

# FIS3 , 为你定制 的前端工程构建 工具

书栈(BookStack.CN)

# 目 录

致谢

README

介绍

安装

起步

- 构建

- 调试

- 内置语法

  - 内容嵌入

  - 定位资源

  - 声明依赖

请转移到 [Release](#) 页面

工作原理

初级使用

中级使用

高级使用

接口文档

- 命令行

- 配置

  - 配置接口

  - 配置属性

  - 常用配置

  - glob

- 内置插件及配置

- 自定义插件

从零开始

代码校验

FIS2 到 FIS3

资源合并

mock 假数据模拟

常用的插件列表

fis3 site

base-css

fis3/lib/fis.js

## 致谢

当前文档《FIS3，为你定制的前端工程构建工具》由 进击的皇虫 使用 书栈 (BookStack.CN) 进行构建，生成于 2018-07-08。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈 (BookStack.CN)，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：<http://www.bookstack.cn/books/fis3>

书栈官网：<http://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

分享，让知识传承更久远！感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

# README



## FIS3

npm v3.4.39

downloads 2k/month

build passing

coverage 88%

dependencies insecure

FIS3 面向前端的工程构建系统。解决前端工程中性能优化、资源加载（异步、同步、按需、预加载、依赖管理、合并、内嵌）、模块化开发、自动化工具、开发规范、代码部署等问题。

如果对FIS先有些了解，但理解不深的，可试着带着这句话去看文档

FIS3 会在配置文件中给文件添加相应属性，用于控制文件的编译、合并等各种操作；文件属性包括基本属性和插件属性，[详细请参考](#)

```
1. npm install -g fis3
```

如果 Node 版本低于 4.x 请安装旧版本

```
1. npm install -g fis3@3.4.36
```

## 文档

快速入门、配置、插件开发以及原理等文档 [doc/docs/INDEX.md](#)

## 例子

```
1. mkdir my-proj
2. cd my-proj
```

3. fis3 init
4. fis3 release
5. fis3 server start --type node

### *fis-conf.js* 的例子

```
1. // default settings. fis3 release
2.
3. // Global start
4. fis.match('*. {js,css}', {
5.   useHash: true
6. });
7.
8. fis.match('::image', {
9.   useHash: true
10. });
11.
12. fis.match('*.js', {
13.   optimizer: fis.plugin('uglify-js') // js 压缩
14. });
15.
16. fis.match('*.css', {
17.   optimizer: fis.plugin('clean-css') // css 压缩
18. });
19.
20. fis.match('*.png', {
21.   optimizer: fis.plugin('png-compressor') // png 图片压缩
22. });
23.
24. // Global end
25.
26. // default media is `dev`
27. fis.media('dev')
28.   .match('*', {
29.     useHash: false,
30.     optimizer: null
31.   });
32.
33. // extends GLOBAL config
34. fis.media('production');
```

## 其他例子

---

<https://github.com/fex-team/fis3-demo>

## 常用插件

---

### 优化类（插件属性：optimizer）

- [fis-optimizer-uglify-js](#) UglifyJS2 压缩插件
- [fis-optimizer-clean-css](#) CleanCss 压缩插件
- [fis-optimizer-png-compressor](#) PNG 压缩插件

### 预处理类（插件属性：parser）

- [fis-parser-less](#) less 解析插件
- [fis-parser-node-sass](#) sass / scss 解析插件
- [fis-parser-handlebars](#) handlebars 解析插件

## 来源(书栈小编注)

---

<https://github.com/fex-team/fis3>

# 介绍

## FIS3

---



### FIS3 是什么？

FIS3 是面向前端的工程构建工具。解决前端工程中性能优化、资源加载（异步、同步、按需、预加载、依赖管理、合并、内嵌）、模块化开发、自动化工具、开发规范、代码部署等问题。

# 安装

## 安装

本文档展示命令，如果是 *Windows* 请打开 *cmd* 输入命令执行，如果是类 *Unix* 系统，请打开任意终端输入命令执行。

## 安装 Node 和 NPM

详细过程参考官网 <https://nodejs.org>

*Node* 版本要求 *0.8.x*, *0.10.x*, *0.12.x*, *4.x*, *6.x*, 不在此列表中的版本不予支持。最新版本 *node* 支持会第一时间跟进，支持后更新支持列表。

- Ubuntu 用户使用 `apt-get` 安装 *node* 后，安装的程序名叫 `nodejs`，需要软链成 `node`
- Windows 用户安装完成后需要在 *CMD* 下确认是否能执行 *node* 和 *npm*

## 安装 FIS3

```
1. npm install -g fis3
```

- `-g` 安装到全局目录，必须使用全局安装，当全局安装后才能在命令行（*cmd*或者终端）找到 `fis3` 命令
- 安装过程中遇到问题具体请参考 [fis#565](#) 解决
- 如果已经安装了 *FIS*，也执行上面的命令进行安装，*FIS3* 和 *FIS* 是不同的构建工具，向下无法完全兼容。如果要从 *FIS* 迁移到 *FIS3*，请参考文档 [FIS 升级 FIS3](#)
- 如果 *npm* 长时间运行无响应，推荐使用 [cnpm](#) 来安装

安装完成后执行 `fis3 -v` 判断是否安装成功，如果安装成功，则显示类似如下信息：

```
1. $ fis3 -v
2.
3. [INFO] Currently running fis3 (/Users/Your/Dev/fis3/dev/fis3)
4.
5. v3.0.0
6.
7.  \\\\\\\\\\\\\\\\\\\\\\\ \\\\\\\\\\\\\\\\\\\ \\\\\\\\\\\\\\\\\\\
8.  \\\\/\\\\\\\\\\\\\\\\\\\\ \\\\\\\\\\\\\\\\\\\ \\\\\\\\\\\\\\\\\\\\\\\
9.  \\\\/ \\\\\\\\\\\\\\\\\\\ \\\\\\\\\\\\\\\\\\\ \\\\\\\\\\\\\\\\\\\
```



10.	\\\\\\\\\\\\\\\\\\\\	\\\\\\	\\///\\\\\\
11.	\\\\\\\\///	\\\\\\	\\///\\\\\\
12.	\\\\\\	\\\\\\	\\///\\\\\\
13.	\\\\\\	\\\\\\	/\\\\ \\//\\\\\\
14.	\\\\\\	/\\\\\\\\\\\\\\\\\\	\\///\\\\\\\\\\\\\\\\\\
15.	\\//	\\\\\\\\\\\\\\\\\\	\\\\\\\\\\\\\\\\\\

如果提示找不到 `fis3` 命令并且 `npm` 安装成功退出，请参考文档 [fis#565](#) 解决

如果windows下提示 `basedir=$(dirname "$(echo "$0" | sed -e 's,\\,/,g'))` 则找到

C:\Users\win-7\AppData\Roaming\npm下面的fis脚本删除，留下fis.cmd就好了（没有识别系统是不是Windows）

## 升级 FIS3

```
1. npm update -g fis3
```

或重装

```
1. npm install -g fis3
```

# 起步

- [构建](#)
- [调试](#)
- [内置语法](#)

# 构建

## 构建

由 `fis3-command-release` 插件提供构建能力

FIS3 的构建不会修改源码，而是会通过用户设置，将构建结果输出到指定的目录。

## 例子

在正式介绍 FIS3 功能之前，我们给定一个简单的例子，例子下载地址 [demo-simple](#)

## 命令

进入项目根目录，执行命令，进行构建。

项目根目录：FIS3 配置文件（默认 `fis-conf.js`）所在的目录为项目根目录。

```
1. fis3 release -d <path>
```

- `<path>` 任意目录
- `fis3 release -h` 获取更多参数

构建发布到项目目录的 `output` 目录下

```
1. fis3 release -d ./output
```

构建发布到项目父级目录的 `dist` 子目录下

```
1. fis3 release -d ../dist
```

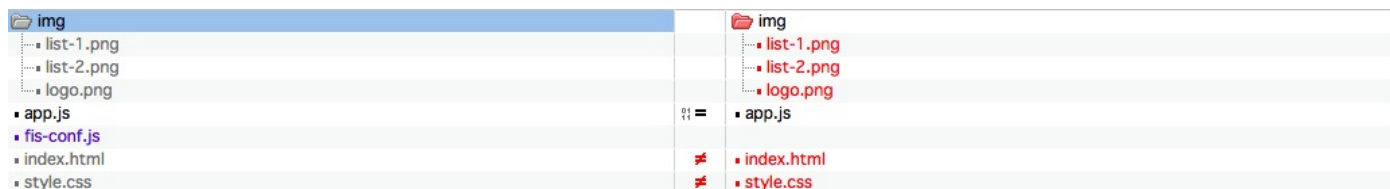
发布到其他盘（Windows）

```
1. fis3 release -d D:\output
```

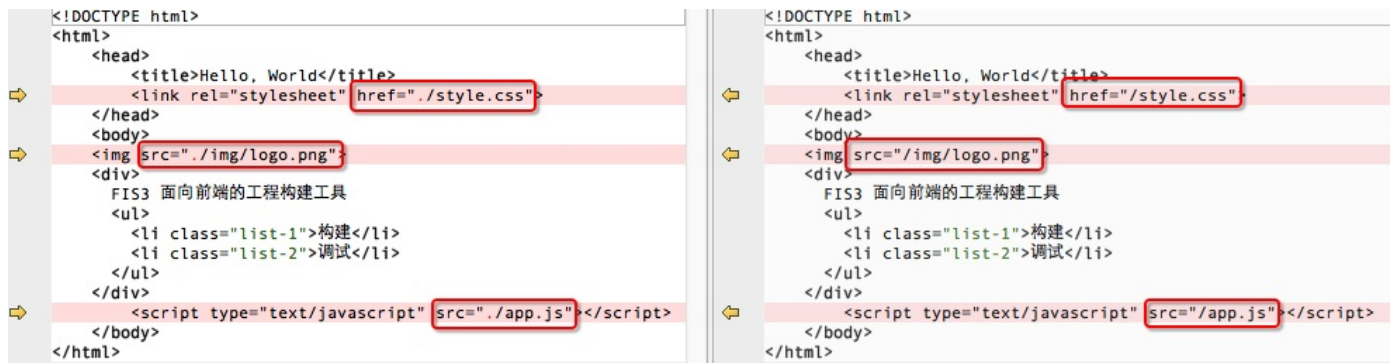
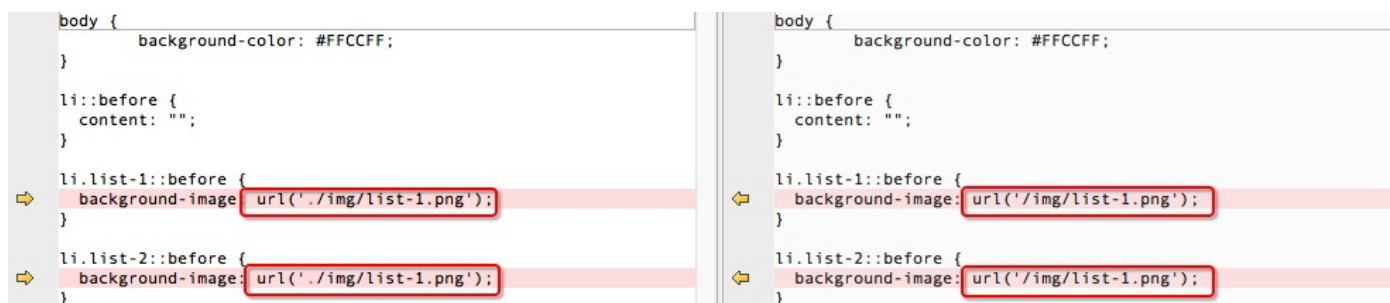
## 资源定位

我们在项目根目录执行命令 `fis3 release -d ../output` 发布到目录 `../output` 下。然后通过 `diff` 工具查看源码和构建结果的内容变化。

## 文件变化



- `index.html`、`style.css` 发生了变化

`index.html``style.css`

如上，构建过程中对资源 URI 进行了替换，替换成了绝对 URL。通俗点讲就是相对路径换成了绝对路径。

这是一个 FIS 的很重要的特性，[资源定位](#)。

资源定位能力，可以有效地分离开发路径与部署路径之间的关系，工程师不再关心资源部署到线上之后去了哪里，变成了什么名字，这些都可以通过配置来指定。而工程师只需要使用相对路径来定位自己的开发资源即可。这样的好处是 资源可以发布到任何静态资源服务器的任何路径上，而不用担心线上运行时找不到它们，而且代码 具有很强的可移植性，甚至可以从一个产品线移植到另一个产品线而不用担心线上部署不一致的问题。

在默认不配置的情况下只是对资源相对路径修改成了绝对路径。通过配置文件可以轻松分离开发路径（源码路径）与部署路径。比如我们想让所有的静态资源构建后到 `static` 目录下。

```

1. // 配置配置文件，注意，清空所有的配置，只留下以下代码即可。
2. fis.match('*.{png,js,css}', {
3.   release: '/static/$0'
4. });

```

同样构建到 `../output` 目录下看变化。

```
1. fis3 release -d ../output
```



以上示例只是更改部署路径，还可以给 `url` 添加 `CDN domain` 或者添加文件指纹（时间戳或者md5戳）。

再次强调，FIS3 的构建是不会对源码做修改的，而是构建产出到了另外一个目录，并且构建的结果才是用来上线使用的。

可能有人会疑惑，修改成了绝对路径，本地如何调试开发？下一节介绍调试的方法。

## 配置文件

默认配置文件为 `fis-conf.js`，FIS3 编译的整个流程都是通过配置来控制的。FIS3 定义了一种类似 CSS 的**配置方式**。固化了构建流程，让工程构建变得简单。

### `fis.match()`

首先介绍设置规则的配置接口

```
1. fis.match(selector, props);
```

- `selector`：FIS3 把匹配文件路径的路径作为selector，匹配到的文件会分配给它设置的 `props`。关于 selector 语法，请参看 [Glob 说明](#)
- `props`：编译规则属性，包括文件属性和插件属性，[更多属性](#)

我们修改例子的配置文件 `fis-conf.js`，添加以下内容

```

1.  fis.match('*.js', {
2.    useHash: false
3.  });
4.
5.  fis.match('*.css', {
6.    useHash: false
7.  });
8.
9.  fis.match('*.png', {
10.   useHash: false
11. });

```

## 重要特性

- 规则覆盖

假设有两条规则 A 和 B，它俩同时命中了文件 `test.js`，如果 A 在 B 前面，B 的属性会覆盖 A 的同名属性。不同名属性追加到 `test.js` 的 File 对象上。

```

1.  // A
2.  fis.match('*', {
3.    release: '/dist/$0'
4.  });
5.
6.  // B
7.  fis.match('test.js', {
8.    useHash: true,
9.    release: '/dist/js/$0'
10. })

```

那么 `test.js` 分配到的属性

```

1.  {
2.    useHash: true, // B
3.    release: '/dist/js/$0' // B
4.  }

```

## fis.media()

`fis.media()` 接口提供多种状态功能，比如有些配置是仅供开发环境下使用，有些则是仅供生产环境使用的。

```

1. fis.match('*', {
2.   useHash: false
3. });
4.
5. fis.media('prod').match('*.js', {
6.   optimizer: fis.plugin('uglify-js')
7. });

```

```
1. fis3 release <media>
```

- `<media>` 配置的 media 值

```
1. fis3 release prod
```

编译时使用 prod 指定的编译配置，即对 js 进行压缩。

如上，fis.media() 可以使配置文件变为多份（多个状态，一个状态一份配置）。

```

1. fis.media('rd').match('*', {
2.   deploy: fis.plugin('http-push', {
3.     receiver: 'http://remote-rd-host/receiver.php'
4.   })
5. });
6.
7. fis.media('qa').match('*', {
8.   deploy: fis.plugin('http-push', {
9.     receiver: 'http://remote-qa-host/receiver.php'
10.  })
11. });

```

- `fis3 release rd` push 到 RD 的远端机器上
- `fis3 release qa` push 到 QA 的远端机器上

`media` `dev` 已经被占用，默认情况下不加 `<media>` 参数时默认为 `dev`

## 更多配置接口

我们执行 `fis3 inspect` 来查看文件命中属性的情况。`fis3 inspect` 是一个非常重要的命令，可以查看文件分配到的属性，这些属性决定了文件将如何被编译处理。

```

1. ~ /app.js
2. -- useHash false `*.js` (0th)

```

```
3.
4.
5.  ~ /img/list-1.png
6.  -- useHash false `*.png`    (2th)
7.
8.
9.  ~ /img/list-2.png
10. -- useHash false `*.png`    (2th)
11.
12.
13. ~ /img/logo.png
14. -- useHash false `*.png`    (2th)
15.
16.
17. ~ /style.css
18. -- useHash false `*.css`    (1th)
19.
20.
21. ~ ::package
22. -- empty
```

`fis3 inspect <media>`查看特定 *media* 的分配情况

## 更多信息

- [常用配置](#)
- [配置属性](#)

## 文件指纹

文件指纹，唯一标识一个文件。在开启强缓存的情况下，如果文件的 URL 不发生变化，无法刷新浏览器缓存。一般都需要通过一些手段来强刷缓存，一种方式是添加时间戳，每次上线更新文件，给这个资源文件的 URL 添加上时间戳。

```
1. 
```

而 FIS3 选择的是添加 MD5 戳，直接修改文件的 URL，而不是在其后添加 `query`。

对 js、css、png 图片引用 URL 添加 md5 戳，配置如下；

```
1. //清除其他配置，只剩下如下配置
2. fis.match('*.{js,css,png}', {
3.   useHash: true
```

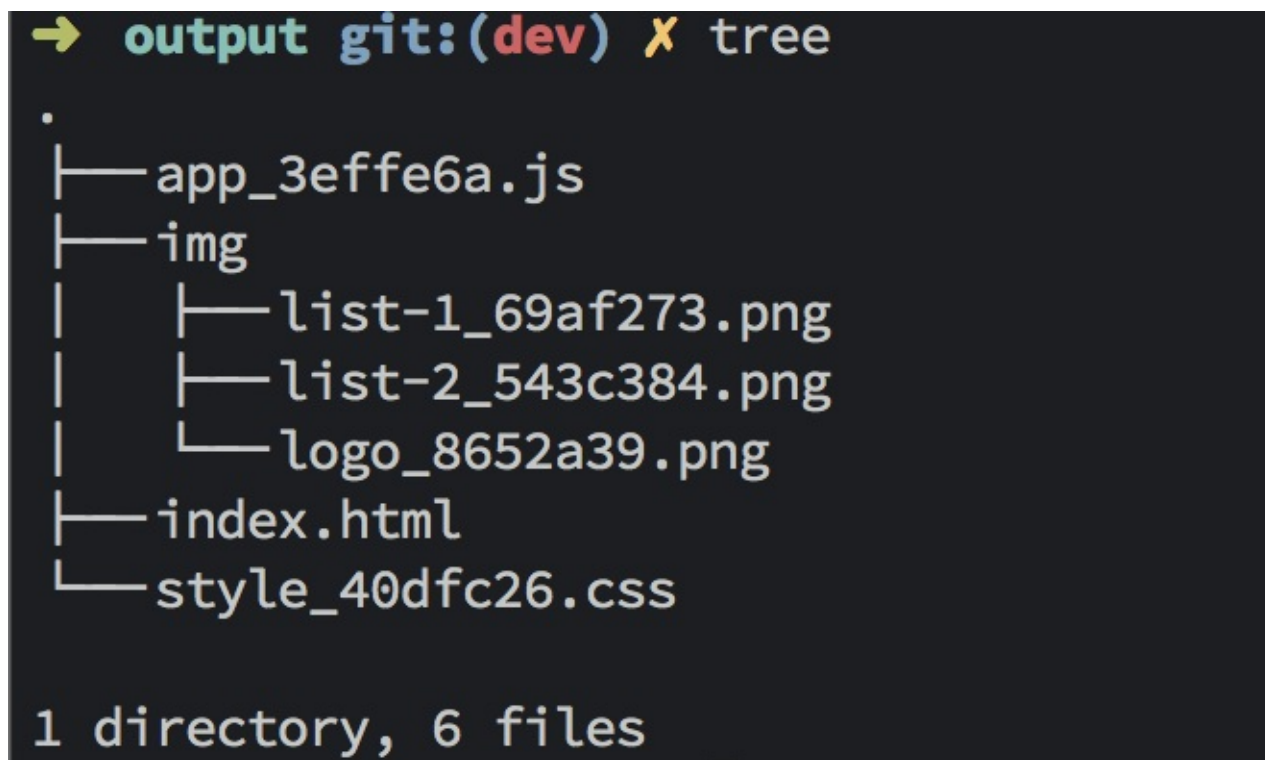


```
4. });
```

构建到 `../output` 目录下看变化。

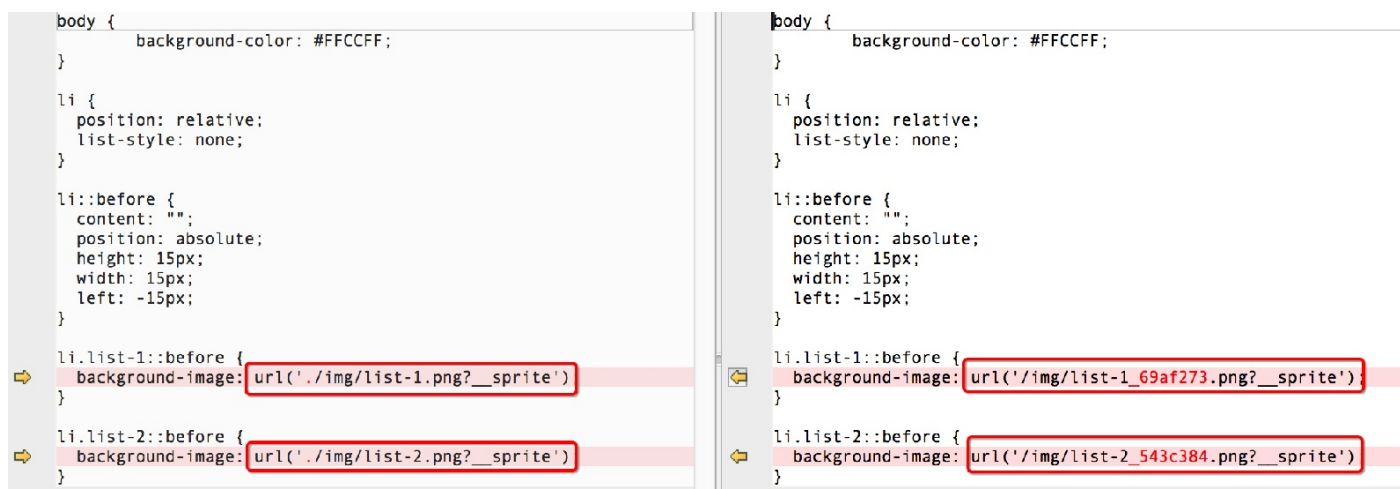
```
1. fis3 release -d ../output
```

文件变化



- 构建出的文件携带了 md5 戳

文件变化



- 对应 url 也带上了 md5 戳

## 片段编译

有些插件会对文件中的一部分先进行片段编译 `fis.compile.partial`，这时可以对相应的片段编译配置对应的规则。

```
1. // vue组件中的less片段处理
2. fis.match('src/vue/**/*.vue:less', {
3.   rExt: 'css',
4.   parser: fis.plugin('less'),
5.   release: 'xxx' // 这个无效
6. });
7.
8. // 注意：因为组件中的样式片段编译只是编译内容，所以上面的release配置是无效的。要配置其
   release，需要针对生成的css文件：
9. fis.match('src/vue/(**.css)', {
10.   release: '/vue-style/$1'
11. });
```

## 压缩资源

为了减少资源网络传输的大小，通过压缩器对 js、css、图片进行压缩是一直以来前端工程优化的选择。在 FIS3 中这个过程非常简单，通过给文件配置压缩器即可。

```
1. // 清除其他配置，只保留如下配置
2. fis.match('*.js', {
3.   // fis-optimizer-uglify-js 插件进行压缩，已内置
4.   optimizer: fis.plugin('uglify-js')
5. });
6.
7. fis.match('*.css', {
8.   // fis-optimizer-clean-css 插件进行压缩，已内置
9.   optimizer: fis.plugin('clean-css')
10. });
11.
12. fis.match('*.png', {
13.   // fis-optimizer-png-compressor 插件进行压缩，已内置
14.   optimizer: fis.plugin('png-compressor')
15. });
```

构建到 `../output` 目录下看变化。

```
1. fis3 release -d ../output
```

查看 `../output` 目录下已经被压缩过的结果。

## CssSprite图片合并

压缩了静态资源，我们还可以对图片进行合并，来减少请求数量。

FIS3 提供了比较简易、使用方便的图片合并工具。通过配置即可调用此工具并对资源进行合并。

FIS3 构建会对 CSS 中，路径带 `?__sprite` 的图片进行合并。为了节省编译的时间，分配到 `useSprite: true` 的 CSS 文件才会被处理。

默认情况下，对打包 `css` 文件启动图片合并功能。

```
1. li.list-1::before {
2.   background-image: url('../img/list-1.png?__sprite');
3. }
4.
5. li.list-2::before {
6.   background-image: url('../img/list-2.png?__sprite');
7. }
```

```
1. // 启用 fis-spriter-csssprites 插件
2. fis.match('::package', {
3.   spriter: fis.plugin('csssprites')
4. })
5.
6. // 对 CSS 进行图片合并
7. fis.match('*.css', {
8.   // 给匹配到的文件分配属性 `useSprite`
9.   useSprite: true
10. });
```

- CssSprites 详细配置参见 [fis-spriter-csssprites](#)

## 功能组合

我们学习了如何用 FIS3 做压缩、文件指纹、图片合并、资源定位，现在把这些功能组合起来，配置文件如下；

```
1. // 加 md5
```

```
2.  fis.match('*.js,css,png', {
3.    useHash: true
4.  });
5.
6.  // 启用 fis-spriter-csssprites 插件
7.  fis.match('/:package', {
8.    spriter: fis.plugin('csssprites')
9.  });
10.
11. // 对 CSS 进行图片合并
12. fis.match('*.css', {
13.   // 给匹配到的文件分配属性 `useSprite`
14.   useSprite: true
15. });
16.
17. fis.match('*.js', {
18.   // fis-optimizer-uglify-js 插件进行压缩, 已内置
19.   optimizer: fis.plugin('uglify-js')
20. });
21.
22. fis.match('*.css', {
23.   // fis-optimizer-clean-css 插件进行压缩, 已内置
24.   optimizer: fis.plugin('clean-css')
25. });
26.
27. fis.match('*.png', {
28.   // fis-optimizer-png-compressor 插件进行压缩, 已内置
29.   optimizer: fis.plugin('png-compressor')
30. });
```

- `fis3 release` 时添加 md5、静态资源压缩、css 文件引用图片进行合并

可能有时候开发的时候不需要压缩、合并图片、也不需要 hash。那么给上面配置追加如下配置；

```
1.  fis.media('debug').match('*.js,css,png', {
2.    useHash: false,
3.    useSprite: false,
4.    optimizer: null
5.  })
```

- `fis3 release debug` 启用 `media debug` 的配置，覆盖上面的配置，把诸多功能关掉。

## 调试

## 调试

FIS3 构建后，默认情况下会对资源的 URL 进行修改，改成绝对路径。这时候本地双击打开文件是无法正常工作的。这给开发调试带来了绝大的困惑。

FIS3 内置一个简易 Web Server，可以方便调试构建结果。

## 目录

构建时不指定输出目录，即不指定 `-d` 参数时，构建结果被发送到内置 Web Server 的根目录下。此目录可以通过执行以下命令打开。

```
1. fis3 server open
```

## 发布

```
1. fis3 release
```

不加 `-d` 参数默认被发布到内置 Web Server 的根目录下，当启动服务时访问此目录下的资源。

## 启动

通过

```
1. fis3 server start
```

来启动本地 Web Server，当此 Server 启动后，会自动浏览器打开

`http://127.0.0.1:8080`，默认监听端口 `8080`

通过执行以下命令得到更多启动参数，可以设置不同的端口号（当 `8080` 占用时）

```
1. fis3 server -h
```

## 预览

启动 Web Server 以后，会自动打开浏览器，访问 `http://127.0.0.1:8080` URL，这时即可查看到页面渲染结果。正如所有其他 Web Server，FIS3 内置的 Server 是常驻的，如果不重启计算

机或者调用命令关闭是不会关闭的。

所以后续只需访问对应链接即可，而不需要每次 `release` 就启动一次 `server`。

## 文件监听

为了方便开发，FIS3 支持文件监听，当启动文件监听时，修改文件会构建发布。而且其编译是增量的，编译花费时间少。

FIS3 通过对 `release` 命令添加 `-w` 或者 `--watch` 参数启动文件监听功能。

```
1. fis3 release -w
```

添加 `-w` 参数时，程序不会执行终止；停止程序用快捷键 `CTRL + C`

## 浏览器自动刷新

文件修改自动构建发布后，如果浏览器能自动刷新，这是一个非常好的开发体验。

FIS3 支持浏览器自动刷新功能，只需要给 `release` 命令添加 `-L` 参数，通常 `-w` 和 `-L` 一起使用。

```
1. fis3 release -wL
```

程序停止用快捷键 `CTRL + C`

## 发布到远端机器

当我们开发项目后，需要发布到测试机（联调机），一般可以通过如 SMB、FTP 等上传代码。FIS3 默认支持使用 HTTP 上传代码，首先需要在测试机部署上传接收脚本（或者服务），这个脚本非常简单，现在给出了 [php 的实现版本](#)，可以把它放到测试机上某个 Web 服务根目录，并且配置一个 url 能访问到即可。

注意：此代码存在很大的安全隐患，没有做任何安全考虑，请不要部署到线上服务。

百度内部请使用：<http://agroup.baidu.com/fis/md/article/196978>

示例脚本是 `php` 脚本，测试机 `web` 需要支持 `PHP` 的解析

如果需要其他语言实现，请参考这个 `php` 脚本实现，如果嫌麻烦，我们提供了一个 `node` 版本的[接收端](#)

假定这个 URL 是：`http://cq.01.p.p.baidu.com:8888/receiver.php`

那么我们只需要在配置文件配置

```

1.  fis.match('*', {
2.    deploy: fis.plugin('http-push', {
3.      receiver: 'http://cq.01.p.p.baidu.com:8888/receiver.php',
4.      to: '/home/work/htdocs' // 注意这个是指的是测试机器的路径，而非本地机器
5.    })
6.  })

```

如果你想通过其他协议上传代码，请参考 [deploy 插件开发](#) 实现对应协议插件即可。

- 当执行 `fis3 release` 时上传测试机器

可能上传测试机是最后联调时才会有的，更好的做法是设置特定 `media`。

```

1.  // 其他配置
2.  ...
3.  fis.media('qa').match('*', {
4.    deploy: fis.plugin('http-push', {
5.      receiver: 'http://cq.01.p.p.baidu.com:8888/receiver.php',
6.      to: '/home/work/htdocs' // 注意这个是指的是测试机器的路径，而非本地机器
7.    })
8.  });

```

- `fis3 release qa` 上传测试机器
- `fis3 release` 产出到本地测试服务器根目录

## 替代内置Server

FIS3 内置了一个 Web Server 提供给构建后的代码进行调试。如果你自己启动了你自己的 Web Server 依然可以使用它们。

假设你的 Web Server 的根目录是 `/Users/my-name/work/htdocs`，那么发布时只需要设置产出目录到这个目录即可。

```

1.  fis3 release -d /Users/my-name/work/htdocs

```

如果想执行 `fis3 release` 直接发布到此目录下，可在配置文件配置；

```

1.  fis.match('*', {
2.    deploy: fis.plugin('local-deliver', {
3.      to: '/Users/my-name/work/htdocs'
4.    })
5.  })

```

# 内置语法

## 内置语法

FIS 项目曾经历了很久的“努力做好编译工具”的时代。那段时间里，FIS 走了很多弯路，那时我们认为前端领域需要很复杂的编译工具才能很好的处理各种开发需求。2013年初，FIS 的编译工具非常庞大复杂，日益暴露出来的问题已经开始不再收敛了，这促使 FIS 小组重新审视 FIS 的编译系统：满足前端开发需求的最小编译规则集是什么？

前端编译工具有必要那么复杂么？答案是 完全没必要！想象一下尺规作图，一把直尺，一只圆规，就可以做出很多基本几何操作。经过 FIS 团队不断实践总结，我们发现支持前端开发所需要的编译能力只有三种：

- 资源定位：获取任何开发中所使用资源的线上路径；
- 内容嵌入：把一个文件的内容(文本)或者 base64 编码(图片)嵌入到另一个文件中；
- 依赖声明：在一个文本文件内标记对其他资源的依赖关系；

一套前端编译工具，只要实现上述3项编译能力，就可以变得非常易用，代码可维护性瞬间提高很多。

这三种编译能力作为 FIS 的内置语法提供：

- [资源定位](#)
- [内容嵌入](#)
- [依赖声明](#)

内置语法主要针对 html、css、js 等三种语言提供不同的编译语法。假设遇到后端模板、异构语言、前端模板等如何让内置语法起效呢？

```
1. // FIS 中前端模板推荐预编译为 js, 所以应该使用 js 的内置语法
2. fis.match('*.tmpl', {
3.   isJsLike: true
4. });
```

```
1. fis.match('*.sass', {
2.   isCssLike: true
3. });
```

```
1. fis.match('*.xxhtml', {
2.   isHtmlLike: true
3. });
```



# 内容嵌入

## 嵌入资源

嵌入资源即内容嵌入，可以为工程师提供诸如图片base64嵌入到css、js里，前端模板编译到js文件中，将js、css、html拆分成几个文件最后合并到一起的能力。有了这项能力，可以有效的减少http请求数，提升工程的可维护性。fis不建议用户使用内容嵌入能力作为组件化拆分的手段，因为[声明依赖](#)能力会更适合组件化开发。

### 在html中嵌入资源

在html中可以嵌入其他文件内容或者base64编码值，可以在资源定位的基础上，给资源加 `__inline` 参数来标记资源嵌入需求。

- html中嵌入图片base64

- 源码

```
1. 
```

- 编译后

```
1. 
```

- html中嵌入样式文件

- 源码

```
1. <link rel="stylesheet" type="text/css" href="demo.css?__inline">
```

- 编译后

```
1. <style>img { border: 5px solid #ccc; }</style>
```

- html中嵌入脚本资源

- 源码

```
1. <script type="text/javascript" src="demo.js?__inline"></script>
```

- 编译后

```
1. <script type="text/javascript">console.log('inline file');</script>
```

- html中嵌入页面文件

- 源码（推荐使用）

```
1. <link rel="import" href="demo.html?__inline">
```

- 编译后

```
1. <!-- this is the content of demo.html -->
2. <h1>demo.html content</h1>
```

- 源码（功能同<link ref="import" href="xxx?\_\_inline">语法，此语法为旧语法，不推荐使用）

```
1. <!--inline[demo.html]-->
```

- 编译后

```
1. <!-- this is the content of demo.html -->
2. <h1>demo.html content</h1>
```

## 在js中嵌入资源

在js中，使用编译函数 `__inline()` 来提供内容嵌入能力。可以利用这个函数嵌入图片的base64编码、嵌入其他js或者前端模板文件的编译内容， 这些处理对html中**script**标签里的内容同样有效。

- 在js中嵌入js文件

- 源码

```
1. __inline('demo.js');
```

- 编译后

```
1. console.log('demo.js content');
```

- 在js中嵌入图片base64

- 源码

```
1.   var img = __inline('images/logo.gif');
```

- 编译后

```
1.   var img =  
    'data:image/gif;base64,R0lGODlhDgGBALMAAGBn6eYxLvvy9PnKyf0...Jzna6853wj
```

- 在js中嵌入其他文本文件

- 源码

```
1.   var css = __inline('a.css');
```

- 编译后

```
1.   var css = "body \n{    color: red;\n}";
```

## 在css中嵌入资源

与html类似，凡是命中了资源定位能力的编译标记，除了**src="xxx"**之外，都可以通过添加 ? **\_\_inline** 编译标记都可以把文件内容嵌入进来。src="xxx"被用在ie浏览器支持的filter内，该属性不支持base64字符串，因此未做处理。

- 在css文件中嵌入其他css文件

- 源码

```
1.   @import url('demo.css?__inline');
```

- 编译后

```
1.   img { border: 5px solid #ccc; };
```

- 在css中嵌入图片的base64

- 源码

```
1.  .style {  
2.      background: url(images/logo.gif?__inline);  
3.  }
```

。编译后

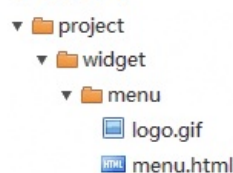
```
1.  .style {  
2.      background:  
    url(data:image/gif;base64,R0lGODlhDgGBALMAAGBn6eYxLvvy9PnKyf0...Jzna685  
3.  }
```

## 定位资源

## 定位资源

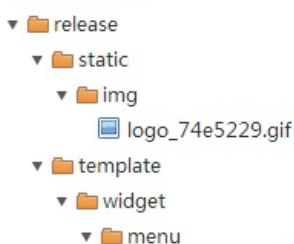
定位资源能力，可以有效地分离开发路径与部署路径之间的关系，工程师不再关心资源部署到线上之后去了哪里，变成了什么名字，这些都可以通过配置来指定。而工程师只需要使用相对路径来定位自己的开发资源即可。这样的好处是：资源可以发布到任何静态资源服务器的任何路径上而不用担心线上运行时找不到它们，而且代码具有很强的可移植性，甚至可以从一个产品线移植到另一个产品线而不用担心线上部署不一致的问题。

### ✓ 开发中：



```
6 
```

### ✓ 上线后：



```
6 
```

资源定位

```
//配置文件
fis.config.merge({
  roadmap {
    path : [
      ...
      {
        reg : 'widget/**/*.gif',
        release : '/static/img/$&'
      },
      ...
    ]
  }
});
```

## 在html中定位资源

FIS3 支持对html中的script、link、style、video、audio、embed等标签的src或href属性进行分析，一旦这些标签的资源定位属性可以命中已存在文件，则把命中文件的url路径替换到属性中，同时可保留原来url中的query查询信息。

例如：

1. `<!-- 源码：`
2. ``
3. `编译后-->`
4. ``
- 5.

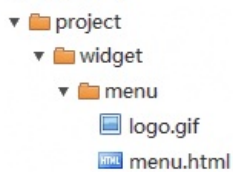
```

6. <!--源码:
7. <link rel="stylesheet" type="text/css" href="demo.css">
8. 编译后-->
9. <link rel="stylesheet" type="text/css" href="/demo_7defa41.css">
10.
11. <!--源码:
12. <script type="text/javascript" src="demo.js"></script>
13. 编译后-->
14. <script type="text/javascript" src="/demo_33c5143.js"></script>

```

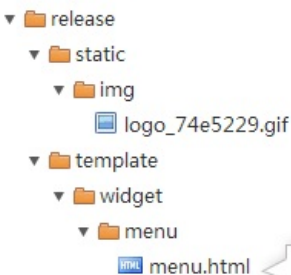
值得注意的是，资源定位结果可以被fis的配置文件控制，比如添加配置，调整文件发布路径：

## ✓ 开发中：

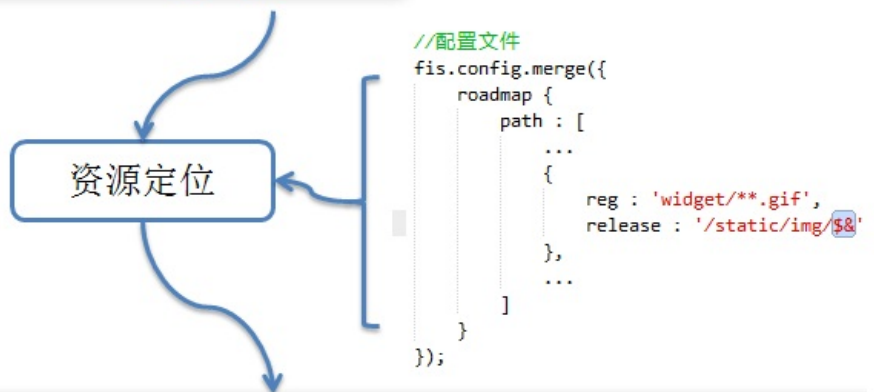


6 

## ✓ 上线后：



6 



```

1. fis.match('*. {js,css,png,gif}', {
2.   useHash: true // 开启 md5 戳
3. });
4.
5. // 所有的 js
6. fis.match('*.js', {
7.   //发布到/static/js/xxx目录下
8.   release : '/static/js$0'
9. });
10.
11. // 所有的 css
12. fis.match('*.css', {
13.   //发布到/static/css/xxx目录下

```

```

14.     release : '/static/css$0'
15. });
16.
17. // 所有image目录下的.png, .gif文件
18. fis.match('/images/(.*.{png,gif})', {
19.     //发布到/static/pic/xxx目录下
20.     release: '/static/pic/$1$2'
21. });

```

再次编译得到：

```

1.  <!--源码：
2.  
3.  编译后-->
4.  
5.
6.  <!--源码：
7.  <link rel="stylesheet" type="text/css" href="demo.css">
8.  编译后-->
9.  <link rel="stylesheet" type="text/css" href="/static/css/demo_7defa41.css">
10.
11. <!--源码：
12. <script type="text/javascript" src="demo.js"></script>
13. 编译后-->
14. <script type="text/javascript" src="/static/js/demo_33c5143.js"></script>

```

我们甚至可以让 **url**和发布目录不一致。比如：

```

1.
2. fis.match('*.{js,css,png,gif}', {
3.     useHash: true // 开启 md5 戳
4. });
5.
6. // 所有的 js
7. fis.match('*.js', {
8.     //发布到/static/js/xxx目录下
9.     release : '/static/js$0',
10.    //访问url是/mm/static/js/xxx
11.    url : '/mm/static/js$0'
12. });
13.
14. // 所有的 css

```

```

15.  fis.match('**.*css', {
16.      //发布到/static/css/xxx目录下
17.      release : '/static/css$0',
18.      //访问url是/pp/static/css/xxx
19.      url : '/pp/static/css$0'
20.  });
21.
22.  // 所有image目录下的.png, .gif文件
23.  fis.match('/images/(.*.{png,gif})', {
24.      //发布到/static/pic/xxx目录下
25.      release: '/static/pic/$1',
26.      //访问url是/oo/static/baidu/xxx
27.      url : '/oo/static/baidu$0'
28.  });

```

再次编译得到：

```

1.  <!--源码：
2.  
3.  编译后-->
4.  
5.
6.  <!--源码：
7.  <link rel="stylesheet" type="text/css" href="demo.css">
8.  编译后-->
9.  <link rel="stylesheet" type="text/css" href="/pp/static/css/demo_7defa41.css">
10.
11. <!--源码：
12. <script type="text/javascript" src="demo.js"></script>
13. 编译后-->
14. <script type="text/javascript" src="/mm/static/js/demo_33c5143.js"></script>

```

## 在js中定位资源

js语言中，可以使用编译函数 `__uri(path)` 来定位资源，fis分析js文件或html中的script标签内容时会替换该函数所指向文件的线上url路径。

- 源码：

```
1.  var img = __uri('images/logo.gif');
```

- 编译后



```
1.   var img = '/images/logo_74e5229.gif';
```

- 源码:

```
1.   var css = __uri('demo.css');
```

- 编译后

```
1.   var css = '/demo_7defa41.css';
```

- 源码:

```
1.   var js = __uri('demo.js');
```

- 编译后

```
1.   var js = '/demo_33c5143.js';
```

资源定位结果可以被fis的配置文件控制，比如添加配置，调整文件发布路径：

```
1.  fis.match('*.js,css,png,gif', {
2.      useHash: true // 开启 md5 戳
3.  });
4.
5.  // 所有的 js
6.  fis.match('*.js', {
7.      //发布到/static/js/xxx目录下
8.      release : '/static/js$0'
9.  });
10.
11. // 所有的 css
12. fis.match('*.css', {
13.     //发布到/static/css/xxx目录下
14.     release : '/static/css$0'
15. });
16.
17. // 所有image目录下的.png, .gif文件
18. fis.match('/images/(*.png,gif)', {
19.     //发布到/static/pic/xxx目录下
20.     release: '/static/pic/$1'
21. });
```

再次编译得到：

- 源码：

```
1.    var img = __uri('images/logo.gif');
```

- 编译后

```
1.    var img = '/static/pic/logo_74e5229.gif';
```

- 源码：

```
1.    var css = __uri('demo.css');
```

- 编译后

```
1.    var css = '/static/css/demo_7defa41.css';
```

- 源码：

```
1.    var js = __uri('demo.js');
```

- 编译后

```
1.    var js = '/static/js/demo_33c5143.js';
```

## 在css中定位资源

fis编译工具会识别css文件或 **html**的**style**标签内容 中 **url(path)** 以及 **src=path** 字段，并将其替换成对应资源的编译后url路径

- 源码：

```
1.    @import url('demo.css');
```

- 编译后

```
1.    @import url('/demo_7defa41.css');
```

- 源码：

```

1.  .style {
2.      background: url('images/body-bg.png');
3.  }

```

- 编译后


```

1.  .style {
2.      background: url('/images/body-bg_1b8c3e0.png');
3.  }

```

- 源码:


```

1.  .style {
2.      _filter 
        DXImageTransform.Microsoft.AlphaImageLoader(src='images/body-bg.png');
3.  }

```

- 编译后

```

1.  .style {
2.      _filter 
        DXImageTransform.Microsoft.AlphaImageLoader(src='/images/body-
        bg_1b8c3e0.png');
3.  }

```

资源定位结果可以被fis的配置文件控制，比如添加配置，调整文件发布路径：

```

1.
2.  fis.match('*.js,css,png,gif', {
3.      useHash: true // 开启 md5 戳
4.  });
5.
6.  //所有的css文件
7.  fis.match('**.css', {
8.      //发布到/static/css/xxx目录下
9.      release : '/static/css$0'
10.  });
11.
12.  //所有image目录下的.png, .gif文件

```

```

13.  fis.match('/images/(.*.{png,gif})', {
14.      //发布到/static/pic/xxx目录下
15.      release : '/static/pic/$1$2'
16.  });

```

再次编译得到：

- 源码：

```

1.  @import url('demo.css');

```

- 编译后

```

1.  @import url('/static/css/demo_7defa41.css');

```

- 源码：

```

1.  .style {
2.      background: url('images/body-bg.png');
3.  }

```

- 编译后


```

1.  .style {
2.      background: url('/static/pic/body-bg_1b8c3e0.png');
3.  }

```

- 源码：

```

1.  .style {
2.      _filter 
        DXImageTransform.Microsoft.AlphaImageLoader(src='images/body-bg.png');
3.  }

```

- 编译后

```

1.  .style {
2.      _filter 
        DXImageTransform.Microsoft.AlphaImageLoader(src='/static/pic/body-

```

```
        bg_1b8c3e0.png' );  
3.    }
```

# 声明依赖

## 声明依赖

声明依赖能力为工程师提供了声明依赖关系的编译接口。

FIS3 在执行编译的过程中，会扫描这些编译标记，从而建立一张 静态资源关系表，资源关系表详细记录了项目内的静态资源id、发布后的线上路径、资源类型以及 依赖关系 和 资源打包 等信息。使用 FIS3 作为编译工具的项目，可以将这张表提交给后端或者前端框架去运行时，根据组件使用情况来 按需加载资源或者资源所在的包，从而提升前端页面运行性能。

## 在html中声明依赖

用户可以在html的注释中声明依赖关系，这些依赖关系最终会被记录下来，当某个文件中包含字符 `__RESOURCE_MAP__` 那么这个记录会被字符串化后替换 `__RESOURCE_MAP__`。为了方便描述呈现，我们假定项目根目录下有个文件 `manifest.json`包含此字符，编译后会表结构替换到这个文件中。

在项目的index.html里使用注释声明依赖关系：

```
1. <!--
2.     @require demo.js
3.     @require "demo.css"
4. -->
```

默认情况下，只有js和css文件会输出到manifest.json表中，如果想将html文件加入表中，需要通过配置 `useMap` 让HTML文件加入 `manifest.json`，例如：

```
1. //fis-conf.js
2. fis.match('*.html', {
3.     useMap: true
4. })
```

配置以下内容到配置文件进行编译

```
1. // fis-conf.js
2. fis.match('*.html', {
3.     useMap: true
4. });
5.
6. fis.match('*.js,css', {
```

```

7.      // 开启 hash
8.      useHash: true
9.    });

```

查看 output 目录下的 manifest.json 文件，则可看到：

```

1.  {
2.    "res" : {
3.      "demo.css" : {
4.        "uri" : "/static/css/demo_7defa41.css",
5.        "type" : "css"
6.      },
7.      "demo.js" : {
8.        "uri" : "/static/js/demo_33c5143.js",
9.        "type" : "js",
10.       "deps" : [ "demo.css" ]
11.     },
12.     "index.html" : {
13.       "uri" : "/index.html",
14.       "type" : "html",
15.       "deps" : [ "demo.js", "demo.css" ]
16.     }
17.   },
18.   "pkg" : {}
19. }

```

## 在js中声明依赖

*fis*支持识别js文件中的 注释中的@require字段 标记的依赖关系，这些分析处理对 *html*的*script*标签内容 同样有效。

```

1.  //demo.js
2.  /**
3.   * @require demo.css
4.   * @require list.js
5.   */

```

经过编译之后，查看产出的 manifest.json 文件，可以看到：

```

1.  {
2.    "res" : {
3.      ...

```

```

4.         "demo.js" : {
5.             "uri" : "/static/js/demo_33c5143.js",
6.             "type" : "js",
7.             "deps" : [ "demo.css", "list.js", "jquery" ]
8.         },
9.         ...
10.     },
11.     "pkg" : {}
12. }

```

注意，`require()` 不再处理，js 中 `require()` 留给各种前端模块化方案，假设你选择的是 `AMD` 那么就得解析，`require([])` 和 `require()`；如果选用的是 `mod.js` 那么就得解析 `require.async()` 和 `require()`，其他亦然。

## 在css中声明依赖

`fis`支持识别css文件 注释中的`@require`字段 标记的依赖关系，这些分析处理对 `html`的`style`标签内容同样有效。

```

1. /**
2.  * demo.css
3.  * @require reset.css
4.  */

```

经过编译之后，查看产出的 `manifest.json` 文件，可以看到：

```

1. {
2.     "res" : {
3.         ...
4.         "demo.css" : {
5.             "uri" : "/static/css/demo_7defa41.css",
6.             "type" : "css",
7.             "deps" : [ "reset.css" ]
8.         },
9.         ...
10.     },
11.     "pkg" : {}
12. }

```



## 请转移到 Release 页面

## 请转移到 Release 页面

---

### 3.2.0 / Fri Aug 28 2015

---

- 升级 fis3-packager-map 支持从配置项中配置复杂的打包规则。
- domain 文件属性支持数组
- media 上下继承逻辑优化
- 解决 lint 重复报错的 bug
- 更新 fis3-command-release 修改 deploy 配置行为，让 deploy 遵循后面的覆盖前面的原则。关联 issues: #186
- 添加 hash 中间码，便于插件单纯获取 hash 值。
- 修复从内存中缓存的文件文件缓存数据丢失的 bug。
- 修复 sass 插件启用 sourcemap，map.json 包含无用字段的 bug。
- 修复match rules 里面有 function 类型的导致程序出错的 bug。
- 修复原始 match rules 被修改的 bug。
- 升级 fis3，去掉 useDomain 属性，给设置 domain 的资源自动启用domain

### 3.1.1 / Wed Aug 12 2015

---

- 升级 server 插件
  - 解决 root 不能修改的问题。
- 优化插件配置，去掉内部属性。详情: <https://github.com/fex-team/fis3/issues/142>

### 3.1 / Fri Aug 07 2015

---

- 升级 server 插件
  - 默认改成开启 node 插件
  - 输出远程访问 ip 地址
  - 内置 node server 而不是通过 npm install 获取，因为经常用人安装不下来。
- 解决内嵌导致的异步丢失 bug

### 3.0.20 / Thu Aug 06 2015

---

- 解决内嵌导致异步依赖丢失的 bug

- 升级 release 插件，解决 livereload 多个项目同时开启的问题。

## 3.0.18 / Tue Aug 04 2015

---

- 更新 map 插件，支持 packOrder
- 去掉诡异的打包排序功能，换成插件配置项中处理。
- 优化 `fis.match('!xxx')` 取反用法，去掉命中特殊选择器的功能。
- 保存 map.json 信息到对应的文件属性上。

## 3.0.16 / Thu Jul 30 2015

---

- deploy 阶段，默认加上编码转换
- 添加 moduleId 中间码，表示获取目标文件的 moduleId
- bugfix [#88](#)

## 3.0.13 / Wed Jul 22 2015

---

- 通过镜像下载 fis-components 和 脚手架。

## 3.0.10 / Wed Jul 15 2015

---

- 解决 watch 时，同时多个文件修改导致多次 release 的问题。
- 解决 standard 流程不能关闭的 bug
- 升级 fis3-command-release

## 3.0.8 / Mon Jul 13 2015

---

- 更新 fis3-command-release 参数验证漏掉 f, file, r, root.
- 默认去掉 useHash: true

## 3.0.7 / Mon Jul 08 2015

---

- 修复同名依赖时，可能自己依赖自己的问题。

## 3.0.5 / Mon Jul 08 2015

---

- 更新 fis3-command-inspect
- 去掉修改 node env 代码

## 3.0.4 / Mon Jul 07 2015

---

- 更新 `fis.uri`, 让其支持 `fis id` 查找。

## 3.0.2 / Mon Jul 06 2015

---

- 更新 `fis3-command-release` 到 1.2.0
  - 解决 `watch` 时, 在缓存依赖中的文件, 没有响应的问题。

## 3.0.1 / Mon Jul 06 2015

---

- 修改 `::pacakger => ::package`

## 3.0.0 / Mon Jul 03 2015

---

# 工作原理

## 工作原理

FIS3 是基于文件对象进行构建的，每个进入 FIS3 的文件都会实例化成一个 File 对象，整个构建过程都对这个对象进行操作完成构建任务。以下通过伪码来阐述 FIS3 的构建流程。

## 构建流程

```
1.  fis.release = function (opt) {
2.    var src = fis.util.find(fis.project.root);
3.    var files = {};
4.    src.forEach(function (f) {
5.      var file = new File(f);
6.      files[file.subpath] = fis.compile(file);
7.    });
8.    var packager = fis.matchRules('::package');
9.    var keys = Object.keys(packager);
10.   var ret = {
11.     files: files,
12.     map: {}
13.   };
14.   if (packager.indexOf('prepackager') > -1) {
15.     pipe('prepackager', ret);
16.   }
17.
18.   if (packager.indexOf('packager') > -1) {
19.     pipe('packager', ret);
20.   }
21.
22.   files.forEach(function (f) {
23.     // 打包阶段产出的 map 表替换到文件
24.     if (f._isResourceMap) {
25.       f._content = f._content.replace(/\b__RESOURCE_MAP__\b/g,
JSON.stringify(ret.map));
26.     }
27.   });
28.
29.   if (packager.indexOf('spriter') > -1) {
30.     pipe('spriter', ret);
```

```

31.     }
32.     if (packager.indexOf('postpackager') > -1) {
33.         pipe('postpackager', ret);
34.     }
35. }

```

如上述代码，整个 FIS3 的构建流程大体概括分为三个阶段。

1. 扫描项目目录拿到文件并初始化出一个文件对象列表
2. 对文件对象中每一个文件进行[单文件编译](#)
3. 获取用户设置的 `package` 插件，进行打包处理（包括合并图片）

其中打包处理开了四个扩展点，通过用户配置启用某些插件。

- **prepackager** 打包前处理插件扩展点
- **packager** 打包插件扩展点，通过此插件收集文件依赖信息、合并信息产出静态资源映射表
- **spriter** 图片合并扩展点，如 `csssprites`
- **postpackager** 打包后处理插件扩展点

## 单文件编译流程

```

1.  fis.compile = function (file) {
2.     if (file.isFile()) {
3.         if (exports.useLint && file.lint) {
4.             pipe('lint', file);
5.         }
6.         if (!file.hasCache) {
7.             process(file);
8.         } else {
9.             file.revertCache();
10.        }
11.    } else {
12.        process(file);
13.    }
14. };
15.
16. function process(file) {
17.     if (file.parser) {
18.         pipe('parser', file);
19.     }
20.     if (file.preprocessor) {
21.         pipe('preprocessor', file);

```

```

22.   }
23.   if (file.standard) {
24.       standard(file); // 标准化处理
25.   }
26.   if (file.postprocessor) {
27.       pipe('postprocessor', file);
28.   }
29.   if (file.optimizer) {
30.       pipe('optimizer', file);
31.   }
32. }

```

其中插件扩展点包括：

- lint：代码校验检查，比较特殊，所以需要 `release` 命令命令行添加 `-l` 参数
- parser：预处理阶段，比如 less、sass、es6、react 前端模板等都在此处预编译处理
- preprocessor：标准化前处理插件
- standard：标准化插件，处理[内置语法](#)
- postprocessor：标准化后处理插件

预处理阶段一般是对异构语言等进行预编译，如 less、sass 编译为标准的 css；前端模板被编译为 js 等等

单文件阶段通过读取文件属性，来执行对应扩展点插件。

举个例子：

```

1.  fis.match('*.es6', {
2.      parser: fis.plugin('babel'),
3.      rExt: '.js' // 代码编译产出时，后缀改成 .js
4.  });

```

给后缀是 `.es6` 的文件配置了一个 parser 属性，属性值是启用了个叫 `babel` 的插件，当执行到预处理阶段时，将 es6 编译为 es5，供浏览器执行。

其他插件扩展点亦然。

## File对象

```

1.  function File(filepath) {
2.      var props = path.info(filepath);
3.      merge(props, fis.matchRules(filepath)); // merge 分配到的属性
4.      assign(this, props); // merge 属性到对象

```

```
5. }
```

当一个文件被实例化为一个 File 对象后，包括一些文件基本属性，如 filename、realpath 等等，当这个文件被处理时，FIS3 还会把用户自定义的属性 merge 到文件对象上。

比如

```
1. fis.match('a.js', {  
2.   myProp: true  
3. });
```

这样 `a.js` 处理的时候就会携带这个属性 `myProp`。`myProp` 是一个自定义属性，FIS3 默认内置了一些属性配置，来方便控制一个文件的编译流程，可参考[配置属性](#)

可能你会问，自定义属性到底有什么用，其实自定义属性可以标注一些文件，提供插件来做一些特定的需求。

## 初级使用

## 初级使用

### 一个复杂一点的例子

为了尝试更多 FIS3 提供的特性，我们设计一个比较复杂的例子。这个例子包含

- 两个页面
- 三个 css 文件，其中俩页面各一个 css 文件，剩下一个 css 文件共用
- 包含一个 less 文件，并被俩页面同时使用
- 两个 png 图片
- 两个 js 文件

例子下载地址 [demo-lv1](#)

### 安装一些插件

FIS3 是一个扩展性很强的构建工具，社区也包含很多 FIS3 的插件。为了展示 FIS3 的预处理、静态合并 js、css 能力，需要安装两个插件。

- `fis-parser-less`：例子引入一个 less 文件，需要 less 预处理插件
- `fis3-postpackager-loader`：可对页面散列文件进行合并

FIS3 的插件都是以 NPM 包形式存在的，所以安装 FIS3 的插件需要使用 `npm` 来安装。

```
1. npm install -g 插件名
```

譬如：

```
1. npm install -g fis-parser-less
2. npm install -g fis3-postpackager-loader
```

### 预处理

FIS3 提供强大的预处理能力，可以对 less、sass 等异构语言进行预处理，还可以对模板语言进行预编译。FIS3 社区已经提供了绝大多数需要预处理的异构语言。

我们给定的例子中有个 less 文件，那么需要对 less 进行预处理，我们已经安装了对应的预处理插件。现在只需要配置启用这个插件就能搞定这个事情。



```

1.  fis.match('*.less', {
2.    // fis-parser-less 插件进行解析
3.    parser: fis.plugin('less'),
4.    // .less 文件后缀构建后被改成 .css 文件
5.    rExt: '.css'
6.  })

```

如同之前强调的，虽然构建后后缀为 `.css`。但在使用 FIS3 时，开发者只需要关心源码路径。所以引入一个 less 文件时，依然是 `.less` 后缀。

```

1. <link rel="stylesheet" type="text/css" href="./test.less">

```

## 简单合并

在起步中我们阐述了[图片合并 CssSprite](#)，为了减少请求。现在介绍一种比较简单的打包 js、css 的方式。

启用打包后处理插件进行合并：

- 基于整个项目打包

```

1.  fis.match(':::package', {
2.    postpackager: fis.plugin('loader')
3.  });
4.
5.  fis.match('*.less', {
6.    parser: fis.plugin('less'),
7.    rExt: '.css'
8.  });
9.
10. fis.match('*.{less,css}', {
11.   packTo: '/static/aio.css'
12. });
13.
14. fis.match('*.js', {
15.   packTo: '/static/aio.js'
16. });

```

这样配置打包的结果是：一个页面最终只会引入一个 css、js：aio.js 和 aio.css。

但两个页面的资源都被打包到同一个包里面了。这个可能不是我们想要的结果，我们想一个页面只包含这个页面用过的资源。

- 基于页面的打包方式

```
1. fis.match(':::package', {
2.   postpackager: fis.plugin('loader', {
3.     allInOne: true
4.   })
5. });
6.
7. fis.match('*.less', {
8.   parser: fis.plugin('less'),
9.   rExt: '.css'
10. });
```

给 `loader` 插件配置 `allInOne` 属性，即可对散列的引用链接进行合并，而不需要进行配置 `packTo` 指定合并包名。

注意，这个插件只针对纯前端的页面进行比较粗暴的合并，如果使用了后端模板，一般都需要从整站出发配置合并。

## 构建调试预览

进入 demo 目录，执行命令，构建即发布到本地测试服务根目录下：

```
1. fis3 release
```

启动内置服务器进行预览；

```
1. fis3 server start --type node
```

## 中级使用

## 中级使用

在初级使用中，为了解析 `less` 和 进行简单的资源合并，我们安装了两个已经提供好的插件，使用插件完成了我们的工作。假设某些情况下，还没有相关插件，该怎么办？

那么这节讨论一下 FIS 中插件如何编写。在[工作原理](#)中，已经介绍了整个构建的过程，以及说明了 FIS 与其他构建工具的不同点。

### 预处理插件编写

假设给定项目中要是用 `es6`，线上运行时解析成标准 `js` 性能堪忧，想用自动化工具进行预处理转换。如原理介绍 `parser` 阶段就是进行归一化的过程，通过预处理阶段，整个文件都会翻译为标准的文件，即浏览器可解析的文件。

这时候我们搜罗开源社区，看转换 `es6` 到 `es5` 哪个转换工具更好一些，发现 `babel` 具有无限的潜能。

### 任务

- 预处理 `es6` 为 `es5`

### 前期准备

- `.es6` 后缀最终变为 `.js`
- 使用 `babel` 进行 `es6` 的转换
- FIS3 实现一个 `parser` 类型的插件，取名叫 `translate-es6`，插件全名 `fis3-parser-translate-es6`

### 开发插件

[开发插件](#)文档详细介绍了开发插件的步骤，但为了更友好的进行接下来的工作，我们在这里简述一下整个过程。

FIS3 支持 `local mode` 加载一个插件。当你调用一个插件的时候，配置如下：

```
1. {
2.   parser: fis.plugin('translate-es6')
3. }
```

如果项目的根目录 `node_modules` 下有这个插件，就能挂载起来。

```
1. my-proj/node_modules/fis3-parser-translate-es6
```

这样我们就知道，插件逻辑放到什么地方能用 `fis.plugin` 接口挂载。

```
1. my-proj/node_modules/fis3-parser-translate-es6/index.js
```

```
1. // vi index.js
2. // babel node.js api 只有 babel-core 不能完成翻译
3. // babel-core 需要安装依赖 babel-preset-es2015
4. // 参考：阮一峰的es6入门 http://es6.ruanyifeng.com/#docs/intro
5. var babel = require('babel-core');
6. module.exports = function (content, file, options) {
7.   var result = babel.transform(content, opts);
8.   return result.code; // 处理后的文件内容
9. }
```

如上我们调用 `babel-core` 封装了一个 `fis3-parser` 插件。

现在我们要在项目使用它

```
1. my-proj/fis-conf.js # 项目 fis3 配置文件
2. my-proj/node_modules/fis3-parser-translate-es6/index.js # 插件逻辑
3. my-proj/style.es6
4. my-proj/index.html
```

## 配置使用

```
1. fis.match('*.es6', {
2.   parser: fis.plugin('translate-es6', {
3.     presets: ['es2015']
4.   }),
5.   rExt: '.js' // .es6 最终修改其后缀为 .js
6. })
```

## 构建

```
1. fis3 release -d ./output
```

## 打包插件编写

在开始之前，我们需要阐述下打包这个名词，打包在前端工程中有两个方面。

- 收集页面用到的 js、css，分别合并这些引用，将资源合并成一个。
- 打包，对某些资源进行打包，而记录它们打包的信息，譬如某个文件打包到了哪个包文件。

其实一般意义上来说，对于第一种情况收集打包只适合于纯前端页面，并且要求资源都是静态引入的。假设出现这种情况：

```
1. <script type="text/javascript">
2. // load common.js and index.js
3. F.load([
4.   'common',
5.   'index'
6. ]);
7. </script>
```

需要通过动态脚本去加载的资源，就无法通过工具静态分析来去做合并了。

还有一种情况，如果模板是后端模板，也依然无法做到这一点，因为加载资源只有在运行时、解析时才能确定。

那么对于这类打包合并资源，需要特殊的处理思路。

1. 直接将所有资源合并成一个文件，进行整站（整个项目）合并；
2. 通过配置文件配置打包，并且合并时记录合并信息，在运行时根据这些打包信息吐给浏览器合适的资源。

第一种，粗暴问题多，并且项目足够大时效率明显不合适。我们主要探讨第二种。

FIS3 默认内置了一个打包插件 `fis3-packager-map`，它根据用户的配置信息对资源进行打包。

```
1. //fis-conf.js
2. fis.match('/widget/*.js', {
3.   packTo: '/static/widget_pkg.js'
4. })
```

标明 `/widget` 目录下的 `js` 被合并成一个文件 `widget_pkg.js`

假设

```
1. /widget/a.js
2. /widget/b.js
3. /widget/c.js
4. /map.json
```

编译发布后：

```
1. /widget/a.js
2. /widget/b.js
3. /widget/c.js
4. /static/widget_pkg.js
5. /map.json
```

我们前面说过

1. 当某文件包含字符 `__RESOURCE_MAP__` 时，最终静态资源表（资源之间的依赖、合并信息）会替换这个字符。

构建后，出现一个合并资源以外，还会产出一张某资源合并到什么文件中的关系信息。

```
1. {
2.   "res": {
3.     "widget/a.js": {
4.       "uri": "/widget/a.js",
5.       "type": "js",
6.       "pkg": "p0"
7.     },
8.     "widget/b.js": {
9.       "uri": "/widget/b.js",
10.      "type": "js",
11.      "pkg": "p0"
12.    },
13.    "widget/c.js": {
14.      "uri": "/widget/c.js",
15.      "type": "js",
16.      "pkg": "p0"
17.    }
18.  },
19.  "pkg": {
20.    "p0": {
21.      "uri": "/static/widget_pkg.js",
22.      "has": [
23.        "widget/a.js",
24.        "widget/b.js",
25.        "widget/c.js"
26.      ],
27.      "type": "js"
```

```
28.     }  
29.     }  
30. }
```

## 发布插件

FIS3 的插件都放在 NPM 平台上，把插件发布到其上即可。

参考链接 [npm publish](#)

发布的插件如何使用

- `npm install -g <plugin>` 安装插件
- FIS3 配置文件中按照配置规则进行配置，`fis.plugin(<plugin-name>)`

## 高级使用

## 高级使用

### 静态资源映射表

记录文件依赖、打包、URL等信息的表结构，在 FIS2 中统称 `map.json`。在 FIS3 中默认不产出 `map.json`，FIS3 中为了方便各种语言下读取 `map.json`，对产出 `map.json` 做了优化。

当某个文件包含字符 `__RESOURCE_MAP__`，就会用表结构数据替换此字符。这样的好处是不再固定把表结构写入某一个特定文件，方便定制。

比如在

*php*

```
1. <?php
2. $map = json_decode('__RESOURCE_MAP__', true);
3. ?>
```

*js*

```
1. var _map = __RESOURCE_MAP__;
```

假设上面的 php 和 js 为分析静态资源映射表的程序，那么就省去了读 `map.json` 的过程。

当然，如果你想继续像 FIS2 一样的产出 `map.json`，只需要在模块下新建文件 `map.json`，内容设置为 `__RESOURCE_MAP__` 即可。

### 模块化开发

模块化开发是工程实践的最佳手段，分而治之维护上带来了很大的益处。

说到模块化开发，首先很多人都会想到 AMD、CMD，同时会想到 `require.js`、`sea.js` 这样的前端模块化框架。主要给 js 提供模块化开发的支持，之后也增加了对 css、前端模板的支持。这些框架就包含了组件依赖分析、保持加载并保持依赖顺序等功能。

但在 FIS 中，依赖本身在构建过程中就已经分析完成，并记录在静态资源映射表中，那么对于线上运行时，模块化框架就可以省掉依赖分析这个步骤了。



在声明依赖内置语法中提到了几种资源之间标记依赖的语法，这样模板可以依赖 js、css，js 可以依赖某些 css，或者某个类型的组件可以互相依赖。

另外，考虑到 js 还需要有运行时支持，所以对于不同前端模板化框架，在 js 代码中 FIS 编译分析依赖增加了几种依赖函数的解析。这些包括

#### AMD

```
1. define()
2. require([]);
3. require('');
```

#### seajs

```
1. define()
2. require('')
3. sea.use([])
```

#### mod.js (extends commonjs)

```
1. define()
2. require('')
3. require.async('')
4. require.async([])
```

考虑到不可能一个框架运用多个模块化框架（因为全都占用同样的全局函数，互斥），所以编译支持这块分成三个插件进行支持。

- [fis3-hook-commonjs](#)
- [fis3-hook-amd](#)
- [fis3-hook-cmd](#)

```
1. // vi fis-conf.js
2. fis.hook('commonjs');
```

插件 [README](#) 有详细的使用文档。

如上面说到的，这个编译插件只是对编译工具做一下扩展，支持前端模块化框架中的组件与组件之间依赖的函数，以及入口函数来标记生成到静态资源映射表中；另外一个功能是针对某些前端模块化框架的特性自动添加 `define`。

有了依赖表，但如何把资源加载到页面上，需要额外的 **FIS** 构建插件或者方案支持。

假设以纯前端（没有后端模板）的项目为例，对于依赖组件的加载就靠插件 `fis3-postpackager-loader`。其是一种基于构建工具的加载组件的方法，构建出的 `html` 已经包含了其使用到的组件以及依赖资源的引用。

```
1. // npm install -g fis3-postpackager-loader
2. fis.match('::package', {
3.   postpackager: fis.plugin('loader', {})
4. });
```

为了方便、统一管理组件以及合并时便利，需要把组件统一放到某些文件夹下，并设置此目录下的资源都是组件资源。

```
1. // widget 目录下为组件
2. fis.match('/widget/**/*.js', {
3.   isMod: true
4. });
```

通过以上三步，纯前端的模块化开发就可实现。

总结一下；

- 编译工具扩展：根据不同前端模块化框架，扩展声明依赖能力
- 静态资源管理：解析静态资源映射表加载页面用到的组件及其组件的依赖
- 目录规范：设置某个文件夹下资源标记为依赖

工具扩展、目录规范，前后端的前端工程项目都需要，其不同之处就在于静态资源管理这部分。

## 资源映射表的模块化方案设计

### 解决方案封装

解决方案：解决一系列特定问题的工具、规范、开发、上线支持的方案，被称为 解决方案。前端工程的解决方案一般包括：

```
1. 研发规范 + 模块化框架 + 测试套件 + 辅助开发工具
```

FIS3 中的包装解决方案，就是把这些集成到一个工具中。

一个解决方案就是继承自 FIS3 并且支持特定模块化开发、特定模板语言、特定处理流程、研发规范的构建工具。

### 封装解决方案的必要性

- 规范开发，对于特定团队业务，应该有特定的目录规范、模块化框架等
- FIS3 只提供一个方便定制前端工程的构建系统，每个团队需要怎么样去处理工程需要自己定制，定制会引入大量的 FIS3 插件，解决方案可统一规定引入哪些插件
- 树立独立技术品牌

## 解决方案封装

### 准备

- 方案名 `foo`
- 构建工具名字 `foo`
- 模板语言 PHP
- 模块化框架选择 `require.js`
- 特定目录规范

### 目录规范

1. `/static` # 静态资源
2. `/page` # 页面
3. `/widget` # 组件
4. `/fis-conf.js` # 配置文件

### 部署规范

1. `/template` # 所有的 PHP 模板
2. `/static` # 所有的静态资源

### 构建工具

1. `foo`
2. `foo/bin/foo.js`
3. `foo/index.js`
4. `package.json`

- 基于 FIS3 配置目录规范和部署规范

```
1. //vi foo/index.js
2. var fis = module.exports = require('fis3');
3. fis.require.prefixes.unshift('foo');
4. fis.cli.name = 'foo';
5. fis.cli.info = require('./package.json');
6.
```

```
7.  fis.match('*', {
8.    release: '/static/$0' // 所有资源发布时产出到 /static 目录下
9.  });
10.
11. fis.match('*.php', {
12.   release: '/template/$0' // 所有 PHP 模板产出后放到 /template 目录下
13. });
14.
15. // 所有js, css 加 hash
16. fis.match('*.{js,css,less}', {
17.   useHash: true
18. });
19.
20. // 所有图片加 hash
21. fis.match('image', {
22.   useHash: true
23. });
24.
25. // fis-parser-less
26. fis.match('*.less', {
27.   parser: fis.plugin('less'),
28.   rExt: '.css'
29. });
30.
31. fis.match('*.js', {
32.   optimizer: fis.plugin('uglify-js')
33. });
34.
35. fis.match('*.{css,less}', {
36.   optimizer: fis.plugin('clean-css')
37. });
38.
39. fis.match('*.png', {
40.   optimizer: fis.plugin('png-compressor')
41. });
42.
43. fis.match('widget/*.{php,js,css}', {
44.   isMod: true
45. });
46.
47. fis.match(':::package', {
48.   spriter: fis.plugin('csssprites')
```

```

49. });
50.
51. //fis3-hook-module
52. fis.hook('module', {
53.   mode: 'amd' // 模块化支持 amd 规范, 适应 require.js
54. });

```

- 实现 `/bin/foo.js`

```

1. #!/usr/bin/env node
2.
3. // vi foo/bin/foo.js
4.
5. var Liftoff = require('liftoff');
6. var argv = require('minimist')(process.argv.slice(2));
7. var path = require('path');
8. var cli = new Liftoff({
9.   name: 'foo', // 命令名字
10.  processTitle: 'foo',
11.  moduleName: 'foo',
12.  configName: 'fis-conf',
13.
14.  // only js supported!
15.  extensions: {
16.    '.js': null
17.  }
18. });
19.
20. cli.launch({
21.  cwd: argv.r || argv.root,
22.  configPath: argv.f || argv.file
23. }, function(env) {
24.  var fis;
25.  if (!env.modulePath) {
26.    fis = require('..');
27.  } else {
28.    fis = require(env.modulePath);
29.  }
30.  // 配置插件查找路径, 优先查找本地项目里面的 node_modules
31.  // 然后才是全局环境下面安装的 fis3 目录里面的 node_modules
32.  fis.require.paths.unshift(path.join(env.cwd, 'node_modules'));
33.  fis.require.paths.push(path.join(path.dirname(__dirname),

```

```

    'node_modules')));
34.   fis.cli.run(argv, env);
35. });

```

以上代码 copy 过来即可，不需要做大的改动，感兴趣可研究其原理。

- 依赖的 NPM 包，需要在 package.json 中加上依赖：
  - **fis-parser-less** 解析 less
  - **fis-optimizer-uglify-js** 压缩 js, fis3 已内置
  - **fis-optimizer-clean-css** 压缩 css, fis3 已内置
  - **fis-optimizer-png-compressor** 压缩 png 图片, fis3 已内置
  - **fis3-hook-module** 模块化支持插件
  - **fis3** fis3 核心
  - **minimist**
  - **liftoff**
- package.json 需要添加

```

1.  "bin": {
2.    "foo": "bin/foo.js"
3.  }

```

- 发布 foo 到 NPM

通过以上步骤可以简单封装一个解决方案，FIS3 提供了大量的插件，已经几乎极其简单的配置方式来搞定研发规范的设置，很轻松即可打造完整的前端集成解决方案。

**foo** [源码下载地址](#)

## 基于Smarty的解决方案

**fis3-smarty** 集成了 **fis-plus** 的目录规范以及处理插件。实现对 Smarty 模板解决方案的工程构建工具支持。

此方案在 FIS3 替代 **fis-plus** 解决方案。

[Smarty 解决方案原理](#)

## 基于纯PHP的解决方案

详细见 [纯php静态资源管理方案](#)

解决问题

- 支持模块化的开发，使用commonJS或者AMD方案来控制前端JS资源的加载
- 支持组件化开发，使用组件时能自动加载对应依赖的静态资源
- 自动分析资源依赖关系，确保依赖资源正常下载
- 自动把css放顶部、JS放底部输出，提升页面渲染性能
- 支持收集组件中的内嵌样式或脚本，合并输出

## 基于Laravel的解决方案

详细请参见 [fis-laravel](#)

# 接口文档

- [命令行](#)
- [配置](#)
- [内置插件及配置](#)
- [自定义插件](#)



# 命令行

## 命令

通过以下命令查看 FIS3 提供了哪些命令。

```

1. ~ fis3 -h
2.
3. [INFO] Currently running fis3 (/usr/local/lib/node_modules/fis3/)
4.
5. Usage: fis3 <command>
6.
7. Commands:
8.
9.   init                scaffold with specied template.
10.  install              install components
11.  release [media name] build and deploy your project
12.  server               launch a php-cgi server
13.  inspect [media name] inspect the result of fis.match
14.
15. Options:
16.
17.   -h, --help          print this help message
18.   -v, --version       print product version and exit
19.   -r, --root <path>   specify project root
20.   -f, --file <filename> specify the file path of `fis-conf.js`
21.   --no-color          disable colored output
22.   --verbose           enable verbose mode

```

通过帮助信息，不难发现 FIS3 默认内置了命令

`release`、`install`、`init`、`server`、`inspect` 等命令，这些命令都是 FIS `release` 插件提供，通过

```
1. fis3 <command>
```

来调用，详见以下文档介绍内置的命令。

## release

`release` 插件提供，默认内置

## 编译发布一个 FIS3 项目

```

1. $ fis3 release -h
2.
3. [INFO] Currently running fis3 (/usr/local/lib/node_modules/fis3/)
4.
5. Usage: fis3 release [media name]
6.
7. Options:
8.
9.   -h, --help          print this help message
10.  -d, --dest <path>   release output destination
11.  -l, --lint           with lint
12.  -w, --watch          monitor the changes of project
13.  -L, --live           automatically reload your browser
14.  -c, --clean          clean compile cache
15.  -u, --unique         use unique compile caching

```

添加 `-h` 或者 `--help` 参数可以看到如上帮助信息，其中标明此命令有哪些参数并且起到什么作用。

- `-h`、`--help` 打印帮助信息
- `-d`、`--dest` 编译产出到一个特定的目录

```
1. fis3 release -d ./output
```

发布到当前命令执行目录下的 `./output` 目录下。

```
1. fis3 release -d ../output
```

发布到当前命令执行目录父目录的 `../output` 目录下，即上一级的 `output` 目录。

- `-l`，`--lint` 启用文件格式检测

```
1. fis3 release -l
```

默认 `fis3 release` 不会启用 `lint` 过程，只有通过命令行参数指定了才会开启。

- `-w`、`--watch` 启动文件监听

```
1. fis3 release -w
```

会启动文件监听功能，当文件变化时会编译发布变化了的文件以及依赖它的文件。加了此参数，命令不会马上退出，而是常驻且监听文件变化，并按需再次执行。想停止命令需要使用快捷键 `CTRL + C` 来强制停止。

- `-L`、`--live` 启动 `livereload` 功能

```
1. fis3 release -L
```

`livereload` 功能应该跟 `watch` 功能一起使用（`-w` 在开启 `livereload` 的前提下，自动开启），当某文档做了修改时，会自动刷新页面。

- `-c`，`--clean` 清除编译缓存

```
1. fis3 release -c
```

默认 `fis` 的每次编译都会检测编译缓存是否有效，如果有效 `fis` 是不会重复编译的。开启此选项后，`fis` 编译前会做一次缓存清理。

- `-u`，`--unique` 启用独立缓存

为了防止多个项目同时编译时缓存文件混乱，启用此选项后，会使用独立的缓存文件夹。一般用于编译机。

## install

`fis-command-install` 插件提供，默认内置

用来从组件平台中下载组件到当前项目中，并自动下载其依赖。默认组件下载来源于 `fis-components` 机构。

更多内容请查看 [components 文档](#)。

```
1. ~/sandbox/test fis3 install bootstrap-datepicker
2.
3. [INFO] Currently running fis3 (/usr/local/lib/node_modules/fis3/)
4.
5. Installed
6. └─ github:fis-components/bootstrap-datepicker@v1.4.0
7. └─ github:fis-components/bootstrap@v3.3.4
8. └─ github:fis-components/jquery@2.1.0
9.
10. ~/sandbox/test tree . -L 3
11. .
12. └─ components
```

```
13.      └─ bootstrap
14.      │   └─ README.md
15.      │   └─ affix.js
16.      │   └─ alert.js
17.      │   └─ bootstrap.js
18.      │   └─ button.js
19.      │   └─ carousel.js
20.      │   └─ collapse.js
21.      │   └─ component.json
22.      │   └─ css
23.      │   └─ dropdown.js
24.      │   └─ fonts
25.      │   └─ modal.js
26.      │   └─ popover.js
27.      │   └─ scrollspy.js
28.      │   └─ tab.js
29.      │   └─ tooltip.js
30.      │   └─ transition.js
31.      └─ bootstrap-datepicker
32.      │   └─ README.md
33.      │   └─ bootstrap-datepicker.css
34.      │   └─ bootstrap-datepicker.js
35.      │   └─ bootstrap-datepicker.standalone.css
36.      │   └─ bootstrap-datepicker3.css
37.      │   └─ bootstrap-datepicker3.standalone.css
38.      │   └─ component.json
39.      └─ jquery
40.          └─ README.md
41.          └─ component.json
42.          └─ jquery.js
43.
44. 6 directories, 25 files
```

## 命令使用说明

```
1. fis3 install --help
2.
3. [INFO] Currently running fis3 (/usr/local/lib/node_modules/fis3/)
4.
5. Usage: install [options] <components...>
6.
7. Options:
```

```

8.
9.      -h, --help          output usage information
10.     --save              save component(s) dependencies into `components.json`
    file.
11.     -r, --root <path>  set project root

```

可以同时下载多个组件，多个组件之间使用空格隔开，如：

```
1. $ fis3 install jquery jquery-ui
```

当设置 `--save` 参数时，除了安装组件外，还会将依赖信息保存在当前项目根目录下面的 `component.json` 文件中。

## init

`fis3-command-init` 插件提供，默认内置

`fis3` 脚手架工具，用来快速初始化项目。在 `fis-scaffold` 机构中的仓库都可以通过 `fis3 init ${模板名称}` 来初始化到当前目录。当不指定模板名称时，`fis3` 会使用 `default` 作为模板用来初始化。

```

1. fis3 init --help
2.
3. [INFO] Currently running fis3 (/usr/local/lib/node_modules/fis3/)
4.
5. Usage: init <template>
6.
7. Options:
8.
9.      -h, --help          output usage information
10.     -r, --root <path>  set project root

```

## server

`fis-command-server` 插件提供，默认内置

`fis3` 内置了一个小型 web server，可以通过 `fis3 server start` 快速开启。如果一切正常，开启后它将自动弹出浏览器打开 `http://127.0.0.1:8080/`。

需要说明的是，`fis3` 自带的 server 默认是通过 java 内嵌 jetty 然后桥接 php-cgi 的方式运行的。所以，要求用户机器上必须安装有 jre 和 php-cgi 程序。

另外，`fis server` 是后台进行运行的，不会随着进程的结束而停止。如果想停止该服务器，请使用

`fis3 server stop` 进行关闭。

更多说明请参考命令行使用说明。

```

1. $ fis3 server --help
2.
3. [INFO] Currently running fis3 (/usr/local/lib/node_modules/fis3/)
4.
5. Usage: server <command> [options]
6.
7. Commands:
8.
9.     start                start server
10.    stop                 shutdown server
11.    restart              restart server
12.    info                 output server info
13.    open                 open document root directory
14.    clean                 clean files in document root
15.    install <name>       install server framework
16.
17. Options:
18.
19.     -h, --help           output usage information
20.     -p, --port <int>    server listen port
21.     --root <path>       document root
22.     --type <php|java|node> process language
23.     --rewrite [script]  enable rewrite mode
24.     --repos <url>       install repository
25.     --timeout <seconds> start timeout
26.     --php_exec <path>   path to php-cgi executable file
27.     --php_exec_args <args> php-cgi arguments
28.     --php_fcgi_children <int> the number of php-cgi processes
29.     --php_fcgi_max_requests <int> the max number of requests
30.     --registry <registry> set npm registry
31.     --include <glob>    clean include filter
32.     --exclude <glob>    clean exclude filter
33.     --https              start https server

```

## inspect

`fis3-command-inspect` 插件提供，默认内置

用来查看文件 `match` 结果。如下所示，将列出项目中所有文件，并显示该文件有哪些属性及属性值，以及该属性是源于哪个 `fis.match` 配置。

```
1.  fis3 inspect
2.
3.  [INFO] Currently running fis3 (/usr/local/lib/node_modules/fis3/)
4.
5.  ~ /README.md
6.  -- useHash false `*` (0th)
7.
8.
9.  ~ /comp/1-0/1-0.js
10. -- useHash false `*` (0th)
11. -- isMod true `/comp/**/*.js` (1th)
12. -- release /static/comp/1-0/1-0.js `/comp/**/*.js` (1th)
13.
14.
15. ~ /comp/2-0/2-0.js
16. -- useHash false `*` (0th)
17. -- isMod true `/comp/**/*.js` (1th)
18. -- release /static/comp/2-0/2-0.js `/comp/**/*.js` (1th)
19.
20.
21. ~ /comp/cal/cal.js
22. -- useHash false `*` (0th)
23. -- isMod true `/comp/**/*.js` (1th)
24. -- release /static/comp/cal/cal.js `/comp/**/*.js` (1th)
25.
26.
27. ~ /index.html
28. -- useHash false `*` (0th)
29.
30.
31. ~ /static/mod.js
32. -- useHash false `*` (0th)
33.
34.
35. ~ ::package
36. -- postpackager [plugin `loader`] `::package` (2th)
```

# 配置

本文提及配置 API & 属性都可以在 [配置 API](#) 和 [配置属性](#) 找到

FIS3 配置设计了一套类似 css 的规则，而就如同 css 一样，有 `!important` 也有 `@media`，那么就在这篇文档中揭露我们的类 css 配置；默认情况下，配置文件写到 `conf.js` 文件中，此文件放到项目的根目录下，或说有此文件的目录被看做是项目根目录

以下例子配置内容，都应该指的是 `fis-conf.js` 的内容，不再特别说明；

通过 API `fis.match()` FIS3 在处理的时候首先会加载项目内的所有文件，然后通过 `fis.match()` 来为某一个文件分配不同的属性，这些属性叫做文件属性。这些属性控制这个文件经过怎么样的操作；

先来一个例子

启用插件的例子

```
1. fis.match('a.js', {
2.     optimizer: fis.plugin('uglify-js')
3. });
4.
5. fis.match('b.min.js', {
6.     optimizer: null
7. })
```

如上面，a.js 文件分配了属性，其中处理过程中会调用 `fis-optimizer-uglify-js` 插件进行压缩；而已经压缩过的 b.min.js 就不需要进行压缩了，那么它的 optimizer 就设置为 `null`；

可以不设置这个属性为 `null` 因为默认为 `null`

规则覆盖的例子

假设 `fis.match()` 给若干文件分配了属性，当两个规则之间匹配的文件相同时，后设置的可以覆盖前面设置的属性，如果前面规则没有某属性则追加；

```
1. fis.match('{a,b,c}.js', {
2.     optimizer: fis.plugin('uglify-js')
3. });
4.
5. fis.match('{a,b}.js', {
6.     isMod: true,
7.     optimizer: null
8. });
```



- `c.js` 分配到的属性是 `{optimizer: fis.plugin('uglify-js')}`，意思是最终会被压缩
- `a.js` 和 `b.js` 分配到的属性是 `{isMod: true, optimizer: null}` 意思是最终会附带属性 `isMod` 并进行组件化处理、不做压缩

通过上面两个例子，大家不难看出；FIS3 设计的是一套类 `css` 的配置体系，那么其中

`fis.match()` 就是用来设置规则的；其中第一个参数可当成是 `selector` 其设置的类型是 `glob` 或者是 `[正则][]`；

## media

多状态，刚才说到过，FIS3 中都靠给文件分配不同属性来决定这个文件经过哪些处理的；那么 `media` 就能让我们在不同状态（情形）下给文件分配不同属性；这个任务就由 `fis.media()` 完成；

假设我们有如下需求，当在开发阶段资源都不压缩，但是在上线时做压缩，那么这个配置如何写呢；

```
1. //default `dev` mode
2. fis.match('**.*js', {
3.
4. });
5.
6. fis.media('prod')
7.   .match('**.*js', {
8.     optimizer: fis.plugin('uglify-js')
9.   })
10.  .match('**.*css', {
11.    optimizer: fis.plugin('clean-css')
12.  });
```

这样就写完了，那么怎么在编译发布的时候使用 `media` 呢，是这样的；

```
1. fis3 release <media>
```

那么对上面的配置

- `fis3 release` 默认开发状态不做压缩
- `fis3 release prod` 上线编译调用

## important

`fis.match()` 的第三个参数就是设置 `!important` 的，那么设置了这个属性后，后面的规则就无法覆盖了。

比如

```
1. fis.match('{a,b,c}.js', {
2.     optimizer: fis.plugin('uglify-js')
3. }, true);
4.
5. fis.match('a.js', {
6.     optimizer: null
7. })
```

这样的设置下，当 `a.js` 处理时还是会被调用压缩器进行压缩；

## ::package

```
1. fis.match('::package', {
2.     packager: fis.plugin('map')
3. });
```

表示当 `packager` 阶段所有的文件都分配某些属性

## ::image

```
1. // 所有被标注为图片的文件添加 hash
2. fis.match('::image', {
3.     useHash: true
4. });
```

- `project.fileType.image`

## ::text

```
1. // 所有被标注为文本的文件去除 hash
2. fis.match('::text', {
3.     useHash: false
4. });
```

- `project.fileType.text`

## :js

匹配模板中的内联 js，支持 `isHtmlLike` 的所有模板

```
1. // 压缩 index.html 内联的 js
2. fis.match('index.html:js', {
3.   optimizer: fis.plugin('uglify-js')
4. });
5.
6. // 压缩 index.tpl 内联的 js
7. fis.match('index.tpl:js', {
8.   optimizer: fis.plugin('uglify-js')
9. })
```

## :CSS

匹配模板中内联 css，支持 `isHtmlLike` 的所有模板

```
1. // 压缩 index.html 内联的 css
2. fis.match('index.html:css', {
3.   optimizer: fis.plugin('clean-css')
4. });
5.
6. // 压缩 index.tpl 内联的 css
7. fis.match('index.tpl:css', {
8.   optimizer: fis.plugin('clean-css')
9. })
```

## 配置接口

`fis3` 通过配置来决定代码、资源该如何处理，包括配置、压缩、CDN、合并等；

## 配置API

### `fis.set()`

设置一些配置，如系统内置属性 `project`、`namespace`、`modules`、`settings`。  
`fis.set` 设置的值通过`fis.get()`获取

语法

```
fis.set(key, value)
```

参数

- key

任意字符串，但系统占用了 `project`、`namespace`、`modules`、`settings` 它们在系统中有特殊含义，[详见](#)

当字符串以 `.` 分割的，`.` 字符后的字符将会是字符前字符同名对象的键

- value

任意变量

```
1. fis.set('namespace', 'home');
2. fis.set('my project namespace', 'common');
3. fis.set('a.b.c', 'some value'); // fis.get('a') => {b: {c: 'some value'}}
```

### `fis.get()`

获取已经配置的属性，和 `fis.set()` 成对使用

语法

```
fis.get(key)
```

参数

- key

任意字符串

```

1. // fis.set('namespace', 'common')
2. var ns = fis.get('namespace');
3.
4. // fis.set('a.b.c', 'd')
5. fis.get('a'); // => {b:{c: 'd'}}
6. fis.get('a.b'); // => {c:'d'}
7. fis.get('a.b.c'); // => 'd'

```

## fis.match()

给匹配到的文件分配属性，[文件属性](#)决定了这个文件进行怎么样的操作；

`fis.match` 模拟一个类似 css 的覆盖规则，负责给文件分配规则属性，这些规则属性决定了这个文件将会被如何处理；

就像 css 的规则一样，后面分配到的规则会覆盖前面的；如

```

1. fis.match('{a,b}.js', {
2.     release: '/static/$0'
3. });
4.
5. fis.match('b.js', {
6.     release: '/static/new/$0'
7. });

```

b.js 最终分配到的规则属性是

```

1. {
2.     release: '/static/new/$0'
3. }

```

那么 b.js 将会产出到 `/static/new` 目录下；

语法

```
fis.match(selector, props[, important])
```

参数

- selector

[glob](#) 或者是任意正则

- props

### 文件属性

- important

bool 设置了这个属性为 true，即表示设置的规则无法被覆盖；具体行为可参考 [css !important](#)

```
1. fis.match('*.js', {
2.   useHash: true,
3.   release: '/static/$0'
4. });
```

## fis.media()

`fis.media` 是模仿自 `css` 的 `@media`，表示不同的状态。这是 `fis3` 中的一个重要概念，其意味着有多份配置，每一份配置都可以让 `fis3` 进行不同的编译；

比如开发时和上线时的配置不同，比如部署测试机时测试机器目录不同，比如测试环境和线上机器的静态资源 domain 不同，一切这些不同都可以设定特定的 `fis.media` 来搞定；

### 语法

```
fis.media(mode)
```

### 参数

- mode

`string` mode，设定不同状态，比如 `rd`、`qa`、`dev`、`production`

### 返回值

```
1. `fis` 对象
```

```
1. fis.media('dev').match('*.js', {
2.   optimizer: null
3. });
4.
5. fis.media('rd').match('*.js', {
6.   domain: 'http://rd-host/static/cdn'
7. });
```

# fis.plugin()

插件调用接口

语法

```
fis.plugin(name [, props [, position]])
```

属性

- name

插件名，插件名需要特殊说明一下，fis3 固定了插件扩展点，每一个插件都有个类型，体现在插件发布的 npm 包名字上；比如

`fis-parser-less` 插件，`parser` 指的是在 `parser` 扩展点做了个解析 `.less` 的插件。

那么设置插件的时候，插件名 `less`，比如设置一个 `parser` 类型的插件是这么设置的；

```
1.  fis.match('*.*less', {  
2.      parser: fis.plugin('less', {}) //属性 parser 表示了插件的类型  
3.  })
```

- props

对象，给插件设置用户属性

```
1.  fis.match('*.*less', {  
2.      parser: fis.plugin('less', {});  
3.  });
```

- position

设置插件位置，如果目标文件已经设置了某插件，默认再次设置会覆盖掉。如果希望在已设插件执行之前插入或者之后插入，请传入 `prepend` 或者 `append`

```
1.  fis.match('*.*less', {  
2.      parser: fis.plugin('another', null, 'append');  
3.  });
```

## 配置属性

## 全局属性

全局属性通过 `fis.set` 设置，通过 `fis.get` 获取；

### 内置的默认配置

```
1. var DEFAULT_SETTINGS = {
2.   project: {
3.     charset: 'utf8',
4.     md5Length: 7,
5.     md5Connector: '_',
6.     files: ['**'],
7.     ignore: ['node_modules/**', 'output/**', '.git/**', 'fis-conf.js']
8.   },
9.
10.  component: {
11.    skipRoadmapCheck: true,
12.    protocol: 'github',
13.    author: 'fis-components'
14.  },
15.
16.  modules: {
17.    hook: 'components',
18.    packager: 'map'
19.  },
20.
21.  options: {}
22. };
```

### project.charset

- 解释：指定项目编译后产出文件的编码。
- 值类型： `string`
- 默认值： `'utf8'`
- 用法：在项目的 `fis-conf.js` 里可以覆盖为

```
1. fis.set('project.charset', 'gbk');
```



使用 `charset` 编码需要使用`encoding`插件发布编译结果

## project.md5Length

- 解释：文件MD5戳长度。
- 值类型： `number`
- 默认值： 7
- 用法：在项目的`fis-conf.js`里可以修改为

```
1.  fis.set('project.md5Length', 8);
```

## project.md5Connector

- 解释：设置md5与文件的连字符。
- 值类型： `string`
- 默认值： `-`
- 用法：在项目的`fis-conf.js`里可以修改为

```
1.  fis.set('project.md5Connector ', '.');
```

## project.files

- 解释：设置项目源码文件过滤器。
- 值类型： `Array`
- 默认值： `['**']`
- 用法：

```
1.  fis.set('project.files', ['*.html']);
```

## project.ignore

- 解释：排除某些文件
- 值类型： `Array`
- 默认值： `['node_modules/**', 'output/**', 'fis-conf.js']`
- 用法

```
1.  fis.set('project.ignore', ['*.bak']); // set 为覆盖不是叠加
```

## project.fileType.text

- 解释：追加文本文件后缀列表。
- 值类型： `Array` | `string`
- 默认值：无
- 说明：fis系统在编译时会对文本文件和图片类二进制文件做不同的处理，文件分类依据是后缀名。虽然内部已列出一些常见的文本文件后缀，但难保用户有其他的后缀文件，内部已列入文本文件后缀的列表为： [ `'css'`, `'tpl'`, `'js'`, `'php'`, `'txt'`, `'json'`, `'xml'`, `'htm'`, `'text'`, `'xhtml'`, `'html'`, `'md'`, `'conf'`, `'po'`, `'config'`, `'tmpl'`, `'coffee'`, `'less'`, `'sass'`, `'jsp'`, `'scss'`, `'manifest'`, `'bak'`, `'asp'`, `'tmp'` ]，用户配置会追加，而非覆盖内部后缀列表。
- 用法：编辑项目的fis-conf.js配置文件

```
1. fis.set('project.fileType.text', 'tpl, js, css');
```

## project.fileType.image

- 解释：追加图片类二进制文件后缀列表。
- 值类型： `Array` | `string`
- 默认值：无
- 说明：fis系统在编译时会对文本文件和图片类二进制文件做不同的处理，文件分类依据是后缀名。虽然内部已列出一些常见的图片类二进制文件后缀，但难保用户有其他的后缀文件，内部已列入文本文件后缀的列表为： [ `'svg'`, `'tif'`, `'tiff'`, `'wbmp'`, `'png'`, `'bmp'`, `'fax'`, `'gif'`, `'ico'`, `'jfif'`, `'jpe'`, `'jpeg'`, `'jpg'`, `'woff'`, `'cur'` ]，用户配置会追加，而非覆盖内部后缀列表。
- 用法：编辑项目的fis-conf.js配置文件

```
1. fis.set('project.fileType.image', 'swf, cur, ico');
```

## 文件属性

fis3 以文件属性控制文件的编译合并以及各种操作；文件属性包括基本属性和插件属性，插件属性是为了方便在不同的插件扩展点设置插件；

### 基本属性

- `release`

- `packTo`
- `packOrder`
- `query`
- `id`
- `url`
- `charset`
- `isHtmlLike`
- `isCssLike`
- `isJsLike`
- `useHash`
- `domain`
- `rExt`
- `useMap`
- `isMod`
- `extras`
- `requires`
- `useSameNameRequire`
- `useCache`
- `useCompile`

## release

- 解释：设置文件的产出路径。默认是文件相对项目根目录的路径，以 `/` 开头。该值可以设置为 `false`，表示为不产出（unreleasable）文件。
- 值类型： `string`
- 默认值：无

```
1.   fis.match('/widget/{*,**/*}.js', {
2.       isMod: true,
3.       release: '/static/$0'
4.   });
```

## packTo

- 解释：分配到这个属性的文件将会合并到这个属性配置的文件中
- 值类型： `string`
- 默认值：无

```
1.   fis.match('/widget/{*,**/*}.js', {
```

```

2.     packTo: '/static/pkg_widget.js'
3.   })

```

widget 目录下的所有 js 文件将会被合并到 /static/pkg\_widget.js 中。

**packTo** 设置的是源码路径，也会受到已经设置的 **fis.match** 规则的影响，比如可以配置 **fis.match**) 来更改 **packTo** 的产出路径或者 url；

```

1.   fis.match('/static/pkg_widget.js', {
2.     release: '/static/${namespace}/pkg/widget.js' //
    fis.set('namespace', 'home'),
3.     url: '/static/new/${namespace}/pkg/widget.js'
4.   })

```

## packOrder

- 解释：用来控制合并时的顺序，值越小越在前面。配合 **packTo** 一起使用。
- 值类型： **Integer**
- 默认值： **0**

```

1.  fis.match('/*.js', {
2.    packTo: 'pkg/script.js'
3.  })
4.
5.  fis.match('/mod.js', {
6.    packOrder: -100
7.  })

```

## query

- 解释：指定文件的资源定位路径之后的query，比如 '?t=123124132'。
- 值类型： **string**
- 默认值：无

```

1.   fis.set('new date', Date.now());
2.   fis.match('*.js', {
3.     query: '?t=' + fis.get('new date')
4.   });

```

## id

- 解释：指定文件的资源id。默认是 **namespace + subpath** 的值

- 值类型: `string`
- 默认值: `namespace + subpath`

如下方例子, 假设 `/static/lib/jquery.js` 设定了特定的 `id` `jquery`, 那么在使用这个组件的时候, 可以直接用这个 `id`;

```
1.  fis.match('/static/lib/jquery.js', {
2.      id: 'jquery',
3.      isMod: true
4.  });
```

使用

```
1.  var $ = require('jquery');
```

## moduleId

- 解释: 指定文件资源的模块id。在插件 `fis3-hook-module` 里面自动包裹 `define` 的时候会用到, 默认是 `id` 的值。
- 类型: `string`
- 默认值: `**namespace + subpath**`

```
1.  fis.match('/static/lib/a.js', {
2.      id: 'a',
3.      moduleId: 'a'
4.      isMod: true
5.  });
```

编译前

```
1.  exports.a = 10
```

编译后

```
1.  define('a',function(require,exports,module){
2.      exports.a = 10
3.  })
```

## url

- 解释：指定文件的资源定位路径，以 / 开头。默认是 `release` 的值，url可以与发布路径 `release` 不一致。
- 值类型： `string`
- 默认值：无

```
1.   fis.match('*.{js,css}', {
2.       release: '/static/$0',
3.       url: '/static/new_project/$0'
4.   })
```

## charset

- 解释：指定文本文件的输出编码。默认是 `utf8`，可以制定为 `gbk` 或 `gb2312`等。
- 值类型： `string`
- 默认值：无

```
1.   fis.match('some/file/path', {
2.       charset: 'gbk'
3.   });
```

使用 `charset` 编码需要使用`encoding`插件发布编译结果

## isHtmlLike

- 解释：指定对文件进行 `html` 相关语言能力处理
- 值类型： `bool`
- 默认值：无

## isCssLike

- 解释：指定对文件进行 `css` 相关的语言能力处理
- 值类型： `bool`
- 默认值：无

## isJsLike

- 解释：指定对文件进行 `js` 相关的语言能力处理
- 值类型： `string`
- 默认值：无

## useHash

- 解释：文件是否携带 md5 戳
- 值类型： `bool`
- 默认值： `false`
- 说明：文件分配到此属性后，其 url 及其产出带 md5 戳；

```

1.   fis.match('*.css', {
2.       useHash: false
3.   });
4.
5.   fis.media('prod').match('*.css', {
6.       useHash: true
7.   });

```

- fis3 release 时不带hash
- fis3 release prod 时带 hash

## domain

- 解释：给文件 URL 设置 domain 信息
- 值类型： `string`
- 默认值：无
- 说明：如果需要给某些资源添加 cdn，分配到此属性的资源 url 会被添加 domain；

```

1.   fis.media('prod').match('*.js', {
2.       domain: 'http://cdn.baidu.com/'
3.   });

```

- fis3 release prod 时添加cdn

## rExt

- 解释：设置最终文件产出后的后缀
- 值类型： `string`
- 默认值：无
- 说明：分配到此属性的资源的真实产出后缀

```

1.   fis.match('*.less', {
2.       rExt: '.css'
3.   });

```

源码为 `.less` 文件产出后修改为 `.css` 文件；

## useMap

- 解释：文件信息是否添加到 `map.json`
- 值类型： `bool`
- 默认值：无
- 说明： 分配到此属性的资源出现在静态资源表中，现在对 `js`、`css` 等文件默认加入了静态资源表中；

```
1.   fis.match('logo.png', {
2.       useMap: true
3.   });
```

## isMod

- 解释：标示文件是否为组件化文件。
- 值类型： `bool`
- 默认值：无
- 说明：标记文件为组件化文件。被标记成组件化的文件会入`map.json`表。并且会对`js`文件进行组件化包装。

```
1.   fis.match('/widget/{*,**/*}.js', {
2.       isMod: true
3.   });
```

## extras

- 注释：在[静态资源映射表][ ]中的附加数据，用于扩展[静态资源映射表][ ]表的功能。
- 值类型： `Object`
- 默认值：无
- 说明：无

```
1.   fis.match('/page/layout.tpl', {
2.       extras: {
3.           isPage: true
4.       }
5.   });
```



## requires

- 注释：默认依赖的资源id表
- 值类型： `Array`
- 默认值：无

- 说明：

```
1.   fis.match('/widget/*.js', {
2.       requires: [
3.           'static/lib/jquery.js'
4.       ]
5.   });
```

## useSameNameRequire

- 注释：开启同名依赖
- 值类型： `bool`
- 默认值： `false`
- 说明：当设置开启同名依赖，模板会依赖同名css、js；js 会依赖同名 css，不需要显式引用。

```
1.   fis.match('/widget/**', {
2.       useSameNameRequire: true
3.   });
```

## useCache

- 注释： 文件是否使用编译缓存
- 值类型： `bool`
- 默认值： `true`
- 说明：当设置使用编译缓存，每个文件的编译结果均会在磁盘中保存以供下次编译使用。设置为 `false` 后，则该文件每次均会被编译。

```
1.   fis.match('**.html', {
2.       useCache: false
3.   });
```

## useCompile

- 注释: FIS是否对文件进行编译
- 值类型: `bool`
- 默认值: `true`
- 说明: 设置为 `false` 后文件会通过FIS发布, 但是FIS不对文件做任何修改

```
1.  fis.match('**.html', {  
2.      useCompile: false  
3.  });
```

## 插件属性

插件属性决定了匹配的文件进行哪些插件的处理;

- `lint`
- `parser`
- `preprocessor`
- `standard`
- `postprocessor`
- `optimizer`

### lint

启用 `lint` 插件进行代码检查

```
1.  fis.match('*.js', {  
2.      lint: fis.plugin('js', {  
3.  
4.      })  
5.  });
```

### 更多插件

### parser

启用 `parser` 插件对文件进行处理;

如编译less文件

```
1.  fis.match('*.less', {  
2.      parser: fis.plugin('less'), //启用fis-parser-less插件  
3.      rExt: '.css'  
4.  });
```

## 如编译sass文件

```
1. fis.match('*.sass', {
2.     parser: fis.plugin('node-sass'), //启用fis-parser-node-sass插件
3.     rExt: '.css'
4. });
```

### [更多插件](#)

## preprocessor

### 标准化前处理

```
1. fis.match('*.{css,less}', {
2.     preprocessor: fis.plugin('image-set')
3. });
```

### [更多插件](#)

## standard

自定义标准化，可以自定义 uri、embed、require 等三种能力，可自定义三种语言能力的语法；

### [更多插件](#)

## postprocessor

### 标准化后处理

```
1. fis.match('*.{js,tpl}', {
2.     postprocessor: fis.plugin('require-async')
3. });
```

### [更多插件](#)

## optimizer

启用优化处理插件，并配置其属性

```
1. fis.match('*.css', {
2.     optimizer: fis.plugin('clean-css')
3. });
```

## 打包阶段插件

打包阶段插件设置时必须分配给所有文件，设置时必须 `match` `::package`，不然不做处理。

```
1. fis.match('::package', {
2.   packager: fis.plugin('map'),
3.   spriter: fis.plugin('csssprites')
4. });
```

### prepackager

- 解释：打包预处理插件
- 值类型： `Array` | `fis.plugin` | `function`
- 默认值：无

- 用法：

```
1. fis.match('::package', {
2.   prepackager: fis.plugin('plugin-name')
3. });
```

### packager

- 解释：打包插件
- 值类型： `Array` | `fis.plugin` | `function`
- 默认值：无

- 用法：

```
1. fis.match('::package', {
2.   packager: fis.plugin('map')
3. });
```

### 例子

```
1. fis.media('prod').match('::package', {
2.   packager: fis.plugin('map')
3. });
```

`fis3 release prod` 当在 `prod` 状态下进行打包

## spriter

- 解释：打包后处理csssprites的插件。
- 值类型： `Array` | `fis.plugin` | `function`
- 默认值：无
- 用法：

```
1.  fis.match('::package', {
2.      spriter: fis.plugin('csssprites')
3.  })
```

## 例子

```
1.  fis.media('prod').match('::package', {
2.      spriter: fis.plugin('csssprites')
3.  });
```

`fis3 release prod` 当在 prod 状态下进行 csssprites 处理

## postpackager

- 解释：打包后处理插件。
- 值类型： `Array` | `fis.plugin` | `function`
- 默认值：无
- 用法：

```
1.  fis.match('::package', {
2.      postpackager: fis.plugin('plugin-name')
3.  })
```

## 例子

```
1.  fis.media('prod').match('::package', {
2.      postpackager: fis.plugin('plugin-name')
3.  });
```

`fis3 release prod` 当在 prod 状态下调用打包后处理插件

## deploy

- 解释：设置项目发布方式

- 值类型: `Array` | `fis.plugin` | `function`
- 默认值: `fis.plugin('local-deliver')`
- 说明: 编译打包后, 新增发布阶段, 这个阶段主要决定了资源的发布方式, 而这些方式都是以插件的方式提供的。比如你想一键部署到远端或者是把文件打包到 Tar/Zip 又或者是直接进行 Git 提交, 都可以通过设置此属性, 调用相应的插件就能搞定了。
- 用法:

假设项目开发完后, 想部署到其他机器上, 我们选择 http 提交数据的方式部署

```
1.   fis.match('**', {
2.       deploy: fis.plugin('http-push', {
3.           receiver: 'http://target-host/receiver.php', // 接收端
4.           to: '/home/work/www' // 将部署到服务器的这个目录下
5.       })
6.   })
```

- 常用插件
  - [local-deliver](#)
  - [http-push](#)
  - [replace](#)
  - [encoding](#)

## 常用配置

### FIS3 常用配置

#### 制定目录规范

相信在前端工程化开发中，目录规范是必不可少的，比如哪些目录下是组件，哪些目录下的 js 要被特殊的插件处理，满足特殊的需求，比如对 commonjs、AMD 的支持。

这一节给大家介绍目录规范的制定，把它跟部署目录衔接起来；

##### 源码目录规范

```
1.  .
2.  |— page
3.  |   |— index.html
4.  |— static
5.  |   |— lib
6.  |— test
7.  |— widget
8.  |   |— header
9.  |   |— nav
10. |— ui
```

- page 放置页面模板
- widget 一切组件，包括模板、css、js、图片以及其他前端资源
- test 一些测试数据、用例
- static 放一些组件公用的静态资源
- static/lib 放置一些公共库，例如 jquery, zepto, lazyload 等

当编译产出时，产出结果目录是这样的；

```
1.  .
2.  |— static
3.  |— template
4.  |— test
```

- static 所有的静态资源都放到这个目录下
- template 所有的模板都放到这个目录下
- test 还是一些测试数据、用例

那么，我们的源码目录规范的指定是为了我们好维护，其产出目录规范是为了我们容易部署。

用 fis3 可以很方便的搞定这个事情；

*fis-conf.js*

```
1.  // 所有的文件产出到 static/ 目录下
2.  fis.match('*', {
3.      release: '/static/$0'
4.  });
5.
6.  // 所有模板放到 tempalte 目录下
7.  fis.match('*.html', {
8.      release: '/template/$0'
9.  });
10.
11. // widget源码目录下的资源被标注为组件
12. fis.match('/widget/**/*', {
13.     isMod: true
14. });
15.
16. // widget下的 js 调用 jswrapper 进行自动化组件化封装
17. fis.match('/widget/**/*.*js', {
18.     postprocessor: fis.plugin('jswrapper', {
19.         type: 'commonjs'
20.     })
21. });
22.
23. // test 目录下的原封不动产出到 test 目录下
24. fis.match('/test/**/*', {
25.     release: '$0'
26. });
```

这样就完成了目录规范的制定

等等，可能我们还需要做一些优化，来实现对整个工程的优化；

需要做以下几个方面的事情

- js, css 在开发时不压缩，但在产品发布时压缩
- 代码进行合理的合并处理

```
1.  // 所有的文件产出到 static/ 目录下
```



```
2.  fis.match('*', {
3.      release: '/static/$0'
4.  });
5.
6.  // 所有模板放到 tempalte 目录下
7.  fis.match('*.html', {
8.      release: '/template/$0'
9.  });
10.
11. // widget源码目录下的资源被标注为组件
12. fis.match('/widget/**/*', {
13.     isMod: true
14. });
15.
16. // widget下的 js 调用 jswrapper 进行自动化组件化封装
17. fis.match('/widget/**/*js', {
18.     postprocessor: fis.plugin('jswrapper', {
19.         type: 'commonjs'
20.     })
21. });
22.
23. // test 目录下的原封不动产出到 test 目录下
24. fis.match('/test/**/*', {
25.     release: '$0'
26. });
27.
28. // optimize
29. fis.media('prod')
30.     .match('*.js', {
31.         optimizer: fis.plugin('uglify-js', {
32.             mangle: {
33.                 expect: ['require', 'define', 'some string'] //不想被压的
34.             }
35.         })
36.     })
37.     .match('*.css', {
38.         optimizer: fis.plugin('clean-css', {
39.             'keepBreaks': true //保持一个规则一个换行
40.         })
41.     });
42.
43. // pack
```

```

44. fis.media('prod')
45.    // 启用打包插件, 必须匹配 ::package
46.    .match('::package', {
47.        packager: fis.plugin('map'),
48.        spriter: fis.plugin('csssprites', {
49.            layout: 'matrix',
50.            margin: '15'
51.        })
52.    })
53.    .match('*.js', {
54.        packTo: '/static/all_others.js'
55.    })
56.    .match('*.css', {
57.        packTo: '/staitc/all_others.css'
58.    })
59.    .match('/widget/**/*.*js', {
60.        packTo: '/static/all_comp.js'
61.    })
62.    .match('/widget/**/*.*css', {
63.        packTo: '/static/all_comp.css'
64.    });

```

- 发布 `fis3 release prod` , 进行合并、压缩等优化
- 发布 `fis3 release` 不做压缩不做合并

## 部署远端测试机

由于前端项目的特殊性, 一般都需要放到服务器上去运行, 那么在本本地开发完成后, 需要把编译产出部署到测试远端机器上面去, 这节就给大家分享一下在 `fis3` 这个操作怎么做;

在 `fis3` 中用 `fis.media` 提供各个状态区分, 那么我们也可以轻松制定不同状态下的发布方式; 比如要部署到 `qa` 的机器上抑或是 `rd` 的机器;

准备工作, 我们先选定自己需要使用的 `deploy` 插件, 在 `fis3` 部署方式都是用插件实现的;

- `fis3-deploy-http-push`

这个插件就是以 HTTP 提交的方式来完成远端部署的, 当然由于安全性等原因这种方式只适用于测试阶段, 请勿直接拿来上线;

HTTP 提交的方式上传就得有一个接受端, `http-push` 提供了一个 php 版本的接收端 `receiver.php` , 其他后端可以模仿实现一个。这个接收端需要放到你的 web 服务 `www` 目录下, 并且可以被访问到;

部署好接收端，并且它能正常被访问到，比如 url 是 `http://receiver.php` 其配置如下

```
1. fis.media('qa').match('**', {
2.     deploy: fis.plugin('http-push', {
3.         receiver: 'http://receiver.php',
4.         to: '/home/work/www'
5.     })
6. });
```

- `fis3 release qa` 当执行时就会部署到配置的 qa 的机器上。

## 异构语言 less 的使用

```
1. fis.match('**.less', {
2.     parser: fis.plugin('less'), // invoke `fis-parser-less`,
3.     rExt: '.css'
4. });
```

## 异构语言 sass 的使用

```
1. fis.match('**.sass', {
2.     parser: fis.plugin('sass'), // invoke `fis-parser-sass`,
3.     rExt: '.css'
4. });
```

## 前端模板的使用

```
1. fis.match('**.tmpl', {
2.     parser: fis.plugin('utc'), // invoke `fis-parser-utc`
3.     isJsLike: true
4. });
```

## 某些资源不产出

```
1. fis.match('*.inline.css', {
2.     // 设置 release 为 FALSE, 不再产出此文件
3.     release: false
4. });
```

## 某些资源从构建中去除

FIS3 会读取全部项目目录下的资源，如果有些资源不想被构建，通过以下方式排除。

```
1. fis.set('project.ignore', [  
2.   'output/**',  
3.   'node_modules/**',  
4.   '.git/**',  
5.   '.svn/**'  
6. ]);
```

# glob

## glob

FIS3 中支持的 glob 规则，FIS3 使用 [node-glob](#) 提供 glob 支持。

### 简要说明

[node-glob](#) 中的使用方式有很多，如果要了解全部，请前往 [node-glob](#)。

这里把常用的一些用法做说明。

- `*` 匹配0或多个除了 `/` 以外的字符
- `?` 匹配单个除了 `/` 以外的字符
- `**` 匹配多个字符包括 `/`
- `{ }` 可以让多个规则用 `,` 逗号分隔，起到 **或者** 的作用
- `!` 出现在规则的开头，表示取反。即匹配不命中后面规则的文件

需要注意的是，fis 中的文件路径都是以 `/` 开头的，所以编写规则时，请尽量严格的以 `/` 开头。

当设置规则时，没有严格的以 `/` 开头，比如 `a.js`，它匹配的是所有目录下面的 `a.js`，包括：`/a.js`、`/a/a.js`、`/a/b/a.js`。如果要严格只命中根目录下面的 `/a.js`，请使用 `fis.match('/a.js')`。

另外 `/foo/*.js`，只会命中 `/foo` 目录下面的所有 js 文件，不包含子目录。

而 `/foo/**/*.js` 是命中所有子目录以及其子目录下面的所有 js 文件，不包含当前目录下面的 js 文件。

如果需要命中 `foo` 目录下面以及所有其子目录下面的 js 文件，请使用 `/foo/**/*.js`。

### 扩展的规则

1. 假设匹配 `widget` 目录下以及其子目录下的所有 js 文件，使用 `node-glob` 需要这么写

```
1. widget/{*.js,**/*.js}
```

这样写起来比较麻烦，所以扩展了这块的语法，以下方式等价于上面的用法

```
1. widget/**/*.js
```

2. `node-glob` 中没有捕获分组，而 `fis` 中经常用到分组信息，如下面这种正则用法：

```
1. // 让 a 目录下面的 js 发布到 b 目录下面，保留原始文件名。
2. fis.match(/^\/a\/(.*\..js$)/i, {
3.   release: '/b/$1'
4. });
```

由于原始 `node-glob` 不支持捕获分组，所以做了对括号用法的扩展，如下用法和正则用法等价。

```
1. // 让 a 目录下面的 js 发布到 b 目录下面，保留原始文件名。
2. fis.match('/a/(**.js)', {
3.   release: '/b/$1'
4. });
```

## 捕获分组

使用 `node-glob` 捕获的分组，可以用于其他属性的设定，如 `release`，`url`，`id` 等。使用的方式与正则替换类似，我们可以用 `$1`，`$2`，`$3` 来代表相应的捕获分组。其中 `$0` 代表的是 `match` 到的整个字符串。

```
1. fis.match('/a/(**.js)', {
2.   release: '/b/$1' // $1 代表 (**.js) 匹配的内容
3. });
```

```
1. fis.match('/a/(**.js)', {
2.   release: '/b/$0' // $0 代表 /a/(**.js) 匹配的内容
3. });
```

## 特殊用法（类 `css` 伪类）

- i. `::package` 用来匹配 `fis` 的打包过程。
- ii. `::text` 用来匹配文本文件。

默认识别这类后缀的文件。

```
1. [
2.   'css', 'tpl', 'js', 'php',
3.   'txt', 'json', 'xml', 'htm',
4.   'text', 'xhtml', 'html', 'md',
```

```

5.  'conf', 'po', 'config', 'tpl',
6.  'coffee', 'less', 'sass', 'jsp',
7.  'scss', 'manifest', 'bak', 'asp',
8.  'tmp', 'haml', 'jade', 'aspx',
9.  'ashx', 'java', 'py', 'c', 'cpp',
10. 'h', 'cshhtml', 'asax', 'master',
11. 'ascx', 'cs', 'ftl', 'vm', 'ejs',
12. 'styl', 'jsx', 'handlebars'
13. ]

```

如果你希望命中的文件类型不在列表中，请通过扩展，多个后缀用 `,` 分割。

```
fis.set('project.fileType.text')
```

```
1. fis.set('project.fileType.text', 'cpp,hhp');
```

iii. `::image` 用来匹配文件类型为图片的文件。

默认识别这类后缀的文件。

```

1. [
2.  'svg', 'tif', 'tiff', 'wbmp',
3.  'png', 'bmp', 'fax', 'gif',
4.  'ico', 'jfif', 'jpe', 'jpeg',
5.  'jpg', 'woff', 'cur', 'webp',
6.  'swf', 'ttf', 'eot', 'woff2'
7. ]

```

如果你希望命中的文件类型不在列表中，请通过

`fis.set('project.fileType.image')` 扩展，多个后缀用 `,` 分割。

```
1. fis.set('project.fileType.image', 'raw,bpg');
```

iv. `*.html:js` 用来匹配命中的 html 文件中的内嵌的 js 部分。

fis3 htmlLike 的文件内嵌的 js 内容也会走单文件编译流程，默认只做标准化处理，如果想压缩，可以进行如下配置。

```

1. fis.match('*.html:js', {
2.   optimizer: fis.plugin('uglify-js')
3. });

```

v. `*.html:css` 用来匹配命中的 html 文件中内嵌的 css 部分。

fis3 htmlLike 的文件内嵌的 css 内容也会走单文件编译流程，默认只做标准化处理，如果想压缩，可以进行如下配置。

```
1. fis.match('*.html:css', {
2.   optimizer: fis.plugin('clean-css')
3. });
```

vi. `*.html:inline-style` 用来匹配命中的 html 文件中的内联样式。可以配置些 `auto prefix` 之类的插件。

vii. `*.html:scss` 用来命中 html 文件中的 scss 部分，具体请参考 [fis3-demo](#) 中的 `use-xlang`

## 注意事项

`fis3` 小于 3.3.4 的版本需要注意，3.3.4 以上的版本已修复此问题。

给 `node-glob` 扩展分组功能确实还存在缺陷。分组 `()` 与 或 `{}` 搭配使用时存在问题。

比如：`/a/({b,c}/**/*.js)` 会拆分成并列的两个规则 `/a/(b/**/*.js)` 和 `/a/(c/**/*.js)`，当这两个合成一个正则的时候，这个时候问题来了，一个分组变成了两个分组，分组 1 为 `(b/**/*.js)` 分组 2 为 `(c/**/*.js)`。那么当希望获取捕获信息时，不能按原来的分组序号去获取了。

```
1. // 错误
2. fis.match('/a/({b,c}/**/*.js)', {
3.   release: '/static/$1'
4. });
5.
6. // 正确
7. fis.match('/a/({b,c}/**/*.js)', {
8.   release: '/static/$1$2'
9. });
```



## 内置插件及配置

### FIS3 内置插件及配置

`fis3` 中内嵌了很多常用的插件。

- [fis-optimizer-clean-css](#)
- [fis-optimizer-png-compressor](#)
- [fis-optimizer-uglify-js](#)
- [fis-spriter-csssprites](#)
- [fis3-deploy-local-deliver](#)
- [fis3-deploy-http-push](#)
- [fis3-hook-components](#)
- [fis3-packager-map](#)

可以连接到仓库介绍页面查看详情，这里将概要描述这些插件的作用及基本配置。

#### [fis-optimizer-clean-css](#)

用于压缩 `css`，一般用于发布产品库代码。

```
1. fis
2.   .media('prod')
3.   .match('*.css', {
4.     optimizer: fis.plugin('clean-css')
5.   });
```

配置项说明，请参考 [how-to-use-clean-css-programmatically](#)。

```
1. fis
2.   .media('prod')
3.   .match('*.css', {
4.     optimizer: fis.plugin('clean-css', {
5.       keepBreaks: true,
6.
7.       // 更多其他配置
8.     })
9.   });
```

## fis-optimizer-png-compressor

用来压缩 png 文件，减少文件体积，详情请见 [pngcrush](#) 和 [pngquant](#) 说明。

```
1.  fis
2.    .media('prod')
3.    .match('*.png', {
4.      optimizer: fis.plugin('png-compressor', {
5.
6.        // pngcrush or pngquant
7.        // default is pngcrush
8.        type : 'pngquant'
9.      })
10.   });
```

## fis-optimizer-uglify-js

用来压缩 js 文件，混淆代码，减少文件体积。

```
1.  fis
2.    .media('prod')
3.    .match('*.js', {
4.      optimizer: fis.plugin('uglify-js', {
5.
6.        // https://github.com/mishoo/UglifyJS2#compressor-options
7.      })
8.    });
```

### 配置说明

## fis-spriter-csssprites

针对 css 规则中的 background-image 做图片优化，将多张零碎小图片合并，并自动修改 css 背景图片位置。

此插件并不会处理所有的 background-image 规则，而只会处理 url 中带 `?__sprite` 图片的规则。

```
1.  li.list-1::before {
2.    background-image: url('./img/list-1.png?__sprite');
3.  }
4.
```

```
5. li.list-2::before {
6.   background-image: url('../img/list-2.png?__sprite');
7. }
```

```
1. // 启用 fis-spriter-csssprites 插件
2. fis.match('::package', {
3.   spriter: fis.plugin('csssprites')
4. })
5.
6. // 对 CSS 进行图片合并
7. fis.match('*.css', {
8.   // 给匹配到的文件分配属性 `useSprite`
9.   useSprite: true
10. });
```

详情请查看 [fis-spriter-csssprites](#)

## [fis3-deploy-local-deliver](#)

用来支持 fis3 本地部署能力，将 fis3 编译产出到指定目录。

```
1. fis.match('*.js', {
2.   deploy: fis.plugin('local-deliver', {
3.     to: '/var/www/myApp'
4.   })
5. })
```

## [fis3-deploy-http-push](#)

用来支持 fis3 远程部署能力，将 fis3 编译通过 http post 方式发送到远程服务端。

```
1. fis.match('*.js', {
2.   deploy: fis.plugin('http-push', {
3.
4.     // 如果配置了receiver, fis会把文件逐个post到接收端上
5.     receiver: 'http://www.example.com:8080/receiver.php',
6.
7.     // 这个参数会作为文件路径前缀附加在 $_POST['to'] 里面。
8.     to: '/home/fis/www'
9.   })
10. })
```

## fis3-hook-components

用来支持 `短路径` 引用安装到本地的 `component`。

如： `fis3 install bootstrap` 后，在页面中可以这么写。

```
1. <link xxx href="bootstrap/css/bootstrap-theme.css" />
2. <script src="bootstrap/button.js"></script>
3.
4. <!--或者直接用模块化的方式引用 js-->
5. <script type="text/javascript">
6.     require(['bootstrap', 'bootstrap/button']);
7. </script>
```

此功能已自动开启。

## fis3-packager-map

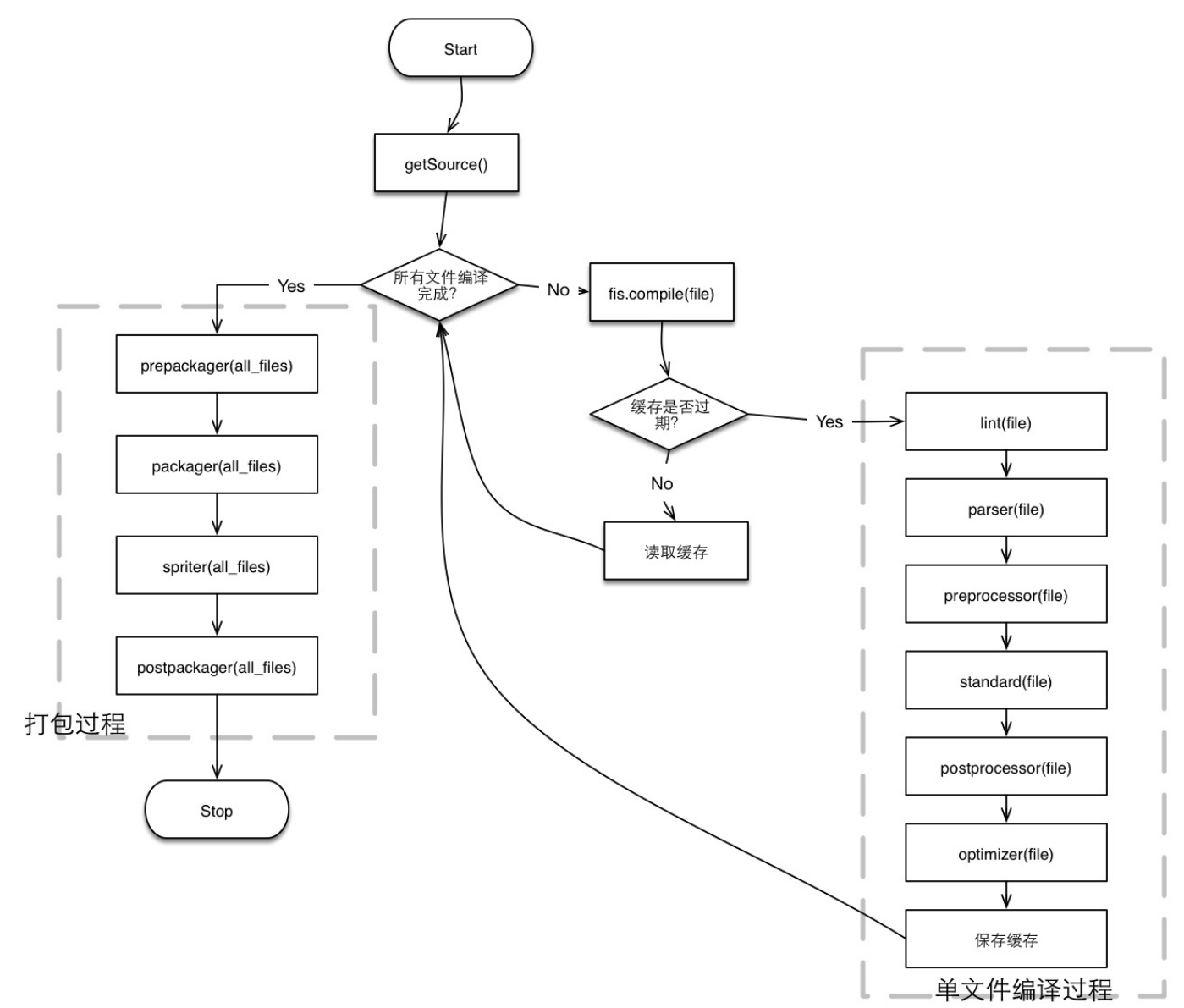
用来支持 `fis` 简单的打包，无需额外设置，已自动开启。

```
1. fis.match('*.css', {
2.     packTo: '/pkg/all.css'
3. });
4.
5. fis.match('*.js', {
6.     packTo: '/pkg/all.js'
7. });
```

# 自定义插件

## 插件开发

FIS3 是以 File 对象为中心构建编译的，每一个 File 都要经历编译、打包、发布三个阶段。[运行原理](#)讲述了 FIS3 中的插件扩展点；那么本节就将说明一个插件如何开发；



如上图，编译起初，扫描项目目录下的所有文件（不包含指定排除文件），后实例化 File 对象，并对 File 内容进行编译分析；

## 编译阶段插件

在编译阶段，文件是单文件进行编译的，这个阶段主要是对文件内容的编译分析；这个阶段分为 `lint`、`parser`、`preprocessor`、`postprocessor`、`optimizer` 等插件扩展点。

对于这些插件扩展点，可详见文档 [单文件编译流程](#)

其扩展点插件接口比较简单；

```

1.  /**
2.   * Compile 阶段插件接口
3.   * @param {string} content    文件内容
4.   * @param {File}   file       fis 的 File 对象 [fis3/lib/file.js]
5.   * @param {object} settings   插件配置属性
6.   * @return {string}           处理后的文件内容
7.   */
8. module.exports = function (content, file, settings) {
9.     return content;
10. };

```

为了搞清楚哪些功能用那种类型的插件去实现比较好，建议详细阅读[单文件编译流程](#) 这篇文档。

fis 的插件是以 NPM 包的形式提供的，这将意味着 fis 的插件都是一个 NPM 包，并且最终也需要发布到 NPM 平台上。在开始之前你需要了解 node 是如何加载一个 NPM 包的

<https://nodejs.org/api/modules.html>

FIS3 不再强制用户必须把插件（一个 NPM 包）进行全局安装，可把插件安装到项目根目录（fis-conf.js 所在目录或 -root 指定的目录），这个遵循 node 加载一个包的规范即可。

```

1. my-proj/
2. my-proj/fis-conf.js
3. my-proj/node_modules/fis3-<type>-<name>/index.js

```

#### • 插件开发

```

1.  // vi my-proj/node_modules/fis3-<type>-<name>/index.js
2.
3.  module.exports = function (content, file, settings) {
4.      // 只对 js 类文件进行处理
5.      if (!file.isJsLike) return content;
6.      return content += '\n// build ' + Date.now()
7.  }

```

#### • 插件调用

```

1.  // fis-conf.js
2.  fis.match('*.js', {

```

```

3.     <type>: fis.plugin('<name>', {
4.         //conf
5.     })
6. })

```

- `<type>`

- 发布 NPM 包

<https://docs.npmjs.com/getting-started/publishing-npm-packages>

FIS团队建议，开发插件时为了方便可放到项目目录下，但发布到 NPM 后还是建议进行全局安装，一个团队使用的插件应该被固化的。

为了便捷解决一些简短功能，也可以插件功能直接写到 `fis-conf.js` 中：

```

1.  fis.match('*.js', {
2.      // 直接设置插件属性的值为插件处理逻辑
3.      postprocessor: function (content, file, settings) {
4.          return content += '\n// build ' + Date.now();
5.      }
6.  });

```

## 参考插件

- lint <https://www.npmjs.com/package/fis-lint-csslint>
- parser <https://www.npmjs.com/package/fis-parser-sass>
- preprocessor <https://www.npmjs.com/package/fis-preprocessor-image-set>
- standard
- postprocessor <https://www.npmjs.com/package/fis-postprocessor-jswrapper>
- optimizer <https://www.npmjs.com/package/fis-optimizer-clean-css>

## 打包阶段插件

原理请详细参考文档 [构建流程](#)。到打包阶段，所有的文件都经过了单文件处理，该压缩的已经被压缩，该预编译的也进行了预编译。这个阶段主要实现一些共性的功能，比如打包合并。所以插件接口也不太一样了。

```

1.  /**
2.   * 打包阶段插件接口
3.   * @param {Object} ret      一个包含处理后源码的结构
4.   * @param {Object} conf    一般不需要关心，自动打包配置文件

```

```

5.  * @param {Object} settings 插件配置属性
6.  * @param {Object} opt      命令行参数
7.  * @return {undefined}
8.  */
9.  module.exports = function (ret, conf, settings, opt) {
10.    // ret.src 所有的源码, 结构是 {'<subpath>': <File 对象>}
11.    // ret.ids 所有源码列表, 结构是 {'<id>': <File 对象>}
12.    // ret.map 如果是 spriter、postpackager 这时候已经能得到打包结果了,
13.    //          可以修改静态资源列表或者其他
14.  }

```

跟编译时打包一样，也可项目本地开发或者是直接写到 `fis-conf.js` 中。参考 [打包阶段插件](#)。其配置方式与单文件编译阶段插件配置方式不同。由于 `packager` 时所有文件都在处理之列，所以需要通以下方式配置；

```

1.  fis.match('::package', {
2.    <type>: fis.plugin('<name>')
3.  })

```

- `<type>`

## 参考插件

- prepackager <https://www.npmjs.com/package/fis-prepackager-widget-inline>
- packager <https://www.npmjs.com/package/fis3-packager-map>
- spriter <https://www.npmjs.com/package/fis-spriter-csssprites>
- postpackager <https://www.npmjs.com/package/fis-postpackager-simple>

## Deploy 插件

deploy 插件是一类比较特殊的插件，它的功能只是发布数据，比如发布到某个文件夹下或者是发布到远端服务器上，以及用什么方式发布（http, ftp, git等等），deploy 插件是异步模型的。

deploy 作用于某些文件或者全部文件；它依然以文件属性的方式分配给某些文件，比如：

```

1.  fis.match('*.js', {
2.    deploy: [
3.      fis.plugin('replace', {
4.        from: 'some-string',
5.        to: 'another-string'
6.      }),
7.      fis.plugin('local-deliver') // 发布到本地, 由 -d 参数制定目录

```



```
8.     ]
9.  })
```

也就是说可以在 `deploy` 阶段做一些转码、`replace` 这样的事情，最后一个插件必然是发布文件到磁盘或者远端的插件。

插件接口如下：

```
1.  /**
2.   * deploy 插件接口
3.   * @param {Object} options 插件配置
4.   * @param {Object} modified 修改了的文件列表（对应watch功能）
5.   * @param {Object} total 所有文件列表
6.   * @param {Function} next 调用下一个插件
7.   * @return {undefined}
8.   */
9.  module.exports = function(options, modified, total, next) {
10.      next(); //由于是异步的如果后续还需要执行必须调用 next
11.  };
```

参考插件

- <https://www.npmjs.com/package/fis3-deploy-local-deliver>
- <https://www.npmjs.com/package/fis3-deploy-http-push>
- <https://www.npmjs.com/package/fis3-deploy-encoding>
- <https://www.npmjs.com/package/fis3-deploy-zip>
- <https://www.npmjs.com/package/fis3-deploy-replace>

## 命令行插件

`fis3` 默认提供了 `release`、`server`、`inspect`、`init` 和 `install` 五个子命令，每个子命令都是通过独立 `npm` 包来完成，命名规范为 `fis3-command-xxxx`。如果希望扩展新的子命令如 `foo`，则需要开发 `npm` 包 `fis3-command-foo`，全局安装或者安装在对应的 `fis` 项目目录，代码请参考如下所示：

```
1.  exports.name = 'foo';
2.  exports.desc = 'description of foo.';
3.  exports.options = {
4.      '-h, --help': 'print this help message',
5.      '--files'    : 'some options.'
6.  };
7.
```

```
8. exports.run = function(argv, cli) {
9.   // 如果输入为 fis3 foo -h
10.  // 或者 fis3 foo --help
11.  // 则输出帮助信息。
12.  if (argv.h || argv.help) {
13.    return cli.help(exports.name, exports.options);
14.  }
15.
16.  // 可以通过 argv 知道命令行中有哪些参数以及是什么值。
17.  console.log('I am working..');
18. };
```

## 参考插件

- <https://github.com/fex-team/fis3-command-release>
- <https://github.com/fex-team/fis3-command-server>
- <https://github.com/fex-team/fis3-command-inspect>
- <https://github.com/fex-team/fis3-command-init>
- <https://github.com/fex-team/fis-command-install>

## 插件类型

- lint
- parser
- preprocessor
- standard
- postprocessor
- optimizer
- prepackager
- packager
- spriter
- postpackager
- deploy

# 从零开始

## 从零开始

从零开启一个 FIS3 项目，并学习 FIS3 如何使用。

### 目录

- [开始一个小项目](#)
- [静态资源压缩](#)
- [修改静态资源发布路径](#)

### 开始一个小项目

- 创建项目目录

```
1. $ mkdir my-proj
```

- 在项目中新建配置文件

在 *fis3* 中，文件处理流程都以配置文件为主，所以配置文件是必须要有的

```
1. $ cd my-proj #进入项目目录
2. $ echo '' > fis-conf.js #最初为空即可
```

通过以上两步，我们建了一个空的 `fis3` 项目

```
1. my-proj/
2. my-proj/fis-conf.js
```

现在我们要开始写代码了，我们需要完成以下几个任务；

新建一个页面 `index.html` 并且引入样式文件 `style.css` 和 JS 脚本文件 `app.js`。并且发布这个项目，查看结果。

```
1. my-proj/index.html
2. my-proj/style.css
3. my-proj/app.js
4. my-proj/fis-conf.js
```

- 写代码

*index.html*

```

1.  <!DOCTYPE html>
2.  <html>
3.      <head>
4.          <title>Hello, world</title>
5.          <link rel="stylesheet" href="./style.css">
6.      </head>
7.      <body>
8.          <script type="text/javascript" src="./app.js"></script>
9.      </body>
10. </html>

```

*style.css*

```

1.  body {
2.      background-color: #FFCCFF;
3.  }

```

*app.js*

```

1.  alert("Hello, World");

```

- 发布项目

```

1.  $ cd my-proj
2.  $ fis3 release

```

`fis3 release -d <dest>` 通过 `-d` 参数指定产出目录，如果未制定并且 `fis-`  
`conf.js` 未配置 `deploy` 属性，则文件默认被发布到 `fis3 server open` 打开的目录

- 查看发布的结果

```

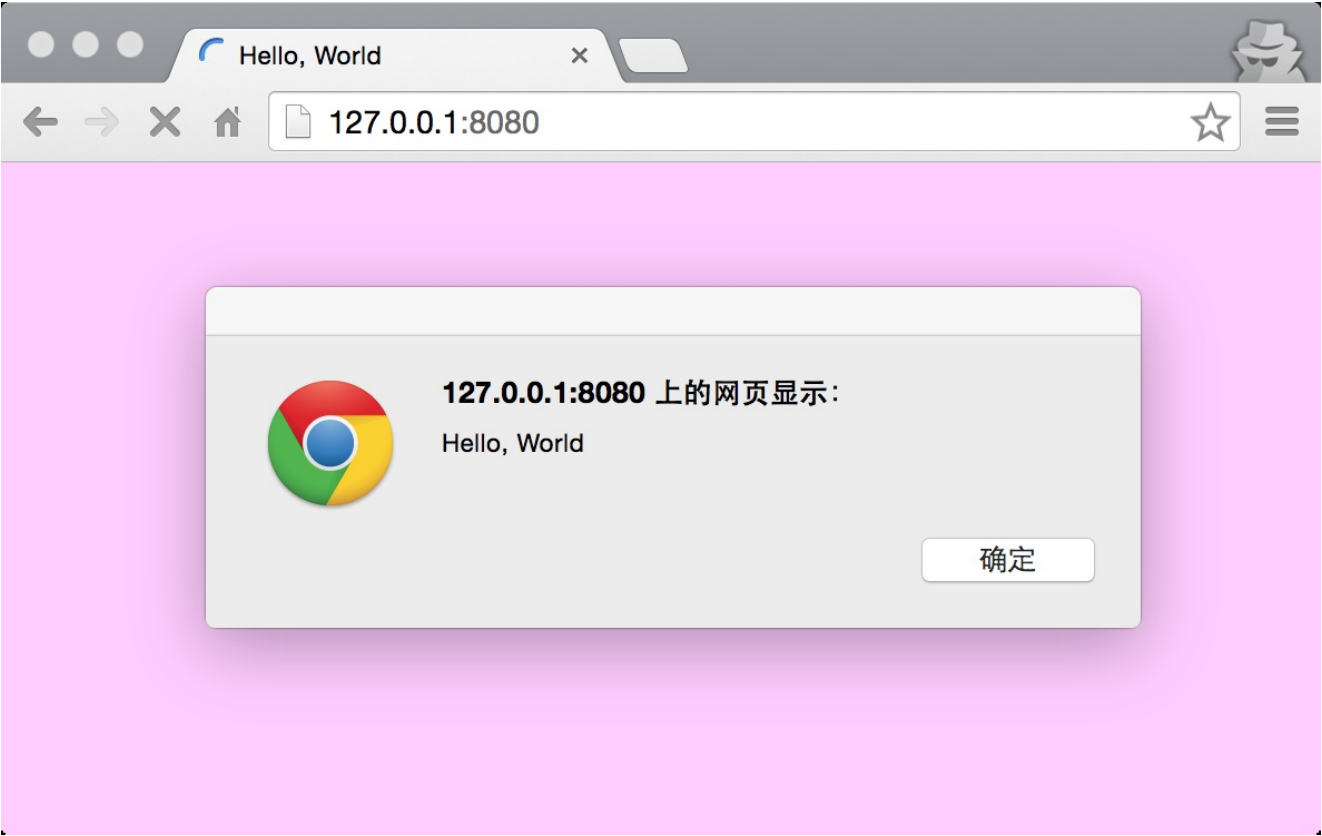
1.  fis3 server start

```

默认启动 `node` 的调试服务器，默认开启端口 `8080`

执行以上命令，会自动启动本地调试服务，并且打开默认浏览器访问

`http://127.0.0.1:8080` 地址



通过以上三步，我们的简易项目发布成功了，而且运行也很 OK。得到了我们想要的结果，那么 FIS3 在整个发布过程中干了什么？不妨比较下源码和产出，来直观感受下。

我们通过给 `release` 命令添加 `-d` 来把产出发布到源码目录的 `my-proj/output` 目录下。

```
1. # pwd == my-proj
2. $ fis3 release -d ./output
```

文件比较

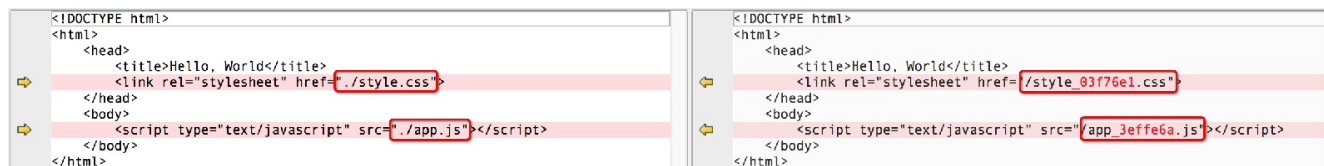
源码	产出
/app.js	/app_3effe6a.js
/style.css	/style_03f76e1.css
/index.html	/index.html
/fis-conf.js	-

- js、css 文件名携带了 `_<md5>`
- html 文件名不变

解释：fis3 默认对部分文件启用添加 md5 戳，其中包含 js、css、图片等，其余文件如果要添加 md5 戳，需要进行定制化配置。

文件内容

- js、css 无变化
- html



- 其中引入的 js、css 的 url 发生了变化

- 相对路径转化为绝对路径
- url 中携带 md5 戳，文件名也加上了对应的 md5 戳，正好能对应上

解释：fis3 中资源引入相对路径都会替换成绝对路径引入，主要是满足资源合并、文件产出路径按需变更。携带 md5 戳是为了方便开启强缓存时文件修改可刷新缓存。

我们做完了第一个小任务，创建了一个简单的项目，并且在默认情况下进行了发布，得到了以上结果。那么我们增加一些难度，来用 `fis-conf.js` 控制文件的编译，实现一些更有意思的事情。

## 静态资源压缩

继续操作 `my-proj` 项目，在其基础上实现 js、css 的压缩。

html 压缩有很多坑，不建议进行压缩，如果有这方面的需求，请参见插件 `fis-optimizer-html-minifier`

FIS3 中文件编译都是通过配置文件去进行控制的，那么其他代码不动，我们这节只关心配置文件 `fis-conf.js`。

- 修改配置文件

```

1.  // vi fis-conf.js
2.
3.  // 给所有 js 分配属性
4.  fis.match('/*,**/*}.js', {
5.      // 启用 fis-optimizer-uglify-js 进行压缩
6.      optimizer: fis.plugin('uglify-js')
7.  });
8.
9.  // 给所有 css 分配属性
10. fis.match('/*,**/*}.css', {
11.      // 启用 fis-optimizer-clean-css 进行压缩
12.      optimizer: fis.plugin('clean-css')
13.  });

```

- `fis.match()`

- 发布预览

和 `task 1` 一样，得到了正确的运行结果；

我们通过查看编译发布前后的代码变化来看看 FIS3 做了什么。

文件路径

源码	产出
/app.js	/app_3effe6a.js
/style.css	/style_1c8b450.css
/index.html	/index.html
/fis-conf.js	-

- 依然js、css 添加了 md5 戳不相同，这是因为此处的文档内容被压缩发生了变化

js 文件的相同，因为 `app.js` 的内容只有一句

`alert("Hello, World")`，压缩前后并没有变化

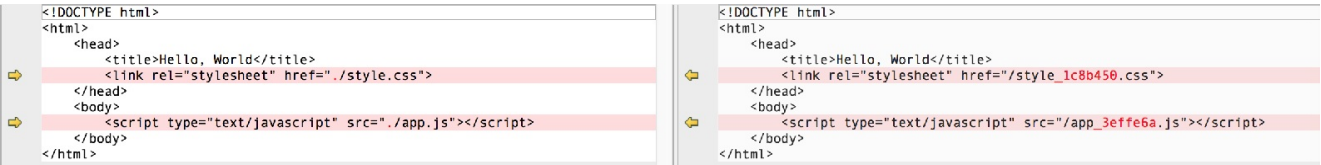
文件内容

- css 内容



编译时根据配置文件 `css` 文件分配到 `optimizer` 并且启用了 `clean-css` 进行压缩，压缩为一行。

- html 内容



- 依然修改 url 为绝对路径
- 资源 url 添加 md5 戳

通过给文件分配 `optimizer` 插件属性，文件在编译的过程中就会调用它，并执行这个插件属性配置的插件。

## 修改静态资源发布路径

修改发布路径，并且添加 **CDN**。

前端项目有这样一个特点，开发完成后都要部署到远端（除了移动打包发布APP），那么由于环境的种种限制，源码组织目录极大可能是和产出目录是对不上号的。比如源码中 `js`、`css` 散落在各地，而最终发布到线上都得放到统一的目录 `public` 或者是 `static` 或者得发布到统一的静态资源服务器上。

一般，这个工作会交给后端的某一个变量，来磨平这种差异；

```
1. <script src="<?=$static_root?>/app.js"></script>
```

或者是定规范使得源码目录规范和线上一模一样；

在 FIS3 里面，由于源码要经过发布，我们可以根据配置的规则，修改发布路径和引用 URL。

操作 `my-proj` 并且修改其配置文件来达到我们的目标。

## 修改发布路径

- 修改配置文件

```
1. // vi fis-conf.js
2.
3. // 给所有 js, css 分配属性
4. fis.match('{*,**/*}.{js,css}', {
5.     // js, css 发布后都放到 `public/static` 目录下。
6.     release: '/public/static/$0'
7. });
```

- `fis.match()`

- 编译发布查看

编译发布

```
1. fis3 release
```

查看结果

浏览器访问 <http://127.0.0.1:8080> 查看结果

我们通过查看编译发布前后的代码变化来看看 FIS3 做了什么。

文件路径



源码	产出
/app.js	/public/static/app_3effe6a.js
/style.css	/public/static/style_03f76e1.css
/index.html	/index.html
/fis-conf.js	-

我们得到了预期的结果 js、css 文件被最终产出到了 `public/static` 目录下。

## 文件内容

- js、css 文件内容没有发生变化，因为我们并没有给它们分配[插件属性](#)来启用某一个插件进行编译时处理。
- html js、css 产出目录发生了变化，那么引用的 url 是否也跟着变了

<pre>&lt;!DOCTYPE html&gt; &lt;html&gt;   &lt;head&gt;     &lt;title&gt;Hello, World&lt;/title&gt;     &lt;link rel="stylesheet" href="./style.css"&gt;   &lt;/head&gt;   &lt;body&gt;     &lt;script type="text/javascript" src="./app.js"&gt;&lt;/script&gt;   &lt;/body&gt; &lt;/html&gt;</pre>	<pre>&lt;!DOCTYPE html&gt; &lt;html&gt;   &lt;head&gt;     &lt;title&gt;Hello, World&lt;/title&gt;     &lt;link rel="stylesheet" href="/public/static/style_03f76e1.css"&gt;   &lt;/head&gt;   &lt;body&gt;     &lt;script type="text/javascript" src="/public/static/app_3effe6a.js"&gt;&lt;/s   &lt;/body&gt; &lt;/html&gt;</pre>
--	---

如我们所愿引用 url 也发生了变化，这种编译时处理能力，就是 FIS3 的[三种语言能力之定位资源能力](#)

可以方便的更改产出后的发布路径以及 url。

等等，我们在这块可能忘了一个事情，产出路径可能和 url 是不一样的。可以通过通过 `url` 属性进行调整。

```
1.  // vi fis-conf.js
2.
3.  // 给所有 js、css 分配属性
4.  fis.match('{*,**/*}.{js,css}', {
5.    // js, css 发布后都放到 `public/static` 目录下。
6.    release: '/public/static/$0',
7.    url: '/static/$0'
8.  });
```

- `fis.match()`

## 添加 CDN

- 修改配置文件

```
1.  fis.match('{*,**/*}.{js,css}', {
```

```

2.      // js, css 发布后都放到 `public/static` 目录下。
3.      release: '/static/$0',
4.      domain: 'http://127.0.0.1:8080'
5.  });

```

- `fis.match()`

1. 其实可能有人会疑问 `url` 里面添加 `domain` 不就行了，但是毕竟概念上要区分出来。所以按部就班，该是啥就是啥！

- 发布查看结果

编译发布

```
1.  fis3 release
```

查看结果

浏览器访问 <http://127.0.0.1:8080> 查看结果

我们通过查看编译发布前后的代码变化来看看 FIS3 做了什么。

文件路径

源码	产出
/app.js	/static/app_3effe6a.js
/style.css	/static/style_03f76e1.css
/index.html	/index.html
fis-conf.js	-

- js、css 按照 `release` 发布到了 static 目录下

文件内容

- js、css 不变
- html 引入的 js、css 的 url 发生了什么变化

<pre> &lt;!DOCTYPE html&gt; &lt;html&gt;   &lt;head&gt;     &lt;title&gt;Hello, World&lt;/title&gt;     &lt;link rel="stylesheet" href="./style.css"&gt;   &lt;/head&gt;   &lt;body&gt;     &lt;script type="text/javascript" src="./app.js"&gt;&lt;/script&gt;   &lt;/body&gt; </pre>	<pre> &lt;!DOCTYPE html&gt; &lt;html&gt;   &lt;head&gt;     &lt;title&gt;Hello, World&lt;/title&gt;     &lt;link rel="stylesheet" href="http://127.0.0.1:8080/static/style_03f76e1.css"&gt;   &lt;/head&gt;   &lt;body&gt;     &lt;script type="text/javascript" src="http://127.0.0.1:8080/static/app_3effe6a.js"&gt;&lt;/sc   &lt;/body&gt; </pre>
--	--

url 的前面添加了对应 cdn 的 `domain`

## 小结

似乎我们已经能轻松搞一个 FIS3 的前端项目了，而且我们也通过一个小例子，实现了静态资源的压缩以及更改产出目录。但我们想实现更多，比如需要

- 资源合并
- 模块化开发
- 保持相对路径
- 使用 less、scss
- 最低成本实现模块化方案
- BigPipe
- Quickling
- BigRender
- .....

那么这一切的事情都是有可能的，而且似乎也比较简单。那么如何让这一切变得简单，这就是 FIS3 的魅力。

- <https://github.com/fex-team/fis3-demo> 大量 fis3 的 Demo，几乎你想要的我们都给你提供了，参考实践即可。
- [配置 API](#) 配置 API，扩展 FIS3 以及配置控制文件的编译都靠这些 API。

## 文件合并

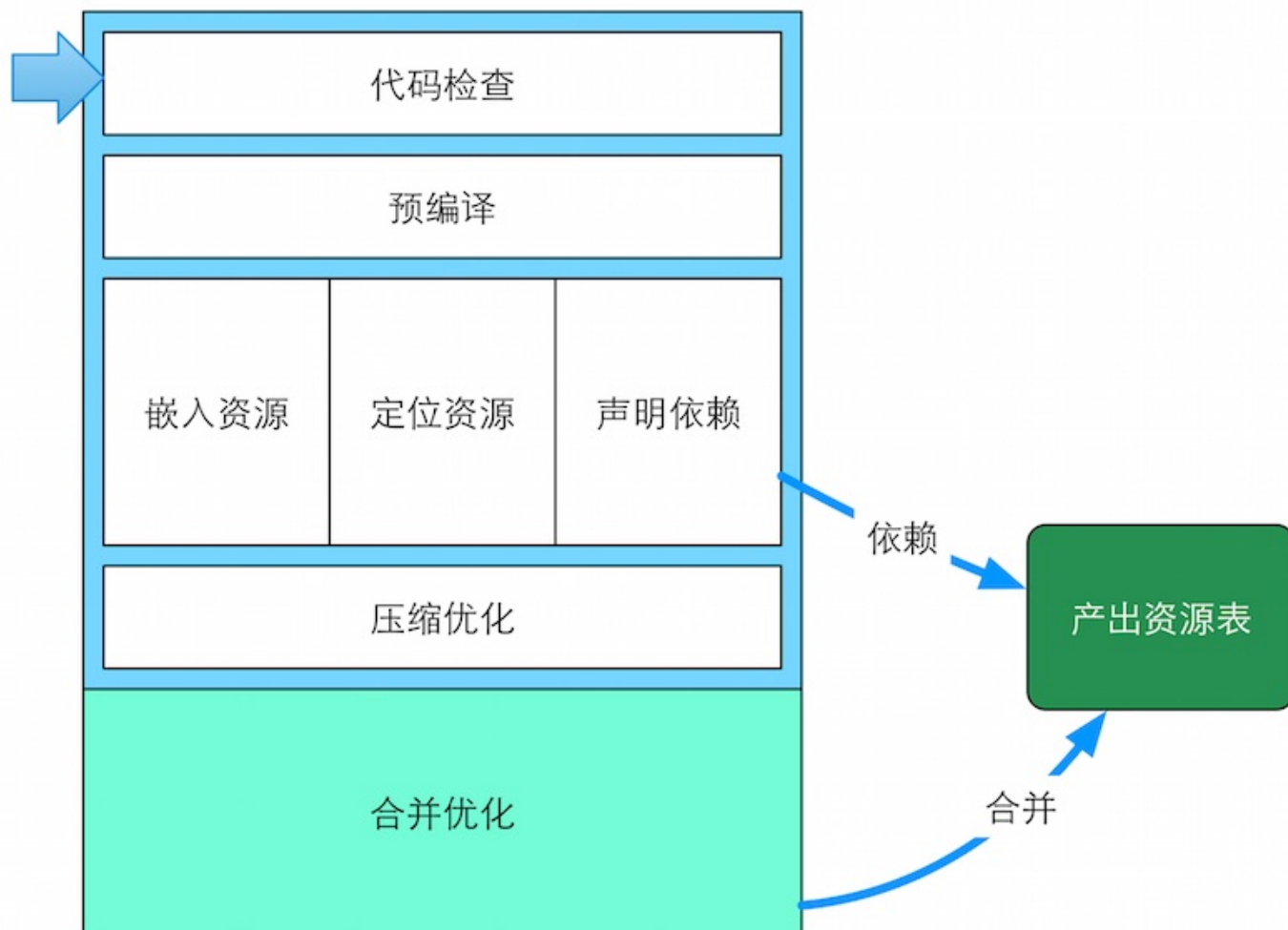
---

## 辅助开发

---

## 代码校验

## 代码校验



一致的代码格式规范利于代码维护性的提高，也是工程构建工具需要注重考虑的功能。

FIS3 通过不同校验插件来实现对不同语言的校验。

## 对 JS 进行校验

```
1. fis.match('*.js', {
2.   // 需要执行 npm install -g fis-lint-jshint 安装此插件
3.   lint: fis.plugin('jshint', {
4.     //ignored some files
5.     //ignored : 'static/libs/**/*.js',
6.     ignored : [ 'static/libs/**/*.js', /jquery\.js$/i ],
7.
8.     //using Chinese reporter
```

```
9.      i18n : 'zh-CN',
10.
11.      //jshint options
12.      camelcase : true,
13.      curly : true,
14.      eqeqeq : true,
15.      forin : true,
16.      immed : true,
17.      latedef : true,
18.      newcap : true,
19.      noarg : true,
20.      noempty : true,
21.      node : true
22.  })
23. });
```

## 对 CSS 进行校验

```
1.  fis.match('*.css', {
2.    // 需要执行 npm install -g fis-lint-csslint 安装此插件
3.    // https://github.com/stubbornella/csslint/wiki/Rules
4.    lint: fis.plugin('csslint', {
5.      /**
6.       * 报告为“WARNING”的规则ID列表，支持数组或以“,”分隔的字符串
7.       */
8.      warnings : ["rule1", "rule2", ...],
9.
10.     /**
11.      * 报告为“ERROR”的规则ID列表，支持数组或以“,”分隔的字符串
12.      */
13.     errors   : ["rule1", "rule2", ...],
14.
15.     /**
16.      * 若ie值为false，则忽略所有与IE兼容性相关的校验规则
17.      */
18.     ie       : false,
19.
20.     /**
21.      * 要忽略的规则ID列表，支持数组或以“,”分隔的字符串
22.      */
23.     ignore   : ["rule1", "rule2", ...]
24.   })
```

```
25. });
```

# FIS2 到 FIS3

## FIS2 to FIS3

*FIS* 以下统称为 *FIS2*

### 简介

FIS3相对FIS2来说接口改动较大，并不是不考虑版本上的兼容，而是不愿意做简单的小修小补，希望从整体的角度打造一个 **易用性和可扩展性** 达到一个全新高度的工具。FIS2 与FIS3将并行维护，并且绝大部分插件是兼容的。

### 功能升级点简介

#### RoadMap目录定制更简单

FIS2中roadmap是最先匹配生效的，如果想覆盖解决方案的默认配置比较麻烦。FIS3中使用了类似css的配置语法，使用叠加的机制，同一个配置最后一个生效：

```
1.  fis.match('{a,b}.js', {
2.      release: '/static/$0'
3.  });
4.
5.  fis.match('b.js', {
6.      release: '/static/new/$0'
7.  });
```

并且通过类似css media的API来控制不同用户、环境的配置：

```
1.  fis.media('prod')
2.      .match('*.js', {
3.          optimizer: fis.plugin('uglify-js')
4.      })
5.      .match('component_modules/*.js',{
6.          packTo: '/static/pkg/common.js'
7.      })
```

另外你还可以像css !important一样设置不可以被覆盖的配置：

```

1. //important设为true表示不可被覆盖
2. fis.match(selector, props[, important])

```

```

1. fis.match('*.js', {
2.   useHash: true,
3.   release: '/static/$0'
4. }, true);

```

## 支持本地插件

FIS2中插件都需要安装到全局才能使用，不方便自定义插件的开发和部署。FIS3中包括FIS3和插件都可以安装在项目本地使用。插件将优先使用本地的。

## 支持相对路径产出

从模块化开发和工程化部署的角度，我们并不推荐使用相对路径产出部署的方式，在FIS2中这个也是基本不支持的（可以关闭标准处理，但变成了一个压缩工具）。

考虑到一些用户的需求，FIS3中可以安装一个插件来实现相对目录的产出：

```

1. //全局或本地安装插件
2. npm install [-g] fis3-hook-relative
3.
4. //fis-conf.js
5. // 启用插件
6. fis.hook('relative');
7.
8. // 让所有文件，都使用相对路径。
9. fis.match('**', {
10.   relative: true
11. })

```

具体见插件说明 <https://github.com/fex-team/fis3-hook-relative>

## FIS3支持更好的按需编译

FIS2中可以通过 `include` 或者 `exclude` 来过滤源码，然而只有规则匹配的目录才生效。

FIS3中可以通过设置files过滤需要编译的源码，同时支持分析files中引用或依赖到的被过滤的资源目录中的文件。



```
1. fis.set('project.files', ['page/**', 'map.json', 'modules/**', 'lib']);
```

典型的使用场景如angular中，bower下面的资源有大量的冗余，并且可能导致编译失败。通过手工配置过滤比较麻烦，而通过files引用来分析资源就能屏蔽bower下载目录的情况下，自动找出其中被使用的资源。 示例见[fis3-demo](#)

## FIS3 不再默认解析 js 中的 `require()` 函数添加依赖

- 为什么这么做？

虽然现在很多模块化框架都以 `require` 来作为依赖，但是其形式是不同的。比如

### `require.js`

```
1. require(['./a.js'])
```

### `mod.js`

```
1. require.async('./a.js');
```

### `sea.js`

```
1. sea.use('./a.js');
```

- 如果要解析 `require()` 需要自己添加 `preprocessor` 插件支持，而这个插件的逻辑相当简单。

### 参考

<https://github.com/fex-team/fis3-hook-commonjs/blob/master/index.js>

- 如果感觉自己写插件太麻烦

可以安装 FIS组 提供的模块化方案的支持插件 `fis3-hook-commonjs`

```
1. fis.hook('commonjs');
2.
3. // 如果 jswrapper 自定义要做
4. fis.hook('commonjs', {wrap: false});
```

## 更简单的纯前端自动合并支持

FIS2纯前端方案中使用`fis-postpackager-simple` 来实现资源自动合并, fis3中我们提供了一个新的扩展插件来实现类似功能, 不仅支持灵活自动的打包配置, 而且能产出适配requireJS、modJS的资源依赖配置。

具体文件见插件: <https://github.com/fex-team/fis3-postpackager-loader>

## 默认关闭同名依赖功能

FIS2中组件化资源默认开启了同名依赖设置, 对初学者可能造成困惑。FIS3中默认关闭了同名依赖, 您需要主动声明资源间的依赖关系。不过你也可以通过配置来开启同名依赖功能:

```
1. fis.match('*.html,js', {
2.   useSameNameRequire: true
3. });
```

## 支持监听fis-conf.js修改自动重启

FIS2 中开启 `watch` 并修改fis-conf.js时, 工具不会自动重启, FIS3中能自动监听fis-conf.js的变化并给出提示, 自动重启。

## 支持内嵌异构语言分析

FIS2 中通过 `<script>` 或 `<style>` 内嵌的语言不能是less、sass等异构语言, fis3中支持直接内嵌异构语言:

```
1. <script type="text/x-coffee">
2.   //...
3. </script>
```

```
1. <style type="text/x-less">
2.   body {
3.     background-color: #F0F0F0;
4.     h1 {
5.       color: red;
6.     }
7.   }
8. </style>
```

## 更加人性化的错误提示

FIS3中对于常见的错误将给出提示, 并提供对应的github issue 链接, 您可以在相应地方寻找解决

办法。如果依旧没有解决欢迎提issue。

## FIS3 删减了部分命令行参数

FIS3 配置上很灵活，通过给文件分配属性，由这些属性控制编译流程。不像 FIS2 给 `release` 添加参数就能搞定很多事情了，比如所有静态资源压缩、加 md5、打包、加 domain 等，这些功能在 FIS3 中必须通过配置文件配置进行操控。

## FIS2 `release` `-o` `-p` `-D` `-m` 在 FIS3 如何施展

`fis release -o` 在 FIS3 中等价配置

```
1.  fis.match('*.js', {
2.    optimizer: fis.plugin('uglify-js')
3.  });
4.
5.  fis.match('*.css', {
6.    optimizer: fis.plugin('clean-css')
7.  });
8.
9.  fis.match('*.png', {
10.   optimizer: fis.plugin('png-compressor')
11. });
```

`fis release -D` 在 FIS3 中等价配置

```
1.  fis.match('*.js', {
2.    domain: 'http://cdn.baidu.com'
3.  })
```

`fis release -m` 在 FIS3 中等价配置

```
1.  fis.match('*.{js,css}', {
2.    useHash: true
3.  });
4.
5.  //命中所有的图片类文件，包括字体等
6.  fis.match('image', {
7.    useHash: true
8.  });
```

`fis release -p` 在 FIS3 中等价配置

```
1.  fis.match('::package', {
2.    packager: fis.plugin('some-pack-plugin'), // 挂载一个打包插件
3.    spriter: fis.plugin('csssprites') // FIS2 默认启用 csssprites
4.  });
5.
6.  fis.match('/widget/*.js', {
7.    packTo: '/static/widget_aio.js' // 匹配文件打包到此文件中
8.  });
```

## MapJSON 产出设置改变

FIS3中默认不产出map.json配置文件，但提供了一个标记符 `__RESOURCE_MAP__` 支持您将map.json产出到任意位置，例如您的项目根目录下有个文件manifest.json里面包含此字符，那么产出后mapjson 静态资源表就会在其中。

## FIS2 `fis.config.set('pack', {})` 合并配置在 FIS3 如何施展

FIS2中通过数组来添加打包配置。FIS3中则是通过属性来控制打包：

```
1.  //fis-conf.js
2.  fis.match('/widget/*.js', {
3.    packTo: '/static/widget_pkg.js'
4.  })
```

您也可以在match中填写数组来控制打包顺序。另外简单的纯前端打包需求可以通过[自动打包插件](#)来实现。

## 丰富的DEMO

FIS2中入门DEMO比较少，FIS3中我们提供了丰富的DEMO，从简单的合并压缩到纯前端的angular、react、vuejs等，到最终结合后端smarty、php和laravel框架的解决方案都有相应的demo。具体见 <https://github.com/fex-team/fis3-demo>

# 资源合并

## 资源合并（打包）

关于资源合并，在 fis3 中有多种方式来实现。为了搞清楚他们都有些什么特点，适用于什么场合，我觉得有必要聚集在一起一一说明下。

### packTo

命中目标文件，设置 packTo 即能完成简单的合并操作。

```
1. fis.match('/static/folderA/**/*.js', {
2.   packTo: '/static/pkg/folderA.js'
3. });
4.
5. fis.match('/static/folderA/**/*.css', {
6.   packTo: '/static/pkg/folderA.css'
7. });
```

合并的列表中，被依赖的文件会自动提前。但是并不是所有的资源都严格的指定了依赖，所以有时候需要控制顺序。可以通过配置 `packOrder` 来控制，`packOrder` 越小越在前面。

```
1. fis.match('/static/folderA/**/*.js', {
2.   packTo: '/static/pkg/folderA.js'
3. });
4.
5. fis.match('/static/folderA/file1.js', {
6.   packOrder: -100
7. });
```

这种打包方式最简单，但是对于顺序配置有点麻烦，如果 A 必须在 B 的前面，最好的方式是让 B 指定依赖 A。

### fis3-packager-map

packTo 其实用的就是这个插件，fis3 内部其实就是把 packTo 转成了这个插件的配置。

以下配置完全等价于上面 packTo 的配置。

```
1. fis.match('::package', {
```

```

2.   packager: fis.plugin('map', {
3.     '/static/pkg/folderA.js': '/static/folderA/**/*.js'
4.   })
5. })

```

为什么这种变体的配置方式，是因为用这种方式很好控制顺序。此插件会按配置的顺序来打包。

```

1.  fis.match('::package', {
2.    packager: fis.plugin('map', {
3.      '/static/pkg/folderA.js': [
4.        '/static/folderA/file1.js',
5.        '/static/folderA/file2.js',
6.        '/static/folderA/**/*.js'
7.      ]
8.    })
9.  })

```

## fis3-packager-deps-pack


deps-pack 是在 map 的基础上再扩展了用法，可以快速的命中目标文件的依赖，比如：

```

1.  fis.match('::package', {
2.    packager: fis.plugin('deps-pack', {
3.
4.      'pkg/hello.js': [
5.
6.        // 将 main.js 加入队列
7.        '/static/hello/src/main.js',
8.
9.        // main.js 的所有同步依赖加入队列
10.       '/static/hello/src/main.js:deps',
11.
12.        // 将 main.js 的所有异步依赖加入队列
13.        '/static/hello/src/main.js:asyncs',
14.
15.        // 移除 comp.js 的所有同步依赖
16.        '!/static/hello/src/comp.js:deps'
17.      ],
18.
19.      // 也可以从将 js 依赖中 css 命中。
20.      'pkg/hello.css': [
21.        // main.js 的所有同步依赖加入队列

```

```
22.     '/static/hello/src/main.js:deps',
23.   ]
24.
25.   })
26. });
```

以上示例中还有中特殊的用法，即  打头的规则。他可以在现有结果集中做排除处理。

这种打包方式最复杂，但是很多情况下，你需要这些规则来做打包细化。

## fis3-postpackager-loader

其实它并不是专门做打包的，而是做资源加载的插件。只是他能提供另一种更简单的打包方式。

他以页面为单位，分析当前页面用到的所有资源，将所有 js 合并成一个 js，所有的 css 合并成一个 css。

```
1. fis.match('::package', {
2.   postpackager: fis.plugin('loader', {
3.     allInOne: true
4.   })
5. });
```

非常简单粗暴，但是有两个缺点。

1. 因为他是前端编译期分析，对于使用了后端模板的页面资源分析无能为力，所以它只能适用于纯前端项目。
2. 它的资源合并是以页面为单位，所以存在公共 js/css 被复制成多份打包多个包里面，导致的结果是，页面间切换，公共部分的 js/css 是没有公用缓存的。

但是这个问题还是能结合前面提到的插件解决的。比如：

```
1. fis.match('/static/folderA/**/*.js', {
2.   packTo: '/static/pkg/folderA.js'
3. });
```

以上配置和 loader 的 allInOne 同时配置了的话，结果会是这样的， folderA 下面的资源被打包到 folderA.js 中，同时页面里面的其他资源走 allInOne 打包。

所以只要勤快点，是可以把公用的资源抽出来的。

# mock 假数据模拟

## Mock 假数据模拟

当后端开发人员还没有准备好后端接口时，为了能让前端项目开发继续进行下去，往往需要提供 **假数据** 来协助前端开发。

`fis` 中默认的 `node` 服务就支持此功能。

## 步骤

1. 准备好假数据文件，如 `sample.json` 文件，放在服务器的 `/mock/sample.json` 目录，确保通过 `http://127.0.0.1:8080/mock/sample.json` 可访问到。

```
1. {  
2.   "error": 0,  
3.   "message": "ok",  
4.   "data": {  
5.     "uname": "foo",  
6.     "uid": 1  
7.   }  
8. }
```

2. 准备一个 `server.conf` 配置文件，放在服务器目录的 `/mock/server.conf`，内容如下。

```
1. rewrite ^\/api\/user$ /mock/sample.json
```

3. 然后当你请求 `http://127.0.0.1:8080/api/user` 的时候，返回的就是 `sample.json` 中的内容。

## 说明

关于 `server.conf` 配置语法，格式如下：

1. 指令名称 正则规则 目标文件

- **指令名称** 支持 `rewrite`、`redirect` 和 `proxy`。



- **正则规则** 用来命中需要作假的请求路径。
- **目标文件** 设置转发的目标地址，需要配置一个可请求的 url 地址。

刚说的是把文件放在服务器目录，操作起来其实并不是很方便，这类假数据文件建议放在项目源码中，然后通过 fis3 release 伴随源码一起发布到调试服务器。

推荐以下存放规范。

1. └─ mock
2.     └─ sample.json
3.     └─ server.conf

## 直接代理到其他服务的 api 地址。

这个功能要求 fis3 >= 3.3.29 使用格式如：

1. proxy ^\s/api\s/test http://127.0.0.1:9119/api/test

ps：还在测试中，目前简单的 API 代理没问题，如果 auth 验证什么的可能就会跪，后续会继续优化。

## 动态假数据

静态的假数据可以通过 json 文件提供，那么动态数据怎么提供？node 服务器可以通过 js 的方式提供动态假数据。

/mock/dynamic.js

```
1. module.exports = function(req, res, next) {
2.
3.   res.write('Hello world ');
4.
5.   // set custom header.
6.   // res.setHeader('xxxx', 'xxx');
7.
8.   res.end('The time is ' + Date.now());
9. };
```

然后结合 server.conf 就可以模拟各种动态请求了。

1. rewrite ^\s/api\s/dynamic\s/time\$ /mock/dynamic.js

如上面的例子，当请求 `http://127.0.0.1:8080/api/dynamic/time` 时，返回：`Hello world`  
`The time is 1442556037130`

## 常用的插件列表

### 常用插件列表

---

#### parser 插件

##### [fis-parser-babel-5.x](#)

支持 es6、es7 或者 jsx 编译成 es5

```
1. fis.match('*.jsx', {
2.   parser: fis.plugin('babel-5.x')
3. })
```

##### [fis3-parser-typescript](#)

支持 typescript、es6 或者 jsx 编译成 js。速度相比 babel 略快，但是 es7 跟进较慢。

```
1. fis.match('*.jsx', {
2.   parser: fis.plugin('typescript')
3. })
```

##### [fis-parser-less-2.x](#)

支持 less 编译成 css

```
1. fis.match('*.less', {
2.   parser: fis.plugin('less-2.x'),
3.   rExt: '.css'
4. })
```

##### [fis-parser-node-sass](#)

支持 sass/scss 编译成 css。

```
1. fis.match('*.scss', {
2.   rExt: '.css',
3.   parser: fis.plugin('node-sass', {
4.     // options...
5.   })
```

```
6. })
```

## fis-parser-jdists

一款强大的代码块预处理工具。比如加上如下配置，在 `debug` 注释中的代码，在正式环境下自动移除。

```
1. fis.media('production').match('*.js', {
2.   parser: fis.plugin('jdists', {
3.     remove: "debug"
4.   })
5. })
```

```
1. /*<debug>*/
2. // 这段代码在 fis3 release production 的时候会被移除。
3. console.log(debug);
4. /*</debug>*/
```

## preprocessor 插件

### fis3-preprocessor-js-require-css

允许你在 js 中直接 `require` css 文件。

```
1. fis.match('*.js,es,es6,jsx,ts,tsx', {
2.   preprocessor: fis.plugin('js-require-css')
3. })
```

### fis3-preprocessor-js-require-file

允许你在 js 中直接 `require` 文件。比如图片，json，其他静态文件。

`require` 部分将会替换成部署后的 url。 同时还支持，如果文件小于 20K 直接替换成 base64 字符串。

```
1. fis.match('*.js,es,es6,jsx,ts,tsx', {
2.   preprocessor: fis.plugin('js-require-file')
3. })
```

### fis3-preprocessor-autoprefixer

自动给 css 属性添加前缀，让标准的 css3 支持更多的浏览器。

```
1. fis.match('*.{css,less,scss}', {
2.   preprocessor: fis.plugin('autoprefixer', {
3.     "browsers": ["Android >= 2.1", "iOS >= 4", "ie >= 8", "firefox >= 15"],
4.     "cascade": true
5.   })
6. })
```

## postprocessor 插件

待补充

## optimizer 插件

### fis-optimizer-uglify-js

压缩 js 代码。

```
1. fis.match('*.{js,jsx,ts,tsx,es6,es}', {
2.   optimizer: fis.plugin('uglify-js')
3. });
```

### fis-optimizer-clean-css

压缩 css 代码。

```
1. fis.match('*.{scss,sass,less,css}', {
2.   optimizer: fis.plugin('clean-css',{
3.     //option
4.   })
5. })
```

### fis-optimizer-png-compressor

压缩 png 图片。

```
1. fis.match('*.png', {
2.   optimizer: fis.plugin('png-compressor',{
3.     //option
4.   })
5. })
```

## `fis-optimizer-smarty-xss`

`smarty xss` 修复插件。 `fis3-smarty` 中已默认配置好。

```
1. fis.match('*.tpl', {
2.   optimizer: fis.plugin('smarty-xss')
3. })
```

## `fis-optimizer-html-compress`

`smarty html` 代码压缩插件。 `fis3-smarty` 中已默认配置好。

```
1. fis.match('*.tpl', {
2.   optimizer: fis.plugin('html-compress')
3. })
```

## `jello-optimizer-velocity-xss`

`velocity` 模板 `xss` 修复插件。

```
1. fis.match('*.vm', {
2.   optimizer: fis.plugin('velocity-xss')
3. })
```

## package 插件

### `fis3-packager-map`

在 `fis3` 中自带了， 默认的打包插件。详情见插件 [Readme](#)

### `fis3-packager-deps-pack`

支持包含依赖的打包插件

```
1. fis.match('::packager', {
2.   packager: fis.plugin('deps-pack', {
3.
4.     'pkg/hello.js': [
5.
6.       // 将 main.js 加入队列
7.       '/static/hello/src/main.js',
```

```

8.
9.      // main.js 的所有同步依赖加入队列
10.     '/static/hello/src/main.js:deps',
11.
12.     // 将 main.js 所以异步依赖加入队列
13.     '/static/hello/src/main.js:asyns',
14.
15.     // 移除 comp.js 所有同步依赖
16.     '!/static/hello/src/comp.js:deps'
17.   ],
18.
19.   // 也可以从将 js 依赖中 css 命中。
20.   'pkg/hello.css': [
21.     // main.js 的所有同步依赖加入队列
22.     '/static/hello/src/main.js:deps',
23.   ]
24. })
25. });

```

## fis3-postpackager-loader

静态资源前端加载器，纯前端项目必备插件。自动加载页面中用到的资源，同时还能按页面级别的All In One 打包。

```

1.  fis.match('::packager', {
2.    postpackager: fis.plugin('loader')
3.  });

```

## deploy 插件

### fis3-deploy-local-deliver

已自带 fis3 中。用来将文件产出到本地。

```

1.  fis.match('*.js', {
2.    deploy: fis.plugin('local-deliver', {
3.      to: './output'
4.    })
5.  })

```

### fis3-deploy-http-push

将产出文件通过 http post 到目标机器。

```
1.  fis.match('*.js', {
2.      deploy: fis.plugin('http-push', {
3.          //如果配置了receiver, fis会把文件逐个post到接收端上
4.          receiver: 'http://www.example.com:8080/receiver.php',
5.          //这个参数会跟随post请求一起发送
6.          to: '/home/fis/www'
7.      })
8.  })
```

## fis3-deploy-tar

将产出文件，打包成 tar 文件。

```
1.  fis.match('**', {
2.      deploy: [
3.          fis.plugin('tar'),
4.
5.          fis.plugin('local-deliver', {
6.              to: './output'
7.          })
8.      ]
9.  })
```

## fis3-deploy-zip

将产出文件，打包成 zip 文件。

```
1.  fis.match('**', {
2.      deploy: [
3.          fis.plugin('zip'),
4.
5.          fis.plugin('local-deliver', {
6.              to: './output'
7.          })
8.      ]
9.  })
```

## fis3-deploy-encoding

将产出的文件做编码处理。



```

1.  fis.match('**', {
2.      charset: 'gbk',
3.      deploy: [
4.          fis.plugin('encoding'),
5.          fis.plugin('local-deliver')
6.      ]
7.  });

```

## fis3-deploy-replace

将产出的文件，做文本替换。

```

1.  fis.match('**', {
2.      deploy: [
3.          fis.plugin('replace', {
4.              from: /(img|cdn)\.baidu\.com/,
5.              to: function ($0, $1) {
6.                  switch ($1) {
7.                      case 'img':
8.                          return '127.0.0.1:8080';
9.                      case 'cdn':
10.                         return '127.0.0.1:8081';
11.                    }
12.                return $0;
13.            }
14.        }),
15.        fis.plugin('local-deliver')
16.    ]
17.  });

```

## fis3-deploy-skip-packed

将产出的文件过滤掉已被打包的。

```

1.  fis.match('**', {
2.      deploy: [
3.          fis.plugin('skip-packed', {
4.              // 配置项
5.          }),
6.
7.          fis.plugin('local-deliver', {
8.              to: 'output'

```

```
9.      })
10.    ]
11.  })
```

## hook 插件

### `fis3-hook-commonjs`

[强烈推荐] CommonJs 模块化支持插件。 详情请见 [README](#)

```
1.  fis.hook('commonjs')
```

### `fis3-hook-amd`

AMD 模块化支持插件。

### `fis3-hook-cmd`

CMD 模块化支持插件。

### `fis3-hook-system`

System 模块化支持插件。

### `fis3-hook-node_modules`

支持 npm 组件的插件，npm 包中的模块，直接通过包名就能 `require` 到。

```
1.  fis.hook('node_modules');
```

### `fis3-hook-relative`

支持产出为相对路径。

## **fis3 site**

# **fis3 site**

---

官网通过 FIS3 构建，本地查看编译查看。

## **安装依赖**

```
1. npm install
```

## **编译发布**

```
1. fis3 release
```

- 依赖 `fis3`

## **查看**

```
1. fis3 server start --type node
```

访问文档链接 <http://127.0.0.1:8080/>

# base-css

## base-css

---

完整版在 master 分支维护，首页优化精简在 mini 分支维护

```
git checkout mini
```

经过压缩精简，同样功能 base-css 从 **9K** 减少到 **5K**

## USAGE

---

为保持主流浏览器体积最小，拆分 IE6-8 为独立版本( `base.ie.css` )

```
1. <!-- for IE6-8 -->
2. <!--[if lt IE 9]>
3.     <link rel="stylesheet" href="base.ie.css">
4. <![endif]-->
5. <!--[if gte IE 9]><!-->
6.     <link rel="stylesheet" href="base.css">
7. <!--<![endif]-->
```

## BUILD & OUTPUT

---

- 安装依赖

```
npm i
```

- 编译打包

```
grunt
```

- 测试(默认访问: 127.0.0.1:8011)

```
grunt test
```

最终产出在 dist 目录，分为 4 份文件

- base.rtl.css
- base.ltr.css
- base.rtl.ie.css
- base.ltr.ie.css

# COMPACT DETAIL

## 1. 不常用的 form 控件 reset

```
1. input[type="search"] {
2.     -webkit-appearance: textfield;
3.     -webkit-box-sizing: content-box;
4.     -moz-box-sizing: content-box;
5.     box-sizing: content-box;
6. }
7.
8. input[type="search"]::-webkit-search-decoration, input[type="search"]::-webkit-
  search-cancel-button {
9.     -webkit-appearance: none;
10. }
```

## 2. html5 标签兼容

```
1. audio, canvas, video {
2.     display: inline-block;
3.     *display: inline;
4.     *zoom: 1;
5. }
6.
7. audio:not([controls]) {
8.     display: none;
9.     height: 0;
10. }
```

## 3. 过旧的样式属性兼容

```
1. .unselect, i, .i, .icon {
2.     -moz-user-select: -moz-none;
3.     -khtml-user-select: none;
4.     -webkit-user-select: none;
5.     -o-user-select: none;
6.     user-select: none;
7. }
```

## 4. IE Hack

```
1. _zoom:expression(function(e1) {
2.     document.execCommand('BackgroundImageCache',false,true);e1.style.zoom =
3.     "1";
4. })(this));
```

## 5. kill jQuery-UI

待定，依赖自定义网址重构

## 6. 不常用的工具类

```
1. sup, .sup {
2.     top: -0.5em;
3. }
4.
5. sub, .sub {
6.     bottom: -0.25em;
7. }
```

## 7. 冗余代码

```
1. @charset "utf-8";
2. ...
3. @charset "utf-8";
```

## 8. 抽象继承冗余部分

```
1. .icon-hot{
2.     display: inline-block;
3.     width: 30px;
4.     height: 11px;
5.     margin-left: 3px;
6.     cursor: pointer;
7.     background: url(../img/i-rtl-hot.png?m=z) no-repeat;
8.     _position: absolute;
9.     font-size:0;
10. }
11. .icon-new{
```

```
12.     display: inline-block;
13.     width: 30px;
14.     height: 11px;
15.     margin-left: 3px;
16.     cursor: pointer;
17.     background: url(../img/i-rtl-new.png?m=z) no-repeat;
18.     _position: absolute;
19.     font-size:0;
20. }
```

调整为：

```
1.  .icon-hot, .icon-new, .icon-new_red{
2.  }
```

另外：base.css 中提供了很多常用工具类，比如：

```
1.  .hide{} /*隐藏*/
2.  .s-ptn{} /*通用margin/padding设置*/
3.  .unselect{} /*禁止文本选择*/
4.  .fl{} /*浮动*/
5.  .g /*grid 布局相关*/
```

请在源码或文档中熟悉，避免业务代码冗余

## TEST

```
/test/index.html
```

## fis3/lib/fis.js

### fis3/lib/fis.js

`require`，若 `fis-` 与 `fis3-` 后插件名称重名则无法调用 `fis-` 组件

### fis3/lib/log.js

`str += '.' + ('00' + d.getMilliseconds()).substr(-4)` 应该改为 `str += '.' + ('00' + d.getMilliseconds()).substr(-3);`

```
1. exports.now = function(withoutMilliseconds) {
2.   var d = new Date(),
3.       str;
4.   str = [
5.     d.getHours(),
6.     d.getMinutes(),
7.     d.getSeconds()
8.   ].join(':').replace(/\b\d\b/g, '0$&');
9.   if (!withoutMilliseconds) {
10.    str += '.' + ('00' + d.getMilliseconds()).substr(-4);
11.   }
12.   return str;
13. };
```

### fis3/lib/config.js

`Config._get` 函数 `var result = this.data || {};`

```
1. _get: function(path) {
2.   var result = this.data || {};
3.   (path || '').split('.').forEach(function(key) {
4.     if (key && (typeof result !== 'undefined')) {
5.       result = result[key];
6.     }
7.   });
8.   return result;
9. }
```

### fis3/lib/util.js

`nohup` 函数中的回调未能表达执行的成功与否



`isEmpty` 对于 `Date` 对象、正则对象、`""` , `Number` , `undefined` 这几种类型未做处理, 均返回`true`

`pipe`

```
1. processors.forEach(function(obj, index) {
2.     var processor = obj.name || obj; // 这里用户传入函数有函数名会产生bug
3.     var key;
4.     ...
5. });
```

`download` 如果 `callback` 中使用 `return` 结束执行导致 `callback` 返回不为 `undefined` 会导致`tmp`文件删除失败

## fis3/lib/project.js

`getSource`

```
1. // 这里传入的rPath和root一样, 会导致include和exclude无法正常工作
2. fis.util.find(root, include, project_exclude, root).forEach(function(file)
3. {
4.     // 产生对应的 File对象
5.     file = fis.file(file);
6.     if (file.release) {
7.         source[file.subpath] = file;
8.     }
9. });
```

## fis3/lib/file.js

`defineLikes`

```
``JavaScript
set: (function(prop) {
return function(val) {
```

```
1. if (val === false) {
2.     this._likes[v] = false; // v应该为prop
3.     return;
4. }
5.
6. var that = this;
7. likes.forEach(function(v) {
8.     if (prop === v) {
9.         that._likes[v] = true
```

```
10.     } else {  
11.         that._likes[v] = false;  
12.     }  
13. });
```

```
}
```

```
})(v)
```

// 这类写法是否也可以使fis判断js.less在编译后为js.js的同名文件

// 实际实现不是

```
fis.match('**.less', {  
  rExt: 'css'  
});  
````
```