**避免全局查找**

在一个函数中会用到全局对象存储为局部变量来减少全局查找，因为访问局部变量的速度要比访问全局变量的速度更快些

**定时器**

如果针对的是不断运行的代码，**不应该使用setTimeout，而应该是用setInterval**，因为setTimeout每一次都会初始化一个定时器，而setInterval只会在开始的时候初始化一个定时器

**字符串连接**

如果要连接多个字符串，应该少使用+=，如

s+=a;s+=b;s+=c;

应该写成s+=a + b + c；

而如果是收集字符串，比如多次对同一个字符串进行+=操作的话，最好使用一个缓存，使用JavaScript数组来收集，最后**使用join方法连接起来**

var buf = [];

for (var i = 0; i < 100; i++) {

buf.push(i.toString());

}

var all = buf.join('');

**避免with语句**

和函数类似 ，with语句会创建自己的作用域，因此会增加其中执行的代码的作用域链的长度，由于额外的作用域链的查找，在with语句中执行的代码肯定会比外面执行的代码要慢，在**能不使用with语句的时候尽量不要使用with语句**。

with (a.b.c.d) {

property1 = 1;

property2 = 2;

}

//可以替换为：

var obj = a.b.c.d;

obj.property1 = 1;

obj.property2 = 2;

**数字转换成字符串**

般最好用'' + 1来将数字转换成字符串，虽然看起来比较丑一点，但事实上这个效率是最高的，性能上来说：

**('' +) > String() > .toString() > new String()**

**浮点数转换成整型**

很多人喜欢使用parseInt()，其实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，我们应该使用Math.floor()或者Math.round()

**各种类型转换**

var myVar = '3.74159',

str = '' + myVar, // to string

i\_int = ~~myVar, // to integer

f\_float = 1 \* myVar, // to float

b\_bool = !!myVar,

/\* to boolean - any string with length

and any number except 0 are true \*/

array = [myVar]; // to array

如果定义了toString()方法来进行类型转换的话，推荐**显式调用toString()**，因为内部的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，所以直接调用这个方法效率会更高

**多个类型声明**

在JavaScript中所有变量都可以使用单个var语句来声明，这样就是组合在一起的语句，以减少整个脚本的执行时间，就如上面代码一样，上面代码格式也挺规范，让人一看就明了。

**插入迭代器**

如var name=values[i]; i++;前面两条语句可以写成var name=values[i++]

**使用直接量**

var aTest = new Array(); //替换为

var aTest = [];

var aTest = new Object; //替换为

var aTest = {};

var reg = new RegExp(); //替换为

var reg = /../;

//如果要创建具有一些特性的一般对象，也可以使用字面量，如下：

var oFruit = new Object;

oFruit.color = 'red';

oFruit.name = 'apple';

//前面的代码可用对象字面量来改写成这样：

var oFruit = { color: 'red', name: 'apple' };

**使用DocumentFragment优化多次append**

一旦需要更新DOM,请考虑使用文档碎片来构建DOM结构，然后再将其添加到现存的文档中。

for (var i = 0; i < 1000; i++) {

var el = document.createElement('p');

el.innerHTML = i;

document.body.appendChild(el);

}

//可以替换为：

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);

**使用一次innerHTML赋值代替构建dom元素**

对于大的DOM更改，使用innerHTML要比使用标准的DOM方法创建同样的DOM结构快得多。

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);

//可以替换为：

var html = [];

for (var i = 0; i < 1000; i++) {

html.push('<p>' + i + '</p>');

}

document.body.innerHTML = html.join('');

**通过模板元素clone，替代createElement**

很多人喜欢在JavaScript中使用document.write来给页面生成内容。事实上这样的效率较低，如果需要直接插入HTML，可以找 一个容器元素，比如指定一个div或者span，并设置他们的innerHTML来将自己的HTML代码插入到页面中。通常我们可能会使用字符串直接写 HTML来创建节点，其实这样做，1无法保证代码的有效性2字符串操作效率低，所以应该是用document.createElement()方法，而如 果文档中存在现成的样板节点，应该是用cloneNode()方法，因为使用createElement()方法之后，你需要设置多次元素的属性，使用 cloneNode()则可以减少属性的设置次数——同样如果需要创建很多元素，应该先准备一个样板节点

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);

//替换为：

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);

**使用firstChild和nextSibling代替childNodes遍历dom元素**

var nodes = element.childNodes;

for (var i = 0, l = nodes.length; i < l; i++) {

var node = nodes[i];

//……

}

//可以替换为：

var node = element.firstChild;

while (node) {

//……

node = node.nextSibling;

}

**删除DOM节点**

**删除dom节点之前,一定要删除注册在该节点上的事件**,不管是用observe方式还是用 attachEvent方式注册的事件,否则将会产生无法回收的内存。另外，在removeChild和innerHTML=’’二者之间,尽量选择后 者. 因为在sIEve(内存泄露监测工具)中监测的结果是用removeChild无法有效地释放dom节点

**使用事件代理**

任何可以冒泡的事件都不仅仅可以在事件目标上进行处理，目标的任何祖先节点上也能处理，使用这个知识就可以将事件处理程序附加到更高的地方负责多个目标的事件处理，同样，**对于内容动态增加并且子节点都需要相同的事件处理函数的情况，可以把事件注册提到父节点上，这样就不需要为每个子节点注册事件监听了**。另外，现有的js库都采用observe方式来创建事件监听,其实现上隔离了dom对象和事件处理函数之间的循环引用,所以应该尽量采用这种方式来创建事件监听

**重复使用的调用结果，事先保存到局部变量**

//避免多次取值的调用开销

var h3 = element1.clientHeight + num1;

var h4 = element1.clientHeight + num2;

//可以替换为：

var eleHeight = element1.clientHeight;

var h3 = eleHeight + num1;

var h4 = eleHeight + num2;

**注意NodeList**

最小化访问NodeList的次数可以极大的改进脚本的性能

var images = document.getElementsByTagName('img');

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

}

编写JavaScript的时候一定要知道何时返回NodeList对象，这样可以最小化对它们的访问

进行了对getElementsByTagName()的调用 获取了元素的childNodes属性 获取了元素的attributes属性 访问了特殊的集合，如document.forms、document.images等等

要了解了当使用NodeList对象时，合理使用会极大的提升代码执行速度

**优化循环**

可以使用下面几种方式来优化循环

**减值迭代**

大多数循环使用一个从0开始、增加到某个特定值的迭代器，在很多情况下，从最大值开始，在循环中不断减值的迭代器更加高效

**简化终止条件**

由于每次循环过程都会计算终止条件，所以必须保证它尽可能快，也就是说避免属性查找或者其它的操作，最好是将循环控制量保存到局部变量中，也就是说对数组或列表对象的遍历时，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。

var list = document.getElementsByTagName('p');

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

//……

}

//替换为：

var list = document.getElementsByTagName('p');

for (var i = 0, l = list.length; i < l; i++) {

//……

}

**简化循环体**

循环体是执行最多的，所以要确保其被最大限度的优化

**使用后测试循环**

在JavaScript中，我们可以使用for(;;),while(),for(in)三种循环，事实上，这三种循环中for(in)的效率极 差，因为他需要查询散列键，只要可以，就应该尽量少用。for(;;)和while循环，while循环的效率要优于for(;;)，可能是因为 for(;;)结构的问题，需要经常跳转回去。

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];

}

//可以考虑替换为：

var arr = [1, 2, 3, 4, 5, 6, 7];

var sum = 0, l = arr.length;

while (l--) {

sum += arr[l];

}

最常用的for循环和while循环都是前测试循环，而如do-while这种后测试循环，可以避免最初终止条件的计算，因此运行更快。

**展开循环**

当循环次数是确定的，消除循环并使用多次函数调用往往会更快。

**避免双重解释**

如果要提高代码性能，尽可能避免出现需要按照JavaScript解释的字符串，也就是

**尽量少使用eval函数**

使用eval相当于在运行时再次调用解释引擎对内容进行运行，需要消耗大量时间，而且使用Eval带来的安全性问题也是不容忽视的。

**不要使用Function构造器**

不要给setTimeout或者setInterval传递字符串参数

var num = 0;

setTimeout('num++', 10);

//可以替换为：

var num = 0;

function addNum() {

num++;

}

setTimeout(addNum, 10);

**缩短否定检测**

if (oTest != '#ff0000') {

//do something

}

if (oTest != null) {

//do something

}

if (oTest != false) {

//do something

}

//虽然这些都正确，但用逻辑非操作符来操作也有同样的效果：

if (!oTest) {

//do something

}

**条件分支**

将条件分支，按可能性顺序从高到低排列：可以减少解释器对条件的探测次数 在同一条件子的多（>2）条件分支时，使用switch优于if：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。 使用三目运算符替代条件分支

if (a > b) {

num = a;

} else {

num = b;

}

//可以替换为：

num = a > b ? a : b;

**使用常量**

**重复值**:任何在多处用到的值都应该抽取为一个常量 **用户界面字符串**:任何用于显示给用户的字符串，都应该抽取出来以方便国际化 **URLs**:在Web应用中，资源位置很容易变更，所以推荐用一个公共地方存放所有的URL **任意可能会更改的值:**每当你用到字面量值的时候，你都要问一下自己这个值在未来是不是会变化，如果答案是“是”，那么这个值就应该被提取出来作为一个常量。

**避免与null进行比较**

由于JavaScript是弱类型的，所以它不会做任何的自动类型检查，所以如果看到与null进行比较的代码，尝试使用以下技术替换

如果值应为一个引用类型，使用instanceof操作符检查其构造函数 如果值应为一个基本类型，作用typeof检查其类型 如果是希望对象包含某个特定的方法名，则使用typeof操作符确保指定名字的方法存在于对象上

**避免全局量**

**全局变量应该全部字母大写，各单词之间用\_下划线来连接**。尽可能避免全局变量和函数, 尽量减少全局变量的使用，因为在一个页面中包含的所有JavaScript都在同一个域中运行。所以如果你的代码中声明了全局变量或者全局函数的话，后面的代码中载入的脚本文件中的同名变量和函数会覆盖掉（overwrite）你的。

//糟糕的全局变量和全局函数

var current = null;

function init(){

//...

}

function change() {

//...

}

function verify() {

//...

}

//解决办法有很多，Christian Heilmann建议的方法是：

//如果变量和函数不需要在“外面”引用，那么就可以使用一个没有名字的方// 法将他们全都包起来。

(function(){

var current = null;

function init() {

//...

}

function change() {

//...

}

function verify() {

//...

}

})();

//如果变量和函数需要在“外面”引用，需要把你的变量和函数放在一个“命名空间”中

//我们这里用一个function做命名空间而不是一个var，因为在前者中声明// // function更简单，而且能保护隐私数据

myNameSpace = function() {

var current = null;

function init() {

//...

}

function change() {

//...

}

function verify() {

//...

}

//所有需要在命名空间外调用的函数和属性都要写在return里面

return {

init: init,

//甚至你可以为函数和属性命名一个别名

set: change

};

};

**尊重对象的所有权**

因为JavaScript可以在任何时候修改任意对象，这样就可以以不可预计的方式覆写默认的行为，所以如果你不负责维护某个对象，它的对象或者它的方法，那么你就不要对它进行修改，具体一点就是说：

不要为实例或原型添加属性 不要为实例或者原型添加方法 不要重定义已经存在的方法 不要重复定义其它团队成员已经实现的方法，永远不要修改不是由你所有的对象，你可以通过以下方式为对象创建新的功能: 创建包含所需功能的新对象，并用它与相关对象进行交互 创建自定义类型，继承需要进行修改的类型，然后可以为自定义类型添加额外功能

**循环引用**

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。

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**

function init() {

var el = document.getElementById('MyElement');

el.onclick = function () {

//……

}

}

init();

//可以替换为：

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：

function init() {

var el = document.getElementById('MyElement');

el.onclick = function () {

//……

}

return el;

}

init();

//可以替换为：

function init() {

var el = document.getElementById('MyElement');

el.onclick = function () {

//……

}

try {

return el;

} finally {

el = null;

}

}

init();

**2) 构造新的context**

function init() {

var el = document.getElementById('MyElement');

el.onclick = function () {

//……

}

}

init();

//可以替换为：

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。

**通过javascript创建的dom对象，必须append到页面中**

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！

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);

}

}

**释放dom元素占用的内存**

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。

**释放javascript对象**

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

**避免string的隐式装箱**

对string的方法调用，比如'xxx'.length，浏览器会进行一个隐式的装箱操作，将字符串先转换成一个String对象。推荐对声明有可能使用String实例方法的字符串时，采用如下写法：

var myString = new String('Hello World');

**松散耦合**

**1、解耦HTML/JavaScript**

JavaScript和HTML的紧密耦合：直接写在HTML中的JavaScript、使用包含内联代码的<script>元素、使用HTML属性来分配事件处理程序等

HTML和JavaScript的紧密耦合：JavaScript中包含HTML，然后使用innerHTML来插入一段html文本到页面

其实应该是保持层次的分离，这样可以很容易的确定错误的来源，所以我们应确保HTML呈现应该尽可能与JavaScript保持分离

**2、解耦CSS/JavaScript**

显示问题的唯一来源应该是CSS，行为问题的唯一来源应该是JavaScript，层次之间保持松散耦合才可以让你的应用程序更加易于维护，所以像 以下的代码element.style.color=”red”尽量改为element.className=”edit”，而且不要在css中通过表达 式嵌入JavaScript

**3、解耦应用程序/事件处理程序**

将应用逻辑和事件处理程序相分离：一个事件处理程序应该从事件对象中提取，并将这些信息传送给处理应用逻辑的某个方法中。这样做的好处首先可以让你更容易更改触发特定过程的事件，其次可以在不附加事件的情况下测试代码，使其更易创建单元测试

**性能方面的注意事项**

**1、尽量使用原生方法**

**2、switch语句相对if较快**

通过将case语句按照最可能到最不可能的顺序进行组织

**3、位运算较快**

当进行数字运算时，位运算操作要比任何布尔运算或者算数运算快

**4、巧用||和&&布尔运算符**

function eventHandler(e) {

if (!e) e = window.event;

}

//可以替换为：

function eventHandler(e) {

e = e || window.event;

}

if (myobj) {

doSomething(myobj);

}

//可以替换为：

myobj && doSomething(myobj);

**避免错误应注意的地方**

**1、每条语句末尾须加分号**

在if语句中，即使条件表达式只有一条语句也要用{}把它括起来，以免后续如果添加了语句之后造成逻辑错误

**2、使用+号时需谨慎**

JavaScript 和其他编程语言不同的是，在 JavaScript 中，'+'除了表示数字值相加，字符串相连接以外，还可以作一元运算符用，把字符串转换为数字。因而如果使用不当，则可能与自增符'++'混淆而引起计算错误

var valueA = 20;

var valueB = '10';

alert(valueA + valueB); //ouput: 2010

alert(valueA + (+valueB)); //output: 30

alert(valueA + +valueB); //output:30

alert(valueA ++ valueB); //Compile error

**3、使用return语句需要注意**

一条有返回值的return语句不要用()括号来括住返回值，如果返回表达式，则表达式应与return关键字在同一行，以避免压缩时，压缩工具自动加分号而造成返回与开发人员不一致的结果

function F1() {

var valueA = 1;

var valueB = 2;

return valueA + valueB;

}

function F2() {

var valueA = 1;

var valueB = 2;

return

valueA + valueB;

}

alert(F1()); //output: 3

alert(F2()); //ouput: undefined

**==和===的区别**

避免在if和while语句的条件部分进行赋值，如if (a = b)，应该写成if (a == b)，但是在比较是否相等的情况下，最好使用全等运行符，也就是使用===和!==操作符会相对于==和!=会好点。==和!=操作符会进行类型强制转换

var valueA = '1';

var valueB = 1;

if (valueA == valueB) {

alert('Equal');

}

else {

alert('Not equal');

}

//output: 'Equal'

if (valueA === valueB) {

alert('Equal');

}

else {

alert('Not equal');

}

//output: 'Not equal'

**不要使用生偏语法**

不要使用生偏语法，写让人迷惑的代码，虽然计算机能够正确识别并运行，但是晦涩难懂的代码不方便以后维护

**函数返回统一类型**

虽然JavaScript是弱类型的，对于函数来说，前面返回整数型数据，后面返回布尔值在编译和运行都可以正常通过，但为了规范和以后维护时容易理解，应保证函数应返回统一的数据类型

**总是检查数据类型**

要检查你的方法输入的所有数据，一方面是为了安全性，另一方面也是为了可用性。用户随时随地都会输入错误的数据。这不是因为他们蠢，而是因为他们很忙，并且思考的方式跟你不同。用typeof方法来检测你的function接受的输入是否合法

**何时用单引号，何时用双引号**

虽然在JavaScript当中，双引号和单引号都可以表示字符串, 为了避免混乱，我们建议在HTML中使用双引号，在JavaScript中使用单引号，但为了兼容各个浏览器，也为了解析时不会出错，定义JSON对象时，最好使用双引号

**部署**

用JSLint运行JavaScript验证器来确保没有语法错误或者是代码没有潜在的问 部署之前推荐使用压缩工具将JS文件压缩 文件编码统一用UTF-8

JavaScript 程序应该尽量放在 .js 的文件中，需要调用的时候在 HTML 中以 <script src='filename.js'> 的形式包含进来。JavaScript 代码若不是该 HTML 文件所专用的，则应尽量避免在 HTML 文件中直接编写 JavaScript 代码。因为这样会大大增加 HTML 文件的大小，无益于代码的压缩和缓存的使用。另外，<script src='filename.js'> 标签应尽量放在文件的后面,最好是放在</body>标签前。这样会降低因加载 JavaScript 代码而影响页面中其它组件的加载时间。

**永远不要忽略代码优化工作，重构是一项从项目开始到结束需要持续的工作，只有不断的优化代码才能让代码的执行效率越来越好**