# Vue2框架实践

# 新旧项目容错

## 项目URL

由于项目本身打包构建后目录为/wap/e，而考虑到访问路径中并不一定存在/wap路径，只能使用相对路径，现实情况如下：

* <http://m.egatee.com/tmpl/product_first_categroy.html>
* <http://ug.egatee.com/wap/tmpl/member/login.html>

相对路径使用技巧：

* 在/wap/e 目录中（新vue项目），使用../tmpl/product\_first\_category.html链接，相对于tmpl目录
* 在/wap/tmpl目录中（旧项目），使用../../e/#login类似进行链接。相对于e目录

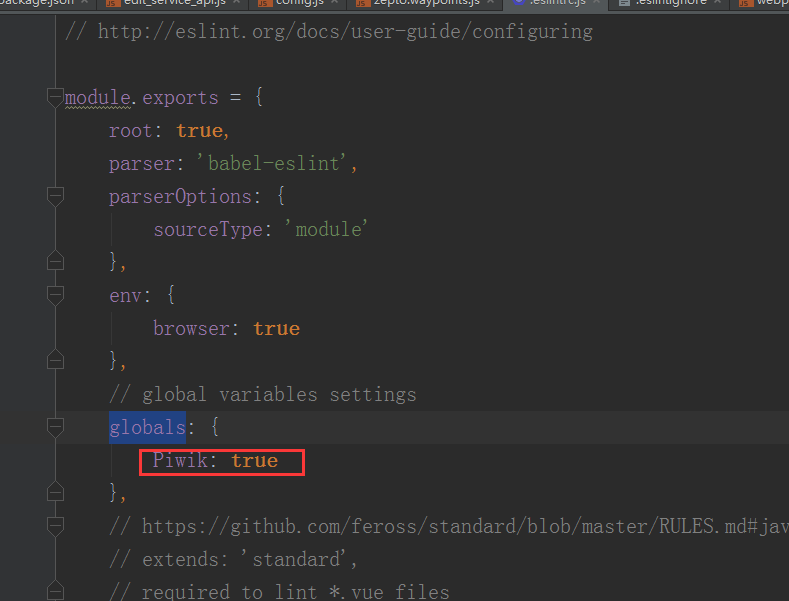
## 项目统一配置

为了兼容旧项目配置，引入/wap/js/config.js文件。统一配置各个环境的全局变量配置。



但为了兼容eslint不允许直接写全局变量，解决方案有两种：

* window.Enviroment的写法
* 在.eslintrc.js文件globals中加入全局变量名字，并设置为true



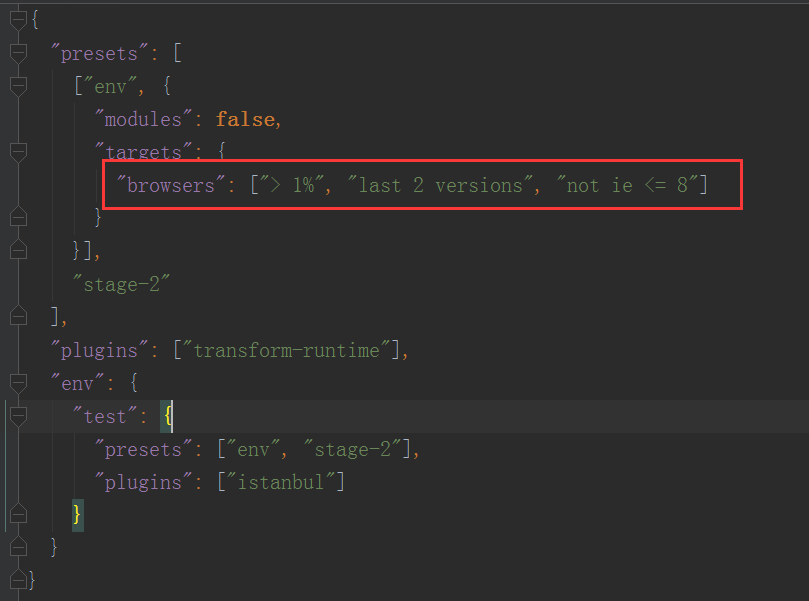
## ES特性浏览器兼容

含义：一般指当前使用的JS特性在环境中不支持，例如Promise在IE10不支持等。

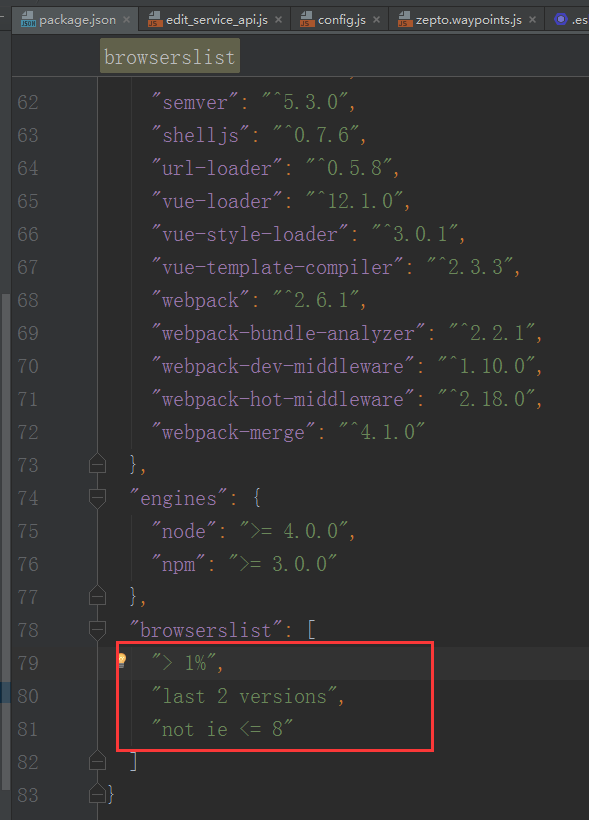
前提：**Vue2体系框架必须支持Html5特性，大于IE8环境才可以运行**

**解决方案：**

* 在.babelrc中presets->env->targets->browsers属性，例如：not ie <=8 或chrome>40等

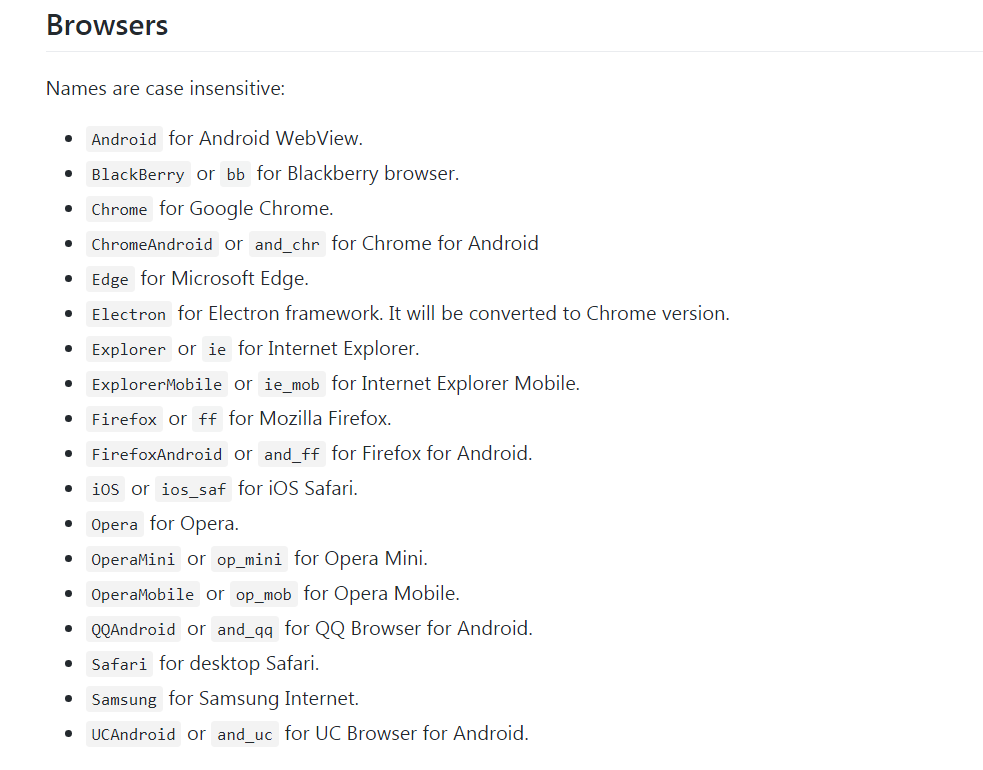


* 如果不存在.babelrc文件，请在package.json中browserslist属性，配置方法一致。

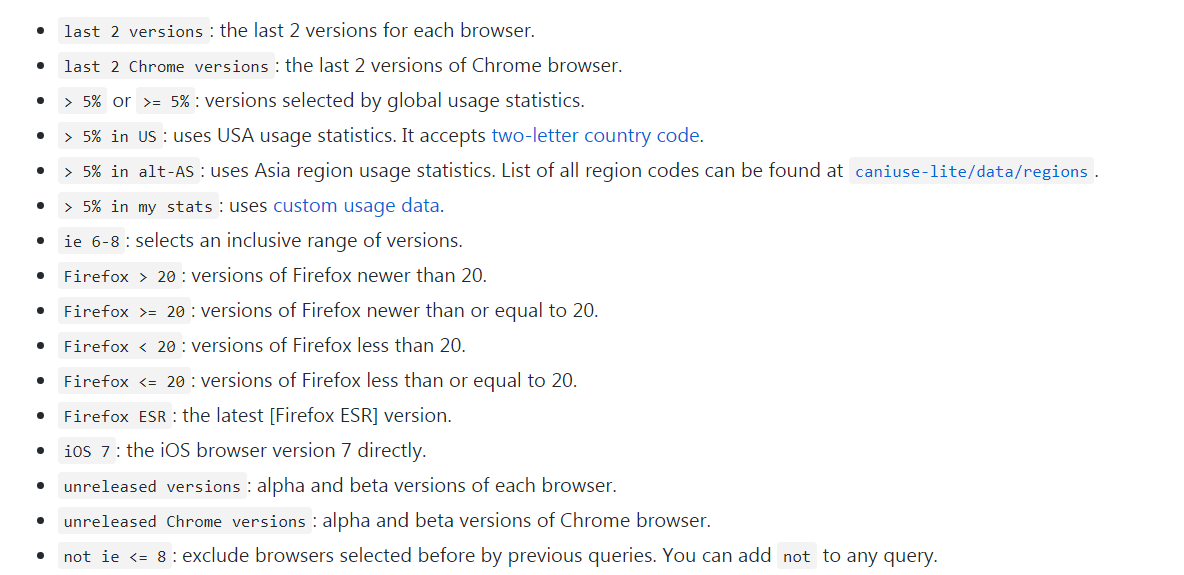


关于browserlist设置规则，请参考：

* <https://github.com/ai/browserslist>
* <https://github.com/babel/babel-preset-env>



（常见的Broswers）



(官方的设置Demo)

最新的ES特性兼容情况请参考：<http://kangax.github.io/compat-table/es6/>

# Webpack/Babel打包分析

## CommonChunk组成

主体上由Vue+Router+Babel-runtime+Axios+Vuex：

* Axios （35.5kb nouglify）
* ***Babel-presets （39.5kb）***
* ***Babel-runtime （8kb按需）***
* Fastclick （26kb）
* Js-cookie（4kb）
* Vue（262kb）
* Vue-router（62kb）
* ***Vue-loader/style（10kb）***
* Vuex（22kb）

**总结：**共计467kb左右，未min化版本，经过webpack min化后161kb，称为基本框架库组成部分，后续**错误排查**、**体积优化**可以从这方面入手。另外关于Babel是因为目前需要兼容ES5等低端机型所致，可以通过[浏览器兼容](#_ES特性浏览器兼容)来平衡体积与性能。

可以了解下Bundler-analyzer：<https://github.com/th0r/webpack-bundle-analyzer>

## Presets-env

常见的ES6/ES2015语法特性：



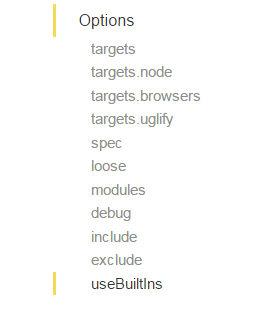
对于ES6**语法特性**让Javascript简洁优雅而又不失便利，但浏览器兼容性问题总让人头疼不已，为了解决这个问题Babel就这样诞生了（**体验最新的ES标准**）。包含TC39中的各种标准和提案：

* ES2015 ES6
* ES2016 ES7
* ES2017 ES8
* Stage-0 建议
* Stage-1 提案
* Stage-2 草案，第一版本提案
* Stage-3 候选
* Stage-4 完成

**等等**

但语法支持比较难以固定而应该是与时俱进，**官方推荐使用presets-env，允许通过browsers来设置支持ES5+特性的转换，而不需要去指定具体的Presets集合。***（根据你设定的环境，结合Compat-table自动侦测你需要的插件）*

常见的Options：



使用方法，亦可以参考[ES特性浏览器兼容](#_ES特性浏览器兼容)：

****.babelrc****

****{****

****"presets": [****

****["env", {****

****"targets": {****

****"chrome": 52,****

****"browsers": ["last 2 versions", "not ie <=8"]****

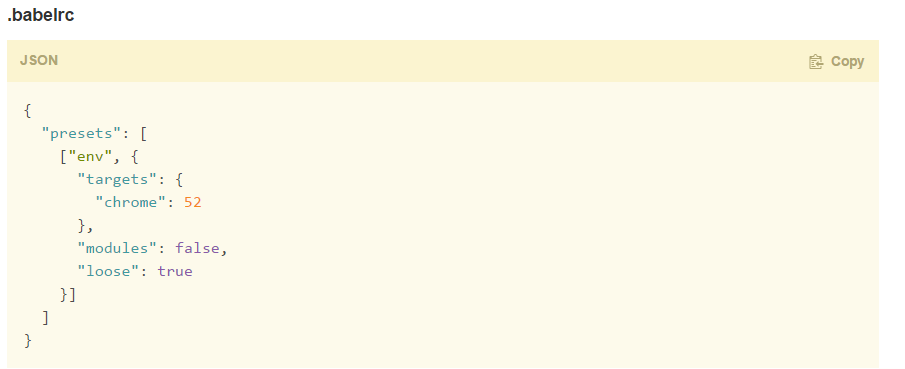
****}****

****}]****

****]****

****}****

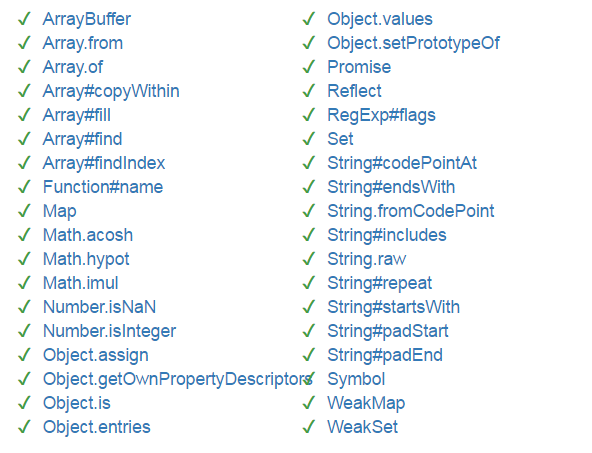
****表示为所有浏览器的最新2个版本和完全兼容IE8以上的IE系列，但不包括edge。****



插件文档和用法：<https://babeljs.io/docs/plugins/preset-env/>

## Api-Polyfill

**由于Babel仅仅在compiler时转换语法，并不会对ES2015+的原生对象及相关API拓展进行处理并兼容**。常见的API特性如下：



API特性对于JavaScript易用性极大的增强，若不考虑过低浏览器兼容性问题。亦能获取更好地脚本性能。*例如，Promise特性于ES2016中API，目前除了IE不支持其它浏览器均于2014年发布的浏览器版本就已经支持该特性，数据来自caniuse.com。***因此为了适应前端的发展，可以适度使用一些API特性提高性能和开发效率，目前解决方案如下：**

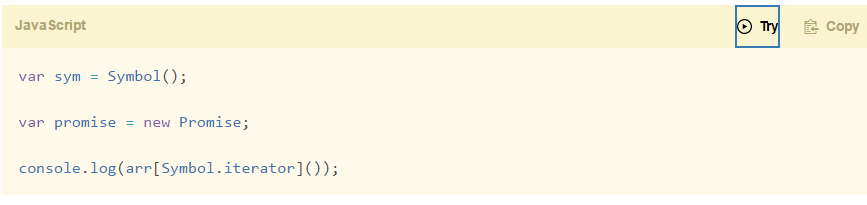
* Babel-polyfill
* Babel-runtime/babel-plugin-transform-runtime

Babel-polyfill特点：

* 入口文件引入，***并且只能引入一次***。
* 污染全局原生对象，例如生成Promise全局对象
* 内部引用babel-runtime和core-js并全部引入，体积比较大
* ES2015+全部功能（原生对象、原生对象静态方法、生成器函数、**实例方法**）

Babel-Plugin-Transform-runtime/Babel-Runtime插件：

* 无需代码引入，webpack等构建工具按需引入，但会适当转换源代码
* 不污染全局变量。打包时引入babel-runtime和core-js中函数库，**对于高级API会存在一定代码增量，但整体而言对体积影响很小**
* ES2015+大部分功能（原生对象、原生对象静态方法、生成器函数）



使用babel-runtime转换后：



## 其它关键点

### 关于presets与plugins顺序

* Plugins优先于presets执行
* Plugins数组从第一个开始执行
* Presets数组从最后一个开始执行

### Webpack内部Promise容错

由于使用的Babel只能对源码及公用库进行转码，对于Webpack内部的模块公共代码无法进行转码。在使用**AMD风格Require**或**CommonJS中Require.Ensure**进行**代码分割按需加载时，Webpack内部依赖Promise**，在IE或低端机型WebView中存在不兼容情况，解决方案：

* 使用babel-polyfill代替babel-runtime
* 使用es-promise兼容库，新增js库体积15kb左右
* 将Babel-runtime中Promise Polyfill 引入至全局变量Promise中。

**在入口文件中加入如下代码进行容错：**

If(!window.Promise){

window.Promise = Promise;

}

# 代码分离/Async

使用Webpack作为构建工具，容易使的一个bunder或chunk越变越大并无法做到***按需使用与资源平行加载，以此来优化性能和节省流量***。因此webpack提供了code spliting的功能来解决此问题，常用的解决方案：

* Entry Point-入口分离，变成多入口文件
* 使用CommonChunkPlugin提取公用代码
* Dynamic Imports-代码内部异步分离

在Vue2中，这三种方法各有优势，我对Code Spliting的实践与理解是：

* Entry Point对于单页不能直接使用，但可以结合**Vue-Router的路由懒加载**机制来实现。亦是[3.1路由懒加载](#_路由代码分割)主体内容
* CommonChunkPlugin指提取公用modules或公用代码，但在SPA应用中只能识别非**路由懒加载内容**。通常会将vue+vue-router+axios+FastClick等框架核心库提取成CommonChunk
* Dynamic Imports一般直接适用于内部代码指定按需。比如：List页面有edit-list-dialog.vue弹窗、Toast提示框、图表组件等等均可用这种方式去处理，加快首屏渲染速度和资源共享。

## Dynamic Imports介绍

Vue2+WebPack+Babel算比较前沿的技术框架，能够使用的**动态加载技术**比较多，下面介绍几种的主流方案，细节需要自己去实践和理解：

* CommonJS中Require.Ensure，也是Webpack推荐使用的方式
* AMD中Require按需
* Import（TC39/proposals Stage-3）内置Promise玩法，由于我们使用了babel-runtime进行按需转换。
* ES2017 Async与Await 今年刚纳入W3C标准

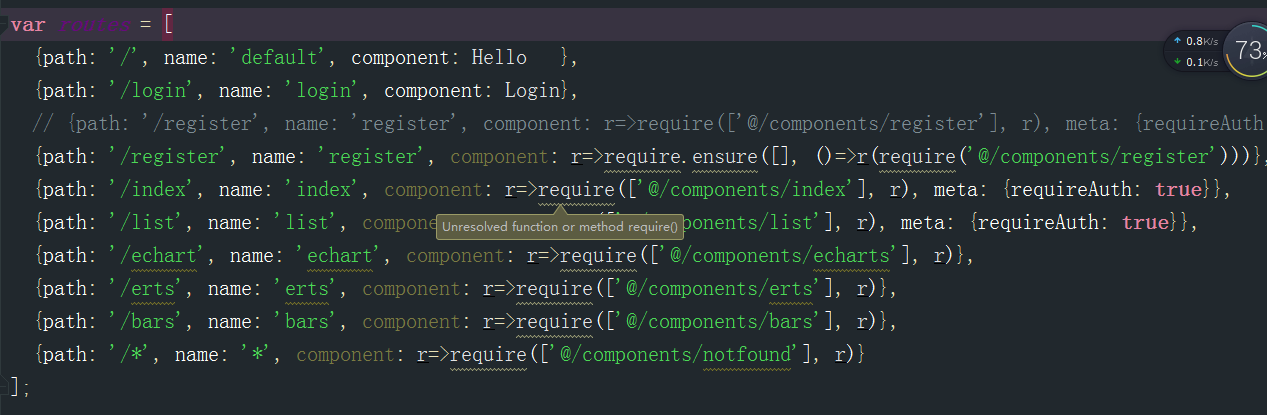
除了上述方案外，webpack 也以loader的方式提供了解决方案：

* Bundle-loader
* Promise-loader

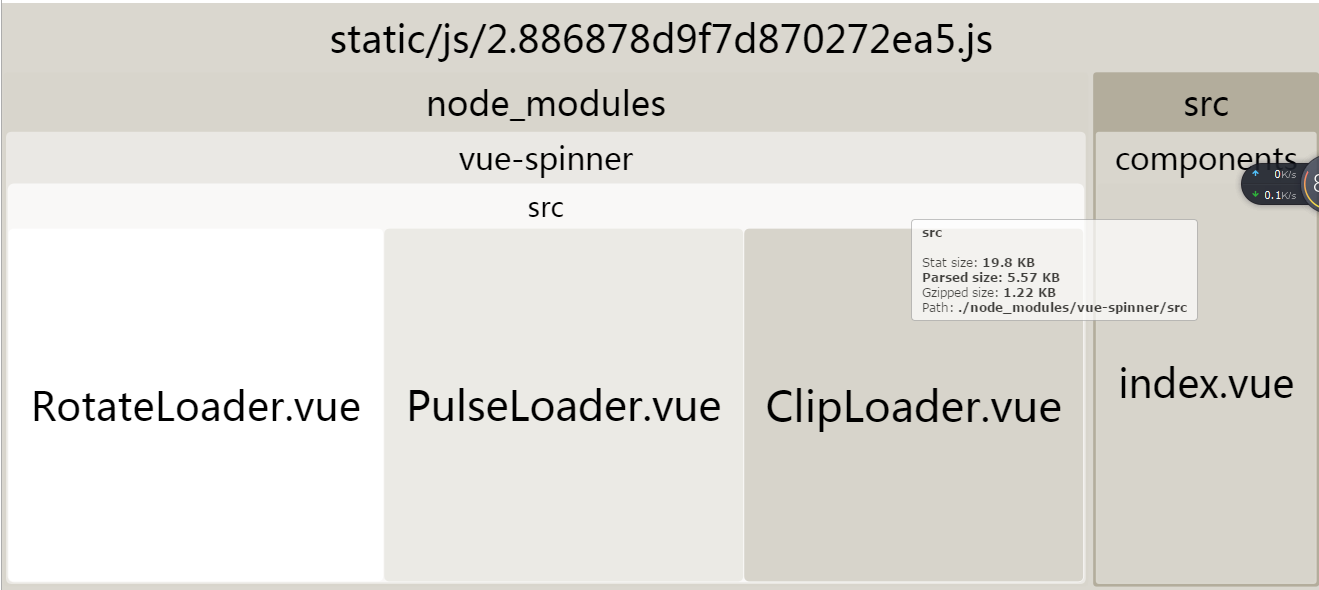
## 路由懒加载

当打包构建应用时，Javascript 包会变得非常大，影响页面加载。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组件，这样就更加高效了。

结合Vue异步组件与Webpack的Code Spliting，轻松实现路由组件的懒惰加载。



不管是采用CommonJS风格还是AMD风格，都会将路由组件指定的组件当成一个新的entry，每个组件路由将会打包自己的依赖（***CommonChunk除外***）



（Index组件打包文件）

备注：***注意在路由组件里面，不存在再次抽取Common Chunk的概念，如果单个组件比较复杂，依然存在JS文件过大问题，并且导致通用资源分别打包到各个路由组件中，并不能浏览器cache进行节流和提升性能。因此需要将Common组件或JS以Dynamic imports的方式去处理。***

## Common Javascript

由于路由组件懒加载能解决大部分问题，但依然有一定的局限性。

例如：路由组件List、Account页面同时依赖下面几个库：

1. 拓展Date操作js文件模块.
2. 拓展Cookie操作库
3. LocalStorage操作库

另外List组件依赖scroll库，而Account组件单独依赖slide。***此时如果用webpack进行打包的话。Date、Cookie、LocalStorage组件会同时打包到List和Account的js中***。

从性能和最优的角度分析：

1. Date、Cookie、LocalStorage重复打包，无法利用浏览器cache能力
2. 打包后List和Account体积都增大，不利于节流
3. Date、Cookie、LocalStorage使用的时机不确定，或许有更优的首屏加载方案去处理。例如：Date和LocalStorage仅仅是在某按钮Click事件中使用，其实可以按照Click事件中做延迟按需，首次则不必要硬性加载。
4. 用户体验和容错能力下降。假定Cookie或Date库在下载脚本后立即执行一部分内容，存在兼容性或代码错误问题，则会直接影响功能加载。

因此对于Common脚本来说，可以适当采用Dynamic import方案去处理，常见处理方法：

1. Vue2生命周期created/mounted 采用require.ensure处理延迟加载内容或库
2. Vue2事件Handle中采用Import、require.ensure延迟按需加载内容库



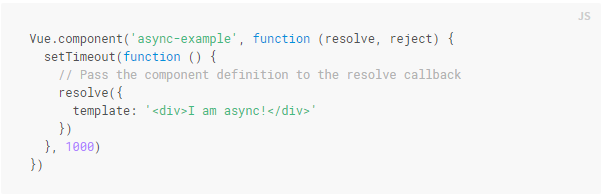
（延迟ECharts加载）

参考Webpack文档：<https://webpack.js.org/guides/code-splitting/>

## Async Component

Vue2框架最强大的是组件系统，用的最多的也算组件（路由组件、弹窗组件、提示框组件、列表组件、按钮组件等），因此基本每个业务功能中都存在多个Vue 组件。组件的复用、缓存与延迟加载也是高性能的关键点。

Vue2本身提供Async Component概念，注册组件时为Promise函数（resolve,rejected）或直接返回Promise。



注册方式主要分两种：

* Global Async Component 全局注册的异步组件



* Scoped Async Component 局部注册的异步组件



Vue2.3+ 新增高级异步组件，支持异步组件异常容错控制。

