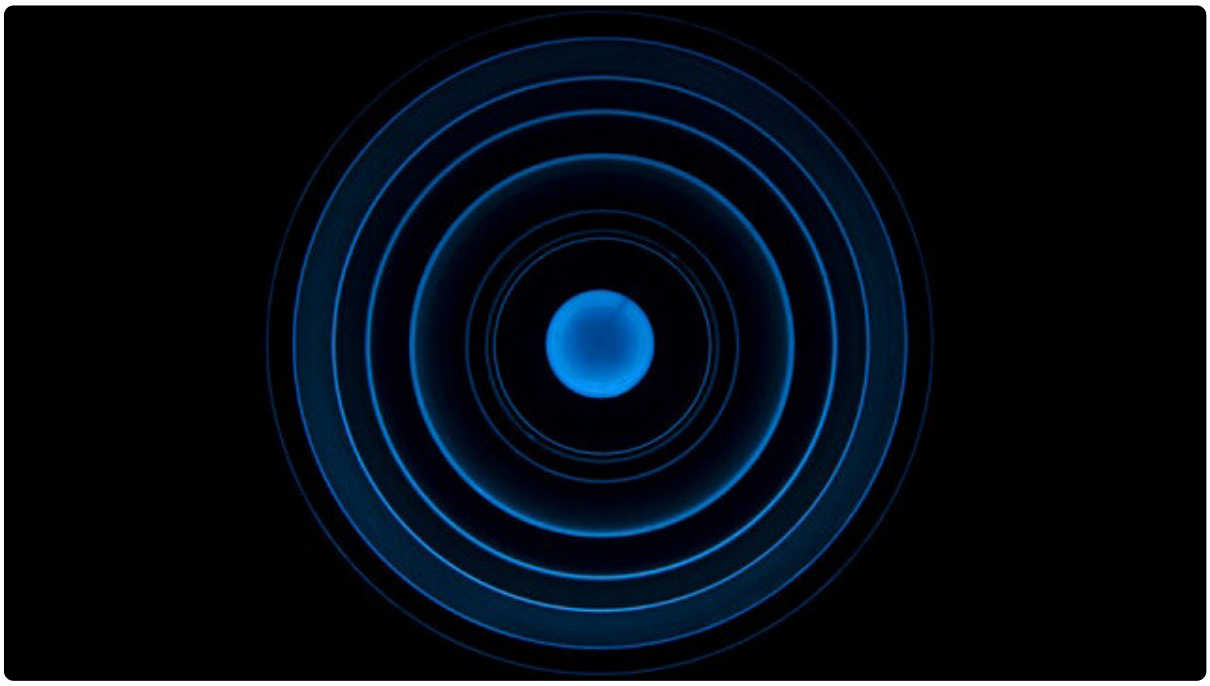
18 Webpack 优化之增强缓存命中率

更新时间：2019-06-24 09:31:21



> 勤学如春起之苗，不见其增，日有所长。
>
> ——陶潜

在 web 开发中，我们应该充分利用 HTTP 协议和浏览器的缓存来做好页面代码的持久化缓存，本文将介绍如何合理配置 Webpack 来更好的提升应用持久化缓存策略，通过将资源缓存在客户端中，可以避免之后每次都重新下载。

浏览器缓存策略和 Webpack 缓存相关配置

使用浏览器的持久化缓存方案分两步走，**第一步**是我们一般会将静态资源（JavaScript 、CSS、图片字体文件等）这些不经常变动的文件寻找更合适的服务器存放，比如放到 CDN 服务器上，并且配置单独的域名，比如百度常用的 CDN 域名是：`x.bdimg.com`和`x.bdstatic.com`域名，这样做的好处是：

1. 保证动静分离，将动态页面和静态资源分开部署有利于服务的更好维护；
2. CDN 可以更加接近用户的终端，提供加速服务，详细可以查看 CDN 的原理；
3. 静态资源不会具有动态逻辑，单独的域名可以减少页面请求中的 Cookie 等不必要字段，减少 HTTP 带宽。

将静态资源存放到单独的服务器之后，需要做的是配置合理的 HTTP 缓存相关协议，比如我们使用`Cache-Control`告诉浏览器，当前文件的`max-age`：

```
Cache-Control: max-age=31536000
```

上面设置了文件的缓存时间是一年（31536000=360024365）。

完成第一步之后，**第二步**就是要针对发生了变更的静态资源进行重命名，这样静态的文件虽然使用了 CDN 和强缓存，但是只要内容变化，那么文件的路径（网址）发生了变化，浏览器还是会重新请求下载的：

```
<!-- 修改前 -->
<script src="./index-v1.js"></script>

<!-- 修改后 -->
<script src="./index-v2.js"></script>
```

这个方法可以告诉浏览器去下载 JavaScript 文件，并将它缓存，之后使用的都是它的缓存副本。浏览器只会在文件名发生改变（或者一年之后缓存失效）时才会请求网络。在使用 Webpack 构建项目的时候，同样可以做到自动更新，但 Webpack 使用的不是版本号，而是指定哈希值（hash），在之前的文章中不止一次的提到过，Webpack 的 hash 值有三种：

- `hash`：每次编译 Compilation 对象的 hash，全局一致，跟单次编译有关，跟单个文件无关，**不推荐使用**；
- `chunkhash`：chunk 的 hash，根据不同的 chunk 及其包含的模块计算出来的 hash，chunk 中包含的任意模块发生变化，则 chunkhash 发生变化，**推荐使用**；
- `contenthash`：CSS 文件特有的 hash 值，是根据 CSS 文件内容计算出来的，CSS 发生变化则其值发生变化，**推荐 CSS 导出中使用**。

我们在 Webpack 中使用`chunkhash`实际使用的是占位符语法，类似如下：

```
// webpack.config.js
module.exports = {
    entry: './index.js',
    output: {
        filename: 'bundle.[chunkhash:8].js'
        // → bundle.8e0d62a3.js
    }
};
```

一般打包后的文件我们会放到 HTML 页面中使用，这时候需要使用 html-webpack-plugin，它可以自动将打包出来的 JavaScript、CSS 等资源嵌入到 HTML 页面中，对于动态加载的 JavaScript 文件（使用`import()`或者`require.ensure()`引入就是动态加载），可以使用 preload 方式来提前预取，在原理篇我们将亲自动手开发一个 PreloadPlugin 的 Webpack 插件，来给 html-webpack-plugin 插件打包出来的页面增加`<link rel="preload">`内容。

我们 Webpack 打包出来的资源，除了通过 HTML 中使用之外，可能还会需要生成一张包含所有内容的清单文件，这个文件可以用于类似 Application Cache或者 PWA方案，这时候我们需要使用webpack-manifest-plugin插件。webpack-manifest-plugin 是一个扩展性极强的插件，它可以帮助你解决服务端逻辑比较复杂的那部分。在打包时，它会生成一个 JSON 文件，里面包含了原文件名和带哈希文件名的映射。在服务端，通过这个 JSON 就能方便的找到我们真正要执行的文件：

```
// manifest.json
{
  "bundle.js": "bundle.8e0d62a03.js"
}
```

将依赖和 Runtime 提取到单独的文件中

我们的 web 应用中，依赖通常比实际应用内的代码变更频率低，比如我们使用的库 Vue、React 甚至是 jQuery，这些代码并不会频繁的变更，但是我们的业务逻辑可能每天都会有需求，这时候我们应该合理的划分我们的代码，根据变更频率分组。将变更频率低的这些依赖模块移到单独的文件中，这样浏览器就可以独立缓存它们，保证每次应用中的代码变更也不会去重新下载它们。在 Webpack 中要将依赖项提取到独立的 chunk 中，需要执行下面三个步骤：

1. 将输出文件名替换为`[name].[chunkname].js`：

```
// webpack.config.js
module.exports = {
    output: {
        // Before
        filename: 'bundle.[chunkhash].js',
        // After
        filename: '[name].[chunkhash].js'
    }
};
```

这样当 Webpack 编译应用时，它会将`[name]`作为 chunk 的名称。如果我们没有添加`[name]`的部分，我们将不得不通过哈希值来区分 chunk，这样通过辨识哈希值来区分文件的作用会让后期维护变得非常糟糕困难！

2. 我们要将`entry`的值改为对象，而不是单独的字符串或者数组：

```
// webpack.config.js
module.exports = {
    // Before
    entry: './index.js',
    // After
    entry: {
        main: './index.js'
    }
};
```

在上面配置中，`main`是 chunk 的名称。这个名称在 Webpack 打包的过程中，会被第一步的`[name]`所替代。

3. 最后一步就是通过合理的设置`optimization.splitChunks`来划分不同的代码，这部分内容不再展开，之后我会有一篇单独的文章来讲解如何配置`optimization.splitChunks`。

Webpack 的 Runtime

Webpack 打包时，除了模块代码之外，Webpack 的 bundle 中还包含了 Runtime（运行时），这部分代码是一小段用来管理模块执行和加载的代码。当我们将代码拆分成多个文件时，这小部分代码在 chunk id 和匹配的文件之间会生成一个映射表，Webpack 将 Runtime 包含在了最新生成的 chunk 中，这个 chunk 就是我们代码中的`optimization.splitChunks`拆分出来的一部分。每次 chunk 有任何变更，这一小部分代码也会随之更改，同时也会导致整个 chunk 发生改变。

为了解决这个问题，我们可以将 runtime 移动到一个独立的文件中。在 Webpack 中，可以通过开启`optimization.runtimeChunk`选项来实现：

```
// webpack.config.js
module.exports = {
    optimization: {
        runtimeChunk: true
    }
};
```

通过上面的配置，我们打包之后，多出来一个`runtime`命名的文件，这个就是 Webpack 的 Runtime 代码，这段代码比较精短，所以为了达到更好的体验，我们可以尝试把 Webpack 的 Runtime 代码内联到 HTML 中。Runtime 的代码不多，内联到 HTML 中可以帮助我们节省 HTTP 请求（在 HTTP/1 中尤为重要；在 HTTP/2 中虽然没那么重要，但仍然能起到一定作用）。想要内联 Runtime 代码到 HTML 页面中，需要使用一个html-webpack-inline-source-plugin，这个插件实际为 html-webpack-plugin 的插件，所以需要和 html-webpack-plugin 一起使用，详细的配置如下：

```
// webpack.config.js
const HtmlWebpackPlugin = require('html-webpack-plugin');
const InlineSourcePlugin = require('html-webpack-inline-source-plugin');

module.exports = {
    plugins: [
        new HtmlWebpackPlugin({
            // 需要内联的文件名正则，runtime 的正则是包含 runtime 关键字
            inlineSource: 'runtime~.+\\.js'
        }),
        // 注意顺序，这个插件需要配置在HtmlWebpackPlugin之后
        new InlineSourcePlugin()
    ]
};
```

> Tips：在 Webpack 4.29.6 版本已经修改了模板输出的 Template，即使 entry 修改，实际 Runtime 部分的内容也不会有变化，所以上面分离 Runtime 的方案适应于低版本的 Webpack。

合理使用动态加载功能来拆分代码

动态加载代码，指的是通过`import()`或者`require.ensure()`方式来引入模块，这种引入代码的方式，在打包之后会将引入的模块放入一个单独的文件，这种方案适用于将一些不是重要的逻辑或者更加深入操作才能触发的代码逻辑拆分开，保证优先加载首页相关的代码。我曾经有个移动个人主页项目，这个页面中有多个 Tab，每个 Tab 中又有很多类似微博新闻动态的卡片，其中有个卡片类型是视频播放器，这部分引入了一个第三方的视频播放器库，这个播放器库的代码大概占了整个项目的**70%+**，其实播放器是一个不经常用的功能，这是因为这个产品中一般用户发视频状态很少，如果为了这么一个用户不常用功能而导致打包进来一个这么大的播放器库，从而导致所有的用户都要加载这么大的一个 JavaScript 代码，这是很不明智的，所以最后我们将播放器代码使用`import()`异步动态加载了，从而减少了绝大多数用户的页面代码体积。所以合理的使用动态加载功能拆分代码是很重要的优化手段，**当然前提是合理！**

多页面项目按照路由拆分代码

如果我们的项目是一个由多个路由或页面组成的，但是代码中只有一个单独的 JavaScript 文件（一个单独的入口 chunk），这样会导致不管访问任何页面都会加载整站资源，让用户付出额外的流量。此外，如果这个用户经常只是访问其中的某个页面，但是当我们更改了其它页面的代码，Webpack 将会重新编译，那么整个 bundle 的文件名哈希值就会发生变化，最终导致用户重新下载整个网站的代码，造成不必要的浪费。

这时候合理的做法是将整个项目利用多页面打包方案进行划分，我们将代码按照页面进行拆分，这样用户访问某个页面的时候，实际下载的只是当前页面的代码，而不是整个网站的代码，浏览器也更好的缓存了这部分代码，当其他页面代码发生变化的时候，当前代码的哈希值不会失效，自然用户不会重复下载相同的代码了。

关于多页面的配置方法可以参考第二章中的多页面配置相关文章 Webpack 中打包 HTML 和多页面配置。

对于 Vue、React 这类多路由页面，也可以通过`import()`方式来动态加载模块，拿 Vue 来举例说明：

```
Vue.use(Router);
import Vue from 'vue';
import Router from 'vue-router';
export default new Router({
routes: [{
    path: '/a',
    component: () =>
        import ( /* webpackChunkName: 'a' */ '@/pages/a'),
    name: 'a'
}, {
    path: '/b',
    component: () => import ( /* webpackChunkName: 'b' */ '@/pages/b'),
    name: 'b',
    children: [{
            path: 'b/m',
            component: require('@/pages/b/m').default,
            name: 'm',
            children: [{
                path: 'b/m/p',
                component: require('@/pages/b/m/p').default,
                name: 'p'
            }, {
                path: 'b/m/q',
                component: require('@/pages/b/m/q').default,
                name: 'q'
            },
        ]
    }]
}]
```

总结

本小节主要讲解缓存相关的 Webpack 打包优化方案，合理利用 HTTP 请求头的缓存相关字段，然后配合 Webpack 的 `chunkhash` 和 `contenthash` 可以做到根据文件内容和依赖关系变化而增强浏览器缓存，另外根据代码的变更频率合理的拆分代码也能够起到缓存的最大作用，Webpack 中拆分代码用到的是动态加载方式和 `optimization.splitChunks`。关于 `optimization.splitChunks` 拆分代码，接下来的文章将详细的讲解。

> 本小节 Webpack 相关面试题：
>
> 本章节一直在回答一个问题：Webpack 怎么优化。本小节主要从提升缓存命中率方面来介绍 Webpack 优化方案。

17 Webpack 优化之体积优化

19 使用 Webpack 的 splitChunks 功能来拆分代码