**网页前端编程基础**

**HTML** 是用来描述网页的一种语言。

HTML 指的是超文本标记语言 (Hyper Text Markup Language)

HTML 不是一种编程语言，而是一种标记语言 (markup language)

标记语言是一套标记标签 (markup tag)

HTML 使用标记标签来描述网页

<html>

<body>

<h1>My First Heading</h1>

<p>My first paragraph.</p>

</body>

</html>

**HTML 标签**

HTML 标记标签通常被称为 HTML 标签 (HTML tag)。

HTML 标签是由尖括号包围的关键词，比如 <html>

HTML 标签通常是成对出现的，比如 <b> 和 </b>

标签对中的第一个标签是开始标签，第二个标签是结束标签

开始和结束标签也被称为开放标签和闭合标签

**典型块级元素与行内元素**

**块元素**(block element)

div

h（x） - 标题

p - 段落

table - 表格

ul - 非排序列表

**内联元素**(inline element)

span - 常用内联容器，定义文本内区块

a - 锚点

i - 斜体

img - 图片

input - 输入框

**行内、块状元素区别**：

(1).块级元素会独占一行，其宽度自动填满其父元素宽度，行内元素不会独占一行，相邻的行内元素会排列在同一行里，直到道一行排不下，才会换行，其宽度随元素的内容而变化

(2). 一般情况下，块级元素可以设置 width, height属性，行内元素设置width, height无效(注意：块级元素即使设置了宽度，仍然是独占一行的)

(3).块级元素可以设置margin 和 padding. 行内元素的水平方向的padding-left,padding-right,margin-left,margin-right 都产生边距效果，但是竖直方向的padding-top,padding-bottom,margin-top,margin-bottom都不会产生边距效果。（水平方向有效，竖直方向无效）

**行内元素和块状元素的说明**

根据CSS规范的规定，每一个网页元素都有一个display属性，用于确定该元素的类型，每一个元素都有默认的display属性值，比如div元素，它的默认display属性值为“block”，成为“块级”元素(block-level)；而span元素的默认display属性值为“inline”，称为“行内”元素。div这样的块级元素，就会自动占据一定矩形空间，可以通过设置高度、宽度、内外边距等属性，来调整的这个矩形的样子；与之相反，像“span”“a”这样的行内元素，则没有自己的独立空间，它是依附于其他块级元素存在的，因此，对行内元素设置高度、宽度、内外边距等属性，都是无效的。

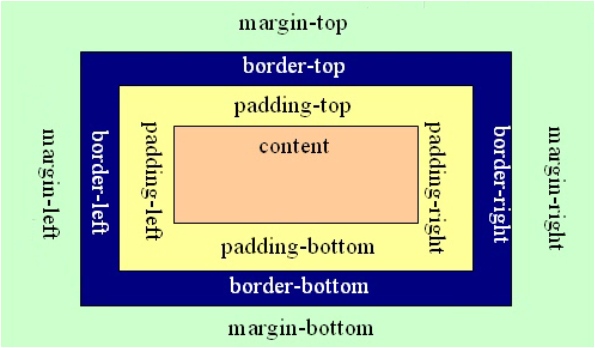
**CSS（层叠样式表）**

层叠样式表(英文全称：Cascading Style Sheets)是一种用来表现HTML（标准通用标记语言的一个应用）或XML（标准通用标记语言的一个子集）等文件样式的计算机语言。CSS不仅可以静态地修饰网页，还可以配合各种脚本语言动态地对网页各元素进行格式化。

h1 {color:red; font-size:14px;}

**CSS盒子模型**

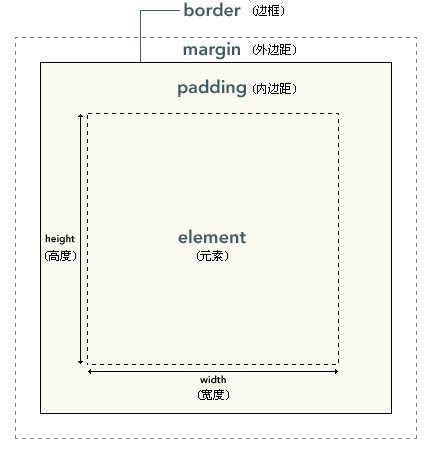
CSS中， Box Model叫盒子模型（或框模型），Box Model规定了元素框处理元素内容（element content）、内边距（padding）、边框（border） 和 外边距（margin） 的方式。在HTML文档中，每个元素（element）都有盒子模型，所以说在Web世界里（特别是页面布局），Box Model无处不在。下面是Box Model的图示：



说明：上图中，由内而外依次是元素内容（content）、内边矩（padding-top、padding-right、padding- bottom、padding-left）、边框（border-top、border-right、border-bottom、border- left）和外边距（marging-top、margin-right、margin-bottom、margin-left）。

内边距、边框和外边距可以应用于一个元素的所有边，也可以应用于单独的边。而且，外边距可以是负值，而且在很多情况下都要使用负值的外边距。

下图是W3School的Box Model 图解：



**Javascript**

JavaScript一种直译式脚本语言，是一种动态类型、弱类型的语言。它的解释器被称为JavaScript引擎，为浏览器的一部分，广泛用于客户端的脚本语言，最早是在HTML网页上使用，用来给HTML网页增加动态功能。

在1995年时，由Netscape公司的Brendan Eich，在网景导航者浏览器上首次设计实现而成。因为Netscape与Sun合作，Netscape管理层希望它外观看起来像Java，因此取名为JavaScript。但实际上它的语法风格与Self及Scheme较为接近。

为了取得技术优势，微软推出了JScript，CEnvi推出ScriptEase，与JavaScript同样可在浏览器上运行。为了统一规格，因为JavaScript兼容于ECMA标准，因此也称为ECMAScript。

直译语言（Interpreted language），又称直译式语言，是一种编程语言，它不需要经过编译器先行编译为机器码，之后直接在CPU中执行，相反的，这种编程语言需要通过直译器，在执行期动态直译（interpreted）

**JavaScript 数据类型**

JavaScript 拥有动态类型。这意味着相同的变量可用作不同的类型：

实例

var x // x 为 undefined

var x = 6; // x 为数字

var x = "Bill"; // x 为字符串

**字符串**

字符串是存储字符（比如 "Bill Gates"）的变量。

字符串可以是引号中的任意文本。您可以使用单引号或双引号：

实例

var carname="Bill Gates";

var carname='Bill Gates';

**数字**

JavaScript 只有一种数字类型。数字可以带小数点，也可以不带：

实例

var x1=34.00; //使用小数点来写

var x2=34; //不使用小数点来写

**布尔**

布尔（逻辑）只能有两个值：true 或 false。

var x=true

var y=false

**数组**

下面的代码创建名为 cars 的数组：

var cars=new Array();

cars[0]="Audi";

cars[1]="BMW";

cars [2] ="Volvo";

或者 (condensed array):

vary cars=new Array("Audi","BMW","Volvo");

或者 (literal array):

实例

var cars=["Audi","BMW","Volvo"];

**对象**

对象由花括号分隔。在括号内部，对象的属性以名称和值对的形式 (name : value) 来定义。属性由逗号分隔：

var person={firstname:"Bill", lastname:"Gates", id:5566};

对象属性有两种寻址方式：

实例

name=person.lastname;

name=person["lastname"];

**Undefined 和 Null**

Undefined 这个值表示变量不含有值。

可以通过将变量的值设置为 null 来清空变量。

**typeof**

可以使用typeof操作符检测变量的数据类型

|  |  |
| --- | --- |
| 类型 | 结果 |
| Undefined | "undefined" |
| Null | "object" |
| Boolean | "boolean" |
| Number | "number" |
| String | "string" |
| 函数对象 ( [[Call]] 在ECMA-262条款中实现了) | "function" |
| 任何其他对象 | "object" |

JavaScript中number、string、boolean、null和undefined型数据都是值类型，由于值类型数据占据的空间都是固定的，所以可以把它们存储在狭窄的内存栈。object、function和array等对象都是引用型数据

**创建对象的方式**

**Object**

**var *p*** = **new** Object();  
***p***.**name** = **"jack"**;  
***p***.**age** = 20;  
***p***.sayName = **function**(){  
 alert(**this**.**name**);  
}

**字面量**

**var *p*** = {**name**:**"jack"**,**age**:20,sayName:**function**(){alert(**this**.**name**);}};

**工厂模式**

**function** *createPerson*(name,age){  
 **var** o = **new** Object();  
 o.**name** = name;  
 o.**age** = age;  
 o.sayName = **function**(){  
 **console**.log(**this**.**name**);  
 }  
 **return** o;  
}  
  
**var *c*** = *createPerson*(**"jason"**,30);

**构造函数**

**function** *Person*(name,age){  
 **this**.**name** = name;  
 **this**.**age** = age;  
 **this**.sayName = **function**(){  
 alert(**this**.**name**);  
 }  
}  
  
**var *p*** = **new** *Person*(**"jack"**,20);

alert(***p* instanceof** *Person*);

构造函数的不同之处：

没有显示的创建对象

直接将属性和方法赋给了this对象

没有return语句

构造函数的问题：

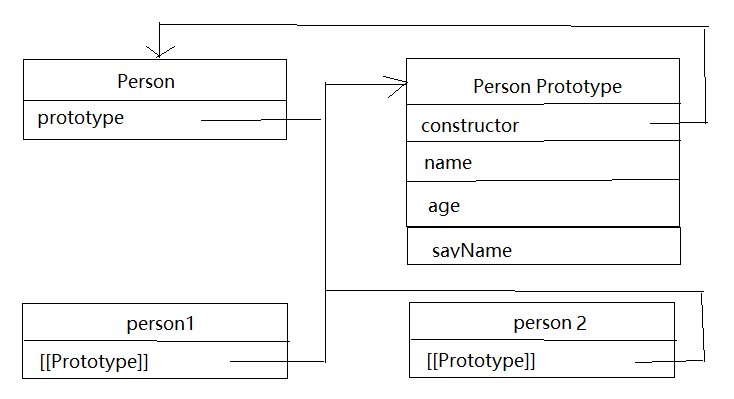
每个函数都在每个实例上重新创建一遍，person1和person2都有一个名为sayName的函数，但是两个sayName函数是两个不同的Function实例

**原型模式**

**function** *Person*(){  
  
}  
  
*Person*.**prototype**.**name** = **"jack"**;  
*Person*.**prototype**.**age** = 20;  
*Person*.**prototype**.sayName = **function**(){  
 alert(**this**.**name**);  
}  
  
**var *p*** = **new** *Person*();  
***p***.sayName();

**理解原型对象prototype**

我们创建的每个函数都有一个prototype属性，这个属性是一个指针，指向一个对象，而这个对象的用途是包含可以由特定类型的所有实例共享的属性和方法。

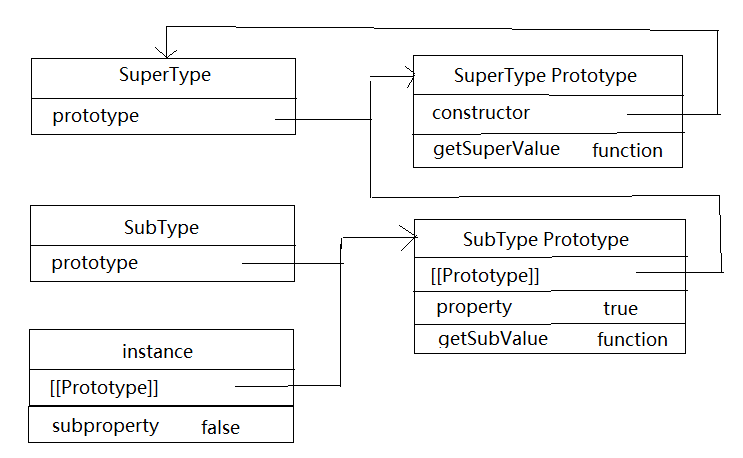


alert(*Person*.**prototype**.isPrototypeOf(***p***));  
alert(Object.getPrototypeOf(***p***) == *Person*.**prototype**);

**如何实现继承**

Javascript实现继承主要依靠原型链

**原型链**



上面的代码中，我们没有使用SubType默认提供的原型，而是给它换了一个新原型，这个新原型就是SuperType的实例。于是，新原型不仅具有作为一个SuperType的实例所有拥有的全部属性和方法，而且其内部还有一个指针，指向了SuperType的原型。

**原型链的问题**

当原型包含引用类型的属性时，这些属性会被所有实例共享

**function** *SuperType*(){  
 **this**.**colors** = [**"red"**,**"green"**,**"blue"**];  
}  
  
**function** *SubType*(){  
}  
*//继承了SuperType  
SubType*.**prototype** = **new** *SuperType*();  
  
**var *instance1*** = **new** *SubType*();  
***instance1***.**colors**.push(**"white"**);  
alert(***instance1***.**colors**);  
  
**var *instance2*** = **new** *SubType*();  
alert(***instance2***.**colors**);

**借用构造函数**

在子类型构造函数内部调用父类型构造函数

call函数

语法：call([thisObj[,arg1[, arg2[, [,.argN]]]]])

参数：thisObj 可选项。将被用作当前对象的对象。 arg1, arg2, , argN 可选项。将被传递方法参数序列。

说明：call 方法可以用来代替另一个对象调用一个方法。call 方法可将一个函数的对象上下文从初始的上下文改变为由 thisObj 指定的新对象。

**function** *SuperType*(){  
 **this**.**colors** = [**"red"**,**"green"**,**"blue"**];  
}  
  
**function** *SubType*(){  
 *//继承了SuperType  
 SuperType*.call(**this**);  
}  
  
**var *instance1*** = **new** *SubType*();  
***instance1***.**colors**.push(**"white"**);  
  
alert(***instance1***.**colors**);  
  
**var *instance2*** = **new** *SubType*();  
alert(***instance2***.**colors**);

借用构造函数的问题：

方法都在构造函数中定义，函数复用是个问题，而且在父类原型中定义的函数，对于子类也是不可见的。

**组合继承**

**function** *SuperType*(name){  
 **this**.**name** = name;  
 **this**.**colors** = [**"red"**, **"green"**, **"blue"**];  
}  
  
*SuperType*.**prototype**.sayName = **function**(){  
 alert(**this**.**name**);  
}  
  
**function** *SubType*(name,age){  
 *//继承属性  
 SuperType*.call(**this**,name);  
 **this**.**age** = age;  
}  
  
*//继承函数  
SubType*.**prototype** = **new** *SuperType*();  
*SubType*.**prototype**.constructor = *SubType*;  
*SubType*.**prototype**.sayAge = **function**(){  
 alert(**this**.**age**);  
}  
  
**var *instance1*** = **new** *SubType*(**"Jack"**,20);  
***instance1***.**colors**.push(**"white"**);  
alert(***instance1***.**colors**);  
***instance1***.sayName();  
***instance1***.sayAge();  
  
**var *instance2*** = **new** *SubType*(**"Rose"**,22);  
alert(***instance2***.**colors**);

**闭包**

**变量的作用域**

根据作用域划分，变量有两种类型：全局变量和局部变量。

在函数内部访问全局变量：

**var *n***=100;  
**function** *f1*(){  
 alert(***n***);  
}  
*f1*(); *// 100*

函数外部自然无法读取函数内的局部变量

**function** *f1*(){  
 **var** n=100;  
}  
alert(n); *// error*

如何从外部读取局部变量？

出于种种原因，我们有时候需要得到函数内的局部变量。但是，前面已经说过了，正常情况下，这是办不到的，只有通过变通方法才能实现。

那就是在函数的内部，再定义一个函数。

**function** *f1*(){  
 **var** n=100;  
 **function** *f2*(){  
 alert(n); *// 100* }  
}

在上面的代码中，函数f2就被包括在函数f1内部，这时f1内部的所有局部变量，对f2都是可见的。但是反过来就不行，f2内部的局部变量，对f1就是不可见的。这就是Javascript语言特有的"链式作用域"结构（chain scope），子对象会一级一级地向上寻找所有父对象的变量。所以，父对象的所有变量，对子对象都是可见的，反之则不成立。

既然f2可以读取f1中的局部变量，那么只要把f2作为返回值，我们不就可以在f1外部读取它的内部变量了吗？

**function** *f1*(){  
 **var** n=100;  
 **function** *f2*(){  
 alert(n);  
 }  
 **return** f2;  
}  
**var *result***=*f1*();  
***result***(); *// 100*

**闭包的概念**

上面代码中的f2函数，就是闭包。

闭包，指有权访问另一个函数作用域中的变量的函数，在本质上，闭包就是将函数内部和函数外部连接起来的一座桥梁。

**闭包的用途**

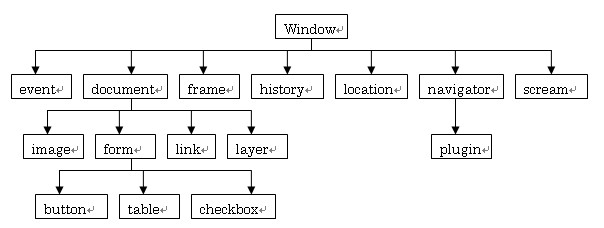
闭包可以用在许多地方。它的最大用处有两个，一个是前面提到的可以读取函数内部的变量，另一个就是让这些变量的值始终保持在内存中。

**function** *f1*(){  
 **var** n=999;  
 nAdd=**function**(){n+=1}  
 **function** *f2*(){  
 alert(n);  
 }  
 **return** f2;  
}  
**var *result***=*f1*();  
***result***(); *// 999*nAdd();  
***result***(); *// 1000*

**DOM编程**

DOM就是HTML页面的模型，将每个标签都做为一个对象，JavaScript通过调用DOM中的属性、方法就可以对网页中的文本框、层等元素进行编程控制。比如通过操作文本框的DOM对象，就可以读取文本框中的值、设置文本框中的值。

HTML DOM定义了访问和操作HTML文档的标准方法。他把HTML文档呈现为带有元素、属性和文本的树结构。在层次图中，每个对象是它的父对象的属性，如Window对象是Document对象的父对象，所以在引用Document对象就可以使用Window.Document，相当于document是window对象的属性。对于每一个页面，浏览器都会自动创建Window对象、Document对象、Location对象、Navigator对象、History对象。



**window**.onload = **function**(){  
 **var** child = **document**.createElement(**"span"**);  
 child.**innerHTML** = **"<i>jason</i>"**;  
 **var** node = **document**.getElementById(**'wrapper'**);  
 node.appendChild(child);  
}