## 创建新的 HTML 元素

如需向 HTML DOM 添加新元素，您必须首先创建该元素（元素节点），然后向一个已存在的元素追加该元素。

### 实例

<div id="div1">

<p id="p1">这是一个段落</p>

<p id="p2">这是另一个段落</p>

</div>

<script>

var para=document.createElement("p");

var node=document.createTextNode("这是新段落。");

para.appendChild(node);

var element=document.getElementById("div1");

element.appendChild(para);

</script>

Js 中 require() 的用法：require.js的诞生是为了解决两大问题，第一实现js文件的异步加载，避免网页失去响应，第二管理模块之间的依赖性，便于代码的编写和维护

**一、为什么要用require.js？**

最早的时候，所有Javascript代码都写在一个文件里面，只要加载这一个文件就够了。后来，代码越来越多，一个文件不够了，必须分成多个文件，依次加载。下面的网页代码，相信很多人都见过。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <script src="1.js"></script>  　<script src="2.js"></script>  　<script src="3.js"></script>  　<script src="4.js"></script>  　<script src="5.js"></script>  　<script src="6.js"></script> |

这段代码依次加载多个js文件。

这样的写法有很大的缺点。首先，加载的时候，浏览器会停止网页渲染，加载文件越多，网页失去响应的时间就会越长；其次，由于js文件之间存在依赖关系，因此必须严格保证加载顺序（比如上例的1.js要在2.js的前面），依赖性最大的模块一定要放到最后加载，当依赖关系很复杂的时候，代码的编写和维护都会变得困难。

## 三．\_ initialized用法

function Car(sColor,iDoors,iMpg) {  
this.color = sColor;  
this.doors = iDoors;  
this.mpg = iMpg;  
this.drivers = new Array("Mike","John");  
  
if (typeof Car.\_initialized == "undefined")   
{   
Car.prototype.showColor = function() {  
alert(this.color);  
};  
  
Car.\_initialized = true; // 这句表示什么意思？  
}  
}

\_initialized 是一个标记位  
用来标记Car类对象是否初始化了

## [js中使用使用原型（prototype）定义方法的好处](http://www.cnblogs.com/yunfour/p/3946444.html)

经常在前端面试或是和其他同行沟通是，在谈到构造在JS定义构造函数的方法是最好使用原型的方式：将方法定义到构造方法的prototype上，这样的好处是，通过该构造函数生成的实例所拥有的方法都是指向一个函数的索引，这样可以节省内存。

当然，这种说法没有任何问题，只是在实现上，并非只有使用prototype的方式才能达到这样的效果，我们可以将方法以函数的形式定义在构造函数之外，然后在构造函数中通过this.method = method的方式，这样生成的实例的方法也都通过索引指向一个函数，具体如下：

[复制代码](javascript:void(0);)

// 不使用原型定义方法：

(function() {

function Constractor() {

this.method1 = method1;

this.method2 = method2;

}

function method1() {

}

function method2() {

}

})();

[复制代码](javascript:void(0);)

一般使用原型定义时代码如下：

[复制代码](javascript:void(0);)

(function () {

function Constractor() {

}

Constactor.prototype = {

method1: function() {

},

method2: function() {

}

};

// 或者

Constactor.prototype.method1 = function() {

};

Constactor.prototype.method2 = function() {

};

})();

[复制代码](javascript:void(0);)

理论和实现都没有什么高深的，只是为了达到同样的目的，可以通过不同的途径，只是此种方式在使用instanceOf运算符来判断继承关系时就不奏效了。

## JS验证方法（表格为空||邮箱格式不正确）

## 必填（或必选）项目：

<html>

<head>

<script type="text/javascript">

function validate\_required(field,alerttxt)

{

with (field)

{

if (value==null||value=="")

{alert(alerttxt);return false}

else {return true}

}

}

function validate\_form(thisform)

{

with (thisform)

{

if (validate\_required(email,"Email must be filled out!")==false)

{email.focus();return false}

}

}

</script>

</head>

<body>

<form action="submitpage.htm" onsubmit="return validate\_form(this)" method="post">

Email: <input type="text" name="email" size="30">

<input type="submit" value="Submit">

</form>

</body>

</html>

## E-mail 验证：

<html>

<head>

<script type="text/javascript">

function validate\_email(field,alerttxt)

{

with (field)

{

apos=value.indexOf("@")

dotpos=value.lastIndexOf(".")

if (apos<1||dotpos-apos<2)

{alert(alerttxt);return false}

else {return true}

}

}

function validate\_form(thisform)

{

with (thisform)

{

if (validate\_email(email,"Not a valid e-mail address!")==false)

{email.focus();return false}

}

}

</script>

</head>

<body>

<form action="submitpage.htm"onsubmit="return validate\_form(this);" method="post">

Email: <input type="text" name="email" size="30">

<input type="submit" value="Submit">

</form>

</body>

</html>