自定义Yeoman生成器

工作笔记　yeoman　generator

1 Getting Started

1.1、设置Node模块

Yeoman提供了generator-generator方便快速编写自己的生成器。

> 安装: npm install -g generator-generator
>
> 运行: yo generator

- 创建一个名为generator-name的文件夹（name为你的生成器名）;【important】
- 创建package.json文件，这是NodeJS模块的“信息图”，可以手动或者使用命令npm init生成

```
{
  "name": "generator-name",
  "version": "0.1.0",
  "description": "",
  "keywords": ["yeoman-generator"],
  "dependencies": {
    "yeoman-generator": "^0.17.3"
  }
}
```

> name属性必须要有generator-前缀；keywords属性必须包含yeoman-generator，务必确保是最新的，可运行命令npm install –save yeoman-generator完成更新/安装

1.2、文件树结构

1. 当调用yo name命令时，默认调用的是app生成器，对于的逻辑放置在app/文件夹下
2. 当调用yo name:subcommand命令时，必须要有对于的subcommand/文件夹

如果文件结构如下，则该生成器暴露yo name和yo name:router两个命令

```
├──────package.json
├──────app/
|     └──────index.js
└──────router/
      └──────index.js
```

如果你不想把所有代码都放在根目录下，Yeoman提供了另外的一种方式：可以放在generators/目录下

```
├──────package.json
└──────generators/
      ├──────app/
      |     └──────index.js
      └──────router/
            └──────index.js
```

1.3、继承generator

结构写好了，需要开始写实际的逻辑代码。Yeoman提供了基础生成器供你继承，这些基础生成器提供了很多方便的方法供你调用。基本写法：

```
var generators = require('yeoman-generator');
module.exports = generators.Base.extend();
```

如果你的生成器需要name参数（比如yo name:router foo中的foo），想将它赋给this.name的话：

```
var generators = require('yeoman-generator');
module.exports = generators.NamedBase.extend();
```

> 上面两种方式都能用于创建app生成器或者子生成器，Base多用于app生成器，NamedBase多用于需要指定文件名的子生成器

1.4、重写构造函数

有些方法只能在constructor方法中调用，常用于状态控制；可以传入构造函数重写默认的构造函数：

```
module.exports = generators.Base.extend({
  // The name `constructor` is important here
  constructor: function () {
    // Calling the super constructor is important so our generator
is correctly set up
    generators.Base.apply(this, arguments);

    // Next, add your custom code
    this.option('coffee'); // This method adds support for a `--cof
fee` flag
  }
});
```

1.5、添加方法

一般给原型添加的方法是按顺序执行的，不过后面我们会看到一些特殊的方法会触发不同的执行顺序：

```
module.exports = generators.Base.extend({
  method1: function () {
    console.log('method 1 just ran');
  },
  method2: function () {
    console.log('method 2 just ran');
  }
});
```

1.6、运行生成器

到了这一步，你已经拥有一个可以运行的生成器了。下一步就是检验生成器是否按自己的逻辑运行。由于是在本地开发生成器，在全局npm模块中并不存在，需要手动链接。进入generator-name/文件夹，运行：

```
npm link
```

这将自动安装工程依赖包，同时将本地文件链接进全局模块；运行完毕之后，你就可以调用yo name并看到之前定义的console.log信息。

至此，恭喜你完成了简单的生成器！

1.7、找到工程根目录

当运行一个生成器，Yeoman将计算当前的文件目录信息。最为关键的是，Yeoman将.yo-rc.json所在的目录作为工程的根目录，之后Yeoman将当前文件目录跳转到根目录下运行请求的生成器。这个.yo-rc.json文件是由Storage模块创建的，在生成器内部调用this.config.save()方法就会创建它。所以，如果你发现你的生成器不是在你当前工作目录下运行，请确保。yo-rc.json不存在你目录的其他层级中

2、运行上下文

2.1、静态方法都是Action

如果一个函数直接作为生成器的原型（prototype）的属性，则会当做action自动（按顺序）执行。如何声明不会自动执行的辅助函数以及私有函数呢？有三种方法：

1. 给方法前面添加前缀（例如：_method）
2. 使用实例函数声明（this.mehtod）

```
generators.Base.extend({
  init: function () {
    this.helperMethod = function () {
      console.log('won\'t be called automatically');
    };
  }
});
```

3. 继承自父类生成器

```
var MyBase = generators.Base.extend({
  helper: function () {
    console.log('won\'t be called automatically');
  }
});

module.exports = MyBase.extend({
  exec: function () {
    this.helper();
  }
});
```

2.2、运行顺序

Yeoman是按照优先级顺序依次执行所定义的方法。当你定义的函数名字是Yeoman定义的优先级函数名时，会自动将该函数列入到所在优先级队列中，否则就会列入到default优先层级队列中。

依次执行的方法名称为：

1. **initializing** - 你的初始化方法（检测当前目录状态，获取配置等）
2. **prompting** – 给用户展示选项提示（调用this.prompt()）
3. **configuring** – 保存用户配置项，同时配置工程（创建.editorconfig文件或者其他metadata文件）
4. **default**
5. **writing** – 用于生成和生成器相关的文件（比如routes,controllers等）
6. **conflicts** – 用于处理冲突异常（内部使用）
7. **install** – 用于安装相关库 (npm, bower)
8. **end** – 最后调用，常用于清理、道别等

3、UI

Yeoman默认是跑在终端的，但不限于终端。因此记住，不要使用console.log()或者process.stdout.write()向用户反馈信息，应当使用generator.log方法。

3.1、提示框

Yeoman中最为主要的UI交互就是提示框，由Inquirer.js组件提供。使用下列方式调用：

```
module.exports = generators.Base.extend({
  prompting: function () {
    var done = this.async();
    this.prompt({
      type    : 'input',
      name    : 'name',
      message : 'Your project name',
      default : this.appname // Default to current folder name
    }, function (answers) {
      this.log(answers.name);
      done();
    }.bind(this));
  }
})
```

> 这里我们使用promoting的优先层级 。由于咨询用户是一个异步的过程，会卡住命令逻辑的运行，所以需要调用yo的异步方法： `var cb = this.async();`

3.2、记住用户偏好

当用户运行你的生成器时，很多时候会输入相同的答案；Yeoman扩展了Inquirer.js的API，额外增加了store的属性表示用户可以将之前填写过的答案作为后续的默认答案：

```
this.prompt({
  type    : 'input',
  name    : 'username',
  message : 'What\'s your Github username',
  store   : true
}, callback);
```

> 提供默认答案时，程序会强制用户输入

3.3、命令行参数

可以直接像在命令中传入参数：

```
yo webapp my-project
```

在这里，my-project作为第一个参数。为了提示系统我们期望用户传入参数，需要调用generator.argument()方法，该方法接受name作为参数，以及额外的限制条件。

> 该argument方法必须在构造器中调用。这些条件是（key/value型）：

```
    'desc':  //Description for the argument
    'required':  // Boolean whether it is required
    'optional':  //Boolean whether it is optional
    'type':  // String, Number, Array, or Object
    'defaults':  //Default value for this argument
    'banner':  //String to show on usage notes (this one is provide
d by default)
```

示例代码：

```
module.exports = generators.Base.extend({
  // note: arguments and options should be defined in the construct
or.
  constructor: function () {
    generators.Base.apply(this, arguments);

    // This makes `appname` a required argument.
    this.argument('appname', { type: String, required: true });
    // And you can then access it later on this way; e.g. CamelCase
d
    this.appname = this._.camelize(this.appname);
  }
});
```

3.4、选项

选项看上去像参数，不过它前面多了两短横杠（flags）：

```
yo webapp --coffee
```

> 使用generator.option()方法获取选项值，该方法也有可选的限制属性（key/value型）：

```
'desc':  // Description for the option
'type' : // Either Boolean, String or Number
'defaults':  // Default value
'hide':  //Boolean whether to hide from help
```

举例：

```
module.exports = generators.Base.extend({
  // note: arguments and options should be defined in the construct
or.
  constructor: function () {
    generators.Base.apply(this, arguments);

    // This method adds support for a `--coffee` flag
    this.option('coffee');
    // And you can then access it later on this way; e.g.
    this.scriptSuffix = (this.options.coffee ? ".coffee": ".js");
  }
});
```

4、处理依赖

在运行生成器时，经常会伴随着npm和bower命令去安装依赖文件，Yeoman已经将这些功能抽离出来方便用户使用

4.1、npm

使用generator.npmInstall()运行npm安装命令，无论你调用多少次，Yeoman会确保该命令只执行一次

```
generators.Base.extend({
  installingLodash: function() {
    var done = this.async();
    this.npmInstall(['lodash'], { 'saveDev': true }, done);
  }
}):
```

上面的代码等价于命令行：

```
npm install lodash --save-dev
```

4.2、bower

使用generator.bowerInstall()运行bower安装命令，无论你调用多少次，Yeoman会确保该命令只执行一次

```
generators.Base.extend({
  end: function () {
    this.spawnCommand('composer', ['install']);
  }
});
```

> 记得在end队列中调用spawnCommand命令,否则用户没有耐心等那么久的。

5、文件系统

方便文件流的输入输出，Yeoman使用两种位置环境： **destination context** 和 **template context**

5.1、destination contex 目标位置上下文

destination context 目标位置上下文，这里的“目标”是指你想架构应用的位置。这个位置要么是当前文件夹，要么就是文件.yo-rc.json所在的父文件夹位置；

> 该.yo-rc.json文件确保所有的终端用户都以同样的方式方法生成器所在的子文件（夹）

使用 `generator.destinationRoot()` 获取目标位置上下文；也可以手动传参重新设置，当然没有人愿意那么做的；用 `generator.destinationPath('sub/path')` 拼接所需要的路径字符串。示例：

```
// Given destination root is ~/projects
generators.Base.extend({
  paths: function () {
    this.destinationRoot();
    // returns '~/projects'

    this.destinationPath('index.js');
    // returns '~/projects/index.js'
  }
});
```

5.2、template context 模板位置上下文

template context 模板位置上下文 ：就是你模板文件所在的文件夹位置，这个文件夹基本上是你读取并拷贝文件的地方。默认的template context是 `./templates/` ，你可以通过 `generator.sourceRoot('new/template/path')` 指定新的模板文件夹位置；与上面类似，可使用 `generator.sourceRoot()` 获取模板位置，使用 `generator.templatePath('app/index.js')` 拼接路径。示例：

```
generators.Base.extend({
  paths: function () {
    this.sourceRoot();
    // returns './templates'

    this.templatePath('index.js');
    // returns '~/templates/index.js'
  }
});
```

5.3、文件操作API

Yeoman把所有的文件方法都放在this.fs中了，它是mem-fs-editor的一个示例对象，可自行查看API接口。

示例：拷贝模板文件

假如。、templates/index.html文件内容为：

```
<html>
  <head>
    <title><%= title %></title>
  </head>
</html>
```

我们使用copyTpl方法拷贝模板：（更多参看Lodash template syntax）

```
generators.Base.extend({
  writing: function () {
    this.fs.copyTpl(
      this.templatePath('index.html'),
      this.destinationPath('public/index.html'),
      { title: 'Templating with Yeoman' }
    );
  }
});
```

一旦生成器运行完成，我们就会获得public/index.html

> Yeoman仍保留了旧的文件API，可参看 API documentation 。旧的文件API总是假设文件来自template context，写文件总是在destination context中，所以它们不要求你传入文件路径信息，程序会自动处理

建议：尽可能使用新的 `fs` API，它的使用起来比较清晰

6、储存用户设置

常常需要存储用户的设置项并在子生成器中使用，比如用户使用什么编程语言（比如使用CoffeeScript？）等这些配置项都存储在.yo-rc.json中（使用 Yeoman Storage API.）,可以通过 `generator.config` 对象获取API方法。

6.1、常用方法

1. `generator.config.save()`
   保存配置项到文件.yo-rc.json文件中（若文件不存在将自动 创建），由于该文件决定工程的根目录，因而一个最佳实践就是：就算什么也没有也应当调用save方法。

   > 每次设置配置项都会自动调用save方法，因此你可以不用显示调用

2. `generator.config.set(key,val)`

Name	描述
key	用于存储的键
val	任何JSON类型的值（String，Number, Array, Object）

3. `generator.config.get()`
   根据键获得配置项
4. `generator.config.getAll()`
   获取可用的所有配置信息；主要返回值不是按引用返回的，所以要更改里面的配置项还是需要调用set方法。
5. `generator.config.delete()`
   删除某个键值（及其值）
6. `generator.config.defaults()`
   将对象作为默认的配置信息，采用不覆盖原则

6.2、.yo-rc.json 文件结构

该文件可存储多个生成器的信息，每个生成器依据名字划分命名空间防止冲突，这也意味着每个生成器的配置项只能被子生成器读取到，不同生成器间的配置信息不能通过 Yeoman Storage API.访问。（使用命令行参数或者选项在不同构造器间传递参数。

文件样本：

```
{
  "generator-backbone": {
    "requirejs": true,
    "coffee": true
  },
  "generator-gruntfile": {
    "compass": false
  }
}
```

参考文档

1. WRITING YOUR OWN YEOMAN GENERATOR
2. 学习Bower – 前端开发包管理工具

    > 原文的2个链接失效了，不放上来了

整理文档

1. 自定义Yeoman生成器 —— JSCON-简时空