前端工程化

前端工程化是使用软件工程的技术和方法来进行前端的开发流程，技术，工具，经验等规范化，标准化，其主要目的是为了提高效率和降低成本，即提高开发过程中的开发效率，减少不必要的重复工作时间

yeoman

```
yarn global add yo

// 要装 generator-node 来配合yo 使用
// 不用的generator生产的项目就不一样
yarn global add generator-node

//生产项目 yo node 前面的generator不需要
```

Module Name 创建的项目名称

The name above already exists on npm . choose another? 意思是你npm中使用过了这个名字，要不要更改名字

Description 描述信息

Project homepage url 你的主页地址

Author's Name 你的名称

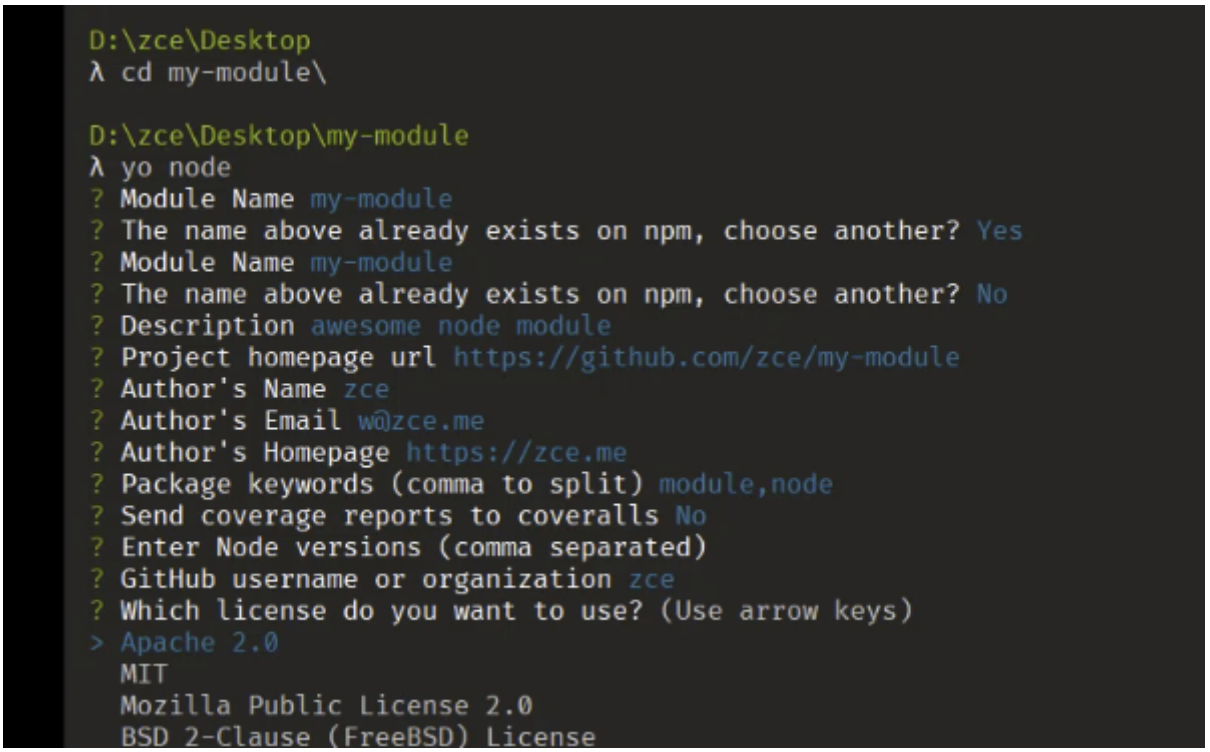
Author's Email 你的邮箱

Package ketwords (comma to split) 输入一些关键字

Send coverage reports to coveralls 持续继承和代码质量保证的工具

Enter node versions (comma separated) 让你输入一个node支持的版本 不输入就默认所有版本

GitHub username or organization 输入GitHub的用户名



上面的创建一整个项目的生成

这里是生产一个小配置例如一个eslint的配置文件

```
// 生成一个子项目
yo node:cli



// 在局部返回通过yarn link 到全局范围

//不是每一个generator都有子集的生成器，有才能创建，要在官网查看是否有子集

 //https://github.com/yeoman/generator-node node生成器的官网
```

npm镜像的加载地址

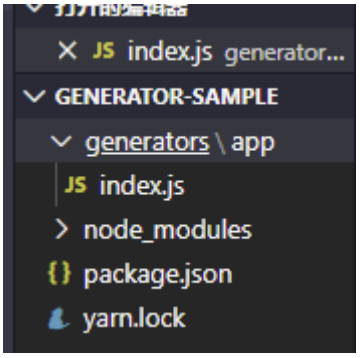
```
# mirror config
sharp_dist_base_url = https://npm.taobao.org/mirrors/sharp-libvips/v8.9.1/
profiler_binary_host_mirror = https://npm.taobao.org/mirrors/node-inspector/
fse_binary_host_mirror = https://npm.taobao.org/mirrors/fsevents
node_sqlite3_binary_host_mirror = https://npm.taobao.org/mirrors
sqlite3_binary_host_mirror = https://npm.taobao.org/mirrors
sqlite3_binary_site = https://npm.taobao.org/mirrors/sqlite3
sass_binary_site = https://npm.taobao.org/mirrors/node-sass
electron_mirror = https://npm.taobao.org/mirrors/electron/
puppeteer_download_host = https://npm.taobao.org/mirrors
chromedriver_cdnurl = https://npm.taobao.org/mirrors/chromedriver
operadriver_cdnurl = https://npm.taobao.org/mirrors/operadriver
phantomjs_cdnurl = https://npm.taobao.org/mirrors/phantomjs
python_mirror = https://npm.taobao.org/mirrors/python
registry = https://registry.npm.taobao.org/
disturl = https://npm.taobao.org/dist
```

自定义Generator

yeoman 的模块名称必须是generator-<name>开头的

```
yarn init
// 这个模块提供了generator的一个基类，让我们创建生成器的时候更加便捷
yarn add yeoman-generator
```

创建generators\app\index.js　此文件是作为Generator的核心入口




```
// 次文件作为Generator 的核心入口

// 需要到处一个继承自Yeoman Generator 的类型
// Yeoman Generator 在工作时会自动调用我们在次类型中定义的一些生命周期方法
// 我们在这些方法中可以通过调用父类提供的一些工具方法实现一些功能，例如文件写入
const Generator = require("yeoman-generator");

module.exports = class extends Generator {
  // Yeoman 在询问用户环节会自动调用此方法
   // 在次方法中可以调用父类的prompt() 方法发出对用户的命令行询问
```

```
  prompting() {
    return this.prompt([
      {
        type: "input",
        // 最终得到结果的键的名称
        name: "name",
        // 询问时候出现的信息
        message: "Your project name",
        default: this.appname, // appname 为项目生成目录名称
      },
    ]).then((answers) => {
      // 里面是一个数组
      this.answers = answers;
    });
  }

  writing() {
    // yeoman 自动在生产文件阶段调用此方法
    // 我们这里尝试往项目目录中写入文件
    // this.fs.write(
    //   // 第一个参数就是路径，通过这个方法可以获取到当前项目下的tem.txt路径
    //   this.destinationPath("tem.txt"),
    //   // 第二个参数就是写入的内容
    //   Math.random().toString()
    // );

    //  使用模板的情况下这样写  创建一个templates
  // 传入值得模板语法是<%= name %>

    // 模板文件路径
    const tmpl = this.templatePath(foo.txt);
    // 输出目标路径
    const output = this.destinationPath(foo.txt);
    // 模板数据上下文
    const context = this.answers;

    this.fs.copyTpl(tmpl, output, context);
  }
};
```

```
yarn link 局部变全局

// 创建对应的项目
yo web
```

发布 Generator

```
创建一个github仓库
然后上传到github上面，
然后使用yarn publish 将代码提交到yran上面，就能从npm上面看到了

使用了其他镜像源会报错
yarn publish --registry=https://registry.yarnpkg.com
上传上去后，就可以用yarn进行generator进行安装，然后进行使用了
```

Plop

```
yarn add plop --dev

// 根目录下创建一个profile.js文件
```

```
// Plop 入口文件，需要导出一个函数
// 此函数接受一个 plop对象， 用于创建生成器任务

module.exports = plop => {
    plop.setGenerator('component', {
        // 信息的描述
        description: 'create a component',
        prompts: [
            {
                type: 'input',
                name: 'name',
                message: 'component name',
                default: 'MyComponent'
            }
        ],
        // 完成命令行交互过后需要执行的一些动作
        actions: [
 // 一个对象代表一个动作
            {
                type: 'add', //代表添加文件
```

```
                path: 'src/components/{{name}}/{{name}}.js',

                // Plop模板一般在根目录下创建一个plop-templates的文件 下的component.hbs文件
                templateFile: 'plop-templates/component.hbs'
            },
  {
                type: 'add', //代表添加文件
                path: 'src/components/{{name}}/{{name}}.js',

                // Plop模板一般在根目录下创建一个plop-templates的文件 下的component.hbs文件
                templateFile: 'plop-templates/component.hbs'
            },
  {
                type: 'add', //代表添加文件
                path: 'src/components/{{name}}/{{name}}.js',

                // Plop模板一般在根目录下创建一个plop-templates的文件 下的component.hbs文件
                templateFile: 'plop-templates/component.hbs'
            }
        ]
    })
}
```

使用方法

yarn plop component

脚手架工作原理

```
创建项目目录
// 初始化信息
yarn init

在package.json中添加bin: "cli.js"

```

cli脚手架的结构代码

```
#!/use/bin/env node

// node CLI 应用入口文件必须要有这样的文件头
// 如果是 linux 或者 macOS 系统下还需要修改此文件读写权限为755
// 具体就是通过 chmod 755


// 脚手架的工作过程:
// 1.通过命令行交互询问用户问题
// 2.根据用户回答的结果生成文件

const inquirer = require('inquirer')
const path = require('path')
const fs = require('fs')
const ejs = require('ejs')

// 帮发起命令行询问
inquirer.prompt([
    {
        type: 'input',
        name: 'name',
        message: 'Project name?'
    }
])
.then( anwsers => {
    // 模板一般都在templateswen文件夹中
    // 根据用户回答的结果生成文件

    // 模板目录
    const tmplDir = path.join(__dirname, 'templates')
    // 目标目录
    const destDir = process.cwd()

    // 将模板下的文件全部转换到目标目录
    fs.readdir(tmplDir, (err, files) => {
```

```
        if (err) throw err
        files.forEach(file => {
            // 通过模板引擎渲染文件
            // 第一个参数是绝对路径
            // 第二个参数数据上下文，就是上面提示用户输入的信息
            // 第三个就是渲染成功后的回调函数
            ejs.renderFile(path.join(tmplDir, file), anwsers, (err, result) => {
                if (err) throw err
                // 将结果内容写入到文件中
                // 第一个是写入的路径，第二个就是文件的内容
                fs.writeFileSync(path.join(destDir, file), result)
            })
        })
    })
})
```

cli正式写法

```
#!/usr/bin/env node

const program = require("commander");
const chalk = require("chalk");
const ora = require("ora");
const download = require("download-git-repo");
const { parse } = require("commander");
const tplObj = require(`${__dirname}/../template`);

// 这里写的只是相当于一个帮助信息
program.usage("<template-name> [project-name]");

program
  .command("init <template> [project]")
  .description("初始化项目模板")
  // 前面是后面的缩写，都是通过force来获取值
  .option("-f, --force <template>")
  // action是前面参数的回调，前面的输入都会在action中拿到，需要后面加.parse(process.argv)，action才会执行
  .action(function (template, project) {
    url = tplObj[template];
    console.log(chalk.white("\n Start generating... \n"));

    // 出现加载图标
```

```
    const spinner = ora("Downloading...");
    // 加载的开始
    spinner.start();

    // 执行下载
    // 这里就是通过github上面的地址，然后把模块下载到本地
    download(`direct:${url}`, project, { clone: true }, (err) => {
      if (err) {
        // 加载错误
        spinner.fail();
        console.log(chalk.red(`Generation failed. ${err}`));
        return;
      }
      // 加载成功
      spinner.succeed();
      console.log(chalk.green("\n Generation completed!"));
      console.log("\n To get started");
      console.log(`\n cd ${project} \n`);
    });
  });
program.parse(process.argv);
```

写完内容后

```
将这个内容发布到全局
yarn link 这样这个文件的名称就可以直接用于生成了

发布到github前
echo node_modules > .gitignore

脚手架的基本原理就是把模板内的文件写入到新的目录
```

- 自动构建
  - sass 安装 yarn add sass --dev
  - 对sass文件进行编译转换为css文件
    - yarn sass ./sass/main.scss css/style.css

- NPM Scripts 是为了解决自动执行构建任务问题的
  - 安装 browser-sync --dev 用于去启动一个测试服务器运行项目
    - Script 钩子函数 preserve 运行命令 它会在serve运行之前运行preserve命令
    - 因为 sass --watch 会阻塞，然后进行监听，
  - yarn add npm-run-all --dev
    - 可以使用 npm-run-all 来解决，同时运行多个命令

这个构建过程就是

sass 负责监听sass文件然后编译成css文件

browser-sync 负责启动服务器，然后监听css的变化

run-all 负责同时启动sass和browser-sync这两个任务

```
{
 "name": "my-web-app",
 "version": "0.1.0",
 "main": "index.js",
 "author": "zce <w@zce.me> (https://zce.me)",
 "license": "MIT",
 "scripts": {
 // 监听scss文件的变化，会阻塞监听，所以需要使用run-p来同时运行多个命令
 "build": "sass scss/main.scss css/style.css --watch",
  // 这里的files是监听文件的变化，将变化立即刷新到浏览器，不用手动刷新
  "serve": "browser-sync . --files \"css/*.css\"",
  "start": "run-p build serve"
  },
  "devDependencies": {
  "browser-sync": "^2.26.7",
  "npm-run-all": "^4.1.5",
  "sass": "^1.22.10"
  }
```

```
}
```

Grunt 基本用法

- yarn add grunt
- 添加gruntfile.js文件

```
// Grunt 的入口文件
// 用于定义一些需要Grunt自动执行的任务
// 需要导出一个函数
// 此函数接受一个 grunt 的形参，内部提供一些创建任务时可以用到的API
module.exports = grunt => {
 // 配置选项，在这里面配置，下面的任务中使用grunt.config('foo') 来使用

 grunt.initConfig({
 foo: 'bar',
  css: 1
  })
  grunt.registerTask('default', () => {
  // 默认执行的任务
  // 后面跟数组，加任务的名字，就会默认执行数组里的任务
  console.log('default task ~')
  })
  // 异步任务
  grunt.registerTask('async-task', function () {
  const done = this.async()
  setTimeout( () => {
  console.log('async task working')
  // 异步中返回失败只要，done(false)就可以了
  done()
  }, 1000)
  })

  // 子啊一个任务返回错误的时候，后面的任务会被停止
  // --force就会强制执行，无视错误


  // 多目标模式，可以让任务根据配置形成多个子任务
  grunt.registerMultiTask('build', function () {

```

```
  // 这里可以执行对应的子任务对应的target和data
  // 执行的时候 grunt build:css
  console.log(`target: ${this.target},data : ${this.data}`)
  }))

  // 插件引用
  // 安装插件后，loadNpmTasks直接引入
  grunt.loadNpmTasks('grunt-contrib-clean')

  // 安装sass模块 grunt-sass sass
  // 插件都是多目标任务
  grunt.initConfig({
  foo: 'bar',
  css: 1
  })
}
```

实际项目

```
// 实现这个项目的构建任务

// grunt-contrib-clean
// grunt-sass
// grunt-babel @babel/core @babel/preset-env
// load-grunt-tasks  //自动加载所有的grunt插件中的任务
// grunt-contrib-watch // 监听插件

const loadGruntTasks = require("load-grunt-tasks");

 const sass = require("sass");
 module.exports = (grunt) => {
   grunt.initConfig({
     sass: {
       options: {
         sourceMap: true,
         implementation: sass,
       },
       main: {
         files: {
           "dist/assets/styles/main.css": "src/assets/styles/main.scss",
         },
```

```
      },
    },
    babel: {
      options: {
        // sourceMap: true,
        presets: ["@babel/preset-env"],
      },
      main: {
        files: {
          "dist/assets/scripts/main.js": "src/assets/scripts/*.js",
        },
      },
    },
    watch: {
      js: {
        files: ["src/assets/script/*.js"],
        tasks: ["babel"],
      },
      css: {
        files: ["src/assets/styles/*.scss"],
        tasks: ["sass"],
      },
    },
    swigtemplates: {
      options: {
        defaultContext: {
          pageTitle: "my title",
        },
        templatesDir: "src/*.html",
      },
      // production: {
      //     dest: 'dist/',
      //     src: ['src/*.html']
      // }
    },
  });
  //   grunt.loadNpmTasks("grunt-contrib-clean");
  //   grunt.loadNpmTasks("grunt-sass");
  loadGruntTasks(grunt); // 自动加载所有的grunt插件中的任务
};
```

```
63
```

Gulp

安装 gulp

创建gulpfile.js文件

```
// gulp 的入口文件

// exports 导出
// gulp在执行的时候，会读取目录下的gulpfile文件，然后调用导出的函数名称
// 在调用时会传入一个done的标识，done执行的时候，就表示函数结束了
// 就是一种异步callback调用
exports.foo = (done) => {
  console.log("foo task working~");
  done();
};

// default 是默认执行的任务
// 在运行是可以省略任务名参数
exports.default = (done) => {
  console.log("default task working");
  done();
};

// v4.0 之前需要通过 gulp.task() 方法注册任务
const gulp = require("gulp");

gulp.task("bar", (done) => {
  console.log("bar task working~");
  done();
});

// 组合任务

const { series, parallel } = require("gulp");

const task1 = (done) => {
  setTimeout(() => {
    console.log("task1 working~");
    done();
  }, 1000);
```

```
};

const task2 = (done) => {
  setTimeout(() => {
    console.log("task2 working~");
    done();
  }, 1000);
};

const task3 = (done) => {
  setTimeout(() => {
    console.log("task3 working~");
    done();
  }, 1000);
};

// 让多个任务按照顺序依次执行
exports.foo = series(task1, task2, task3);

// 让多个任务同时执行
exports.bar = parallel(task1, task2, task3);

// 异步任务

exports.callback = (done) => {
  console.log("callback task");
  done();
};
// gulp遵循的是错误优先
// done返回错误会终止后面的任务执行
exports.callback_error = (done) => {
  console.log("callback task");
  done(new Error("task failed"));
};

// promise 方式
// 返回promise的方式，就可以不用上面的callback的方式了
// promise返回的值会忽略，检测到promise状态为resolve就表示结束了
exports.promise = () => {
  console.log("promise task");
  return Promise.resolve();
```

```
};
// 检测到promise状态为reject就表示报错了，也会遵循错误优先
exports.promise_error = () => {
  console.log("promise task");
  return Promise.reject(new Error("task failed"));
};

// 文件读取的形式
exports.stream = () => {
  // 读取一个yarn.lock文件的内容，并放入文件流池
  const read = fs.createReadStream("yarn.lock");
  // 创建一个a.txt的文件内容，也放入到文件流池
  const write = fs.createWriteStream("a.txt");
  // 将读到的数据导入到创建的流中
  read.pipe(write);
  // 然后返回
  return read;
};
// 下面的return实际到的事情
// exports.stream = done => {
//   const read = fs.createReadStream('yarn.lock')
//   const write = fs.createWriteStream('a.txt')
//   read.pipe(write)
//   read.on('end', () => {
//     done()
//   })
// }
```

Gulp 构建过程核心工作原理

```
const fs = require('fs')
const { Transform } = require('stream')

exports.default = () => {
  // 文件读取流
  const readStream = fs.createReadStream('normalize.css')

  // 文件写入流
  const writeStream = fs.createWriteStream('normalize.min.css')

```

```
  // 文件转换流
  const transformStream = new Transform({
  // 核心转换过程
  transform: (chunk, encoding, callback) => {
  const input = chunk.toString()
  const output = input.replace(/\s+/g, '').replace(/\/\*.+?\*\//g, '')
  callback(null, output)
  }
  })

  return readStream
  .pipe(transformStream) // 转换
  .pipe(writeStream) // 写入
}
```

Gulp文件操作API

插件gulp-clean-css --dev　用于压缩转换css文件

```
const { src, dest } = require('gulp')
const cleanCSS = require('gulp-clean-css')
const rename = require('gulp-rename')

exports.default = () => {
 // src 对目录的文件进行转译成流文件
 return src('src/*.css')
 // 对流文件进行编译压缩
 .pipe(cleanCSS())
 // 对另存为文件的扩展名进行更改
 .pipe(rename({ extname: '.min.css' }))
 // 将流文件写入成文件
 .pipe(dest('dist'))
}
```

完整压缩转换

```
const { src, dest, parallel, series, watch } = require('gulp')

const del = require('del')
const browserSync = require('browser-sync')
```

```
5
6  const loadPlugins = require('gulp-load-plugins')
7
8  const plugins = loadPlugins()
9  const bs = browserSync.create()
10
11  const data = {
12   menus: [
13   {
14   name: 'Home',
15   icon: 'aperture',
16   link: 'index.html'
17   },
18   {
19   name: 'Features',
20   link: 'features.html'
21   },
22   {
23   name: 'About',
24   link: 'about.html'
25   },
26   {
27   name: 'Contact',
28   link: '#',
29   children: [
30   {
31   name: 'Twitter',
32   link: 'https://twitter.com/w_zce'
33   },
34   {
35   name: 'About',
36   link: 'https://weibo.com/zceme'
37   },
38   {
39   name: 'divider'
40   },
41   {
42   name: 'About',
43   link: 'https://github.com/zce'
44   }
```

```
  ]
  }
  ],
  pkg: require('./package.json'),
  date: new Date()
}

const clean = () => {
  return del(['dist', 'temp'])
}

const style = () => {
  return src('src/assets/styles/*.scss', { base: 'src' })
  .pipe(plugins.sass({ outputStyle: 'expanded' }))
  .pipe(dest('temp'))
  .pipe(bs.reload({ stream: true }))
}

const script = () => {
  return src('src/assets/scripts/*.js', { base: 'src' })
  .pipe(plugins.babel({ presets: ['@babel/preset-env'] }))
  .pipe(dest('temp'))
  .pipe(bs.reload({ stream: true }))
}

const page = () => {
  return src('src/*.html', { base: 'src' })
  .pipe(plugins.swig({ data, defaults: { cache: false } })) // 防止模板缓存导致页面不能及时更新
  .pipe(dest('temp'))
  .pipe(bs.reload({ stream: true }))
}

const image = () => {
  return src('src/assets/images/**', { base: 'src' })
  .pipe(plugins.imagemin())
  .pipe(dest('dist'))
}

const font = () => {
  return src('src/assets/fonts/**', { base: 'src' })
```

```
  .pipe(plugins.imagemin())
  .pipe(dest('dist'))
}

const extra = () => {
 return src('public/**', { base: 'public' })
  .pipe(dest('dist'))
}

const serve = () => {
 watch('src/assets/styles/*.scss', style)
 watch('src/assets/scripts/*.js', script)
 watch('src/*.html', page)
 // watch('src/assets/images/**', image)
 // watch('src/assets/fonts/**', font)
  // watch('public/**', extra)
  watch([
  'src/assets/images/**',
  'src/assets/fonts/**',
  'public/**'
  ], bs.reload)

  bs.init({
  notify: false,
  port: 2080,
  // open: false,
  // files: 'dist/**',
  server: {
  baseDir: ['temp', 'src', 'public'],
  routes: {
  '/node_modules': 'node_modules'
  }
  }
  })
}

const useref = () => {
 return src('temp/*.html', { base: 'temp' })
  .pipe(plugins.useref({ searchPath: ['temp', '.'] }))
  // html js css
```

```
  .pipe(plugins.if(/\.js$/, plugins.uglify()))
  .pipe(plugins.if(/\.css$/, plugins.cleanCss()))
  .pipe(plugins.if(/\.html$/, plugins.htmlmin({
  collapseWhitespace: true,
  minifyCSS: true,
  minifyJS: true
  })))
  .pipe(dest('dist'))
}

const compile = parallel(style, script, page)

// 上线之前执行的任务
const build = series(
  clean,
  parallel(
  series(compile, useref),
  image,
  font,
  extra
  )
)

const develop = series(compile, serve)

module.exports = {
  clean,
  build,
  develop
}

```

在 package.json里面加上，让命令更加方便使用

```
  "scripts": {
    "clean": "gulp clean",
    "build": "gulp build",
    "develop": "gulp develop"
  },
```

封装工作流的准备工作

生产模板

```
yarn global add zce-cli
```

```
生产模板
zce init nm zce-pages

将项目加载到github上面

将原项目的开发依赖，放在封装项目的运行依赖中
然后删除原项目的开发依赖


通过link将项目发布到全局
yarn link 将当前文件夹发布到全局

yarn link "zce-pages" 在引用目录来引用这个目录

在引用目录的gulpfile.js 中
module.exports = require('zce-pages')
在gulpfile文件中导出zce-pages中入口文件
因为zce-pages中的入口文件就是gulpfile的配置文件内容，所以写在本目录是一样的效果

因为这是引入的外部的文件夹，所以node_modules目录中没有gulp这个插件，可以先安装一个
这两个问题一般就出现在link这种引用的模式下才会有，在发布之后，这些插件会自动安装在目录下
yarn add gulp-cli --dev
yarn add gulp --dev


gulp里面的data数据不应该放在gulpfile里面
常规的应该把data这种数据，放在pages.config.js里面

后面报错找不到@babel/preset-env,是因为他到构建目录找了，需要在原目录才有，所以用require导入就能找到了，因为require会从本地开始找

省略掉目录中需要创建的 gulpfile.js 文件

则需要在运行时加上额外的参数了
--gulpfile 是指定gulpfile所在的位置，
--cwd 是指定工作目录，. ,如果不指定的话，就会在lib/index.js当作工作目录，就会不受控制了
yarn gulp build --gulpfile ./node_modules/zce-pages/lib/index.js --cwd .

```

```
想要使用zce-pages自己给自己传后面需要的参数，
就要创建一个bin的入口文件zce-pages.js
在package.json 中需要表明bin这个入口 bin: bin/zce-pages.js
此文件里面必须要有 #!/usr/bin/env node 这个声明
process.argv.push('--cwd')
process.argv.push(process.cwd())
process.argv.push('--gulpfile')
// require是返回一个包，resolve是返回一个文件对应的路径
process.argv.push(require.resolve('..'))

然后需要重新link

在发布的时候，默认只会发布files里面的记录的文件，所以还要把bin目录加到files里面

发布 git 提交 上传

然后 yarn publish
--resgistry https://registry.yarnpkg.com
```