atk

atk 是 A tool kit的简写，其实也有写一系列工具库的想法，目前仅仅实现了一个简单的伪指令分析工具。

灵感来自gulp-tidt。

该项目是一个基于NodeJS编写的可扩展的html伪指令解析器，主要目的是让前端开发的代码更容易维护。比如：迁移及目录结构整理等。

支持使用npm,bower进行项目中外部库依赖管理，打包时，会从依赖库中挑选出项目真实依赖的文件。

快速上手

安装

```
npm install atk
```

使用

project/build.js

```
const atk = require("atk");
const File = require("vinyl");
const cwd = process.cwd();

let dr = new atk.DirectiveResolver({
    includePaths: {
        js: ["scripts", "assets/lib"],
        tpl: ["templates"]
    }
});

let result = dr.parse(new File({
    base: path.join(cwd, "src"),
    cwd: cwd,
    path: path.join(cwd, "src/pages/example.html"),
    contents: fs.readFileSync(path.join(cwd, "src/pages/example
.html"))
});

console.log(reusult);
```

project/src/pages/example.html

```
<!DOCTYPE html>
<html>
<head>
    <!--atk tpl="head"-->
    <title>Title</title>
</head>
<body>
<!--atk js="abc;ccd"-->
</body>
</html>
```

procject/src/templates/head.html

```
<meta charset="utf-8">
```

同时存在如下两个文件
project/src/assets/lib/abc.js
project/src/scripts/ccd.js

在project目录下执行命令：

```
node --harmony build.js
```

打印如下结果:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
  <title>Title</title>
</head>
<body>

<script src="../assets/lib/abc.js"></script>
<script src="../scripts/ccd.js"></script>
</body>
</html>
```

内置指令

特别说明:所有指令都支持使用';'分开引用多个值，且多个值是有顺序的。

指令名	作用
js	指定引用一个或者多个*js*文件
css	指定引用一个或者多个*css*文件
tpl	指定引用一个或者多个*模板*文件
njss	指定引用一个或者多个`npm`管理的*js*文件
ncss	指定引用一个或者多个`npm`管理的*css*文件
bjss	指定引用一个或者多个`bower`管理的*js*文件

bcss	指定引用一个或者多个`bower`管理的css文件
rule	指定引用一个或者多个*规则*

高级用法

规则定义

规则通过`atk.DirectiveResolver`构造函数的options传入，数据格式如下:

```
new atk.DirectiveResolver({
    rules: {
        rule1: [
            {   //该记录表示引用了一个别的规则-->rule2
                name: "rule",
                value: [
                    "rule2"
                ]
            },
            {  //该记录表示引用了一个js，与<!--atk js="a/b/c"-->作用
相同
                name: "js",
                value: [
                    "a/b/c",
                    ...
                ]
            }
        ],
        rule2: [
            {   //该记录表示引用了两个css，分别是reset.css和base.css
                name: "css",
                value: [
                    "reset",
                    "base"
                ]
            },
            {   //该条记录表示引用一个npm管理的bootstrap.css，具体路
径基于node_modules目录，bower同理
                name: "ncss",
                value: [
                    "bootstrap/dist/css/bootstrap"
                ]
            }
```

```
        ]
        ...
    }
})
```

可配置项

```
{
    //node的启动路径，执行查找npm或者bower作为依赖管理时，进行打包的查找根路径，一般这一层和npm和bower在同一层，如果有改动，需要外部配置
    cwd: process.cwd(),
    //写在页面中的指令名称，做成可配，以便各种不同项目做工具的时候，快速把工具变成项目本身所有，默认atk
    directiveName: "atk",
    //源码目录根目录,默认src
    srcBase: "src",
    //在打包bower管理的依赖库时，将被引用的文件拷贝到的目标目录,默认dist/npm
    npmDist: "dist/npm",
    //在打包bower管理的依赖库时，将被引用的文件拷贝到的目标目录,默认dist/bower
    bowerDist: "assets/bower",
    //配置查找不同指令文件的相对源的路径,默认无配置就是到当前制定的srcBase中去寻找
    includePaths: {
        js: ["."],
        css: ["."],
        tpl: ["."]
    },
    //路径变量，这里的变量值可以用于覆盖路径中的{variable}值，根据环境生成路径
    envSetting: {

    },
    //规则列表
    rules: {

    },
    //通过第三方依赖管理库管理的指令对应的库名称，默认如下，可通过此处配置支持别的依赖管理库支持
    thirdDepDirectiveNameMap: {
        njs: "npm",
        ncss: "npm",
        bjs: "bower",
        bcss: "bower"
```

```
    },
    //第三方依赖管理库名对应的根目录，默认支持npm和bower
    thirdDepBase: {
        npm: "node_modules",
        bower: "bower_components"
    },
    //第三方依赖管理库名对应的资源打包的目标目录(会生成vinly)根据该路径生成， 默认如下
    thirdDepDist: {
        //在打包npm管理的依赖库时，将被引用的文件拷贝到的目标目录,默认dist/npm
        npm: "assets/npm",
        //在打包bower管理的依赖库时，将被引用的文件拷贝到的目标目录,默认dist/bower
        bower: "assets/bower"
    },
    //每个指令对应的文件的扩展名，扩展名可以写空串，这样的话指令值默认需要加上扩展名
    directiveExtensions: {
        js: ".js",
        css: ".css",
        tpl: ".html",
        njs: ".js",
        ncss: ".css",
        bjs: ".js",
        bcss: ".css"
    },
    //指令解析器配置，支持扩展，可在这里添加自定义的指令的解析器，当然也可以覆盖掉默认的指令解析器
    parsers: {
        //基础的指令解析器
        js: require("../lib/JSParser"),
        css: require("../lib/CSSParser"),
        tpl: require("../lib/TplParser"),

        //第三方指令解析器(n开头代表npm，b开头的代表bower)
        njs: require("../lib/ThirdDepBasedParser").jsParser,
        ncss: require("../lib/ThirdDepBasedParser").cssParser,
        bjs: require("../lib/ThirdDepBasedParser").jsParser,
        bcss: require("../lib/ThirdDepBasedParser").cssParser,

        //规则指令解析器
        rule: require("../lib/RuleParser")
    }
}
```

自定义指令及指令名修改

通过如下面的方式即可实现指令及修改扩展：

```
new atk.DirectiveResolver({
    //修改指令解析器名称为xxx  也就是以后写指令时，atk就要改成xxx
    directiveName: "xxx"
    //定义自定义指令的解析器即可
    parsers: {
        /**
         * 自定义指令解析器
         * @param matchedCmd {String} 匹配到的指令全貌,如 <!--xxx
mydirective="value"-->
         * @param cmdName {String} 指令名称,示例中是mydirective
         * @param cmdValue {String} 指令值, 示例中是value
         * @param contents {String} 文件内容
         * @param index {Number} 指令所在内容中的索引
         * @returns {string} 解析之后的模板内容
         */
        mydirective: function(matchedCmd, cmdName, cmdValue, co
ntents, index){
            //这里写你的指令解析器内容，返回一个字符串，替换页面中你编写
的指令占位符

        }
    }
})
```

使用路径变量

示例：

```
new atk.DirectiveResolver({
    //路径变量，这里的变量值可以用于覆盖路径中的{variable}值，根据环境生
成路径
    envSetting: {
        //在指令中编写  <--atk js="configs/{env}.js"-->  执行的时候
会将env替换成stg
        env: "stg"
    }
})
```

当然在实际作为项目工具的一部分时，这个变量被构造时，会动态指定。比如使

用gulp，在gulpfile中，可能执行 gulp dev时候 和执行gulp stg时 这个env值是动态的，而非这样写死的

API

atk.DirectiveResolver

methods

parse(file)

根据当前配置，对文件中的指令进行解析

params:

`file` {File} vinly 实例

return: {String} 解析后生成的文件内容的字符串表示

getFiles()

获取当前上下文在指令执行后，新生成(比如：执行引用npm和bower管理的资源时，会将被引用的资源以及被引用的资源依赖的资源按照配置，构造成vinly实例，存放到当前上下文中)的所有文件的数组

params:

return: {Array<File>} 生成的所有文件

addFile(file)

添加一个文件到当前上下文

params:

`file` {File} vinly 实例

addFiles(files)

添加多个文件到当前上下文

params:

`files` {Array<File>} vinly 实例数组

injectPath(path)

给字符串注入当前上下文，envSetting中定义的变量

params:

`path` {String} 含有变量路径表达式的字符串

return: {String} 将路径变量表达式替换成实际路径变量中的值之后的内容