**1：使用DocumentFragments或innerHTML取代复杂的元素注入**

DOM操作在浏览器上是要付税的。尽管性能提升是在浏览器，DOM很慢，如果你没有注意到，你可能会察觉浏览器运行非常的慢。这就是为什么减少创建集中的DOM节点以及快速注入是那么的重要了。

现在假设我们页面中有一个

元素，调用AJAX获取JSON列表，然后使用JavaScript更新元素内容。通常，程序员会这么写：

**Javascript代码**

var list = document.querySelector('ul');

ajaxResult.items.forEach(function(item) {

// 创建<li>元素

var li = document.createElement('li');

li.innerHTML = item.text;

// <li>元素常规操作，例如添加class，更改属性attribute，添加事件监听等

// 迅速将<li>元素注入父级<ul>中

list.apppendChild(li);

});

**Javascript代码**

var list = document.querySelector('ul');

ajaxResult.items.forEach(function(item) {

// 创建<li>元素

var li = document.createElement('li');

li.innerHTML = item.text;

// <li>元素常规操作，例如添加class，更改属性attribute，添加事件监听等

// 迅速将<li>元素注入父级<ul>中

list.apppendChild(li);

});

上面的代码其实是一个错误的写法，将

元素带着对每一个列表的DOM操作一起移植是非常慢的。如果你真的想要 使用document.createElement，并且将对象当做节点来处理，那么考虑到性能问题，你应该使用DocumentFragement。

DocumentFragement是一组子节点的“虚拟存储”，并且它没有父标签。在我们的例子中，将DocumentFragement想象成看不见的

元素，在 DOM外，一直保管着你的子节点，直到他们被注入DOM中。那么，原来的代码就可以用DocumentFragment优化一下：

**Javascript代码**

var frag = document.createDocumentFragment();

ajaxResult.items.forEach(function(item) {

// 创建<li>元素

var li = document.createElement('li');

li.innerHTML = item.text;

// <li>元素常规操作

// 例如添加class，更改属性attribute，添加事件监听，添加子节点等

// 将<li>元素添加到碎片中

frag.appendChild(li);

});

// 最后将所有的列表对象通过DocumentFragment集中注入DOM

document.querySelector('ul').appendChild(frag);

为DocumentFragment追加子元素，然后再将这个DocumentFragment加到父列表中，这一系列操作仅仅是一个DOM操作，因此它比起集中注入要快很多。

如果你不需要将列表对象当做节点来操作，更好的方法是用字符串构建HTML内容：

**Javascript代码**

var htmlStr = '';

ajaxResult.items.forEach(function(item) {

// 构建包含HTML页面内容的字符串

htmlStr += '<li>' + item.text + '</li>';

});

// 通过innerHTML设定ul内容

document.querySelector('ul').innerHTML = htmlStr;

这当中也只有一个DOM操作，并且比起DocumentFragment代码量更少。在任何情况下，这两种方法都比在每一次迭代中将元素注入DOM更高效。

**2：高频执行事件/方法的防抖**

通常，开发人员会在有用户交互参与的地方添加事件，而往往这种事件会被频繁触发。想象一下窗口的resize事件或者是一个元素的onmouseover事件 - 他们触发时，执行的非常迅速，并且触发很多次。如果你的回调过重，你可能使浏览器死掉。

这就是为什么我们要引入防抖。

这里涉及到的知识点是高阶函数，感兴趣的朋友可以移步到我的另一篇博文全面理解Javascript的面向对象(一)，里面的函数节流部分就是实现高频执行事件/方法的防抖。

防抖可以限制一个方法在一定时间内执行的次数。以下代码是个防抖示例：

**Javascript代码**

function debounce(func, wait, immediate) {

var timeout;

return function() {

var context = this, args = arguments;

var later = function() {

timeout = null;

if (!immediate) func.apply(context, args);

};

var callNow = immediate && !timeout;

clearTimeout(timeout);

timeout = setTimeout(later, wait);

if (callNow) func.apply(context, args);

};

}

添加resize的回调函数，但是只允许它每300毫秒执行一次

window.addEventListener('resize', debounce(function(event) {

// 这里写resize过程

}, 300));

debounce方法返回一个方法，用来包住你的回调函数，限制他的执行频率。使用这个防抖方法，就可以让你写的频繁回调的方法不会妨碍用户的浏览器！

**3：网络存储的静态缓存和非必要内容优化**

Web Storage的API曾经是Cookie API一个显著的进步，并且为开发者使用了很多年了。这个API是合理的，更大存储量的，而且是更为健全理智的。一种策略是去使用Session存储来存 储非必要的，更为静态的内容，例如侧边栏的HTML内容，从Ajax加载进来的文章内容，或者一些其他的各种各样的片断，是我们只想请求一次的。

我们可以使用JavaScript编写一段代码，利用Web Storage使这些内容加载更加简单:

**Javascript代码**

define(function() {

var cacheObj = window.sessionStorage || {

getItem: function(key) {

return this[key];

},

setItem: function(key, value) {

this[key] = value;

}

};

return {

get: function(key) {

return this.isFresh(key);

},

set: function(key, value, minutes) {

var expDate = new Date();

expDate.setMinutes(expDate.getMinutes() + (minutes || 0));

try {

cacheObj.setItem(key, JSON.stringify({

value: value,

expires: expDate.getTime()

}));

}

catch(e) { }

},

isFresh: function(key) {

// 返回值或者返回false

var item;

try {

item = JSON.parse(cacheObj.getItem(key));

}

catch(e) {}

if(!item) return false;

// 日期算法

return new Date().getTime() > item.expires ? false : item.value;

}

}

});

这个工具提供了一个基础的get和set方法，同isFresh方法一样，保证了存储的数据不会过期。调用方法也非常简单:

**Javascript代码**

require(['storage'], function(storage) {

var content = storage.get('sidebarContent');

if(!content) {

// Do an AJAX request to get the sidebar content

// ... and then store returned content for an hour

storage.set('sidebarContent', content, 60);

}

});

现在同样的内容不会被重复请求，你的应用运行的更加有效。花一点儿时间，看看你的网站设计，将那些不会变化，但是会被不断请求的内容挑出来，你可以使用Web Storage工具来提升你网站的性能。

**4：使用异步加载，延迟加载依赖**

RequireJS已经迎来了异步加载和AMD格式的巨大浪潮。XMLHttpRequest(该对象可以调用AJAX)使得资源的异步加载变得流行起来，它允许无阻塞资源加载，并且使 onload 启动更快，允许页面内容加载，而不需要刷新页面。

我所用的异步加载器是John Hann的curl。curl加载器是基本的异步加载器，可以被配置，拥有很好的插件。以下是一小段curl的代码：

**Javascript代码**

// 基本使用: 加载一部分AMD格式的模块

curl(['social', 'dom'], function(social, dom) {

dom.setElementContent('.social-container', social.loadWidgets());

});

// 定义一个使用Google Analytics的模块，该模块是非AMD格式的

define(["js!//google-analytics.com/ga.js"], function() {

// Return a simple custom Google Analytics controller

return {

trackPageView: function(href) {

\_gaq.push(["\_trackPageview", url]);

},

trackEvent: function(eventName, href) {

\_gaq.push(["\_trackEvent", "Interactions", eventName, "", href || window.location, true]);

}

};

});

// 加载一个不带回调方法的非AMD的js文件

curl(['js!//somesite.com/widgets.js']);

// 将JavaScript和CSS文件作为模块加载

curl(['js!libs/prism/prism.js', 'css!libs/prism/prism.css'], function() {

Prism.highlightAll();

});

// 加载一个AJAX请求的URL

curl(['text!sidebar.php', 'storage', 'dom'], function(content, storage, dom) {

storage.set('sidebar', content, 60);

dom.setElementContent('.sidebar', content);

});

你可能早就了解，异步加载可以大大提高万展速度，但是我想在此说明的是，你要使用异步加载！使用了之后你可以看到区别，更重要的是，你的用户可以看到区别。

当你可以根据页面内容延迟加载依赖的时候，你就可以体会到异步加载的好处了。例如，你可以只加载Twitter，Facebook和Google Plus到应用了名为social的CSS样式的div元素中。“在加载前检查是否需要”策略可以为我的用户节省好几KB的莫须有的加载。

**5：使用Array.prototype.join代替字符串连接**

有一种非常简单的客户端优化方式，就是用Array.prototype.join代替原有的基本的字符连接的写法。在上面的“最佳实践1”中，我在代码中使用了基本字符连接：

**Javascript代码**

htmlStr += '<li>' + item.text + '</li>';

但是下面这段代码中，我用了优化：

**Javascript代码**

var items = [];

ajaxResult.items.forEach(function(item) {

// 构建字符串

items.push('<li>', item.text, '</li>');

});

// 通过innerHTML设置列表内容

document.querySelector('ul').innerHTML = items.join('');

也许你需要花上一点儿时间来看看这个数组是做什么用的，但是所有的用户都从这个优化中受益匪浅。

**6：尽可能使用CSS动画**

网站设计对美观特性和可配置元素动画的大量需求，使得一些JavaScript类库，如jQuery，MooTools大量的被使用。尽管现在浏览器支持CSS的transformation和keyframe所做的动画，现在仍有很多人使用JavaScript制作动画效果，但是实际上使用CSS动画比起JavaScript驱动的动画效率更高。CSS动画同时需要更少的代码。很多的CSS动画是用GPU处理的，因此动画本身很流畅，当然你可以使用下面这个简单的CSS强制使你的硬件加速：

**Javascript代码**

.myAnimation {

animation: someAnimation 1s;

transform: translate3d(0, 0, 0); /\* 强制硬件加速 \*/

}

tansform:transform(0,0,0)在不会影响其他动画的同时将通话送入硬件加速。在不支持CSS动画的情况下（IE8及以下版本的浏览器），你可以引入JavaScript动画逻辑：

**Javascript代码**

<!--[if 低于IE8版本]>

<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>

<script src="/js/ie-animations.js"></script>

<![endif]-->

在上例中，ie-animations.js文件必须包含你自定义的jQuery代码，用于当CSS动画在早期IE中不被支持的情况下，来替代CSS动画完成动画效果。完美的通过CSS动画来优化动画，通过JavaScript来支持全局动画效果。

**7：使用事件委托**

想象一下，如果你有一个无序列表，里面有一堆

元素，每一个

元素都会在点击的时候触发一个行为。这个时候，你通常会在每一个元素上添加一个事件监听，但是如果当这个元素或者你添加了监听的这个对象会被频繁的移除添加呢？这个时候，你在移除添加元素的同时需要处理事件监听的移除和添加。这个时候，我们就需要引入事件委托了。

事件委托是在父级元素上添加一个事件监听，来替代在每一个子元素上添加事件监听。当事件被触发时，event.target会评估相应的措施是否需要被执行。下面我们给出了一个简单的例子：

**Javascript代码**

// 获取元素，添加事件监听

window.onload = function(){

　　var oUl = document.getElementById("ul1");

　　oUl.onclick = function(ev){

　　　　var ev = ev || window.event;

　　　　var target = ev.target || ev.srcElement;

　　　　if(target.nodeName.toLowerCase() == 'li'){

　　　　　　　 alert(target.innerHTML);

　　　　}

　　}

}

上面的例子是不可思议的简单，当事件发生的时候，它没有轮询父节点去寻找匹配的元素或选择器，且它不支持基于选择器的查询（例如用class name，或者id来查询）。所有的JavaScript框架提供了委托选择器匹配。重点是，你避免了为每一个元素加载事件监听，而是在父元素上加一个事件监听。这样大大的增加了效率，并且减少了很多维护！

**8：使用Data URI代替图片SRC**

提升页面大小的效率，不仅仅是取决于使用精灵或是压缩代码，给定页面的请求数量在前端性能中也占有了很不小的重量。减少请求可以让你的网站加载更快，而其中一种减少页面请求的方法就是用Data URI代替图片的src属性：

**Javascript代码**

<!-- 以前的写法 -->

<img src="/images/logo.png" />

<!-- 使用data URI的写法 -->

<img src="data: image/jpeg;base64,/9j/4AAQSkZJRgABAgAAZABkAAD/7AARRHVja3kAAQAEAAAAPAAA/+4ADkFkb2JlAGTAAAAAAf/bAIQABgQEBAUEBgUFBgkGBQYJCwgGBggLDAoKCwoKDBAMDAwMDAwQDA4PEA8ODBMTFBQTExwbGxscHx8fHx8fHx8fHwEHBwcNDA0YEBAYGhURFRofHx8fHx8fHx8fHx8fHx8fH ...." />

<-- 范例: http://davidwalsh.name/demo/data-uri-php.php -->

当然页面大小会增加（如果你的服务器使用适当的gzip内容，这个增加会很小），但是你减少了潜在的请求，同时也在过程中减少了服务器请求的数量。现在大多数浏览器都支持Data URI，在CSS中的背景骨片也可以使用Data URI，因此这个策略现在已经可以在应用层级，广泛应用。

**9：使用媒体查询加载指定大小的背景图片**

直到CSS @supports被广泛支持，CSS媒体查询的使用接近于CSS中写逻辑控制。我们经常用CSS媒体查询来根据设备调整CSS属性（通常根据屏幕宽度调整CSS属性），例如根据不同的屏幕宽度来设置不同的元素宽度或者是悬浮位置。那么我们为什么不用这种方式来改变背景图片呢？

**Javascript代码**

/\* 默认是为桌面应用加载图片 \*/

.someElement { background-image: url(sunset.jpg); }

@media only screen and (max-width : 1024px) {

.someElement { background-image: url(sunset-small.jpg); }

}

上面的代码片段是为手机设备或是类似的移动设备加载一个较小尺寸的图片，特别是需要一个特别小的图片时（例如图片的大小几乎不可视）。

**10：使用索引对象**

这一篇，我们将讲讲使用索引对象检索代替遍历数组，提高遍历速度。

AJAX和JSON一个最常见的使用案例是接收包含一组对象的数组，然后从这组数组中根据给定的值搜索对象。让我们看一个简单的例子，下面这个例子中，你从用户接收一个数组，然后你可以根据username的值来搜索用户对象：

**Javascript代码**

function getUser(desiredUsername) {

var searchResult = ajaxResult.users.filter(function(user) {

return user.username == desiredUsername;

});

return searchResult.length ? searchResult[0] : false;

}

// 根据用户名获取用户

var davidwalsh = getUser("davidwalsh");

// 根据用户名获取另一个用户.

var techpro = getuser("tech-pro");

上面这段代码可以运行，但是并不是很有效，当我们想要获取一个用户时，我们就要遍历一次数组。那么更好的方法是创建一个新的对象，对每一个唯一的值建立一个索引，在上面这个例子中，用username作为索引，这个数组对象可以写成：

**Javascript代码**

var userStore = {};

ajaxResult.users.forEach(function(user) {

userStore[user.username] = user;

});

现在当你想要找一个用户对象时，我们可以直接通过索引找到这个对象：

**Javascript代码**

var davidwalsh = userStore.davidwalsh;

var techpro = userStore["tech-pro"];

这样的代码写起来更好一些，也很简便，通过索引搜索比起遍历整个数组更加快捷。

**11：控制DOM大小**

这一篇中，我们要说如何控制DOM的大小，来优化前端性能。

DOM很慢是众所周知的，使得网站变慢的罪魁祸首是大量的DOM。想象一下，假如你有一个有着上千节点的DOM，在想象一下，使用querySelectorAll或者getElementByTagName，或者是其他以DOM为中心的搜索方式来搜索一个节点，即使是使用内置方法，这也将是一个非常费力的过程。你要知道，多余的DOM节点会使其他的实用程序也变慢的。

我见过的一种情况，DOM的大小悄然增加，是在一个AJAX网站，它将所有的页面都存在了DOM中，当一个新的页面通过AJAX被加载时，旧的页面就会被存入隐藏的DOM节点。对于DOM的速度，将有灾难性的降低，特别是当一个页面是动态加载的。所以你需要一种更好的方法。

在这种情况下，当页面是通过AJAX加载的，并且以前的页面是存储在客户端的，最好的方法就是将内容通过String HTML存储（将内容从DOM中移除），然后使用事件委托来避免特定元素事件。这么做的同时，当在客户端缓存内容的时候，可以避免大量的DOM生成。

通常控制DOM大小的技巧包括：

使用:before和:after伪元素

延迟加载和呈现内容

使用事件委托，更简便的将节点转换成字符串存储

简单一句话：尽量使你的DOM越小越好。

**12：在繁重的执行上使用Web Workers**

这一篇我们将介绍Web Workder，一种可以将繁重操作移到独立进程的方法。

Web Workders在前段时间被引入流行的浏览器中，但是好像并没有被广泛应用。Web Workers的主要功能是在一般浏览器执行范围外执行繁重的方法。它不会访问DOM，所以你必须传入方法涉及的节点。

以下是一段Web Workder的示例代码：

**Javascript代码**

/\* 使用Web Worker \*/

// 启动worker

var worker = new Worker("/path/to/web/worker/resource.js");

worker.addEventListener("message", function(event) {

// 我们从web worker获取信息!

});

// 指导Web Worker工作！

worker.postMessage({ cmd: "processImageData", data: convertImageToDataUri(myImage) });

/\* resource.js就是一个Web workder \*/

self.addEventListener("message", function(event) {

var data = event.data;

switch (data.cmd) {

case 'process':

return processImageData(data.imageData);

});

function processImageData(imageData) {

// 对图像进行操作

// 例如将它改成灰度

return newImageData;

}

以上这段代码是一个教你如何使用Web Worker在其他进程中做一些繁重工作的简单示例。它要执行的是将一个图片从普通颜色转个灰度，因为这是一个比较繁重的过程，所以你可以将这个进程提交给Web Worker，使你的浏览器负载不是很大。Data通过message事件传回。

你可以仔细阅读以下MDN上关于Web Workder的使用，也许在你的网站上有一些功能可以移到其他的独立进程中去执行。

**13：链接CSS，避免使用@import**

有时候，@import太好用以至于很难抗拒它的诱惑，但是为了减少令人抓狂的请求，你必须要拒绝它！最常见的用法是在一个”main”CSS文件中，没有任何的内容，只有@import规则。有时，多个@import规则往往会造成事件嵌套：

**Javascript代码**

// 主CSS文件(main.css)

@import "reset.css";

@import "structure.css";

@import "tutorials.css";

@import "contact.css";

// 然后在tutorials.css文件中，会继续有@import

@import "document.css";

@import "syntax-highlighter.css";

我们这样写CSS文件，在文件中多了两个多余链接，因此会使页面加载变慢。SASS可以读取@import语句，链接CSS内容到一个文件中，减少了多余的请求，控制了CSS文件的大小。

**14：在CSS文件中包含多种介质类型**

在上面第13个最佳实践中我们说过，多个CSS文件可以通过@import规则合并到一起。但是很多程序员不知道的是，多种CSS介质类型也可以合并到一个文件中。

**Javascript代码**

/\* 以下全部写在一个CSS文件中 \*/

@media screen {

/\* 所有默认的结构设计和元素样式写在这里 \*/

}

@media print {

/\* 调整打印时的样式 \*/

}

@media only screen and (max-width : 1024px) {

/\* 使用ipad或者移动电话时的样式设定 \*/

}

对于文件的大小，什么时候必须合并介质，或是什么时候必须分开设定，CSS并没有硬性规定，但是我会比较建议将所有的介质合并，除非其中一个介质所占的比例比起其他大了许多。少一个请求对于客户端和服务器都将轻松不少，而且在大多数情况下，附赠的介质类型相比主屏介质类型要相对小很多。

****15：字符串链接优化****

由于字符串是不可变的，所以在进行字符串连接时，需要创建临时字符串。频繁创建、销毁临时字符串会导致性能低下。

当然，这个问题在新版本浏览器包括IE8+中都得到了优化，所以不需要担心

在低版本浏览器（IE6、IE7）中，我们可以种数组的join方法来代替。

var temp = [];var i = 0;

temp[i++] = "Hello";

temp[i++] = " ";

temp[i++] ="everyone";

var outcome = temp.join("");

****16：快速循环****

1、循环总次数使用局部变量

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

}// 改成var len = arr.length;for( var i = 0;i < len;i++) {

}

这样就避免了每次循环的属性查找。这点尤其重要，因为在进行dom操作时，很多人会这样写：

var divList = document.getElementsByTagName("div");for( var i = 0;i < divList.length;i++) {

}

查找DOM元素的属性是相对耗时的，所以应该避免这种写法。

2、如果可以，递减代替递增

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

}// 改成for(var i = arr.length - 1;i--;) {

}

var i = 0;while(i < arr.length) {

i++;

}// 改成var i = arr.length - 1;while(i--) {

}

i=0的时候会直接跳出，循环次数比较多时还是很有用的。

**17：展开循环**

var i = arr.length - 1;while(i--) {

dosomething(arr[i]);

}

遇到这样的情况时，执行一次循环的时候我们可以选择不止执行一次函数。

var interations = Math.floor(arr.length / 8);var left = arr.length % 8;var i = 0;

if(left) {

do {

dosomething(arr[i++]);

} while(--left);

}do {

dosomething(arr[i++]);

dosomething(arr[i++]);

dosomething(arr[i++]);

dosomething(arr[i++]);

dosomething(arr[i++]);

dosomething(arr[i++]);

dosomething(arr[i++]);

dosomething(arr[i++]);

} while(--interations);

当遇到大数组，减少循环的开销，性能不就提上去了嘛。（至于为什么是每次循环，调8次函数，大牛测出来的，这样达到最佳）

**18：高效存取数据**

JavaScript中4种地方可以存取数据：

字面量值；变量；数组元素；对象属性

字面量值和变量中存取数据是最快的，从数组元素和对象属性中存取数据相对较慢，并且随着深度增加，存取速度会越来越慢，譬如obj.item.value就比obj.item慢。

某些情况下我们可以将对象、数组属性存成局部变量来提高速度，譬如：

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

}// 改成var len = arr.length;for( var i = 0;i < len;i++) {

}

var divList = document.getElementsByTagName("div");for( var i = 0;i < divList.length;i++) {

}// 改成

// var divList = document.getElementsByTagName("div");for( var i = 0,len = divList.length;i < len;i++) {

}

****19：事件委托****

事件委托就是利用冒泡的原理，将原本应该添加在某些元素身上的监听事件，添加到其父元素身上，来达到提高性能的效果。

举个栗子：

<div>

<ul>

<li>1</li>

<li>2</li>

<li>3</li>

<li>4</li>

<li>5</li>

<li>6</li>

<li>7</li>

<li>8</li>

<li>9</li>

<li>10</li>

</ul></div><script>

window.onload = function() {

var ul = document.getElementsByTagName('ul')[0];

var liList = document.getElementsByTagName('li');

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

liList[i].onclick = function() {

alert(this.innerHTML);

}

}

}</script>

这样我们就为每个li添加了监听事件了。

显然，我们通过循环为每个li添加监听事件是不优化的。这样不仅浪费了内存，在新的li加入的时候我们还要重新为它添加监听事件。

我们可以这样写：

<div>

<ul>

<li>1</li>

<li>2</li>

<li>3</li>

<li>4</li>

<li>5</li>

<li>6</li>

<li>7</li>

<li>8</li>

<li>9</li>

<li>10</li>

</ul></div><script>

window.onload = function() {

var ul = document.getElementsByTagName('ul')[0];

var liList = document.getElementsByTagName('li');

ul.onclick = function(e) {

var e = e || window.event;

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

if(target.nodeName.toLowerCase() == "li") {

alert(target.innerHTML);

}

}

}</script>

这样写的好处：

只添加一个监听事件，节省了内存；新加入li的时候我们也不用为它单独添加监听事件；在页面中添加事件处理程序所需的时候更少，因为我们只需要为一个DOM元素添加事件处理程序。

如果是原创文章，转载请注明出处：[http://www.cnblogs.com/MarcoHan](http://www.cnblogs.com/MarcoHan/" \o "MarcoHan" \t "https://www.cnblogs.com/MarcoHan/p/_blank)/

最后，提一点不会提高性能的建议

if( 2 == value) {

}

类似这种判断的时候推荐常量放在左边，这样就可以预防类似 if( value = 2){} 的错误了，因为如果少写了一个等号， if( value = 2) {} 是合法的语句，而且代码量变大的时候不容易检查出来。if( 2 = value) {} 这样少写了等号JavaScript引擎会直接报错，我们就可以愉快地改过来了。（只是建议）

****20：条件判断****

当出现条件判断时，我们采用什么样的结构才能使性能最优？

if(val == 0) {

return v0;

}else if(val == 1) {

return v1;

}else if(val == 2) {

return v2;

}else if(val == 3) {

return v3;

}else if(val == 4) {

return v4;

}

当条件分支比较多时，我们可以斟酌哪种条件出现的概率比较大，并将对应的语句放在最上面，这样可以减少判断次数。

使用switch语句，新版的浏览器基本上都对switch做了优化，这样层数比较深时，性能比if会更好

使用数组：

var v = [v0,v1,v2,v3,v4];return v[valeue];

要求：对应的结果是单一值，而不是一系列操作

另外，其他方面的优化，譬如

if(condition1) {

return v1;

}else {

return v2

}// 改成if(condition1) {

return v1;

}return v2;