目录

15 浏览器缓存（下）

更新时间：2019-08-26 10:07:46

> 书籍乃世人积累智慧之长明灯。
>
> ——寇第斯

缓存字段的配置

上一节我们介绍了强制缓存与协商缓存，那么这一节我们就来介绍如何配置这些字段。我们会分别介绍当前使用最为广泛的Apache和Nginx如何配置缓存字段。关于Apache和Nginx的安装我们就不详细介绍了，大家可以使用搜索引擎进行相关查询。

- **expires和cache-control**

  如果使用的是Nginx，那么打开Nginx的配置文件nginx.conf，具体配置方法如下：

```
// expires:给图片设置过期时间30天，这里也可以设置其它类型文件
location ~ \.(gif|jpg|jpeg|png)$ {
        root /var/www/img/;
        expires 30d;
}
// cache-control:给图片设置过期时间36秒，这里也可以设置其它类型文件
location ~ \.(gif|jpg|jpeg|png)$ {
        root /var/www/img/;
        add_header    Cache-Control  max-age=3600;
}
```

  如果使用的是Apache，那么打开Apache的配置文件http.conf，具体的配置方法如下：

```
  // expires
  <IfModule expires_module>
        #打开缓存
        ExpiresActive on
        # 给图片设置过期时间30天，这里也可以设置其它类型文件
```

```
      ExpiresByType image/png access plus 30 days
</IfModule>
// cache-control
<FilesMatch "\.(gif/jpg/jpeg/png)$">
       Header set Cache-Control "max-age=604800, public"
      </FilesMatch>
```

- **Last-Modified和Etag**

  Last-Modified在Nginx和Apache当中都是默认启用的，所以无需我们手动进行相关配置。这里我们直接介绍如何配置Etag，Etag的配置不管是在Nginx还是Apache当中都非常简单。

  如果我们使用的是Apache，那么只需要在.htaccess文件当中添加如下配置，即可开启Etag：

```
FileETag MTime Size
```

  如果我们使用的是Nginx，需要先安装Etag模块；安装成功之后，仍然打开nginx.conf文件，确保当中没有出现etgoff，然后添加如下配置，即可开启Etag：

```
location ~ .*\.(gif|jpg|jpeg|png)$ {
   FileETag on;
   etag_format "%X%X%X"; //这里格式化规则可以修改
 }
```

Webpack打包时如何应对缓存

首先来想这样一个问题，如果我们的静态资源在缓存期内被修改，那么我们该如何通知浏览器从服务端拉取最新的资源呢？在没有Webpack之前，我们一般的处理方法就是对CSS、JavaScript、图片这些静态资源设置为强制缓存，而入口文件(index.html)使用协商缓存或者干脆强制不缓存。这样可以通过修改入口文件(index.html)中对强制缓存静态资源的引入 URL 来达到即时更新的目的。

那么后来Webpack出现之后，我们有了新的解决方案，那么目前采用的主流方案是Webpack增量更新，增量更新的含义就是只更新发生了改动的静态文件，对于没有发生改动的静态文件，则继续使用缓存好的，无需进行修改。Webpack增量更新使用文件的hash指纹来确定一个文件是否进行了更新。hash指纹其实和我们前面介绍的Etag是非常类似的，只要文件修改，hash指纹就会发生修改。

Webpack一共有三种hash，分别是hash、chunkhash、contenthash。

- hash(the hash of the module identifier)是跟整个项目的构建相关，默认长度为20，我们可以根据实际情况设置合适长度。hash根据入口文件打包出的文件都使用相同的hash。如果当中有引用图片，那么不同的图片有不同的hash值，不会和前面打包出的文件使用相同的hash。因此，一般的图片和字体文件都是用hash来固定其缓存。
- chunkhash(the hash of the chunk content)根据不同的入口文件进行依赖文件分析然后生成对应的chunkhash。chunkhash对应的是每个文件，所以每个入口文件打包后的chunkhash都是不同的，所以我们的JavaScript文件用chunkhash来固定缓存。

JavaScript文件是使用的是相同的chunkhash。这个时候，如果我们只是修改了CSS，并没有修改JavaScript，你会发现JavaScript的chunkhash还是跟着CSS的一起变化，这显然不是我们需要的结果。所以针对这个问题，单独抽取的CSS文件我们都使用contenthash来固定。

下面我们根据上面的描述来简单配置一下Webpack，具体配置信息如下：

```
module.exports = {
    entry: {
        index: './test/test.js',
        about:  './test/about.js'
    },
    output: {
                // 打包出来的JavaScript文件用chunkhash
        filename: '[name].[chunkhash:8].js',
        path: path.resolve(__dirname, 'dist')
    },
        module: {
        rules: [
            {
                test: /\.css$/,
                use: [{loader: MiniCssExtractPlugin.loader},'css-loader']
            },
            {
                 test: /\.(png|jpe?g|gif|svg)(\?.*)?$/,
                                        loader: 'file-loader',
                 query: {
                     // 图片文件使用hash
                     name: '[name].[ext]?[hash]',
                     outputPath: 'static/img/',
                     publicPath: '/dist/static/img/'
                 }
            }
        ]
    },
   plugins: [
        new CleanWebpackPlugin(),
                new MiniCssExtractPlugin({
                    // 抽离的CSS文件用contenthash
            filename: '[name].[contenthash:8].css',
                  chunkFilename: '[id].css'
        }),
    ]
}
```

上图是我最近在做的项目。可以看到，使用我们上面介绍的方法，可以起到很好的固定缓存的作用。

首先这部分属于扩展内容，根据自己的精力自行选择是否学习即可。前面我介绍的是缓存的相关字段，那么缓存下来的东西到底存放到了哪里呢？下面我们就介绍根据缓存位置的不同进行的分类：

- **Service Worker**

Service Worker是运行在浏览器背后的独立线程。可能大家在看到Service Worker这个概念的时候发现它经常和PWA一起出现。可以这么说，PWA应用在开发的时候必会用到Service Worker，这得益于Service强大的缓存能力。使用Service Worker前提必须是HTTPS协议，因为Service Worker涉及到了HTTP的拦截，所以必须使用HTTPS协议来保证安全。要使用Service Worker，首先来引入Service Worker，如下：

```
if ('serviceWorker' in navigator) {
    //scope参数是可选的，用来指定Service Worker控制内容的目录
  navigator.serviceWorker.register('/test/sw.js', { scope: '/test/' }).then(
    // 注册成功
    console.log('Registration succeeded. Scope is ' + reg.scope);
  }).catch(function(error) {
    // 注册失败
    console.log('Registration failed with ' + error);
  });
}
```

注册完Service Worker之后，浏览器会激活Service Worker，到此我们就可以使用Service Worker来缓存相关内容了。

```
self.addEventListener('install', event => {
  event.waitUntil(
    caches.open('test').then(cache => {
      return cache.addAll([
        '/sw.html',
        '/sw.css',
        '/sw.js'
        '/sw.png'
      ])
    })
  )
})
```

上面我们创建了test缓存，然后caches.open方法返回一个promise。如果添加成功，我们就在test下缓存sw.html、sw.css、sw.js、sw.png这些列表信息。

除了可以缓存内容，Service Worker还可以拦截各种请求，返回我们想让用户看到的内容：

```
self.addEventListener('fetch', function (event) {
    if (/\.jpg|.png|.gif$/.test(event.request.url)) {
        event.respondWith(fetch('/images/default.png'));
    }
});
```

Service Worker的更多用法推荐大家阅读MDN。

> Tips：Service Worker本身是非常强大的，但是目前在项目中实践还有待商榷，国内这方面的实践还比较少，不过作为扩展知识了解还是非常有必要的。

- **Memory Cache**

  Memory Cache指得是内存中的缓存，大家都知道内存的读取是非常快的，但是空间确有限，而且持续性很短，一旦我们关闭当前tab页面，Memory Cache也就随着消失了。Memory Cache缓存的大部分是preloader指令下的静态资源。

- **Disk Cache**

  Disk Cache是存储在硬盘上的缓存，我们为静态资源设置的缓存一般都是缓存到Disk Cache当中，所以我们平时接触到最多的缓存就是Disk Cache。

- **Push Cache**

  Push Cache是推送缓存，它是HTTP2.0新增加的内容，目前实际的应用还偏少。一个原因是它是比较新的技术，另一个原因是它优先级最低，所以相关实践比较少，所以这部分我们也简单介绍一下即可，如果你感兴趣，可以阅读这篇文章。

小结

到此我们用两个小节把浏览器缓存的所有内容都介绍完毕。那么当有面试官再次问到你关于浏览器缓存方面的知识，相信你可以很好地作答相关问题。缓存一直以来都是非常重要的内容，所以这2小节的内容希望大家不仅要记住，还要下去多实践。

← 14 浏览器缓存（上） 16 渲染原理与性能优化 →

精选留言 1

欢迎在这里发表留言，作者筛选后可公开显示

qq_笑对人生

disk缓存，默认不应该是在缓存到内存，etag才是缓存在disk吧?

0 回复 2019-12-16

千学不如一看，千看不如一练