如何加载以及执行 JavaScript 确保性能最优

随着用户体验的日益重视，前端性能对用户体验的影响备受关注，但由于引起性能问题的原因相对复杂，我们很难但从某一方面或某几个方面来全面解决它，接下来用一系列文章来深层次探讨与梳理有关 Javascript 性能的方方面面，以填补并夯实大家的知识结构。

不得不说的 JavaScript 的阻塞特性

前端开发者应该都知道，JavaScript 是单线程运行的，也就是说，在 JavaScript 运行一段代码块的时候，页面中其他的事情（UI 更新或者别的脚本加载执行等）在同一时间段内是被挂起的状态，不能被同时处理的，所以在执行一段 js 脚本的时候，这段代码会影响其他的操作。这是 JavaScript 本身的特性，我们无法改变。下面说一下页面渲染的过程：先下载解析 HTML 并建立 DOM 树，再解析 css 绘制渲染树。前者搭建页面结构，后者增添页面样式。而在建立 DOM 树的过程就会遇到诸如 img、外联 css 和 script 标签，此时就要加载外部资源了。**加载资源是由单独的下载线程进行异步加载的**，浏览器会并行加载，不过具体并行最大数量是有一定限制的，不同浏览器可能不一样。但是加载 css 和 js 资源比较特殊，它们的加载会影响页面渲染。css 加载不会阻塞 DOM 树解析，但会阻塞渲染（这是由于渲染依赖于 css，如果不等 css 加载完就渲染的话那么等 css 加载解析完又得重新渲染，可能又要重绘或者回流）。对于 js 资源的加载，则会阻塞 DOM 树的构建和渲染，除非设置了 script 标签的异步属性。放在 head 中会在解析 DOM 树和渲染页面前就加载，并阻塞页面。js 正常情况下加载完就会立即执行，在 js 脚本中只能访问当前<script>以上的 DOM，脚本执行结束后再继续解析 DOM。**js 执行引擎和页面渲染是由不同的线程来执行，但这两者是互斥的，也就是说 js 执行过程是无法构建 DOM 和渲染页面的**。这是一种优化机制，由于 js 可能会对 DOM 及样式进行修改，如果解析 js 过程中同时构建 DOM，就可能造成前后内容不一致或者重复构建。

我们把 JavaScript 的这一特性叫做**阻塞特性**，正因为这个阻塞特性，让前端的性能优化尤其是在对 JavaScript 的性能优化上变得相对复杂。

为什么要阻塞？

也许你还会问，既然 JavaScript 的阻塞特性会产生这么多的问题，为什么 JavaScript 语言不能像 Java 等语言一样，采用多线程，不就 OK 了么？

要彻底理解 JavaScript 的单线程设计，其实并不难，简单总结就是：最初设计 JavaScript 的目的只是用来在浏览器端改善网页的用户体验，去处理一些页面中类似表单验证的简单任务。所以，那个时候 JavaScript 所做的事情很少，并且代码不会太多，这也奠定了 JavaScript 和界面操作的强关联性。

既然 JavaScript 和界面操作强相关，我们不妨这样理解：试想，如果在某个页面中有两段 js 脚本都会去更改某一个 dom 元素的内容，如果 JavaScript 采用了多线程的处理方式，那么最终页面元素显示的内容到底是哪一段 js 脚本操作的结果就不确定了，因为两段 js 是通过不同线程加载的，我们无法预估谁先处理完，这是我们不想要的结果，而这种界面数据更新的操作在 JavaScript 中比比皆是。因此，我们就不难理解 JavaScript 单线程的设计原因：**JavaScript 采用单线程，是为了避免在执行过程中页面内容被不可预知的重复修改**。

从加载上优化：合理放置脚本位置

由于 JavaScript 的阻塞特性，在每一个<script>出现的时候，无论是内嵌还是外链的方式，它都会让页面等待脚本的加载解析和执行，并且<script>标签可以放在页面的<head>或者<body>中，因此，如果我们页面中的 css 和 js 的引用顺序或者位置不一样，即使是同样的代码，加载体验都是不一样的。举个栗子：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <title>js 引用的位置性能优化</title>
    <script type="text/javascript" src="index-1.js"></script>
    <script type="text/javascript" src="index-2.js"></script>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <div id="app"></div>
  </body>
</html>
```

以上代码是一个简单的 html 界面，其中加载了两个 js 脚本文件和一个 css 样式文件，由于 js 的阻塞问题，当加载到 index-1.js 的时候，其后面的内容将会被挂起等待，直到 index-1.js 加载、执行完毕，才会执行第二个脚本文件 index-2.js，这个时候页面又将被挂起等待脚本的加载和执行完成，一次类推，这样用户打开该界面的时候，界面内容会明显被延迟，我们就会看到一个空白的页面闪过，这种体验是明显不好的，因此**我们应该尽量的让内容和样式先展示出来，将 js 文件放在<body>最后，以此来优化用户体验**。

```
<!DOCTYPE html>
<html>
```

```
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <title>js 引用的位置性能优化</title>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <div id="app"></div>
    <script type="text/javascript" src="index-1.js"></script>
    <script type="text/javascript" src="index-2.js"></script>
  </body>
</html>
```

这段代码展示了在 HTML 文档中放置<script>标签的推荐位置。尽管脚本下载会阻塞另一个脚本，但是页面的大部分内容都已经下载完成并显示给了用户，因此页面下载不会显得太慢。这是雅虎特别性能小组提出的优化 JavaScript 的首要规则：将脚本放在底部。

从请求次数上优化： 减少请求次数

由于每个<script>标签初始下载时都会阻塞页面渲染，所以减少页面包含的<script>标签数量有助于改善这一情况。这不仅针对外链脚本，内嵌脚本的数量同样也要限制。浏览器在解析 HTML 页面的过程中每遇到一个<script>标签，都会因执行脚本而导致一定的延时，因此最小化延迟时间将会明显改善页面的总体性能。

这个问题在处理外链 JavaScript 文件时略有不同。考虑到 HTTP 请求会带来额外的性能开销，因此下载单个 100Kb 的文件将比下载 5 个 20Kb 的文件更快。也就是说，减少页面中外链脚本的数量将会改善性能。

通常一个大型网站或应用需要依赖数个 JavaScript 文件。您可以把多个文件合并成一个，这样只需要引用一个<script>标签，就可以减少性能消耗。文件合并的工作可通过离线的打包工具或者一些实时的在线服务来实现。

需要特别提醒的是，把一段内嵌脚本放在引用外链样式表的<link>之后会导致页面阻塞去等待样式表的下载。这样做是为了确保内嵌脚本在执行时能获得最精确的样式信息。因此，建议不要把内嵌脚本紧跟在<link>标签后面。

有一点我们需要知道：页面加载的过程中，最耗时间的不是 js 本身的加载和执行，相比之下，每一次去后端获取资源，客户端与后台建立链接才是最耗时的，也就是大名鼎鼎的 Http 三次握手，当然，http 请求不是我们这一次讨论的主题，

因此，减少 HTTP 请求，是我们着重优化的一项，事实上，在页面中 js 脚本文件加载很很多情况下，它的优化效果是很显著的。要减少 HTTP 的请求，就不得不提起文件的精简压缩了。

文件的精简与压缩

要减少访问请求，则必然会用到 js 的**精简(minifucation)和压缩(compression)**了，需要注意的是，精简文件实际并不复杂，但不适当的使用也会导致错误或者代码无效的问题，因此在实际的使用中，最好在压缩之前对 js 进行语法解析，帮我们避免不必要的问题（例如文件中包含中文等 unicode 转码问题）。

解析型的压缩工具常用有三：YUI Compressor、Closure Complier、UglifyJs

YUI Compressor: YUI Compressor 的出现曾被认为是最受欢迎的基于解析器的压缩工具，它将去去除代码中的注释和额外的空格并且会用单个或者两个字符去代替局部变量以节省更多的字节。但默认会关闭对可能导致错误的替换，例如 with 或者 eval();

Closure Complier: Closure Complier 同样是一个基于解析器的压缩工具，他会试图去让你的代码变得尽可能小。它会去除注释和额外的空格并进行变量替换，而且会分析你的代码进行相应的优化，比如他会删除你定义了但未使用的变量，也会把只使用了一次的变量变成内联函数。

UglifyJs：UglifyJs 被认为第一个基于 node.js 的压缩工具，它会去除注释和额外的空格，替换变量名，合并 var 表达式，也会进行一些其他方式的优化

每种工具都有自己的优势，比如说 YUI 压缩后的代码准确无误，Closure 压缩的代码会更小，而 UglifyJs 不依靠于 Java 而是基于 JavaScript，相比 Closure 错误更少，具体用哪个更好我觉得没有个确切的答案，开发者应该根据自己项目实际情况酌情选择。

从加载方式上优化：无阻塞脚本加载

在 JavaScript 性能优化上，减少脚本文件大小并限制 HTTP 请求的次数仅仅是让界面响应迅速的第一步，现在的 web 应用功能丰富，js 脚本越来越多，光靠精简源码大小和减少次数不总是可行的，即使是一次 HTTP 请求，但文件过于庞大，界面也会被锁死很长一段时间，这明显不好的，因此，无阻塞加载技术应运而生。

简单来说，**就是页面在加载完成后才加载 js 代码，也就是在 window 对象的 load 事件触发后才去下载脚本**。 要实现这种方式，常用以下几种方式：

延迟脚本加载（defer）

HTML4 为<script>标签定义了一个扩展属性：defer。Defer 属性指明本元素所含的脚本不会修改 DOM，因此代码能安全地延迟执行。defer 属性只被 IE 4 和 Firefox 3.5 更高版本的浏览器所支持，所以它不是一个理想的跨浏览器解决方案。在其他浏览器中，defer 属性会被直接忽略，因此<script>标签会以默认的方式处理，也就是说会造成阻塞。然而，如果您的目标浏览器支持的话，这仍然是个有用的解决方案。

```
<script type="text/javascript" src="index-1.js" defer></script>
```

带有 defer 属性的<script>标签可以放置在文档的任何位置。对应的 JavaScript 文件将在页面解析到<script>标签时开始下载，但不会执行，直到 DOM 加载完成，即 onload 事件触发前才会被执行。当一个带有 defer 属性的 JavaScript 文件下载时，它不会阻塞浏览器的其他进程，因此这类文件可以与其他资源文件一起并行下载。•

任何带有 defer 属性的<script>元素在 DOM 完成加载之前都不会被执行，无论内嵌或者是外链脚本都是如此。

延迟脚本加载（async）

HTML5 规范中也引入了 async 属性，用于异步加载脚本，其大致作用和 defer 是一样的，都是采用的并行下载，下载过程中不会有阻塞，但**不同点在于他们的执行时机，async 需要加载完成后就会自动执行代码，但是 defer 需要等待页面加载完成后才会执行**。

从加载方式上优化：动态添加脚本元素

把代码以动态的方式添加的好处是：无论这段脚本是在何时启动下载，它的下载和执行过程都不会则色页面的其他进程，我们甚至可以直接添加带头部 head 标签中，都不会影响其他部分。

因此，作为开发的你肯定见到过诸如此类的代码块：

```
var script = document.createElement('script');
script.type = 'text/javascript';
script.src = 'file.js';
document.getElementsByTagName('head')[0].appendChild(script);
```

这种方式便是动态创建脚本的方式，也就是我们现在所说的动态脚本创建。通过这种方式下载文件后，代码就会自动执行。但是在现代浏览器中，这段脚本会等待所有动态节点加载完成后再执行。这种情况下，为了确保当前代码中包含的别的代码的接口或者方法能够被成功调用，就必须在别的代码加载前完成这段代码的准备。解决的具体操作思路是：

现代浏览器会在 script 标签内容下载完成后接收一个 load 事件，我们就可以在 load 事件后再去执行我们想要执行的代码加载和运行，在 IE 中，它会接收 loaded 和 complete

事件，理论上是 loaded 完成后才会有 completed，但实践告诉我们他两似乎并没有个先后，甚至有时候只会拿到其中的一个事件，我们可以单独的封装一个专门的函数来体现这个功能的实践性,因此一个统一的写法是：

```
 function LoadScript(url, callback) {
        var script = document.createElement('script');
        script.type = 'text/javascript';
        // IE 浏览器下
        if (script.readyState) {
          script.onreadystatechange = function () {
            if (script.readyState == 'loaded' || script.readyState ==
'complete') {
              // 确保执行两次
              script.onreadystatechange = null;
              // todo 执行要执行的代码
              callback()
            }
          }
        } else {
          script.onload = function () {
            callback();
          }
        }
        script.src = 'file.js';
        document.getElementsByTagName('head')[0].appendChild(script);
      }
```

LoadScript 函数接收两个参数，分别是要加载的脚本路径和加载成功后需要执行的回调函数，LoadScript 函数本身具有特征检测功能，根据检测结果（IE 和其他浏览器），来决定脚本处理过程中监听哪一个事件。

> 实际上这里的 LoadScript()函数，就是我们所说的 LazyLoad.js（懒加载）的原型。

有了这个方法，我们可以实现一个简单的多文件按某一固定顺序加载代码块：

```
LoadScript('file-1.js', function(){
  LoadScript('file-2.js', function(){
    LoadScript('file-3.js', function(){
        console.log('loaded all')
    })
  })
```

```
})
```

以上代码执行的时候，将会首先加载 file-1.js,加载完成后再去加载 file-2.js,以此类推。当然这种写法肯定是有待商榷的（多重回调嵌套写法简直就是地狱），但这种动态脚本添加的思想，和加载过程中需要注意的和避免的问题，都在 LoadScript 函数中得以澄清解决。

当然，如果文件过多，并且加载的顺序有要求，最好的解决方法还是建议按照正确的顺序合并一起加载，这从各方面讲都是更好的法子。

从加载方式上优化：XMLHttpRequest 脚本注入

通过 XMLHttpRequest 对象来获取脚本并注入到页面也是实现无阻塞加载的另一种方式，这个我觉得不难理解，这其实和动态添加脚本的方式是一样的思想，来看具体代码：

```
var xhr = new XMLHttpRequest();
xhr.open('get', 'file-1.js', true);
xhr.onreadystatechange = function() {
  if(xhr.readyState === 4) {
    if(xhr.status >= 200 && xhr.status < 300 || xhr.status === 304){
      // 如果从后台或者缓存中拿到数据，则添加到 script 中并加载执行。
      var script = document.createElement('script');
      script.type = 'text/javascript';
      script.text = xhr.responseText;
      // 将创建的 script 添加到文档页面
      document.body.appendChild(script);
    }
  }
}
```

通过这种方式拿到的数据有两个优点：其一，我们可以控制脚本是否要立即执行，因为我们知道新创建的 script 标签只要添加到文档界面中它就会立即执行，因此，在添加到文档界面之前，也就是在 appendChild()之前，我们可以根据自己实际的业务逻辑去实现需求，到了想要让它执行的时候，再 appendChild()即可。其二：它的兼容性很好，所有主流浏览器都支持，它不需要想动态添加脚本的方式那样，我们自己去写特性检测代码；

但由于是使用了 XHR 对象，所以不足之处是获取这种资源有“域”的限制。资源 必须在同一个域下才可以，不可以跨域操作。

总结

减少 JavaScript 对性能的影响有以下几种方法：

- 将所有的<script>标签放到页面底部，也就是</body>闭合标签之前，这能确保在脚本执行前页面已经完成了渲染。
- 尽可能地合并脚本。页面中的<script>标签越少，加载也就越快，响应也越迅速。无论是外链脚本还是内嵌脚本都是如此。
- 采用无阻塞下载 JavaScript 脚本的方法：
    - 使用<script>标签的 defer 属性（仅适用于 IE 和 Firefox 3.5 以上版本）；
    - 使用动态创建的<script>元素来下载并执行代码；
    - 使用 XHR 对象下载 JavaScript 代码并注入页面中。

通过以上策略，可以在很大程度上提高那些需要使用大量 JavaScript 的 Web 网站和应用的实际性能。