一．执行效率  
1. DOM  
**1.1** 使用 **DocumentFragment** 优化多次 **append**  
说明：添加多个 dom 元素时，先将元素 append 到 DocumentFragment  
中，最后统一将 DocumentFragment 添加到页面。  
该做法可以减少页面渲染 dom 元素的次数。经 IE 和 Fx 下测试，在  
append1000 个元素时，效率能提高 10%-30%， Fx 下提升较为明显。  
服用前：  
for (var i = 0; i < 1000; i++) {  
var el = document.createElement('p');  
el.innerHTML = i;  
document.body.appendChild(el);  
}  
服用后：  
var frag = document.createDocumentFragment();  
for (var i = 0; i < 1000; i++) {  
var el = document.createElement('p');  
el.innerHTML = i;  
frag.appendChild(el);  
}  
document.body.appendChild(frag);  
**1.2** 通过模板元素 **clone**，替代 **createElement**  
说明：通过一个模板 dom 对象 cloneNode，效率比直接创建 element  
高。  
性能提高不明显，约为 10%左右。在低于 100 个元素 create 和 append  
操作时，没有优势。  
服用前：  
var frag = document.createDocumentFragment();  
for (var i = 0; i < 1000; i++) {  
var el = document.createElement('p');  
el.innerHTML = i;  
frag.appendChild(el);  
}  
document.body.appendChild(frag);  
服用后：  
var frag = document.createDocumentFragment();  
var pEl = document.getElementsByTagName('p')[0];  
for (var i = 0; i < 1000; i++) {  
var el = pEl.cloneNode(false);  
el.innerHTML = i;  
frag.appendChild(el);  
}  
document.body.appendChild(frag);  
**1.3** 使用一次 **innerHTML** 赋值代替构建 **dom** 元素  
说明：根据数据构建列表样式的时候，使用设置列表容器 innerHTML 的  
方式，比构建 dom 元素并 append 到页面中的方式，效率有数量级上的  
提高。  
服用前：  
var frag = document.createDocumentFragment();  
for (var i = 0; i < 1000; i++) {  
var el = document.createElement('p');  
el.innerHTML = i;  
frag.appendChild(el);  
}  
document.body.appendChild(frag);  
服用后：  
var html = [];  
for (var i = 0; i < 1000; i++) {  
html.push('<p>' + i + '</p>');  
}  
document.body.innerHTML = html.join('');  
**1.4** 使用 **firstChild** 和 **nextSibling** 代替 **childNodes** 遍历 **dom** 元  
素  
说明：约能获得 30%-50%的性能提高。逆向遍历时使用 lastChild 和  
previousSibling。  
服用前：  
var nodes = element.childNodes;  
for (var i = 0, l = nodes.length; i < l; i++) {  
var node = nodes[i];  
……  
}  
服用后：  
var node = element.firstChild;  
while (node) {  
……  
node = node.nextSibling;  
}  
2. 字符串  
**2.1** 使用 **Array** 做为 **StringBuffer**，代替字符串拼接的操作  
说明： IE 在对字符串拼接的时候，会创建临时的 String 对象；经测试，  
在 IE 下，当拼接的字符串越来越大时，运行效率会急剧下降。Fx 和 Opera  
都对字符串拼接操作进行了优化；经测试，在 Fx 下，使用 Array 的 join  
方式执行时间约为直接字符串拼接的 1.4 倍。  
服用前：  
var now = new Date();  
var str = '';  
for (var i = 0; i < 10000; i++) {  
str += '123456789123456789';  
}  
alert(new Date() - now);  
服用后：  
var now = new Date();  
var strBuffer = [];  
for (var i = 0; i < 10000; i++) {  
strBuffer.push('123456789123456789');  
}  
var str = strBuffer.join('');  
alert(new Date() - now);  
3. 循环语句  
**3.1** 将循环控制量保存到局部变量  
说明：对数组和列表对象的遍历时，提前将 length 保存到局部变量中，  
避免在循环的每一步重复取值。  
服用前：  
var list = document.getElementsByTagName('p');  
for (var i = 0; i < list.length; i++) {  
……  
}服  
用后：  
var list = document.getElementsByTagName('p');  
for (var i = 0, l = list.length; i < l; i++) {  
……  
}  
**3.2** 顺序无关的遍历时，用 **while** 替代 **for**  
说明：该方法可以减少局部变量的使用。比起效率优化，更能直接看到的  
是字符数量的优化。该做法有程序员强迫症的嫌疑。  
服用前：  
var arr = [1,2,3,4,5,6,7];  
var sum = 0;  
for (var i = 0, l = arr.length; i < l; i++) {  
sum += arr[i];  
}  
服用后：  
var arr = [1,2,3,4,5,6,7];  
var sum = 0, l = arr.length;  
while (l--) {  
sum += arr[l];  
}  
4. 条件分支  
**4.1** 将条件分支，按可能性顺序从高到低排列  
说明：可以减少解释器对条件的探测次数。  
**4.2** 在同一条件子的多（ **>2**）条件分支时，使用 **switch** 优于 **if**  
说明： switch 分支选择的效率高于 if，在 IE 下尤为明显。 4 分支的测试，  
IE 下 switch 的执行时间约为 if 的一半。  
**4.3** 使用三目运算符替代条件分支  
服用前：  
if (a > b) {  
num = a;  
} else {  
num = b;  
}  
服用后：  
num = a > b ? a : b;  
5. 定时器  
**5.1** 需要不断执行的时候，优先考虑使用 **setInterval**  
说明： setTimeout 每一次都会初始化一个定时器。 setInterval 只会在  
开始的时候初始化一个定时器  
服用前：  
var timeoutTimes = 0;  
function timeout () {  
timeoutTimes++;  
if (timeoutTimes < 10) {  
setTimeout(timeout, 10);  
}  
}  
timeout();  
服用后：  
var intervalTimes = 0;  
function interval () {  
intervalTimes++;  
if (intervalTimes >= 10) {  
clearInterval(interv);  
}  
}  
var interv = setInterval(interval, 10);  
**5.2** 使用 **function** 而不是 **string**  
说明：如果把字符串作为 setTimeout 和 setInterval 的参数，浏览器会  
先用这个字符串构建一个 function 。  
服用前：  
var num = 0;  
setTimeout('num++', 10);  
服用后：  
var num = 0;  
function addNum () {  
num++;  
}  
setTimeout(addNum, 10);  
6. 其他  
**6.1** 尽量不使用动态语法元素  
说明： eval、 Function 、 execScript 等语句会再次使用 javascript 解析  
引擎进行解析，需要消耗大量的执行时间。  
**6.2** 重复使用的调用结果，事先保存到局部变量  
说明：避免多次取值的调用开销。  
服用前：  
var h1 = element1.clientHeight + num1;  
var h2 = element1.clientHeight + num2;  
服用后：  
var eleHeight = element1.clientHeight;  
var h1 = eleHeight + num1;  
var h2 = eleHeight + num2;  
**6.3** 使用直接量  
说明：  
var a = new Array(param,param,...) -> var a = []  
var foo = new Object() -> var foo = {}  
var reg = new RegExp() -> var reg = /.../  
**6.4** 避免使用 **with**  
说明： with 虽然可以缩短代码量，但是会在运行时构造一个新的 scope。  
OperaDev 上还有这样的解释，使用 with 语句会使得解释器无法在语法  
解析阶段对代码进行优化。对此说法，无法验证。  
服用前：  
with (a.b.c.d) {  
property1 = 1;  
property2 = 2;  
}  
服用后：  
var obj = a.b.c.d;  
obj.property1 = 1;  
obj.property2 = 2;  
**6.5** 巧用**||** 和**&&**布尔运算符  
重要程度： ★★★  
服用前：  
function eventHandler (e) {  
if(!e) e = window.event;  
}  
服用后：  
function eventHandler (e) {  
e = e || window.event;  
}  
服用前：  
if (myobj) {  
doSomething(myobj);  
}  
服用后：  
myobj && doSomething(myobj);  
**6.6** 类型转换  
说明：  
1). 数字转换成字符串，应用"" + 1 ，性能上： ("" +) >  
String() > .toString() > new String()；  
2). 浮点数转换成整型，不使用 parseInt()， parseInt()是用于将字  
符串转换成数字，而不是浮点数和整型之间的转换，建议使用 Math.floor()  
或者 Math.round()  
3). 对于自定义的对象，推荐显式调用 toString()。内部操作在尝试所  
有可能性之后，会尝试对象的 toString()方法尝试能否转化为 String。  
二．内存管理  
**2.1** 循环引用  
说明：如果循环引用中包含 DOM 对象或者 ActiveX 对象，那么就会发生  
内存泄露。内存泄露的后果是在浏览器关闭前，即使是刷新页面，这部分  
内存不会被浏览器释放。  
简单的循环引用：  
var el = document.getElementById('MyElement');  
var func = function () {…}  
el.func = func;  
func.element = el;  
但是通常不会出现这种情况。通常循环引用发生在为 dom 元素添加闭包  
作为 expendo 的时候。  
如：  
function init() {  
var el = document.getElementById('MyElement');  
el.onclick = function () {……}  
}  
init();  
init 在执行的时候，当前上下文我们叫做 context。这个时候， context  
引用了 el， el 引用了 function， function 引用了 context。这时候形成  
了一个循环引用。  
下面 2 种方法可以解决循环引用：  
1) 置空 dom 对象  
服用前：  
function init() {  
var el = document.getElementById('MyElement');  
el.onclick = function () {……}  
}  
init();  
服用后：  
function init() {  
var el = document.getElementById('MyElement');  
el.onclick = function () {……}  
el = null;  
}  
init();  
将 el 置空， context 中不包含对 dom 对象的引用，从而打断循环应用。  
如果我们需要将 dom 对象返回，可以用如下方法：  
服用前：  
function init() {  
var el = document.getElementById('MyElement');  
el.onclick = function () {……}  
return el;  
}  
init();  
服用后：  
function init() {  
var el = document.getElementById('MyElement');  
el.onclick = function () {……}  
try{  
return el;  
} finally {  
el = null;  
}  
init();  
2) 构造新的 context  
服用前：  
function init() {  
var el = document.getElementById('MyElement');  
el.onclick = function () {……}  
}  
init();  
服用后：  
function elClickHandler() {……}  
function init() {  
var el = document.getElementById('MyElement');  
el.onclick = elClickHandler;  
}  
init();  
把 function 抽到新的 context 中，这样， function 的 context 就不包含  
对 el 的引用，从而打断循环引用。  
**2.2** 通过 **javascript** 创建的 **dom** 对象，必须 **append** 到页面中  
说明： IE 下，脚本创建的 dom 对象，如果没有 append 到页面中，刷新  
页面，这部分内存是不会回收的！  
示例代码：  
function create () {  
var gc = document.getElementById('GC');  
for (var i = 0; i < 5000 ; i++)  
{  
var el = document.createElement('div');  
el.innerHTML = "test";  
//下面这句可以注释掉，看看浏览器在任务管理器中，点击按钮  
然后刷新后的内存变化  
gc.appendChild(el);  
}  
}  
**2.3** 释放 **dom** 元素占用的内存  
说明：  
将 dom 元素的 innerHTML 设置为空字符串，可以释放其子元素占用的  
内存。  
在 rich 应用中，用户也许会在一个页面上停留很长时间，可以使用该方法  
释放积累得越来越多的 dom 元素使用的内存。  
**2.4** 释放 **javascript** 对象  
说明：在 rich 应用中，随着实例化对象数量的增加，内存消耗会越来越大。  
所以应当及时释放对对象的引用，让 GC 能够回收这些内存控件。  
对象： obj = null  
对象属性： delete obj.myproperty  
数组 item：使用数组的 splice 方法释放数组中不用的 item  
**2.5** 避免