# 前端性能优化指南

## AJAX优化

* 缓存AJAX：
  + 异步并不等于即时。
* 请求使用GET：
  + 当使用XMLHttpRequest时，而URL长度不到2K，可以使用GET请求数据，GET相比POST更快速。
    - POST类型请求要发送两个TCP数据包。
      * 先发送文件头。
      * 再发送数据。
    - GET类型请求只需要发送一个TCP数据包。
      * 取决于你的cookie数量。

## COOKIE专题

* 减少COOKIE的大小。
* 使用无COOKIE的域。
  + 比如图片CSS等静态文件放在静态资源服务器上并配置单独域名，客户端请求静态文件的时候，减少COOKIE反复传输时对主域名的影响。

## DOM优化

* 优化节点修改。
  + 使用cloneNode在外部更新节点然后再通过replace与原始节点互换。
* 优化节点添加  
  >多个节点插入操作，即使在外面设置节点的元素和风格再插入，由于多个节点还是会引发多次reflow。
  + 优化的方法是创建DocumentFragment，在其中插入节点后再添加到页面。
    - 如JQuery中所有的添加节点的操作如append，都是最终调用DocumentFragment来实现的，
    - createSafeFragment(document) {
    - var list = nodeNames.split( "|" ),
    - safeFrag = document.createDocumentFragment();
* 优化CSS样式转换。  
  >如果需要动态更改CSS样式，尽量采用触发reflow次数较少的方式。
  + 如以下代码逐条更改元素的几何属性，理论上会触发多次reflow。
  + element.style.fontWeight = 'bold' ;
  + element.style.marginLeft= '30px' ;

element.style.marginRight = '30px' ;

* + 可以通过直接设置元素的className直接设置，只会触发一次reflow。
* 减少DOM元素数量
  + 在console中执行命令查看DOM元素数量。

`document.getElementsByTagName( '\*' ).length`

* + 正常页面的DOM元素数量一般不应该超过1000。
  + DOM元素过多会使DOM元素查询效率，样式表匹配效率降低，是页面性能最主要的瓶颈之一。
* DOM操作优化。
  + DOM操作性能问题主要有以下原因。
    - DOM元素过多导致元素定位缓慢。
    - 大量的DOM接口调用。
      * JAVASCRIPT和DOM之间的交互需要通过函数API接口来完成，造成延时，尤其是在循环语句中。
    - DOM操作触发频繁的reflow(layout)和repaint。
    - layout发生在repaint之前，所以layout相对来说会造成更多性能损耗。
      * reflow(layout)就是计算页面元素的几何信息。
      * repaint就是绘制页面元素。
    - 对DOM进行操作会导致浏览器执行回流reflow。
  + 解决方案。
    - 纯JAVASCRIPT执行时间是很短的。
    - 最小化DOM访问次数，尽可能在js端执行。
    - 如果需要多次访问某个DOM节点，请使用局部变量存储对它的引用。
    - 谨慎处理HTML集合（HTML集合实时连系底层文档），把集合的长度缓存到一个变量中，并在迭代中使用它，如果需要经常操作集合，建议把它拷贝到一个数组中。
    - 如果可能的话，使用速度更快的API，比如querySelectorAll和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，并减少访问布局的次数。
    - 动画中使用绝对定位，使用拖放代理。
    - 使用事件委托来减少事件处理器的数量。
* 优化DOM交互  
  >在JAVASCRIPT中，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。
  + 最小化现场更新。
    - 当需要访问的DOM部分已经已经被渲染为页面中的一部分，那么DOM操作和交互的过程就是再进行一次现场更新。
      * 现场更新是需要针对现场（相关显示页面的部分结构）立即进行更新，每一个更改（不管是插入单个字符还是移除整个片段），都有一个性能损耗。
      * 现场更新进行的越多，代码完成执行所花的时间也越长。
  + 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。
      * 使用innerHTML。
        + 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JAVASCRIPT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。
* 回流reflow。
  + 发生场景。
    - 改变窗体大小。
    - 更改字体。
    - 添加移除stylesheet块。
    - 内容改变哪怕是输入框输入文字。
    - CSS虚类被触发如 :hover。
    - 更改元素的className。
    - 当对DOM节点执行新增或者删除操作或内容更改时。
    - 动态设置一个style样式时（比如element.style.width="10px"）。
    - 当获取一个必须经过计算的尺寸值时，比如访问offsetWidth、clientHeight或者其他需要经过计算的CSS值。
  + 解决问题的关键，就是限制通过DOM操作所引发回流的次数。
    - 在对当前DOM进行操作之前，尽可能多的做一些准备工作，保证N次创建，1次写入。
    - 在对DOM操作之前，把要操作的元素，先从当前DOM结构中删除：
      * 通过removeChild()或者replaceChild()实现真正意义上的删除。
      * 设置该元素的display样式为“none”。
    - 每次修改元素的style属性都会触发回流操作。
    - element.style.backgroundColor = "blue";
      * 使用更改className的方式替换style.xxx=xxx的方式。
      * 使用style.cssText = '';一次写入样式。
      * 避免设置过多的行内样式。
      * 添加的结构外元素尽量设置它们的位置为fixed或absolute。
      * 避免使用表格来布局。
      * 避免在CSS中使用JavaScript expressions(IE only)。
    - 将获取的DOM数据缓存起来。这种方法，对获取那些会触发回流操作的属性（比如offsetWidth等）尤为重要。
    - 当对HTMLCollection对象进行操作时，应该将访问的次数尽可能的降至最低，最简单的，你可以将length属性缓存在一个本地变量中，这样就能大幅度的提高循环的效率。

## eval优化

* 避免eval：
  + eval会在时间方面带来一些效率，但也有很多缺点。
    - eval会导致代码看起来更脏。
    - eval会需要消耗大量时间。
    - eval会逃过大多数压缩工具的压缩。

## HTML优化

* 插入HTML。
  + JavaScript中使用document.write生成页面内容会效率较低，可以找一个容器元素，比如指定一个div，并使用innerHTML来将HTML代码插入到页面中。
* 避免空的src和href。
  + 当link标签的href属性为空、script标签的src属性为空的时候，浏览器渲染的时候会把当前页面的URL作为它们的属性值，从而把页面的内容加载进来作为它们的值。
* 为文件头指定Expires。
  + 使内容具有缓存性，避免了接下来的页面访问中不必要的HTTP请求。
* 重构HTML，把重要内容的优先级提高。
* Post-load（次要加载）不是必须的资源。
* 利用预加载优化资源。
* 合理架构，使DOM结构尽量简单。
* 利用LocalStorage合理缓存资源。
* 尽量避免CSS表达式和滤镜。
* 尝试使用defer方式加载Js脚本。
* 新特性：will-change，把即将发生的改变预先告诉浏览器。
* 新特性Beacon，不堵塞队列的异步数据发送。
* 不同之处：网络缓慢，缓存更小，不令人满意的浏览器处理机制。
* 尽量多地缓存文件。
* 使用HTML5 Web Workers来允许多线程工作。
* 为不同的Viewports设置不同大小的Content。
* 正确设置可Tap的目标的大小。
* 使用响应式图片。
* 支持新接口协议（如HTTP2）。
* 未来的缓存离线机制：Service Workers。
* 未来的资源优化Resource Hints(preconnect, preload, 和prerender)。
* 使用Server-sent Events。
* 设置一个Meta Viewport。

## JIT与GC优化

* untyped（无类型）。
  + JAVASCRIPT是个无类型的语言，这导致了如x=y+ｚ这种表达式可以有很多含义。
    - y，z是数字，则+表示加法。
    - y，z是字符串，则+表示字符串连接。

而JS引擎内部则使用“细粒度”的类型，比如：

* + - 32-bit\* integer。
    - 64-bit\* floating-point。

这就要求js类型-js引擎类型，需要做“boxed/unboxed（装箱/解箱）”，在处理一次x=y+z这种计算，需要经过的步骤如下。

* + - 从内存，读取x=y+z的操作符。
    - 从内存，读取y，z。
    - 检查y，z类型，确定操作的行为。
    - unbox y，z。
    - 执行操作符的行为。
    - box x。
    - 把x写入内存。

只有第5步骤是真正有效的操作，其他步骤都是为第5步骤做准备/收尾，JAVASCRIPT的untyped特性很好用，但也为此付出了很大的性能代价。

* JIT。
  + 先看看JIT对untyped的优化，在JIT下，执行x=y+z流程。
    - 从内存，读取x=y+z的操作符。
    - 从内存，读取 y，z。
    - 检查y，z类型，确定操作的行为。
    - unbox y，z。
    - 执行 操作符 的行为。
    - box x。
    - 把x写入内存。
  + 新引擎还对“对象属性”访问做了优化，解决方案叫inline caching，简称：IC。简单的说，就是做cache。但如果当list很大时，这种方案反而影响效率。
* Type-specializing JIT  
  >Type-specializing JIT引擎用来处理typed类型（声明类型）变量，但JAVASCRIPT都是untype类型的。
  + Type-specializing JIT的解决方案是：
    - 先通过扫描，监测类型。
    - 通过编译优化（优化对象不仅仅只是“类型”，还包括对JS代码的优化，但核心是类型优化），生成类型变量。
    - 再做后续计算。
  + Type-specializing JIT的执行x=y+z流程：
    - * 从内存，读取x=y+z的操作符。
      * 从内存，读取y，z。
      * 检查y，z类型，确定操作的行为。
      * unbox y，z。
      * 执行操作符的行为。
      * box x。
      * 把x写入内存。
      * 代价是：
    - 前置的扫描类型
    - 编译优化。

所以·Type-specializing JIT·的应用是有选择性，选择使用这个引擎的场景包括：

* + - 热点代码。
    - 通过启发式算法估算出来的有价值的代码。

另外，有2点也需要注意：

* + - 当变量类型 发生变化时，引擎有2种处理方式：
      * 少量变更，重编译，再执行。
      * 大量变更，交给JIT执行。
    - 数组，object properties， 闭包变量 不在优化范畴之列。

## js载入优化

* 加快JavaScript装入速度的工具：
  + Lab.js
    - 借助LAB.js（装入和阻止JavaScript），你就可以并行装入JavaScript文件，加快总的装入过程。此外，你还可以为需要装入的脚本设置某个顺序，那样就能确保依赖关系的完整性。此外，开发者声称其网站上的速度提升了2倍。
* 使用适当的CDN：
  + 现在许多网页使用内容分发网络（CDN）。它可以改进你的缓存机制，因为每个人都可以使用它。它还能为你节省一些带宽。你很容易使用ping检测或使用Firebug调试那些服务器，以便搞清可以从哪些方面加快数据的速度。选择CDN时，要照顾到你网站那些访客的位置。记得尽可能使用公共存储库。
* 网页末尾装入JavaScript：
  + 也可以在头部分放置需要装入的一些JavaScript，但是前提是它以异步方式装入。
* 异步装入跟踪代码：  
  >脚本加载与解析会阻塞HTML渲染，可以通过异步加载方式来避免渲染阻塞，步加载的方式很多，比较通用的方法如下。
* var \_gaq = \_gaq || [];
* \_gaq.push(['\_setAccount', 'UA-XXXXXXX-XX']);
* \_gaq.push(['\_trackPageview']);
* (function() {
* var ga = document.createElement('script'); ga.type = 'text/JavaScript'; ga.async = true;
* ga.src = ('https:' == document.location.protocol ? 'https://ssl' : 'http://www') + '.google-analytics.com/ga.js';
* var s = document.getElementsByTagName('script')[0]; s.parentNode.insertBefore(ga, s);
* })();

或者

function loadjs (script\_filename){

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

script.setAttribute( 'type' , 'text/javascript' );

script.setAttribute( 'src' , script\_filename);

script.setAttribute( 'id' , 'script-id' );

scriptElement = document.getElementById( 'script-id' );

if (scriptElement){

document.getElementsByTagName( 'head' )[0].removeChild(scriptElement);

}

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

}

var script = 'scripts/alert.js' ;

loadjs(script);

* 把你的JavaScript打包成PNG文件
  + 将JavaScript/css数据打包成PNG文件。之后进行拆包，只要使用画布API的getImageData()。可以在不缩小数据的情况下，多压缩35%左右。而且是无损压缩，对比较庞大的脚本来说，在图片指向画布、读取像素的过程中，你会觉得有“一段”装入时间。
* 设置Cache-Control和Expires头

通过Cache-Control和Expires头可以将脚本文件缓存在客户端或者代理服务器上，可以减少脚本下载的时间。  
>  
Expires格式:  
>

Expires = "Expires" ":" HTTP-date

Expires: Thu, 01 Dec 1994 16:00:00 GMT

Note: if a response includes a Cache-Control field with the max-age directive that directive overrides the

Expires field.

>  
Cache-Control格式：  
>

Cache-Control = "Cache-Control" ":" 1#cache-directive

Cache-Control: public

具体的标准定义可以参考http1.1中的定义，简单来说Expires控制过期时间是多久，Cache-Control控制什么地方可以缓存 。

## with优化

* 尽可能地少用with语句，因为它会增加with语句以外的数据的访问代价。
* 避免使用with
* >
* `with`语句将一个新的可变对象推入作用域链的头部，函数的所有局部变量现在处于第二个作用域链对象中，从而使局部变量的访问代价提高。

## 变量专题

* 全局变量
  + 当一个变量被定义在全局作用域中，默认情况下JAVASCRIPT引擎就不会将其回收销毁。如此该变量就会一直存在于老生代堆内存中，直到页面被关闭。
  + 全局变量缺点。
    - 使变量不易被回收。
      * 多人协作时容易产生混淆。
    - 在作用域链中容易被干扰。
  + 可以通过包装函数来处理全局变量。
* 局部变量。
  + 尽量选用局部变量而不是全局变量。
  + 局部变量的访问速度要比全局变量的访问速度更快，因为全局变量其实是window对象的成员，而局部变量是放在函数的栈里的。
* 手工解除变量引用
  + 在业务代码中，一个变量已经确定不再需要了，那么就可以手工解除变量引用，以使其被回收。
* 变量查找优化。
  + 变量声明带上var，如果声明变量忘记了var，那么JAVASCRIPT引擎将会遍历整个作用域查找这个变量，结果不管找到与否，都会造成性能损耗。
    - 如果在上级作用域找到了这个变量，上级作用域变量的内容将被无声的改写，导致莫名奇妙的错误发生。
    - 如果在上级作用域没有找到该变量，这个变量将自动被声明为全局变量，然而却都找不到这个全局变量的定义。
  + 慎用全局变量。
    - 全局变量需要搜索更长的作用域链。
    - 全局变量的生命周期比局部变量长，不利于内存释放。
    - 过多的全局变量容易造成混淆，增大产生bug的可能性。  
      >
  + 具有相同作用域变量通过一个var声明。
  + jQuery.extend = jQuery.fn.extend = function () {
  + var options,
  + name,
  + src,
  + copy,
  + copyIsArray,
  + clone,target = arguments[0] || {},
  + i = 1,
  + length = arguments.length,
  + deep = false ;

}

* + 缓存重复使用的全局变量。
    - 全局变量要比局部变量需要搜索的作用域长
    - 重复调用的方法也可以通过局部缓存来提速
    - 该项优化在IE上体现比较明显
* 善用回调。
  + 除了使用闭包进行内部变量访问，我们还可以使用现在十分流行的回调函数来进行业务处理。
  + function getData(callback) {
  + var data = 'some big data';
  + callback(null, data);
  + }
  + getData(function(err, data) {
  + console.log(data);

});

* + - 回调函数是一种后续传递风格(Continuation Passing Style, CPS)的技术，这种风格的程序编写将函数的业务重点从返回值转移到回调函数中去。而且其相比闭包的好处也有很多。
      * 如果传入的参数是基础类型（如字符串、数值），回调函数中传入的形参就会是复制值，业务代码使用完毕以后，更容易被回收。
      * 通过回调，我们除了可以完成同步的请求外，还可以用在异步编程中，这也就是现在非常流行的一种编写风格。
      * 回调函数自身通常也是临时的匿名函数，一旦请求函数执行完毕，回调函数自身的引用就会被解除，自身也得到回收。

## 常规优化

* 传递方法取代方法字符串

一些方法例如setTimeout()、setInterval()，接受字符串或者方法实例作为参数。直接传递方法对象作为参数来避免对字符串的二次解析。

* + - 传递方法
    - setTimeout(test, 1);
  + 传递方法字符串
* 使用原始操作代替方法调用

方法调用一般封装了原始操作，在性能要求高的逻辑中，可以使用原始操作代替方法调用来提高性能。

* + 原始操作

var min = a<b?a:b;

* + 方法实例
* 定时器

如果针对的是不断运行的代码，不应该使用setTimeout，而应该是用setInterval。setTimeout每次要重新设置一个定时器。

* 避免双重解释

当JAVASCRIPT代码想解析JAVASCRIPT代码时就会存在双重解释惩罚，双重解释一般在使用eval函数、new Function构造函数和setTimeout传一个字符串时等情况下会遇到，如。

eval("alert('hello world');");

var sayHi = new Function("alert('hello world');");

setTimeout("alert('hello world');", 100);

> 上述alert('hello world');语句包含在字符串中，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，而实例化一个新的解析器有很大的性能损耗。

我们看看下面的例子：

var sum, num1 = 1, num2 = 2;

/\*\*效率低\*\*/

for(var i = 0; i < 10000; i++){

var func = new Function("sum+=num1;num1+=num2;num2++;");

func();

//eval("sum+=num1;num1+=num2;num2++;");

}

/\*\*效率高\*\*/

for(var i = 0; i < 10000; i++){

sum+=num1;

num1+=num2;

num2++;

}

第一种情况我们是使用了new Function来进行双重解释，而第二种是避免了双重解释。

* 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。
* 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

* + 多个变量声明
  + /\*\*不提倡\*\*/
  + 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();
  + 插入迭代值
  + /\*\*不提倡\*\*/
  + var name = values[i];
  + i++;
  + /\*\*提倡\*\*/
  + var name = values[i++];
  + 使用数组和对象字面量，避免使用构造函数Array(),Object()
  + /\*\*不提倡\*\*/
  + 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
  + };
* 避免使用属性访问方法
  + JavaScript不需要属性访问方法，因为所有的属性都是外部可见的。
  + 添加属性访问方法只是增加了一层重定向 ，对于访问控制没有意义。
  + 使用属性访问方法示例
  + function Car() {
  + this .m\_tireSize = 17;
  + this .m\_maxSpeed = 250;
  + this .GetTireSize = Car\_get\_tireSize;
  + this .SetTireSize = Car\_put\_tireSize;
  + }
  + function Car\_get\_tireSize() {
  + return this .m\_tireSize;
  + }
  + function Car\_put\_tireSize(value) {
  + this .m\_tireSize = value;
  + }
  + var ooCar = new Car();
  + var iTireSize = ooCar.GetTireSize();
  + ooCar.SetTireSize(iTireSize + 1);
  + 直接访问属性示例
  + function Car() {
  + this .m\_tireSize = 17;
  + this .m\_maxSpeed = 250;
  + }
  + var perfCar = new Car();
  + var iTireSize = perfCar.m\_tireSize;
  + perfCar.m\_tireSize = iTireSize + 1;
* 减少使用元素位置操作
  + 一般浏览器都会使用增量reflow的方式将需要reflow的操作积累到一定程度然后再一起触发，但是如果脚本中要获取以下属性，那么积累的reflow将会马上执行，已得到准确的位置信息。

## 代码压缩

* 代码压缩工具

> 精简代码就是将代码中的空格和注释去除，也有更进一步的会对变量名称混淆、精简。根据统计精简后文件大小会平均减少21%，即使Gzip之后文件也会减少5%。

* + YUICompressor
  + Dean Edwards Packer
  + JSMin
  + GZip压缩
    - GZip缩短在浏览器和服务器之间传送数据的时间，缩短时间后得到标题是Accept-Encoding:gzip,deflate的一个文件。不过这种压缩方法同样也有缺点。
      * 它在服务器端和客户端都要占用处理器资源（以便压缩和解压缩）。
      * 占用磁盘空间。
    - Gzip通常可以减少70%网页内容的大小，包括脚本、样式表、图片等任何一个文本类型的响应，包括XML和JSON。Gzip比deflate更高效，主流服务器都有相应的压缩支持模块。
    - Gzip的工作流程为
      * 客户端在请求Accept-Encoding中声明可以支持Gzip。
      * 服务器将请求文档压缩，并在Content-Encoding中声明该回复为Gzip格式。
      * 客户端收到之后按照Gzip解压缩。
  + Closure compiler

## 代码优化

* 优化原则：
* JS优化总是出现在大规模循环的地方：
* 这倒不是说循环本身有性能问题，而是循环会迅速放大可能存在的性能问题，所以第二原则就是以大规模循环体为最主要优化对象。

以下的优化原则，只在大规模循环中才有意义，在循环体之外做此类优化基本上是没有意义的。

目前绝大多数JS引擎都是解释执行的，而解释执行的情况下，在所有操作中，函数调用的效率是较低的。此外，过深的prototype继承链或者多级引用也会降低效率。JScript中，10级引用的开销大体是一次空函数调用开销的1/2。这两者的开销都远远大于简单操作（如四则运算）。

* 尽量避免过多的引用层级和不必要的多次方法调用：
* 特别要注意的是，有些情况下看似是属性访问，实际上是方法调用。例如所有DOM的属性，实际上都是方法。在遍历一个NodeList的时候，循环 条件对于nodes.length的访问，看似属性读取，实际上是等价于函数调用的。而且IE DOM的实现上，childNodes.length每次是要通过内部遍历重新计数的。（My god，但是这是真的！因为我测过，childNodes.length的访问时间与childNodes.length的值成正比！）这非常耗费。所以 预先把nodes.length保存到js变量，当然可以提高遍历的性能。

同样是函数调用，用户自定义函数的效率又远远低于语言内建函数，因为后者是对引擎本地方法的包装，而引擎通常是c,c++,java写的。进一步，同样的功能，语言内建构造的开销通常又比内建函数调用要效率高，因为前者在JS代码的parse阶段就可以确定和优化。

* 尽量使用语言本身的构造和内建函数：

## 动画优化

* 动画效果在缺少硬件加速支持的情况下反应缓慢，例如手机客户端。
  + 特效应该只在确实能改善用户体验时才使用，而不应用于炫耀或者弥补功能与可用性上的缺陷。
  + 至少要给用户一个选择可以禁用动画效果。
  + 设置动画元素为absolute或fixed。
    - position: static或position: relative元素应用动画效果会造成频繁的reflow。
    - position: absolute或position: fixed的元素应用动画效果只需要repaint。
  + 使用一个timer完成多个元素动画。
    - setInterval和setTimeout是两个常用的实现动画的接口，用以间隔更新元素的风格与布局。。
  + 动画效果的帧率最优化的情况是使用一个timer完成多个对象的动画效果，其原因在于多个timer的调用本身就会损耗一定性能。
  + setInterval(function() {
  + animateFirst('');
  + }, 10);
  + setInterval(function() {
  + animateSecond('');
  + }, 10);
  + 使用同一个`timer`。
* 以脚本为基础的动画，由浏览器控制动画的更新频率。

## 对象专题

* 减少不必要的对象创建：
  + 创建对象本身对性能影响并不大，但由于JAVASCRIPT的垃圾回收调度算法，导致随着对象个数的增加，性能会开始严重下降（复杂度O(n^2)）。
    - 如常见的字符串拼接问题，单纯的多次创建字符串对象其实根本不是降低性能的主要原因，而是是在对象创建期间的无谓的垃圾回收的开销。而Array.join的方式，不会创建中间字符串对象，因此就减少了垃圾回收的开销。
  + 复杂的JAVASCRIPT对象，其创建时时间和空间的开销都很大，应该尽量考虑采用缓存。
  + 尽量作用JSON格式来创建对象，而不是var obj=new Object()方法。前者是直接复制，而后者需要调用构造器。
* 对象查找
  + 避免对象的嵌套查询，因为JAVASCRIPT的解释性，a.b.c.d.e嵌套对象，需要进行4次查询，嵌套的对象成员会明显影响性能。
  + 如果出现嵌套对象，可以利用局部变量，把它放入一个临时的地方进行查询。
* 对象属性
  + 访问对象属性消耗性能过程（JAVASCRIPT对象存储）。
    - 先从本地变量表找到对象。
    - 然后遍历属性。
    - 如果在当前对象的属性列表里没找到。
    - 继续从prototype向上查找。
    - 且不能直接索引，只能遍历。

## 服务端优化

* 避免404。
  + 更改404错误响应页面可以改进用户体验，但是同样也会浪费服务器资源。
  + 指向外部JAVASCRIPT的链接出现问题并返回404代码。
    - 这种加载会破坏并行加载。
    - 其次浏览器会把试图在返回的404响应内容中找到可能有用的部分当作JavaScript代码来执行。
* 删除重复的JAVASCRIPT和CSS。
  + 重复调用脚本缺点。
    - 增加额外的HTTP请求。
    - 多次运算也会浪费时间。在IE和Firefox中不管脚本是否可缓存，它们都存在重复运算JAVASCRIPT的问题。
* ETags配置Entity标签。
  + ETags用来判断浏览器缓存里的元素是否和原来服务器上的一致。
    - 与last-modified date相比更灵活。
* 权衡DNS查找次数
  + 减少主机名可以节省响应时间。但同时也会减少页面中并行下载的数量。
    - IE浏览器在同一时刻只能从同一域名下载两个文件。当在一个页面显示多张图片时，IE用户的图片下载速度就会受到影响。
* 通过Keep-alive机制减少TCP连接。
* 通过CDN减少延时。
* 平行处理请求（参考BigPipe）。
* 通过合并文件或者Image Sprites减少HTTP请求。
* 减少重定向（ HTTP 301和40x/50x）。

## 类型转换专题

* 把数字转换成字符串。
  + 应用""+1，效率是最高。
    - 性能上来说：""+字符串>String()>.toString()>new String()。
      * String()属于内部函数，所以速度很快。
      * .toString()要查询原型中的函数，所以速度略慢。
      * new String()最慢。
* 浮点数转换成整型。
  + 错误使用使用parseInt()。
    - parseInt()是用于将字符串转换成数字，而不是浮点数和整型之间的转换。
  + 应该使用Math.floor()或者Math.round()。
    - Math是内部对象，所以Math.floor()其实并没有多少查询方法和调用的时间，速度是最快的。

## 逻辑判断优化

* switch语句。
  + 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。

## 内存专题

* JAVASCRIPT的内存回收机制
  + 以Google的V8引擎为例，在V8引擎中所有的JAVASCRIPT对象都是通过堆来进行内存分配的。当我们在代码中声明变量并赋值时，V8引擎就会在堆内存中分配一部分给这个变量。如果已申请的内存不足以存储这个变量时，V8引擎就会继续申请内存，直到堆的大小达到了V8引擎的内存上限为止（默认情况下，V8引擎的堆内存的大小上限在64位系统中为1464MB，在32位系统中则为732MB）。
  + 另外，V8引擎对堆内存中的JAVASCRIPT对象进行分代管理。
    - 新生代。
      * 新生代即存活周期较短的JAVASCRIPT对象，如临时变量、字符串等
    - 老生代。
      * 老生代则为经过多次垃圾回收仍然存活，存活周期较长的对象，如主控制器、服务器对象等。
* 垃圾回收算法。
  + 垃圾回收算法一直是编程语言的研发中是否重要的​​一环，而V8引擎所使用的垃圾回收算法主要有以下几种。
    - Scavange算法：通过复制的方式进行内存空间管理，主要用于新生代的内存空间；
    - Mark-Sweep算法和Mark-Compact算法：通过标记来对堆内存进行整理和回收，主要用于老生代对象的检查和回收。
* 对象进行回收。
  + 引用。
    - 当函数执行完毕时，在函数内部所声明的对象不一定就会被销毁。
    - 引用(Reference)是JAVASCRIPT编程中十分重要的一个机制。
      * + 是指代码对对象的访问这一抽象关系，它与C/C++的指针有点相似，但并非同物。引用同时也是JAVASCRIPT引擎在进行垃圾回收中最关键的一个机制。
        + var val = 'hello world';
        + function foo() {
        + return function() {
        + return val;
        + };
        + }
        + global.bar = foo();
      * 当代码执行完毕时，对象val和bar()并没有被回收释放，JAVASCRIPT代码中，每个变量作为单独一行而不做任何操作，JAVASCRIPT引擎都会认为这是对对象的访问行为，存在了对对象的引用。为了保证垃圾回收的行为不影响程序逻辑的运行，JAVASCRIPT引擎不会把正在使用的对象进行回收。所以判断对象是否正在使用中的标准，就是是否仍然存在对该对象的引用。
* JAVASCRIPT的引用是可以进行转移的，那么就有可能出现某些引用被带到了全局作用域，但事实上在业务逻辑里已经不需要对其进行访问了，这个时候就应该被回收，但是JAVASCRIPT引擎仍会认为程序仍然需要它。
* IE下闭包引起跨页面内存泄露。
* JAVASCRIPT的内存泄露处理
  + 给DOM对象添加的属性是一个对象的引用。
  + var MyObject = {};
  + document.getElementByIdx\_x('myDiv').myProp = MyObject;
  + 解决方法：在window.onunload事件中写上:
  + document.getElementByIdx\_x('myDiv').myProp = null;
  + DOM对象与JS对象相互引用。
  + function Encapsulator(element) {
  + this.elementReference = element;
  + element.myProp = this;
  + }
  + new Encapsulator(document.getElementByIdx\_x('myDiv'));
  + 解决方法：在onunload事件中写上:
  + document.getElementByIdx\_x('myDiv').myProp = null;
  + 给DOM对象用attachEvent绑定事件。
  + function doClick() {}
  + element.attachEvent("onclick", doClick);
  + 解决方法：在onunload事件中写上:
  + element.detachEvent('onclick', doClick);
  + 从外到内执行appendChild。这时即使调用removeChild也无法释放。
  + var parentDiv = document.createElement\_x("div");
  + var childDiv = document.createElement\_x("div");
  + document.body.appendChild(parentDiv);
  + parentDiv.appendChild(childDiv);
  + 解决方法：从内到外执行appendChild:
  + var parentDiv = document.createElement\_x("div");
  + var childDiv = document.createElement\_x("div");
  + parentDiv.appendChild(childDiv);
  + document.body.appendChild(parentDiv);
  + 反复重写同一个属性会造成内存大量占用(但关闭IE后内存会被释放)。
  + for(i = 0; i < 5000; i++) {
  + hostElement.text = "asdfasdfasdf";
  + }
* 内存不是缓存。
  + 不要轻易将内存当作缓存使用。
  + 如果是很重要的资源，请不要直接放在内存中，或者制定过期机制，自动销毁过期缓存。
* CollectGarbage。
  + CollectGarbage是IE的一个特有属性,用于释放内存的使用方法,将该变量或引用对象设置为null或delete然后在进行释放动作，在做CollectGarbage前,要必需清楚的两个必备条件:（引用）。
    - 一个对象在其生存的上下文环境之外，即会失效。
    - 一个全局的对象在没有被执用(引用)的情况下，即会失效

## 事件优化

* 使用事件代理
  + 当存在多个元素需要注册事件时，在每个元素上绑定事件本身就会对性能有一定损耗。
  + 由于DOM Level2事件模 型中所有事件默认会传播到上层文档对象，可以借助这个机制在上层元素注册一个统一事件对不同子元素进行相应处理。

捕获型事件先发生。两种事件流会触发DOM中的所有对象，从document对象开始，也在document对象结束。

<ul id="parent-list">

<li id="post-1">Item 1

<li id="post-2">Item 2

<li id="post-3">Item 3

<li id="post-4">Item 4

<li id="post-5">Item 5

<li id="post-6">Item 6

</li></ul>

// Get the element, add a click listener...

document.getElementById("parent-list").addEventListener("click",function(e) {

// e.target is the clicked element!

// If it was a list item

if(e.target && e.target.nodeName == "LI") {

// List item found! Output the ID!

console.log("List item ",e.target.id.replace("post-")," was clicked!");

}

});

## 数组专题

* 当需要使用数组时，可使用JSON格式的语法
  + 即直接使用如下语法定义数组：[parrm,param,param...],而不是采用new Array(parrm,param,param...)这种语法。使用JSON格式的语法是引擎直接解释。而后者则需要调用Array的构造器。
* 如果需要遍历数组，应该先缓存数组长度，将数组长度放入局部变量中，避免多次查询数组长度。
  + 根据字符串、数组的长度进行循环，而通常这个长度是不变的，比如每次查询a.length，就要额外进行一个操作，而预先把var len=a.length，则每次循环就少了一次查询。

## 同域跨域

* 避免跳转
  + 同域：注意避免反斜杠 “/” 的跳转；
  + 跨域：使用Alias或者mod\_rewirte建立CNAME（保存域名与域名之间关系的DNS记录）

## 性能测试工具

* js性能优化和内存泄露问题及检测分析工具
  + - 性能优化ajax工具diviefirebug
    - [web性能分析工具YSlow]
      * performance性能评估打分，右击箭头可看到改进建议。
      * stats缓存状态分析，传输内容分析。
      * components所有加载内容分析，可以查看传输速度，找出页面访问慢的瓶颈。
      * tools可以查看js和css，并打印页面评估报告。
    - 内存泄露检测工具sIEve
      * sIEve是基于IE的内存泄露检测工具，需要下载运行，可以查看dom孤立节点和内存泄露及内存使用情况。
        1. 列出当前页面内所有dom节点的基本信息(html id style 等)
        2. 页面内所有dom节点的高级信息 (内存占用,数量,节点的引用)
        3. 可以查找出页面中的孤立节点
        4. 可以查找出页面中的循环引用
        5. 可以查找出页面中产生内存泄露的节点
    - 内存泄露提示工具leak monitor
      * leak monitor在安装后，当离开一个页面时，比如关闭窗口，如果页面有内存泄露，会弹出一个文本框进行即时提示。
    - 代码压缩工具
      * YUI压缩工具
      * Dean Edwards Packer
      * JSMin
  + Blink/Webkit浏览器
    - 在Blink/Webkit浏览器中（Chrome, Safari, Opera），我们可以借助其中的Developer Tools的Profiles工具来对我们的程序进行内存检查。
* Node.js中的内存检查
  + 在Node.js中，我们可以使用node-heapdump和node-memwatch模块进​​行内存检查。
  + var heapdump = require('heapdump');
  + var fs = require('fs');
  + var path = require('path');
  + fs.writeFileSync(path.join(\_\_dirname, 'app.pid'), process.pid);

在业务代码中引入node-heapdump之后，我们需要在某个运行时期，向Node.js进程发送SIGUSR2信号，让node-heapdump抓拍一份堆内存的快照。

$ kill -USR2 (cat app.pid)

* 分析浏览器提供的Waterfall图片来思考优化入口。
* 新的测试手段（Navigation, Resource, 和User timing。

## 循环专题

* 循环是一种常用的流程控制。
  + JAVASCRIPT提供了三种循环。
    - for(;;)。
      * 推荐使用for循环，如果循环变量递增或递减，不要单独对循环变量赋值，而应该使用嵌套的++或–-运算符。
      * 代码的可读性对于for循环的优化。
      * 用-=1。
      * 从大到小的方式循环（这样缺点是降低代码的可读性）。
    - while()。
      * for(;;)、while()循环的性能基本持平。
    - for(in)。
      * 在这三种循环中for(in)内部实现是构造一个所有元素的列表，包括array继承的属性，然后再开始循环，并且需要查询hasOwnProperty。所以for(in)相对for(;;)循环性能要慢。
* 选择正确的方法
  + 避免不必要的属性查找。
    - 访问变量或数组是O(1)操作。
    - 访问对象上的属性是一个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。
  + 优化循环。
    - 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。
  + 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。
* 避免在循环中使用try-catch。
  + try-catch-finally语句在catch语句被执行的过程中会动态构造变量插入到当前域中，对性能有一定影响。
  + 如果需要异常处理机制，可以将其放在循环外层使用。
    - * 循环中使用try-catch
      * for ( var i = 0; i < 200; i++) {
      * try {} catch (e) {}
      * }
    - 循环外使用try-catch
    - try {
    - for ( var i = 0; i < 200; i++) {}
    - } catch (e) {}
* 避免遍历大量元素：
  + 避免对全局DOM元素进行遍历，如果parent已知可以指定parent在特定范围查询。
  + var elements = document.getElementsByTagName( '\*' );
  + for (i = 0; i < elements.length; i++) {
  + if (elements[i].hasAttribute( 'selected' )) {}
  + }
  + 如果已知元素存在于一个较小的范围内，

## 原型优化

* 通过原型优化方法定义。
  + 如果一个方法类型将被频繁构造，通过方法原型从外面定义附加方法，从而避免方法的重复定义。
  + 可以通过外部原型的构造方式初始化值类型的变量定义。（这里强调值类型的原因是，引用类型如果在原型中定义，一个实例对引用类型的更改会影响到其他实例。）
    - 这条规则中涉及到JAVASCRIPT中原型的概念，构造函数都有一个prototype属性，指向另一个对象。这个对象的所有属性和方法，都会被构造函数的实例继承。可以把那些不变的属性和方法，直接定义在prototype对象上。
      * 可以通过对象实例访问保存在原型中的值。
      * 不能通过对象实例重写原型中的值。
      * 在实例中添加一个与实例原型同名属性，那该属性就会屏蔽原型中的属性。
      * 通过delete操作符可以删除实例中的属性。

## 运算符专题

* 使用运算符时，尽量使用+＝，－＝、\*＝、\=等运算符号，而不是直接进行赋值运算。
* 位运算。
  + 当进行数学运算时位运算较快，位运算操作要比任何布尔运算或算数运算快，如取模，逻辑与和逻辑或也可以考虑用位运算来替换。

## 重绘专题

* 减少页面的重绘。
  + 减少页面重绘虽然本质不是JAVASCRIPT优化，但重绘往往是由JAVASCRIPT引起的，而重绘的情况直接影响页面性能。
  + var str = "<div>这是一个测试字符串</div>";
  + /\*\*效率低\*\*/
  + var obj = document.getElementsByTagName("body");
  + for(var i = 0; i < 100; i++){
  + obj.innerHTML += str + i;
  + }
  + /\*\*效率高\*\*/
  + var obj = document.getElementsByTagName("body");
  + var arr = [];
  + for(var i = 0; i < 100; i++){
  + arr[i] = str + i;
  + }
  + obj.innerHTML = arr.join("");

一般影响页面重绘的不仅仅是innerHTML，如果改变元素的样式，位置等情况都会触发页面重绘，所以在平时一定要注意这点。

* 使用HTML5和CSS3的一些新特性。
* 避免在HTML里面缩放图片。
* 避免使用插件。
* 确保使用正确的字体大小。
* 决定当前页面是不是能被访问。

## 字符串专题

* 对字符串进行循环操作。
  + 替换、查找等操作，使用正则表达式。
    - 因为JAVASCRIPT的循环速度较慢，而正则表达式的操作是用C写成的API，性能比较好。
* 字符串的拼接。
  + 字符串的拼接在我们开发中会经常遇到，所以我把其放在首位，我们往往习惯的直接用+=的方式来拼接字符串，其实这种拼接的方式效率非常的低，我们可以用一种巧妙的方法来实现字符串的拼接，那就是利用数组的join方法，具体请看我整理的：[Web前端开发规范文档](http://kang.cool/modules/web_develop_standard/index.html%20)中的javaScript书写规范倒数第三条目。
  + 不过也有另一种说法，通常认为需要用Array.join的方式，但是由于SpiderMonkey等引擎对字符串的“+”运算做了优化，结果使用Array.join的效率反而不如直接用“+”，但是如果考虑IE6，则其他浏览器上的这种效率的差别根本不值一提。具体怎么取舍，诸君自定。

## 作用域链和闭包优化

* 作用域。
  + 作用域(scope)是JAVASCRIPT编程中一个重要的运行机制，在JAVASCRIPT同步和异步编程以及JAVASCRIPT内存管理中起着至关重要的作用。
  + 在JAVASCRIPT中，能形成作用域的有如下几点。
    - 函数的调用
    - with语句
      * with会创建自已的作用域，因此会增加其中执行代码的作用域的长度。
    - 全局作用域。
    - 以下代码为例：
    - var foo = function() {
    - var local = {};
    - };
    - foo();
    - console.log(local); //=undefined
    - var bar = function() {
    - local = {};
    - };
    - bar();
    - console.log(local); //={}
    - /\*\*这里我们定义了foo()函数和bar()函数，他们的意图都是为了定义一个名为local的变量。在foo()函数中，我们使用var语句来声明定义了一个local变量，而因为函数体内部会形成一个作用域，所以这个变量便被定义到该作用域中。而且foo()函数体内并没有做任何作用域延伸的处理，所以在该函数执行完毕后，这个local变量也随之被销毁。而在外层作用域中则无法访问到该变量。而在bar()函数内，local变量并没有使用var语句进行声明，取而代之的是直接把local作为全局变量来定义。故外层作用域可以访问到这个变量。\*\*/
* 作用域链
  + 在JAVASCRIPT编程中，会遇到多层函数嵌套的场景，这就是典型的作用域链的表示。
  + function foo() {
  + var val = 'hello';
  + function bar() {
  + function baz() {
  + global.val = 'world;'
  + };
  + baz();
  + console.log(val); //=hello
  + };
  + bar();
  + };
  + foo();
* 减少作用域链上的查找次数
  + JAVASCRIPT代码在执行的时候，如果需要访问一个变量或者一个函数的时候，它需要遍历当前执行环境的作用域链，而遍历是从这个作用域链的前端一级一级的向后遍历，直到全局执行环境。
* 闭包
  + JAVASCRIPT中的标识符查找遵循从内到外的原则。
  + function foo() {
  + var local = 'Hello';
  + return function() {
  + return local;
  + };
  + }
  + var bar = foo();
  + console.log(bar()); //=Hello

/\*\*这里所展示的让外层作用域访问内层作用域的技术便是闭包(Closure)。得益于高阶函数的应用，使foo()函数的作用域得到`延伸`。foo()函数返回了一个匿名函数，该函数存在于foo()函数的作用域内，所以可以访问到foo()函数作用域内的local变量，并保存其引用。而因这个函数直接返回了local变量，所以在外层作用域中便可直接执行bar()函数以获得local变量。\*\*/

* + - 闭包是JAVASCRIPT的高级特性，因为把带有​​内部变量引用的函数带出了函数外部，所以该作用域内的变量在函数执行完毕后的并不一定会被销毁，直到内部变量的引用被全部解除。所以闭包的应用很容易造成内存无法释放的情况。
    - 良好的闭包管理。
      * 循环事件绑定、私有属性、含参回调等一定要使用闭包时，并谨慎对待其中的细节。
        + 循环绑定事件，我们假设一个场景：有六个按钮，分别对应六种事件，当用户点击按钮时，在指定的地方输出相应的事件。
* 避开闭包陷阱
  + 闭包是个强大的工具，但同时也是性能问题的主要诱因之一。不合理的使用闭包会导致内存泄漏。
  + 闭包的性能不如使用内部方法，更不如重用外部方法。
    - 由于IE 9浏览器的DOM节点作为COM对象来实现，COM的内存管理是通过引用计数的方式，引用计数有个难题就是循环引用，一旦DOM引用了闭包(例如event handler)，闭包的上层元素又引用了这个DOM，就会造成循环引用从而导致内存泄漏。
* 善用函数
  + 使用一个匿名函数在代码的最外层进行包裹。
  + ;(function() {
  + // 主业务代码
  + })();

有的甚至更高级一点：

;(function(win, doc, $, undefined) {

// 主业务代码

})(window, document, jQuery);

甚至连如RequireJS, SeaJS, OzJS 等前端模块化加载解决方案，都是采用类似的形式：

/\*\*RequireJS\*\*/

define(['jquery'], function($) {

// 主业务代码

});

/\*\*SeaJS\*\*/

define('m​​odule', ['dep', 'underscore'], function($, \_) {

// 主业务代码

});

被定义在全局作用域的对象，可能是会一直存活到进程退出的，如果是一个很大的对象，那就麻烦了。比如有的人喜欢在JavaScript中做模版渲染：

<?php

$db = mysqli\_connect(server, user, password, 'myapp');

$topics = mysqli\_query($db, "SELECT \* FROM topics;");

?>

<!doctype html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>你是猴子请来的逗比么？</title>

</head>

<body>

<ul id="topics"></ul>

<script type="text/tmpl" id="topic-tmpl">

<li class="topic">

<h1><%=title%></h1>

<p><%=content%></p>

</li>

</script>

<script type="text/javascript">

var data = <?php echo json\_encode($topics); ?>;

var topicTmpl = document.querySelector('#topic-tmpl').innerHTML;

var render = function(tmlp, view) {

var complied = tmlp

.replace(/\n/g, '\\n')

.replace(/<%=([\s\S]+?)%>/g, function(match, code) {

return '" + escape(' + code + ') + "';

});

complied = [

'var res = "";',

'with (view || {}) {',

'res = "' + complied + '";',

'}',

'return res;'

].join('\n');

var fn = new Function('view', complied);

return fn(view);

};

var topics = document.querySelector('#topics');

function init()

data.forEach(function(topic) {

topics.innerHTML += render(topicTmpl, topic);

});

}

init();

</script>

</body>

</html>

在从数据库中获取到的数据的量是非常大的话，前端完成模板渲染以后，data变量便被闲置在一边。可因为这个变量是被定义在全局作用域中的，所以JAVASCRIPT引擎不会将其回收销毁。如此该变量就会一直存在于老生代堆内存中，直到页面被关闭。可是如果我们作出一些很简单的修改，在逻辑代码外包装一层函数，这样效果就大不同了。当UI渲染完成之后，代码对data的引用也就随之解除，而在最外层函数执行完毕时，JAVASCRIPT引擎就开始对其中的对象进行检查，data也就可以随之被回收。

### GITHUB:[前端性能优化指南](https://github.com/kahn1990/web_performance_optimization)

## 参考和借鉴了大家的经验，收集整理了这一篇开发规范，感谢所有的原作者，众人拾柴火焰高，技术无国界，持续更新中。

# 前端性能优化 - 资源预加载

当提到前端性能优化时，我们首先会联想到**文件的合并、压缩**，**文件缓存**和**开启服务器端的 gzip 压缩**等，这使得页面加载更快，用户可以尽快使用我们的 Web 应用来达到他们的目标。

资源预加载是另一个性能优化技术，我们可以使用该技术来**预先告知浏览器某些资源可能在将来会被使用到**。

引用 [Patrick Hamann](https://twitter.com/patrickhamann) 的[解释](http://patrickhamann.com/workshops/performance/tasks/2_Critical_Path/2_3.html)：

预加载是浏览器对将来可能被使用资源的一种暗示，一些资源可以在当前页面使用到，一些可能在将来的某些页面中被使用。作为开发人员，我们比浏览器更加了解我们的应用，所以我们可以对我们的核心资源使用该技术。

这种做法曾经被称为 prebrowsing，但这并不是一项单一的技术，可以细分为几个不同的技术：DNS-prefetch、subresource 和标准的 prefetch、preconnect、prerender。

## DNS 预解析 DNS-Prefetch

通过 DNS 预解析来告诉浏览器未来我们可能从某个特定的 URL 获取资源，当浏览器真正使用到该域中的某个资源时就可以尽快地完成 DNS 解析。例如，我们将来可能从 example.com 获取图片或音频资源，那么可以在文档顶部的 <head> 标签中加入以下内容：

|  |  |
| --- | --- |
| 1 | <link rel="dns-prefetch" href="//example.com"> |

当我们从该 URL 请求一个资源时，就不再需要等待 DNS 的解析过程。该技术对使用第三方资源特别有用。

在 Harry Roberts 的[文章](http://csswizardry.com/2013/01/front-end-performance-for-web-designers-and-front-end-developers/#section:dns-prefetching)中提到：

通过简单的一行代码就可以告知那些兼容的浏览器进行 DNS 预解析，这意味着当浏览器真正请求该域中的某个资源时，DNS 的解析就已经完成了。

这似乎是一个非常微小的性能优化，显得也并非那么重要，但事实并非如此 – [Chrome 一直都做了类似的优化](https://docs.google.com/presentation/d/18zlAdKAxnc51y_kj-6sWLmnjl6TLnaru_WH0LJTjP-o/present?slide=id.g120f70e9a_041)。当在浏览器的地址栏中输入 URL 的一小段时，Chrome 就自动完成了 DNS 预解析（甚至页面预渲染），从而为每个请求节省了至关重要的时间。

## 预连接 Preconnect

与 DNS 预解析类似，preconnect 不仅完成 DNS 预解析，同时还将进行 TCP 握手和建立传输层协议。可以这样使用：

|  |  |
| --- | --- |
| 1 | <link rel="preconnect" href="http://example.com"> |

在 Ilya Grigorik 的[文章](https://www.igvita.com/2015/08/17/eliminating-roundtrips-with-preconnect/)中有更详细的介绍：

现代浏览器都试着预测网站将来需要哪些连接，然后预先建立 socket 连接，从而消除昂贵的 DNS 查找、TCP 握手和 TLS 往返开销。然而，浏览器还不够聪明，并不能准确预测每个网站的所有预链接目标。好在，在 Firefox 39 和 Chrome 46 中我们可以使用 preconnect 告诉浏览器我们需要进行哪些预连接。

## 预获取 Prefetching

如果我们确定某个资源将来一定会被使用到，我们可以让浏览器预先请求该资源并放入浏览器缓存中。例如，一个图片和脚本或任何可以被浏览器缓存的资源：

|  |  |
| --- | --- |
| 1 | <link rel="prefetch" href="image.png"> |

与 DNS 预解析不同，预获取真正请求并下载了资源，并储存在缓存中。但预获取还依赖于一些条件，某些预获取可能会被浏览器忽略，例如从一个非常缓慢的网络中获取一个庞大的字体文件。并且，Firefox 只会在[浏览器闲置](https://developer.mozilla.org/en-US/docs/Web/HTTP/Link_prefetching_FAQ)时进行资源预获取。

在 Bram Stein 的[帖子](http://www.bramstein.com/writing/preload-hints-for-web-fonts.html)中说到，这对 webfonts 性能提升非常明显。目前，字体文件必须等到 DOM 和 CSS 构建完成之后才开始下载，使用预获取就可以轻松绕过该瓶颈。

**注意：**要测试资源的预获取有点困难，但在 Chrome 和 Firefox 的网络面板中都有资源预获取的记录。还需要记住，预获取的资源没有同源策略的限制。

## Subresources

这是另一个预获取方式，这种方式指定的预获取资源具有最高的优先级，在所有 prefetch 项之前进行：

|  |  |
| --- | --- |
| 1 | <link rel="subresource" href="styles.css"> |

根据 [Chrome 文档](https://www.chromium.org/spdy/link-headers-and-server-hint/link-rel-subresource)：

rel=prefetch 为将来的页面提供了一种低优先级的资源预加载方式，而 rel=subresource 为当前页面提供了一种高优先级的资源预加载。

所以，如果资源是当前页面必须的，或者资源需要尽快可用，那么最好使用 subresource 而不是 prefetch。

## 预渲染 Prerender

这是一个核武器，因为 prerender 可以预先加载文档的所有资源：

|  |  |
| --- | --- |
| 1 | <link rel="prerender" href="http://example.com"> |

Steve Souders 在他的[一篇文章](http://www.stevesouders.com/blog/2013/11/07/prebrowsing/)中写到：

这类似于在一个隐藏的 tab 页中打开了某个链接 – 将下载所有资源、创建 DOM 结构、完成页面布局、应用 CSS 样式和执行 JavaScript 脚本等。当用户真正访问该链接时，隐藏的页面就切换为可见，使页面看起来就是瞬间加载完成一样。Google 搜索在其即时搜索页面中已经应用该技术多年了，微软也宣称将在 IE11 中支持该特性。

需要注意的是不要滥用该特性，当你知道用户一定会点击某个链接时才可以进行预渲染，否则浏览器将无条件地下载所有预渲染需要的资源。

更多相关讨论：

所有预加载技术都存在一个潜在的风险：对资源预测错误，而预加载的开销（抢占 CPU 资源，消耗电池，浪费带宽等）是高昂的，所以必须谨慎行事。虽然很难确定用户下一步将访问哪些资源，但高可信的场景确实存在：

* 如果用户完成一个带有明显结果的搜索，那么结果页面很可能会被加载
* 如果用户进入到登陆页面，那么登陆成功的页面很可能会被加载
* 如果用户阅读一个多页的文章或访问一个分页的结果集，那么下一页很可能会被加载

最后，使用 [Page Visibility API](http://www.w3.org/TR/page-visibility/) 可以防止页面真正可见前被执行。

## Preload

preload 是一个新规范，与 prefetch 不同（可能被忽略）的是，浏览器一定会预加载该资源：

|  |  |
| --- | --- |
| 1 | <link rel="preload" href="image.png">☄ |

虽然该规范还没有被所有浏览器兼容，但其背后的思想还是非常有意思的。

## 总结

预测用户下一步将访问哪些资源是困难的，需要进行大量的测试，但是这带来的性能提升是明显的。如果我们愿意尝试这些预获取技术，一定会显著提升用户的体验。

# 网站性能优化·前端篇

我们的目标是优先显示与用户要在网页上执行的主要操作有关的内容。

上面这句话出自《[关键呈现路径](https://developers.google.com/web/fundamentals/performance/critical-rendering-path/)》。

## 什么是前端性能？

前端性能，即页面性能，简单来说就是**能看到页面内容的时间**以及**可以开始在页面上操作的时间**。

前端性能与浏览器的运作方式密切相关，故想要优化性能就要从了解浏览器是如何显示页面的入手。

### 浏览器显示页面的原理

基本流程：

1. 获取 HTML 文档及样式表文件
2. 解析成对应的树形数据结构  
     
   * DOM tree
   * CSSOM tree
3. 计算可见节点形成 render tree
4. 计算 DOM 的形状及位置进行布局
5. 将每个节点转化为实际像素绘制到视口上（栅格化）

render tree（页面上所显示的最终结果）是由 DOM tree（开发工具中所显示的 HTML 所定义的内容结构）与 CSSOM tree（样式表所定义的规则结构）合并并剔除不可见的节点所形成的，其中不包含如下节点：

* 本身不可见的  
    
  + <html>
  + <head>
  + <meta>
  + <link>
  + <style>
  + <script>
* 设置了 display: none; 样式的

资源加载：

* 执行 JavaScript 会阻止 DOM tree 构建
* 加载 CSS 会阻止 render tree 构建  
    
  + 无论是否为阻止呈现的 CSS，都会被浏览器下载

默认情况下，JavaScript 脚本会在引入它的位置执行（如果是外联脚本则还需要等待加载完毕），这时会阻断 DOM tree 的构建；如果在运行脚本时浏览器尚未完成 CSS 的下载和 CSSOM tree 的构建，浏览器会将脚本执行延迟到这些操作结束之后。

影响性能的因素：

* 白屏  
    
  + HTML 和 CSS 的加载及解析速度
  + <head> 内的脚本加载及执行
* 首屏  
    
  + 图片加载
  + <body> 内的脚本加载及执行
* render tree 的构建  
    
  + HTML 的复杂度
  + CSS 的复杂度
* render tree 的绘制（栅格化）  
    
  + 颜色的复杂度
  + 形状的复杂度

## 怎么提高前端性能？

提高以下几个方面，总体性能就会得到大幅度提升：

* 缩短白屏时间；
* 加快首屏显示；
* 尽快监听主要操作的事件。

所要达到的理想指标：

* 60 f/s

### [优化关键呈现路径](https://developers.google.com/web/fundamentals/performance/critical-rendering-path/optimizing-critical-rendering-path)

为了在首次渲染时尽可能快，我们需要优化以下三个变量：

* 最小化关键资源数
* 最小化关键字节数
* 最小化关键路径长度

常规步骤：

1. 分析并描述关键路径：资源数、字节数和长度；
2. 减少关键资源的数量：删掉、延迟下载或标记为异步等等；
3. 优化剩余关键资源的加载顺序：尽早下载所有关键资源以缩短关键路径长度；
4. 优化关键字节数以减少下载时间（往返次数）。

### 搜集性能数据

通过 [Navigation Timing](http://www.w3.org/TR/navigation-timing-2/#processing-model) API 可以获取浏览器在处理网页的关键步骤的时间戳。

其中，各步骤的意义如下：

1. **domLoading** 表示开始解析第一批收到的 HTML 文档的字节
2. **domInteractive** 表示完成全部 HTML 的解析并且 DOM 构建完毕
3. **domContentLoaded** 表示 [DOM 与 CSSOM 皆已准备就绪](http://calendar.perfplanet.com/2012/deciphering-the-critical-rendering-path/)  
     
   * 如果没有解析器阻塞 JavaScript，DOMContentLoaded 事件会在 domInteractive 之后立即触发
   * 很多 JavaScript 框架会在执行它们自己的逻辑前等待这个事件的触发
4. **domComplete** 表示所有的处理都已完成并且所有的附属资源都已经下载完毕
5. **loadEvent** 作为网页加载的最后一步以便触发附加的应用逻辑

### [PageSpeed 规则和建议](https://developers.google.com/web/fundamentals/performance/critical-rendering-path/page-speed-rules-and-recommendations/)

* 排除阻止呈现的 JavaScript 和 CSS
* 优化 JavaScript 的用法  
    
  + 推荐使用异步 JavaScript 资源
  + Avoid synchronous server calls
  + 延迟解析 JavaScript
  + 避免运行时间长的 JavaScript
* 优化 CSS 的用法  
    
  + 将 CSS 放到文档头部
  + 避免使用 CSS import
  + 内联阻止呈现的 CSS
* [2015年03月10日发布](https://segmentfault.com/a/1190000002589116)

# Java Web 前端高性能优化（一）

**Web 发展的速度让许多人叹为观止，层出不穷的组件、技术，只需要合理的组合、恰当的设置，就可以让 Web 程序性能不断飞跃。所有 Web 的思想都是通用的，它们也可以运用到 Java Web。这一系列的文章，主要讲解网页**[**前端性能优化**](http://www.oneapm.com/bi/feature.html)**，是与用户最直接接触的。事实证明，与其消耗大量时间在服务器端，在前端进行的优化更易获得用户的肯定。**

###### 一.引言

前端的高性能部分，主要是指减少请求数、减少传输的数据以及提高用户体验，在这个部分，图片的优化显得至关重要。许多网站的美化，都是靠绚丽的图片达到的，图片恰恰是占用带宽的元凶。每个 img 标签，浏览器都会试图发起一个下载请求。本文就详细介绍了图片优化的几种方式，介绍了使用的工具以及优化后的结果。

###### 二.图片压缩

减少图片的大小，可以明显的提高性能，而对于已有图片，要想减少图片的大小，只能改变图片的格式，这里推荐的是 PNG8 的格式，它可以在基本保持清晰度的情况下，减少图片的大小。知道这个原理以后，可以用 Windows 的画图工具、以及 PhotoShop 工具逐个的改变。但是这样做的缺点是单张处理，效率太慢。本文推荐一个在线转换工具 Smush.it，可以批量的进行压缩与转换。它的地址是：www.smushit.com/ysmush.it 打开后效果如下图所示。

**图 1. Yahoo 提供的在线压缩工具**



我们上传了一张大小为 3790K 的图片，待在线程序处理完毕后，点击 Download Smushed Images 下载查看结果。下载界面如下图所示。

**图 2. 压缩后的结果**



打开下载下来的压缩包，查看结果可以看到，图片从 3790 减少到了 3344，就如下图所示。对于大批量的图片网站，这个方法会帮助快速实现批量图片压缩。

**图 3. 压缩后的结果**

http://news.oneapm.com/content/images/2015/12/12.jpg

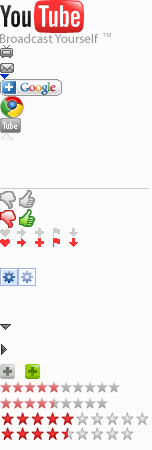
###### 三.图像合并实现 CSS Sprites

CSS Sprites 是一个吸引人的技术，它其实就是把网页中一些背景图片整合到一张图片文件中，再利用 CSS 的「background-image」，「background- repeat」，「background-position」的组合进行背景定位，background-position 可以用数字能精确的定位出背景图片的位置。

利用CSS Sprites 能很好地减少网页的 HTTP 请求，从而大大的提高了页面的性能，这也是 CSS Sprites 最大的优点，也是其被广泛传播和应用的主要原因。

CSS Sprites 能减少图片的字节，由于图像合并后基本信息不用重复，那么多张图片合并成 1 张图片的字节往往总是小于这些图片的字节总和。同时 CSS Sprites 解决了网页设计师在图片命名上的困扰，只需对一张集合的图片上命名就可以了，不需要对每一个小元素进行命名，从而提高了网页的制作效率。更换风格方便，只需要在一张或少张图片上修改图片的颜色或样式，整个网页的风格就可以改变。维护起来更加方便。同时，由于将图片合并到一张图片，因此图片的请求数就被缩减到 1 个。其他的请求都可以用到本地缓存，不需要访问服务器。下图是一个合并以后的图片。它将很多小图标都拼到了一起。

**图 4. 合并后的图片**

  
这里介绍一个小工具 ---「CSS Sprites 样式生成工具 2.0」，可以从这里下载。这是一个简单免费的小工具，用该工具打开上面的图片，选中图片中的某块。如下图的「绿色大拇指」部分，工具会计算出这个部分的长、宽、距离左上角的距离。勾选复制类名、复制宽、复制高，再点击「复制当前样式」按钮。这样生成的样式会被复制到剪切板上。

**图 5. 小工具的使用**



生成的 CSS 代码如清单 1 所示。

**清单 1. 小工具生成的 CSS 代码**

<pre>  
.div\_6148{width:18px;height:20px;background-position:-17px -209px;}  
</pre>

将这段代码运用在网页上，

**清单 2. 测试 CSS Sprites 代码**

<html>

<head>

<style>

.div\_6148

{

width:18px;

height:20px;

background-image:url(css-sprites-source.gif);

background-position:-17px -209px;

}

</style>

</head>

<body>

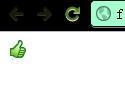
<div class="div\_6148"></div>

</body>

</html>

打开测试网页显示结果如下图所示。

**图 6. 测试网页效果**



可以看到，网页只显示工具选择的「绿色大拇指」部分，这样的代码可以运用在网页的多个部分，而图片只需要下载一次，这就是该技术的最大优势，减少了因为小图片引起的多个请求。

###### 四.多域名请求

有时候，图片数据太多，一些公司的解决方法是将图片数据分到多个域名的服务器上，这在一方面是将服务器的请求压力分到多个硬件服务器上。另一方面，是利用了浏览器的特性。一般来说，浏览器对于相同域名的图片，最多用 2-4 个线程并行下载。不同浏览器的并发下载数，都是不同的，并发数如下清单 3 所示。

**清单 3. 各浏览器的并发下载数**  
<pre>  
Browsers HTTP/1.1 HTTP/1.0   
IE6,7 2 4   
IE8 6 6   
FireFox 2 2 8   
FireFox 3 6 6   
Safari 3,4 4 4   
Chrome 1,2 6 ?   
Chrome 3 4 4   
Opera 9.63,10.00alpha 4 4  
</pre>

而相同域名的其他图片，则要等到其他图片下载完后才会开始下载。 这里我做了一个测试，选择了多个相同域名的图片在同一网页上。代码如清单 4 所示。

**清单 4. 单域名的多图片下载**

<html>

<body>

<img src="http://img1.gtimg.com/news/pics/hv1/123/231/804/52339128.jpg"><br>

<img src="http://img1.gtimg.com/news/pics/hv1/87/235/804/52340112.jpg"><br>

<img src="http://img1.gtimg.com/finance/pics/hv1/41/119/804/52310486.jpg"><br>

<img src="http://img1.gtimg.com/sports/pics/hv1/246/198/804/52330836.jpg"><br>

<img src="http://img1.gtimg.com/ent/pics/hv1/101/54/805/52358996.jpg"><br>

<img src="http://img1.gtimg.com/blog/pics/hv1/169/226/804/52337899.jpg">

</body>

</html>

接下来，使用 FireFox 的 Firebug 插件监控网络(Firebugde 使用请参考[Firebug,Debugger javascript 调试利器 附下载地址](http://club.oneapm.com/t/firebug-debugger-javascript/233))。结果如下图所示。

**图 7. 单域名多图片的监控效果**



可以看到，相同域名的多张图片，它们下载的起始点是存在延迟的。它们并不是并行下载。当我们将其中的 3 张图片换成别的域名图片。如清单 5 所示。

**清单 5. 多域名多图片下载**

<html>

<body>

<img src="http://img1.gtimg.com/news/pics/hv1/123/231/804/52339128.jpg"><br>

<img src="http://img1.gtimg.com/news/pics/hv1/87/235/804/52340112.jpg"><br>

<img src="http://img1.gtimg.com/finance/pics/hv1/41/119/804/52310486.jpg"><br>

<img src="http://i0.itc.cn/20110624/64a\_2ee7d710\_2ec6\_b38d\_b678\_dc3af28392be\_1.jpg"><br>

<img src="http://i0.itc.cn/20110624/3b0\_643eaea5\_1233\_b543\_82b7\_9c7273c7f97c\_1.jpg"><br>

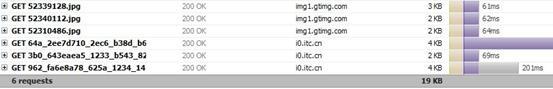
<img src="http://i0.itc.cn/20110623/962\_fa6e8a78\_625a\_1234\_147f\_3a627fe17033\_1.jpg">

</body>

</html>

再次查看网络监控，可以看到，这些图片是并行下载的。

**图 8. 多域名多图片测试结果**



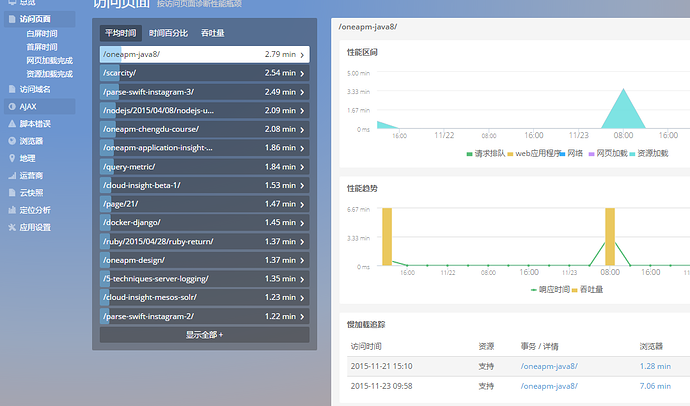
多域名的下载固然很好，但是太多域名并不太好，一般在 2-3 个域名下载就差不多。

###### 五.如何确定慢加载

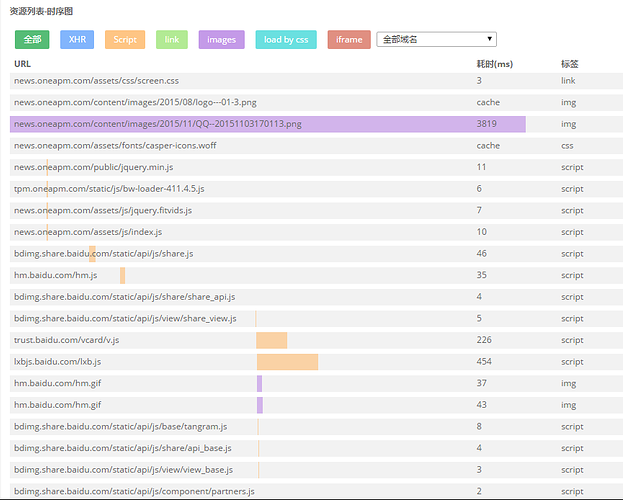
上文介绍的主要是如何对图片进行优化合并，但是我们不可能对每个页面的每个图片都进行上面的操作，所以，如何获知当前页面的慢加载图片则尤为重要。

Browser Insight 主要是OneAPM开发的一款针对于前端页面监控以及前端性能优化的工具，功能维度很丰富，本文主要介绍的是其慢加载的瀑布流图功能。

**图9.Browser Insight 页面响应时间**



**图10.资源列表-时序图**



参考上面两个图片，我们可以在 Browser Insight 的「访问页面」的「慢加载追踪」模块进入「资源列表时序图」页面，我们可以在这个页面及其详细的看到拖慢整个页面加载的图片、脚本、css 等，进而进行有针对性的优化  
（备注：本文转载自 [IBM 社区](http://www.ibm.com/developerworks/cn/java/j-lo-javawebhiperf1/#icomments)，由 OneAPM 产品运营编译整理。）

[**Browser Insight**](http://www.oneapm.com/bi/feature.html?utm_source=Community&utm_medium=Article&utm_term=201512javawebOne&utm_campaign=AiJavaArti&from=jscljdqcme)**是一个基于真实用户的Web前端性能监控平台，能够帮大家定位网站性能瓶颈，网站加速效果可视化；支持浏览器、微信、App浏览HTML和HTML5页面。想阅读更多技术文章，请访问**[**OneAPM 官方博客**](http://news.oneapm.com/?utm_source=Community&utm_medium=Article&utm_term=201512javawebOne&utm_campaign=AiJavaArti&from=jscljdqcme)**。**

* [2015年12月04日发布](https://segmentfault.com/a/1190000004096390)