**snailfwd**

**规范：**

本地目录：

project/

project/ components/ (存放本地模块)

project/ components/模块名/

project/ component\_modules/ （存放远程下载的component模块，暂不考虑）

project/spm\_modules/（存放spm远程下载模块,默认从内部spm平台安装组件）

project/ components\_ modules /模块名/version/（暂不考虑）

project/spm\_modules/模块名/version/

project/views/ （存放模板资源）

project/static/ （存放非模板化资源）

project/static/lib/ （存放非模块js，如加载器等）

发布目录：

project/version/

project/ version/components/ (存放所有本地模块)

project/ version/components / 模块名/

project/ version/spm\_ modules /(存放所有从spm安装的模块)

project/ version/ spm \_ modules /模块名/version/

project/version/views/ （存放解析后的模块html）

project/version/static/ （存放非模板化资源）

project/version/static/lib/ （存放非模块js，如加载器等）

上线目录：

根据目前结构待定

**默认配置：**

设置name（项目名）

设置version（版本名）

设置roadmap.path为

1.md后缀的文件不发布

2. txt后缀的文件不发布

3. publish-bak文件夹不发布

4. test后缀的文件夹不发布

5.spm \_module，component\_module,目录下的文件发布到【项目名/ version/c/】目录下,并设置isComponentModules、useMap为true

6. component目录下的文件发布到【项目名/ version /c/】目录下,并设置isComponent、useMap为true

7. views目录下的文件发布到【项目名/ version /views/】目录下

8. static目录下的文件发布到【项目名/ version /static/】目录下

9.其他文件就不发布，不优化，不进行资源定位处理

**1.parser阶段**

1.使用snailfwd-parser-component插件解析views目录下的\*\*.tpl文件，解析<!--load('xxx')-->或者<!-- load ("xxx@version")-->为一个组件的引用,解析带有data-main属性的script标签，如果该标签有src则将其设在file对象的mainJs属性中，如果没有src属性则mainJs属性为“self”。把解析到组件名字提取出来放到一个数组里保存在该文件file对象的require components属性里，同时把匹配标签的的函数保存在file对象的components Reg属性中，以便以后检索该标签。

该结束后，ret.src里tpl文件的file对象里会多两个属性，如：

file.requireComponents =["xxx","xxx@version"]

file.mainJs=src

file. components Reg=function components LabelReg(name){

switch (name){

case '\*\*':

name='(.)\*';

break;

}

return new RegExp('<!--component\\(("|\')('+name+')("|\')\\)-->','g');

}

2.使用

**2.postprocessor阶段**

使用snailfwd-postprocessor-jswrapper插件对file.isComponents&&!isInitFile || file.isComponentModules&&!isInitFile || conf.wrapAll的文件进行amd包装，其中isInitFile是指该文件的文件名是否带有init，如果带有为true，否则false。

包装格式为：

'define(\'' + file.id + '\', function(require, exports, module){ \r\n' + content + ' \r\n});';

**3.postpackager阶段**

调用bulidSnailFwd函数，根据返回的ret, conf, settings, opt进行snailfwd数据重构。

1.建立一个components别名表componentsAliasMap对象，规则是该文件需在components或者component\_modules或者spm\_modules目录下，每个目录对应一个组件（空目录除外），目录名即默认为组件名，没有版本号的建立别名： 组件名：组件完整路径, 如果有版本号，则建立一个别名：组件名@version：组件完整路径。

componentsAliasMap={

"a": "a/",

"b@1.0.0": "b/1.0.0/"

}

2.建立一个组件结构表componentsMap对象，目前默认安装的是spm组件，所以默认spm\_modules里的组件就根据组件里的package.json去分析资源, spm包默认的功能定义是提供某个单一的功能，比如一个特定功能的js，一段特定的css（spm.main the only entry point of the package,default index.js, or could be set to index.css for a css-only package ），但是这次我们的构想是一个组件应该是一个html、css、js、图片的随机结合，所以需在他的包里做些扩展定义，spm.main定义保持不变，可以是js，可以是css，但不能是html，不然导致该组件在使用seajs加载的时候报错，凡是包里的名为index或与组件同名的html，css，init.js,解析为该包的html，js，init入口文件，spm.main指定的是给seajs加载时使用的模块。解析优先级：spm.main>index文件>组件名文件；spm\_modules里的组件代码可能有require('./pop.css'); 这样的代码，处理方式是，在snailfwd.js里容错掉这种seajs写法的样式加载，而采用系统编译后加入到\_\_COMPONENT\_CSS\_\_里。由于有优先级流程，所以必须分阶段解析组件资源：

2-1.循环找到所有组件的json文件，然后根据spm.main的值文件类型，把值存入相应的组件的相应属性里，并且要转换为相对于根目录的绝对路径(最开始没有/)。

2-2.循环所有组件，把index或组件名.html加入到该组件的html属性里，把index.init.js或组件名.init.js加入到init属性里，如果该组件的css属性为空，说明它的spm.main不是css文件，把index.css或组件名.css加入到css属性里，如果该组件的js为空，说明它的spm.main不是js文件，把index.js或组件名.js加入到js属性里，最后再去寻找该组件入口js的内部本地依赖，如果有则加入到组件的js属性里。

componentsMap={

'a':{

html:'/a/a.html'

css:'/a/a.css','/a/aa.css',

js:['/a/a.js','/a/aa.js'],

init:'/a/a.init.js',

deps:['b@version','c']//目前没有设这个属性，因为在解析js组件的时候会去计算所有依赖

},

'b@version':{

html:'/b/version/b.html'

css:'/b/version/a.css'

js:['/a/version/a.js'],

init:'/a/version/a.init.js',

deps:['d','c']

},

'c':{

html:'/c/c.html'

css:'/c/c.css',

js:[],

init:'',

deps:[]

},

'd@version':{

html:''

css:''

js:['/d/version/d.js'],

init:'',

deps:[]

}

}

3.建立一个别名表aliasMap对象，目的是给snailfwd识别模块别名用，只需提取componentsAliasMap里有js的组件和其别名即可

即该文件需在components或者component\_modules或者spm\_modules目录下的js（目前暂不支持component模块），

aliasMap={

"a": "a/a.js",

"b@1.0.0": "spm\_modules/b/1.0.0/b.js"，

"component/b@1.0.0": "component\_modules/b/1.0.0/b.js"

}

4.建立一个模板(现在模板定义为凡是有<!--load('\*\*')-->标签的文件都视为模板)组件依赖表templateDepsMap对象，记录每个模板用到的所有不重复组件资源地址（包括内部依赖组件），和初始化调用的记录。

templateDepsMap={

"static/a/a.css": {

"css": [

"base/0.0.1/base.css"

],

"js": [],

"init": [],

"pkg": {

"css": {

"path": ""

},

"js": {

"path": ""

}

},

"deps": [

"base@0.0.1"

]

},

'views/a.tpl':{

deps:['a','b@version','c'],

"css": [

"a/0.0.1/a.css",

"b/b.css"

],

"js": [

"a/0.0.1/a.js",

"a/0.0.1/aa.js",

"b/1.10.2/b.js",

"c/c.js",

"d/1.0.0/d.js"

],

"init": [

"a/0.0.1/a.init.js",

"b/b.init.js"

],

pkg:{

css:{

path:''

},

js:{  
 path:''

},

}

}

}

5.根据templateDepsMap找到每个模板对应的资源引用，然后把所有引用样式写到/static/pkg/模板名\_ components.css下，此处应留一个插件接口，可以对打包的组件css进行一些个性化处理和并校正样式里的相对路径资源引用为绝对路径，相对于tpl结构,目前接口是设在getFilesContents函数，通过给该函数是第二个参数传一个自定义编译函数数组去做自定义的编译，被传进去的函数会接受到文件的content和该文件的uri参数。所有模块写到/static/pkg/模板名\_ components.js里；

如果该模板是tpl文件，则替换模板里的\_\_COMPONENTS\_CSS\_\_为“/static/pkg/模板名\_ components.css”的样式引用，替换\_\_COMPONENTS\_JS\_\_为“/static/pkg/模板名\_ components.js”的脚本引用，通过页面里带有data-main的脚本标签的src去寻找\_\_COMPONENTS\_INIT\_\_和\_\_COMPONENTS\_ALIAS\_\_标签进行相应的替换，如果有data-main标签但是没有src则对当前页面进行\_\_COMPONENTS\_INIT\_\_，\_\_COMPONENTS\_ALIAS\_\_标签替换，如果没有data-main标签脚本则编译提示其加入。

如果该文件是css文件，则只替换相应的组件调用标签为相应组件的css。

目前暂不考虑组件内部html依赖的情况，css组件内部本地调用可以用fis自带的文件合并功能实现如：在css里写import url('aa.css?\_\_inline')即可，调用其他组件可以用snailfwd标签<!--load("aa")-->，就可以将其他组件css的加载到当前文件来, js依赖支持。

对于一个组件可能在页面里重复引用的情况，比如有两个bigEye，虽然结构可能有点不同（例如只是id不同），定义在各自的html里也不为过，但是样式可能可以公用，这时候，如果分成bigEye1和bigEye2，里面的样式也有bigEye1.css和bigEye2.css，但其实这两个样式几乎是一样的，难免代码显得冗余、不灵活，所以这种情况，可以考虑让bigEye1.css里直接可以引用bigEye2.css的代码。比如（@import ('BigEye2')）,同时bigEye1.css里还可以写一些自定义可能需要覆盖的样式。系统编译后，应该去掉样式里的@import ('BigEye2')标签，同时把BigEye2组件的样式，单独作为一个组件样式加入到\_\_COMPONENTS\_CSS\_\_，去掉@import ('BigEye2')的bigEye1.css作为bigEye1组件的样式加入到\_\_COMPONENTS\_CSS\_\_，该功能已经实现调用标签<!--load("aa")-->。

**模块生态**

**1.打包上传到组件库：**

可以把所有组件信息定义在组件目录的component.json里，目前能想到的字段应该有name，version，repo，description，keywords，html，css，scripts，initScripts，dependencies，license，doc。其中doc里可以定义一些该组件的文档信息，具体细节待定。上传到组件库时，组件平台会解析该包的component.json，和目录里的文件，保存该包到平台的服务器一个目录里，目录结构：

组件名/version/

组件名/version/dist/

组件名/version/doc/

组件名/version/dist/（存放包的该版本的原始文件）

组件名/version/doc/（存放包的该版本的文档文件，以及下面生成的demo页面）

把包的和组件同名的html，css，js（amd包装过后的js），init.js拼合到一个html模板里，当做这个组件的demo页面，和提供组件渲染图片的基础页面。然后根据dependencies去安装相应的依赖组件，因为暂不考虑组件的html和css依赖，所以只加载js依赖组件的amd包装后的代码去替换doc模板页面的\_\_COMPONENTS\_JS\_\_。同时doc模板页面应该可以设置相应的变量标识符，以便可以用组件的文档信息去自定义doc模板的一些信息，比如title之类的，当然doc模板页面应该自动引入snailfwd以便识别amd模块和组件正常运行。

**2.下载安装组件：**

根据项目依赖数组里的组件名和组件版本号去组件平台寻找对应的组件即可，只需下载该组件的dist目录到本地的components\_modules即可。

**上线**

**todo**

**内网上线：**

**todo**

**公网上线：**

**todo**

**待完善功能：**

1.可以用spm安装的模块

2.传压缩参数-o后，\_\_COMPONENTS\_INIT\_\_标签不见了。

3.多次编译的时候，一定要加上-c参数，不然有缓存，文件编译无效。

4.别名最好可以配，考虑纯js模块开发的情况。

5.模块内部的文件引用。

6.修改组件调用标签component为load

7.默认只有一个版本的模块别名后面不加版本号，但是这样可能导致下次安装新版本后，版本错乱。

8.spm上的模块，有些可能有require('.xx/'+name)这种写法的代码（比如：moment。js），这样的模块就无法在浏览器端用模块加载的方式使用，只能在服务器端的node环境使用。

9.支持domain设置功能

snialfwd init命令开发

des：初始化不同游戏的项目文件

1.请选择游戏名字？（列出所有游戏名字，然后让用户输入

）

九阴真经 大三国 。。。

\_gameName（等待输入)

2.项目类型？

官网 专题

\_projectype(等待输入)

然后开始根据条件初始化项目文件，把seo关键字、gameid生成进去，目录，index.html