# 如何优化你的JS代码

JS代码的执行效率往往直接影响了页面的性能，有的时候，实现同样的功能，不同的JS代码往往在效率上相差很多，有的时候仅仅是由于我们的书写习惯导致的，当然在高级点的浏览器中，它们大多都已经帮我们优化了，但是在中国，万恶的IE6仍然大量的存在，我们不得不去考虑它。对于JS代码的优化，实际上有很多的情况，有些影响是比较小的，而有些是比较严重的，本文中，我把几个我认为影响比较严重的情况列出来，供大家参考。

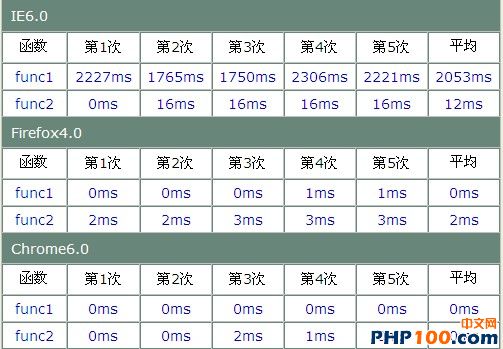
**1、字符串的拼接**

　　字符串的拼接在我们开发中会经常遇到，所以我把其放在首位，我们往往习惯的直接用+=的方式来拼接字符串，其实这种拼接的方式效率非常的低，我们可以用一种巧妙的方法来实现字符串的拼接，那就是利用数组的join方法。

<div class="one" id="one"></div>  
<input type="button" value="效率低" onclick="func1()" />  
<input type="button" value="效率高" onclick="func2()" />

//效率低的  
function func1(){  
var start = new Date().getTime();  
var template = "";  
for(var i = 0; i < 10000; i++){  
template += "<input type='button' value='a'>";  
}  
var end = new Date().getTime();  
document.getElementById("one").innerHTML = template;  
alert("用时:" + (end - start) + "毫秒");  
}  
//效率高的  
function func2(){  
var start = new Date().getTime();  
var array = [];  
for(var i = 0; i < 10000; i++){  
array[i] = "<input type='button' value='a'>";  
}  
var end = new Date().getTime();  
document.getElementById("one").innerHTML = array.join("");  
alert("用时:" + (end - start) + "毫秒");  
}

　　我们看看其在不同浏览器下执行的情况

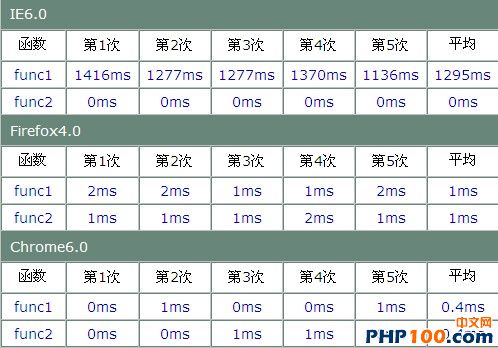


　　我们会发现，在IE6下其差别是相当明显的，其实这种情况在IE的高版本中体现的也非常明显，但是在Firefox下却没有多大的区别，相反第二种的相对效率还要低点，不过只是差别2ms左右，而Chrome也和Firefox类似。另外在这里顺便说明一下，在我们给数组添加元素的时候，很多人喜欢用数组的原生的方法push，其实直接用arr[i]或者arr[arr.length]的方式要快一点，大概在10000次循环的情况IE浏览器下会有十几毫秒的差别。

**2、for循环**

　　for循环是我们经常会遇到的情况，我们先看看下面例子：

<input type="button" value="效率低" onclick="func1()" />  
<input type="button" value="效率高" onclick="func2()" />  
var arr = [];  
for(var i = 0; i < 10000; i++){  
arr[i] = "<div>" + i + "</div>";  
}  
document.body.innerHTML += arr.join("");  
  
//效率低的  
function func1(){  
var divs = document.getElementsByTagName("div");  
var start = new Date().getTime();  
for(var i = 0; i < divs.length; i++){  
//"效率低"  
}  
var end = new Date().getTime();  
alert("用时：" + (end - start) + "毫秒");  
}  
//效率高的  
function func2(){  
var divs = document.getElementsByTagName("div");  
var start = new Date().getTime();  
for(var i = 0, len = divs.length; i < len; i++){  
//"效率高"  
}  
var end = new Date().getTime();  
alert("用时：" + (end - start) + "毫秒");  
}



　　由上表可以看出，在IE6.0下，其差别是非常明显，而在Firefox和Chrome下几乎没有差别，之所以在IE6.0下会有这种情况，主要是因为for循环在执行中，第一种情况会每次都计算一下长度，而第二种情况却是在开始的时候计算长度，并把其保存到一个变量中，所以其执行效率要高点，所以在我们使用for循环的时候，特别是需要计算长度的情况，我们应该开始将其保存到一个变量中。但是并不是只要是取长度都会出现如此明显的差别，如果我们仅仅是操作一个数组，取得的是一个数组的长度，那么其实两种方式的写法都差不多，我们看下面的例子：

<input type="button" value="效率低" onclick="func1()" />  
<input type="button" value="效率高" onclick="func2()" />  
var arr2 = [];  
for(var i = 0; i < 10000; i++){  
arr2[i] = "<div>" + i + "</div>";  
}  
//效率低的  
function func1(){  
var start = new Date().getTime();  
for(var i = 0; i < arr2.length; i++){  
//"效率低"  
}  
var end = new Date().getTime();  
alert("用时：" + (end - start) + "毫秒");  
}  
//效率高的  
function func2(){  
var start = new Date().getTime();  
for(var i = 0, len = arr2.length; i < len; i++){  
//"效率高"  
}  
var end = new Date().getTime();  
alert("用时：" + (end - start) + "毫秒");  
}



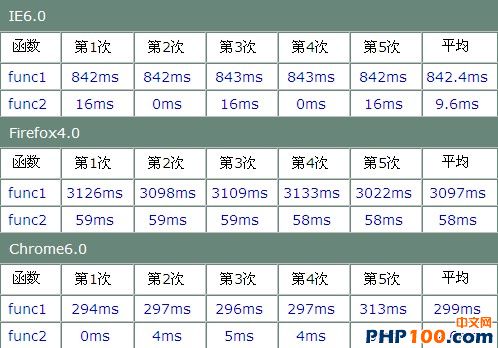
　　从上表可以看出，如果仅仅是一个数组的话，我们看到其实两种写法都是差不多的，其实如果我们把循环再上调到100000次的话，也仅仅是差别几毫秒而已，所以在数组的情况下，我认为都是一样的。对于for循环的优化，也有人提出很多点，有人认为用-=1，或者从大到小的方式循环等等，我认为是完全没有必要的，这些优化往往实际情况下根本没有表现出来，换句话说只是计算机级别的微小的变化，但是给我们带来的却是代码的可读性大大的降低，所以实在是得不偿失。

**3、减少页面的重绘**

　　减少页面重绘虽然本质不是JS本身的优化，但是其往往是由JS引起的，而重绘的情况往往是严重影响页面性能的，所以完全有必要拿出来，我们看下面例子：

<div id="demo"></div>  
<input type="button" value="效率低" onclick="func1()" />  
<input type="button" value="效率高" onclick="func2()" />  
var str = "<div>这是一个测试字符串</div><div>这是一个测试字符串</div><div>这是一个测试字符串</div><div>这是一个测试字符串</div><div>这是一个测试字符串</div><div>这是一个测试字符串</div><div>这是一个测试字符串</div><div>这是一个测试字符串</div><div>这是一个测试字符串</div><div>这是一个测试字符串</div><div>这是一个测试字符串</div><div>这是一个测试字符串</div><div>这是一个测试字符串</div>";  
//效率低的  
function func1(){  
var obj = document.getElementById("demo");  
var start = new Date().getTime();  
for(var i = 0; i < 100; i++){  
obj.innerHTML += str + i;  
}  
var end = new Date().getTime();  
alert("用时 " + (end - start) + " 毫秒");  
}  
//效率高的  
function func2(){  
var obj = document.getElementById("demo");  
var start = new Date().getTime();  
var arr = [];  
for(var i = 0; i < 100; i++){  
arr[i] = str + i;  
}  
obj.innerHTML = arr.join("");  
var end = new Date().getTime();  
alert("用时 " + (end - start) + " 毫秒");  
}

　　在例子中，我只是用了100次的循环，因为如果用10000次循环的话，浏览器基本上就卡住不动了，但是即使是100次的循环，我们看看下面的执行结果。



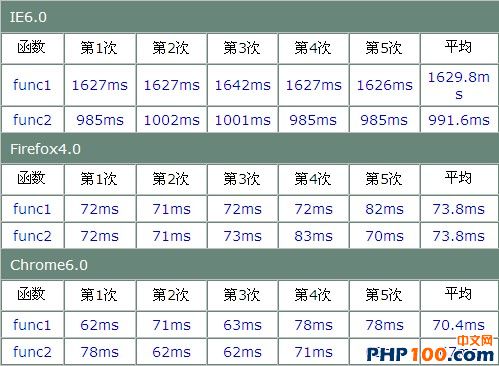
　　可以看到的是，这简直是一个惊人的结果，仅仅100次的循环，不管是在什么浏览器下，都出现了如此之大的差别，另外我们还发现，在这里，IE6的执行效率居然比起Firefox还要好很多，可见Firefox在页面重绘这方面并没有做一些的优化。这里还要注意的是，一般影响页面重绘的不仅仅是innerHTML，如果改变元素的样式，位置等情况都会触发页面重绘，所以在平时一定要注意这点。

**4、减少作用域链上的查找次数**

　　我们知道，js代码在执行的时候，如果需要访问一个变量或者一个函数的时候，它需要遍历当前执行环境的作用域链，而遍历是从这个作用域链的前端一级一级的向后遍历，直到全局执行环境，所以这里往往会出现一个情况，那就是如果我们需要经常访问全局环境的变量对象的时候，我们每次都必须在当前作用域链上一级一级的遍历，这显然是比较耗时的，我们看下面的例子：

<div id="demo"></div>  
<input id="but1" type="button" onclick="func1()" value="效率低"/>  
<input id="but2" type="button" onclick="func2()" value="效率高"/>  
  
function func1(){  
var start = new Date().getTime();  
for(var i = 0; i < 10000; i++){  
var but1 = document.getElementById("but1");  
var but2 = document.getElementById("but2");  
var inputs = document.getElementsByTagName("input");  
var divs = document.getElementsByTagName("div");  
var but1 = document.getElementById("but1");  
var but2 = document.getElementById("but2");  
var inputs = document.getElementsByTagName("input");  
var divs = document.getElementsByTagName("div");  
var but1 = document.getElementById("but1");  
var but2 = document.getElementById("but2");  
var inputs = document.getElementsByTagName("input");  
var divs = document.getElementsByTagName("div");  
var but1 = document.getElementById("but1");  
var but2 = document.getElementById("but2");  
var inputs = document.getElementsByTagName("input");  
var divs = document.getElementsByTagName("div");  
var but1 = document.getElementById("but1");  
var but2 = document.getElementById("but2");  
var inputs = document.getElementsByTagName("input");  
var divs = document.getElementsByTagName("div");  
var but1 = document.getElementById("but1");  
var but2 = document.getElementById("but2");  
var inputs = document.getElementsByTagName("input");  
var divs = document.getElementsByTagName("div");  
}  
var end = new Date().getTime();  
alert("用时 " + (end - start) + " 毫秒");  
}  
function func2(){  
var start = new Date().getTime();  
var doc = document;  
for(var i = 0; i < 10000; i++){  
var but1 = doc.getElementById("but1");  
var but2 = doc.getElementById("but2");  
var inputs = doc.getElementsByTagName("input");  
var divs = doc.getElementsByTagName("div");  
var but1 = doc.getElementById("but1");  
var but2 = doc.getElementById("but2");  
var inputs = doc.getElementsByTagName("input");  
var divs = doc.getElementsByTagName("div");  
var but1 = doc.getElementById("but1");  
var but2 = doc.getElementById("but2");  
var inputs = doc.getElementsByTagName("input");  
var divs = doc.getElementsByTagName("div");  
var but1 = doc.getElementById("but1");  
var but2 = doc.getElementById("but2");  
var inputs = doc.getElementsByTagName("input");  
var divs = doc.getElementsByTagName("div");  
var but1 = doc.getElementById("but1");  
var but2 = doc.getElementById("but2");  
var inputs = doc.getElementsByTagName("input");  
var divs = doc.getElementsByTagName("div");  
var but1 = doc.getElementById("but1");  
var but2 = doc.getElementById("but2");  
var inputs = doc.getElementsByTagName("input");  
var divs = doc.getElementsByTagName("div");  
}  
var end = new Date().getTime();  
alert("用时 " + (end - start) + " 毫秒");  
}

　　上面代码中，第二种情况是先把全局对象的变量放到函数里面先保存下来，然后直接访问这个变量，而第一种情况是每次都遍历作用域链，直到全局环境，我们看到第二种情况实际上只遍历了一次，而第一种情况却是每次都遍历了，所以我们看看其执行结果：



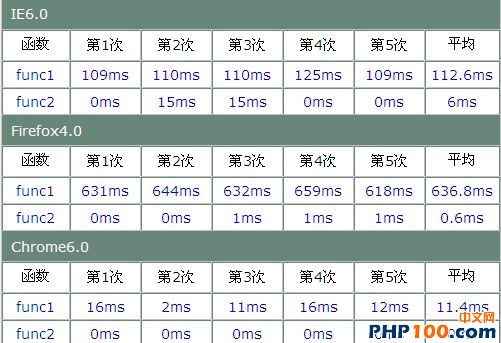
　　从上表中可以看出，其在IE6下差别还是非常明显的，而且这种差别在多级作用域链和多个全局变量的情况下还会表现的非常明显。

**5、避免双重解释**

　　双重解释的情况也是我们经常会碰到的，有的时候我们没怎么考虑到这种情况会影响到效率，双重解释一般在我们使用eval、new Function和setTimeout等情况下会遇到，我们看看下面的例子：

<div id="demo"></div>  
<input id="but1" type="button" onclick="func1()" value="效率低"/>  
<input id="but2" type="button" onclick="func2()" value="效率高"/>  
var sum, num1 = 1, num2 = 2;  
function func1(){  
var start = new Date().getTime();  
for(var i = 0; i < 10000; i++){  
var func = new Function("sum+=num1;num1+=num2;num2++;");  
func();  
}  
var end = new Date().getTime();  
alert("用时 " + (end - start) + " 毫秒");  
}  
  
function func2(){  
var start = new Date().getTime();  
for(var i = 0; i < 10000; i++){  
sum+=num1;  
num1+=num2;  
num2++;  
}  
var end = new Date().getTime();  
alert("用时 " + (end - start) + " 毫秒");  
}

　　第一种情况我们是使用了new Function来进行双重解释，而第二种是避免了双重解释，我们看看在不同浏览器下的表现：



　　可以看到，在所有的浏览器中，双重解释都是有很大开销的，所以在实际当中要尽量避免双重解释。

　　感谢"SeaSunK"对第四点测试报告错误的指正，现在已经修改过来了。至于最后一点提出的func1每次都初始化，没有可比性，所以我给换了eval，结果发现，在IE6.0下还是有影响，而且在Firefox下，使用eval对效率的影响程度更加厉害，在Firefox下，如果10000次循环，需要十多秒的时间，所以我把循环都变成了1000次。看代码和报告。

var sum, num1 = 1, num2 = 2;  
function func1(){  
var start = new Date().getTime();  
for(var i = 0; i < 1000; i++){  
eval("sum+=num1;num1+=num2;num2++;");  
}  
var end = new Date().getTime();  
alert("用时 " + (end - start) + " 毫秒");  
}  
function func2(){  
var start = new Date().getTime();  
for(var i = 0; i < 1000; i++){  
sum+=num1;  
num1+=num2;  
num2++;  
}  
var end = new Date().getTime();  
alert("用时 " + (end - start) + " 毫秒");  
}

