

- 01 | 构建配置包设计
- 02 | 功能模块设计和目录结构
- 03 | 使用ESlint 规范构建脚本
- 04 | 冒烟测试 (smoke testing) 介绍和实际运用 🌟 (保证构建包 基本可用)
- 05 | 单元测试和测试覆盖率 (保证更加细节的)
- 06 | 持续集成和Travis CI
- 07 | 发布构建包到社区
- 08 | Git Commit规范 和changelog生成
- 09 | 语义化版本 (Semantic Versioning) 规范格式

## 01 | 构建配置包设计

---

- 构建配置抽离成 npm 包的意义
  - 通用性
    - 业务开发者无需关注构建配置
    - 统一团队构建脚本
  - 可维护性
    - 构建配置合理的拆分
    - README 文档、ChangeLog文档等
  - 质量 🌟
    - 冒烟测试、单元测试、测试覆盖率
    - 持续集成
- 构建配置管理的可选方案
  - 方案一：通过多个配置文件管理不同环境的构建，比如：webpack --config 参数进行控制
  - 方案二：将构建配置设置成一个库，比如：hjs-webpack、Neutrino、webpack-blocks (推荐)
  - 方案三：抽成一个工具进行管理，比如：create-react-app、kyt、nwb(业内)
  - 方案四：将所有的配置放在一个文件，通过 --env参数控制分支选择
  - 小结：结合 1、2方案构建 团队不大 (十几二十个人)，当团队规模足够大可以做工具
- 构建配置包设计
  - 通过多个配置文件管理不同环境的 webpack 配置 这样设计方便后面的扩展
    - 基础配置：webpack.base.js
    - 开发环境：webpack.dev.js
    - 生产环境：webpack.prod.js
    - SSR环境：webpack.ssr.js
    - PWA环境：webpack.pwa.js
    - 其他环境...
  - 抽离成一个 npm 包统一管理
    - 规范：Git commit 日志、README、ESLint规范、Semver规范
    - 质量：冒烟测试、单元测试、测试覆盖率和CI(持续集成)
- 通过 webpack-merge 组合配置
  - 合并配置: module.exports = merge(baseConfig, devConfig)

```
const merge = require('webpack-merge')
// ...
merge(
```

```
{ a: [1], b: 5, c: 20},
{ a: [2], b: 10, d: 421 }
)
// => { a: [1, 2], b: 10, c: 20, d: 421 }
// ...
// 合并配置
module.exports = merge(baseConfig, devConfig)
```

## 02 | 功能模块设计和目录结构

- 功能模块设计
  - 基础配置
    - 资源解析
      - 01 解析ES6 babel-loader & @babel/babel-env
      - 02 解析react react & react-dom & @babel/babel-react
      - 03 解析CSS css-loader style-loader | mini-css-extra-webpack-plugin
      - 04 解析Less less & less-loader
      - 05 解析图片 url-loader
      - 06 解析字体 file-loader
    - 样式增强
      - CSS 前缀补齐 postcss & autoprefixer
      - CSS px转换成rem
    - 目录清理 clean-webpack-plugin
    - 多页面打包 html-webpack-plugin 动态获取 entry
    - 命令行信息显示优化 stats: 'errors-only' & friendly-errors-webpack-plugin
    - 错误信息显示优化 compiler.hooks.done & process.exit(1)
    - CSS 提取成一个单独的文件 mini-css-extract-plugin 的 loader
  - 开发阶段配置
    - 代码热更新
      - CSS 热更新
      - JS 热更新
    - source map
      - 设置 devtool: 'source-map'
  - 生产阶段配置 (mode: 'production')
    - 代码压缩
      - JS 默认压缩
      - CSS
      - HTML
    - 文件指纹
      - hash
      - chunkhash
      - contenthash
    - Tree Shaking 默认开启
    - Scope Hoisting 默认开启
    - 速度优化 🌟
      - 基础包CDN

- ...
- 体积优化 🌟
  - 代码分割 code-split
  - ...
- SSR配置
  - output 的 libraryTarget 设置
  - CSS 解析 ignore

## 功能模块设计



- - 目录结构设计
    - /test 放置测试代码
    - /lib 放置源代码
      - webpack.dev.js
      - webpack.prod.js
      - webpack.srr.js
      - webpack.base.js
  - README.md
  - CHANGELOG.md
  - .eslint.js
  - package.json
  - index.js

## 03 | 使用ESlint 规范构建脚本

- 使用 eslint-config-airbnb-base
- eslint --fix 可以自动处理空格
- 命令使用 ./node\_modules/.bin/eslint lib/
- 跳过eslint检查行 // eslint-disable-line

```
module.exports = {
  "parser": "babel-eslint",
  "extends": "airbnb-base",
  "env": {
    "browser": true,
    "node": true
  }
}
```

## 04 | 冒烟测试 (smoke testing) 介绍和实际运用 🌟 (保证构建包 基本可用)

- 【定义】冒烟测试 是指 对提交测试的软件在进行详细深入的测试之前而进行的测试 这种测试的主要目的是 暴露导致软件需重新发布的基本功能失效等严重问题 保证基础功能可用
- 冒烟测试执行
  - 构建是否成功
  - 每次构建完成build目录是否有内容输出
  - 是否有 JS、CSS 等静态资源文件
  - 是否有 HTML 文件
- 如何判断构建是否成功
  - 在示例项目里面运行构建，看看是否报错
  - 方式一：在命令行里面运行npm run build / dev 可以惊醒一次构建
  - 方式二：把我们编辑好的配置传给webpack函数 webpack函数执行这个功能执行之后 里面的回调函数 里面会有err 和 stats 我们把一些基本的统计信息打印出来
  - 使用 node test/smoke/index.js 运行 冒烟测试

```
/* 每次构建前 利用 rimraf 库 要先把 dist 目录删掉 */
const path = require('path');
const webpack = require('webpack');
const rimraf = require('rimraf');
const Mocha = require('mocha')

const mocha = new Mocha({
  timeout: '10000ms'
})

process.chdir(path.join(__dirname, 'template')) // 先进入到这个
smoke/template目录来

rimraf('./dist', () => {
  const prodConfig = require('../lib/webpack.prod')

  webpack(prodConfig, (err, stats) => {
    if (err) {
```

```
        console.error(err)
        process.exit(2)
    }
    console.log(stats.toString({
        color: true,
        modules: false,
        children: false
    })))

    console.log('Webpack build success, begin run test.')

    mocha.addFile(path.join(__dirname, 'html-test.js')) // 加入测试用例
    mocha.addFile(path.join(__dirname, 'css-js-test.js')) // 加入测试用例

    mocha.run()

    })
})
```

- 判断基本功能是否正常
  - 编写 mocha 测试用例
    - 是否有 JS、CSS等静态资源文件
    - 是否有 HTML文件

```
const glob = require('glob-all');

describe('Checking generated html files', () => {
    it ('should generate html files', (done) => {
        // 通了就运行 done
        const files = glob.sync([
            './dist/index.html',
            './dist/search.html'
        ])

        if (files.length > 0) {
            done()
        } else {
            throw new Error('not html files generated')
        }
    })
});
```

## 05 | 单元测试和测试覆盖率（保证更加细节的）

- 市面上单元测试库框架

- mocha (简单 & 成熟)、ava 单纯的测试框架, 需要断言库
  - chai
  - should.js
  - expect
  - better-assert
- jasmine 和 jest (react 官方推荐) 集成框架, 开箱即用
- 极简 API
- 选择 mocha + chai 编写单元测试用例
  - 技术选型: Mocha + Chai
  - 测试代码: describe(描述文件 可以有多个it)、it、expect(期望)
  - 测试命令: mocha、add.test.js

```
const expect = require('chai').expect
const add = require('../src/add')

describe('use expect: src/add.js', () => {
  it('add(1, 2) === 3', () => {
    expect(add(1, 2).to.equal(3))
  })
})
```

- 单元测试接入 (异步请求也ok)
  - 1、安装 [mocha](#) + [chai](#) -> npm i mocha chai -D
  - 2、新建test目录 并增加 xx.test.js 测试文件
  - 3、在 webpack.json 中的 script 字段 添加 test 命令

```
{
  "test": "node_modules/mocha/bin/_mocha"
}
```

- 4、执行测试命令 npm run test

```
// test/index.js
const path = require('path')

process.chdir(path.join(__dirname, 'smoke/template')) // 先去冒烟测试

describe('builder-webpack-zy test case', () => {
  require('../unit/webpack-base.test')
});

// test/unit/webpack-base.test.js
const assert = require('assert'); // 断言

describe('webpack.base.js test case', () => {
  const baseConfig = require('../../lib/webpack.base')
```

```
// console.log('base config :', baseConfig)

it ('entry', () => {
  assert.equal(baseConfig.entry.index,
    '/Users/lizhenyu/Career/github/webpack-course/配置/build-webpack-zy/src/index/index.js');
  assert.equal(baseConfig.entry.search,
    '/Users/lizhenyu/Career/github/webpack-course/配置/build-webpack-zy/src/search/index.js');
})
});
```

- 测试覆盖率使用 [istanbul](#) 不再维护了 替换使用nyc
  - 安装 istanbul : npm i istanbul -D 推荐使用 nyc - npm i nyc -D
  - 使用: istanbul cover test.js - "coverage": "nyc npm run test"

## 06 | 持续集成和Travis CI

---

- 持续集成的作用
  - 优点：（确保质量）
    - 快速发现问题
    - 防止分支大幅偏离主干
  - 核心措施是，代码集成到主干之前，必须通过自动化测试。只要有一个测试用例失败，就不能集成
- Github 最流行的CI Travis CI 占据了一般以上
- 接入 Travis CI
  1. [travis 登录](#) 使用github 账号登录
  2. 在 [travis仓库](#) 为项目开启
  3. 项目根目录下新增 .travis.yml
- travis.yml 文件内容
  - install 安装项目依赖
  - script 运行测试用例

```
language: node_js

sudo: false # 是否使用sudo全县

cache: # 是否开启缓存
  apt: true
  directories:
    - node_modules # 开启node_modules缓存

node_js: stable # 设置响应的版本

install:
  - npm install -D # 安装构建器依赖
  - cd ./test/template-project
```

```
- npm install -D # 安装模板项目依赖

script:
  - npm test
```

## 07 | 发布构建包到社区

---

- 先去 npm 看一下名字是否重复
- 先登录 npm login (若登录 则省去)
- 添加用户: npm adduser
- 升级版本 (会自动添加 git log 和 git tag)
  - 升级补丁版本号: npm version patch (修改了一个 bug)
  - 升级小版本号: npm version minor (增加了一个 feature)
  - 升级打版本号: npm version major (重大功能更新)
- 发布版本: npm publish
- 撤销版本: npm unpublish --force
- 要求
  - 包不能太大了 移除一些很大的静态资源
- 更新版本之前记得 git add commit 一下
- 更新

## 08 | Git Commit规范和changelog生成

---

- 良好的Git Commit 规范优势:
  - 加快 Code Review 的流程
  - 根据 Git Commit 的元数据生成 Changelog
  - 后续维护者可以知道Feature 被修改的原因
- 技术方案
  - Git 提交格式
    - ✨ 统一团队 Git Commit 日志标准, 便于后续 code review 和 版本发布
    - 使用angular的git commit 日志作为基本规范
      - 提交类型限制为:
        - feat: 新增feature
        - fix: 修复bug
        - docs: 仅仅修改了文档, 比如README, CHANGELOG, CONTRIBUTE等等
        - style: 仅仅修改了空格、格式缩进、逗号等, 不改变代码逻辑
        - refactor: 代码重构, 没有加新功能或者修复bug
        - perf: 优化相关, 比如提升性能、体验
        - test: 测试用例, 包括单元测试、集成测试等
        - chore: 改变构建流程、或者增加依赖库、工具等
        - revert: 回滚到上一个版本
        - ...
      - 提交信息分为两部分, 标题 (首字母不大写。末尾不要标点)、主体内容 (正常的描述信息即可)



- 日志提交时友好的类型选择提示 -- 使用commitize工具
- 不符合要求格式的日志拒绝提交的保障机制
  - 使用 validate-commit-msg 工具
  - 需要同时在客户端、gitlab server hook做
- 统一 changelog 文档信息生成 -- 使用conventional-changelog-cli 工具



- 提交格式要求

```
<type>(<scope>):<subject>
<BLANK LINE>
<body>
<BLANK LINE>
<footer> 修复bug 或 issue 附上连接
```

- 本地开发阶段增加 precommit 钩子
  - 安装 husky - npm i husky -D
  - 通过 commitmsg 钩子校验信息

```
{
  "scripts": {
    "commitmsg": "validate-commit-msg",
    "changelog": "conventional-changelog -p angular -i CHANGELOG.md -s -r 0"
  },
  "devDependencies": {
    "validate-commit-msg": "^2.11.1",
    "conventional-changelog-cli": "^1.2.0",
    "husky": "^0.13.1"
  }
}
```

## 09 | 语义化版本（Semantic Versioning）规范格式

- 由 github 提出来 版本打 tag 版本号怎么加
- 为了解决软件开发领域 版本依赖地狱的问题
- 开源项目版本信息案例
  - 软件的版本通常由三位组成，形如 x.y.z
  - 版本是严格递增的，此处是:16.2.0 -> 16.3.0 -> 16.3.1
  - 在发布重要版本时，可以发布 alpha、rc 等先行版本
    - alpha 版本号 一般用于内部的灰度测试
    - bata 版本号 一般用于外部的小范围测试

- rc 版本号 一般用于公测

## 开源项目版本信息案例



软件的版本通常由三位组成，形如：  
X.Y.Z



版本是严格递增的，此处是：16.2.0 –  
> 16.3.0 -> 16.3.1

在发布重要版本时，可以发布alpha，  
rc等先行版本

16.4.0-alpha.3174632	a year ago
16.4.0-alpha.7926752	a year ago
16.3.0-alpha.1	a year ago
16.3.0-alpha.0	a year ago
16.2.0	2 years ago
16.1.1	2 years ago
16.1.0	2 years ago
16.1.0-rc	2 years ago
16.1.0-beta.1	2 years ago
16.1.0-beta	2 years ago
16.0.0	2 years ago
15.6.2	2 years ago
16.0.0-rc.3	2 years ago
16.0.0-rc.2	2 years ago
16.0.0-rc.1	2 years ago
16.0.0-beta.5	2 years ago
16.0.0-beta.4	2 years ago

### React 版本信息

- 遵守 semver 规范的优势
  - 避免出现循环依赖
  - 依赖冲突减少
- 语义化版本（Semantic Versioning）规范格式
  - 主版本号：当你做了不兼容的API修改
  - 次版本号：当你做了向下兼容的功能性新增 - 一般是指 新增 feature
  - 修订号：当你做了向下兼容的问题修正 - 一般是指 bug fix
- 先行版本号
  - 先行版本号可以作用发布正式版之前的版本，格式是在修订版本号后面加上一个连接号 (-) 再加上一连串以点 (.) 分割的表示服，标示符可以由英文、数字和连接号 ([0-9A-Za-z]) 组成
    - alpha：是内部测试版，一般不向外部发布，会有很多bug。一般只有测试人员使用
    - beta：也是测试版，这个阶段的版本会一直加入新的功能。在Alpha版之后推出
    - rc：Release Candidate 系统平台上就是发行候补版本。RC 版不会再加入新的功能了，主要着重于除错