## **Vue.js 源码构建**

# **Vue.js 源码构建**

Vue.js 源码是基于 [Rollup](https://github.com/rollup/rollup" \t "https://coding.imooc.com/lesson/_blank) 构建的，它的构建相关配置都在 scripts 目录下。

## **构建脚本**

通常一个基于 NPM 托管的项目都会有一个 package.json 文件，它是对项目的描述文件，它的内容实际上是一个标准的 JSON 对象。

我们通常会配置 script 字段作为 NPM 的执行脚本，Vue.js 源码构建的脚本如下：

{

"script": {

"build": "node scripts/build.js",

"build:ssr": "npm run build -- web-runtime-cjs,web-server-renderer",

"build:weex": "npm run build --weex"

}

}

这里总共有 3 条命令，作用都是构建 Vue.js，后面 2 条是在第一条命令的基础上，添加一些环境参数。

当在命令行运行 npm run build 的时候，实际上就会执行 node scripts/build.js，接下来我们来看看它实际是怎么构建的。

## **构建过程**

我们对于构建过程分析是基于源码的，先打开构建的入口 JS 文件，在 scripts/build.js 中：

let builds = require('./config').getAllBuilds()

// filter builds via command line arg

if (process.argv[2]) {

const filters = process.argv[2].split(',')

builds = builds.filter(b => {

return filters.some(f => b.output.file.indexOf(f) > -1 || b.\_name.indexOf(f) > -1)

})

} else {

// filter out weex builds by default

builds = builds.filter(b => {

return b.output.file.indexOf('weex') === -1

})

}

build(builds)

这段代码逻辑非常简单，先从配置文件读取配置，再通过命令行参数对构建配置做过滤，这样就可以构建出不同用途的 Vue.js 了。接下来我们看一下配置文件，在 scripts/config.js 中：

const builds = {

// Runtime only (CommonJS). Used by bundlers e.g. Webpack & Browserify

'web-runtime-cjs': {

entry: resolve('web/entry-runtime.js'),

dest: resolve('dist/vue.runtime.common.js'),

format: 'cjs',

banner

},

// Runtime+compiler CommonJS build (CommonJS)

'web-full-cjs': {

entry: resolve('web/entry-runtime-with-compiler.js'),

dest: resolve('dist/vue.common.js'),

format: 'cjs',

alias: { he: './entity-decoder' },

banner

},

// Runtime only (ES Modules). Used by bundlers that support ES Modules,

// e.g. Rollup & Webpack 2

'web-runtime-esm': {

entry: resolve('web/entry-runtime.js'),

dest: resolve('dist/vue.runtime.esm.js'),

format: 'es',

banner

},

// Runtime+compiler CommonJS build (ES Modules)

'web-full-esm': {

entry: resolve('web/entry-runtime-with-compiler.js'),

dest: resolve('dist/vue.esm.js'),

format: 'es',

alias: { he: './entity-decoder' },

banner

},

// runtime-only build (Browser)

'web-runtime-dev': {

entry: resolve('web/entry-runtime.js'),

dest: resolve('dist/vue.runtime.js'),

format: 'umd',

env: 'development',

banner

},

// runtime-only production build (Browser)

'web-runtime-prod': {

entry: resolve('web/entry-runtime.js'),

dest: resolve('dist/vue.runtime.min.js'),

format: 'umd',

env: 'production',

banner

},

// Runtime+compiler development build (Browser)

'web-full-dev': {

entry: resolve('web/entry-runtime-with-compiler.js'),

dest: resolve('dist/vue.js'),

format: 'umd',

env: 'development',

alias: { he: './entity-decoder' },

banner

},

// Runtime+compiler production build (Browser)

'web-full-prod': {

entry: resolve('web/entry-runtime-with-compiler.js'),

dest: resolve('dist/vue.min.js'),

format: 'umd',

env: 'production',

alias: { he: './entity-decoder' },

banner

},

// ...

}

这里列举了一些 Vue.js 构建的配置，关于还有一些服务端渲染 webpack 插件以及 weex 的打包配置就不列举了。

对于单个配置，它是遵循 Rollup 的构建规则的。其中 entry 属性表示构建的入口 JS 文件地址，dest 属性表示构建后的 JS 文件地址。format 属性表示构建的格式，cjs 表示构建出来的文件遵循 [CommonJS](http://wiki.commonjs.org/wiki/Modules/1.1" \t "https://coding.imooc.com/lesson/_blank) 规范，es 表示构建出来的文件遵循 [ES Module](http://exploringjs.com/es6/ch_modules.html" \t "https://coding.imooc.com/lesson/_blank) 规范。 umd 表示构建出来的文件遵循 [UMD](https://github.com/umdjs/umd" \t "https://coding.imooc.com/lesson/_blank) 规范。

以 web-runtime-cjs 配置为例，它的 entry 是  
resolve('web/entry-runtime.js')，先来看一下 resolve 函数的定义。

源码目录：scripts/config.js

const aliases = require('./alias')

const resolve = p => {

const base = p.split('/')[0]

if (aliases[base]) {

return path.resolve(aliases[base], p.slice(base.length + 1))

} else {

return path.resolve(\_\_dirname, '../', p)

}

}

这里的 resolve 函数实现非常简单，它先把 resolve 函数传入的参数 p 通过 / 做了分割成数组，然后取数组第一个元素设置为 base。在我们这个例子中，参数 p 是 web/entry-runtime.js，那么 base 则为 web。base 并不是实际的路径，它的真实路径借助了别名的配置，我们来看一下别名配置的代码，在 scripts/alias 中：

const path = require('path')

module.exports = {

vue: path.resolve(\_\_dirname, '../src/platforms/web/entry-runtime-with-compiler'),

compiler: path.resolve(\_\_dirname, '../src/compiler'),

core: path.resolve(\_\_dirname, '../src/core'),

shared: path.resolve(\_\_dirname, '../src/shared'),

web: path.resolve(\_\_dirname, '../src/platforms/web'),

weex: path.resolve(\_\_dirname, '../src/platforms/weex'),

server: path.resolve(\_\_dirname, '../src/server'),

entries: path.resolve(\_\_dirname, '../src/entries'),

sfc: path.resolve(\_\_dirname, '../src/sfc')

}

很显然，这里 web 对应的真实的路径是 path.resolve(\_\_dirname, '../src/platforms/web')，这个路径就找到了 Vue.js 源码的 web 目录。然后 resolve 函数通过 path.resolve(aliases[base], p.slice(base.length + 1)) 找到了最终路径，它就是 Vue.js 源码 web 目录下的 entry-runtime.js。因此，web-runtime-cjs 配置对应的入口文件就找到了。

它经过 Rollup 的构建打包后，最终会在 dist 目录下生成 vue.runtime.common.js。

## **Runtime Only VS Runtime+Compiler**

通常我们利用 vue-cli 去初始化我们的 Vue.js 项目的时候会询问我们用 Runtime Only 版本的还是 Runtime+Compiler 版本。下面我们来对比这两个版本。

* Runtime Only

我们在使用 Runtime Only 版本的 Vue.js 的时候，通常需要借助如 webpack 的 vue-loader 工具把 .vue 文件编译成 JavaScript，因为是在编译阶段做的，所以它只包含运行时的 Vue.js 代码，因此代码体积也会更轻量。

* Runtime+Compiler

我们如果没有对代码做预编译，但又使用了 Vue 的 template 属性并传入一个字符串，则需要在客户端编译模板，如下所示：

// 需要编译器的版本

new Vue({

template: '<div>{{ hi }}</div>'

})

// 这种情况不需要

new Vue({

render (h) {

return h('div', this.hi)

}

})

因为在 Vue.js 2.0 中，最终渲染都是通过 render 函数，如果写 template 属性，则需要编译成 render 函数，那么这个编译过程会发生运行时，所以需要带有编译器的版本。

很显然，这个编译过程对性能会有一定损耗，所以通常我们更推荐使用 Runtime-Only 的 Vue.js。

## **总结**

通过这一节的分析，我们可以了解到 Vue.js 的构建打包过程，也知道了不同作用和功能的 Vue.js 它们对应的入口以及最终编译生成的 JS 文件。尽管在实际开发过程中我们会用 Runtime Only 版本开发比较多，但为了分析 Vue 的编译过程，我们这门课重点分析的源码是 Runtime+Compiler 的 Vue.js。

****任务****

请仔细阅读本文档，为下面学习视频内容做准备。