

1. 前言
2. 一、代码层面的优化
3. 二、Webpack 层面的优化
4. 三、基础的 Web 技术优化
5. 原文地址: https://blog.csdn.net/qq_37939251/article/details/100031285

前言

Vue 框架通过数据双向绑定和虚拟 DOM 技术，帮我们处理了前端开发中最脏最累的 DOM 操作部分，我们不再需要去考虑如何操作 DOM 以及如何最高效地操作 DOM；但 Vue 项目中仍然存在项目首屏优化、Webpack 编译配置优化等问题，所以我们仍然需要去关注 Vue 项目性能方面的优化，使项目具有更高效的性能、更好的用户体验。本文是作者通过实际项目的优化实践进行总结而来，希望读者读完本文，有一定的启发思考，从而对自己的项目进行优化起到帮助。本文内容分为以下三部分组成：

Vue 代码层面的优化；

webpack 配置层面的优化；

基础的 Web 技术层面的优化。

一、代码层面的优化

1.1、v-if 和 v-show 区分使用场景

v-if 是真正的条件渲染，因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建；也是惰性的：如果在初始渲染时条件为假，则什么也不做——直到条件第一次变为真时，才会开始渲染条件块。

v-show 就简单得多，不管初始条件是什么，元素总是会被渲染，并且只是简单地基于 CSS 的 display 属性进行切换。

所以，v-if 适用于在运行时很少改变条件，不需要频繁切换条件的场景；v-show 则适用于需要非常频繁切换条件的场景。

1.2、computed 和 watch 区分使用场景

computed：是计算属性，依赖其它属性值，并且 computed 的值有缓存，只有它依赖的属性值发生改变，下一次获取 computed 的值时才会重新计算 computed 的值；

watch: 更多的是「观察」的作用，类似于某些数据的监听回调，每当监听的数据变化时都会执行回调进行后续操作；

运用场景：

当我们需要进行数值计算，并且依赖于其它数据时，应该使用 **computed**，因为可以利用 **computed** 的缓存特性，避免每次获取值时，都要重新计算；

当我们需要在数据变化时执行异步或开销较大的操作时，应该使用 **watch**，使用 **watch** 选项允许我们执行异步操作（访问一个 **API**），限制我们执行该操作的频率，并在我们得到最终结果前，设置中间状态。这些都是计算属性无法做到的。

1.3、v-for 遍历必须为 item 添加 key，且避免同时使用 v-if

（1）v-for 遍历必须为 item 添加 key

在列表数据进行遍历渲染时，需要为每一项 **item** 设置唯一 **key** 值，方便 **Vue.js** 内部机制精准找到该条列表数据。当 **state** 更新时，新的状态值和旧的状态值对比，较快地定位到 **diff**。

（2）v-for 遍历避免同时使用 v-if

v-for 比 **v-if** 优先级高，如果每一次都需要遍历整个数组，将会影响速度，尤其是当之需要渲染很小一部分的时候，必要情况下应该替换成 **computed** 属性。

推荐：

```
<ul>
  <li
    v-for="user in activeUsers"
    :key="user.id">
    {{ user.name }}
  </li>
</ul>
computed: {
  activeUsers: function () {
    return this.users.filter(function (user) {
      return user.isActive
    })
  }
}
```

不推荐：

```
<ul>
  <li
    v-for="user in users">
```

```
      v-if="user.isActive"
      :key="user.id">
    {{ user.name }}
  </li>
</ul>
```

1.4、长列表性能优化

Vue 会通过 `Object.defineProperty` 对数据进行劫持，来实现视图响应数据的变化，然而有些时候我们的组件就是纯粹的数据展示，不会有任何改变，我们就不需要 **Vue** 来劫持我们的数据，在大量数据展示的情况下，这能够很明显的减少组件初始化的时间，那如何禁止 **Vue** 劫持我们的数据呢？可以通过 `Object.freeze` 方法来冻结一个对象，一旦被冻结的对象就再也不能被修改了。

```
export default {
  data: () => ({
    users: {}
  }),
  async created() {
    const users = await axios.get("/api/users");
    this.users = Object.freeze(users);
  }
};
```

1.5、事件的销毁

Vue 组件销毁时，会自动清理它与其它实例的连接，解绑它的全部指令及事件监听器，但是仅限于组件本身的事件。如果在 `js` 内

```
created() {
  addEventListener('click', this.click, false)
},
beforeDestroy() {
  removeEventListener('click', this.click, false)
}
```

1.6、图片资源懒加载

对于图片过多的页面，为了加速页面加载速度，所以很多时候我们需要将页面内未出现在可视区域内的图片先不做加载，等到滚动到可视区域后再去加载。这样对于页面加载性能上会有很大的提升，也提高了用户体验。我们在项目中使用 **Vue** 的 `vue-lazyload` 插件：

（1）安装插件

```
npm install vue-lazyload --save-dev
```

(2) 在入口文件 `main.js` 中引入并使用

```
import VueLazyload from 'vue-lazyload'
```


然后再 `vue` 中直接使用

```
Vue.use(VueLazyload)
```

或者添加自定义选项

```
Vue.use(VueLazyload, {  
  preLoad: 1.3,  
  error: 'dist/error.png',  
  loading: 'dist/loading.gif',  
  attempt: 1  
})
```

(3) 在 `vue` 文件中将 `img` 标签的 `src` 属性直接改为 `v-lazy`，从而将图片显示方式更改为懒加载显示：

 以上为 `vue-lazyload` 插件的简单使用，如果要看插件的更多参数选项，可以查看 `vue-lazyload` 的 [github](#) 地址。

1.7、路由懒加载 `Vue` 是单页面应用，可能会有很多的路由引入，这样使用 `webpack` 打包后的文件很大，当进入首页时，加载的资源过多，页面会出现白屏的情况，不利于用户体验。如果我们能把不同路由对应的组件分割成不同的代码块，然后当路由被访问的时候才加载对应的组件，这样就更加高效了。这样会大大提高首屏显示的速度，但是可能其他的页面的速度就会降下来。

路由懒加载：

```
const Foo = () => import('./Foo.vue')  
const router = new VueRouter({  
  routes: [  
    { path: '/foo', component: Foo }  
  ]  
})
```

1.8、第三方插件的按需引入

我们在项目中经常会需要引入第三方插件，如果我们直接引入整个插件，会导致项目的体积太大，我们可以借助 **babel-plugin-component**，然后可以只引入需要的组件，以达到减小项目体积的目的。以下为项目中引入 **element-ui** 组件库为例：

(1) 首先，安装 **babel-plugin-component**：

```
npm install babel-plugin-component -D
```

(2) 然后，将 **.babelrc** 修改为：

```
{
  "presets": [["es2015", { "modules": false }]],
  "plugins": [
    [
      "component",
      {
        "libraryName": "element-ui",
        "styleLibraryName": "theme-chalk"
      }
    ]
  ]
}
```

(3) 在 **main.js** 中引入部分组件：

```
import Vue from 'vue';
import { Button, Select } from 'element-ui';

Vue.use(Button)
Vue.use(Select)
```

1.9、优化无限列表性能

如果你的应用存在非常长或者无限滚动的列表，那么需要采用 窗口化 的技术来优化性能，只需要渲染少部分区域的内容，减少重新渲染组件和创建 **dom** 节点的时间。你可以参考以下开源项目 **vue-virtual-scroll-list** 和 **vue-virtual-scroller** 来优化这种无限列表的场景的。

1.10、服务端渲染 SSR or 预渲染

服务端渲染是指 **Vue** 在客户端将标签渲染成的整个 **html** 片段的工作在服务端完成，服务端形成的 **html** 片段直接返回给客户端这个过程就叫做服务端渲染。

（1）服务端渲染的优点：

更好的 **SEO**：因为 **SPA** 页面的内容是通过 **Ajax** 获取，而搜索引擎爬取工具并不会等待 **Ajax** 异步完成后再抓取页面内容，所以在 **SPA** 中是抓取不到页面通过 **Ajax** 获取到的内容；而 **SSR** 是直接由服务端返回已经渲染好的页面（数据已经包含在页面中），所以搜索引擎爬取工具可以抓取渲染好的页面；

更快的内容到达时间（首屏加载更快）：**SPA** 会等待所有 **Vue** 编译后的 **js** 文件都下载完成后，才开始进行页面的渲染，文件下载等需要一定的时间等，所以首屏渲染需要一定的时间；**SSR** 直接由服务端渲染好页面直接返回显示，无需等待下载 **js** 文件及再去渲染等，所以 **SSR** 有更快的内容到达时间；

（2）服务端渲染的缺点：

更多的开发条件限制：例如服务端渲染只支持 **beforeCreate** 和 **created** 两个钩子函数，这会导致一些外部扩展库需要特殊处理，才能在服务端渲染应用程序中运行；并且与可以部署在任何静态文件服务器上的完全静态单页面应用程序 **SPA** 不同，服务端渲染应用程序，需要处于 **Node.js server** 运行环境；

更多的服务器负载：在 **Node.js** 中渲染完整的应用程序，显然会比仅提供静态文件的 **server** 更加大量占用 **CPU** 资源，因此如果你预料在高流量环境下使用，请准备相应的服务器负载，并明智地采用缓存策略。

如果你的项目的 **SEO** 和 首屏渲染是评价项目的关键指标，那么你的项目就需要服务端渲染来帮助你实现最佳的初始加载性能和 **SEO**，具体的 **Vue SSR** 如何实现，可以参考作者的另一篇文章《**Vue SSR** 踩坑之旅》。如果你的 **Vue** 项目只需改善少数营销页面（例如 **/**，**/about**，**/contact** 等）的 **SEO**，那么你可能需要预渲染，在构建时 (**build time**) 简单地生成针对特定路由的静态 **HTML** 文件。优点是设置预渲染更简单，并可以将你的前端作为一个完全静态的站点，具体你可以使用 **prerender-spa-plugin** 就可以轻松地添加预渲染。

二、Webpack 层面的优化

2.1、Webpack 对图片进行压缩

在 **vue** 项目中除了可以在 **webpack.base.conf.js** 中 **url-loader** 中设置 **limit** 大小来对图片处理，对小于 **limit** 的图片转化为 **base64** 格式，其余的不做操作。所以对有些较大的图片资源，在请求资源的时候，加载会很慢，我们可以用 **image-webpack-loader** 来压缩图片：

（1）首先，安装 **image-webpack-loader**：

```
npm install image-webpack-loader --save-dev
```

(2) 然后，在 `webpack.base.conf.js` 中进行配置：

```
{
  test: /\. (png|jpe?g|gif|svg) (\?.*)?$/,
  use: [
    {
      loader: 'url-loader',
      options: {
        limit: 10000,
        name: utils.assetsPath('img/[name].[hash:7].[ext]')
      }
    },
    {
      loader: 'image-webpack-loader',
      options: {
        bypassOnDebug: true,
      }
    }
  ]
}
```

2.2、减少 ES6 转为 ES5 的冗余代码 Babel 插件会在将 ES6 代码转换成 ES5 代码时会注入一些辅助函数，例如下面的 ES6 代码：

```
class HelloWebpack extends Component{...}
```

这段代码再被转换成能正常运行的 ES5 代码时需要以下两个辅助函数：

```
babel-runtime/helpers/createClass // 用于实现 class 语法
babel-runtime/helpers/inherits // 用于实现 extends 语法
```

在默认情况下，**Babel** 会在每个输出文件中内嵌这些依赖的辅助函数代码，如果多个源代码文件都依赖这些辅助函数，那么这些辅助函数的代码将会出现很多次，造成代码冗余。为了不让这些辅助函数的代码重复出现，可以在依赖它们时通过 `require('babel-runtime/helpers/createClass')` 的方式导入，这样就能做到只让它们出现一次。**babel-plugin-transform-runtime** 插件就是用来实现这个作用的，将相关辅助函数进行替换成导入语句，从而减小 **babel** 编译出来的代码的文件大小。

(1) 首先，安装 **babel-plugin-transform-runtime**：

```
npm install babel-plugin-transform-runtime --save-dev
```

(2) 然后, 修改 `.babelrc` 配置文件为:

```
"plugins": [  
  "transform-runtime"  
]
```

如果要看插件的更多详细内容, 可以查看 `babel-plugin-transform-runtime` 的 详细介绍。

2.3、提取公共代码

如果项目中没有去将每个页面的第三方库和公共模块提取出来, 则项目会存在以下问题:

相同的资源被重复加载, 浪费用户的流量和服务器的成本。

每个页面需要加载的资源太大, 导致网页首屏加载缓慢, 影响用户体验。

所以我们需要将多个页面的公共代码抽离成单独的文件, 来优化以上问题。Webpack 内置了专门用于提取多个 `Chunk` 中的公共部分的插件 `CommonsChunkPlugin`, 我们在项目中 `CommonsChunkPlugin` 的配置如下:

// 所有在 `package.json` 里面依赖的包, 都会被打包进 `vendor.js` 这个文件中。

```
new webpack.optimize.CommonsChunkPlugin({  
  name: 'vendor',  
  minChunks: function(module, count) {  
    return (  
      module.resource &&  
      /\.js$/.test(module.resource) &&  
      module.resource.indexOf(  
        path.join(__dirname, '../node_modules')  
      ) === 0  
    );  
  },  
}),  
// 抽取出代码模块的映射关系  
new webpack.optimize.CommonsChunkPlugin({  
  name: 'manifest',  
  chunks: ['vendor']  
})
```

如果要看插件的更多详细内容, 可以查看 `CommonsChunkPlugin` 的 详细介绍。

2.4、模板预编译

当使用 **DOM** 内模板或 **JavaScript** 内的字符串模板时，模板会在运行时被编译为渲染函数。通常情况下这个过程已经足够快了，但对性能敏感的应用还是最好避免这种用法。

预编译模板最简单的方式就是使用单文件组件——相关的构建设置会自动把预编译处理好，所以构建好的代码已经包含了编译出来的渲染函数而不是原始的模板字符串。

如果你使用 **webpack**，并且喜欢分离 **JavaScript** 和模板文件，你可以使用 **vue-template-loader**，它也可以在构建过程中把模板文件转换成为 **JavaScript** 渲染函数。

2.5、提取组件的 CSS

当使用单文件组件时，组件内的 **CSS** 会以 **style** 标签的方式通过 **JavaScript** 动态注入。这有一些小小的运行时开销，如果你使用服务端渲染，这会导致一段“无样式内容闪烁 (fouc)”。将所有组件的 **CSS** 提取到同一个文件可以避免这个问题，也会让 **CSS** 更好地进行压缩和缓存。

查阅这个构建工具各自的文档来了解更多：

webpack + vue-loader (**vue-cli** 的 **webpack** 模板已经预先配置好)

Browserify + vueify

Rollup + rollup-plugin-vue

2.6、优化 SourceMap

我们在项目进行打包后，会将开发中的多个文件代码打包到一个文件中，并且经过压缩、去掉多余的空格、**babel**编译化后，最终将编译得到的代码会用于线上环境，那么这样处理后的代码和源代码会有很大的差别，当有 **bug**的时候，我们只能定位到压缩处理后的代码位置，无法定位到开发环境中的代码，对于开发来说不好调式定位问题，因此 **sourceMap** 出现了，它就是为了解决不好调式代码问题的。

SourceMap 的可选值如下（+ 号越多，代表速度越快，- 号越多，代表速度越慢, o 代表中等速度）

开发环境推荐：cheap-module-eval-source-map

生产环境推荐：cheap-module-source-map

原因如下：

cheap：源代码中的列信息是没有任何作用，因此我们打包后的文件不希望包含列相关信息，只有行信息能建立打包前后的依赖关系。因此不管是开发环境或生产环境，我们

都希望添加 `cheap` 的基本类型来忽略打包前后的列信息；

module：不管是开发环境还是正式环境，我们都希望能定位到bug的源代码具体的位置，比如说某个 `Vue` 文件报错了，我们希望能定位到具体的 `Vue` 文件，因此我们也需要 `module` 配置；

source-map：`source-map` 会为每一个打包后的模块生成独立的 `soucemap` 文件，因此我们需要增加 `source-map` 属性；

eval-source-map：`eval` 打包代码的速度非常快，因为它不生成 `map` 文件，但是可以对 `eval` 组合使用 `eval-source-map` 使用会将 `map` 文件以 `DataURL` 的形式存在打包后的 `js` 文件中。在正式环境中不要使用 `eval-source-map`，因为它会增加文件的大小，但是在开发环境中，可以试用下，因为他们打包的速度很快。

2.7、构建结果输出分析

Webpack 输出的代码可读性非常差而且文件非常大，让我们非常头疼。为了更简单、直观地分析输出结果，社区中出现了许多可视化分析工具。这些工具以图形的方式将结果更直观地展示出来，让我们快速了解问题所在。接下来讲解我们在 `Vue` 项目中用到的分析工具：`webpack-bundle-analyzer`。

我们在项目中 `webpack.prod.conf.js` 进行配置：

```
if (config.build.bundleAnalyzerReport) {
  var BundleAnalyzerPlugin = require('webpack-bundle-analyzer').BundleAnalyzerPlugin;
  webpackConfig.plugins.push(new BundleAnalyzerPlugin());
}
```

执行 `$ npm run build --report` 后生成分析报告如下：

2.8、Vue 项目的编译优化

如果你的 `Vue` 项目使用 **Webpack** 编译，需要你喝一杯咖啡的时间，那么也许你需要对项目的 **Webpack** 配置进行优化，提高 **Webpack** 的构建效率。具体如何进行 `Vue` 项目的 **Webpack** 构建优化，可以参考作者的另一篇文章《`Vue` 项目 **Webpack** 优化实践》

三、基础的 Web 技术优化

3.1、开启 `gzip` 压缩

gzip 是 GNUzip 的缩写，最早用于 UNIX 系统的文件压缩。HTTP 协议上的 gzip 编码是一种用来改进 web 应用程序性能的技术，web 服务器和客户端（浏览器）必须共同支持 gzip。目前主流的浏览器，Chrome，firefox，IE等都支持该协议。常见的服务器如 Apache，Nginx，IIS 同样支持，gzip 压缩效率非常高，通常可以达到 70% 的压缩率，也就是说，如果你的网页有 30K，压缩之后就变成了 9K 左右

以下我们以服务端使用我们熟悉的 express 为例，开启 gzip 非常简单，相关步骤如下：

安装：

npm install compression --save 添加代码逻辑：

```
var compression = require('compression');
var app = express();
app.use(compression());
```

重启服务，观察网络面板里面的 response header，如果看到如下红圈里的字段则表明 gzip 开启成功：

3.2、浏览器缓存

为了提高用户加载页面的速度，对静态资源进行缓存是非常必要的，根据是否需要重新向服务器发起请求来分类，将 HTTP 缓存规则分为两大类（强制缓存，对比缓存），如果对缓存机制还不是了解很清楚的，可以参考作者写的关于 HTTP 缓存的文章《深入理解HTTP缓存机制及原理》，这里不再赘述。

3.3、CDN 的使用

浏览器从服务器上下载 CSS、js 和图片等文件时都要和服务器连接，而大部分服务器的带宽有限，如果超过限制，网页就半天反应不过来。而 CDN 可以通过不同的域名来加载文件，从而使下载文件的并发连接数大大增加，且CDN 具有更好的可用性，更低的网络延迟和丢包率。

3.4、使用 Chrome Performance 查找性能瓶颈

Chrome 的 Performance 面板可以录制一段时间内的 js 执行细节及时间。使用 Chrome 开发者工具分析页面性能的步骤如下。

打开 Chrome 开发者工具，切换到 Performance 面板

点击 Record 开始录制

刷新页面或展开某个节点

原文地址:

https://blog.csdn.net/qq_37939251/article/details/100031285
