然后内部调用了一个外部文件的函数，该函数调用了window的close方法，这样错误是不是隐蔽了很多呢。

### 简单的命名空间

由于JavaScript没有文件作用域，不同的函数分散在不同的文件中，甚至由不同的人编写，重名的概率大大增加。是不是足够小心就可以了呢？也不尽然，还有些意外情况，比如经常会用到继承，于是写了一个没出现过的函数名extend，不料在EcmaScript5中加入了extend函数，命名空间的必要性就体现出来了。

JavaScript有函数的作用域，可以利用这点把自定义的函数写到一个函数体内，这样函数内的变量、对象、函数就像在一个命名空间内一样和外部隔离。

<input type="button" value="test" onclick="(new namespace()).alert();"/>

<script type="text/javascript">

function namespace(){

this.alert=function(){

console.log('test');

}

}

</script>

这样自定义的alert方法就不会和window的alert冲突了。

### 简单进化

这样可以是可以，但也有问题，最大的问题在于调用方式复杂而丑陋！每次调用的时候都要实例化对象，然后调用其方法，简单修改代码让其实现自动实例化。

<input type="button" value="test" onclick="NS.alert();"/>

<script type="text/javascript">

(function namespace(){

this.alert=function(){

console.log('test');

}

window.NS=this;

})();

</script>

要看明白上面代码首先要了解一下“立即执行函数”（江湖人是这么称呼的）的技巧结构类似这样

(function xxx(){

//function body

})();

这样写xxx函数就可以在定义完后自动执行，看起来神奇，其实上面写法可以拆成这样

function xxx(){

//function body

}

xxx();

就是定义一个函数，然后使用括号语法调用，而函数定义外面的一层括号只起到将函数声明转为函数定义表达式，因为只有表达式才可以使用括号调用。看明白这些妖蛾子之后上面代码就简单了，在自定义namespace函数最后把this赋值为window的NS属性，在调用的时候直接使用NS.xx就可以了。看起来好了很多。

### 美化一下

上面的写法看起来不错了，但是函数名namespace貌似是多余的了，可以美化一下

(function (){

this.alert=function(){

console.log('test');

}

window.NS=this;

})();

变成了一个立即执行的匿名函数，美化了一些，不过看起来还是怪怪的，对呀，明明是实例化的function，为什么方法定义不写到prototype中呢，匿名函数怎么写prototype。。。，还得动动脑筋

(function(){

var \_NS=function(){

}

\_NS.prototype.alert=function(){

console.log('test');

}

window.NS=new \_NS();

})();

### 写几个有用的函数

querySelector和querySelectorAll是W3C提供的新的查询接口，但是名字好长，自己写个简单的，innerHTML属性也常用到，写个简单版仿jQuery的html方法

(function () {

var \_NS = function () {

}

\_NS.prototype.select = function (selector,context) {

var context = context || document;

return context.querySelectorAll(selector);

}

\_NS.prototype.isArrayLike=function(obj){

if(obj instanceof Array){

return true;

}

var length=obj.length;

if ( obj.nodeType === 1 && length ) {

return true;

}

return false;

}

\_NS.prototype.html = function (obj,value) {

var isArray=this.isArrayLike(obj), i=0;

if (typeof value == 'string') {

if (!isArray) {

obj.innerHTML = value;

} else {

var length = obj.length;

while (i < length) {

obj[i].innerHTML = value;

i += 1;

}

}

} else {

if (!isArray) {

return obj.innerHTML;

} else {

return obj[0].innerHTML;

}

}

}

window.NS = new \_NS();

})();

这样一个带有命名空间的简单JavaScript库就写成了，不用担心命名冲突了，但是用起来很不方便啊，做前端的同学都用过jQuery，人家用起来那叫一个简单,jQuery是怎么做的？欲知后事如何，且听下回分解。

**Java 中的堆和栈**   
Java把内存划分成两种：一种是栈内存，一种是堆内存。

1.栈(stack)与堆(heap)都是Java用来在Ram中存放数据的地方。与C++不同，Java自动管理栈和堆，程序员不能直接地设置栈或堆。

2.栈的优势是，存取速度比堆要快，仅次于直接位于CPU中的寄存器。但缺点是，存在栈中的数据大小与生存期必须是确定的，缺乏灵活性。另外，**栈数据可以共享**。堆的优势是可以动态地分配内存大小，生存期也不必事先告诉编译器，Java的垃圾收集器会自动收走这些不再使用的数据。但缺点是，由于要在运行时动态分配内存，存取速度较慢。    
3. Java中的数据类型有两种。

　　一种是**基本类型**(primitive types), 共有8种，即int, short, long, byte, float, double, boolean, char(注意，并没有string的基本类型)。这种类型的定义是通过诸如int a = 3; long b = 255L;的形式来定义的，称为自动变量。值得注意的是，自动变量存的是字面值，不是类的实例，即不是类的引用，这里并没有类的存在。如int a = 3; 这里的a是一个指向int类型的引用， 指向3这个字面值。这些字面值的数据，由于大小可知，生存期可知(这些字面值固定定义在某个程序块里面，程序块退出后，字段值就消失了)， 出于追求速度的原因，就存在于栈中。另外，栈有一个很重要的特殊性，就是存在栈中的数据可以共享。假设我们同时定义：    
int a = 3;    
int b = 3;    
　　编译器先处理int a = 3；首先它会在栈中创建一个变量为a的引用，然后查找有没有字面值为3的地址，没找到，就开辟一个存放3这个字面值的地址，然后将a指向3的地址。接着处理int b = 3；在创建完b的引用变量后，由于在栈中已经有3这个字面值，便将b直接指向3的地址 。这样，就出现了a与b同时均指向3的情况。    
　　特别注意的是，这种字面值的引用与类对象的引用不同。假定两个类对象的引用同时指向一个对象，如果一个对象引用变量修改了这个对象的内部状态，那么另一个对象引用变量也即刻反映出这个变化。相反，通过字面值的引用来修改其值，不会导致另一个指向此字面值的引用的值也跟着改变的情况。如上例，我们定义完a与b的值后，再令a=4；那么，b不会等于4，还是等于3。在编译器内部，遇到a=4；时，它就会重新搜索栈中是否有4的字面值，如果没有，重新开辟地址存放4的值；如果已经有了，则直接将a指向这个地址。因此a值的改变不会影 响到b的值。

另一种是**包装类数据**，如Integer, String, Double等将相应的基本数据类型包装起来的类。这些类数据全部存在于堆中，**Java用new()语句来显示地告诉编译器，在运行时才根据需要动态创建，因此比较灵活，但缺点是要占用更多的时间。**   
在JAVA中，有六个不同的地方可以存储数据：    
1. 寄存器（register）。这是最快的存储区，因为它位于不同于其他存储区的地方——处理器内部。但是寄存器的数量极其有限，所以寄存器由编译器根据需求进行分配。你不能直接控制，也不能在程序中感觉到寄存器存在的任何迹象。    
2. 堆栈（stack）。位于通用RAM中，但通过它的“堆栈指针”可以从处理器哪里获得支持。堆栈指针若向下移动，则分配新的内存；若向上移动，则释放那些内存。这是一种快速有效的分配存储方法，仅次于寄存器。创建程序时候，JAVA编译器必须知道存储在堆栈内所有数据的确切大小和生命周期，因为它必须生成相应的代码，以便上下移动堆栈指针。这一约束限制了程序的灵活性，所以虽然某些JAVA数据存储在堆栈中——特别是对象引用，但是JAVA对象不存储其中。    
3. 堆（heap）。一种通用性的内存池（也存在于RAM中），用于存放所以的JAVA对象。堆不同于堆栈的好处是：编译器不需要知道要从堆里分配多少存储区域，也不必知道存储的数据在堆里存活多长时间。因此，在堆里分配存储有很大的灵活性。当你需要创建一个对象的时候，只需要new写一行简单的代码，当执行这行代码时，会自动在堆里进行存储分配。当然，为这种灵活性必须要付出相应的代码。用堆进行存储分配比用堆栈进行存储存储需要更多的时间。    
4. 静态存储（static storage）。这里的“静态”是指“在固定的位置”。静态存储里存放程序运行时一直存在的数据。你可用关键字static来标识一个对象的特定元素是静态的，但JAVA对象本身从来不会存放在静态存储空间里。    
5. 常量存储（constant storage）。常量值通常直接存放在程序代码内部，这样做是安全的，因为它们永远不会被改变。有时，在嵌入式系统中，常量本身会和其他部分分割离开，所以在这种情况下，可以选择将其放在ROM中    
6. 非RAM存储。如果数据完全存活于程序之外，那么它可以不受程序的任何控制，在程序没有运行时也可以存在。    
就速度来说，有如下关系：    
寄存器 < 堆栈 < 堆 < 其他    
  
问题一：    
  
String str1 = "abc";    
String str2 = "abc";    
System.out.println(str1==str2); //true    
  
问题二：    
String str1 =new String ("abc");    
String str2 =new String ("abc");    
System.out.println(str1==str2); // false    
问题三：    
String s1 = "ja";    
String s2 = "va";    
String s3 = "java";    
String s4 = s1 + s2;    
System.out.println(s3 == s4);//false    
System.out.println(s3.equals(s4));//true    
  
在函数中定义的一些基本类型的变量和对象的引用变量都在函数的栈内存中分配。    
  
当在一段代码块定义一个变量时，Java就在**栈**中为这个变量分配内存空间，当超过变量的作用域后，Java会自动释放掉为该变量所分配的内存空间，该内存空间可以立即被另作他用。    
  
**堆内存用来存放由new创建的对象和数组。**    
  
在堆中分配的内存，由Java虚拟机的自动垃圾回收器来管理。

在堆中产生了一个数组或对象后，还可以在栈中定义一个特殊的变量，让栈中这个变量的取值等于数组或对象在堆内存中的首地址，栈中的这个变量就成了数组或对象的引用变量。

引用变量就相当于是为数组或对象起的一个名称，以后就可以在程序中使用栈中的引用变量来访问堆中的数组或对象。    
具体的说：    
栈与堆都是Java用来在Ram中存放数据的地方。与C++不同，Java自动管理栈和堆，程序员不能直接地设置栈或堆。    
Java的堆是一个运行时数据区,类的对象从中分配空间。这些对象通过new、newarray、anewarray和multianewarray等指令建立，它们不需要程序代码来显式的释放。堆是由垃圾回收来负责的，堆的优势是可以动态地分配内存大小，生存期也不必事先告诉编译器，因为它是在运行时动态分配内存的，Java的垃圾收集器会自动收走这些不再使用的数据。但缺点是，由于要在运行时动态分配内存，存取速度较慢。    
      栈的优势是，存取速度比堆要快，仅次于寄存器，栈数据可以共享。但缺点是，存在栈中的数据大小与生存期必须是确定的，缺乏灵活性。**栈中主要存放一些基本类型的变量**（int, short, long, byte, float, double, boolean, char）**和对象句柄**。    
 String是一个特殊的包装类数据。可以用：    
String str = new String("abc");    
String str = "abc";    
两种的形式来创建，第一种是用new()来新建对象的，它会在**存放于堆中**。每调用一次就会创建一个新的对象。    
而第二种是先在栈中创建一个对String类的对象引用变量str，然后查找栈中有没有存放"abc"，如果没有，则将"abc"存放进栈，并令str指向”abc”，如果已经有”abc” 则直接令str指向“abc”。    
  
比较类里面的数值是否相等时，用equals()方法；当测试两个包装类的引用是否指向同一个对象时，用==，下面用例子说明上面的理论。    
String str1 = "abc";    
String str2 = "abc";    
System.out.println(str1==str2); //true    
可以看出str1和str2是指向同一个对象的。    
  
String str1 =new String ("abc");    
String str2 =new String ("abc");    
System.out.println(str1==str2); // false    
用new的方式是生成不同的对象。每一次生成一个。  因此用第二种方式创建多个”abc”字符串,在内存中其实只存在一个对象而已. 这种写法有利与节省内存空间. 同时它可以在一定程度上提高程序的运行速度，因为JVM会自动根据栈中数据的实际情况来决定是否有必要创建新对象。而对于String str = new String("abc")；的代码，则一概在堆中创建新对象，而不管其字符串值是否相等，是否有必要创建新对象，从而加重了程序的负担。  另一方面, 要注意: 我们在使用诸如String str = "abc"；的格式定义类时，总是想当然地认为，创建了String类的对象str。（不一定，因为如果事先没有，那么就会创建，这就是创建对象了，如果原来就有，那就指向那个原来的对象就可以了）！对象可能并没有被创建！而可能只是指向一个先前已经创建的对象。只有通过new()方法才能保证每次都创建一个新的对象。由于String类的immutable性质，**当String变量需要经常变换其值时，应该考虑使用StringBuffer类，以提高程序效率**。

**XMLHttpRequest对象详解**

**属性**



方法



**XMLHttpRequest的发展历程**

XMLHttpRequest一开始只是微软浏览器提供的一个接口，后来各大浏览器纷纷效仿也提供了这个接口，再后来W3C对它进行了标准化，提出了[XMLHttpRequest标准](https://www.w3.org/TR/XMLHttpRequest/)。XMLHttpRequest标准又分为Level 1和Level 2。  
XMLHttpRequest Level 1主要存在以下缺点：

* 受同源策略的限制，不能发送跨域请求；
* 不能发送二进制文件（如图片、视频、音频等），只能发送纯文本数据；
* 在发送和获取数据的过程中，无法实时获取进度信息，只能判断是否完成；

那么Level 2对Level 1 进行了改进，XMLHttpRequest Level 2中新增了以下功能：

* 可以发送跨域请求，在服务端允许的情况下；
* 支持发送和接收二进制数据；
* 新增formData对象，支持发送表单数据；
* 发送和获取数据时，可以获取进度信息；
* 可以设置请求的超时时间；

XMLHttpRequest**兼容性**

关于xhr的浏览器兼容性：

* IE8/IE9、Opera Mini 完全不支持xhr对象
* IE10/IE11部分支持，不支持 xhr.responseType为json
* 部分浏览器不支持设置请求超时，即无法使用xhr.timeout
* 部分浏览器不支持xhr.responseType为blob

## 细说XMLHttpRequest如何使用

先来看一段使用XMLHttpRequest发送Ajax请求的简单示例代码。

function sendAjax() {

//构造表单数据

var formData = new FormData();

formData.append('username', 'johndoe');

formData.append('id', 123456);

//创建xhr对象

var xhr = new XMLHttpRequest();

//设置xhr请求的超时时间

xhr.timeout = 3000;

//设置响应返回的数据格式

xhr.responseType = "text";

//创建一个 post 请求，采用异步

xhr.open('POST', '/server', true);

//注册相关事件回调处理函数

xhr.onload = function(e) {

if(this.status == 200||this.status == 304){

alert(this.responseText);

}

};

xhr.ontimeout = function(e) { ... };

xhr.onerror = function(e) { ... };

xhr.upload.onprogress = function(e) { ... };

//发送数据

xhr.send(formData);

}

**如何设置request header**

在发送Ajax请求（实质是一个[HTTP](http://www.tutorialspoint.com/http/http_header_fields.htm)请求）时，我们可能需要设置一些请求头部信息，比如content-type、connection、cookie、accept-xxx等。xhr提供了setRequestHeader来允许我们修改请求 header。

void setRequestHeader(DOMString header, DOMString value);

**注意点**：

* 方法的第一个参数 header 大小写不敏感，即可以写成content-type，也可以写成Content-Type，甚至写成content-Type;
* Content-Type的默认值与具体发送的数据类型有关，请参考本文【可以发送什么类型的数据】一节；
* setRequestHeader必须在open()方法之后，send()方法之前调用，否则会抛错；
* setRequestHeader可以调用多次，最终的值不会采用覆盖override的方式，而是采用追加append的方式。

**如何获取request header**

DOMString getAllResponseHeaders();  
DOMString getResponseHeader(DOMString header);

W3C的xhr标准有限制，这两个方法只能获取限制以外的header字段。

**如何指定xhr.response的数据类型**

xhr.overrideMimeType('text/xml; charset = utf-8')

**如何获取response的数据**

xhr提供了3个属性来获取请求返回的数据，分别是：xhr.response、xhr.responseText、xhr.responseXML

* xhr.response
  + 默认值：空字符串""
  + 当请求完成时，此属性才有正确的值
  + 请求未完成时，此属性的值可能是""或者 null，具体与 xhr.responseType有关：当responseType为""或"text"时，值为""；responseType为其他值时，值为 null
* xhr.responseText
  + 默认值为空字符串""
  + 只有当 responseType 为"text"、""时，xhr对象上才有此属性，此时才能调用xhr.responseText，否则抛错
  + 只有当请求成功时，才能拿到正确值。以下2种情况下值都为空字符串""：请求未完成、请求失败
* xhr.responseXML
  + 默认值为 null
  + 只有当 responseType 为"text"、""、"document"时，xhr对象上才有此属性，此时才能调用xhr.responseXML，否则抛错
  + 只有当请求成功且返回数据被正确解析时，才能拿到正确值。以下3种情况下值都为null：请求未完成、请求失败、请求成功但返回数据无法被正确解析时

**如何追踪xhr当前状态**



**如何设置请求超时时间**

xhr.timeout，单位毫秒，默认值为0。

由xhr.onloadstart事件触发，即在调用send()方法的时候开始。

在xhr.onloadend事件触发时结束。

xhr.timeout可以在send()之前调用，但仍以send()为开始。xhr为一个同步请求时，xhr.timeout必须为0。

**open()方法**

open(method, url [, async = true [, username = null [, password = null]]])

* method: 请求的方式，如GET/POST/HEADER等，这个参数不区分大小写
* url: 请求的地址，可以是相对地址如example.php，这个**相对**是相对于当前网页的url路径；也可以是绝对地址如http://www.example.com/example.php
* async: 默认值为true，即为异步请求，若async=false，则为同步请求

**send()方法**

xhr.send(data)的参数data可以是以下几种类型：

* ArrayBuffer
* Blob
* Document
* DOMString
* FormData
* null

如果是 GET/HEAD请求，send()方法一般不传参或传 null。不过即使你真传入了参数，参数也最终被忽略，xhr.send(data)中的data会被置为 null.

xhr.send(data)中data参数的数据类型会影响请求头部content-type的默认值：

* 如果data是 Document 类型，同时也是HTML Document类型，则content-type默认值为text/html;charset=UTF-8;否则为application/xml;charset=UTF-8；
* 如果data是 DOMString 类型，content-type默认值为text/plain;charset=UTF-8；
* 如果data是 FormData 类型，content-type默认值为multipart/form-data; boundary=[xxx]
* 如果data是其他类型，则不会设置content-type的默认值

当然这些只是content-type的默认值，但如果用xhr.setRequestHeader()手动设置了中content-type的值，以上默认值就会被覆盖。

**xhr相关事件**

interface XMLHttpRequestEventTarget : EventTarget {

// event handlers

attribute EventHandler onloadstart;

attribute EventHandler onprogress;

attribute EventHandler onabort;

attribute EventHandler onerror;

attribute EventHandler onload;

attribute EventHandler ontimeout;

attribute EventHandler onloadend;

};

interface XMLHttpRequestUpload : XMLHttpRequestEventTarget {

};

interface XMLHttpRequest : XMLHttpRequestEventTarget {

// event handler

attribute EventHandler onreadystatechange;

readonly attribute XMLHttpRequestUpload upload;

};



1. XMLHttpRequestEventTarget接口定义了7个事件：
   * onloadstart
   * onprogress
   * onabort
   * ontimeout
   * onerror
   * onload
   * onloadend
2. 每一个XMLHttpRequest里面都有一个upload属性，而upload是一个XMLHttpRequestUpload对象
3. XMLHttpRequest和XMLHttpRequestUpload都继承了同一个XMLHttpRequestEventTarget接口，所以xhr和xhr.upload都有第一条列举的7个事件
4. onreadystatechange是XMLHttpRequest独有的事件

所以这么一看就很清晰了：  
xhr一共有8个相关事件：7个XMLHttpRequestEventTarget事件+1个独有的onreadystatechange事件；而xhr.upload只有7个XMLHttpRequestEventTarget事件。

**事件触发顺序**

当请求一切正常时，相关的事件触发顺序如下：

1. 触发xhr.onreadystatechange(之后每次readyState变化时，都会触发一次)
2. 触发xhr.onloadstart  
   //上传阶段开始：
3. 触发xhr.upload.onloadstart
4. 触发xhr.upload.onprogress//提交请求的响应回调
5. 触发xhr.upload.onload
6. 触发xhr.upload.onloadend  
   //上传结束，下载阶段开始：
7. 触发xhr.onprogress//上传成功的响应回调
8. 触发xhr.onload
9. 触发xhr.onloadend

java进制转换

java中进行二进制，八进制，十六进制，十进制间进行相互转换

十进制转成十六进制：

Integer.toHexString(int i)

十进制转成八进制

Integer.toOctalString(int i)

十进制转成二进制

Integer.toBinaryString(int i)

十六进制转成十进制

Integer.valueOf("FFFF",16).toString()

八进制转成十进制

Integer.valueOf("876",8).toString()

二进制转十进制

Integer.valueOf("0101",2).toString()java进制转换

java中进行二进制，八进制，十六进制，十进制间进行相互转换

十进制转成十六进制：

Integer.toHexString(int i)

十进制转成八进制

Integer.toOctalString(int i)

十进制转成二进制

Integer.toBinaryString(int i)

十六进制转成十进制

Integer.valueOf("FFFF",16).toString()

八进制转成十进制

Integer.valueOf("876",8).toString()

二进制转十进制

Integer.valueOf("0101",2).toString()

定义一个构造函数：

function CO(){

this.p = “I'm in constructed object”;

this.alertP = function(){

alert(this.p);

}

}

var o2 = newCO();

在使用new操作符来调用一个构造函数的时候，发生了以下四件事：

var obj ={};

obj.\_\_proto\_\_ = CO.prototype;

CO.call(obj);

return obj;

第一行，创建一个空对象obj。

第二行，将这个空对象的\_\_proto\_\_成员指向了构造函数对象的prototype成员对象，这是最关键的一步，具体细节将在下文描述。

第三行，将构造函数的作用域赋给新对象，因此CA函数中的this指向新对象obj，然后再调用CO函数。于是我们就给obj对象赋值了一个成员变量p，这个成员变量的值是“ I'm in constructed object”。

第四行，返回新对象obj。当构造函数里包含返回语句时情况比较特殊，这种情况会在下文中说到。

最重要的是第二步，将新生成的对象的\_\_prop\_\_属性赋值为构造函数的prototype属性，使得通过构造函数创建的所有对象可以共享相同的原型。这意味着同一个构造函数创建的所有对象都继承自一个相同的对象，因此它们都是同一个类的对象。

不同于其它的主流编程语言，JavaScript的构造函数并不是作为类的一个特定方法存在的；当任意一个普通函数用于创建一类对象时，它就被称作构造函数，或构造器。一个函数要作为一个真正意义上的构造函数，需要满足下列条件：

1、 在函数内部对新对象（this）的属性进行设置，通常是添加属性和方法。

2、 构造函数可以包含返回语句（不推荐），但返回值必须是this，或者其它非对象类型的值。

上文定义的构造函数CO就是一个标准的、简单的构造函数。下面例子定义的函数C1返回了一个对象，我们可以使用new表达式来调用它，该表达式可以正确返回一个对象：

function C1(){

var o = {

p:”I’m p in C1”

}

return o;

}

var o1 = new C1();

alert(o1.p);//I’m p in C1

但这种方式并不是值得推荐的方式，因为对象o1的原型是函数C1内部定义的对象o的原型，也就是Object.prototype。这种方式相当于执行了正常new表达式的前三步，而在第四步的时候返回了C1函数的返回值。该方式同样不便于创建大量相同类型的对象，不利于使用继承等高级特性，并且容易造成混乱，应该摒弃。

一个构造函数在某些情况下完全可以作为普通的功能函数来使用，这是JavaScript灵活性的一个体现。下例定义的C2就是一个“多用途”函数：

function C2(a, b){

this.p = a + b;

this.alertP = function(){

alert(this.p);

}

return this.p;//此返回语句在C2作为构造函数时没有意义

}

var c2 = new C2(2,3);

c2.alertP();//结果为5

alert(C2(2, 3)); //结果为5

该函数既可以用作构造函数来构造一个对象，也可以作为普通的函数来使用。用作普通函数时，它接收两个参数，并返回两者的相加的结果。为了代码的可读性和可维护性，建议作为构造函数的函数不要掺杂除构造作用以外的代码；同样的，一般的功能函数也不要用作构造对象。

VO的声明

当前ECMAScript规范指出独立的作用域只能通过函数代码类型的执行上下文创建：

var a = 10; // 全局上下文中的变量

(function () {

var b = 20; // function上下文中的局部变量

})();

alert(a); // 10

alert(b); // 全局变量 "b" 没有声明

for (var k in {a: 1, b: 2}) {

alert(k);

}

alert(k); // 尽管循环已经结束但变量k依然在当前作用域

如果变量与执行上下文相关，那变量自己应该知道它的数据存储在哪里，并且知道如何访问。这种机制称为变量对象(variable object)。

变量对象(缩写为VO)是一个与执行上下文相关的特殊对象，它存储着在上下文中声明的以下内容：

1）变量 (var, 变量声明);

2）函数声明 (FunctionDeclaration, 缩写为FD);

3）函数的形参

举例来说，我们可以用普通的ECMAScript对象来表示一个变量对象：VO = {};

就像我们所说的, VO就是执行上下文的属性(property)：

activeExecutionContext = {

VO: {

// 上下文数据（var, FD, function arguments)

}

};

只有全局上下文的变量对象允许通过VO的属性名称来间接访问(因为在全局上下文里，全局对象自身就是变量对象，稍后会详细介绍)，在其它上下文中是不能直接访问VO对象的，因为它只是内部机制的一个实现。

当我们声明一个变量或一个函数的时候，和我们创建VO新属性的时候一样没有别的区别（即：有名称以及对应的值）。例如：

var a = 10;

function test(x) {

var b = 20;

};

test(30);

对应的变量对象是：

// 全局上下文的变量对象

VO(globalContext) = {

a: 10,

test: <reference to function>

};

// test函数上下文的变量对象

VO(test functionContext) = {

x: 30,

b: 20

};

在具体实现层面(以及规范中)变量对象只是一个抽象概念。(从本质上说，在具体执行上下文中，VO名称是不一样的，并且初始结构也不一样。

注：只需清楚函数作用域，以及全局上下文变量对象和函数上下文变量对象。全局上下文变量对象允许通过VO的属性名称来间接访问，而后者不行。

对于所有类型的执行上下文来说，变量对象的一些操作(如变量初始化)和行为都是共通的。从这个角度来看，把变量对象作为抽象的基本事物来理解更为容易。同样在函数上下文中也定义和变量对象相关的额外内容。

抽象变量对象VO (变量初始化过程的一般行为)

║

╠══> 全局上下文变量对象GlobalContextVO

║ (VO === this === global)

║

╚══> 函数上下文变量对象FunctionContextVO

(VO === AO, 并且添加了<arguments>和<formal parameters>)

全局对象(Global object)是在进入任何执行上下文之前就已经创建了的对象。这个对象只存在一份，它的属性在程序中任何地方都可以访问，全局对象的生命周期终止于程序退出那一刻。

1.制作一个网页的工作流程

内容分析：分清展现在网络中内容的层次和逻辑关系

结构设计：写出合理的HTML结构代码

布局设计：使用HTML和css进行布局

样式设计：首先使用reset.css

交互设计：鼠标特效

行为设计：js代码，ajax页面行为和从服务器获取数据

兼容性测试

性能优化

2.HTML引入css的方法

1）行内式：行内式是在标记的style属性中设定CSS样式。这种方式没有体现出CSS的优势，不推荐使用。

2）嵌入式：嵌入式是将CSS样式集中写在网页的<head></head>标签对的<style></style>标签对中。格式如下：

<head>

<style type="text/css">

...此处写CSS样式

</style>

</head>

缺点是对于一个包含很多网页的网站，在每个网页中使用嵌入式，进行修改样式时非常麻烦。单一网页可以考虑使用嵌入式。

3）导入式：将一个独立的.css文件引入HTML文件中，导入式使用CSS规则引入外部CSS文件，<style>标记也是写在<head>标记中，使用的语法如下：

<style type="text/css">

@import"mystyle.css"; 此处要注意.css文件的路径

</style>

导入式会在整个网页装载完后再装载CSS文件，因此这就导致了一个问题，如果网页比较大则会儿出现先显示无样式的页面，闪烁一下之后，再出现网页的样式。这是导入式固有的一个缺陷。

4）链接式：也是将一个.css文件引入到HTML文件中，但它与导入式不同的是链接式使用HTML规则引入外部CSS文件，它在网页的<head></head>标签对中使用<link>标记来引入外部样式表文件，使用语法如下：

<link href="mystyle.css" rel="stylesheet" type="text/css"/>

使用链接式时与导入式不同的是它会以网页文件主体装载前装载CSS文件，因此显示出来的网页从一开始就是带样式的效果的，它不会象导入式那样先显示无样式的网页，然后再显示有样式的网页，这是链接式的优点。

总结：一般来说，做网站时把样式多写在多个样式表文件中，因此我们先用链接式引入一个总的CSS文件，然后在这个CSS文件中在使用导入式来引入其他的CSS文件。但如果通过JavaScrip来动态引入CSS文件则只能使用链接式。

3.渐进增强和优雅降级

.transition { /\*渐进增强写法\*/

-webkit-transition: all .5s;

-moz-transition: all .5s;

-o-transition: all .5s;

transition: all .5s;

}

.transition { /\*优雅降级写法\*/

transition: all .5s;

-o-transition: all .5s;

-moz-transition: all .5s;

-webkit-transition: all .5s;

}

优雅降级和渐进增强印象中是随着css3流出来的一个概念。由于低级浏览器不支持css3，但css3的效果又太优秀不忍放弃，所以在高级浏览中使用css3而低级浏览器只保证最基本的功能。咋一看两个概念差不多，都是在关注不同浏览器下的不同体验，关键的区别是他们所侧重的内容，以及这种不同造成的工作流程的差异。

渐进增强 progressive enhancement：针对低版本浏览器进行构建页面，保证最基本的功能，然后再针对高级浏览器进行效果、交互等改进和追加功能达到更好的用户体验。

优雅降级 graceful degradation：一开始就构建完整的功能，然后再针对低版本浏览器进行兼容。

区别：优雅降级是从复杂的现状开始，并试图减少用户体验的供给，而渐进增强则是从一个非常基础的，能够起作用的版本开始，并不断扩充，以适应未来环境的需要。降级（功能衰减）意味着往回看；而渐进增强则意味着朝前看，同时保证其根基处于安全地带。

“渐进增强”观点认为应关注于内容本身。

内容是我们建立网站的诱因。有的网站展示它，有的则收集它，有的寻求，有的操作，还有的网站甚至会包含以上的种种，但相同点是它们全都涉及到内容。这使得“渐进增强”成为一种更为合理的设计范例。这也是它立即被 Yahoo! 所采纳并用以构建其“分级式浏览器支持 (Graded Browser Support)”策略的原因所在。

4.网站资源优化及存储

1）合并文件，减少http请求。Web性能优化最佳实践中最重要的一条是减少HTTP 请求，它也是YSlow中比重最大的一条规 则。减少HTTP请求的方案主要有合并JavaScript和CSS文件、CSS Sprites、图像映射 （Image Map）和使用Data URI来编码图片。CSS Sprites和图像映射现在已经随处可见了，但由于IE6和IE7不支持Data URI以及性能问题，这项技术尚未大量使用。目前大部分网页中的JavaScript和CSS文件数量和开发时一致，少量的网页会根据实际情况采取本地合 并，这些合并中相当多的是有选择地手动完成，每次新的合并都需要重新在本地完成并上传到服务器，比较的随意和繁琐，同样文件的压缩也有类似的情况。而利用 服务端的合并和压缩，我们就可以按照开发的逻辑尽可能让文件的颗粒度变小，利用网页中URL的规则来自动实现文件的合并和压缩，这会相当的灵活和高效。

减少http请求有这样几个优点:

(1) 减少DNS请求所耗费的时间

(2) 减少服务器压力(CPU,IO)

(3) 减少http请求头(当我们对服务器发起一个请求的时候,我们会携带着这个域名下的cookie和一些其他的信息在http头部里,然后服务器响应请求的时候也会带回一些cookie之类的头部信息.这些信息有的时候会很大,在这种请求和响应的时候会影响带宽性能)

2）文件压缩，目的是减小文件下载的体积，常用工具是YUI Compressor。

为什么将静态图片，js，css存放到单独的域名?

使用单独域名，浏览器不会把主域名下的 cookie 传给该域，减少了网络开销，一定程度提高了页面加载速度。

动静分离，就是将 Web 应用程序中静态和动态的内容分别放在不同的 Web 服务器上，有针对性的处理动态和静态内容，从而达到性能的提升。

3）使用 CDN（内容分发网络）来托管资源；"其基本思路是尽可能避开互联网上有可能影响数据传输速度和稳定性的瓶颈和环节，使内容传输的更快、更稳定。通过在网络各处放置节点服务器所构成的在现有的互联网基础之上的一层智能虚拟网络，CDN系统能够实时地根据网络流量和各节点的连接、负载状况以及到用户的距离和响应时间等综合信息将用户的请求重新导向离用户最近的服务节点上。"形象点说：古代打仗大家一定都知道，由于古代的交通很不发达，所以当外族进攻的时候往往不能及时的反击，等朝廷征完兵再把兵派往边境的时候那些侵略者却是早已不见了踪影，这个让古代的帝王很是郁闷。后来帝王们学聪明了，都将大量的兵员提前派往边境驻扎，让他们平时屯田，战时当兵，这样的策略起到了很显著的作用。

4）缓存的使用（并且多个域名来提供缓存）

5）GZIP 压缩你的 JS 和 CSS 文件

前端技术的逐渐成熟，还衍生了domain hash, cookie free, css sprites, js/css combine, max expires time, loading images on demand等等技术。这些技术的出现和大量使用都和并发资源数有关。

按照普通设计，当网站cookie信息有1 KB、网站首页共150个资源时，用户在请求过程中需要发送150 KB的cookie信息，在512 Kbps的常见上行带宽下，需要长达3秒左右才能全部发送完毕。 尽管这个过程可以和页面下载不同资源的时间并发，但毕竟对速度造成了影响。 而且这些信息在js/css/images/flash等静态资源上，几乎是没有任何必要的。 解决方案是启用和主站不同的域名来放置静态资源，也就是cookie free。

将css放置在页面最上方应该是很自然的习惯，但第一个css内引入的图片下载是有可能堵塞后续的其他js的下载的。而在目前普遍过百的整页请求数的前提下，浏览器提供的仅仅数个并发，对于进行了良好优化甚至是前面有CDN的系统而言，是极大的性能瓶颈。 这也就衍生了domain hash技术来使用多个域名加大并发量（因为浏览器是基于domain的并发控制，而不是page），不过过多的散布会导致DNS解析上付出额外的代价，所以一般也是控制在2-4之间。 这里常见的一个性能小坑是没有机制去确保URL的哈希一致性（即同一个静态资源应该被哈希到同一个域名下），而导致资源被多次下载。

再怎么提速，页面上过百的总资源数也仍然是很可观的，如果能将其中一些很多页面都用到的元素如常用元素如按钮、导航、Tab等的背景图，指示图标等等合并为一张大图，并利用css background的定位来使多个样式引用同一张图片，那也就可以大大的减少总请求数了，这就是css sprites的由来。

全站的js/css原本并不多，其合并技术的产生却是有着和图片不同的考虑。 由于cs/js通常可能对dom布局甚至是内容造成影响，在浏览器解析上，不连贯的载入是会造成多次重新渲染的。因此，在网站变大需要保持模块化来提高可维护性的前提下，js/css combine也就自然衍生了，同时也是minify、compress等对内容进行多余空格、空行、注释的整理和压缩的技术出现的原因。

随着cookie free和domain hash的引入，网站整体的打开速度将会大大的上一个台阶。 这时我们通常看到的问题是大量的请求由于全站公有header/footer/nav等关系，其对应文件早已在本地缓存里存在了，但为了确保这个内容没有发生修改，浏览器还是需要请求一次服务器，拿到一个304 Not Modified才能放心。 一些比较大型的网站在建立了比较规范的发布制度后，会将大部分静态资源的有效期设置为最长，也就是Cache-Control max-age为10年。 这样设置后，浏览器就再也不会在有缓存的前提下去确认文件是否有修改了。 超长的有效期可以让用户在访问曾访问过的网站或网页时，获得最佳的体验。 带来的复杂性则体现在每次对静态资源进行更新时，必须发布为不同的URL来确保用户重新加载变动的资源。

即使是这样做完，仍然还存在着一个很大的优化空间，那就是很多页面浏览量很大，但其实用户直接很大比例直接就跳走了，第一屏以下的内容用户根本就不感兴趣。 对于超大流量的网站如淘宝、新浪等，这个问题尤其重要。 这个时候一般是通过将图片的src标签设置为一个loading或空白的样式，在用户翻页将图片放入可见区或即将放入可见区时再去载入。 不过这个优化其实和并发资源数的关系就比较小了，只是对一些散布不合理，或第一页底部的资源会有一定的帮助。 主要意图还是降低带宽费用。

总的来说，各类技术都是为了能让用户更快的看到页面进行下一步操作，但却不必将宝贵的资源浪费在没有必要的重复请求、不看的内容上。

5.为什么很多网站的内容储存用别的域名？有什么好处？

三个最主流的原因:

1）CDN缓存更方便

2）突破浏览器并发限制 (你随便挑一个 G家的 url: https://lh4.googleusercontent.com/-si4dh2myPWk/T81YkSi\_\_AI/AAAAAAAAQ5o/LlwbBRpp58Q/w497-h373/IMG\_20120603\_163233.jpg, 把前面的 lh4换成 lh3,lh6啥的，都照样能够访问，像地图之类的需要大量并发下载图片的站点，这个非常重要。)

3）Cookieless, 节省带宽，尤其是上行带宽 一般比下行要慢。。。

举例说一下：

twitter 的主站 http://twitter.com ，用户的每次访问，都会带上自己的cookie ，挺大的。假如twitter 的图片放在主站域名下，那么用户每次访问图片时，request header 里就会带有自己的cookie ，header 里的cookie 还不能压缩，而图片是不需要知道用户的cookie 的，所以这部分带宽就白白浪费了。

写主站程序时，set-cookie 也不要set 到图片的域名上。

在小流量的网站，这个cookie 其实节省不了多少带宽，当流量如facebook twitter 时，节省下来就很可观了。

还有另外两个非常规原因:

4）对于UGC的内容和主站隔离，防止不必要的安全问题( 上传js窃取主站cookie之类的) 。

正是这个原因要求用户内容的域名必须不是自己主站的子域名，而是一个完全独立的第三方域名。

5）数据做了划分，甚至切到了不同的物理集群，通过子域名来分流比较省事. 这个可能被用的不多。

关于Cookie的问题，带宽是次要的，安全隔离才是主要的。

关于多域名，也不是越多越好，虽然服务器端可以做泛解释，浏览器做dns解释也是耗时间的，而且太多域名，如果要走 https的话，还有要多买证书和部署的问题。

除了节约cookie带宽的因素，另一个重要因素是节约主域名的连接数，从而提高客户端网络带宽的利用率，优化页面响应。因为老的浏览器（IE6是典型），针对同一个域名只允许同时保持两个HTTP连接。将图片等资源请求分配到其他域名上，避免了大图片之类的并不一定重要的内容阻塞住主域名上其他后续资源的连接（比如ajax请求）。

6.什么是 FOUC (无样式内容闪烁)？你如何来避免 FOUC？

原因：a.使用@import方法导入CSS时，由于IE会先加载整个HTML文档的DOM，然后再去导入外部的CSS文件，因此，在页面DOM加载完成到CSS导入完成中间会有一段时间页面上的内容是没有样式的，这段时间跟网速、电脑都有关系。b.将样式表链接放在页面中不同位置时，在IE5/6下某些页面会无样式显示内容且瞬间闪烁，这现象就是文档样式短暂失效（Flash Of Unstyled Content），即FOUC。

解决方法：避免使用@import方法导入CSS，且将样式表链接都放在head中即可避免此问题。

7.CSS动画和JavaScript动画

CSS3的动画的优点：

在性能上会稍微好一些，浏览器会对CSS3的动画做一些优化（比如专门新建一个图层用来跑动画）

代码相对简单

但其缺点也很明显：

在动画控制上不够灵活

兼容性不好

部分动画功能无法实现（如滚动动画，视差滚动等）

JavaScript的动画正好弥补了这两个缺点，控制能力很强，可以单帧的控制、变换，同时写得好完全可以兼容IE6，并且功能强大。但想想CSS动画的transform矩阵是C++级的计算，必然要比javascript级的计算要快。另外对库的依赖也是一个很让人头疼的问题。

所以，对于一些复杂控制的动画，使用javascript会比较靠谱。而在实现一些小的交互动效的时候，就多考虑考虑CSS吧

8.CORS

中文名：跨域资源共享，用来Ajax解决跨域问题。

传统的JSONP只支持get，当传输数据量较大时，使用会出现问题。

CORS的原理：

CORS定义一种跨域访问的机制，可以让AJAX实现跨域访问。CORS 允许一个域上的网络应用向另一个域提交跨域 AJAX 请求。实现此功能非常简单，只需由服务器发送一个响应标头即可。

CORS浏览器支持情况：Chrome3+，Firefox3.5+，Opera12+，Safari4+，IE8+

假设我们页面或者应用已在 http://www.test1.com 上了，而我们打算从 http://www.test2.com 请求提取数据。一般情况下，如果我们直接使用 AJAX 来请求将会失败，浏览器也会返回“源不匹配”的错误，"跨域"也就以此由来。

　　利用 CORS，http://www.test2.com 只需添加一个标头，就可以允许来自 http://www.test1.com 的请求，下图是我在PHP中的 hander() 设置，“\*”号表示允许任何域向我们的服务端提交请求：header("Access-Control-Allow-Origin:\*"),也可以设置指定的域名，如域名 http://www.test2.com ，那么就允许来自这个域名的请求：header("Access-Control-Allow-Origin:http://www.test2.com")

自认为的cors使用场景：

cors在移动终端支持的不错，可以考虑在移动端全面尝试；PC上有不兼容和没有完美支持，所以小心踩坑。当然浏览器兼容就是个伪命题，说不准某个浏览器的某个版本就完美兼容了！

jsonp是get形式，承载的信息量有限，所以信息量较大时CORS是不二选择；

配合新的JSAPI(fileapi、xhr2等)一起使用，实现强大的新体验功能。

HTML相关问题

1.doctype作用

DOCTYPE是docunment type（文档定义）的简写，用来说明web设计中所用的html或xhtml的类型，指出浏览器或者其他阅读程序按照什么样的规则（W3C所发布的一个文档类型定义即DTD）集去解释文档中的标记（标准模式即严格模式，兼容模式即怪异模式即混杂模式）。

1）浏览器要使老旧的网页正常工作，但这部分网页是没有doctype声明的，所以浏览器对老旧的没有doctype声明的网页采用quirks mode解析。

2）对于那些浏览器不能识别的doctype声明，浏览器采用strict mode解析

3）在doctype声明中，没有使用DTD声明或者使用HTML4以下（不包括HTML4）的DTD声明时，基本所有的浏览器都是使用quirks mode呈现，其他的则使用strict mode解析。

4）在现有有doctype声明的网页，绝大多数是采用strict mode进行解析渲染的。

2.HTML与XHTML

可扩展超文本标记语言XHTML（eXtensible HyperText Markup Language），是HTML 4.01的第一个修订版本，是3种HTML4文件根据XML1.0标准重组而成的。也就是说是，XHTML是HTML 4.01和XML1.0的杂交。由于XHTML1.0是基于HTML4.01的，并没有引入任何新标签或属性（XHTML可以看作是HTML的一个子集），表现方式与超文本标记语言HTML类似，只是语法上更加严格，几乎所有的网页浏览器在正确解析HTML的同时，可兼容XHTML。如：XHTML中所有的标签必须小写，所有标签必须闭合，每一个属性都必须使用引号包住。<br>要写成<br />，不能写为<BR />（同hr）；使用了<p>之后必须有一个</p>以结束段落。

HTML历史：在HTML的早期发展中，W3C成立之前，很多标准的制定都是在浏览器的开发者们互相讨论的情况下完成的，比如HTML 2.0, 3.2直到4.0, 4.01，这些标准大部分都是所谓的retro-spec，即先有实现后有标准。在这种情况下，HTML标准不是很规范，浏览器也对HTML页面中的错误相当宽容。这反过来又导致了HTML作者写出了大量的含有错误的HTML页面。据说，时至今日web上99%的页面都含有HTML错误。

W3C随后意识到了这个问题，并认为这是互联网的一个基础性问题，应该加以解决。为了规范HTML，W3C结合XML制定了XHTML 1.0标准，这个标准没有增加任何新的tag，只是按照XML的要求来规范HTML，并定义了一个新的MIME type，application/xhtml+xml。W3C的初衷是对这个MIME type浏览器要实行强错误检查，既如果页面有HTML错误，就要显示错误信息。但是由于已有的web页面中已经有了大量的错误，很多开发者拒绝使用新的MIME type。W3C不得已，在XHTML 1.0的标准之后加了一个附录C，允许开发者使用XHTML语法来写页面，同时使用旧的MIME type，application/html，来分发页面。这个旧的MIME type不会触发浏览器的强错误检查。这就是我们今天看到的情况，很多网站宣称自己遵守XHTML 1.0标准，那只不过是说，他的页面中用了XHTML语法，但并不能保证完全没有错误。要验证XHTML有没有真正起效，需要查看web服务器使用哪种MIME type来分发页面的。

W3C随后在XHTML 1.1中取消了附录C，即使用XHTML 1.1标准的页面必须用新的MIME type来分发。于是这个标准并没有很多人采用。这种情况同样发生在尚未完成的XHTML 2.0身上，它要求强错误检查，于是没有人采用。XHTML的故事也告诉我们，有时候先有标准再来实现，是行不通的。

有了XHTML的教训，WHAT Working Group和W3C在制定下一代HTML标准，也就是HTML5的时候，就将向后兼容作为了一个很重要的原则。HTML5确实引入了许多新的特性，但是它最重要的一个特性是，不会break已有的网页。你可以将任何已有的网页的第一行改成<!DOCTYPE html>，它就成也一个HTML5页面，并且可以照样在浏览器里正常的展示。

3.使用data-\*属性的好处是什么？

在HTML5中添加了data-\*的方式来自定义属性，所谓data-\*实际上上就是data-前缀加上自定义的属性名，使用这样的结构可以进行数据存放。使用data-\*可以解决自定义属性混乱无管理的现状。

data-\*有两种设置方式：

1）直接在html元素标签上上书写：

<div id="test" data-age="24">

Click Here

</div>

其中的data-age就是一种自定义属性。

2）也可以通过JavaScript来对其进行操作，HTML5中元素都会有一个dataset的属性，这是一个DOMStringMap类型的键值对集合

var test = document.getElementById('test');

test.dataset.my = 'Byron';

这样就为div添加了一个data-my的自定义属性，使用JavaScript操作dataset有两个需要注意的地方

1）我们在添加或读取属性的时候需要去掉前缀data-\*，像上面的例子我们没有使用test.dataset.data-my = 'Byron';的形式。

2）如果属性名称中还包含连字符(-)，需要转成驼峰命名方式，但如果在CSS中使用选择器，我们需要使用连字符格式

<style type="text/css">

[data-birth-date]

{

background-color: #0f0;

width:100px;

margin:20px;

}

</style>

这样我们通过JavaScript设置了data-birth-date自定义属性，在CSS样式表为div添加了一些样式。

读取的时候也是通过dataset对象，使用”.”来获取属性，同样需要去掉data-前缀，连字符需要转化为驼峰命名

var test = document.getElementById('test');

test.dataset.my = 'Byron';

test.dataset.birthDate = '19890615';

test.onclick = function () {

alert(this.dataset.my + ' ' + this.dataset.age+' '+this.dataset.birthDate);

}

比较不好的消息就是data-\*的浏览器兼容性情况十分不乐观

Internet Explorer 11+

Chrome 8+

Firefox 6.0+

Opera 11.10+

Safari 6+

4.div，section和artical

div

这个标签是我们见得最多、用得最多的一个标签。本身没有任何语义，用作布局以及样式化或脚本的钩子(hook)。

section

与 div 的无语义相对，简单地说 section 就是带有语义的 div 了。section表示一段专题性的内容，一般会带有标题。当一个标签只是为了样式化或者方便脚本使用时，应该使用div。一般来说，当元素内容明确地出现在文档大纲、文章章节、博客条目、用户评论部分或者论文中有编号的部分时，section就是适用的。

article

article是一个特殊的 section 标签，它比 section 具有更明确的语义, 它代表一个独立的、完整的相关内容块。一般来说， article 会有标题部分(通常包含在 header 内)，有时也会 包含 footer 。虽然 section 也是带有主题性的一块内容，但是无论从结构上还是内容上来说，article 本身就是独立的、完整的。

当 article 内嵌 article 时，原则上来说，内部的 article 的内容是和外层的 article 内容是相关的。例如，一篇博客文章中，包含用户提交的评论的 article 就应该嵌套在包含博客文章 article 之中。

总结

div section article ，语义是从无到有，逐渐增强的。div 无任何语义，仅仅用作样式化或者脚本化的钩子(hook)，对于一段主题性的内容，则就适用 section，而假如这段内容可以脱离上下文，作为完整的独立存在的一段内容，则就适用 article。原则上来说，能使用 article 的时候，也是可以使用 section 的，但是实际上，假如使用 article 更合适，那么就不要使用 section 。nav 和 aside 的使用也是如此，这两个标签也是特殊的 section，在使用 nav 和 aside 更合适的情况下，也不要使用 section 了。

对于 div 和 section、 article 以及其他标签的区分比较简单。对于 section 和 article 的区分乍看比较难，其实重点就是看看这段内容脱离了整体是不是还能作为一个完整的、独立的内容而存在，这里面的重点又在完整身上。因为其实说起来 section 包含的内容也能算作独立的一块，但是它只能算是组成整体的一部分，article 才是一个完整的整体。

class与id的区别

1.书写上的差别：class名用“.”号开头来定义，id名用“#”号开头来定义；

2.调用上的区别：在同一个html网页页面中class是可以被多次调用的（在不同的地方）。而id名作为标签的身份则是唯一的，id在页面中只能出现一次。在js脚本中经常会用到id来修改一个标签的属性

3.id作为元素的标签，用于区分不同结构和内容，而class作为一个样式，它可以应用到任何结构和内容上。

4.在布局思路上，一般坚持这样的原则：id是先确定页面的结构和内容，然后再为它定义样式：而class相反，它先定义好一类样式，然后再页面中根据需要把类样式应用到不同的元素和内容上面。

5.在实际应用时，class更多的被应用到文字版块以及页面修饰等方面，而id更多地被用来实现宏伟布局和设计包含块，或包含框的样式。

请问 “resetting” 和 “normalizing” CSS 之间的区别？你会如何选择，为什么？

Normalize 相对「平和」，注重通用的方案，重置掉该重置的样式，保留有用的 user agent 样式，同时进行一些 bug 的修复，这点是 reset 所缺乏的。

Reset 相对「暴力」，不管你有没有用，统统重置成一样的效果，且影响的范围很大，讲求跨浏览器的一致性。

Normalize.css是一种CSS reset的替代方案。它们的区别有：

1. Normalize.css 保护了有价值的默认值，Reset通过为几乎所有的元素施加默认样式，强行使得元素有相同的视觉效果。相比之下，Normalize.css保持了许多默认的浏览器样式。这就意味着你不用再为所有公共的排版元素重新设置样式。当一个元素在不同的浏览器中有不同的默认值时，Normalize.css会力求让这些样式保持一致并尽可能与现代标准相符合。

2. Normalize.css 修复了浏览器的bug，它修复了常见的桌面端和移动端浏览器的bug。这往往超出了Reset所能做到的范畴。关于这一点，Normalize.css修复的问题包含了HTML5元素的显示设置、预格式化文字的font-size问题、在IE9中SVG的溢出、许多出现在各浏览器和操作系统中的与表单相关的bug。

3.Normalize.css 不会让你的调试工具变的杂乱

4. Normalize.css 是模块化的

5. Normalize.css 拥有详细的文档

选择Normalize.css ，主要是reset.css为几乎所有的元素施加默认样式，所以需要对所有公共的排版元素重新设置样式，这是一件很麻烦的工作。

请解释浮动 (Floats) 及其工作原理。

答：float属性定义了元素是否浮动及在哪个方向浮动，在CSS中任何元素都可以浮动，且浮动元素会生成一个块级框，而不论它本身是何种元素。并且盒子的宽度不在伸展，而是根据盒子里面的内容的宽度来确定。浮动属性会使得浮动的元素脱离文档流，所以文档的普通流中的块框会表现的像浮动框不存在一样。

描述z-index和叠加上下文是如何形成的。

首先来看在CSS中叠加上下文形成的原因：

1.负边距

margin为负值时元素会依参考线向外偏移。margin-left/margin-top的参考线为左边的元素/上面的元素（如无兄弟元素则为父元素的左内侧/上内侧）,margin-right和margin-bottom的参考线为元素本身的border右侧/border下侧。一般可以利用负边距来就行布局，但没有计算好的话就可能造成元素重叠。堆叠顺序由元素在文档中的先后位置决定，后出现的会在上面。

2.position的relative/absolute/fixed定位

当为元素设置position值为relative/absolute/fixed后，元素发生的偏移可能产生重叠，且z-index属性被激活。z-index值可以控制定位元素在垂直于显示屏方向（Z 轴）上的堆叠顺序（stack order），值大的元素发生重叠时会在值小的元素上面。

z-index属性

z-index只能在position属性值为relative或absolute或fixed的元素上有效。

基本原理：z-index值可以控制定位元素在垂直于显示屏方向（Z 轴）上的堆叠顺序（stack order），值大的元素发生重叠时会在值小的元素上面。

使用相对性：z-index值只决定同一父元素中的同级子元素的堆叠顺序。父元素的z-index值（如果有）为子元素定义了堆叠顺序（css版堆叠“拼爹”）。向上追溯找不到含有z-index值的父元素的情况下，则可以视为自由的z-index元素，它可以与父元素的同级兄弟定位元素或其他自由的定位元素来比较z-index的值，决定其堆叠顺序。同级元素的z-index值如果相同，则堆叠顺序由元素在文档中的先后位置决定，后出现的会在上面。所以如果当你发现一个z-index值较大的元素被值较小的元素遮挡了，请先检查它们之间的dom结点关系，多半是因为其父结点含有激活并设置了z-index值的position定位元素。

请描述 BFC(Block Formatting Context) 及其如何工作。

BFC（Block Formatting Context）直译为“块级格式化范围”。

block，inline和inline-block的概念以及区别

首先这是display中的三个属性值，不是指元素类型，元素类型在HTML5之前分为两种分别是块级元素（ block-level elements）和内联元素（ inline elements）。但是当display的属性值被设置为block时，元素会以块级元素（ block-level elements）显示，而设置为inline时会以联元素（ inline elements）显示。

display:block

block元素会独占一行，多个block元素会各自新起一行。默认情况下，block元素宽度自动填满其父元素宽度。

block元素可以设置width,height属性。块级元素即使设置了宽度,仍然是独占一行。

block元素可以设置margin和padding属性。

display:inline

inline元素不会独占一行，多个相邻的行内元素会排列在同一行里，直到一行排列不下，才会新换一行，其宽度随元素的内容而变化。

inline元素设置width,height属性无效。

inline元素的margin和padding属性，水平方向的padding-left, padding-right, margin-left, margin-right都产生边距效果；但竖直方向的padding-top, padding-bottom, margin-top, margin-bottom不会产生边距效果。

display:inline-block

简单来说就是将对象呈现为inline对象，但是对象的内容作为block对象呈现。之后的内联对象会被排列在同一行内。比如我们可以给一个link（a元素）inline-block属性值，使其既具有block的宽度高度特性又具有inline的同行特性。即对inline-block元素可以设置宽度和高度，同时inline-block元素又可以在同行进行排列。

备注：属性为inline-block元素之间的空格或者换行在浏览器上会是一个空白的间隙。且IE6和7是不支持这个属性的，需要通过\*display:inline;\*zoom:1做hack处理。

请解释 CSS sprites，以及你要如何在页面或网站中实现它。

通常被意译为“CSS图像拼合”或“CSS贴图定位”。

CSS Sprites就是把网页中一些背景图片整合到一张图片文件中，再利用CSS的“background-image”，“background- repeat”，“background-position”的组合进行背景定位，background-position可以用数字能精确的定位出背景图片的位置。

优点：当页面加载时，不是加载每个单独图片，而是一次加载整个组合图片。这是一个了不起的改进，它大大减少了HTTP请求的次数，减轻服务器压力，同时缩短了悬停加载图片所需要的时间延迟，使效果更流畅，不会停顿。

缺点：做图像拼合的时候很麻烦。

题目一：

if（！（"a" in window））{

var a = 1;

}

alert(a);

分析：相当于

var a；

if（！（"a" in window））{

a = 1;

}

alert(a);

所以是undefined。

题目二：

var a=1,

b= function a(x){

x&&a(-xx);

};

alert(a);

分析：

function a(){

return 1;

}

var a;

alert(typeof a); //"function"

这个题有点意思，上面说过函数声明会提前，变量声明也会提前，现在var a；要提前，function a也要提前。。。。。。到底谁会覆盖谁呢？

function a(){

return 1;

}

var a;

alert(typeof a); //"function"

很明显上面上面中函数声明会覆盖变量声明，优先级较高一点。但是改成var a=1;之后结果却是截然相反的。

function a(){

return 1;

}

var a =1;

alert(typeof a); //“number”

当变量申明遇到VO中已经有同名的时候，不会影响已经存在的属性。var a；只是一个变量声明，不会影响已经存在的function a 的值。

而当赋值的时候，就会相当于重写a。但要记住函数声明会比变量声明提前，所以一般会是被赋值重写。

也许有人或说，a被重写，在用a(--)岂不是会出现错误。在这里a没有被执行，一般 函数没有被执行 内部没有明显语法错误 可以看成黑盒

题目三：

function a(x){

return x\*2;

}

var a;

alert (a);

分析见上面，输出函数体本身。

题目四：

function b(x,y,a){

arguments[2] = 10;

alert(a);

}

b(1,2,3);

b(1,2);

分析：

活动对象是在进入函数上下文时刻被创建的，它通过函数的arguments属性初始化。arguments属性的值是Arguments对象：

AO = { arguments: <ArgO> };

Arguments对象是活动对象的一个属性，它包括如下属性：

callee — 指向当前函数的引用

length — 真正传递的参数个数

properties-indexes (字符串类型的整数) 属性的值就是函数的参数值(按参数列表从左到右排列)。 properties-indexes内部元素的个数等于arguments.length. properties-indexes 的值和实际传递进来的参数之间是共享的。

这个共享其实不是真正的共享一个内存地址，而是2个不同的内存地址，使用JavaScript引擎来保证2个值是随时一样的，当然这也有一个前提，那就是这个索引值要小于你传入的参数个数，也就是说如果你只传入2个参数，而还继续使用arguments[2]赋值的话，就会不一致，例如：

function b(x, y, a) { arguments[2] = 10; alert(a); } b(1, 2);

这时候因为没传递第三个参数a，所以赋值10以后，alert(a)的结果依然是undefined，而不是10，但如下代码弹出的结果依然是10，因为和a没有关系。

function b(x, y, a) { arguments[2] = 10; alert(arguments[2]); } b(1, 2);

题目五：

function a() {

alert(this);

}

a.call(null);

分析：

根据ECMAScript262规范规定：如果第一个参数传入的对象调用者是null或者undefined的话，call方法将把全局对象（也就是window）作为this的值。所以，不管你什么时候传入null，其this都是全局对象window，所以该题目可以理解成如下代码：

function a() { alert(this); } a.call(window);

所以答案是object window。

纯CSS实现表格奇偶行变色：

tr:nth-child(odd) {

background-color: gray;

}

tr:nth-child(even) {

background-color: gray;

}

看一个简单的例子：

var element = document.getElementById('element');

var user = {

firstname: 'Wilson',

greeting: function(){

alert('My name is ' + this.firstname);

}

};

// Attach user.greeting as a callback

element.addEventListener('click', user.greeting);

// alert => 'My name is undefined'

这里我们期待输出My name is Wilson，但实际输出的却是My name is undefined。因为我们传递greeting函数的时候，传递的是一个函数的引用，user相应的上下文并没有传递过去。运行的时候，这个回调函数实际上是在element的上下文中执行了，this指向的是element而不是user。

两种方式避免这种问题：

第一种方法，我们可以在一个匿名函数内部调用user.greeting()方法，从而获得正确的函数执行上下文(user)。

element.addEventListener('click', function() {

user.greeting();

// alert => 'My name is Wilson'

});

上一种方式并不是非常好，因为我们不能获得回调函数的句柄以便后面通过.removeEventListener()移除事件监听。另外，这种方式也比较丑陋。

另一种方法是使用.bind()方法(做为ECMAScript 5的标准内建在所有的函数对象中）来生成一个新的函数（被绑定过的函数），这个函数会在指定的上下文中被执行。然后我们将这个被绑定过的函数作为参数传给.addEventListener()的回调函数。

// Overwrite the original function with

// one bound to the context of 'user'

user.greeting = user.greeting.bind(user);

// Attach the bound user.greeting as a callback

button.addEventListener('click', user.greeting);

与此同时，我们获得了回调函数的句柄，从而可以随时从元素上移除相应的事件监听。

button.removeEventListener('click', user.greeting);

事件捕获阶段（Capture Phase）

事件的第一个阶段是捕获阶段。事件从文档的根节点出发，随着DOM树的结构向事件的目标节点流去。途中经过各个层次的DOM节点，并在各节点上触发捕获事件，直到到达事件的目标节点。捕获阶段的主要任务是建立传播路径，在冒泡阶段，事件会通过这个路径回溯到文档跟节点。

正如文章一开始的地方提到，我们可以通过将addEventListener的第三个参数设置成true来为事件的捕获阶段添加监听回调函数。在实际应用中，我们并没有太多使用捕获阶段监听的用例，但是通过在捕获阶段对事件的处理，我们可以阻止类似clicks事件在某个特定元素上被触发。

**JavaScript面向对象**

**1.对象**

对象是一种基本的数据结构，是一些键值对的集合。键只能是string类型，值可以是任意类型。对象还有很多其他叫法：散列（hash）、散列表（hashtable）、字典（dictionary），关联数组（associative arry）。

除字符串、数字、true、false、undefined和null之外，JavaScript中的值全部都是对象。

对象是可变的，我们通过引用而非值来操作对象。

**创建对象的三种方法**

对象直接量：var person = {"name":"Jim","age":18};

new运算符：var o = new Object();

Object.create()：var o = Object.create({"name":"Jim","age":18});

**查询和设置对象属性的两种方法：(.)和([])**

var name = person.name;

var age = person["age"];

通过"."操作符获取对象的属性,必须得知道属性的名字。一般来说"[]"操作符获取对象属性的功能更强大一些,可以在[]中放入一些表达式来取属性的值，比如可以用在循环控制语句中,而"."操作符则没有这种灵活性。

查找不存在的属性并不会报错，而是返回undefined；但是查找不存在的对象的属性会报错。给null和undefined设置属性也会抛出类型错误。严格模式中任何失败的属性设置操作都会抛出一个类型错误异常。

delete操作符可以删除对象的某个属性，delete只能删除自有属性，不能删除继承属性；in操作符可以判断对象中是否存在某个属性；枚举属性用for...in。

**属性的特性**

可写（writable）：表明是否可以设置该属性的值

可枚举（enumerable）：表明是否可以用for...in来循环返回该属性

可配置（configurable）：表明该属性是否可以删除或修改

**检测属性**

in//属性

isPropertyOf()

hasOwnProperty()//自有属性

propertyIsEnumerable()//自有属性且可枚举

**属性的getter和setter**

在ECMAScript5中，属性值可以用一个或两个方法代替，这两个方法就是getter和setter。由这两个方法定义的属性称作“**存取器属性**”，它不同于数据属性，数据属性只有一个简单的值。

JavaScript调用getter方法查询存储器属性的值，调用setter方法设置存储器属性的值。当属性具有getter/setter方法时，它是一个可读/写属性；只有getter方法，它是一个只读属性；只有setter，它是一个只写属性。

存取器属性定义为一个或两个**与属性同名**的函数，这个函数定义没有使用function关键字，而是使用get/set。属性名和函数体之间也没有冒号隔开。

var oo = {

name : 'Jim',

**get sex(){**

**return 'man';**

**}**

};

oo.sex = 'woman';//sex为只读

console.log(oo.sex); //结果依然是man

**属性的特性**

属性包括一个名字和四个特性。数据属性的四个特性分别是它的值、可写性、可枚举性和可配置性。存取器属性分别是读取（get）、写入（set）、可枚举性和可配置性。

Object.getOwnPropertyDescriptor()方法可以得到对象自有属性的描述符。

**序列化与反序列化**

var o = {

"name" : "Lucy",

"gender": "girl",

"score" : {

"math" : 90,

"chinese": 88

}

}

var s = JSON.stringify(o); //对象-->字符串

var p = JSON.parse(s); //字符串-->对象

console.log(s);

console.log(p);

**对象的constructor属性**  
每一个javascript对象都有一个constructor属性。这个属性对应了对象初始化时的构造函数(函数也是对象)。

var date = new Date();

alert(date.constructor); //Date

alert(date.constructor == "Date"); //false

alert(date.constructor == Date); //true

**2.数组**

对象是无序数据的集合，而数组则是有序数据的集合，数组中的数据(元素)通过索引(从0开始)来访问，数组中的数据可以是任何的数据类型.数组本身仍旧是对象，但是由于数组的很多特性，通常情况下把数组和对象区别开来分别对待。

**创建数组**

数组直接量：var a = [1,2,3];

构造函数Array()：var a = new Array(10);

可以用delete操作符删除数组的元素，注意这个删除仅仅是将数组在该位置的元素设为undefined，数组的长度并没有改变。如果从数组中删除一个元素，它就会变成稀疏数组。我们已经使用过了数组的length属性，length属性是一个可以读/写的属性，也就是说我们可以通过改变数组的length属性来任意的改变数组的长度。如果将length设为小于数组长度的值，则原数组中索引大于length-1的值都会被删除。如果length的值大于原始数组的长度，则在它们之间的值设为undefined。

**数组方法**

Array.join()：将数组中的元素转化为字符串拼接在一起并返回。

Array.reverse()：将数组中元素颠倒顺序，返回逆序的数组。

Array.sort()：将数组中的元素排序并返回排序后的数组。可以接受一个参数。不带参数时以字母表顺序排序。

举个栗子：

var a = [111,33,4,222];

a.sort();//按字母表先后，返回[111,222,33,4]

a.sort(function(a,b){//返回[4,33,111,222]

return a-b;

});

a.sort(function(a,b){//数值大小相反的顺序

return b-a;

});

Array.concat()：返回一个新的数组，包含调用concat()的原始数组的元素和concat()的参数。

举个栗子：

var a = [111,33,4,222];

a.concat([1,2],3,[4,[5,6]]);//返回[111,33,4,222,1,2,3,4,[5,6]]

Array.slice()：返回指定数组的一个片段或子数组。

a.slice(2,-1);//返回[4],其中-n指定了倒数第n个元素。

Array.splice(a,b,n1,n2,...);

表示从第a个元素开始，删除b个元素，然后在这个地方插入n1,n1,...

举个栗子：

var a = [1,2,3,4,5,6];

var b = a.splice(4);

console.log(a);

console.log(b);

var c = a.splice(2,2);

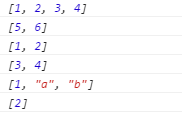
console.log(a);

console.log(c);

var d = a.splice(1,1,'a','b');

console.log(a);

console.log(d);



Array.push()/pop()/shift()/unshift()在数组的头部/尾部添加/删除元素并返回数组长度。

**3.函数**

定义函数可以用函数声明和函数表达式。

函数声明语句被提前到外部脚本或外部函数作用域的顶部，以这种方式声明的函数，可以被在它之前出现的代码调用；函数表达式定义函数之前必须把它赋值给一个变量，变量声明提前了，但给变量赋值是不会提前的都是，所以以表达式方式定义的函数在定义之前无法调用。

**函数调用**

作为函数

作为方法

作为构造函数

通过call和apply方法间接调用

作为函数调用时，ECMAScript5非严格模式和ECMAScript3中，this指向全局对象，在ECMAScript5严格模式中，this为undefined。

作为方法调用时，this指向调用它的对象。

和变量不同，关键字this没有作用域的限制，嵌套的函数不会从调用它的函数中继承this。嵌套函数作为方法调用时，this指向调用它的对象。this作为函数调用时，this不是指向全局对象（非严格模式）就是指向undefined（严格模式）。

var o = {

name:"hehe",

print:function(){

console.log(this.name);

function print1(){

console.log(this);

}

print1();//这里打出来的是Window对象

}

};

o.print();

如果传给函数的参数个数不够，则从最左边起依次对应，其余的用undefined赋值，如果传给函数的参数多于函数定义参数的个数,则多出的参数被忽略。

**callee和caller属性**

caller返回一个函数的引用，这个函数调用了当前的函数;callee放回正在执行的函数本身的引用，它是arguments的一个属性。

1. **var** a = **function**() {
2. alert(a.caller);
3. }
4. **var** b = **function**() {
5. a();
6. }
7. b();

上面的代码中，b调用了a，那么a.caller返回的是b的引用，结果如下:

1. **var** b = **function**() {
2. a();
3. }

如果直接调用a(即a在任何函数中被调用，也就是顶层调用),返回null:

1. **var** a = **function**() {
2. alert(a.caller);
3. }
4. **var** b = **function**() {
5. a();
6. }
7. //b();
8. a();

输出为null

使用callee时要注意:  
1 这个属性只有在函数执行时才有效  
2 它有一个length属性，可以用来获得形参的个数，因此可以用来比较形参和实参个数是否一致，即比较arguments.length是否等于arguments.callee.length。

3 它可以用来递归匿名函数

1. **var** a = **function**() {
2. alert(arguments.callee);
3. }
4. **var** b = **function**() {
5. a();
6. }
7. b();

a在b中被调用，但是它返回了a本身的引用，返回结果如下：

1. **var** a = **function**() {
2. alert(arguments.callee);
3. }

乱入几个**正则表达式相关方法**

String支持四种正则表达式的方法

1.**search()**：参数是一个正则表达式，返回第一个与之匹配的子串的起始位置。若参数不为正则表达式，则首先会通过RegExp构造函数将其转换为正则表达式。该方法不支持全局搜索，因为它会忽略掉正则参数中的修饰符g。

用法："JavaScript".search(/script/i);//返回4，显然默认是忽略大小写的

2.**replace()**：用以执行检索与替换工作。第一个参数是正则表达式，第二个参数是要进行替换的字符串。

用法："this is a ishehe".replace(/is/g,"IS");//返回："thIS IS a IShehe"

高级用法：name = "Doe, John"; name.replace(/(\w+)\s\*, \s\*(\w+)/, "$2 $1"); //返回John Doe

原理：正则表达式中用圆括号括起来的子表达式是有索引编号的，正则表达式会记忆与每个子表达式匹配的文本。如果替换字符中出现了$加数字，replace将会用与指定的子表达式相匹配的文本来替换这两个表达式。

3.**match()**：只有一个参数，是一个正则表达式。返回一个由匹配结果匹配的数组。如果该正则表达式设置了修饰符g，该方法返回的数组中包含字符串中的所有匹配结果。

用法："1 plus 2 is 3".match(/\d+/g);//返回['1','2','3']

如果没有修饰符g，则只检索第一个匹配，返回的也是一个数组。数组的第一个元素是匹配的字符串，余下的元素则是与正则表达式中用圆括号括起来的子表达式相匹配的字符串。

例子：解析一个URL

var url = /(\w+):\/\/([\w.]+)\/(\S\*)/;

var text = "this is a url http://www.hehe.com/~liu";

var result = text.match(url);

if(result != null){

var fullurl = result[0];// http://www.hehe.com/~liu

var protocol = result[1];// http

var host = result[2];// www.hehe.com

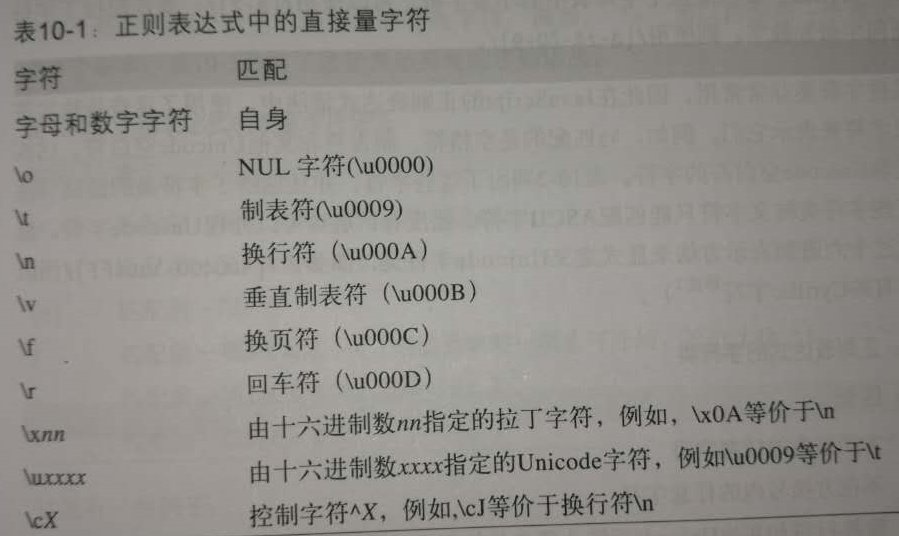
var path = result[3];// ~liu

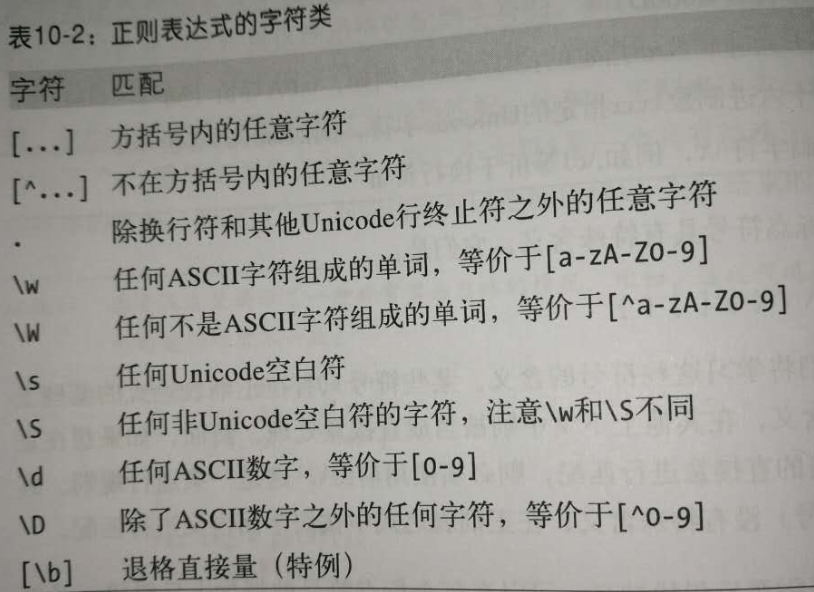
}

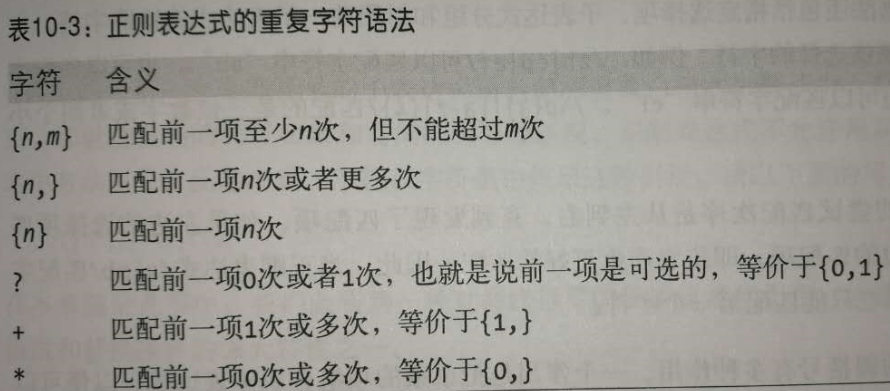
4.**split()**：将调用它的字符串拆分为一个子串组成的数组，使用的分隔符是split()的参数。

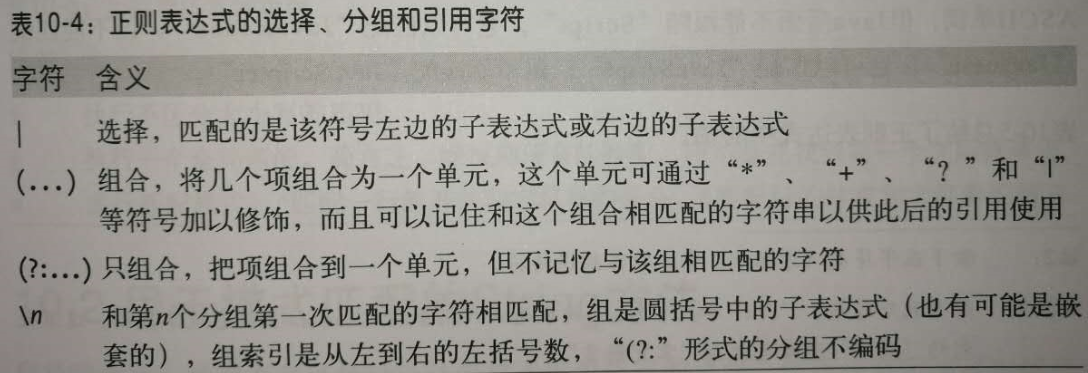
用法："123,234,345,234".split(",");//返回["123","234","345","234"]

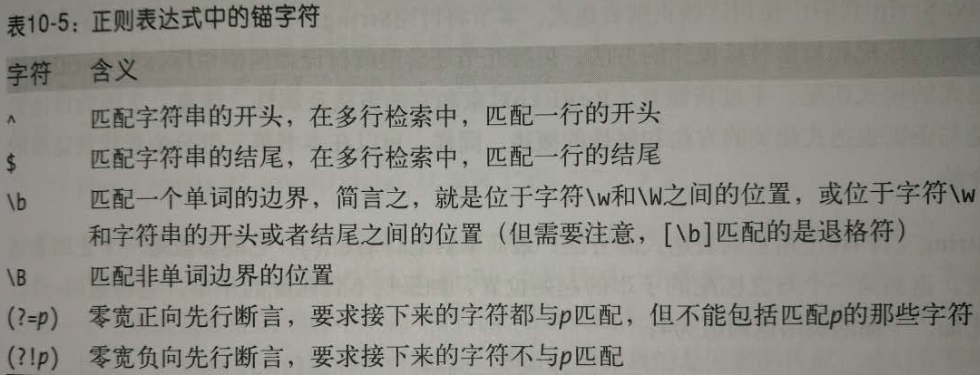
"1,2,3,4,5".split(/\s\*,\s\*/);//返回["1","2","3","4","5"]。,前后的\s分别表示,前后的空白

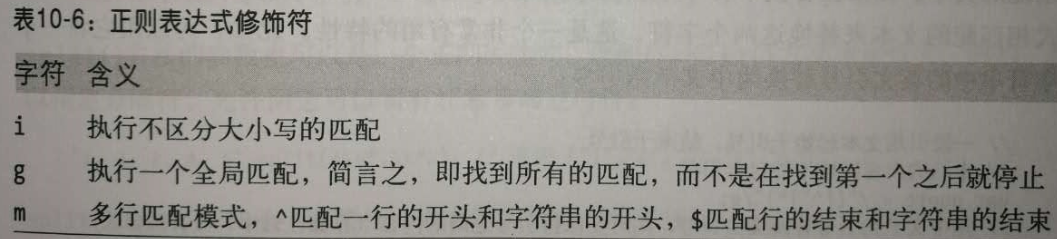












**原生js操作cookie**

cookie是浏览器提供的一种机制，它将document 对象的cookie属性提供给JavaScript。可以由JavaScript对其进行控制，而并不是JavaScript本身的性质。cookie是存于用户硬盘的一个文件，这个文件通常对应于一个域名，当浏览器再次访问这个域名时，便使这个cookie可用。因此，cookie可以跨越一个域名下的多个网页，但不能跨越多个域名使用。

不同的浏览器对cookie的实现也不一样，但其性质是相同的。例如在Windows 2000以及Windows xp中，cookie文件存储于documents and settings\userName\cookie\文件夹下。通常的命名格式为：userName@domain.txt。

cookie机制将信息存储于用户硬盘，因此可以作为全局变量，这是它最大的一个优点。它可以用于以下几种场合。

（1）保存用户登录状态。例如将用户id存储于一个cookie内，这样当用户下次访问该页面时就不需要重新登录了，现在很多论坛和社区都提供这样的功能。cookie还可以设置过期时间，当超过时间期限后，cookie就会自动消失。因此，系统往往可以提示用户保持登录状态的时间：常见选项有一个月、三个 月、一年等。

（2）跟踪用户行为。例如一个天气预报网站，能够根据用户选择的地区显示当地的天气情况。如果每次都需要选择所在地是烦琐的，当利用了 cookie后就会显得很人性化了，系统能够记住上一次访问的地区，当下次再打开该页面时，它就会自动显示上次用户所在地区的天气情况。因为一切都是在后台完成，所以这样的页面就像为某个用户所定制的一样，使用起来非常方便。

（3）定制页面。如果网站提供了换肤或更换布局的功能，那么可以使用cookie来记录用户的选项，例如：背景色、分辨率等。当用户下次访问时，仍然可以保存上一次访问的界面风格。

（4）创建购物车。正如在前面的例子中使用cookie来记录用户需要购买的商品一样，在结账的时候可以统一提交。例如淘宝网就使用cookie记录了用户曾经浏览过的商品，方便随时进行比较。

当然，上述应用仅仅是cookie能完成的部分应用，还有更多的功能需要全局变量。cookie的缺点主要集中于安全性和隐私保护。主要包括以下几种：

（1）cookie可能被禁用。当用户非常注重个人隐私保护时，他很可能禁用浏览器的cookie功能；

（2）cookie是与浏览器相关的。这意味着即使访问的是同一个页面，不同浏览器之间所保存的cookie也是不能互相访问的；

（3）cookie可能被删除。因为每个cookie都是硬盘上的一个文件，因此很有可能被用户删除；

（4）cookie安全性不够高。所有的cookie都是以纯文本的形式记录于文件中，因此如果要保存用户名密码等信息时，最好事先经过加密处理。

设置cookie

每个cookie都是一个名/值对，可以把下面这样一个字符串赋值给document.cookie：

document.cookie="userId=828";

如果要一次存储多个名/值对，可以使用分号加空格（; ）隔开，例如：

document.cookie="userId=828; userName=hulk";

在cookie 的名或值中不能使用分号（;）、逗号（,）、等号（=）以及空格。在cookie的名中做到这点很容易，但要保存的值是不确定的。如何来存储这些值呢？方法是escape()函数进行编码，它能将一些特殊符号使用十六进制表示，例如空格将会编码为“20%”，从而可以存储于cookie值中，而且使用此种方案还可以避免中文乱码的出现。例如：

document.cookie="str="+escape("I love ajax");

相当于：

document.cookie="str=I%20love%20ajax";

当使用escape()编码后，在取出值以后需要使用unescape()进行解码才能得到原来的cookie值，这在前面已经介绍过。

尽管document.cookie看上去就像一个属性，可以赋不同的值。但它和一般的属性不一样，改变它的赋值并不意味着丢失原来的值，例如连续执行下面两条语句：

代码如下:

document.cookie="userId=828";

document.cookie="userName=hulk";

这时浏览器将维护两个cookie，分别是userId和userName，因此给document.cookie赋值更像执行类似这样的语句：

代码如下:

document.addCookie("userId=828");

document.addCookie("userName=hulk");

事实上，浏览器就是按照这样的方式来设置cookie的，如果要改变一个cookie的值，只需重新赋值，例如：

document.cookie="userId=929";

这样就将名为userId的cookie值设置为了929。

获取cookie的值

下面介绍如何获取cookie的值。cookie的值可以由document.cookie直接获得：

var strCookie=document.cookie;

这将获得以分号隔开的多个名/值对所组成的字符串，这些名/值对包括了该域名下的所有cookie。例如：

代码如下:

<script language="JavaScript" type="text/javascript">

<!--

document.cookie="userId=828";

document.cookie="userName=hulk";

var strCookie=document.cookie;

alert(strCookie);

//-->

</script>

从输出可知，只能够一次获取所有的cookie值，而不能指定cookie名称来获得指定的值，这正是处理cookie值最麻烦的一部分。用户必须自己分析这个字符串，来获取指定的cookie值，例如，要获取userId的值，可以这样实现：

代码如下:

<script language="JavaScript" type="text/javascript">

<!--

//设置两个cookie

document.cookie="userId=828";

document.cookie="userName=hulk";

//获取cookie字符串

var strCookie=document.cookie;

//将多cookie切割为多个名/值对

var arrCookie=strCookie.split("; ");

var userId;

//遍历cookie数组，处理每个cookie对

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

var arr=arrCookie[i].split("=");

//找到名称为userId的cookie，并返回它的值

if("userId"==arr[0]){

userId=arr[1];

break;

}

}

alert(userId);

//-->

</script>

这样就得到了单个cookie的值。

用类似的方法，可以获取一个或多个cookie的值，其主要的技巧仍然是字符串和数组的相关操作。

给cookie设置终止日期

到现在为止，所有的cookie都是单会话cookie，即浏览器关闭后这些cookie将会丢失，事实上这些cookie仅仅是存储在内存中，而没有建立相应的硬盘文件。

在实际开发中，cookie常常需要长期保存，例如保存用户登录的状态。这可以用下面的选项来实现：

document.cookie="userId=828; expiress=GMT\_String";

其中GMT\_String是以GMT格式表示的时间字符串，这条语句就是将userId这个cookie设置为GMT\_String表示的过期时间，超过这个时间，cookie将消失，不可访问。例如：如果要将cookie设置为10天后过期，可以这样实现：

代码如下:

<script language="JavaScript" type="text/javascript">

<!--

//获取当前时间

var date=new Date();

var expiresDays=10;

//将date设置为10天以后的时间

date.setTime(date.getTime()+expiresDays\*24\*3600\*1000);

//将userId和userName两个cookie设置为10天后过期

document.cookie="userId=828; userName=hulk; expires="+date.toGMTString();

//-->

</script>

删除cookie

为了删除一个cookie，可以将其过期时间设定为一个过去的时间，例如：

代码如下:

<script language="JavaScript" type="text/javascript">

<!--

//获取当前时间

var date=new Date();

//将date设置为过去的时间

date.setTime(date.getTime()-10000);

//将userId这个cookie删除

document.cookie="userId=828; expires="+date.toGMTString();

//-->

</script>

指定可访问cookie的路径

默认情况下，如果在某个页面创建了一个cookie，那么该页面所在目录中的其他页面也可以访问该cookie。如果这个目录下还有子目录，则在子目录中也可以访问。例如在www.xxxx.com/html/a.html中所创建的cookie，可以被www.xxxx.com/html/b.html或www.xxx.com/ html/ some/c.html所访问，但不能被www.xxxx.com/d.html访问。

为了控制cookie可以访问的目录，需要使用path参数设置cookie，语法如下：

document.cookie="name=value; path=cookieDir";

其中cookieDir表示可访问cookie的目录。例如：

document.cookie="userId=320; path=/shop";

就表示当前cookie仅能在shop目录下使用。

如果要使cookie在整个网站下可用，可以将cookie\_dir指定为根目录，例如：

代码如下:

document.cookie="userId=320; path=/";

指定可访问cookie的主机名

和路径类似，主机名是指同一个域下的不同主机，例如：www.google.com和gmail.google.com就是两个不同的主机名。默认情况下，一个主机中创建的cookie在另一个主机下是不能被访问的，但可以通过domain参数来实现对其的控制，其语法格式为：

document.cookie="name=value; domain=cookieDomain";

以google为例，要实现跨主机访问，可以写为：

document.cookie="name=value;domain=.google.com";

这样，所有google.com下的主机都可以访问该cookie。

综合示例：构造通用的cookie处理函数

cookie的处理过程比较复杂，并具有一定的相似性。因此可以定义几个函数来完成cookie的通用操作，从而实现代码的复用。下面列出了常用的cookie操作及其函数实现。

1．添加一个cookie：addCookie(name,value,expiresHours)

该函数接收3个参数：cookie名称，cookie值，以及在多少小时后过期。这里约定expiresHours为0时不设定过期时间，即当浏览器关闭时cookie自动消失。该函数实现如下：

代码如下:

<script language="JavaScript" type="text/javascript">

<!--

function addCookie(name,value,expiresHours){

var cookieString=name+"="+escape(value);

//判断是否设置过期时间

if(expiresHours>0){

var date=new Date();

date.setTime(date.getTime+expiresHours\*3600\*1000);

cookieString=cookieString+"; expires="+date.toGMTString();

}

document.cookie=cookieString;

}

//-->

</script>

2．获取指定名称的cookie值：getCookie(name)

该函数返回名称为name的cookie值，如果不存在则返回空，其实现如下：

代码如下:

<script language="JavaScript" type="text/javascript">

<!--

function getCookie(name){

var strCookie=document.cookie;

var arrCookie=strCookie.split("; ");

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

var arr=arrCookie[i].split("=");

if(arr[0]==name)return arr[1];

}

return "";

}

//-->

</script>

3．删除指定名称的cookie：deleteCookie(name)

该函数可以删除指定名称的cookie，其实现如下：

代码如下:

<script language="JavaScript" type="text/javascript">

<!--

function deleteCookie(name){

var date=new Date();

date.setTime(date.getTime()-10000);

document.cookie=name+"=v; expires="+date.toGMTString();

}

//-->

</script>

**单例模式**

用于只能实例化一个对象的地方。

比如说点击一个按钮弹出一个对话框，多次点击按钮时就会弹出多个对话框，这时候就可以使用单例模式。符合单一职责原则和对象自制原则。

很多时候我们要保证类的实例只有一个。我们可能在自己的代码中判断需要的类的实例有无，无就new一个。这样看似不错。问题是，你用到这个类的地方有n个，这样你就需要n个判断。为什么不把这个职责交给类本身呢?然后让类提供一个接口访问。

在浏览BBS、SNS网站的时候，常常会看到“当前在线人数”这样的一项内容。对于这样的一项功能，我们通常的做法是把当前的在线人数存放到一个内存、文件或者数据库中，每次用户登录的时候，就会马上从内存、文件或者数据库中取出，在其基础上加1后，作为当前的在线人数进行显示，然后再把它保存回内存、文件或者数据库里，这样后续登录的用户看到的就是更新后的当前在线人数；同样的道理，当用户退出后，当前在线人数进行减1的工作。所以，对于这样的一个需求，我们按照面向对象的设计思想，可以把它抽象为“在线计数器”这样一个对象。

网站代码中凡是用到计数器的地方，只要new一个计数器对象，然后就可以获取、保存、增加或者减少在线人数的数量。不过，我们的代码实际的使用效果并不好。假如有多个用户同时登录,那么在这个时刻，通过计数器取到的在线人数是相同的，于是他们使用各自的计数器加1后存入文件或者数据库。这样操作后续登陆的用户得到的在线人数，与实际的在线人数并不一致。所以，把这个计数器设计为一个全局对象，所有人都共用同一份数据，就可以避免类似的问题，这就是我们所说的单例模式的其中的一种应用。

单例模式能够保证一个类仅有唯一的实例，并提供一个全局访问点。

我们是不是可以通过一个全局变量来实现单例模式的要求呢?我们只要仔细地想想看，全局变量确实可以提供一个全局访问点，但是它不能防止别人实例化多个对象。通过外部程序来控制的对象的产生的个数，势必会系统的增加管理成本，增大模块之间的耦合度。所以，最好的解决办法就是让类自己负责保存它的唯一实例，并且让这个类保证不会产生第二个实例，同时提供一个让外部对象访问该实例的方法。自己的事情自己办，而不是由别人代办，这非常符合面向对象的封装原则。

**简单工厂模式的优缺点分析：**

       优点：工厂类是整个模式的关键所在。它包含必要的判断逻辑，能够根据外界给定的信息，决定究竟应该创建哪个具体类的对象。用户在使用时可以直接根据工厂类去创建所需的实例，而无需了解这些对象是如何创建以及如何组织的。有利于整个软件体系结构的优化。

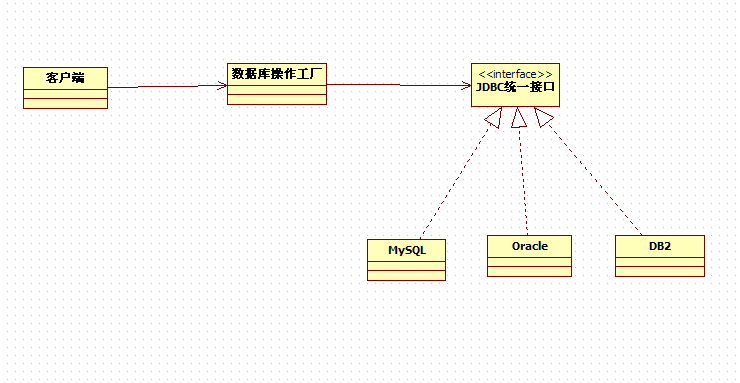
      缺点：由于工厂类集中了所有实例的创建逻辑，这就直接导致一旦这个工厂出了问题，所有的客户端都会受到牵连；而且由于简单工厂模式的产品室基于一个共同的抽象类或者接口，这样一来，但产品的种类增加的时候，即有不同的产品接口或者抽象类的时候，工厂类就需要判断何时创建何种种类的产品，这就和创建何种种类产品的产品相互混淆在了一起，违背了单一职责，导致系统丧失灵活性和可维护性。而且更重要的是，简单工厂模式违背了“开放封闭原则”，就是违背了“系统对扩展开放，对修改关闭”的原则，因为当我新增加一个产品的时候必须修改工厂类，相应的工厂类就需要重新编译一遍。

      总结一下：简单工厂模式分离产品的创建者和消费者，有利于软件系统结构的优化；但是由于一切逻辑都集中在一个工厂类中，导致了没有很高的内聚性，同时也违背了“开放封闭原则”。另外，简单工厂模式的方法一般都是静态的，而静态工厂方法是无法让子类继承的，因此，简单工厂模式无法形成基于基类的继承树结构。

**简单工厂模式的实际应用简介：**

       作为一个最基本和最简单的设计模式，简单工厂模式却有很非常广泛的应用，我们这里以Java中的JDBC操作数据库为例来说明。

        JDBC是SUN公司提供的一套数据库编程接口API，它利用Java语言提供简单、一致的方式来访问各种关系型数据库。Java程序通过JDBC可以执行SQL语句，对获取的数据进行处理，并将变化了的数据存回数据库，因此，JDBC是Java应用程序与各种关系数据进行对话的一种机制。用JDBC进行数据库访问时，要使用数据库厂商提供的驱动程序接口与数据库管理系统进行数据交互。



客户端要使用使用数据时，只需要和工厂进行交互即可，这就导致操作步骤得到极大的简化，操作步骤按照顺序依次为：注册并加载数据库驱动，一般使用Class.forName();创建与数据库的链接Connection对象；创建SQL语句对象preparedStatement(sql)；提交SQL语句，根据实际情况使用executeQuery()或者executeUpdate()；显示相应的结果；关闭数据库。

jsonp跨域（亲测有效）

Java代码



页面代码：

