# 加载和执行

* 将所有的SCRIPT标签尽量放在body底部的位置，尽量减少对整个页面下载的影响。
* 尽量少的SCRIPT标签，原因就是因为SCRIPT标签会阻塞界面。Yahoo Combo,TEngine Contact.
* 保持JS文件小，并限制HTTP请求数量，只能算第一步，如果功能越多，一个大的JS文件下载下来也会阻塞界面。需要向页面中逐步添加Javascript文件。非阻塞的秘密在于，等页面加载完成后，再加载JS源码，这意味着

一般在Window的load事件发出之后下载代码。

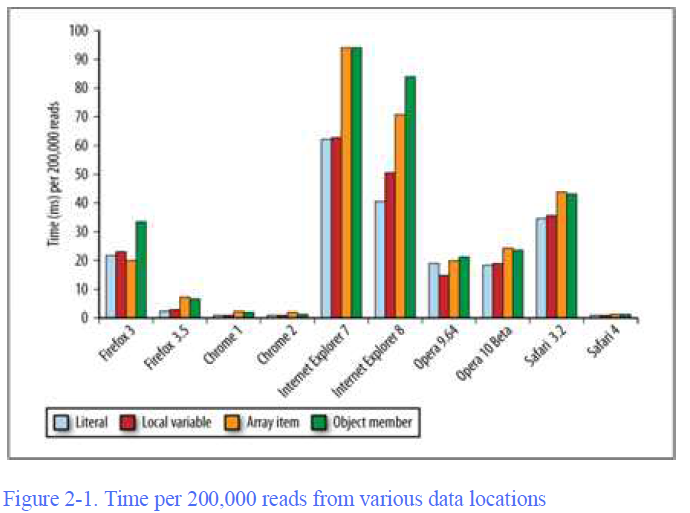
* 增加defer
* 动态脚本加载.可以创建多个动态脚本，但是这些脚本的执行顺序并不是按照指定的顺序。因此会有模块的依赖。
* XHR下载远程脚本处理
* 推荐的方式：包含动态加载JS代码，加载页面初始化所需要的除JS之外的部分。用它来加载其他的代码。

框架：

* LazyLoad
* LABjs

# 数据访问

## 作用域



直接量，本地变量，数组和对象在各个浏览器下的测试情况。

### 基本原理

每个Javascript函数都是一个对象，他是一个Function的实例，既然他是一个对象，就像其他对象一样，有可以访问的属性，而这些属性中，有些只能被Javascript引擎访问到，不能被外界程序访问。其中一个属性就是SCOPE.（ECMASCRIPT-262标准定义）

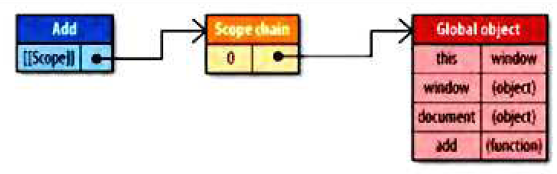
SCOPE属性包含了一个函数被创建时的作用域中对象的集合，此集合被称作是函数的**作用域链。**作用域链中的每个对象以键值的形式存在，每一个都是一个可变对象，当函数被创建后，他的作用域链被填充了一个对象，这个**对象代表创建此函数的环境中可以访问的数据**，通常这个对象是全局对象。

Function add(num1, num2){

Var sum = num1 + num2;

Return sum;

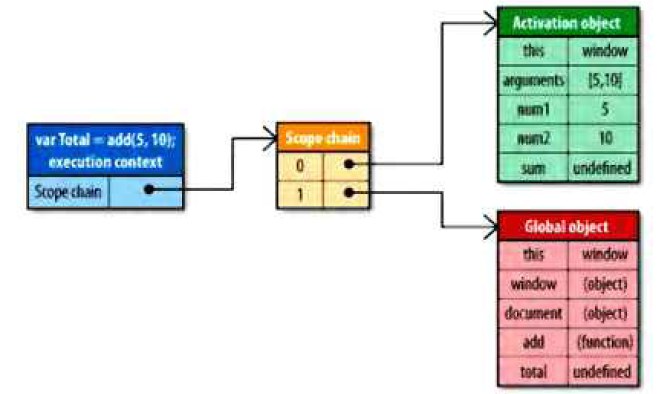
}



函数SCOPE与全局对象之间的关系

运行add函数时，会创建**运行期上下文**，他定义了一个函数运行时的环境，每次函数的运行，上下文都是独立的，多次调用会导致多次创建上下文运行环境，函数执行完毕时，运行期上下文被销毁。

运行期上下文创建时，相应的作用域链也就创建好了。此时，会创建一个**激活对象**，他是一个可变对象，这个对象中包含了运行时的函数中各个变量值等，包括局部变量，命名参数，参数集合，this，此对象会被加入到作用域链的最前端,当作用域链被销毁时，激活对象也会被一期销毁。如下所示：

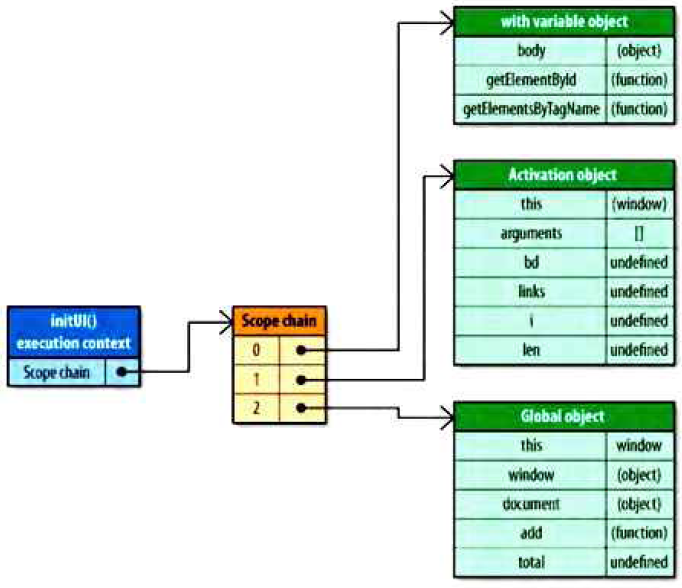


函数运行过程中，会从这些作用域链的对象中查找对应的标示符，激活对象中找不到，就回去链条中的其他对象中搜索，最后会搜索到全局变量，实际上全局变量永远处于作用域链的最后一个位置。所以这种过程会影响性能。对应的**解决方案是**：**将全局变量的引用存储在一个局部变量中，然后使用这个局部变量来代替全局变量**。

### 改变作用域

* With

With为所有对象创建了一个默认的操作变量，这个会改变作用域，因为一个新的对象会被创建，并且插入到作用域链的顶端，函数所有的局部变量都被推入到第二端，这样with中的变量访问速度很快，但是函数的其他局部变量访问速度变慢。



* Try Catch

Try catch的catch块也会引起作用域链的改变，当异常发生的时候，程序会自动流入Catch块，并将异常对象推入作用域链的顶端，其他局部变量放在作用域链的第二个位置。Catch语句只要执行完毕，作用域链就会返回原来的状态。

可以精简代码来减少Catch对性能的影响，一个有用的办法是直接用一个处理函数来处理异常，而不要访问局部变量。

* With和Try Catch都是动态改变作用域，这些代码并不能在静态分析期间确定，而是在动态运行期确定的。

### 闭包与作用域、内存

Function assignEvent(){

Var id = “xdi9528”;

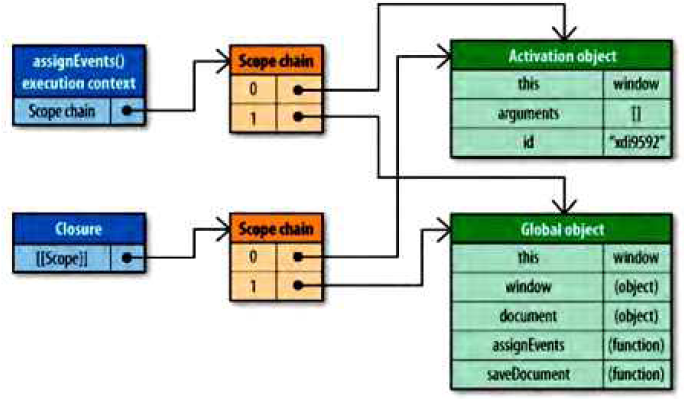
Document.getElemenetById(“save-btn”).onclick = function(){

saveDocument(id);

}

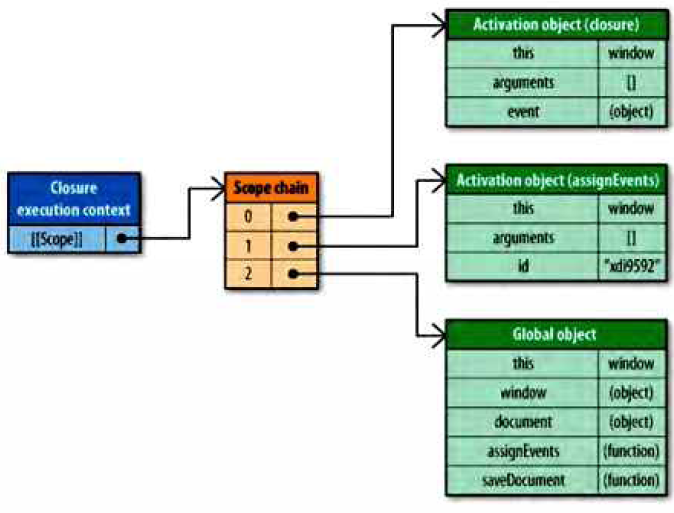
}

* 上面在assignEvent中创建了到DOM绑定的事件，并且事件的处理函数中访问了assignEvent中的变量id.
* 当assignEvent被执行时，一个激活对象被创建，包含了id变量。他将成为作用域链中的第一个对象，全局对象为第二个。当闭包创建时，SCOPE属性与这些对象一起被初始化, 并和assignEvent共享了相同的作用域链。如下中展示了assignEvent和闭包的SCOPE属性：



可见，闭包的SCOPE属性和assignEvent包含了**相同的作用域链对象**，通常一个函数的激活对象与运行期上下文一同销毁，当涉及闭包时，闭包对象就无法销毁，因为引用还存在，这样导致会存在更多的内存开销。

* 当闭包被执行时，一个运行期的上下文被创建，他的作用域链与SCOPE属性中引用的两个相同的作用域链同时被初始化，然后一个新的激活对象为闭包自身创建。如下图：



从图中可以看出，id和saveDocument都存在于作用域链第一个对象之后，这样就会导致搜索起来比较麻烦，每次访问都会导致一些性能损失。

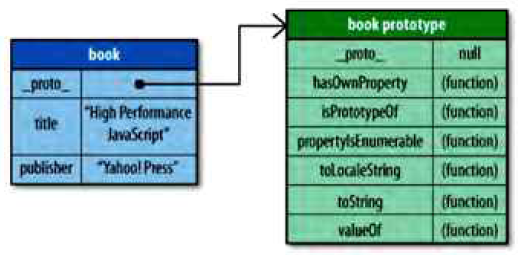
* 如何避免闭包造成的性能损失和内存膨胀？

<http://www.jibbering.com/faq/notes/closures/#clIRExSc>

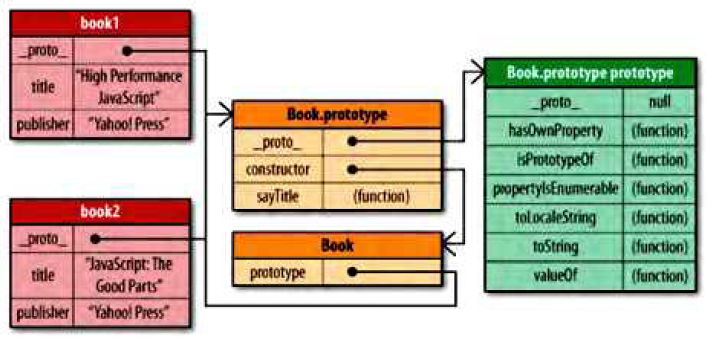
<http://stackoverflow.com/questions/13722581/should-i-avoid-creating-this-javascript-closure>

## 原型链

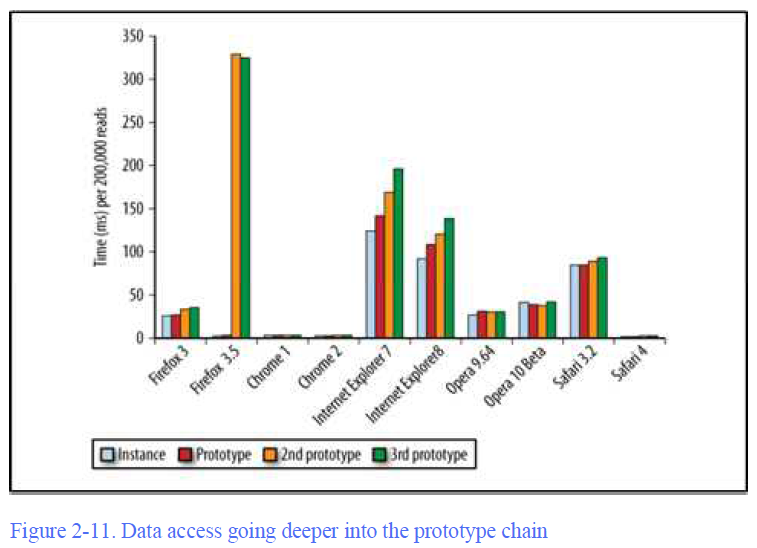
* Javascript的对象是基于原型的，原型是对象的基础，和传统的类的概念不一致。原型对象为所有给定类型的对象实例所共享。通过属性\_proto\_来控制原型。对象有两种类型的成员，分别是实例成员和原型成员。实例成员存在于实例中，原型成员则从原型链中继承下来。



* HasOwnProperty函数来判断一个属性是否是实例成员，in操作符来判断属性是否属于对象。多个实例对象，他们的实例变量是不同的，但是原型中的操作和属性是共享的。
* Book对象中包含了title和publisher两个属性。如下图：



当调用toString方法时，搜索工作会去原型链中搜索，所以原型链越深入，搜索的速度就越慢，各个浏览器下原型链深度与搜索的对比图：



搜索实例成员的过程比访问直接量或者局部变量负担更加重，增加遍历原型链的开销正好放大了这种效果, (**先去作用域链中搜索，再去原型链中搜索**)

* 嵌套的成员变量搜索会更加严重，因此应该使用局部变量来代替，在一个函数中，绝对不应该对一个对象成员搜索超过一次，除非这个值可能改变。

# DOM编程

重点关注的三个问题：

* 访问和修改DOM元素
* 修改DOM元素的样式，造成重绘和重排
* 通过注册DOM事件来处理用户响应

## 访问和修改DOM

DOM是独立于语言的，使用XML或者HTML来作为API接口，DOM API主要用于访问DOM相关的内容。客户端大部分的程序是用Javascript编写的，所以DOM称为了Javascript经常操作的对象。浏览器通常要求DOM和Javascript保持独立，比如IE使用Trident来渲染DOM, 脚本可以使用Javascript，也可以使用VBScript. Safari使用WebCore来渲染DOM,使用SquirrelFish作为Javascript引擎。Chrome使用WebCore来渲染DOM,使用V8 Javascript引擎。这意味着这两者之间交互时会彼此通过接口来交互，会降低一部分性能。

尽量少的访问DOM, 轻轻的触摸DOM, 并尽量保持在ECMAScript范围内部。

### InnerHTML

* 现代浏览器中使用document.createElement这样的DOM方法更快，在老的浏览器中使用innerHTML更快。

### CloneNode

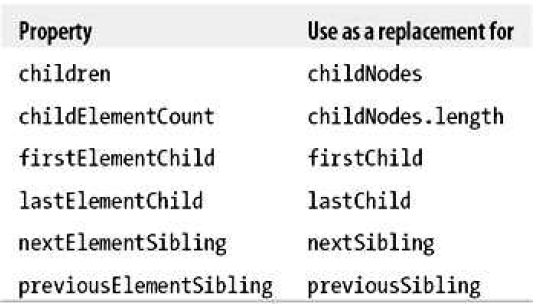
* 在大多数的浏览器中，cloneNode都有很不错的性能。

### HTML集合

* Document.images,document.forms,document.links等返回的结果都是HTML Collection集合对象。
* 他不是一个数组，但提供了类似的功能。
* 他虚拟存在，当集合中的HTML元素被更新时，他也会被更新。
* **需要将集合的length属性存放到一个变量中**。
* **遍历数组比遍历集合快，先将集合元素拷贝到数组，然后再访问数组中的集合元素的属性，这样操作会更快**。
* **任何类型的DOM访问，如果同一个DOM属性或方法被访问一次以上，最好使用一个局部变量缓存**。

### DOM抓取

* NextSibling在旧的浏览器下速度较快，比如IE6.
* ChildNode是个集合对象，需要小心处理，尤其是缓存他的length属性。
* 元素节点：DOM属性，比如childNode、firstChild、nextSibling不区分元素节点和其他类型节点，如注释节点和文本节点。因此很多浏览器提供了下面的方法来访问元素节点。



很明显，children要比child快，因为操作的集合更少。

### 选择器API

* QuerySelectAll: 根据css选择器来返回选择的节点，返回值为NodeList对象，返回值不是一个HTML集合，所以返回的节点不呈现出存在性结构，避免了HTML集合的性能问题。
* QuerySelectAll可以实现联合查询，比如:

Document.querySelectAll(“div.warning,div.notice”);

* 此方法被IE8支持。如果浏览器支持，最好使用。
* QuerySelector和上面这个类似，但是只是获取第一个。

### 重绘和重排

* DOM描述节点信息，渲染树计算节点的几何信息，位置等。这些构造完成后，浏览器绘制到窗口上。
* DOM改变影响到元素的几何属性时，比如位置，长度和宽度等，浏览器需要重新计算这些几何属性，那势必会影响到其他节点的几何属性。因此需要重新构造渲染树，这个过程叫做重排，重排之后会进行重绘到窗口中。
* 如果只是修改了DOM元素的非几何属性，比如背景色等，只需要重绘，不需要重排。
* 重排发生的情况
  + 添加和删除可见的DOM元素。
  + 元素位置改变。
  + 元素尺寸改变，包括边距，边框等。
  + 内容改变，比如图片或者文字的改变。
  + 初始化渲染时。
  + 浏览器窗口尺寸改变时。
* 大多数浏览器通过队列和批量修改的方式来优化重排，但以下属性会强迫浏览器刷新队列并计算真正的值。
  + Offset/scrollTop/clientTop/getComputedStyle/currentStyle
* 最小化重排和重绘操作
  + 合并多次的DOM风格改变的代码操纵（cssText）；
  + 使用类名代替内联风格代码；
  + 从文档流中摘除元素（float，absolute，none，**文档片段**，cloneNode）
  + IE8下如果大量使用:hover也会造成性能下降。
* 事件托管
  + 事件总是冒泡到父元素，因此在父元素进行捕获处理。

# 算法和流程控制

## 循环

### 类型

* For
* While
* Do…while
* For-in: 这是最明显的性能慢的循环，因为要搜索实例和原型的属性，所以比较慢。因此要尽量避免使用。如果迭代一个有限的，已知的属性列表，使用其他类型的循环更快。比如：

var props = ["prop1", "prop2"],i = 0;

while (i < props.length){

process(object[props[i]]);

}

除了for-in慢点之外，其他几个循环性能相当，要减少性能，只能从下面两个方面减少：

* 减少循环中的工作量
* 减少循环的次数

### 循环性能

* 不用在每个迭代中使用item.length
* 倒序循环可以略微提高性能
* 当数据量比较大时，循环中可以采用**达夫设备**循环体来操作。达夫设备循环可以**降低循环次数**。
* 基于函数的迭代，比如forEach方法，这些方法的性能会比普通的循环慢点，因为需要调用函数，因为每个数组要关联额外的函数调用。
* **减少循环中的运算量**

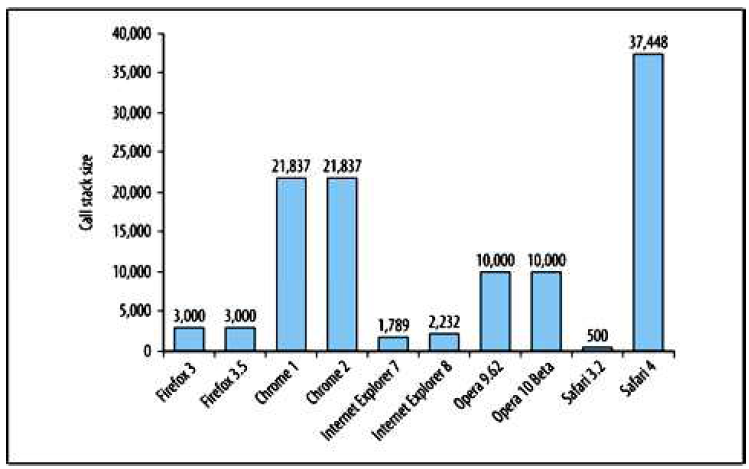
## 条件

## If-else/switch

* 大多数情况下，switch比if-else快，尤其是在条件体内容较多时。条件较少时，使用if-else，条件体较多时，使用switch.
* 优化之一总是将概率最大的可能点放在if-else的前面，减少运算量。
* 优化之二使用嵌套的if-else来减少运算量。
* 数组查表法：彻底消除if-else，使用数组查询来代替，转换为对象属性的获取等，这种方式尤其适用于大数据量的if-else时。

## 递归

* 合适的终止条件
* 各个浏览器的调用栈的大小，超过调用栈会出错。



## 迭代

* 迭代可以代替递归，这涉及到算法问题，后续讨论。
* 如果递归算法超出了堆栈大小，将方法修改为一个迭代算法或者使用制表法可以避免重复工作。

## 制表

* 制表，通过缓存先前计算结果为后续计算所重复使用，避免了重复工作，这使得制表成为递归算法中有用的技术。

# 字符串和正则表达

* 当连接数量巨大或者尺寸巨大的字符串时，数组联合是IE7和他早期版本中唯一具有合理性能的方法。
* 如果不关心IE7和他的早期版本，数组联合是最慢的方法之一，使用+或者+=直接代替，可以避免不必要的中间字符串。
* 回朔即是正则表达式匹配的基本组成部分，又是影响性能的常见原因。
* 正则表达式需要专门学习总结。

# 响应接口