2011-6-20

第一单 加载和执行

1. 基本概念：

当浏览器在执行Js代码时，不能同时做其它任何事情。大多数浏览器都使用单一进程来处理用户界面更新和执行JS脚本。

无论JS是外连的还是内嵌的，页面的下载和渲染都必须停下来等待脚本执行完成，因为脚本执行过程可能会修改页面内容。

浏览器在解析HTML页面的过程中每遇到一个script标签，都会因为执行脚本而导致一定的延时。

浏览器在解析到body标签之前，是不会渲染页面的任何部分的。

把脚本放在head里，将会导致明显的延迟，通常表现为空白页。

由于脚本阻塞问题，推荐把脚本放在body底部。

1. 无阻塞脚本

关键：在页面加载完成后才加载JS代码。即在window对象的load事件触发后再下载脚本。

推荐的无阻塞模式：先添加动态加载所需的代码，然后加载初始化页面所需的剩下的代码。

* 1. defer属性：

defer属性指明不会修改DOM，因此代码会延迟执行,页面解析到script标签，开始下载，但不会执行，**直到DOM加载完成，在document的load事件触发前执行。**

但，只有IE4+ 和FF3.5+的浏览器支持此属性。

Eg:

<script type=”…” src=”…” defer></script>

<script type=”…” defer>…</script>

Eg:

<body>

<script type=”…” defer>

Alert(“defer”); //2

</script>

<script type=”…” >

Alert(“script”); //1

</script>

<script type=”…” >

Window.onload = function(){

Alert(“load”); //3

}

</script>

</body>

* 1. 动态脚本元素

用js动态创建script标签，并插入到页面中。

使用这种方法，返回的代码会立刻执行。即当script创建，为其赋了src后，就开始下载，append到页面时，就开始执行。（不知道这理解对否？？？）

可以通过捕捉script元素接收完成，来解决自动执行的问题。

Eg:

function loadScript( url, callback){

var script = document.createElement(“script”);

script.type = “text/javascript”;

if(script.readyState){ //IE

script.onreadystatechange =function(){

if(script.readyState == “loaded” || script.readyState == “complete”){

callback();

}

};

}else{ //other browser

Script.load = function(){

Callback();

}

}

script.src = url;

document.getElementByTagName(“head”)[0].appendChild(script);

}

* 1. XMLHttpRequest 脚本注入

这种方法优点：下载js代码，但代码不会立即执行，因为代码是在script标签之外返回的。

Eg:

var xhr = new XMLHttpRequest();

xhr.open(“get”,”file.js”,true);

xhr.onreadystatechange = function(){

if(xhr.status >= 200 && xhr.status < 300 || xhr.status == 304){

var script = document.createElement(“script”);

script.type = “text/javascript”;

script.text = xhr.responseText;

document.body.appendChild(script);

}

}

xhr.send(null);

* 1. 工

1. 工
2. 工

2011-6-20

第二章 数据访问

1. 基础知识

数据存储的位置，关系到代码执行过程中数据被检索到的速度。

数据存储共有4种方式:直接量、变量、数组项、对象成员。

大多数情况下，从一个直接量和一个局部变量中存取数据的性能差异是微不足道的。访问数组和对象成员代价高一些。

1. 作用域链和标识符解析

每个function都是一个Function 对象实例，拥有可以编程访问的属性和一系列不能通过代码访问而仅供Javascript引擎存取的内部属性， [[Scope]]属性。

函数每次执行，都会创建一个称为“运行期上下文”的内部对象，它定义了这个函数执行时的环境。执行完毕会销毁。

局部变量

eg:

function add(num1,num2){  
 var sum = num1 + num2;

Activation object

return sum;

window

this

}

add(1,5);

[1,5]

arguments

Scope chain

Var Total = add(1,5)；execution context

1

num1

5

num2

·

0

undefinded

sum

·

1

·

Scope chain

Global object

this

(function)

add

(object)

window

window

全局变量

document

(object)

undefinded

total

在这个运行期上下文中，先存的是局部变量，再存全局变量。函数执行过程中，每遇到一个变量，都会经历一次标识符搜索运行期上下文的过程，这个搜索过程影响了性能。标识符越深，读写速度越慢。

**总结：如果某个跨作用域的值在函数中被引用一次以上，就把它存到局部变量里。**

1. 改变作用域链
   1. with语句

eg:

function initUI(){

with(document){

links = getElementsByTagName(“a”);

…

}

}

当代码执行到with语句时，运行期上下文的作用域链临时被改变了，一个新的可变对象被创建。这个新的变量被推入作用域链的头部，把局部和全局的再次往下移了，访问代价加大了。

with variable object(临时变量)

(object)

body

Scope chain

2

·

1

·

·

0

Activation object（局部变量）

Global object（全局变量）

* 1. try catch语句

eg:

try{

….

}catch(e){

alert(e.message); //作用域链在此被改变

}

在执行到catch语句时，会把异常对象推入一个可变对象并置于作用域的头部。在catch代码块内部，函数所有局部变量将会放在第二个作用域链。可能通过精减catch里的代码（不访问可变对象外的变量），来减少对性能的影响。一个很好的模式是把错误委托给一个函数来处理。

* 1. 工

1. 动态作用域链

with ， try…catch..， eval()函数，这三者会创建动态作用域链。动态作用域链只存在于代码执行过程。

1. 闭包，作用域和内存

eg:

function assignEvents(){

var id = “111”;

document.getElementById(“save1”).onclick = function(event){ //这是一个闭包，也是一个函数

saveDocument(id);

}

}

闭包在assignEvents()函数执行时创建。

当assignEvents()函数被执行时，运行期上下文中的作用域中存了局部和全局变量。当运行到创建闭包时，闭包被创建，它的scope属性被初始化。

id

arguments

this

Activation object

1

·

0

Scope chain

·

assignEvents()

id

“111”

arguments

[]

window

this

Scope chain

·

Scope chain

·

0

1

·

saveDocument

(function)

assignEvents

(object)

document

(object)

window

window

this

Global object

(function)

Global object

this

window

window

(object)

document

(object)

assignEvents

(function)

saveDocument

(function)

Closure

·

[[Scope]]

通常来说，函数的活动对象会随同运行期上下文一同销毁。但引入闭包时，由于引用仍然存在于闭包的[[Scope]]属性中，因此激活对象无法销毁。

当闭包被执行时，一个运行期上下文被创建，它的作用域链与属性[[Scope]]中引用的两个相同的作用域链接对象同时被初始化，然后一个活动对象为闭包自身所创建。

为闭包所创建

(object)

event

[]

arguments

window

this

Activation object(closure)

Activation object(assignEvents)

·

1

·

0

Scope chain

·

2

window

[]

Closure

[[Scope]]

·

“111”

这闭包里，要访问id 和saveDocuemnt，这二个标识符存在于第二和第三个作用域对象里。**使用闭包最主要的性能关注点：要经常访问大量跨作用域的标识符。**

1. 原型

JS中的对象是基于原型的。原型是其它对象的基础，定义并实现了一个新对象必须包含的成员列表。

对象通过一个内部属性绑定到它的原型。在FF，Safari，Chrome浏览器中，这个属性为“\_proto\_”，对开发者可见，其它浏览器却不允许脚本访问此属性。

对象有二种成员类型：实例成员，原型成员。实例成员存在于对象实例中，原型成员则由对象原型继承而来。eg

var book = {

title: “….”, //title,

publisher: “…” //publisher这二者是实例成员

}

book.toString(); //toString()是原型成员

book prototype

null

\_proto\_

(function)

hasOwnProperty

(function)

isPrototypeof

publisher

…

title

…

\_proto\_

book

·

(function)

propertyIsEnumerable

(function)

toLocaleString

(function)

toString

(function)

valueOf

解析对象成员的过程同解析变量十分相似，当book.toString()被调用时，会从对象的实例开始搜索。一旦没有名为toString的实例成员，会继续搜索其原型对象，直到找到并执行。

1. 原型链

function Book(title, publisher){

this.title = title;

this.publisher = publisher;

}

Book.prototype.sayTitle = function(){…..};

var book1 = new Book(“”,””);

var book2 = new Book(“”,””);

book1.sayTitle();

book1.toString();

(function)

valueOf

(function)

toString

(function)

toLocaleString

(function)

propertyIsEnumerable

(function)

isPrototypeof

(function)

hasOwnProperty

null

\_proto\_

Book.prototype prototype

sayTitle

constructor

\_proto\_

publisher

…

title

·

\_proto\_

book2

…

publisher

…

title

·

\_proto\_

book1

…

Book.prototype

·

·

(function)

Book

prototype

·

book1的原型是Book.prototype，而Book.prototype的原型是Object。这个过程就是一个原型链。

book1，book2共享一个原型链，每个实例有各自的实例属性。

**注：在老的浏览器，像IE , FF3.5，每深入一层原型链都会拉回性能损失。搜索实例成员比从直接量或是局部变量中访问数据代价更高，再加遍历原型链带来的开销。**

1. 嵌套成员

对象成员可能包含其他成员，如window.location.href.

**对象成员嵌套的越深，访问速度就会越慢。location.href 要比window.location.href要快。**

**在大部分浏览器中，通过“.”操作和“[]”操作没有明显的区别，只有在Safari中，点符号始终保持要更快些。**

1. 小结

在函数中，如果要多次读取同一个对象属性，最佳的做法是将属性值保存到局部变量中。

访问直接量和局部变量的速度最快，相反，访问数组元素和对象成员相对较慢。

由于局部变量存在于作用域链的起始位置，因此访问局部变量比访问跨作用域变量更快。变量在作用域链中的位置越深，访问所需的时间就越长。由于全局变量总处在作用域链的最末端，因此访问速度也最慢。

避免使用with语句，它会改变运行期上下文作用域链。try-catch语句中的catch子句也会改变运行期上下文的作用域链。

嵌套的对象成员会明显影响性能，尽量少用。

属性或方法在原型链中的位置越深，访问它的速度也越慢。

可能通过把常用的对象成员、数组元素、跨域变量保存在局部变量中来改善JS的性能。

1. 工
2. 工
3. 工
4. 工
5. 工

2011-7-14

第三章 DOM编程

1. 浏览器中的DOM

文档对象模型DOM是个语言无关的，用于操作XML和HTML文档的应用程序API。它在浏览器中的接口却是用JS实现的。

浏览器中，通常把DOM和JS独立实现。在IE中，JS的实现名为Jscript，位于jscript.dll文件中。DOM的实现则在另一个库中，名为mshtml.dll。分离有利于其它语言共享DOM。

两个相互独立的功能只要通过接口彼此连接，就会产生消耗。把DOM和JS各自想象成二个岛屿，它们之间用收费桥梁连接，每次JS访问DOM，都要跨过此桥，交过路费，过的越多，交的钱越多。

1. DOM访问与修改

推荐：**减少访问DOM的次数，把运算尽量留在JS这一端处理。**

在innerHTML 和 DOM方法相比：除开最新版本的webkit内核（Chrome , Safari）外，二都性能上相关无几，innerHTML相对更快一些。webkit内核的浏览器，DOM的createElement方法更快些。

推荐：如果在一个对性能有着苛刻要求的操作中更新一大段HTML，推荐使用innerHTML，它在大多数浏览器中运行的更快些。但对于大多数日常操作而言，并没有太大的区别，所以更应该根据可读性、稳定性、团队习惯、代码风格来综合决定。

在大多数浏览器中，节点克隆都更有效率，但也不是特别明显。element.cloneNode()

1. HTML集合

document.getElementsByName(), document.getElementsByClassName(), document.getElementsByTagName(), document.images, document.links, document.forms, document.forms[0].elements

这些都返回了HTML集合对象，它类似数组，但不是数组。

HTML集合处于一种“实时状态”实时存在，当底层文档对象更新时，它会**自动**更新。

HTML集合一直与文档保持着连接，每次你需要最新信息时，都会重复执行查询过程，哪怕只是获取元素个数，也会重新查询。

由于遍历数组比遍历集合快，因此如果先将集合元素拷贝到数组中，访问它的属性会更快。但这会因额外的步骤带来消耗，会多遍历一次集合，需要认真评估下。

1. 访问集合元素时使用局部变量

当同一个DOM属性或方法需要多次访问时，最好使用一个局部变量缓存此成员。当遍历一个集合时，首要优化原则是把集合存储在局部变量中，把length缓存在循环外部，使用局部变量访问这些需要多次访问的元素。

eg:

function fun1(){

var coll = document.getElementsByTagName(“div”),

len = coll.length,

name = ‘’,

el = null;

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

el = coll[count];

name = el.nodeName;

name = el.nodeType;

….

}

return name;

}

1. 遍历DOM

eg1:

function testNextSibling(){

var el = document.getElementById(“mydiv”), ch = el.firstChild, name = ‘’;

do{

name = ch.nodeName;

}while(ch = ch.nextSibling);

return name;

}

eg2:

function testChildNodes(){

var el = document.getElementById(“mydiv”), ch = el.childNodes, len = ch.length, name = ‘’;

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

name = ch[i].nodeName;

}

return name;

}

在不同的浏览器中，这二种方法运行时间几乎相等。但在IE中，方法一比方法二快很多。推荐：**在性能要求极高时，在老版本的IE中更推荐用nextSibling方法来查找DOM节点。**

1. 重绘和重排

浏览器下载完页面中的所有组件（HTML标记，js，CSS，图片）后，会解析关生成两个内部数据结构，**DOM树**(表示页面结构)和**渲染树**（表示DOM节点如何显示）。

DOM树中的每一个需要显示的节点在渲染树中至少存在一个对应的节点（隐藏的DOM元素在渲染树中没有对应的节点）。

渲染树中的节点被称为帧（frames）或盒子(boxes)。一当DOM和渲染树构建完成，浏览器就开始显示器（绘制paint）页面元素。

当DOM的变化影响了元素的几何属性，浏览器会使渲染树中受到影响的部分失效，并重新构造渲染树，称为重排(reflow)，重排后，浏览器会重新绘制受影响的部分到屏幕中，称为重绘(repaint)。

**重绘和得排都是代价昂贵的，会导致web应用程序的UI反应迟钝。**

引起重排的有：添加或删除可见DOM元素、元素位置改变、元素尺寸改变（外边框，内边距，边框厚度，宽度，高度等属性）、内容改变、页面渲染器初始化、浏览器窗口尺寸改变。

1. 渲染树变化的排除与刷新

由于每次重排都会产生计算消耗，大多数浏览器通过队列化修改并批量执行来优化重排过程。获取布局信息的操作会导致列队刷新。如

offsetTop, offsetLeft, offsetWidth, offHeight

scrollTop, scrollLeft, scrollWidth, scrollHeight

clientTop, clientLeft, clientWidth, clientHeight

getComputedStyle( currentStyle in IE)

1. 最小化重绘和重排

应该合并多次对DOM和样式的修改，然后一交处理掉。

没用过这个属性

eg:

var el = document.getElementById(“mydiv”);

el.style.borderLeft = “1px”;

el.style.borderRight = “1px”; 🡪 el.style.cssText = “ border-left:1px; bprder-right:1px; padding:5px;”

el.style.padding = “5px”;

或者修改元素的class样式名，这样也可以批量修改样式。

当你需要对DOM元素进行一系列操作时，可以通过下列步骤减少重绘和重排次数：

1）、使元素脱离文档流；

2）、对其应用多重改变；

3）、把元素带回文档中；

这样，会在脱离和带回的步骤1和3时会产生重绘和重排。

脱离文档流的方法：

* 隐藏元素，应用修改，重新修改；
* 使用文档片断（document fragement）在当前DOM之外构建一个子树，再把它拷贝回文档；
* 将原始元素拷贝到一个脱离文档的节点中，修改副本，完成后再替换原始元素；

1. IE和:hover

从IE7开始，IE允许在任何元素（严格模式上）使用:hover，如果你有大量元素使用:hover，那么会降低响应速度，此问题在IE8中更明显。

1. 事件委托

当页面中存在大量元素，而且每一个都要一次或多次绑定事件处理器，这种情况下可能会影响性能。

每绑定一个事件处理器都是有代价的，它要么加重了页面负担（更多标签或更多JS），要么增加了运行期的执行时间。

事件绑定占用了处理时间，浏览器需要跟踪每个事件处理器，也会占用更多的内存。当这些工作结束时，这些事件处理器中的绝大部分都不再需要（因为并不是100%的按钮或是链接会被用户点击），因此很多工作没有必要。

事件委托基于这样一个事实：事件逐层冒泡并能被父级元素捕获。使用事件代理，只需给外层元素绑定一个处理器，就可以处理在其子元素上触发的所有事件。

eg:

<body>

<ul id = “menu”>

<li><a href=”….html”>menu #</a></li>

…..

</ul>

document.getElementById(“menu”).onclick = function(e){

e = e || window.event;

var target = e.target || e.srcElement;

var pageid, hrefparts;

if(target.nodeName != ‘A’){

return;

}

hrefparts = target.href.split(‘/’);

pageid = hrefparts[hrefparts.length - 1];

pageid = pageid.replace(“.html”,’’);

ajaxRequest(…..);

if(typeof e.preventDefault === ‘function’){

e.preventDefault();

e.stopPropagation();

}else{

e.returnValue = false;

e.cancelBubble =true;

}

}

解说：为元素a的父元素ul加了一个事件，判断事件是否来自于a元素，对不是来自于a的事件，一概不处理。处理完事情后，取消冒泡，阻止默认行为。如果用户禁用JS，a仍然会跳到相应的目的地，如果JS可用，运行JS的，阻止了默认的跳转。

1. 小结：

* 最小化DOM访问次数，尽可能在JS端处理。
* 如果需要多次访问某个DOM节点，请使用局部变量存储它的引用。
* 小心处理HTML集合，它实时连系着底层文档。需要经常访问集合，建议把它拷贝到一个数组中。
* 如果可能的话，使用更快的API，如querySelectorAll(), firstElementChild。
* 要留意重绘和重排，批量修改样式时，离线操作DOM树，使用缓存，减少访问布局信息的次数。
* 动画中使用绝对定位，使用拖放代理。
* 使用事件委拖来减少事件处理器的数量。

1. a

2011-7-19

第四章 算法和流程控制

1. 循环类型

for循环：

for(var I = 0 ; i < 10; i++){

//循环体

}

for循环由四部分组成：初始化、前测条件、后执行体、循环体。

当代码运行中遇到for时，先运行初始化代码，然后进入前测条件，如果前测条件为真，则运行循环体。循环体执行完后，后执行代码开始运行。var i=0 🡪 I < 10 🡪 循环体 🡪 i++

for循环初始化中的var语句创建的是一个函数级的变量，而不是循环级的。因此var I = 0， 在for()里定义和在循环体外，函数内定义是等价的。

while循环：

var I = 0;

while(I < 10){

//循环体

I ++;

}

while循环是最简单的前测循环，先计算前测条件，前测条件成真，运行循环体。for和while间可以互转。

do-while循环：

var I =0;

do{

//循环体

}while( I ++ < 10);

do-while是一种后测循环，循环体至少会运行一次，再去计算后测条件。

for-in循环：

for( var prop in object){

//循环体

}

它可以枚举任何对象的属性名。循环体每次运行时，prop变量被赋值为object的一个属性名（字符串），直到所有属性遍历完成才返回。所返回的属性包括对象实例属性以及从原型链中继承而来的属性。

1. 循环性能

在JS的四种循环中，for-in循环比其他几种明显要慢。

由于每次迭代操作会同时搜索实例或原型属性，for-in的每次迭代都会产生更多开销。最终是其他类型的1/7。除非你明确需要迭代一个属性数量未知的对象，否则请避免使用for-in。

不要使用for-in来遍历数组成员。

1. 减少迭代的工作量

如果一次迭代要花大量时间，多次的话，就更花大在量时间了，因此推荐限制循环体中耗时的数量。

在JS中，倒序循环会略微提升性能，只要你排除那些额外操作的影响。倒序循环是编程语言中一种通用的性能优化方法。

eg:

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

process(items[i]);

}

每次运行循环体会产生如下操作：

1. 一次控制条件中的属性查找(items.length)。
2. 一次控制条件中的数值大小比较(I < itmes.length)；
3. 一次控制条件结果是否为true的比较（I < items.length == true）；
4. 一次自增操作(i++)
5. 一次数组查找(items[i])；
6. 一次函数调用(process(items[i]))；

优化的方向就是尽量减少每次迭代的工作量，减少每次迭代的操作。

1. 减少迭代次数

**当循环复杂度为O(n)时，减少每次迭代的工作量是最有效的方法。当复杂度大于O(n)，建议关重减少迭代次数。**

最广为人知的一种限制循环迭代次数的模式是达夫设备(Duff’s Device)。

eg: (这个例子有些没想明白)

var iterations = Math.floor(items.length / 8 ), startAt = items.length % 8, I = 0;

do{

switch(startAt){

case 0: process(items[i++]);

case 7: process(items[i++]);

case 6: process(items[i++]);

case 5: process(items[i++]);

case 4: process(items[i++]);

case 3: process(items[i++]);

case 2: process(items[i++]);

case 1: process(items[i++]);

}

startAt = 0;

}while( -- iterations);

达夫设备的基本理念是： 每次循环最多可调用8次process();循环迭代的次数为总数除以8。变量startAt用来存放余数，表示第一次循环中应调用多少次process()。如果是12，那么第一次循环会调用process()4次，第二次循环会调用process()8次。用二盜循环替代了12次循环。

1. 基于函数的迭代

forEach()，此方法遍历一个数组的所有成员，并在每个成员上执行一个函数，此函数以参数的方法传递给forEach

eg:

arr.forEach(function( value, index ,array){ // value🡪当前数组项的值、index🡪索引 array🡪 数组本身

…

});

FF，Chrome, Safari中，forEach被原生支持。

对每个数组项调用外部方法所带来的开销是速度慢的主要原因。基于循环的迭代比基于函数的迭代要快8倍。

1. if-else对比switch

从代码易读性来说，条件少时，用if-else，条件较多时，用switch。

大多数情况下switch比if-else运行得要快，但只有条件数量很大时才快的明显。大多数语言，对switch语句的实现都采用了branch table(分支表)索引来进行优化。在JS中，switch比较值时，使用全等操作符，不会发生类型转换的损耗。

1. 优化if-else

优化的目的：最小化到达正确分支前所需要判断的条件数量。

最简单的方法：确保最可能出现的条件放在首位。if-else语句中，总是按照最大概率到最小概率的顺序排列。

另一方法：把if-else组织成一系列嵌套的if-else语句。使用单个庞大的if-else通常会导致运行缓慢。

1. 查找表

当有大量离散值需要测试时，if-else和switch都比使用查找表慢的多。

JS中可以使用数组和普通对象来构造查找表。

eg:

switch(value){

case 1:

…

case 1:

…

…..

}

🡪

var res = [res0, res1, res2, res3…];

return res[value];

1. 调用栈限制——递归

JS引擎支持的递归数量与JS调用栈大小直接相关。只有IE例外，它与系统空闲内在有关，其它浏览器都有固定数量的调用栈限制。

FF3.0:3000

FF3.5:3000

Chrome1:21837

Chrome2:21837

IE7:1789

IE8:2232

Opera:10000

Safari3.2:500

Safari4:37448

调用栈溢出错误，每个浏览器也各不一样。

1. 迭代——递归

任何递归能实现的算法也可以用迭代实现。

使用优化后的循环替代长时间运行的递归函数可以提升性能，因为运行一个循环比反复调用 一个函数的开销要少得多。

1. Memoization

Memoization：技术是一种避免重复工作的方法，它缓存前一个计算结果供后续计算使用，大大减少了同一个函数的重复调用。

eg:

function factorial(n){

if(n==0){

return 1;

}else{

return n\* factorial(n-1);

}

}

var fac6 = factorial(6);

var fac5 = factorial(5);

var fac4 = factorial(4);

//factorial函数调用 了18次

🡪

function memfactorial(n){

if(!memfactorial.cache){

memfactorial.cache = {

“0”:1,

“1”:1

};

}

if(!memfactorial.cache.hasOwnProperty(n)){

memfactorial.cache[n] = n\* memfactorial(n-1); //没有缓存值的话，把新的值缓存进去

}

return memfactorial.cache[n];

}

var fac6 = memfactorial (6);

var fac5 = memfactorial (5);

var fac4 = memfactorial (4);

// memfactorial 函数调用了8次

🡪

function memorize(fun, cache){ //fun 递归函数, cache指定的缓存，也可以不传入此参数

cache = cache || {}; //不传入指定的缓存，则新建一个

var shell = function(arg){

if(!cache.hasOwnProperty(arg)){

cache[arg] = fun(arg);

}

return cache[arg];

};

return shell;

}

var memfactorial = memorize(factorial,{“0”:1,”1”:1});

var fac6 = memfactorial(6);

var fac5 = memfactorial(5);

var fac4 = memfactorial(4);

// factorial函数调用 了8次

1. 小结：

JS是解释性语言，它无需编译，而是将代码以字符串的形式交给JS引擎来执行，因此代码性能在一定程度上取决于客户端浏览器的JS引擎。

for, while , do-while循环性能特性相似，所以没有一种循环类型明显快于或慢于其他类型。

避免使用for-in循环，除非你需要遍历一个属性数量未知的对象。

改善循环性能的最佳方式是减少每次迭代的运算量和减少循环迭代次数。

通常来说，switch总比if-else快，但并不总是最佳解决方案。

在判断条件较多时，使用查找表比if-else和switch更快。

浏览器调用栈大小限制了递归算法在JS中的应用，栈溢出错误会导致其他代码中断运行。

如果你遇到栈溢出错误，可将方法改为迭代算法，或使用Memoization来避免重复计算。

1. 工
2. 工

2011-7-25

第五章 字符串和正则表达式

* 1. 字符串连接—— + 和 +=

除<=IE7 浏览器外，都对连接字符串方法进行了优化。

str += “one” + “two”;

1. 在内存中创建一个临时字符串；
2. 连接后的字符串“onetwo”被赋值给该临时字符串；
3. 临时字符串与str当前的值连接；
4. 结果赋值给str;

str += “one”;

str +=”two”;

避免了生产临时字符串，可提升10%--40%

str = str + “one” + “two”;

赋值表达式由str开始作为基础，每次给它附加一个字符串，由左向右依次连接，避免了临时字符串。如果换成 str = “one” + str + “two” 等 str不在第一个位置，都将生成临时字符串。

除IE外，其它浏览器都将会尝试为表达式左侧的字符串分配更多的内存，然后简单地将第二个字符串拷贝到它的末尾。

IE8中，连接字符串只是记录现有的字符串的引用来构造新字符串。在最后时刻（当你真正要使用连接后的字符串时），字符串的各个部分才会逐个拷贝到一个新的“真正的”字符串中，然后用它取代先前的字符串引用，所以并非每次使用字符串时都发生合并操作。

<=IE7中，每连接一对字符串都要把它复制到一块新分配的内存中。str + = s1+s2 比 str = str + s1 +s2 更快，特别是str是个长字符串时。

* 1. 数组项连接

在大多数浏览器中，数组项连接比其它字符串方法更慢，但在IE7及更早版本中，却是合并大量字符串唯一高效的途径。

var str = “I’m a thirty-five character string”, newStr = “”, appends = 5000;

while(appends --){

newStr += str;

}

🡪

var str = “I’m a thirty-five character string ”, strs = [], newStr, appends = 5000;

while(appends --){

strs[strs.length] = str;

}

newStr = str.join(“”);

随着加大appends的值，在IE7里，第一种耗时越来越长，第二种方法却有了戏剧性的性能提升。

* 1. String.prototype.concat

这种原生方法能接收任意数量的参数，可以附加一个字符串，或附加多个字符串，或是整个字符串数组。

eg:

str = str.concat(s1);

str = str.concat(s1,s2,s3);

str = String.concat.apply(str,sArr);

在多数情况下，使用concat比使用简单的+ 和 += 稍慢，特别是在IE，Opera 和 Chrome中慢的更明显。它也潜伏着空难性的性能问题。

* 1. 正则表达式工作原理

一个正则表达式的基本步骤：

* + 1. 编译

当你创建了一个正则表达式对象（使用正则直接量或是RegExp构造函数），浏览器会验证你的模式，然后把它转化为一个原生代码程序，用于执行匹配工作。

**如果你把正则对象赋值给一个变量，就能避免重复执行此步骤。**

* + 1. 设置初始位置

当正则类进入使用状态，首先要确定目标字符串的起始搜索位置。它是字符串的起始字符，或是由正则表达式的lastIndex属性指定的，但是当它从第四步返回到这里时（由于尝试匹配失败），此位置则在最后一次匹配的起始位置的下一位字符的位置上。

浏览器厂商优化正则表达式引擎的方法是，通过提前决定跳过一些不必要的步骤，来避免大量无意义的工作。如，如果正则表达式由^开始，IE和Chrome通常会判断字符串的起始位置能否匹配，如果匹配失败，那么可以避免愚蠢地搜索后续位置。如，匹配第三个字母是X的字符串，一个聪明的做法是先找到X，然后再将起始位置退回两个字符（最新版的Chrome就包含了这项优化）。

* + 1. 匹配每个正则表达式字元

一旦正则表达式知道开始位置，它会逐个检查文本和正则表达式模式。当一个特定的字元匹配失败时，正则表达式会试着回溯到之前尝试匹配的位置上，然后尝试其他可能的路径。

* + 1. 匹配在成功或失败

如果在字符串当前的位置发现了一个完全匹配，那么正则表达式宣布匹配成功。如果正则表达式所有的可能路径都没匹配到，正则表达式引擎会退回到第二步，然后从下一个字符得新尝试。只有当字符串的每个字符（以及最后一个字符吕后面的位置）都经历这个过程后，还没有成功匹配，那么正则表达式就宣布彻底匹配失败。

* + 1. 工
  1. 理解回溯

当正则表达式扫描目标字符串时，它从左到右逐个测试其组成部分，看是否能找到匹配项。对每一个量词和分支都必须决定接下来如何处理。如果遇到量词（如\*,+, ?,{2}），正则表达式必须决定何时尝试匹配更多的字符，如遇到分支时（来自｜），必须从可能的选项中选一个做尝试。

分支与回溯：

eg:

/h(ello|appy) hippo/.test(“hello there, happy hippo”);

1)、找h：找到hello 中的h；

2)、找分支中的ello 或 appy : 找到hello中的ello ；

3)、找后续中的空格：找到hello后的空格；

4)、找后续中的hippo： hello 后是 t，所以此次匹配失败；

5)、回溯到最近的决策点（匹配到首字母h后的位置）：在第二步中，匹配了分支的ello，这次，开始匹配分支的appy，但hello后是ello，失败；

6)、开始第一步，找h：从hello的h后的e开始，找h，找到happy中的h后，找到了h；

7)、开始第二步：匹配分支，这次找到了分支中的appy；

8)、开始第三步：找到了空格

9)、开始第四步：找到了hippo，匹配成功。

**总结：一个字母一个字母查找，遇到分支，记录此点，如果此次匹配不成功，回溯到此点，换另外一个条件再次匹配。分支的所有可能都未有匹配成功，则进入下一个字母。（自己总结的）**

重复与回溯：

var str = “<p>Para 1.</p” + “<img src=’smiley.jpg’>” + “<p>Para 2.</p>” + “<div>Div.</div>”;

/p.\*<\/p>/i.test(str);

这个正则表达式一开始就匹配了字符串的前三个字符<p>，然后匹配“.”。“.”能匹配除换行符以外的任意字符，贪婪量词“\*”表示重复零次或多次。因为目标字符串中没有换行符，那么它会吞并其他所有字符。于是，正则表达式尝试匹配“<”。它在字符串的末尾处匹配失败，因此正则表达式每次回溯一个字符，继续尝试匹配”<”，直到回溯至<div>标签处的“<”。然后开始尝试匹配“\/”，匹配成功。然后匹配p，匹配失败。正则表达式继续回溯，并重复这一过程，直到最后匹配第二个”</p>”。匹配成功返回，扫描的范围从第一个P到第二个P的尾部。这可能并不是你想要的结果。

可以把贪婪符\*，换成惰性（非贪婪）量词，即“\*?”，以匹配单个p段落。

惰性量词的回溯工作以相反的方式进行。当正则表达式匹配到“\*?”时，它先尝试全部跳过并匹配接下来的“</p>”。因为“\*?”会重复匹配前一个字符零次或多次，次数尽可能地少（书上写的有些费话，总之惰性量词，会尽可能少地去匹配“.”，即非换行符，先从0次开始，也就是它希望下一个字符就是”<”，很不幸，不是，那就开始希望中间有1个非换行符的字符，然后就是“<”也不是，如此一个个增加中间的匹配非换行符的个数，直到找到第一个“>”，匹配成功，返回）。

* 1. 回溯失控

书讲得实在是太抽象了，看不懂P91，以后回来看

* 1. 工
  2. 工
  3. 工
  4. 工
  5. 工
  6. 工

2011-7-26

第六章 快速响应的用户界面

* 1. 浏览器UI线程

共用于执行JS和更新用户界面的进程通常称为“浏览器UI线程”。

UI线程的工作基于一个简单的队列系统，任务会被保存到队列中直到进程空闲。一旦空闲，队列中的下一个任务就被重新提取出来并运行。这些任务要么是JS代码，要么是UI更新，包括重绘和重排。

这个进程中，最有趣的是每一次输入（包括响应用户事件，自执行JS代码）可能会导致一个或多个任务被加入到队列。

eg:

<button onclick=”fun1();”>Click</button>

function fun1(){

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

div.innerHTML = “clicked!”;

document.body.append(div);

}

用户点击button时，UI线程里有二个任务：按钮的UI更新（点击后，按钮外观会改变以表示它被点击了）， JS代码块。当执行JS代码块时，又会往UI线程里加一个UI更新的任务（往body里加一个div）。

用户所有的交互都会立刻触发UI更新。如果用户试图在任务运行期间与页面交互，不仅没有即时更新UI，甚至可能新的UI更新任务都不会被创建入队列。**大多数浏览器在JS运行时会停止把新任务加入UI线程中**。

* 1. 浏览器限制

浏览器限制了JS任务的运行时间，分二种：调用栈大小限制（回调函数里用到），长时间运行脚本限制。后者有时被称长时间运行脚本定时器或失控脚本定时器，原理是浏览器会记录一个脚本的运行时间，并在达到一定限制时终止它。一种是记录执行语句的数量，一种是记录脚本执行的总时长。

不同浏览器检测长时间运行脚本的方法略有不同：

* IE4开始，设置默认限制为500万条语句，此限制放在windows注册表中
* FF默认限制为10秒，此限制记录在浏览器配置设置中（可通过地址栏里输入about:config访问），键名dom.max\_script\_run\_time
* Safari的默认限制为5秒，该限制无法更改，但可以通过Develop菜单选择Disable Runaway JavaScript Timer来禁用定时器
* Chrome没有单独的长运行脚本限制，替代做法是依赖其通用崩溃检测系统来处理此类问题
* Opera没有长运行脚本限制，它会继续执行JS代码直到结束，鉴于Opera的架构，脚本运行结束时不会导致系统不稳定
* 工

从开发人员角度来看，没办法改变一个长时间运行脚本对话框的外观，你检测不到它，因此不能用它来判断可能出现的问题。

* 1. 多久长算“长久”

你的JS代码运行的持续时间应远远小于浏览器的限制。

单位个JS操作花费的总时间（最大值）不应该超过**100毫秒**。

各个浏览器的行为大致相同。当脚本执行是，UI不随用户交互而更新。执行时间段内用户交互行为所引发的JS任务被加入队列中，并在最初的JS任务完成后依次执行。而这段时间内由用户交互行为引发的UI更新会自动跳过，因为页面中的动态变化部分会被优先考虑。因此，在一个脚本运行期间点击一个按钮，将无法看到它被按下的样式，尽管它的onclick事件处理器会被执行。

IE会控制用户交互行为触发的JS任务，因此它会识别连续二次的重复动作，例如，当有脚本运行时点击一个按钮四次，最终按钮的onclick事件处理器只被调用两次。

* 1. 使用定时器让出时间片段
  2. 工
  3. 工
  4. 工
  5. 工
  6. 工
  7. 工
  8. 工
  9. 组合继承

代码：



book

Book

·

prototype

(function)

valueOf

(function)

toString

(function)

toLocaleString

(function)

propertyIsEnumerable

(function)

isPrototypeof

(function)

hasOwnProperty

null

\_proto\_

Book.prototype prototype

sayTitle

constructor

\_proto\_

publisher

title

\_proto\_

publisher

title

\_proto\_

·

Book.prototype

“book”

·

“000…”

·

(function)

nodebook

·

“nodejs”

“abc ..”

JSBooks.prototype

Array

·

prototype

authors

Function

sayAuthors

undefined

title

r

undefined

publisher