**JavaScript 性能调优**

JavaScript 语言由于它的单线程和解释执行的两个特点，决定了它本身有很多地方有性能问题，所以可改进的地方有不少。

**eval 的问题：**

比较下述代码：

**清单 1. eval 的问题**

var reference = {}, props = “p1”;

eval(“reference.” + props + “=5”)

var reference = {}, props = “p1”;

reference[props] = 5

有“eval”的代码比没有“eval”的代码要慢上 100 倍以上。

主要原因是：JavaScript 代码在执行前会进行类似“预编译”的操作：首先会创建一个当前执行环境下的活动对象，并将那些用 var 申明的变量设置为活动对象的属性，但是此时这些变量的赋值都是 undefined，并将那些以 function 定义的函数也添加为活动对象的属性，而且它们的值正是函数的定义。但是，如果你使用了“eval”，则“eval”中的代码（实际上为字符串）无法预先识别其上下文，无法被提前解析和优化，即无法进行预编译的操作。所以，其性能也会大幅度降低。

**Function 的用法：**

比较下述代码：

**清单 2. function 的用法**

var func1 = new Function(“return arguments[0] + arguments[1]”);

func1(10, 20);

var func2 = function(){ return arguments[0] + arguments[1] };

func2(10, 20);

这里类似之前提到的“eval”方法，这里“func1”的效率会比“func2”的效率差很多，所以推荐使用第二种方式。

**函数的作用域链（scope chain）：**

JavaScript 代码解释执行，在进入函数内部时，它会预先分析当前的变量，并将这些变量归入不同的层级（level），一般情况下：

局部变量放入层级 1（浅），全局变量放入层级 2（深）。如果进入“with”或“try – catch”代码块，则会增加新的层级，即将“with”或“catch”里的变量放入最浅层（层 1），并将之前的层级依次加深。

参考如下代码：

**清单 3. 函数作用域链**

var myObj = … ..

… ..

function process(){

var images = document.getElementsByTagName("img"),

widget = document.getElementsByTagName("input"),

combination = [];

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

combination.push(combine(images[i], widget[2\*i]));

}

myObj.container.property1 = combination[0];

myObj.container.property2 = combination[combination.length-1];

}

这里我们可以看到，“images”，“widget”，“combination”属于局部变量，在层 1。“document”，“myObj”属于全局变量，在层 2。

变量所在的层越浅，访问（读取或修改）速度越快，层越深，访问速度越慢。所以这里对“images”，“widget”，“combination”的访问速度比“document”，“myObj”要快一些。所以推荐尽量使用局部变量，可见如下代码：

**清单 4. 使用局部变量**

var myObj = … ..

… ..

function process(){

var doc = document;

var images = doc.getElementsByTagName("img"),

widget = doc.getElementsByTagName("input"),

combination = [];

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

combination.push(combine(images[i], widget[2\*i]));

}

myObj.container.property1 = combination[0];

myObj.container.property2 = combination[combination.length-1];

}

我们用局部变量“doc”取代全局变量“document”，这样可以改进性能，尤其是对于大量使用全局变量的函数里面。

再看如下代码：

**清单 5. 慎用 with**

var myObj = … ..

… ..

function process(){

var doc = document;

var images = doc.getElementsByTagName("img"),

widget = doc.getElementsByTagName("input"),

combination = [];

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

combination.push(combine(images[i], widget[2\*i]));

}

with (myObj.container) {

property1 = combination[0];

property2 = combination[combination.length-1];

}

}

加上“with”关键字，我们让代码更加简洁清晰了，但是这样做性能会受影响。正如之前说的，当我们进入“with”代码块时，“combination”便从原来的层 1 变到了层 2，这样，效率会大打折扣。所以比较一下，还是使用原来的代码：

**清单 6. 改进 with**

var myObj = … ..

… ..

function process(){

var doc = document;

var images = doc.getElementsByTagName("img"),

widget = doc.getElementsByTagName("input"),

combination = [];

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

combination.push(combine(images[i], widget[2\*i]));

}

myObj.container.property1 = combination[0];

myObj.container.property2 = combination[combination.length-1];

}

但是这样并不是最好的方式，JavaScript 有个特点，对于 object 对象来说，其属性访问层级越深，效率越低，比如这里的“myObj”已经访问到了第 3 层，我们可以这样改进一下：

**清单 7. 缩小对象访问层级**

var myObj = … ..

… ..

function process(){

var doc = document;

var images = doc.getElementsByTagName("img"),

widget = doc.getElementsByTagName("input"),

combination = [];

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

combination.push(combine(images[i], widget[2\*i]));

}

var ctn = myObj.container;

ctn.property1 = combination[0];

ctn.property2 = combination[combination.length-1];

}

我们用局部变量来代替“myObj”的第 2 层的“container”对象。如果有大量的这种对对象深层属性的访问，可以参照以上方式提高性能。

**字符串（String）相关**

**字符串拼接**

经常看到这样的代码：

**清单 8. 字符串简单拼接**

str += “str1” + “str2”

这是我们拼接字符串常用的方式，但是这种方式会有一些临时变量的创建和销毁，影响性能，所以推荐使用如下方式拼接：

**清单 9. 字符串数组方式拼接**

var str\_array = [];

str\_array.push(“str1”);

str\_array.push(“str2”);

str = str\_array.join(“”);

这里我们利用数组（array）的“join”方法实现字符串的拼接，尤其是程序的老版本的 Internet Explore（IE6）上运行时，会有非常明显的性能上的改进。

当然，最新的浏览器（如火狐 Firefox3+，IE8+ 等等）对字符串的拼接做了优化，我们也可以这样写：

**清单 10. 字符串快速拼接**

str +=“str1”

str +=“str2”

新的浏览器对“+=”做了优化，性能略快于数组的“join”方法。在不久的将来更新版本浏览器可能对“+”也会做优化，所以那时我们可以直接写：str += “str1” + “str2”。

**隐式类型转换**

参考如下代码：

**清单 11. 隐式类型转换**

var str = “12345678”, arr = [];

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

arr.push( str.charAt(i));

}

这里我们在每个循环时都会调用字符串的“charAt”方法，但是由于我们是将常量“12345678”赋值给“str”，所以“str”这里事实上并不是一个字符串对象，当它每次调用“charAt”函数时，都会临时构造值为“12345678”的字符串对象，然后调用“charAt”方法，最后再释放这个字符串临时对象。我们可以做一些改进：

**清单 12. 避免隐式类型转换**

var str = new Stirng(“12345678”), arr = [];

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

arr.push( str.charAt(i));

}

这样一来，变量“str”作为一个字符串对象，就不会有这种隐式类型转换的过程了，这样一来，效率会显著提高。

**字符串匹配**

JavaScript 有 RegExp 对象，支持对字符串的正则表达式匹配。是一个很好的工具，但是它的性能并不是非常理想。相反，字符串对象（String）本身的一些基本方法的效率是非常高的，比如“substring”，“indexOf”，“charAt”等等，在我们需要用正则表达式匹配字符串时，可以考虑一下：

1. 是否能够通过字符串对象本身支持的基本方法解决问题。
2. 是否可以通过“substring”来缩小需要用正则表达式的范围。

这些方式都能够有效的提高程序的效率。

关于正则表达式对象，还有一点需要注意，参考如下代码：

**清单 13. 正则表达式**

for(var i = 0; i <= str\_array.length; i++){

if(str\_array[i].match(/^s\*extra\s/)){

……………………

}

}

这里，我们往“match”方法传入“/^s\*extra\s/”是会影响效率的，它会构建临时值为“/^s\*extra\s/”的正则表达式对象，执行“match”方法，然后销毁临时的正则表达式对象。我们可以这样做：

**清单 14. 利用变量**

var sExpr = /^s\*extra\s/;

for(var i = 0; i <= str\_array.length; i++){

if(str\_array[i].match(sExpr)){

……………………

}

}

这样就不会有临时对象了。

**setTimeout 和 setInterval**

“setTimeout”和“setInterval”这两个函数可以接受字符串变量，但是会带来和之前谈到的“eval”类似的性能问题，所以建议还是直接传入函数对象本身。

**利用提前退出**

参考如下两段代码：

**清单 15. 利用提前退出**

// 代码 1

var name = … .;

var source = …… ;

if(source.match(/ …… /)){

……………………………

}

// 代码 2

var name = … .;

var source = …… ;

if(name.indexOf( … ) &&source.match(/ …… /)){

……………………………

}

代码 2 多了一个对“name.indexOf( … )”的判断，这使得程序每次走到这一段时会先执行“indexOf”的判断，再执行后面的“match”，在“indexOf”比“match”效率高很多的前提下，这样做会减少“match”的执行次数，从而一定程度的提高效率。

[**回页首**](http://www.ibm.com/developerworks/cn/web/1107_zhouxiang_tunejs/#ibm-pcon)

**DOM 操作性能调优**

JavaScript 的开发离不开 DOM 的操作，所以对 DOM 操作的性能调优在 Web 开发中也是非常重要的。

**Repaint 和 Reflow**

Repaint 也叫 Redraw，它指的是一种不会影响当前 DOM 的结构和布局的一种重绘动作。如下动作会产生 Repaint 动作：

1. 不可见到可见（visibility 样式属性）
2. 颜色或图片变化（background, border-color, color 样式属性）
3. 不改变页面元素大小，形状和位置，但改变其外观的变化

Reflow 比起 Repaint 来讲就是一种更加显著的变化了。它主要发生在 DOM 树被操作的时候，任何改变 DOM 的结构和布局都会产生 Reflow。但一个元素的 Reflow 操作发生时，它的所有父元素和子元素都会放生 Reflow，最后 Reflow 必然会导致 Repaint 的产生。举例说明，如下动作会产生 Repaint 动作：

1. 浏览器窗口的变化
2. DOM 节点的添加删除操作
3. 一些改变页面元素大小，形状和位置的操作的触发

**减少 Reflow**

通过 Reflow 和 Repaint 的介绍可知，每次 Reflow 比其 Repaint 会带来更多的资源消耗，我们应该尽量减少 Reflow 的发生，或者将其转化为只会触发 Repaint 操作的代码。

参考如下代码：

**清单 16. reflow 介绍**

var pDiv = document.createElement(“div”);

document.body.appendChild(pDiv);----- reflow

var cDiv1 = document.createElement(“div”);

var cDiv2 = document.createElement(“div”);

pDiv.appendChild(cDiv1);----- reflow

pDiv.appendChild(cDiv2);----- reflow

这是我们经常接触的代码了，但是这段代码会产生 3 次 reflow。再看如下代码：

**清单 17. 减少 reflow**

var pDiv = document.createElement(“div”);

var cDiv1 = document.createElement(“div”);

var cDiv2 = document.createElement(“div”);

pDiv.appendChild(cDiv1);

pDiv.appendChild(cDiv2);

document.body.appendChild(pDiv);----- reflow

这里便只有一次 reflow，所以我们推荐这种 DOM 节点操作的方式。

关于上述较少 Reflow 操作的解决方案，还有一种可以参考的模式：

**清单 18. 利用 display 减少 reflow**

var pDiv = document.getElementById(“parent”);

pDiv.style.display = “none”----- reflow

pDiv.appendChild(cDiv1);

pDiv.appendChild(cDiv2);

pDiv.appendChild(cDiv3);

pDiv.appendChild(cDiv4);

pDiv.appendChild(cDiv5);

pDiv.style.width = “100px”;

pDiv.style.height = “100px”;

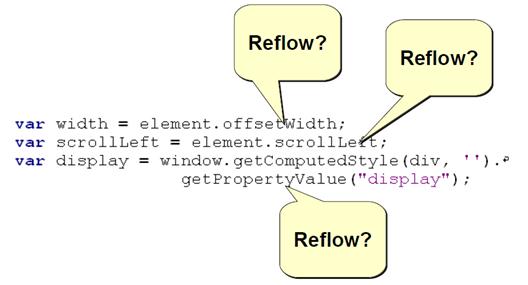
pDiv.style.display = “block”----- reflow

先隐藏 pDiv，再显示，这样，隐藏和显示之间的操作便不会产生任何的 Reflow，提高了效率。

**特殊测量属性和方法**

DOM 元素里面有一些特殊的测量属性的访问和方法的调用，也会触发 Reflow，比较典型的就是“offsetWidth”属性和“getComputedStyle”方法。

**图 1. 特殊测量属性和方法**



这些测量属性和方法大致有这些：

* offsetLeft
* offsetTop
* offsetHeight
* offsetWidth
* scrollTop/Left/Width/Height
* clientTop/Left/Width/Height
* getComputedStyle()
* currentStyle(in IE))

这些属性和方法的访问和调用，都会触发 Reflow 的产生，我们应该尽量减少对这些属性和方法的访问和调用，参考如下代码：

**清单 19. 特殊测量属性**

var pe = document.getElementById(“pos\_element”);

var result = document.getElementById(“result\_element”);

var pOffsetWidth = pe.offsetWidth;

result.children[0].style.width = pOffsetWidth;

result.children[1].style.width = pOffsetWidth;

result.children[2].style.width = pOffsetWidth;

…………其他修改…………

这里我们可以用临时变量将“offsetWidth”的值缓存起来，这样就不用每次访问“offsetWidth”属性。这种方式在循环里面非常适用，可以极大地提高性能。

**样式相关**

我们肯定经常见到如下的代码：

**清单 20. 样式相关**

var sElement = document.getElementById(“pos\_element”);

sElement.style.border = ‘ 1px solid red ’

sElement.style.backgroundColor = ‘ silver ’

sElement.style.padding = ‘ 2px 3px ’

sElement.style.marginLeft = ‘ 5px ’

但是可以看到，这里的每一个样式的改变，都会产生 Reflow。需要减少这种情况的发生，我们可以这样做：

**解决方案 1：**

**清单 21. className 解决方案**

.class1 {

border: ‘ 1px solid red ’

background-color: ‘ silver ’

padding: ‘ 2px 3px ’

margin-left: ‘ 5px ’

}

document.getElementById(“pos\_element”).className = ‘class1’ ;

用 class 替代 style，可以将原有的所有 Reflow 或 Repaint 的次数都缩减到一个。

**解决方案 2：**

**清单 22. cssText 解决方案**

var sElement = document.getElementById(“pos\_element”);

var newStyle = ‘ border: 1px solid red; ’ + ‘ background-color: silver; ’ +

‘ padding: 2px 3px; ’ + “margin-left: 5px;”

sElement.style.cssText += newStyle;

一次性设置所有样式，也是减少 Reflow 提高性能的方法。

**XPath**

一个页面上往往包含 1000 多页面元素，在定位具体元素的时候，往往需要一定的时间。如果用 id 或 name 定位可能效率不会太慢，如果用元素的一些其他属性（比如 className 等等）定位，可能效率有不理想了。有的可能只能通过遍历所有元素（getElementsByTagName）然后过滤才能找到相应元素，这就更加低效了，这里我们推荐使用 XPath 查找元素，这是很多浏览器本身支持的功能。

**清单 23. XPath 解决方案**

if(document.evaluate){

var tblHeaders = document.evaluate(“//body/div/table//th”);

var result = tblHeaders.iterateNext();

while(result) {

result.style.border = “1px dotted blue”;

result ………………

result = xpathResult.iterateNext();

}

} else{ //getElementsByTagName() ……

// 处理浏览器不支持 XPath 的情况

………………………………

}

浏览器 XPath 的搜索引擎会优化搜索效率，大大缩短结果返回时间。

**HTMLCollection 对象**

这是一类特殊的对象，它们有点像数组，但不完全是数组。下述方法的返回值一般都是 HTMLCollection 对象：

* document.images, document.forms
* getElementsByTagName()
* getElementsByClassName()

这些 HTMLCollection 对象并不是一个固定的值，而是一个动态的结果。它们是一些比较特殊的查询的返回值，在如下情况下，它们会重新执行之前的查询而得到新的返回值（查询结果），虽然多数情况下会和前一次或几次的返回值都一样：

* Length 属性
* 具体的某个成员

所以，HTMLCollection 对象对这些属性和成员的访问，比起数组来要慢很多。当然也有例外，Opera 和 Safari 对这种情况就处理的很好，不会有太大性能问题。

参考如下代码：

**清单 24. HTMLConnection 对象**

var items = [“test1”, “test2”, “test3”, ……………… ];

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

………………………………

}

var items = document.getElementsByTagName(“div”);

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

…………………………………… .

}

上述两端代码，下面的效率比起上面一段要慢很多，因为每一个循环都会有“items.length”的触发，也就会导致“document.getElementsByTagName(..)”方法的再次调用，这便是效率便会大幅度下降的原因。我们可以这样解决：

**清单 25. HTMLConnection 对象解决方案**

var items = document.getElementsByTagName(“div”);

var len = items.length

for(var i = 0; i < len; i++){

…………………………………… .

}

这样一来，效率基本与普通数组一样。

**动态创建 script 标签**

加载并执行一段 JavaScript 脚本是需要一定时间的，在我们的程序中，有时候有些 JavaScript 脚本被加载后基本没有被使用过 （比如：脚本里的函数从来没有被调用等等）。加载这些脚本只会占用 CPU 时间和增加内存消耗，降低 Web 应用的性能。所以推荐动态的加载 JavaScript 脚本文件，尤其是那些内容较多，消耗资源较大的脚本文件。

**清单 26. 创建 script 标签**

if(needXHR){

document.write(“<script type= ’ test\/JavaScript ’ src= 'dojo\_xhr.js' >”);

}

if(dojo.isIE){

document.write(“<script type= ’ test\/JavaScript ’ src= 'vml.js' >”);

}