**从Element-ui源码聊搭建UI库**

目录

[一、 背景 3](#_Toc72941346)

[什么是组件库？ 3](#_Toc72941347)

[为什么需要组件库？ 3](#_Toc72941348)

[二、 目录结构 4](#_Toc72941349)

[三、 packages目录分析 6](#_Toc72941350)

[四、 package.json 8](#_Toc72941351)

[五、 scripts脚本命令解析 9](#_Toc72941352)

[bootstrap 9](#_Toc72941353)

[build:file 9](#_Toc72941354)

[node build/bin/iconInit.js 10](#_Toc72941355)

[node build/bin/build-entry.js 12](#_Toc72941356)

[node build/bin/i18n.js 15](#_Toc72941357)

[node build/bin/version.js 18](#_Toc72941358)

[build:theme 20](#_Toc72941359)

[node build/bin/gen-cssfile 20](#_Toc72941360)

[gulp build --gulpfile packages/theme-chalk/gulpfile.js 21](#_Toc72941361)

[cp-cli packages/theme-chalk/lib lib/theme-chalk 24](#_Toc72941362)

[build:utils 25](#_Toc72941363)

[build:umd 26](#_Toc72941364)

[clean 28](#_Toc72941365)

[deploy:build 28](#_Toc72941366)

[cross-env NODE\_ENV=production webpack --config build/webpack.demo.js 28](#_Toc72941367)

[deploy:extension 33](#_Toc72941368)

[dev:extension 33](#_Toc72941369)

[dev 33](#_Toc72941370)

[dev:play 34](#_Toc72941371)

[dist 35](#_Toc72941372)

[webpack --config build/webpack.conf.js 35](#_Toc72941373)

[webpack --config build/webpack.common.js 38](#_Toc72941374)

[webpack --config build/webpack.component.js 38](#_Toc72941375)

[lint 39](#_Toc72941376)

[pub 39](#_Toc72941377)

[test 43](#_Toc72941378)

[六、 makefile 43](#_Toc72941379)

[make new 45](#_Toc72941380)

[make new-lang 51](#_Toc72941381)

[七、 其他工程化脚本 52](#_Toc72941382)

[md-loader 52](#_Toc72941383)

[config.js 54](#_Toc72941384)

正文

# 背景

组件库是组里一直想要搞的玩意，但是迫于业务繁忙，一直往后拖。如果真的需要做组件库，那么下面的两个问题肯定多少是有答案的。

## 什么是组件库？

简单地说，组件库就是一套详细的设计样式控件库，他把交互规范、视觉规范、前端代码融合到一起。我们看到的每一个组件，背后都对应实现的代码。在我们做项目的时候，只要在组件库中找到对应的组件就可以直接使用，非常方便。

## 为什么需要组件库？

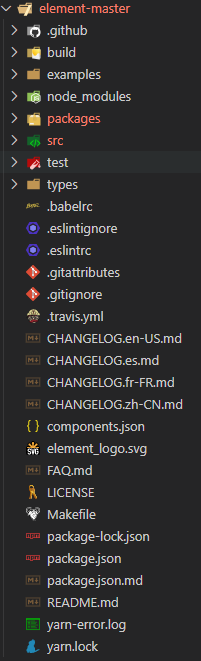
1. 组里大大小小的项目有20多个，风格各异，没有统一。每个项目的交互也各不相同。用户们就需要适应20多个项目的交互，但是这样大大降低了用户使用的热情。
2. 在开发的时候，产品经理为了节省设计成本，可能会口头说一下需求。没有统一的UI库，我们完成的组件可能没有达到经理的要求。

总而言之，磨刀不误砍柴工，UI组件库可以大大加快我们组里的开发效率，所以搞一套基于vue的UI组件库迫在眉睫。而业界已经有比较成熟的一些UI库了，比如Element、AntDesign等。但是我们的设计往往喜欢不一样的风格和功能，所以业界的UI库不能完全满足。

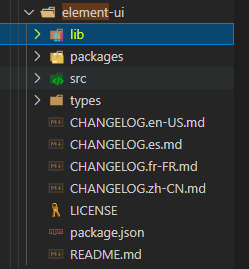
结合框架Vue，我们选择在Element基础上进行二次开发。保留其架构、文档、工程化和基础组件，对其中组件进行删减和增加。但造轮子绝非易事，首先需要先去下载源码，了解它整个工程化、官网构建、组件库、测试和组件类型声明等。本文通过分析Element完整的项目架构，为搭建我们组里的组件库做一些准备工作。

# 目录结构

首先我们来看下未打包目录：



再来看下打包后的目录：



那么，每个文件的作用是什么呢？

1. build：webpack相关的打包配置文件
2. examples：官网的那些组件demo，构成官网的项目架构
3. packages：每个组件源码
4. src：公共方法和入口文件
5. test：测试文件, 使用 karma 框架
6. types：类型声明文件, 每个组件的类型声明文件，TS 项目使用组件库时有更好的代码提示
7. .babelrc：babel的配置文件
8. .eslintignore：忽略eslint的配置文件
9. .eslintrc：eslint配置文件
10. .gitattributes：允许我们指定由git使用的文件和路径的属性
11. .gitignore：忽略git的配置文件
12. .npmignore：上传npm的配置文件
13. .travis.yml：持续集成ci的配置文件
14. CHANGELOG：更新日志，提供了四种语言
15. components.json：表明了组件的文件路径，方便webpack打包时获取组件的文件路径
16. FAQ.md：常见的组件库问题
17. LICENSE：mit协议
18. Makefile：是一个适用于c/c++的工具，make环境下，输入make命令将会执行makefile文件中的某个目标命令
19. package-lock.json：锁定安装时的包的版本号
20. package.json：安装信息

在打包编译后，很多文件都消失殆尽，不仅缩小了组件库的体积，而且满足了我们项目的需求。

比如少了构成官网的examples文件目录、做单元测试的test文件目录等，多了核心库lib文件目录。

# packages目录分析

从上面我们知道了packages是UI库中的组件源码所在的目录，所有组件都放于其中。其中分为组件目录和theme-chalk主题样式目录。

组件目录通常由index.js和src组成，src下的vue文件是生成组件的逻辑代码，index.js文件是组件的注册和导出。如果有README.md文件，一般就是该组件的解释说明。

theme-chalk主题样式目录包含了所有组件的样式，用sass编写的。其中src\common目录下的var.scss文件包含了所有颜色、边框以及单个组件抽取出来的的样式变量。比如我想修改dialog的样式，我可以在dialog.scss中对其修改调整，如果要修改涉及到全局变量，那么就在var.scss中定义。

比如输入框input组件：

element-master\packages\input\index.js

注册input

import ElInput from './src/input';

/\* istanbul ignore next \*/

ElInput.install = function(Vue) {

  Vue.component(ElInput.name, ElInput);

};

export default ElInput;

element-master\packages\input\src\input.vue

实现input的vue组件：

<template>

  <div :class="[

    type === 'textarea' ? 'el-textarea' : 'el-input',

    inputSize ? 'el-input--' + inputSize : '',

    {

      'is-disabled': inputDisabled,

      'is-exceed': inputExceed,

      'el-input-group': $slots.prepend || $slots.append,

      'el-input-group--append': $slots.append,

      'el-input-group--prepend': $slots.prepend,

      'el-input--prefix': $slots.prefix || prefixIcon,

      'el-input--suffix': $slots.suffix || suffixIcon || clearable || showPassword

    }

    ]"

    @mouseenter="hovering = true"

    @mouseleave="hovering = false"

  >

  ……此处省略input相关html

    <span v-if="isWordLimitVisible && type === 'textarea'" class="el-input\_\_count">{{ textLength }}/{{ upperLimit }}</span>

  </div>

</template>

<script>

  export default {

    name: 'ElInput',

    componentName: 'ElInput',

    ……此处省略input相关逻辑

  };

</script>

element-master\packages\theme-chalk\src\input.scss

输入框的样式，如果我们是基于其二次开发，改动最多的就是样式了：

@import "mixins/mixins";

@import "common/var";

@include b(textarea) {

 ……此处省略多行文本输入框的样式

}

@include b(input) {

  ……此处省略input的样式

}

@include b(input-group) {

  ……此处省略输入框组的样式

}

/\*\* disalbe default clear on IE \*/

.el-input\_\_inner::-ms-clear {

  display: none;

  width: 0;

  height: 0;

}

# package.json

package.json是项目的解释性文档。属性描述如下：

1. name：项目名称，不能重复
2. version：版本号
3. description：项目描述
4. main：项目主要入口点
5. files：被项目包含的文件名数组（除非是被规则忽略的文件）
6. scripts：项目脚本命令
7. repository：指明代码仓库
8. homepage：项目主页url
9. keywords：项目关键词，有助于其他人发现包
10. license：指定license来让用户知道他们的使用权力和任务限制
11. bugs：项目的issue地址
12. dependencies：指定了生产环境的依赖的包名和其版本范围的映射，不要将测试或过渡性的依赖放入其中
13. peerDependencies：当主机无法导入依赖包时，指明哪些库与这个依赖包兼容
14. devDependencies：指定了开发环境的依赖的包名和其版本范围的映射

# scripts脚本命令解析

其中最重要的scripts对象的中脚本命令需要一一描述：

## bootstrap

"bootstrap": "yarn || npm i"：安装依赖

## build:file

"build:file": "node build/bin/iconInit.js & node build/bin/build-entry.js & node build/bin/i18n.js & node build/bin/version.js"

这个命令自动构建一些文件，让我们拆开来一一剖析：

### node build/bin/iconInit.js

将icon.scss文件中的icon抽离出来放入icon.json文件

var fontFile = fs.readFileSync(path.resolve(\_\_dirname, '../../packages/theme-chalk/src/icon.scss'), 'utf8');

var nodes = postcss.parse(fontFile).nodes;

var classList = [];

nodes.forEach((node) => {

  var selector = node.selector || '';

  var reg = new RegExp(/\.el-icon-([^:]+):before/);

  var arr = selector.match(reg);

  if (arr && arr[1]) {

    classList.push(arr[1]);

  }

});

classList.reverse(); // 希望按 css 文件顺序倒序排列

fs.writeFile(path.resolve(\_\_dirname, '../../examples/icon.json'), JSON.stringify(classList), () => {});

如将platform=eleme放入icon.json的数组中：

element-master\packages\theme-chalk\src\header.scss

.el-icon-platform-eleme:before {

  content: "\e7ca";

}

element-master\examples\icon.json

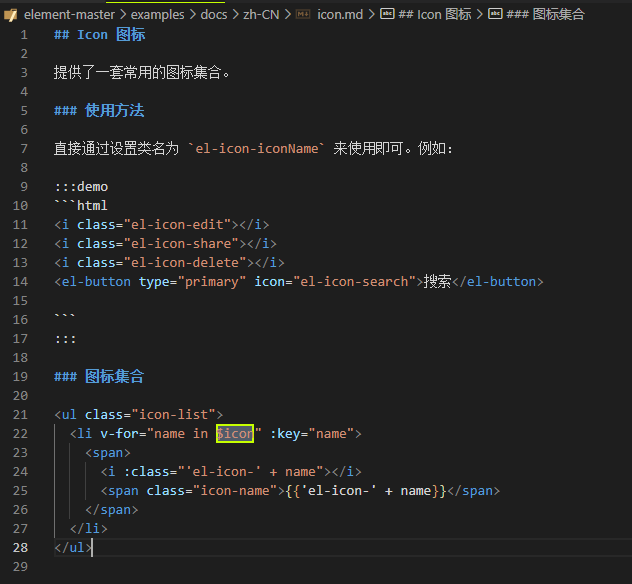
"platform-eleme"

生成icon.json文件，在entry.js中引入，并且绑定在vue原型上

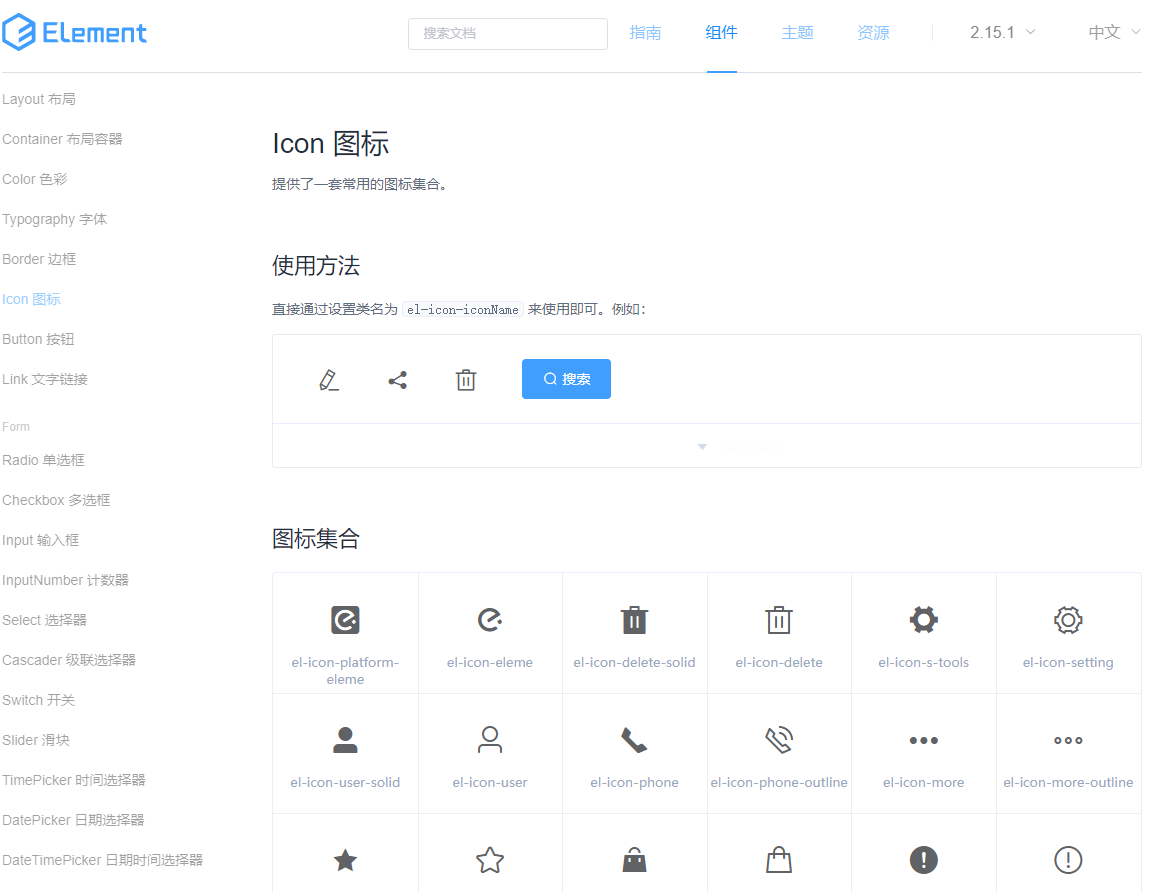
import icon from './icon.json';

Vue.prototype.$icon = icon; // Icon 列表页用

应用在文档中element-master\examples\docs\zh-CN\icon.md：



通过之后说到的md-loader解析成vue组件，渲染在官网上



### node build/bin/build-entry.js

执行后自动生成element-master\src\index.js，也就是项目的入口文件，用json-templater结合变量生成模板，避免每次新增组件时手动在src\index.js中引入并导出组件。如何自动新增组件后文也会提及。

// 导入组件列表

var Components = require('../../components.json');

var fs = require('fs');

// 替换方法

var render = require('json-templater/string');

// 将破折号/点/下划线/空格分隔的字符串转换为UpperCamelCase:foo-bar → FooBar

var uppercamelcase = require('uppercamelcase');

var path = require('path');

// 行尾

var endOfLine = require('os').EOL;

// 输出文件路径

var OUTPUT\_PATH = path.join(\_\_dirname, '../../src/index.js');

// 导入模板

var IMPORT\_TEMPLATE = 'import {{name}} from \'../packages/{{package}}/index.js\';';

// 安装组件模板

var INSTALL\_COMPONENT\_TEMPLATE = '  {{name}}';

// 主模板 过渡动画CollapseTransition纯js，特殊，不注册在packages目录下

var MAIN\_TEMPLATE = `/\* Automatically generated by './build/bin/build-entry.js' \*/

{{include}}

import locale from 'element-ui/src/locale';

import CollapseTransition from 'element-ui/src/transitions/collapse-transition';

const components = [

{{install}},

  CollapseTransition

];

const install = function(Vue, opts = {}) {

  locale.use(opts.locale);

  locale.i18n(opts.i18n);

  components.forEach(component => {

    Vue.component(component.name, component);

  });

  Vue.use(InfiniteScroll);

  Vue.use(Loading.directive);

  Vue.prototype.$ELEMENT = {

    size: opts.size || '',

    zIndex: opts.zIndex || 2000

  };

  Vue.prototype.$loading = Loading.service;

  Vue.prototype.$msgbox = MessageBox;

  Vue.prototype.$alert = MessageBox.alert;

  Vue.prototype.$confirm = MessageBox.confirm;

  Vue.prototype.$prompt = MessageBox.prompt;

  Vue.prototype.$notify = Notification;

  Vue.prototype.$message = Message;

};

/\* istanbul ignore if \*/

if (typeof window !== 'undefined' && window.Vue) {

  install(window.Vue);

}

export default {

  version: '{{version}}',

  locale: locale.use,

  i18n: locale.i18n,

  install,

  CollapseTransition,

  Loading,

{{list}}

};

`;

delete Components.font;

// 获取组件的所有键放入数组中

var ComponentNames = Object.keys(Components);

// 包含组件模板

var includeComponentTemplate = [];

// 安装模板

var installTemplate = [];

// 列表模板

var listTemplate = [];

ComponentNames.forEach(name => {

  var componentName = uppercamelcase(name);

  includeComponentTemplate.push(render(IMPORT\_TEMPLATE, {

    name: componentName,

    package: name

  }));

  // 输出如下

  //  "import Popconfirm from '../packages/popconfirm/index.js';"

  if (['Loading', 'MessageBox', 'Notification', 'Message', 'InfiniteScroll'].indexOf(componentName) === -1) {

    installTemplate.push(render(INSTALL\_COMPONENT\_TEMPLATE, {

      name: componentName,

      component: name

    }));

  }

  // 输出如下

  // '  Pagination',     '  Dialog',

  if (componentName !== 'Loading') listTemplate.push(`  ${componentName}`);

  // 输出如下

  // '  Pagination',     '  Dialog',

});

// 替换主模板中的变量 include、install、version、list

var template = render(MAIN\_TEMPLATE, {

  include: includeComponentTemplate.join(endOfLine),

  install: installTemplate.join(',' + endOfLine),

  version: process.env.VERSION || require('../../package.json').version,

  list: listTemplate.join(',' + endOfLine)

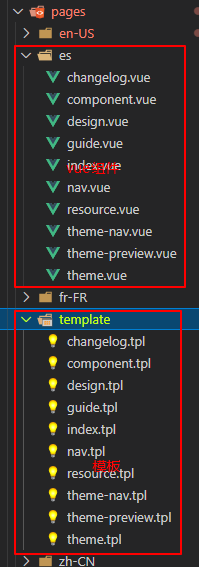
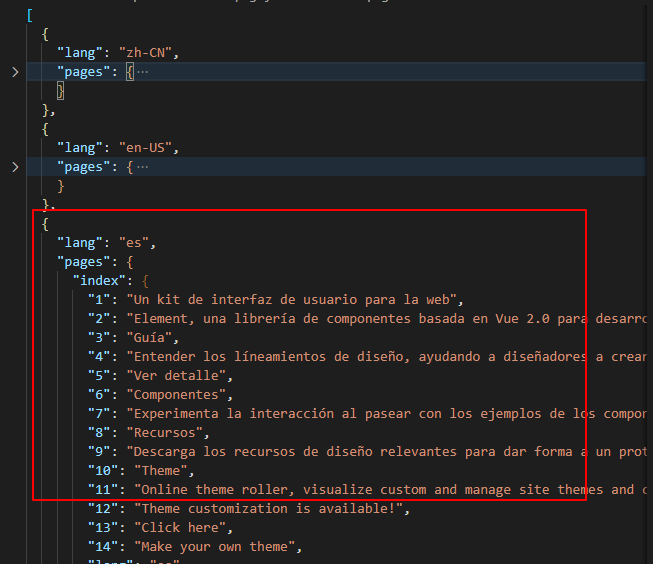
});

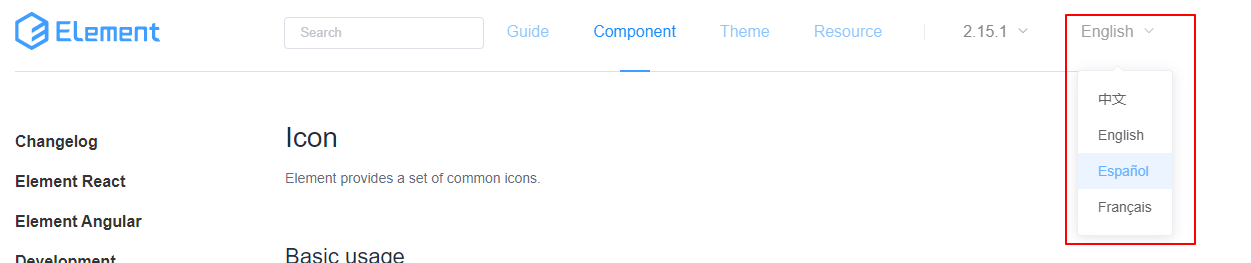
fs.writeFileSync(OUTPUT\_PATH, template);

console.log('[build entry] DONE:', OUTPUT\_PATH);

### node build/bin/i18n.js

根据模板创建各个语言下的组件，只需要维护page.json文件，执行命令自动生成官网上的语言组件





来看下具体实现：

'use strict';

var fs = require('fs');

var path = require('path');

// 从examples/i18n/page.json获取国际化的配置

var langConfig = require('../../examples/i18n/page.json');

// 遍历配置

langConfig.forEach(lang => {

  // 创建 zh-CN 等在内的文件夹

  try {

    fs.statSync(path.resolve(\_\_dirname, `../../examples/pages/${ lang.lang }`));

  } catch (e) {

    fs.mkdirSync(path.resolve(\_\_dirname, `../../examples/pages/${ lang.lang }`));

  }

  Object.keys(lang.pages).forEach(page => {

    // 按照键创建模板下面的tpl

    var templatePath = path.resolve(\_\_dirname, `../../examples/pages/template/${ page }.tpl`);

    // 按照键创建对应语言文件夹下的vue文件

    var outputPath = path.resolve(\_\_dirname, `../../examples/pages/${ lang.lang }/${ page }.vue`);

    var content = fs.readFileSync(templatePath, 'utf8');

    var pairs = lang.pages[page];

    // 通过tpl模板获取内容

    Object.keys(pairs).forEach(key => {

      content = content.replace(new RegExp(`<%=\\s\*${ key }\\s\*>`, 'g'), pairs[key]);

    });

    // 往各个语言文件夹下的vue文件写入内容

    fs.writeFileSync(outputPath, content);

  });

});

### node build/bin/version.js

根据 /package.json 文件，自动生成 /examples/version.json，用于记录组件库的版本信息，这些版本会渲染在官网组件页面的头部导航栏。



来看下具体实现：

var fs = require('fs');

var path = require('path');

var version = process.env.VERSION || require('../../package.json').version;

var content = {

    '1.4.13': '1.4',

    '2.0.11': '2.0',

    '2.1.0': '2.1',

    '2.2.2': '2.2',

    '2.3.9': '2.3',

    '2.4.11': '2.4',

    '2.5.4': '2.5',

    '2.6.3': '2.6',

    '2.7.2': '2.7',

    '2.8.2': '2.8',

    '2.9.2': '2.9',

    '2.10.1': '2.10',

    '2.11.1': '2.11',

    '2.12.0': '2.12'

};

if (!content[version]) content[version] = '2.13';

// 往examples/versions.json写入所有版本

fs.writeFileSync(path.resolve(\_\_dirname, '../../examples/versions.json'), JSON.stringify(content));

examples\components\header.vue

   created() {

      const xhr = new XMLHttpRequest();

      xhr.onreadystatechange = \_ => {

        if (xhr.readyState === 4 && xhr.status === 200) {

          const versions = JSON.parse(xhr.responseText);

          this.versions = Object.keys(versions).reduce((prev, next) => {

            prev[next] = versions[next];

            return prev;

          }, {});

        }

      };

      xhr.open('GET', '/versions.json');

      xhr.send();

      let primaryLast = '#409EFF';

      bus.$on(ACTION\_USER\_CONFIG\_UPDATE, (val) => {

        let primaryColor = val.global['$--color-primary'];

        if (!primaryColor) primaryColor = '#409EFF';

        const base64svg = 'data:image/svg+xml;base64,';

        const imgSet = document.querySelectorAll('h1 img');

        imgSet.forEach((img) => {

          img.src = `${base64svg}${window.btoa(window.atob(img.src.replace(base64svg, '')).replace(primaryLast, primaryColor))}`;

        });

        primaryLast = primaryColor;

      });

    }

至于其中的逻辑之后我们细讲。

## build:theme

"build:theme": "node build/bin/gen-cssfile && gulp build --gulpfile packages/theme-chalk/gulpfile.js && cp-cli packages/theme-chalk/lib lib/theme-chalk"：

主要是处理样式相关的脚本。拆开来分析下：

### node build/bin/gen-cssfile

执行该文件通过组件列表生成对应的css文件和theme-chalk/index.scss文件，并将所有组件的样式都导入。以后每次新增一个组件不用手动导入，执行命令自动导入。

比如创建了新组件aaa，执行命令后生成aaa.scss，并且index.scss多了aaa。

来看下具体实现：

var fs = require('fs');

var path = require('path');

var Components = require('../../components.json');

var themes = [

  'theme-chalk'

];

Components = Object.keys(Components);

var basepath = path.resolve(\_\_dirname, '../../packages/');

function fileExists(filePath) {

  try {

    return fs.statSync(filePath).isFile();

  } catch (err) {

    return false;

  }

}

themes.forEach((theme) => {

  var isSCSS = theme !== 'theme-default';

  var indexContent = isSCSS ? '@import "./base.scss";\n' : '@import "./base.css";\n';

  Components.forEach(function(key) {

    // 导入的组件不在packages文件夹下,需要过滤

    // 以下是option-group的代码

//     import ElOptionGroup from '../select/src/option-group';

// /\* istanbul ignore next \*/

// ElOptionGroup.install = function(Vue) {

  //   Vue.component(ElOptionGroup.name, ElOptionGroup);

  // };

  // export default ElOptionGroup;

  if (['icon', 'option', 'option-group'].indexOf(key) > -1) return;

  var fileName = key + (isSCSS ? '.scss' : '.css');

  // @import "./popconfirm.scss";

  indexContent += '@import "./' + fileName + '";\n';

  // 组装组件css文件路径 E:\element-master\element-master\packages\theme-chalk\src\popconfirm.scss

  var filePath = path.resolve(basepath, theme, 'src', fileName);

  // 文件不存在就创建遗漏的css文件

  if (!fileExists(filePath)) {

      fs.writeFileSync(filePath, '', 'utf8');

      console.log(theme, ' 创建遗漏的 ', fileName, ' 文件');

    }

  });

  // 往index.scss中写入导入 如:@import "./base.scss";

  fs.writeFileSync(path.resolve(basepath, theme, 'src', isSCSS ? 'index.scss' : 'index.css'), indexContent);

});

### gulp build --gulpfile packages/theme-chalk/gulpfile.js

把所有的scss文件通过gulp编译成css。

打包和压缩的工作平时一般交给webpack来做，但是这一次用gulp更加快捷和方便，基于工作流。

'use strict';

const {

  series,

  src,

  dest

} = require('gulp');

// 编译gulp工具

const sass = require('gulp-sass');

// 添加厂商前缀

const autoprefixer = require('gulp-autoprefixer');

// 压缩css

const cssmin = require('gulp-cssmin');

// src下面的所有文件编译到lib下

function compile() {

  return src('./src/\*.scss')

    .pipe(sass.sync()) //把scss编译成css

    .pipe(autoprefixer({ //基于目标浏览器版本,添加厂商前缀

      browsers: ['ie > 9', 'last 2 versions'],

      cascade: false

    }))

    .pipe(cssmin()) //压缩css

    // dest: 流会将 vinyl File保存到指定目录下

    .pipe(dest('./lib')); //输出到lib下

}

// 读取src下的fonts文件目录输出到lib下

function copyfont() {

  return src('./src/fonts/\*\*')

    .pipe(cssmin())

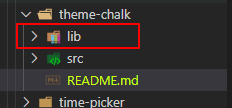
    .pipe(dest('./lib/fonts'));

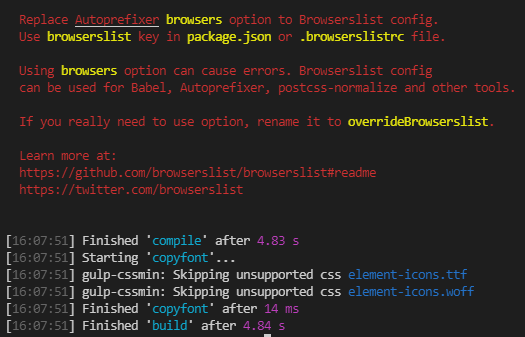
}

// series: 接受可变数量的字符串(taskName)和/或函数(fn)，并返回组合任务或函数的一个函数

exports.build = series(compile, copyfont);

执行命令输出css文件，放入lib目录：





为什么需要编译呢？

因为element在使用时有两种引入方式：

全局引用：

import Vue from 'vue';

import ElementUI from 'element-ui';

import 'element-ui/lib/theme-chalk/index.css';

import App from './App.vue';

Vue.use(ElementUI);

new Vue({

  el: '#app',

  render: h => h(App)

});

引入了lib\theme-chalk\index.css文件

局部引入：

import Vue from 'vue';

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

import App from './App.vue';

Vue.component(Button.name, Button);

Vue.component(Select.name, Select);

/\* 或写为

 \* Vue.use(Button)

 \* Vue.use(Select)

 \*/

new Vue({

  el: '#app',

  render: h => h(App)

});

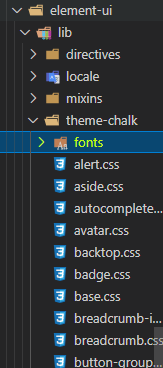
不需要引入css文件，只需引入对应的scss文件。

这就是为什么需要编译scss的原因。

### cp-cli packages/theme-chalk/lib lib/theme-chalk

cp-cli 是一个跨平台的copy工具

将gulp build --gulpfile .\packages\theme-chalk\gulpfile.js编译生成的css目录（packages/theme-chalk/lib）复制到lib/theme-chalk下



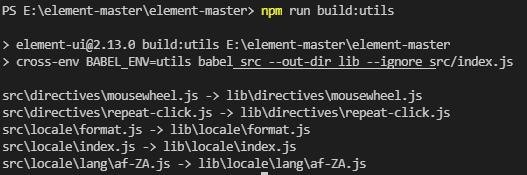
方便全局引用，导入css

import 'element-ui/lib/theme-chalk/index.css';

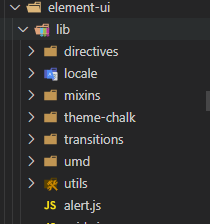
## build:utils

"build:utils": "cross-env BABEL\_ENV=utils babel src --out-dir lib --ignore src/index.js"

将工具函数通过babel转译后移动到lib下，方便我们使用。

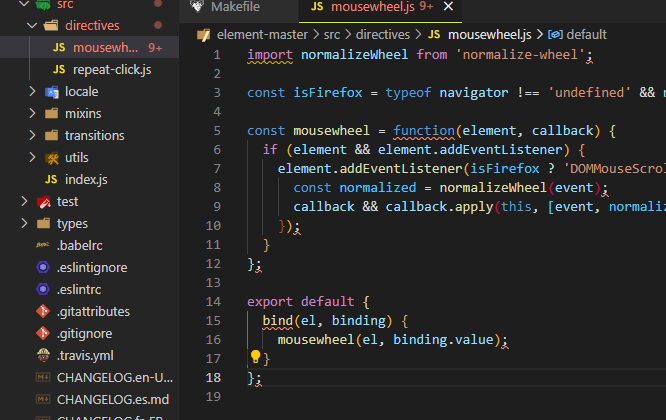


将src目录下的内容忽略index.js通过babel转译，然后移动到lib下

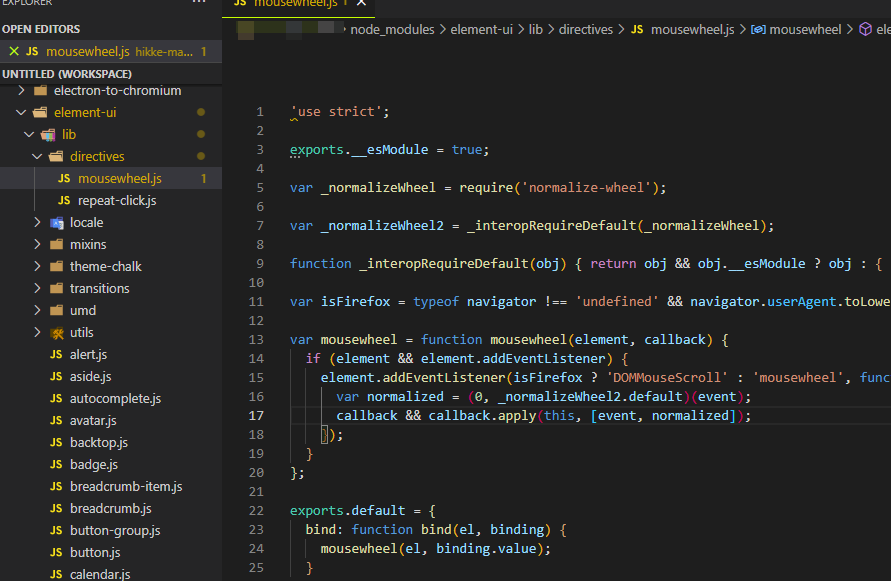


比如mousewheel文件

没转译的：



转译后的：



## build:umd

"build:umd": "node build/bin/build-locale.js",

执行后生成umd模块的语言包。

将src/locale/lang下的语言包都编译到lib/umd/locale下。

var fs = require('fs');

// 导出文件

var save = require('file-save');

// 解析为绝对路径

var resolve = require('path').resolve;

// 获取扩展名,返回path最后一部分 path.basename('/foo/bar/quux.html', '.html'); // 返回：‘quux’

var basename = require('path').basename;

var localePath = resolve(\_\_dirname, '../../src/locale/lang');

// 读取src/locale/lang下的列表

var fileList = fs.readdirSync(localePath);

// 通过babel转译

var transform = function (filename, name, cb) {

  // https://babel.docschina.org/docs/en/6.26.3/babel-core/

  require('babel-core').transformFile(resolve(localePath, filename), {

    plugins: [

      'add-module-exports',

      ['transform-es2015-modules-umd', {

        loose: true

      }]

    ],

    moduleId: name

  }, cb);

};

fileList

  // 过滤js文件

  .filter(function (file) {

    return /\.js$/.test(file);

  })

  .forEach(function (file) {

    var name = basename(file, '.js');

    // 异步转译文件中的全部内容

    transform(file, name, function (err, result) {

      if (err) {

        console.error(err);

      } else {

        var code = result.code;

        code = code

          .replace('define(\'', 'define(\'element/locale/')

          .replace('global.', 'global.ELEMENT.lang = global.ELEMENT.lang || {}; \n    global.ELEMENT.lang.');

        save(resolve(\_\_dirname, '../../lib/umd/locale', file)).write(code);

        console.log(file);

      }

    });

  });

## clean

"clean": "rimraf lib && rimraf packages/\*/lib && rimraf test/\*\*/coverage"

清除打包好后的文件

## deploy:build

"deploy:build": "npm run build:file && cross-env NODE\_ENV=production webpack --config build/webpack.demo.js && echo element.eleme.io>>examples/element-ui/CNAME"

npm run build:file前文分析过了，主要构建官网文件。接下来分析新的构建脚本。

### cross-env NODE\_ENV=production webpack --config build/webpack.demo.js

生产环境下构建官网。

官网项目的 webpack 配置。

// webpack第三方插件

// https://github.com/webpack-contrib

const path = require('path');

const webpack = require('webpack');

// 这个插件将CSS提取到单独的文件中。它为每个包含CSS的JS文件创建一个CSS文件。它支持按需加载CSS和SourceMaps。

const MiniCssExtractPlugin = require('mini-css-extract-plugin');

const CopyWebpackPlugin = require('copy-webpack-plugin');

const HtmlWebpackPlugin = require('html-webpack-plugin');

const ProgressBarPlugin = require('progress-bar-webpack-plugin');

const VueLoaderPlugin = require('vue-loader/lib/plugin');

const OptimizeCSSAssetsPlugin = require('optimize-css-assets-webpack-plugin');

const UglifyJsPlugin = require('uglifyjs-webpack-plugin');

const config = require('./config');

const isProd = process.env.NODE\_ENV === 'production';

// "dev:play": "npm run build:file && cross-env NODE\_ENV=development PLAY\_ENV=true webpack-dev-server --config build/webpack.demo.js",

const isPlay = !!process.env.PLAY\_ENV;

const webpackConfig = {

  mode: process.env.NODE\_ENV,

  entry: isProd ? {

    docs: './examples/entry.js'

  } : (isPlay ? './examples/play.js' : './examples/entry.js'),

  output: {

    path: path.resolve(process.cwd(), './examples/element-ui/'),

    publicPath: process.env.CI\_ENV || '',

    filename: '[name].[hash:7].js',

    chunkFilename: isProd ? '[name].[hash:7].js' : '[name].js'

  },

  resolve: {

    extensions: ['.js', '.vue', '.json'],

    alias: config.alias,

    modules: ['node\_modules']

  },

  // 开发服务器

  // webpack-dev-server 可用于快速开发应用程序

  devServer: {

    host: '0.0.0.0',

    port: 8085,

    publicPath: '/',

    hot: true

  },

  performance: {

    hints: false

  },

  stats: {

    children: false

  },

  module: {

    rules: [{

        enforce: 'pre',

        test: /\.(vue|jsx?)$/,

        exclude: /node\_modules/,

        loader: 'eslint-loader'

      },

      {

        test: /\.(jsx?|babel|es6)$/,

        include: process.cwd(),

        exclude: config.jsexclude,

        loader: 'babel-loader'

      },

      {

        test: /\.vue$/,

        loader: 'vue-loader',

        options: {

          compilerOptions: {

            preserveWhitespace: false

          }

        }

      },

      {

        test: /\.(scss|css)$/,

        use: [

          isProd ? MiniCssExtractPlugin.loader : 'style-loader',

          'css-loader',

          'sass-loader'

        ]

      },

      // 使用md-loader加载模块

      {

        test: /\.md$/,

        use: [{

            loader: 'vue-loader',

            options: {

              compilerOptions: {

                preserveWhitespace: false

              }

            }

          },

          {

            loader: path.resolve(\_\_dirname, './md-loader/index.js')

          }

        ]

      },

      {

        test: /\.(svg|otf|ttf|woff2?|eot|gif|png|jpe?g)(\?\S\*)?$/,

        loader: 'url-loader',

        // todo: 这种写法有待调整

        query: {

          limit: 10000,

          name: path.posix.join('static', '[name].[hash:7].[ext]')

        }

      }

    ]

  },

  plugins: [

    new webpack.HotModuleReplacementPlugin(),

    new HtmlWebpackPlugin({

      template: './examples/index.tpl',

      filename: './index.html',

      favicon: './examples/favicon.ico'

    }),

    new CopyWebpackPlugin([{

      from: 'examples/versions.json'

    }]),

    new ProgressBarPlugin(),

    new VueLoaderPlugin(),

    new webpack.DefinePlugin({

      'process.env.FAAS\_ENV': JSON.stringify(process.env.FAAS\_ENV)

    }),

    new webpack.LoaderOptionsPlugin({

      vue: {

        compilerOptions: {

          preserveWhitespace: false

        }

      }

    })

  ],

  optimization: {

    minimizer: []

  },

  devtool: '#eval-source-map'

};

// 生产环境更换插件\优化\扩展等

if (isProd) {

  webpackConfig.externals = {

    vue: 'Vue',

    'vue-router': 'VueRouter',

    'highlight.js': 'hljs'

  };

  webpackConfig.plugins.push(

    new MiniCssExtractPlugin({

      filename: '[name].[contenthash:7].css'

    })

  );

  webpackConfig.optimization.minimizer.push(

    new UglifyJsPlugin({

      cache: true,

      parallel: true,

      sourceMap: false

    }),

    new OptimizeCSSAssetsPlugin({})

  );

  // https://webpack.js.org/configuration/optimization/#optimizationsplitchunks

  webpackConfig.optimization.splitChunks = {

    cacheGroups: {

      vendor: {

        test: /\/src\//,

        name: 'element-ui',

        chunks: 'all'

      }

    }

  };

  webpackConfig.devtool = false;

}

module.exports = webpackConfig;

## deploy:extension

"deploy:extension": "cross-env NODE\_ENV=production webpack --config build/webpack.extension.js"

在生产环境下构建主题插件，主题编辑器的 chorme 插件项目的 webpack 配置，项目在 extension 目录下。执行命令后会在extension目录下生成dist目录，其中包含了chorme插件，在浏览器加载已解压的扩展程序就可以使用主题生成插件。

## dev:extension

"dev:extension": "rimraf examples/extension/dist && cross-env NODE\_ENV=development webpack --watch --config build/webpack.extension.js"

启动主题插件的开发环境，可以进行开发调试。

## dev

"dev": "npm run bootstrap && npm run build:file && cross-env NODE\_ENV=development webpack-dev-server --config build/webpack.demo.js & node build/bin/template.js"

首先npm run bootstrap是用来安装依赖的。npm run build:file在前面也有提到，主要用来自动化生成一些文件。主要是node build/bin/build-entry.js，用于生成Element的入口js：先是读取根目录的components.json，这个json文件维护着Element所有的组件路径映射关系，键为组件名，值为组件源码的入口文件；然后遍历键值，将所有组件进行import，对外暴露install方法，把所有import的组件通过Vue.component(name, component)方式注册为全局组件，并且把一些弹窗类的组件挂载到Vue的原型链上。

在生成了入口文件的src/index.js之后就会运行webpack-dev-server。

启动组件库本地开发环境。在更改后可以热更新官网。

具体webpack配置见webpack.demo.js。

## dev:play

"dev:play": "npm run build:file && cross-env NODE\_ENV=development PLAY\_ENV=true webpack-dev-server --config build/webpack.demo.js"

组件测试项目，在 examples/play/index.vue 中可以引入组件库任意组件，也可以直接使用 dev 启动的项目，在文档中使用组件。

用于查看某个组件的效果。适用于组件按需加载的显示效果。在webpack.demo.js通过环境变量配置输入。

element-master\build\webpack.demo.js

const isPlay = !!process.env.PLAY\_ENV;

……省略webpack具体配置

 entry: isProd ? {

    docs: './examples/entry.js'

  } : (isPlay ? './examples/play.js' : './examples/entry.js'),

element-master\examples\play.js

import Vue from 'vue';

import Element from 'main/index.js';

import App from './play/index.vue';

import 'packages/theme-chalk/src/index.scss';

Vue.use(Element);

new Vue({ // eslint-disable-line

  render: h => h(App)

}).$mount('#app');

element-master\examples\play\index.vue

<template>

  <div style="margin: 20px;">

    <el-input v-model="input" placeholder="请输入内容"></el-input>

  </div>

</template>

<script>

  export default {

    data() {

      return {

        input: 'Hello Element UI!'

      };

    }

  };

</script>

## dist

"dist": "npm run clean && npm run build:file && npm run lint && webpack --config build/webpack.conf.js && webpack --config build/webpack.common.js && webpack --config build/webpack.component.js && npm run build:utils && npm run build:umd && npm run build:theme"

npm run clean && npm run build:file && npm run lint都已经解释过了，分别是清除上一次打包产物、生成入口文件以及i18n文件和eslint检测

### webpack --config build/webpack.conf.js

生成umd格式的js文件（index.js）

const path = require('path');

const ProgressBarPlugin = require('progress-bar-webpack-plugin');

const VueLoaderPlugin = require('vue-loader/lib/plugin');

const TerserPlugin = require('terser-webpack-plugin');

const config = require('./config');

console.log(config)

module.exports = {

  // 模式

  mode: 'production',

  // 入口

  entry: {

    app: ['./src/index.js']

  },

  // 输出

  output: {

    path: path.resolve(process.cwd(), './lib'),

    publicPath: '/dist/',

    // 输出的文件名

    filename: 'index.js',

    // 初始的chunk文件名称

    chunkFilename: '[id].js',

    //  library 暴露为 AMD 模块。 在 AMD 或 CommonJS 的 require 之后可访问（libraryTarget:'umd'）

    libraryTarget: 'umd',

    // 入口的默认导出将分配给 library target：

    // if your entry has a default export of `MyDefaultModule`

    // var MyDefaultModule = \_entry\_return\_.default;

    libraryExport: 'default',

    // 输出一个库，为你的入口做导出。

    library: 'ELEMENT',

    // 会把 AMD 模块命名为 UMD 构建

    umdNamedDefine: true,

    // 为了使 UMD 构建在浏览器和 Node.js 上均可用，应将 output.globalObject 选项设置为 'this'。对于类似 web 的目标，默认为 self。

    globalObject: 'typeof self !== \'undefined\' ? self : this'

  },

  // 解析

  resolve: {

    // 能够使用户在引入模块时不带扩展.尝试按顺序解析这些后缀名。如果有多个文件有相同的名字，但后缀名不同，webpack 会解析列在数组首位的后缀的文件 并跳过其余的后缀。

    extensions: ['.js', '.vue', '.json'],

    // 创建 import 或 require 的别名，来确保模块引入变得更简单。

    alias: config.alias

  },

  // 外部扩展

  externals: {

    vue: config.vue

  },

  // 优化

  optimization: {

    // 允许你通过提供一个或多个定制过的 TerserPlugin 实例， 覆盖默认压缩工具(minimizer)

    minimizer: [

      new TerserPlugin({

        terserOptions: {

          output: {

            comments: false

          }

        }

      })

    ]

  },

  // 性能

  performance: {

    // 不展示警告或错误提示。

    // 官网推荐使用error,有助于防止把体积大的bundle部署到生产环境,从而影响网页的性能

    // 很奇怪这里要把它关闭

    hints: false

  },

  // stats对象

  stats: {

    // 告知 stats 是否添加关于子模块的信息。

    children: false

  },

  // 模块

  module: {

    // 使用babel-loader和vue-loader

    rules: [

      {

        test: /\.(jsx?|babel|es6)$/,

        include: process.cwd(),

        exclude: config.jsexclude,

        loader: 'babel-loader'

      },

      {

        test: /\.vue$/,

        loader: 'vue-loader',

        options: {

          compilerOptions: {

            preserveWhitespace: false

          }

        }

      }

    ]

  },

  // 插件

  plugins: [

    new ProgressBarPlugin(),

    new VueLoaderPlugin()

  ]

};

### webpack --config build/webpack.common.js

生成commonjs格式的js文件（element-ui.common.js），require时默认加载的是这个文件。

    libraryTarget: 'commonjs2'

与webpack.conf.js不同在于输出output的libraryExport。

前者暴露的是commonjs2，后者暴露的是umd。

### webpack --config build/webpack.component.js

与前两者的index.js入口不同，以components.json为入口，将每一个组件打包生成一个文件，用于按需加载。

npm run build:utils && npm run build:umd && npm run build:theme也已经讲过，分别是转译工具方法、转译语言包、生成样式文件。

## lint

"lint": "eslint src/\*\*/\* test/\*\*/\* packages/\*\*/\* build/\*\*/\* --quiet"

eslint校验src 和build目录下的文件。

## pub

"pub": "npm run bootstrap && sh build/git-release.sh && sh build/release.sh && node build/bin/gen-indices.js && sh build/deploy-faas.sh"

npm run bootstrap下载依赖。

sh build/git-release.sh：主要是检测dev分支是否冲突。

#!/usr/bin/env sh

# 切换到dev分支

git checkout dev

# 检测本地是否有未提交文件

if test -n "$(git status --porcelain)"; then

# 输出日志

  echo 'Unclean working tree. Commit or stash changes first.' >&2;

  exit 128;

fi

# 检测本地分支是否有误

if ! git fetch --quiet 2>/dev/null; then

  # 输出日志

  echo 'There was a problem fetching your branch. Run `git fetch` to see more...' >&2;

  exit 128;

fi

# 检测是否有最新提交

if test "0" != "$(git rev-list --count --left-only @'{u}'...HEAD)"; then

  # 输出日志

  echo 'Remote history differ. Please pull changes.' >&2;

  exit 128;

fi

# 输出日志

echo 'No conflicts.' >&2;

sh build/release.sh

脚本完成了以下工作：

1. 合并 dev 分支到 master
2. 修改样式包和组件库的版本号
3. 发布样式包和组件库
4. 提交 master 和 dev 分支到远程仓库

该脚本在发布组件库时可以使用，特别是其中自动更改版本号的功能（每次 publish 时都忘改版本号）。这里提交代码到远程仓库的日志很简单。

#!/usr/bin/env sh

set -e

# 切换到master

git checkout master

# 合并dev分支

git merge dev

# npx: 使用本地已安装的可执行工具，而不需要配置 scripts

VERSION=`npx select-version-cli`

# 更新版本号

read -p "Releasing $VERSION - are you sure? (y/n)" -n 1 -r

echo    # (optional) move to a new line

if [[ $REPLY =~ ^[Yy]$ ]]

then

  # 输出:压缩版本

  echo "Releasing $VERSION ..."

  # build

  # 编译打包

  VERSION=$VERSION npm run dist

  # ssr test

  node test/ssr/require.test.js

  # 发布到npm

  # publish theme

  # 输出:压缩theme-chalk版本

  echo "Releasing theme-chalk $VERSION ..."

  cd packages/theme-chalk

  # 更改主题包的版本信息

  npm version $VERSION --message "[release] $VERSION"

  # 如果是beta版本则打个beta标签

  if [[ $VERSION =~ "beta" ]]

  then

    npm publish --tag beta

  else

    npm publish

  fi

  cd ../..

  # commit

  git add -A

  git commit -m "[build] $VERSION"

  # 更改组件库的版本信息

  npm version $VERSION --message "[release] $VERSION"

  # publish

  # 发布到远程仓库

  git push eleme master

  git push eleme refs/tags/v$VERSION

  git checkout dev

  git rebase master

  git push eleme dev

  # 发布组件库

  if [[ $VERSION =~ "beta" ]]

  then

    npm publish --tag beta

  else

    npm publish

  fi

fi

 node build/bin/gen-indices.js

生成目录，支持搜索：

'use strict';

// 生成目录

const fs = require('fs');

const path = require('path');

// 是一个托管的全文、数字和分面搜索引擎，能够从第一次击键交付实时结果。

const algoliasearch = require('algoliasearch');

// 将Unicode str转换为段字符串，确保在URL或文件名中使用它是安全的。

// https://www.npmjs.com/package/transliteration?activeTab=readme

// demo:

// slugify('你好，世界');

// // ni-hao-shi-jie

const slugify = require('transliteration').slugify;

// 密钥

const key = require('./algolia-key');

const client = algoliasearch('4C63BTGP6S', key);

const langs = {

  'zh-CN': 'element-zh',

  'en-US': 'element-en',

  'es': 'element-es',

  'fr-FR': 'element-fr'

};

// 四种语言

['zh-CN', 'en-US', 'es', 'fr-FR'].forEach(lang => {

  const indexName = langs[lang];

  const index = client.initIndex(indexName);

  index.clearIndex(err => {

    if (err) return;

    // 读取/examples/docs/中的文件

    fs.readdir(path.resolve(\_\_dirname, `../../examples/docs/${ lang }`), (err, files) => {

      if (err) return;

      let indices = [];

      files.forEach(file => {

        console.log(file)

        const component = file.replace('.md', '');

        const content = fs.readFileSync(path.resolve(\_\_dirname, `../../examples/docs/${ lang }/${ file }`), 'utf8');

        const matches = content

          .replace(/:::[\s\S]\*?:::/g, '')

          .replace(/```[\s\S]\*?```/g, '')

          .match(/#{2,4}[^#]\*/g)

          .map(match => match.replace(/\n+/g, '\n').split('\n').filter(part => !!part))

          .map(match => {

            const length = match.length;

            if (length > 2) {

              const desc = match.slice(1, length).join('');

              return [match[0], desc];

            }

            return match;

          });

        indices = indices.concat(matches.map(match => {

          const isComponent = match[0].indexOf('###') < 0;

          const title = match[0].replace(/#{2,4}/, '').trim();

          const index = { component, title };

          index.ranking = isComponent ? 2 : 1;

          index.anchor = slugify(title);

          index.content = (match[1] || title).replace(/<[^>]+>/g, '');

          return index;

        }));

      });

      index.addObjects(indices, (err, res) => {

        console.log(err, res);

      });

    });

  });

});

## test

"test": "npm run lint && npm run build:theme && cross-env CI\_ENV=/dev/ BABEL\_ENV=test karma start test/unit/karma.conf.js --single-run"

"test:watch": "npm run build:theme && cross-env BABEL\_ENV=test karma start test/unit/karma.conf.js"

单元测试：UI 组件作为高度抽象的基础公共组件，编写单元测试是很有必要的。

# makefile

makefile带来的好处就是——“自动化编译”，一旦写好，只需要一个make命令，整个工程完全自动编译，极大的提高了软件开发的效率。 make是一个命令工具，是一个解释makefile中指令的命令工具，一般来说，大多数的IDE都有这个命令，比如：Delphi的make，Visual C++的nmake，Linux下GNU的make。可见，makefile都成为了一种在工程方面的编译方法。

make xxx其实执行的还是对应的npm xxxx，比如执行make dev实际上就是执行npm run dev，更加的快捷。

# Makefile中，.PHONY后面的target表示的也是一个伪造的target, 而不是真实存在的文件target，注意Makefile的target默认是文件。

.PHONY: dist test

# 执行make默认指向help

default: help

# 构建主题

# build all theme

build-theme:

    npm run build:theme

# 安装依赖

install:

    npm install

# 使用淘宝源安装依赖

install-cn:

    npm install --registry=http://registry.npm.taobao.org

# 构建应用

dev:

    npm run dev

play:

    npm run dev:play

# 新增组件

new:

    node build/bin/new.js $(filter-out $@,$(MAKECMDGOALS))

# 为网站添加新语言

new-lang:

    node build/bin/new-lang.js $(filter-out $@,$(MAKECMDGOALS))

# 打包

dist: install

    npm run dist

# 部署

deploy:

    @npm run deploy

# 发布

pub:

    npm run pub

# 单元测试

test:

    npm run test:watch

# 帮助信息

help:

    @echo "   \033[35mmake\033[0m \033[1m命令使用说明\033[0m"

    @echo "   \033[35mmake install\033[0m\t\033[0m\t\033[0m\t\033[0m\t---  安装依赖"

    @echo "   \033[35mmake new <component-name> [中文名]\033[0m\t---  创建新组件 package. 例如 'make new button 按钮'"

    @echo "   \033[35mmake dev\033[0m\t\033[0m\t\033[0m\t\033[0m\t---  开发模式"

    @echo "   \033[35mmake dist\033[0m\t\033[0m\t\033[0m\t\033[0m\t---  编译项目，生成目标文件"

    @echo "   \033[35mmake deploy\033[0m\t\033[0m\t\033[0m\t\033[0m\t---  部署 demo"

    @echo "   \033[35mmake pub\033[0m\t\033[0m\t\033[0m\t\033[0m\t---  发布到 npm 上"

    @echo "   \033[35mmake new-lang <lang>\033[0m\t\033[0m\t\033[0m\t---  为网站添加新语言. 例如 'make new-lang fr'"

除了make new 和make new-lang两个命令外，其他脚本命令都已经分析过了

## make new

node build/bin/new.js $(filter-out $@,$(MAKECMDGOALS))

新增组件

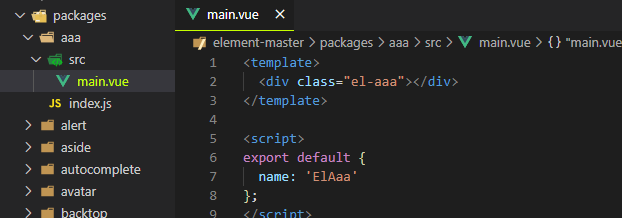
可以执行node .\build\bin\new.js aaa来生成aaa组件，方便快捷多了

添加新组件aaa，优势出来了：

1. 在/packages目录下新建组件目录，并完成目录结构的构建/packages/aaa
2. 创建组件文档，/examples/docs/{lang}/aaa.md
3. 创建组件单元测试文件，/test/unit/specs/aaa.spec.js
4. 创建组件样式文件，/packages/theme-chalk/src/aaa.scss
5. 创建组件类型声明文件，/types/aaa.d.ts
6. 配置: 在 /components.json 文件中配置组件信息, 在 /examples/nav.config.json 中添加该组件的路由配置,  在 /packages/theme-chalk/src/index.scss 文件中自动引入该组件的样式文件, 将类型声明文件在 /types/element-ui.d.ts 中自动引入



生成组件aaa如下：



来看下具体实现：

'use strict';

console.log();

process.on('exit', () => {

  console.log();

});

// 非填抛出异常

if (!process.argv[2]) {

  console.error('[组件名]必填 - Please enter new component name');

  // 程序退出

  process.exit(1);

}

const path = require('path');

const fs = require('fs');

// 写文件

const fileSave = require('file-save');

// 首字母大写

const uppercamelcase = require('uppercamelcase');

// 组件英文名

const componentname = process.argv[2];

// 组件中文名

const chineseName = process.argv[3] || componentname;

const ComponentName = uppercamelcase(componentname);

console.log(ComponentName)

const PackagePath = path.resolve(\_\_dirname, '../../packages', componentname);

const Files = [

  // 在src/main中写入index.js,注册组件

  {

    filename: 'index.js',

    content: `import ${ComponentName} from './src/main';

    /\* istanbul ignore next \*/

    ${ComponentName}.install = function(Vue) {

      Vue.component(${ComponentName}.name, ${ComponentName});

    };

    export default ${ComponentName};`

  },

  // 在packages中xxx组件的src/main下新建模板

  {

    filename: 'src/main.vue',

    content: `<template>

    <div class="el-${componentname}"></div>

</template>

<script>

export default {

  name: 'El${ComponentName}'

};

</script>`

  },

  // 四种语言的文档

  {

    filename: path.join('../../examples/docs/zh-CN', `${componentname}.md`),

    content: `## ${ComponentName} ${chineseName}`

  },

  {

    filename: path.join('../../examples/docs/en-US', `${componentname}.md`),

    content: `## ${ComponentName}`

  },

  {

    filename: path.join('../../examples/docs/es', `${componentname}.md`),

    content: `## ${ComponentName}`

  },

  {

    filename: path.join('../../examples/docs/fr-FR', `${componentname}.md`),

    content: `## ${ComponentName}`

  },

  // 在test下新建组件的单元测试

  {

    filename: path.join('../../test/unit/specs', `${componentname}.spec.js`),

    content: `import { createTest, destroyVM } from '../util';

import ${ComponentName} from 'packages/${componentname}';

describe('${ComponentName}', () => {

  let vm;

  afterEach(() => {

    destroyVM(vm);

  });

  it('create', () => {

    vm = createTest(${ComponentName}, true);

    expect(vm.$el).to.exist;

  });

});

`

  },

  // 新建组件的主题样式

  {

    filename: path.join('../../packages/theme-chalk/src', `${componentname}.scss`),

    content: `@import "mixins/mixins";

@import "common/var";

@include b(${componentname}) {

}`

  },

  // 新建组件的类型声明

  {

    filename: path.join('../../types', `${componentname}.d.ts`),

    content: `import { ElementUIComponent } from './component'

/\*\* ${ComponentName} Component \*/

export declare class El${ComponentName} extends ElementUIComponent {

}`

  }

];

// 添加到 components.json

const componentsFile = require('../../components.json');

if (componentsFile[componentname]) {

  console.error(`${componentname} 已存在.`);

  process.exit(1);

}

componentsFile[componentname] = `./packages/${componentname}/index.js`;

fileSave(path.join(\_\_dirname, '../../components.json'))

  .write(JSON.stringify(componentsFile, null, '  '), 'utf8')

  .end('\n');

// 添加到 index.scss

const sassPath = path.join(\_\_dirname, '../../packages/theme-chalk/src/index.scss');

const sassImportText = `${fs.readFileSync(sassPath)}@import "./${componentname}.scss";`;

fileSave(sassPath)

  .write(sassImportText, 'utf8')

  .end('\n');

// 添加到 element-ui.d.ts

const elementTsPath = path.join(\_\_dirname, '../../types/element-ui.d.ts');

let elementTsText = `${fs.readFileSync(elementTsPath)}

/\*\* ${ComponentName} Component \*/

export class ${ComponentName} extends El${ComponentName} {}`;

const index = elementTsText.indexOf('export') - 1;

const importString = `import { El${ComponentName} } from './${componentname}'`;

elementTsText = elementTsText.slice(0, index) + importString + '\n' + elementTsText.slice(index);

fileSave(elementTsPath)

  .write(elementTsText, 'utf8')

  .end('\n');

// 创建 package

Files.forEach(file => {

  fileSave(path.join(PackagePath, file.filename))

    .write(file.content, 'utf8')

    .end('\n');

});

// 添加到 nav.config.json

const navConfigFile = require('../../examples/nav.config.json');

Object.keys(navConfigFile).forEach(lang => {

  let groups = navConfigFile[lang][4].groups;

  groups[groups.length - 1].list.push({

    path: `/${componentname}`,

    title: lang === 'zh-CN' && componentname !== chineseName ?

      `${ComponentName} ${chineseName}` : ComponentName

  });

});

fileSave(path.join(\_\_dirname, '../../examples/nav.config.json'))

  .write(JSON.stringify(navConfigFile, null, '  '), 'utf8')

  .end('\n');

console.log('DONE!');

## make new-lang

node build/bin/new-lang.js $(filter-out $@,$(MAKECMDGOALS))

添加新语言

'use strict';

console.log();

process.on('exit', () => {

  console.log();

});

if (!process.argv[2]) {

  console.error('[language] is required!');

  process.exit(1);

}

var fs = require('fs');

const path = require('path');

const fileSave = require('file-save');

const lang = process.argv[2];

// const configPath = path.resolve(\_\_dirname, '../../examples/i18n', lang);

// 添加到 components.json

const componentFile = require('../../examples/i18n/component.json');

if (componentFile.some(item => item.lang === lang)) {

  console.error(`${lang} already exists.`);

  process.exit(1);

}

let componentNew = Object.assign({}, componentFile.filter(item => item.lang === 'en-US')[0], { lang });

componentFile.push(componentNew);

fileSave(path.join(\_\_dirname, '../../examples/i18n/component.json'))

  .write(JSON.stringify(componentFile, null, '  '), 'utf8')

  .end('\n');

// 添加到 page.json

const pageFile = require('../../examples/i18n/page.json');

let pageNew = Object.assign({}, pageFile.filter(item => item.lang === 'en-US')[0], { lang });

pageFile.push(pageNew);

fileSave(path.join(\_\_dirname, '../../examples/i18n/page.json'))

  .write(JSON.stringify(pageFile, null, '  '), 'utf8')

  .end('\n');

// 添加到 route.json

const routeFile = require('../../examples/i18n/route.json');

routeFile.push({ lang });

fileSave(path.join(\_\_dirname, '../../examples/i18n/route.json'))

  .write(JSON.stringify(routeFile, null, '  '), 'utf8')

  .end('\n');

// 添加到 nav.config.json

const navFile = require('../../examples/nav.config.json');

navFile[lang] = navFile['en-US'];

fileSave(path.join(\_\_dirname, '../../examples/nav.config.json'))

  .write(JSON.stringify(navFile, null, '  '), 'utf8')

  .end('\n');

// docs 下新建对应文件夹

try {

  fs.statSync(path.resolve(\_\_dirname, `../../examples/docs/${ lang }`));

} catch (e) {

  fs.mkdirSync(path.resolve(\_\_dirname, `../../examples/docs/${ lang }`));

}

console.log('DONE!');

# 其他工程化脚本

## md-loader

除了md-loader，build下面的脚本基本上是分析了一遍。

说到md-loader，官网的文档展示和demo展示多亏了他。

它是一个 loader，官网组件页面的组件demo + 文档的模式一大半的功劳都是源自于它。

可以在 /examples/route.config.js 中看到 registerRoute 方法生成组件页面的路由配置时，使用 loadDocs 方法加载/examples/docs/{lang}/comp.md 。

注意，这里加载的 markdown 文档，而不是平时常见的 vue 文件，但是却能像 vue 文件一样在页面上渲染成一个 Vue 组件，这是怎么做到的呢？

我们知道，webpack 的理念是一切资源都可以 require，只需配置相应的 loader 即可。在 /build/webpack.demo.js 文件中的 module.rules 下可以看到对 markdown规则的处理，先通过 md-loader 处理 markdown 文件，从中解析出 vue 代码，然后交给 vue-loader，最终生成vue 单文件组件渲染到页面。这就能看到组件页面的文档和组件demo展示效果。

      {

        test: /\.md$/,

        use: [

          {

            loader: 'vue-loader',

            options: {

              compilerOptions: {

                preserveWhitespace: false

              }

            }

          },

          {

            loader: path.resolve(\_\_dirname, './md-loader/index.js')

          }

        ]

      },

至于如何将markdown解析成vue组件，可以阅读[《谈谈 Element 文档中的 Markdown 解析》](https://zhuanlan.zhihu.com/p/65174076) 。

## config.js

webpack 的公共配置，比如 externals、alias 等。通过 externals 的配置解决了组件库部分代码的冗余问题，比如组件和组件库公共模块的代码，但是组件样式冗余问题没有得到解决；alias 别名配置为开发组件库提供了方便。

// webpack 公共配置，比如 externals、alias

var path = require('path');

var fs = require('fs');

var nodeExternals = require('webpack-node-externals');

var Components = require('../components.json');

var utilsList = fs.readdirSync(path.resolve(\_\_dirname, '../src/utils'));

var mixinsList = fs.readdirSync(path.resolve(\_\_dirname, '../src/mixins'));

var transitionList = fs.readdirSync(path.resolve(\_\_dirname, '../src/transitions'));

var externals = {};

Object.keys(Components).forEach(function (key) {

  externals[`element-ui/packages/${key}`] = `element-ui/lib/${key}`;

});

// externals 解决组件依赖其它组件并按需引入时代码冗余的问题

// 比如 Table 组件依赖 Checkbox 组件，在项目中如果我同时引入 Table 和 Checkbox 时，会不会产生冗余代码

// 如果没有以下内容的的话，会，这时候你会看到有两份 Checkbox 组件代码。

// 包括 locale、utils、mixins、transitions 这些公共内容，也会出现冗余代码

// 但有了 externals 的设置，就会将告诉 webpack 不需要将这些 import 的包打包到 bundle 中，运行时再从外部去

// 获取这些扩展依赖。这样就可以在打包后 /lib/tables.js 中看到编译后的 table.js 对 Checkbox 组件的依赖引入：

// module.exports = require("element-ui/lib/checkbox")

// 这么处理之后就不会出现冗余的 JS 代码，但是对于 CSS 部分，element-ui 并未处理冗余情况。

// 可以看到 /lib/theme-chalk/table.css 和 /lib/theme-chalk/checkbox.css 中都有 Checkbox 组件的样式

externals['element-ui/src/locale'] = 'element-ui/lib/locale';

utilsList.forEach(function (file) {

  file = path.basename(file, '.js');

  externals[`element-ui/src/utils/${file}`] = `element-ui/lib/utils/${file}`;

});

mixinsList.forEach(function (file) {

  file = path.basename(file, '.js');

  externals[`element-ui/src/mixins/${file}`] = `element-ui/lib/mixins/${file}`;

});

transitionList.forEach(function (file) {

  file = path.basename(file, '.js');

  externals[`element-ui/src/transitions/${file}`] = `element-ui/lib/transitions/${file}`;

});

externals = [Object.assign({

  vue: 'vue'

}, externals), nodeExternals()];

exports.externals = externals;

// 设置别名

exports.alias = {

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

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

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

  'element-ui': path.resolve(\_\_dirname, '../')

};

exports.vue = {

  root: 'Vue',

  commonjs: 'vue',

  commonjs2: 'vue',

  amd: 'vue'

};

exports.jsexclude = /node\_modules|utils\/popper\.js|utils\/date\.js/;

# Yabby-ui

阅读了Element整体架构后，在Element基础上二次开发组里的Yabby-ui组件库，其实不是很复杂了，网站和组件库的构建流程可以不用变更太多。网站的样式可以转化成组里独特的样式风格。在原有组件的样式基础上根据设计稿修改全局颜色变量。通过导入全局样式变量文件更改主题，使用组里所有的项目。

总结

Element整体架构是真的非常棒，利用脚本实现工程化，值得我们在开发中学习和应用。涉及到添加新组件、添加新语言、构建应用、打包编译应用、发布应用、单元测试等等。比如添加新组件，执行完脚本可以帮助开发者解决创建新组件的目录、四种语言下的文档、配置官网展示文件等。

开发者只需要编写具体组件的逻辑即可，根本不需要每一次新建一个组件改动多个文件，拒绝重复劳动。切记，在项目搭建的过程中，可以使用脚本解决的事情，坚决不要手动解决。

参考文献

[ElementUI的构建流程](https://juejin.cn/post/6844903680357957645#heading-0)

[从 Element UI 源码的构建流程来看前端 UI 库设计](https://juejin.cn/post/6844904197863964685#heading-0)

[如何快速为团队打造自己的组件库（上）—— Element 源码架构](https://juejin.cn/post/6935977815342841892)