前端性能优化最佳实践

 世勋 sehu... 前端开发 6 天前 1154 浏览

来源：http://www.imooc.com/article/1647

如今浏览器能够实现的特性越来越多，并且网络逐渐向移动设备转移，使我们的前端代码更加紧凑，如何优化，就变得越来越重要了。

开发人员普遍会将他们的代码习惯优先于用户体验。但是很多很小的改变可以让用户体验有个飞跃提升，所以任何一点儿小小的优化都会提升你网站的性能。

前端给力的地方是可以有许多种简单的策略和代码习惯让我们可以保证最理想的前端性能。我们这个系列的主题就是要告诉你一些前端性能优化的最佳实践，只需要一分钟，就可以优化你现有的代码。

最佳实践 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<ul>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);
});
```

上面的代码其实是一个错误的写法，将<ul>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<ul>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。

最佳实践 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。

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

```
// 取自 UnderscoreJS 实用框架
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！

最佳实践 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。

最佳实践 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, "", h
ref || 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'], functio
n() {
    Prism.highlightAll();
});

// 加载一个 AJAX 请求的 URL
curl(['text!sidebar.php', 'storage', 'dom'], function(content, storag
e, 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。

最佳实践 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。

最佳实践 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。

最佳实践 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<li>元素，每一个<li>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
document.querySelector('#parent-list').addEventListener('click', fun
ction(e) {
    // e.target 是一个被点击的元素!
    // 如果它是一个列表元素
    if(e.target && e.target.tagName == 'LI') {
        // 我们找到了这个元素，对他的操作可以写在这里。
    }
});
```

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！

最佳实践 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/7AARRH
Vja3kAAQAEAAAAPAAA/+4ADkFkb2JlAGTAAAAAAf/bAIQABgQEBAUEBgUFBgkGBQYJCw
gGBggLDAoKCwoKDBAMDAwMDAwQDA4PEA8ODBMTFBQTExwbGxscHx8fHx8fHx8fHwEHBw
cNDA0YEBAYGhURFRofHx8fHx8fHx8fHx8fHx8fH ...." />

<-- 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。

最佳实践 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）。

最佳实践 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。

最佳实践 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。

最佳实践 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: convertImageToDat
aUri(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。

最佳实践 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。

最佳实践 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。
作者：Nelly
网址：http://www.gbtags.com/gb/tag/usertag/5929.htm