JS性能优化

摘自：

http://www.china125.com/design/js/3631.htm

首先，由于JS是一种解释型语言，执行速度要比编译型语言慢得多。（注：，Chrome是第一款内置优化引擎，将JS编译成本地代码的浏览器，其它浏览器也陆续实现了JS的编译过程。但是，即使到了编译执行JS的新阶段，仍然会存在低效率的代码。）以下总结一些可以改进代码的整体性能的方法。

1.注意作用域

记住一点，随着作用域中的作用域数量的增加，访问当前作用域以外的变量的时间也在增加。所以，访问全局变量总是比访问局部变量要慢，因为需要遍历作用域链。只要能减少花费在作用域链上的时间，就能增加脚本的整体性能。

1). 避免全局查找（因为涉及作用域上的查找）

```
function updateUI() {
    var imgs =
document.getElementByTagName("img");
    for(var i = 0, len = imgs.length; i < len; i++)
{
```

```
        imgs[i].title = document.title + " image " +
i;
    }
}
```

注意，updateUI中包含了二个对于全局变量document对象的引用，特别是循环中的document引用，查到次数是O(n)，每次都要进行作用域链查找。通过创建一个指向document的局部变量，就可以通过限制一次全局查找来改进这个函数的性能。

```
function updateUI() {
    var doc = document;
    var imgs = doc.getElementByTagName("img");
    for(var i = 0, len = imgs.length; i < len; i++)
{
        imgs[i].title = doc.title + " image " + i;
    }
}
```

2). 避免with语句（with会创建自己的作用域，因此会增加其中执行代码的作用域的长度）

2.选择正确的方法

和其它语言一样，性能问题的一部分是和用于解决问题的算法或方法有关的，所以通过选择正确的方法也能起到优化作用。

1.避免不必要的属性查找

在JS中访问变量或数组都是O(1)操作，比访问对象上的属性更有效率，后者是一个O(n)操作。对象上的任何属性查找都要比访问变量或数组花费更长时间，因为必须在原型链中对拥有该名称的属性进行一次搜索，即属性查找越多，执行时间越长。所以针对需要多次用到对象属性，应将其存储在局部变量。

2.优化循环

循环是编程中最常见的结构，优化循环是性能优化过程中很重要的一部分。一个循环的基本优化步骤如下：

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有效。

简化终止条件——由于每次循环过程都会计算终止条件，故必须保证它尽可能快，即避免属性查找或其它O(n)的操作。

简化循环体——循环体是执行最多的，故要确保其被最大限度地优化。确保没有某些可以被很容易移出循环的密集计算。

使用后测试循环——最常用的for和while循环都是前测试循环，而如do-while循环可以避免最初终止条件的计算，因些计算更快。

```
for(var i = 0; i < values.length; i++) {
    process(values[i]);
}
```

优化1：简化终止条件

```
for(var i = 0, len = values.length; i < len; i++) {
    process(values[i]);
}
```

优化2：使用后测试循环（注意：使用后测试循环需要确保要处理的值至少有一个）

```
var i values.length - 1;
if(i > -1) {
    do {
        process(values[i]);
    }while(--i >= 0);
}
```

3.展开循环

当循环的次数确定时，消除循环并使用多次函数调用往往更快

当循环的次数不确定时，可以使用Duff装置来优化。Duff装置的基本概念是通过计算迭代的次数是否为8的倍数将一个循环展开为一系列语句。如下：

```
// Jeff Greenberg for JS implementation of Duff's Device
// 假设 : values.length > 0
function process(v) {
    alert(v);
}
var values = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17];
var iterations = Math.ceil(values.length / 8);
var startAt = values.length % 8;
var i = 0;

do {
    switch(startAt) {
        case 0 : process(values[i++]);
        case 7 : process(values[i++]);
        case 6 : process(values[i++]);
        case 5 : process(values[i++]);
        case 4 : process(values[i++]);
```

```
        case 3 : process(values[i++]);
        case 2 : process(values[i++]);
        case 1 : process(values[i++]);
    }
    startAt = 0;
}while(--iterations > 0);
```

如上展开循环可以提升大数据集的处理速度。接下来给出更快的Duff装置技术，将do-while循环分成2个单独的循环。（注：这种方法几乎比原始的Duff装置实现快上40%。）

```
// Speed Up Your Site(New Riders, 2003)
function process(v) {
    alert(v);
}
var values =
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17];
var iterations = Math.floor(values.length / 8);
var leftover = values.length % 8;
var i = 0;
if(leftover > 0) {
    do {
        process(values[i++]);
```

```
    }while(--leftover > 0);
}
do {
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
}while(--iterations > 0);
```

针对大数据集使用展开循环可以节省很多时间，但对于小数据集，额外的开销则可能得不偿失。

4.避免双重解释

当JS代码想解析JS代码时就会存在双重解释惩罚，当使用eval()函数或是Function构造函数以及使用setTimeout()传一个字符串时都会发生这种情况。如下

```
eval("alert('hello world');"); // 避免
var sayHi = new Function("alert('hello world');");
// 避免
```

```
setTimeout("alert('hello world');", 100);// 避免
```

以上代码是包含在字符串中的，即在JS代码运行的同时必须新启运一个解析器来解析新的代码。实例化一个新的解析器有不容忽视的开销，故这种代码要比直接解析要慢。以下这几个例子，除了极少情况下eval是必须的，应尽量避免使用上述。对于Function构造函数，直接写成一般的函数即可。对于setTimeout可以传入函数作为第一个参数。如下：

```
alert('hello world');
var sayHi = function() {
    alert('hello world');
};
setTimeout(function() {
    alert('hello world');
}, 100);
```

总之，若要提高代码性能，尽可能避免出现需要按照JS解释的代码。

5.性能的其它注意事项

原生方法更快——只要有可能，使用原生方法而不是自已用JS重写。原生方法是用诸如C/C++之类的编译型语言写出来的，要比JS的快多了。

switch语句较快——若有一系列复杂的if-else语句，可以转换成单个switch语句则可以得到更快的代码，还可以通过将case语句按照最可能的到最不可能的顺序进行组织，来进一步优化。
位运算较快——当进行数学运算时，位运算操作要比任何布尔运算或算数运算快。选择性地用位运算替换算数运算可以极大提升复杂计算的性能，诸如取模，逻辑与和逻辑或也可以考虑用位运算来替换。
3.最小化语句数

JS代码中的语句数量也会影响所执行的操作的速度，完成多个操作的单个语句要比完成单个操作的多个语句块快。故要找出可以组合在一起的语句，以减来整体的执行时间。这里列举几种模式

1.多个变量声明

```
// 避免
var i = 1;
var j = "hello";
var arr = [1,2,3];
var now = new Date();
```

```
// 提倡
var i = 1,
    j = "hello",
    arr = [1,2,3],
    now = new Date();
```

2.插入迭代值

```
// 避免
var name = values[i];
i++;

// 提倡
var name = values[i++];
```

3.使用数组和对象字面量，避免使用构造函数Array(),Object()

```
// 避免
var a = new Array();
a[0] = 1;
a[1] = "hello";
a[2] = 45;
```

```
var o = new Obejct();
o.name = "bill";
o.age = 13;

// 提倡
var a = [1, "hello", 45];
var o = {
    name : "bill",
    age : 13
};
```

4.优化DOM交互

在JS中，DOM无疑是最慢的一部分，DOM操作和交互要消耗大量时间，因为它们往往需要重新渲染整个页面或者某一个部分，故理解如何优化与DOM的交互可以极大提高脚本完成的速度。

1.最小化现场更新

一旦你需要访问的DOM部分是已经显示的页面的一部分，那么你就是在进行一个现场更新。之所以叫现场更新，是

因为需要立即（现场）对页面对用户的显示进行更新，每一个更改，不管是插入单个字符还是移除整个片段，都有一个性能惩罚，因为浏览器需要重新计算无数尺寸以进行更新。现场更新进行的越多，代码完成执行所花的时间也越长。

2.多使用innerHTML

有两种在页面上创建DOM节点的方法：使用诸如createElement()和appendChild()之类的DOM方法，以及使用innerHTML。对于小的DOM更改，两者效率差不多，但对于大的DOM更改，innerHTML要比标准的DOM方法创建同样的DOM结构快得多。

当使用innerHTML设置为某个值时，后台会创建一个HTML解释器，然后使用内部的DOM调用来创建DOM结构，而非基于JS的DOM调用。由于内部方法是编译好的而非解释执行，故执行的更快。