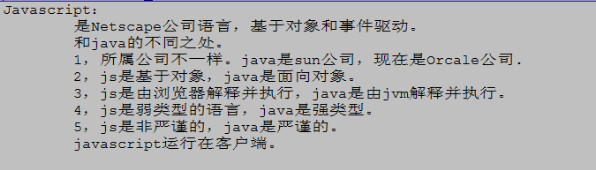
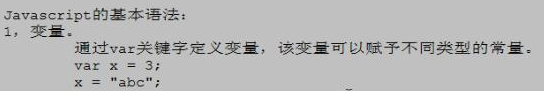
Javascript语法与基本操作

1. **Java与javascript的区别**



1. **变量**



**变量名必须以一个字母、下划线、美元符号开头，后面可以接数字等。**

1. **布尔型boolean**

Js中的boolean型，对于空字符串、NaN值、null、undefined、数值0都被解析为false，其他都为true

1. **Undefined类型**

在IE8及以下，undefined是可以被改变值的。

**undefined** = **true**;

**if**(**undefined**){

alert("fff");

}

这段代码在IE8及以下会弹出对话框，内容为fff

在火狐和谷歌最新版本中，如上的代码是不会弹出对话框的，不过下面代码也是可以弹出对话框的。注意区分

**varundefined** = **true**;

**if**(**undefined**){

alert("fff");

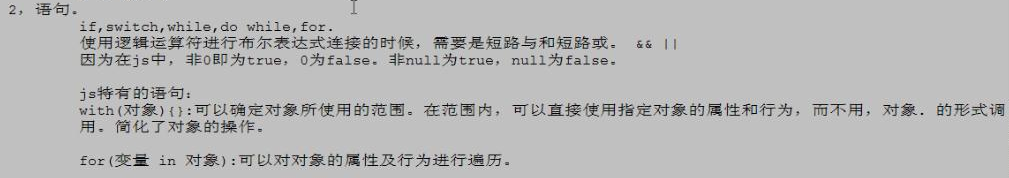
}

**总结：因此在使用if判断时，不建议使用undefined进行判断。**

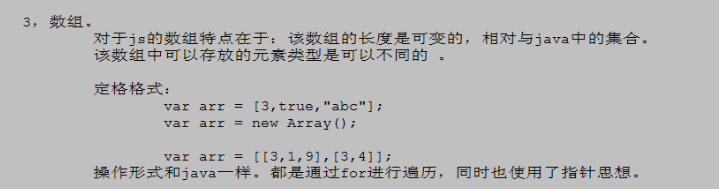
1. **小数的直接量声明**

在js中.1与0.1的意思是一样的

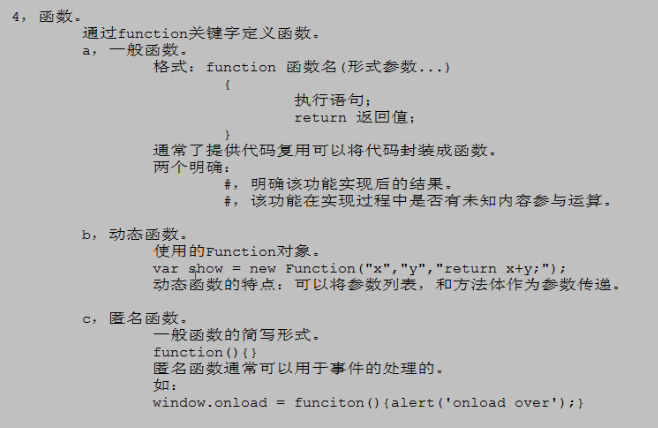
1. **语句**

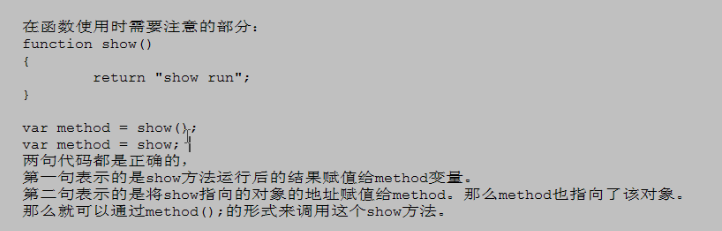


1. **数组**

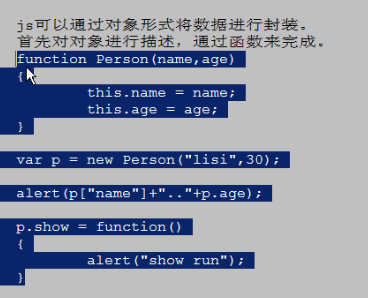


1. **函数**





1. **定义javascript对象**



**var** user = {

"name":"gh",

"handleNum":**function**(val){

alert(val);

},

"date":**function**(){

alert(**new** Date());

}

};

$(**function**(){

alert(user.name);

user.handleNum(10);

user.date();

//false

alert(user.hasOwnProperty("date01"));

//true

alert(user.hasOwnProperty("date"));

});

1. **数组和对象的创建——方式的对比**

**推荐的方法：**使用直接量创建。优势——速度更快，代码更简洁

//create an object

**var** myObject = {

name: "Nicholas",

count: 50,

flag: **true**,

pointer: **null**

};

//create an array

**var** myArray = ["Nicholas", 50, **true**, **null**];

**不推荐的方式：**

//create an object

**var** myObject = **new** Object();

myObject.name = "Nicholas";

myObject.count = 50;

myObject.flag = **true**;

myObject.pointer = **null**;

//create an array

**var** myArray = **new** Array();

myArray[0] = "Nicholas";

myArray[1] = 50;

myArray[2] = **true**;

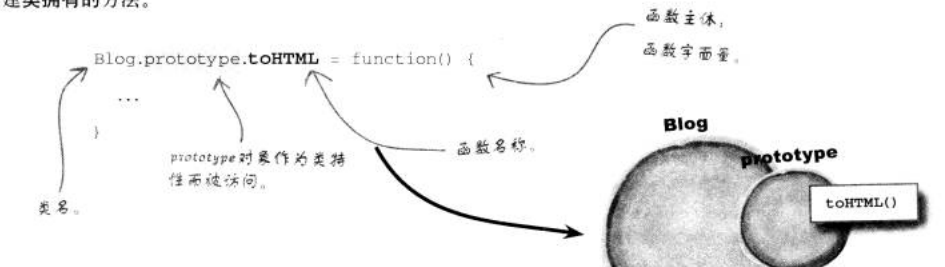
myArray[3] = **null**;

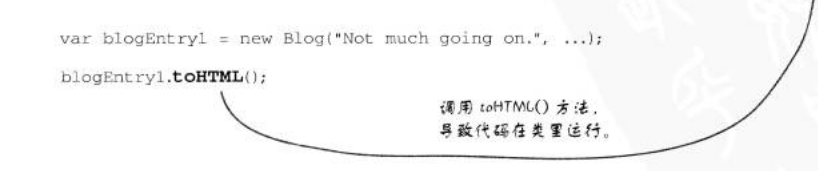
1. **Javascript中定义对象的方法**

**实例方法：**



**类方法：**





1. **Prototype概念**

**原型法设计模式：**在.Net中可以使用clone()来实现原型法原型法的主要思想是，现在有1个类A,我想要创建一个类B,这个类是以A为原型的,并且能进行扩展。我们称B的原型为A。

**javascript的方法可以分为三类：**a 类方法、b 对象方法、c 原型方法

例子：

**function** People(name)

{

**this**.name=name;

//对象方法

**this**.Introduce=**function**(){

alert("My name is "+**this**.name);

};

}

//类方法

People.Run=**function**(){

alert("I can run");

};

//原型方法

People.prototype.IntroduceChinese=**function**(){

alert("我的名字是"+**this**.name);

};

//测试

**var** p1=**new** People("Windking");

p1.Introduce();

People.Run();

p1.IntroduceChinese();

**obj1.func.call(obj)方法 ：**意思是将obj看成obj1,调用func方法。好了，下面一个一个问题解决： prototype是什么含义？

javascript中的每个对象都有prototype属性，Javascript中对象的prototype属性的解释是：返回对象类型原型的引用。

A.prototype = new B();

理解prototype不应把它和继承混淆。A的prototype为B的一个实例，可以理解A将B中的方法和属性全部克隆了一遍。A能使用B的方 法和属性。这里强调的是克隆而不是继承。可以出现这种情况：A的prototype是B的实例，同时B的prototype也是A的实例。

先看一个实验的例子：

**function** baseClass()

{

**this**.showMsg = **function**()

{

alert("baseClass::showMsg");

};

}

**function** extendClass()

{

}

extendClass.prototype = **new** baseClass();

**var** instance = **new** extendClass();

instance.showMsg(); // 显示baseClass::showMsg

我们首先定义了baseClass类，然后我们要定义extentClass，但是我们打算以baseClass的一个实例为原型，来克隆的extendClass也同时包含showMsg这个对象方法。extendClass.prototype = new baseClass()就可以阅读为：extendClass是以baseClass的一个实例为原型克隆创建的。那么就会有一个问题，如果extendClass中本身包含有一个与baseClass的方法同名的方法会怎么样？

下面是扩展实验2：

**function** baseClass()

{

**this**.showMsg = **function**()

{

alert("baseClass::showMsg");

};

}

**function** extendClass()

{

**this**.showMsg =**function** ()

{

alert("extendClass::showMsg");

};

}

extendClass.prototype = **new** baseClass();

**var** instance = **new** extendClass();

instance.showMsg();//显示extendClass::showMsg

实验证明：函数运行时会先去本体的函数中去找，如果找到则运行，找不到则去prototype中寻找函数。或者可以理解为prototype不会克隆同名函数。那么又会有一个新的问题：如果我想使用extendClass的一个实例instance调用baseClass的对象方法showMsg怎么办？

答案是可以使用call:

extendClass.prototype = **new** baseClass();

**var** instance = **new** extendClass();

**var** baseinstance = **new** baseClass();

baseinstance.showMsg.call(instance);//显示baseClass::showMsg

这里的baseinstance.showMsg.call(instance);阅读为“将instance当做baseinstance来调用，调用它的对象方法showMsg”好了，这里可能有人会问，为什么不用baseClass.showMsg.call(instance);这就是对象方法和类方法的区别，我们想调用的是baseClass的对象方法。

最后，下面这个代码如果理解清晰，那么这篇文章说的就已经理解了：

**function** baseClass()

{

**this**.showMsg = **function**()

{

alert("baseClass::showMsg");

};

**this**.baseShowMsg = **function**()

{

alert("baseClass::baseShowMsg");

};

}

baseClass.showMsg = **function**()

{

alert("baseClass::showMsg static");

};

**function** extendClass()

{

**this**.showMsg =**function** ()

{

alert("extendClass::showMsg");

};

}

extendClass.showMsg = **function**()

{

alert("extendClass::showMsg static");

};

extendClass.prototype = **new** baseClass();

**var** instance = **new** extendClass();

instance.showMsg(); //显示extendClass::showMsg

instance.baseShowMsg(); //显示baseClass::baseShowMsg

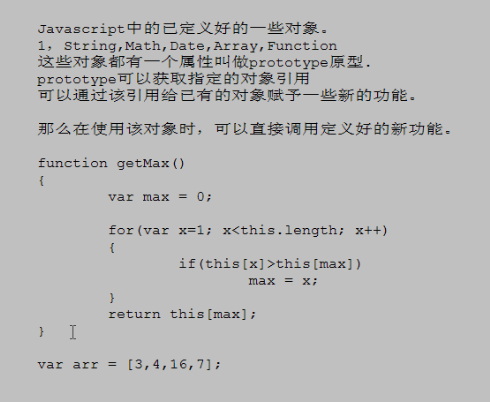
instance.showMsg(); //显示extendClass::showMsg

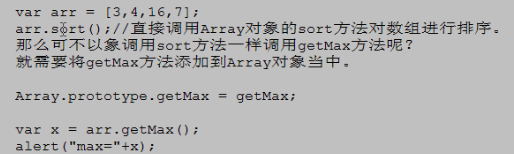
baseClass.showMsg.call(instance);//显示baseClass::showMsg static

**var** baseinstance = **new** baseClass();

baseinstance.showMsg.call(instance);//显示baseClass::showMsg

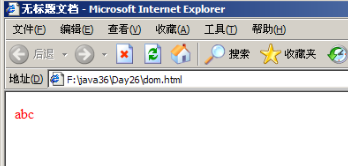
1. **利用prototype属性为对象赋予新功能**







**效果如下所示：**



**方法2：**

$(**function**(){

jQuery.fn.abc();

});

/\*\*

\*为jQuery.fn添加扩展abc方法。

\*/

jQuery.fn = jQuery.prototype = {

abc:alert('abc')

};

1. **Js中对象继承**

以下是打算让类B继承类A，并且扩展自己的方法

//父类A

**var** A = **function**() {

};

A.prototype = {

sayHello: **function**() {

alert("sayHello A");

}

};

//子类B

**var** B = **function**() {

};

//先添加父类(A)成员也就是子类B继承A的写法

Object.extend(B.prototype, A.prototype);

//继承的另外一种写法

B.prototype = A.prototype;

**var** b = **new** B();

b.sayHello();

//继承的另外一种写法

**var** bencode = **new** A();

bencode.sayHello();

//子类B扩展自己的方法

Object.extend(B.prototype, { // 再添加B类成员, 如果是同名，则覆盖，行为符合 "多态"

sayHi: **function**() {

alert("sayHi B");

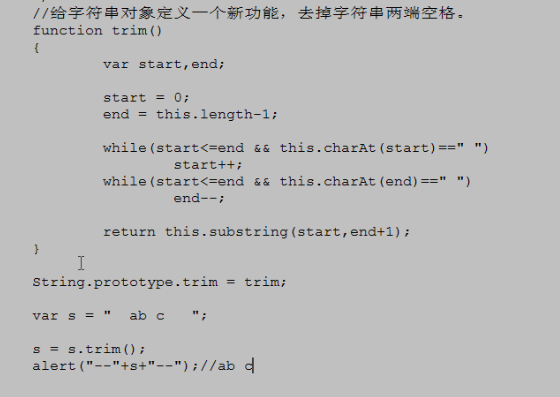
}

});

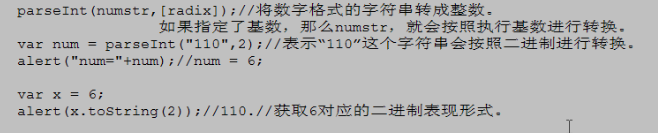
**var** b = **new** B();

b.sayHi();

1. **去掉字符串的两端空格**



1. **进行进制转换**



1. **Js数据类型**

**Text:**文本数据只是一串字符，字符串也常用于称呼文本类的数据，在javascript里，文本数据常出现在双引号（””）或者单引号（’’）中。

**Number：**数字用于存储质量、物品数量等数值数据。Javascript的数字可能是整数或浮点数。

**Boolean：**布尔只有两种可能情况——成立或者不成立。

1. **页面动态加载js文件**

**注：动态加载js文件可以优化页面的性能**

**方法1：**

$(**function**(){

//test。js文件与本js文件在同一个目录下

loadScript("js/test.js", callback);

});

/\*\*

\*此方法用于动态加载js文件，动态加载js文件有利于页面性能非优化，在使用的时候再去加载js文件

\***@param**url动态加载的js文件的url

\***@param**callback回调函数

\*/

**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"){

script.onreadystatechange = **null**;

callback();

}

};

} **else** { //Others

script.onload = **function**(){

callback();

};

}

script.src = url;

document.getElementsByTagName("head")[0].appendChild(script);

}

/\*\*

\*回调函数，当js加载完成所触发的函数

\*/

**function** callback(){

alert("callback");

}

**方法2：使用lazyload小框架**

**网址：https://github.com/rgrove/lazyload/**

<!DOCTYPEhtmlHTMLPUBLIC"-//W3C//DTD HTML 4.01 Transitional//EN">

<html>

<head>

<title>Insert title here</title>

<metahttp-equiv=*"content-type"*content=*"text/html; charset=UTF-8"*/>

</head>

<body>

<divclass=*"content"*>

</div>

<scripttype=*"text/javascript"* charset=*"utf-8"*src=*"js/lazyload.js"*></script>

<scripttype=*"text/javascript"*charset=*"utf-8"*>

LazyLoad.js([ "jquery-1.6.1.min.js", "js/test.js", "js/jsTest.js", "js/yui-min.js",

"js/jquery.easyui.min.js" ], **function**() {

//进行一些初始化的操作

$('.content').text("nihao");

});

</script>

</body>

</html>

Test.js文件

$(**function**(){

});

**方法3：使用LABjs框架**

**网址：http://labjs.com/**

<!DOCTYPEhtmlHTMLPUBLIC"-//W3C//DTD HTML 4.01 Transitional//EN">

<html>

<head>

<title>Insert title here</title>

<metahttp-equiv=*"content-type"*content=*"text/html; charset=UTF-8"*/>

</head>

<body>

<divclass=*"content"*>

</div>

<scripttype=*"text/javascript"*charset=*"utf-8"*src=*"js/LAB.js"*></script>

<scripttype=*"text/javascript"*charset=*"utf-8"*>

//下面代码的意思是等等jquery-1.6.1.min.js文件加载完后再去加载test.js文件，这样保证顺序性

$LAB.script("jquery-1.6.1.min.js").wait().script("js/test.js")

.wait(**function**() {

alert("初始化操作");

});

</script>

</body>

</html>

Test.js文件

$(**function**(){

$('.content').text("nihao");

});

1. **闭包的概念、应用以及注意事项**

**概念**：通俗讲闭包就是能够读取其他函数内部变量的函数。闭包就是将函数内部和函数外部连接起来的一座桥梁。

另外一种解释：当内部函数在定义它的作用域的外部被引用时,就创建了该内部函数的闭包，如果内部函数引用了位于外部函数的变量,当外部函数调用完毕后,这些变量在内存不会被释放,因为闭包需要它们。

**基本代码结构：**

/\*\*

\*闭包产生的条件是：父函数a内部的b函数引用了父函数a的局部变量，并且a函数将b函数作为自己的值返回，当一个全局变量c引用到a函数时，如果其他地方调用了c全局变量

\*那么就形成了一个基本的闭包，此处我们称b函数就是一个闭包

\***@returns**{Function}

\*/

**function** a(){

**var** n = 0;

**function** b(){

n = n + 1;

alert(n);

}

**return** b;

}

//将a赋给一个全局变量c，因此GC回收机制不会回收变量c，因此函数b也不会被回收，b所依赖的a函数也不会被回收，这样就产生了闭包

**var** c = a();

//第一次运行n=1,c运行完毕时，由于c是全局变量，因此不会被回收

c();

//由于c没有被回收，因此第二次运行时是在第一次的基础上再加1，运行结果是n=2

c();

//将c置为null释放闭包所占用的资源

c = null;

**应用：**闭包可以用在许多地方。它的最大用处有两个，一个是前面提到的可以读取函数内部的变量，另一个就是让这些变量的值始终保持在内存中。

闭包的另一个重要用途是实现面向对象中的对象。

**function** Person(){

**var** name = "default";

**return** {

getName : **function**(){

**return** name;

},

setName : **function**(newName){

name = newName;

}

} ;

};

**var** john = Person();

print(john.getName());

john.setName("john");

print(john.getName());

**var** jack = Person();

print(jack.getName());

jack.setName("jack");

print(jack.getName());

**注意事项**：

由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包，否则会造成网页的性能问题，在IE中可能导致内存泄露。解决方法是，在退出函数之前，将不使用的局部变量全部删除。 （就是将全局变量置为null）

闭包会在父函数外部，改变父函数内部变量的值。所以，如果你把父函数当作对象（object）使用，把闭包当作它的公用方法（Public Method），把内部变量当作它的私有属性（private value），这时一定要小心，不要随便改变父函数内部变量的值。

1. **作用域链**

**概念：**简单来说,作用域链就是函数在定义的时候创建的,用于寻找使用到的变量的值的一个索引,而他内部的规则是,把函数自身的本地变量放在最前面,把自身的父级函数中的变量放在其次,把再高一级函数中的变量放在更后面,以此类推直至全局对象为止.当函数中需要查询一个变量的值的时候,js解释器会去作用域链去查找,从最前面的本地变量中先找,如果没有找到对应的变量,则到下一级的链上找,一旦找到了变量,则不再继续.如果找到最后也没找到需要的变量,则解释器返回undefined.

1. **判断对象是否有某属性，hasOwnProperty与in**

$(**function**(){

alert(book.hasOwnProperty("title")); //true

alert(book.hasOwnProperty("toString")); //false

alert("title"**in** book); //true

alert("toString"**in** book); //true

/\*\*

\*注：hasOwnProperty方法只会搜索本实例自己的属性，而不会搜索原型对象的属性，因此第二个会为false，而in不同，它会去搜索book所继承的原型对象的属性，因此

\*第二个也是true。注意此中的区别

\*/

});

**var** book = {

title: "High Performance JavaScript",

publisher: "Yahoo! Press"

};

1. **成员嵌套的优劣**

由于对象成员可能包含其它成员，例如不太常见的写法window.location.href这种模式。每遇到一个点号，JavaScript 引擎就要在对象成员上执行一次解析过程。

结果并不奇怪，成员嵌套越深，访问速度越慢。location.href总是快于window.location.href，而后者也要比window.location.href.toString()更快。如果这些属性不是对象的实例属性，那么成员解析还要在每个点上搜索原形链，这将需要更长时间。

1. **重绘和重排版**

**重绘：**当DOM树中的元素大小位置不变化，只是改变其背景颜色，那么页面就需要重绘。

**重排版：**当DOM树中的元素大小位置变化时，那么页面就需要重排版。

如下情况就会发生重排版：

①添加或删除可见的 DOM 元素

②元素位置改变

③元素尺寸改变（因为边距，填充，边框宽度，宽度，高度等属性改变）

④内容改变，例如，文本改变或图片被另一个不同尺寸的所替代

⑤最初的页面渲染

⑥浏览器窗口改变尺寸

⑦某些改变可导致重排版整个页面：例如，当一个滚动条出现时。

**由于重绘和重排版对性能的消耗非常高，因此应该尽量减少重绘和重排版。为减少发生次数，你应该将多个 DOM 和风格改变合并到一个批次中一次性执行。**

方法如下：

**方法1**：

反例：

**var** el = document.getElementById('mydiv');

el.style.borderLeft = '1px';

el.style.borderRight = '2px';

el.style.padding = '5px';

这里改变了三个风格属性，每次改变都影响到元素的几何属性。在这个糟糕的例子中，它导致浏览器重排版了三次。

正确的做法：

一个达到同样效果而效率更高的方法是：将所有改变合并在一起执行，只修改 DOM 一次。可通过使用cssText属性实现

**var** el = document.getElementById('mydiv');

el.style.cssText = 'border-left: 1px; border-right: 2px; padding: 5px;';

这个例子中的代码修改cssText属性，覆盖已存在的风格信息。如果你打算保持当前的风格，你可以将它附加在cssText字符串的后面。

el.style.cssText += '; border-left: 1px;';

**方法2：**

另一个一次性改变风格的办法是修改 CSS 的类名称，而不是修改内联风格代码。这种方法适用于那些风格不依赖于运行逻辑，不需要计算的情况。改变 CSS 类名称更清晰，更易于维护；它有助于保持脚本免除显示代码，虽然它可能带来轻微的性能冲击，因为改变类时需要检查级联表。

**var** el = document.getElementById('mydiv');

el.className = 'active';

**方法3：批量修改DOM。**

当你需要对 DOM 元素进行多次修改时，你可以通过以下步骤减少重绘和重排版的次数：

①从文档流中摘除该元素（方法有三：第一，隐藏元素，进行修改，然后再显示它。第二，使用一个文档片断在已存 DOM 之外创建一个子树，然后将它拷贝到文档中。第三，将原始元素拷贝到一个脱离文档的节点中，修改副本，然后覆盖原始元素）

②对其应用多重改变

③将元素带回文档中

此过程引发两次重排版——第一步引发一次，第三步引发一次。如果你忽略了这两个步骤，那么第二步中每次改变都将引发一次重排版。

**隐藏元素实例：**

Html片段

<ulid=*"mylist"*>

<li><ahref=*"http://phpied.com"*>Stoyan</a></li>

<li><ahref=*"http://julienlecomte.com"*>Julien</a></li>

</ul>

需要填充的数据

**var** data = [

{

"name": "Nicholas",

"url": "http://nczonline.net"

},

{

"name": "Ross",

"url": "http://techfoolery.com"

}

];

//下面是一个通用的函数，用于将新数据更新到指定节点中：

**function**appendDataToElement(appendToElement, data) {

**var** a, li;

**for** (**var** i = 0, max = data.length; i < max; i++) {

a = document.createElement('a');

a.href = data[i].url;

a.appendChild(document.createTextNode(data[i].name));

li = document.createElement('li');

li.appendChild(a);

appendToElement.appendChild(li);

}

};

糟糕的用法，强烈不建议使用这种方法：

**var**ul = document.getElementById('mylist');

appendDataToElement(ul, data);

推荐使用：

**var**ul = document.getElementById('mylist');

ul.style.display = 'none';

appendDataToElement(ul, data);

ul.style.display = 'block';

**在文档之外创建并更新一个文档片断，然后将它附加在原始列表上的实例：**

文档片断一个便利的语法特性是当你向节点附加一个片断时，实际添加的是文档片断的子节点群，而不是片断自己。下面的例子减少一行代码，只引发一次重排版，只触发“存在 DOM”一次。

**var** fragment = document.createDocumentFragment();

appendDataToElement(fragment, data);

document.getElementById('mylist').appendChild(fragment);

**创建要更新节点的副本，然后在副本上操作，最后用新节点覆盖老节点的实例：**

**var** old = document.getElementById('mylist');

**var** clone = old.cloneNode(**true**);

appendDataToElement(clone, data);

old.parentNode.replaceChild(clone, old);

**总结：推荐尽可能使用文档片断（第二种解决方案）因为它涉及最少数量的 DOM 操作和重排版。唯一潜在的缺点是，当前文档片断还没有得到充分利用，开发者可能不熟悉此技术。**

1. **缓冲布局信息，如：偏移量、滚动条位置、风格属性等**

当你查询布局信息如偏移量、滚动条位置，或风格属性时，浏览器刷队列并执行所有修改操作，以返回最新的数值。最好是尽量减少对布局信息的查询次数，查询时将它赋给局部变量，并用局部变量参与计算。

**如下属性都会刷新队列动作，因此应尽量减少查询这些属性，查询时将它赋给局部变量，并用局部变量参与计算。**

offsetTop, offsetLeft, offsetWidth, offsetHeight

scrollTop, scrollLeft, scrollWidth, scrollHeight

clientTop, clientLeft, clientWidth, clientHeight

getComputedStyle() (currentStyle in IE)（在 IE 中此函数称为currentStyle）

1. **将元素提出动画流，避免对大部分页面进行重排版**

对于动画或者广告，使用以下步骤可以避免对大部分页面进行重排版：

①使用绝对坐标定位页面动画的元素，使它位于页面布局流之外。页面顶部可以“折叠/展开”的元素称作“动画元素”，用绝对坐标对它进行定位，当它的尺寸改变时，就不会推移页面中其他元素的位置，而只是覆盖其他元素。

②启动元素动画。当它扩大时，它临时覆盖部分页面。这是一个重绘过程，但只影响页面的一小部分，避免重排版并重绘一大块页面。

展开动作只在“动画元素”上进行。这时其他元素的坐标并没有改变，换句话说，其他元素并没有因为“动画元素”的扩大而随之下移，而是任由动画元素覆盖。

③当动画结束时，重新定位，从而只一次下移文档其他元素的位置。“动画元素”的动画结束时，将其他元素的位置下移到动画元素下方，界面“跳”了一下。

1. **对于IE的伪选择器:hover，避免使用**

自从版本 7 之后，IE 可以在任何元素（严格模式）上使用:hover 这个 CSS 伪选择器。然而，如果大量的元素使用了:hover 那么会降低反应速度。此问题在 IE8 中更显著。

例如，如果你创建了一个由 500-1000 行 5 列构成的表，并使用tr:hover改变背景颜色，高亮显示鼠标光标所在的行，当鼠标光标在表上移动时，性能会降低。使用高亮是个慢速过程，CPU 使用率会提高到80%-90%。所以当元素数量很多时避免使用这种效果，诸如很大的表或很长的列表。

1. **全局变量与局部变量性能对比**
2. 全局变量更耗内存，因为全局变量会在内存中一直存在，方便其他地方访问，因此，比起局部变量更占内存。
3. 全局变量访问速度比局部变量要慢一点，根据实验，局部变量访问速度比全局变量访问速度至少快一倍以上。

总结：尽量少用全局变量，多用局部变量。

1. **嵌套if语句对性能的影响**

根据实验，嵌套6层if判断语句，进行1百万次的循环，总共进行了600万次的判断。与没有进行if判断，进行100万次的循环的性能对比。

结果得到在火狐和谷歌下，进行if判断的语句会慢1/2，也就是没有if语句的快了一倍，显然if嵌套对性能影响不是很大。

但在IE8下，他们性能差距有点大，大约没有if判断的快了5倍左右。

1. **对for循环进行性能优化——达夫设备**

**方法1：**在这里我们只进行了两次循环，每次循环处理8个事务，减少循环次数可以有效的提高性能，特别是循环次数达到1000次以上，最好使用循环优化方法。

**var** items = [1,2,3,4,5,6,7,8,9,10,11];

$(**function**(){

set();

// setInner();

// innerHTMLLoop();

});

**function** set(){

**var** iterations = Math.floor(items.length / 8),

startAt = items.length % 8,

i = 0;

**var** count = 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;

count ++;

} **while** (iterations--);

alert(count);

}

**function** process(val){

}

**方法2：**由于方法1 使用了switch，因此性能也会受一定影响，最优的方式是去掉switch。

**var** items = [1,2,3,4,5,6,7,8,9,10,11];

$(**function**(){

set();

// setInner();

// innerHTMLLoop();

});

**function** set(){

**var** i = items.length % 8;

**var** index = 0;

**while**(i){

process(items[index++]);

i--;

}

i = Math.floor(items.length / 8);

**while**(i){

process(items[index++]);

process(items[index++]);

process(items[index++]);

process(items[index++]);

process(items[index++]);

process(items[index++]);

process(items[index++]);

process(items[index++]);

i--;

}

}

**function** process(val){

$('.content').text($('.content').text()+val+";");

}

**实例对比：仔细观察**

/\*\*

\*内部有一个10此循环,当这段代码与下一个方法在同等条件下执行时，当执行次数达到百万的时候，该方法执行时间大概是loopTwo方法的10倍，性能差距明显。

\*/

**function** loop(){

**var** start = **new** Date().getTime();

**var**i = **null** ;

**for** (**var** v = 0; v < 1000000; v++) {

**for** (**var**int = 0; int< 10; int++) {

i = 10;

}

}

**var** end = **new** Date().getTime();

alert(end - start);

}

/\*\*

\*内部没有循环，而是10次顺序执行的代码，与上一个方法实现的功能一样

\*/

**function**loopTwo(){

**var** start = **new** Date().getTime();

**var**i = **null** ;

**for** (**var** v = 0; v < 1000000; v++) {

i = 10;

i = 10;

i = 10;

i = 10;

i = 10;

i = 10;

i = 10;

i = 10;

i = 10;

i = 10;

}

**var** end = **new** Date().getTime();

alert(end - start);

}

1. **If-else和switch的对比**

**从易读性出发：**

使用 if-else 或者 switch 的流行理论是基于测试条件的数量：条件数量较大，倾向于使用 switch 而不是if-else。这通常归结到代码的易读性。这种观点认为，如果条件较少时， if-else 容易阅读，而条件较多时 switch更容易阅读。

**从性能出发：**

事实证明，大多数情况下 switch 表达式比 if-else 更快，但只有当条件体数量很大时才明显更快。两者间的主要性能区别在于：当条件体增加时，if-else 性能负担增加的程度比 switch 更多。因此，我们的自然倾向认为条件体较少时应使用if-else而条件体较多时应使用switch表达式，如果从性能方面考虑也是正确的。

**总结：**一般来说， if-else 适用于判断两个离散的值或者判断几个不同的值域。如果判断多于两个离散值， switch表达式将是更理想的选择。

1. **If-else性能优化**

**优化1：概率最优**

注意以下代码：

**if** (value < 5) {

//do something

} **elseif** (value > 5 && value < 10) {

//do something

} **else** {

//do something

}

这段代码只有当 value 值经常小于 5 时才是最优的。如果 value 经常大于等于 10，那么在进入正确分支之前，必须两次运算条件体，导致表达式的平均时间提高。if-else 中的条件体应当总是按照从最大概率到最小概率的顺序排列，以保证理论运行速度最快。

**优化2：**二分搜索法

1. **查表法，尽量不使用if-else和switch**

**方法1：**数组查表法

**var** results = [result0, result1, result2, result3, result4, result5, result6, result7, result8, result9, result10];

//return the correct result

**return** results[value];

当使用查表法时，必须完全消除所有条件判断。操作转换成一个数组项查询或者一个对象成员查询。使用查表法的一个主要优点是：由于没有条件判断，当候选值数量增加时，很少，甚至没有增加额外的性能开销。

查表法最常用于一个键和一个值形成逻辑映射的领域（如前面的例子）。一个 switch 表达式更适合于每个键需要一个独特的动作，或者一系列动作的场合

1. **字符串连接优化性能**

**优化1：注意对比以下代码**

str += "one" + "two";

此代码有四个步骤

①内存中创建了一个临时字符串。

②临时字符串的值被赋予“onetwo”。

③临时字符串与str的值进行连接。

④结果赋予str。

str = str + "one" + "two";**（建议使用）**

此代码有两个步骤

①字符串"one"和"two"与str的值进行连接。

②结果赋予str。

通过对比上面的两种方式，我们发现方式二会少了两个步骤，少了创建临时字符串的过程，这样会有利于提高性能，大概会快10%-40%。

**注：**str = "one" + str + "two";这种方式和方式1没什么区别，建议不这样使用，使用方式2

**优化2：使用数组来进行字符串连接**

**在IE7下效果显著**

**function**processStr(){

**var** start = **new** Date().getTime();

**var**str = "I'm a thirty-five character string.";

**var**strs = [];

**var**newStr;

appends = 50000;

**while** (appends--) {

strs[strs.length] = str;

}

newStr = strs.join("");

**var** end = **new** Date().getTime();

alert(end-start);

}

**注：在IE8及以上、火狐以及谷歌的最新版本，这种方法反而会比较慢，还不如字符串直接连接的效率高，因此多加注意。**

1. **长脚本的解决方法——定时器，html5后用工人线程代替，效果更好**

**问题描述：**当用户点击按钮，如果按钮的响应事件是个长脚本，也就是说该脚本的响应时间超过100毫秒，言外之意就是响应时间太长，给用户造成不好的交互体验。

**解决方法：**用定时器让出时间片，让出对 UI 线程的控制，使 UI 更新可以进行。让出控制意味着停止 JavaScript 运行，给 UI 线程机会进行更新，然后再继续运行 JavaScript。

代码如下：

**var** button = document.getElementById("my-button");

button.onclick = **function**(){

oneMethod();

setTimeout(**function**(){

document.getElementById("notice").style.color = "red";

}, 50);

anotherMethod();

};

如果anotherMethod()执行时间超过 50 毫秒，那么定时器代码将在onclick处理完成之前加入到队列中。其结果是等onclick处理运行完毕，定时器代码立即执行，而察觉不出其间的延迟。

**注：我们一般把耗时的脚本也就是长脚本放到定时器中运行。**

1. **用定时器取代循环**

**前提条件：**

①此处理过程必须是同步处理吗？

②数据必须按顺序处理吗？

③process()的复杂吗？，items数组或者对象大吗？

如果处理过程不需要同步，且数据不需要按顺序进行，并且process函数比较复杂，items数组也比较大的情况下，我们可以采用定时器取代循环。

代码如下：

**var** items = [1,2,3,4,5,6,7,8,9,10,11];

$(**function**(){

**var**todo = items.concat();

setTimeout(**function**(){

//get next item in the array and process it

process(todo.shift());

//if there's more items to process, create another timer

**if**(todo.length> 0){

//arguments.callee这个是返回正在执行的函数，也就是返回定时器里面执行的这个匿名函数

setTimeout(arguments.callee, 25);

} **else** {

callback(items);

}

}, 25);

});

**function** process(val){

}

**注：在你的网页应用中限制高频率重复定时器的数量。同时，Thomas 建议创建一个单独的重复定时器，每次执行多个操作**

1. **Ajax传输的数据格式**

总的来说越轻量级的格式越好，最好是 JSON 和字符分隔的自定义格式。如果数据集很大或者解析时间成问题，那么就使用这两种格式之一。

字符分隔的格式，比如：a;b;c;d这里是以;字符进行分隔，方便解析，使用split()方法进行解析。

**注：其中速度最快的是自定义字符分隔格式，json格式次之。**

1. **浏览器缓存数据**

**方法1：**在服务器端，设置 HTTP 头，确保返回报文将被缓存在浏览器中。它的意思就是由服务器通过设置响应报头http来告诉浏览器是否缓存该数据，并且可以设置缓存的时间。

请求方式必须是get，设置缓存时间使用Expires属性（在时间没过期的时候，获取的数据都是从缓存中获取，如果过期，则会重新发送ajax请求从服务器获取数据）。

**方法2：**在客户端，于本地缓存已获取的数据，不要多次请求同一个数据。这个方法最简单，直接在浏览器的js文件中声明一个js变量来缓存该数据就行

1. **可以运行字符串代码的四种方法**

**方法1：**eval()

实例代码：

**var**num1 = 5,

num2 = 6,

//eval() evaluating a string of code

result = eval("num1 + num2");

alert(result);

**方法2：**Function()

**var** num1 = 5,

num2 = 6,

sum = **new** Function("arg1", "arg2", "return arg1 + arg2");

alert(sum(num1,num2));

**方法3：**setTimeout()

**var** num1 = 5,num2 = 6;

setTimeout("alert("+num1+"+"+num2+")", 100);

**方法4：**setInterval()

**var** num1 = 5,num2 = 6;

setInterval("alert("+num1+"+"+num2+")", 100);

总结：这四种特别是前两种都不建议或者少使用，至于setTimeout和setInterval，最好第一个参数传递一个函数进来，而不应该传递字符串。

1. **js对json数据操作**

**遍历json:**

**params是json字符串{name:’abc’,age:12}**

**for**(**var** key **in**params){

**try**{

**if**(key!='name'){//key不为name的字符串删除

**delete** params[key];

}

}**catch**(e){

console.log('处理json字符出错！');

}

}

**添加json元素：**

params.phone = ‘18780103245’；

**删除元素：**

**delete** params['name'];

**修改元素值：**

params.name = 'gh';

1. **判断一个变量值是否为NaN**

**问题：**判断变量x是否为NaN

**方法：**x!=x true表示x值是NaN，false表示不是

**原因：**在js中NaN是不等于NaN的，所以必须通过这种方式判断。

1. **ds**