Grunt中文版

极客学院出版

前言

Grunt是一个自动化的项目构建工具。 如果你需要重复的执行像压缩， 编译， 单元测试， 代码检查以及打包发布的任务。 那么你可以使用Grunt来处理这些任务， 你所需要做的只是配置好Grunt， 这样能很大程度的简化你的工作。

本教程内容来源于 GruntJS中文 (http://www.gruntjs.org/)

英文官网Grunt：GruntJS官方 (http://www.gruntjs.com/)

更新日期	更新内容
2015-04-27	Grunt 中文版

如果在团队中使用Grunt, 你只需要与其他人员约定好使用Grunt应该规避的问题, 就能够很方便的自动化的处理大部分的常见工作任务, 你所付出的努力几乎为0.

为什么要使用Grunt?

Grunt是一个庞大的生态系统, 每天都在成长. 你可以自由的选择数以百计的插件以帮助你自动化的处理任务.

如果你所需要的插件还没有被人创建, 那么你可以自己创建插件并通过npm很方便的发布以供更多人使用并一起完善.

Grunt都有哪些插件?

大多数的任务Grunt都提供了可用的Grunt插件, 并且每天都有插件诞生并发布到社区中. 我想你所熟悉的有:

- JSHint
- Uglify
- Less
- Sass
- CoffeeScript
- CSSLint

等等. 更多的插件可以在Grunt官方的插件清单 (http://gruntjs.com/plugins) 中查看.

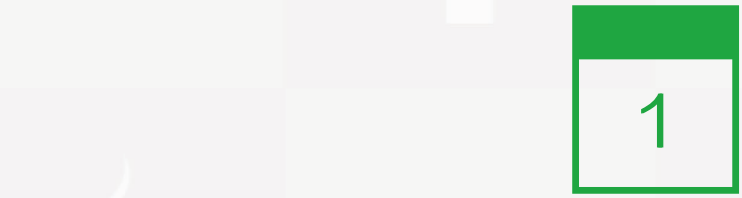
哪些人都在使用Grunt?

众所周知的有:

- jQuery
- Modernizr
- Twitter
- Adobe

等等. 还有更多的人在使用Grunt, 比如: 你.

目录



Grunt 教程

快速入门

Grunt和Grunt的插件都是通过Node.js (http://nodejs.org/) 的包管理器npm (https://npmjs.org/) 来安装和管理的。

Grunt 0.4.x要求Node.js的版本 >=0.8.0 (也就是0.8.0及以上版本的Node.js才能很好的运行Grunt)。

> 安装Grunt之前，可以在命令行中运行 `node -v` 查看你的Node.js版本。

安装CLI

如果你是从Grunt 0.3升级而来的，请查看 Grunt 0.3的说明 (http://gruntjs.com/getting-started#grunt-0.3-notes) 。(在这篇文档的底部)

为了方便使用Grunt，你应该在全局范围内安装Grunt的命令行接口(CLI)。要做到这一点，你可能需要使用sudo(OS X，*nix，BSD等平台中)权限或者作为超级管理员(Windows平台)来运行shell命令。

```
npm install -g grunt-cli
```

这条命令将会把 `grunt` 命令植入到你的系统路径中，这样就允许你从任意目录来运行它(定位到任意目录运行 `grunt` 命令)。

注意，安装 `grunt-cli` 并不等于安装了grunt任务运行器！Grunt CLI的工作很简单：在 `Gruntfile` 所在目录调用运行已经安装好的相应版本的Grunt。这就意味着可以在同一台机器上同时安装多个版本的Grunt。

CLI如何工作

每次运行 `grunt` 时，它都会使用node的 `require()` 系统查找本地已安装好的grunt。正因为如此，你可以从你项目的任意子目录运行 `grunt` 。

如果找到本地已经安装好的Grunt，CLI就会加载这个本地安装好的Grunt库，然后应用你项目中的 `Gruntfile` 中的配置(这个文件用于配置项目中使用的任务，Grunt也正是根据这个文件中的配置来处理相应的任务)，并执行你所指定的所有任务。

想要真正的了解这里发生了什么，可以阅读源码 (https://github。com/gruntjs/grunt-cli/blob/master/bin/grunt) 。这份代码很短。

用一个现有的Grunt项目进行工作

假设已经安装好Grunt CLI并且项目也已经使用一个 `package.json` 和一个 `Gruntfile` 文件配置好了，那么接下来用Grunt进行工作就非常容易了：

1. 进入到项目的根目录(在命令行面板定位到项目根目录。在windows系统下，也可以进入项目根目录的文件夹后，按Shift+鼠标右键，打开右键菜单，选择“在此处打开命令窗口(W)”)。
2. 运行 `npm install` 安装项目相关依赖(插件，Grunt内置任务等依赖)。
3. 使用 `grunt` (命令)运行Grunt。

就是这么简单。已经安装的Grunt任务可以通过运行 `grunt --help` 列出来，但是通常最好还是先查看一下项目的文档。

准备一个新的Grunt项目

一个典型的配置过程通常只涉及到两个文件： `package.json` 和 `Gruntfile` 。

package.json：这个文件被用来存储已经作为npm模块发布的项目元数据(也就是依赖模块)。你将在这个文件中列出你的项目所依赖的Grunt(通常我们在这里配置Grunt版本)和Grunt插件(相应版本的插件)。

Gruntfile：通常这个文件被命名为 `Gruntfile.js` 或者 `Gruntfile.coffee` ，它用于配置或者定义Grunt任务和加载Grunt插件。

package.json

`package.json` 与 `Gruntfile` 相邻，它们都应该归属于项目的根目录中，并且应该与项目的源代码一起被提交。在上述目录(`package.json` 所在目录)中运行 `npm install` 将依据 `package.json` 文件中所列出的每个依赖来自动安装适当版本的依赖。

这里有一些为项目创建 `package.json` 文件的方式：

- 大多数的grunt-init (http://gruntjs.com/project-scaffolding) 模板都会自动创建一个项目特定的 `package.json` 文件。
- npm init 命令会自动创建一个基本的 `package.json` 文件。
- 从下面的例子开始并根据规范来按需扩展。

```
{
  "name": "my-project-name", // 项目名称
  "version": "0.1.0", // 项目版本
  "devDependencies": { // 项目依赖
    "grunt": "~0.4.1", // Grunt库
    "grunt-contrib-jshint": "~0.6.0", //以下三个是Grunt内置任务
    "grunt-contrib-nodeunit": "~0.2.0",
    "grunt-contrib-uglify": "~0.2.2"
  }
}
```

> 原文中注释仅作说明，使用时请自行检查编辑。其他地方如有雷同，参考这条提示。

安装Grunt和grunt插件

添加Grunt和Grunt插件到一个现有的 package.json 中最简单的方式就是使用 npm install <module> --save-dev 命令。这不仅会在本地安装 <module> ，它还会使用一个波浪形字符的版本范围自动将所安装的 <module> 添加到项目依赖中。

例如使用下面的命令将会安装最新版的Grunt到你的项目中，并自动将它添加到你的项目依赖中：

```
npm install grunt --save-dev
```

上述命令也可以用于Grunt插件和其他的node模块的安装。当完成操作后请确保更新后的 package.json 文件也要与你的项目一起提交。

Gruntfile

Gruntfile.js 或者 Gruntfile.coffee 文件都是归属于你项目根目录中的一个有效的JavaScript或者CoffeeScript文件(和 package.json 文件一样都在根目录中)，并且它(Gruntfile)也应该与你的项目源文件一起提交。

一个Gruntfile由下面几部分组成：

- "wrapper"函数(包装函数)
- 项目和任务配置
- 加载的Grunt插件和任务
- 自定义任务

一个Gruntfile示例

在下面的Gruntfile中，项目的元数据会从项目的 package.json 文件中导入到grunt配置中，同时grunt-contrib-uglify (http://github.com/gruntjs/grunt-contrib-uglify) 插件的 uglify 任务被配置用于压缩一个源文件，同

时使用该元数据(导入的元数据)动态的生成一个标语(banner)注释。在命令行运行 `grunt` 时默认会运行 `uglify` 任务。

```
module.exports = function(grunt){

  // 项目配置
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    uglify: {
      options: {
        banner: '/*! <%= pkg.name %> <%= grunt.template.today("yyyy-mm-dd") %> */\n'
      },
      build: {
        src: 'src/<%=pkg.name %>.js',
        dest: 'build/<%= pkg.name %>.min.js'
      }
    }
  });

  // 加载提供"uglify"任务的插件
  grunt.loadNpmTasks('grunt-contrib-uglify');

  // 默认任务
  grunt.registerTask('default', ['uglify']);
}
```

现在你已经看到到了一个完整的Gruntfile，下面让我们来看看它的各个组成部分：

"wrapper"函数

每个Gruntfile(和Grunt插件)都使用这个基本格式，并且所有你的Grunt代码都必须指定在这个函数里面：

```
module.exports = function(grunt) {
  // 在这里处理Grunt相关的事情
}
```

项目和任务配置

大多数Grunt任务所依赖的配置数据都被定义在传递给grunt.initConfig (http://gruntjs.com/grunt#grunt.initconfig) 方法的一个对象中。

在这个例子中， `grunt.file.readJSON('package.json')` 会把存储在 `package.json` 中的JSON元数据导入到Grunt配置中。由于 `<% %>` 模板字符串可以引用任意的配置属性，因此可以通过这种方式来指定诸如文件路径和文件列表类型的配置数据，从而减少一些重复的工作(比如我们通常需要通过复制粘贴的方式来在不同的地方引用同一属性, 使用 `<% %>` 的方式可以简单的理解为将某些特定的数据存储在变量中，然后在其他地方像使用变量一样就可以使用这些数据属性)。

你可以在这个配置对象中(传递给initConfig()方法的对象)存储任意的数据，只要它不与你任务配置所需的属性冲突，否则会被忽略。此外，由于这本身就是JavaScript，你不仅限于使用JSON；你可以在这里使用任意的有效的JS代码。如果有必要，你甚至可以以编程的方式生成配置。

与大多数任务一样，grunt-contrib-uglify (http://github.com/gruntjs/grunt-contrib-uglify) 插件的 uglify 任务要求它的配置被指定在一个同名属性中。在这里有一个例子, 我们指定了一个 banner 选项(用于在文件顶部生成一个注释)，紧接着是一个单一的名为 build 的uglify目标，用于将一个js文件压缩为一个目标文件(比如将src目录 jquery-1.9.0.js 压缩为 jquery-1.9.0.min.js 然后存储到dest目录)。

```
// 项目配置
grunt.initConfig({
  pkg: grunt.file.readJSON('package.json'),
  uglify: {
    options: {
      banner: '/*! <%= pkg.name %> <%= grunt.template.today("yyyy-mm-dd") %> */\n'
    },
    build: {
      src: 'src/<%=pkg.name %>.js',
      dest: 'build/<%= pkg.name %>.min.js'
    }
  }
});
```

加载grunt插件和任务

许多常用的任务像concatenation (https://github.com/gruntjs/grunt-contrib-concat) ，minification (http://github.com/gruntjs/grunt-contrib-uglify) 和linting (https://github.com/gruntjs/grunt-contrib-jshint) 都被作为grunt插件 (https://github.com/gruntjs) 来使用。只要一个插件被作为一个依赖指定在项目的 package.json 文件中，并且已经通过 npm install 安装好，都可以在你的 Gruntfile 文件中使用下面这个简单的命令启用它(所依赖的任务)。

```
// 加载提供"uglify"任务的插件
grunt.loadNpmTasks('grunt-contrib-uglify');
```

注意: grunt --help 命令可以列出所有可用的任务。

自定义任务

你可以通过定义一个 default 任务来配置Grunt，让它默认运行一个或者多个任务。在下面的例子中，在命令行中运行 grunt 而不指定特定的任务将自动运行 uglify 任务。这个功能与显示的运行 grunt uglify 或者等价的 grunt default 一样。你可以在任务参数数组中指定任意数量的任务(这些任务可以带参数，也可以不带参数)。

```
// 默认任务
grunt.registerTask('default', ['uglify']);
```

如果你的项目所需的任务没有对应的Grunt插件提供相应的功能，你可以在 Gruntfile 内定义自定义的任务。例如，下面的Gruntfile就定义了一个完整的自定义的 default 任务，它甚至没有利用任务配置(没有使用grunt.initConfig()方法)：

```
module.exports = function(grunt) {
  // 一个非常基础的default任务
  grunt.registerTask('default', 'Log some stuff.', function() {
    grunt.log.write('Logging some stuff...').ok();
  });
};
```

自定义的项目特定的任务可以不定义在Gruntfile中；它们可以定义在一个外部 .js 文件中，然后通过grunt.loadTasks (http://gruntjs.com/grunt#grunt.loadtasks) 方法来加载。

扩展阅读

- 安装Grunt () 指南中有关于安装特定版本的，发布的或者开发中版本的Grunt和Grunt-cli的详细信息。
- 配置任务 () 指南中有对于如何在Gruntfile中配置任务，目标，选项和文件的详细解释，还有模板，匹配模式和导入外部数据相关的说明。
- 创建任务 () 指南列出了Grunt任务类型之间的不同，还展示了许多实例任务和配置。
- 对于关于编写自定义任务或者Grunt插件的更多信息，请参考开发者文档 () 。

Grunt 0.3说明

如果你从Grunt 0.3升级而来的，请确保先卸载全局的 grunt (使用下面的命令)：

```
npm uninstall -g grunt
```

上面这些说明文档是针对Grunt 0.4.x编写的，但仍然适用于Grunt 0.3.x。只是注意0.3.x版本中的插件名称和任务配置选项可能与上面的"Gruntfile"中所展示的不同。

对于0.3.x版本的Grunt, Grunfile名为 grunt.js 。

配置任务

这个指南解释了如何使用 Gruntfile 来为你的项目配置任务。如果你还不知道Gruntfile是什么，请先阅读新手上路 () 指南并参考Gruntfile示例 (http://gruntjs.com/sample-gruntfile/) 。

Grunt配置

Grunt的任务配置都是在你Gruntfile中的 grunt.initConfig 方法中指定。这个配置主要都是一些命名任务属性(通常任务都被定义为一个对象传递给 grunt.initConfig 方法, 而任务都是作为这个对象的属性定义的)，也可以包含任意其他数据。但这些属性(其他属性)不能与你的任务所需要的属性相冲突，否则它将被忽略(一般情况下任务中的属性命名都是约定俗成的)。

此外，由于这本身就是JavaScript，因此你不仅限于使用JSON；你可以在这里使用任何有效的JavaScript。必要的情况下，你甚至可以以编程的方式生成配置(比如通过其他的程序生成一个或多个任务配置)。

```
grunt.initConfig({
  concat: {
    //这里是concat任务的配置信息
  },
  uglify: {
    //这里是uglify任务的配置信息
  },
  //任意非任务特定属性
  my_property: 'whatever',
  my_src_file: ['foo/*.js', 'bar/*.js']
});
```

任务配置和目标

当运行一个任务时，Grunt会自动查找配置对象中的同名属性。多个任务可以有多个配置，每个任务可以使用任意的命名'targets'来定义多个任务目标。在下面的例子中， concat 任务有名为 foo 和 bar 两个目标，而 uglify 任务仅仅只有一个名为 bar 目标。

```
grunt.initConfig({
  concat: {
    foo: {
      // 这里是concat任务'foo'目标的选项和文件
    },
```

```
        bar: {
            // 这里是concat任务'bar'目标的选择和文件
        }
    },
    uglify: {
        bar: {
            // 这里是uglify任务'bar'目标的选项和文件
        }
    }
});
```

指定一个像 `grunt concat:foo` 或者 `grunt concat:bar` 的任务和目标只会处理指定的任务目标配置，而运行 `grunt concat` 将遍历所有的(定义在 `concat` 任务中的)目标并依次处理。注意，如果一个任务使用grunt.renameTask (https://github.com/gruntjs/grunt/wiki/grunt#wiki-grunt-renameTask) 重命名过，Grunt将在配置对象中查找新的任务名称属性。

options

在一个任务配置中， `options` 属性可以用来指定覆盖属性的内置默认值。此外，每一个任务目标中更具体的目标都可以拥有一个 `options` 属性。目标级的选项将会覆盖任务级的选项(就近原则———— `options` 离目标越近,其优先级越高)。

`options` 对象是可选，如果不需要，可以省略。

```
grunt.initConfig({
    concat: {
        options: {
            // 这里是任务级的Options，覆盖任务的默认值
        },
        foo: {
            options: {
                // 这里是'foo'目标的options，它会覆盖任务级的options.
            }
        },
        bar: {
            // 没有指定options，这个目标将使用任务级的options
        }
    }
});
```

文件

由于大多数的任务执行文件操作，Grunt有一个强大的抽象声明说明任务应该操作哪些文件。这里有好几种定义src-dest(源文件-目标文件)文件映射的方式，都提供了不同程度的描述和控制操作方式。任何一种多任务(包含多个任务目标的任务)都能理解下面的格式，所以你只需要选择满足你需求的格式就行。

所有的文件格式都支持 `src` 和 `dest` 属性，此外"Compact"[简洁]和"Files Array"[文件数组]格式还支持以下一些附加的属性：

- `filter` 它通过接受任意一个有效的fs.Stats方法名 (http://nodejs.org/docs/latest/api/fs.html#fs_class_fs_stats) 或者一个函数来匹配 `src` 文件路径并根据匹配结果返回 `true` 或者 `false` 。
- `nonull` 当一个匹配没有被检测到时，它返回一个包含模式自身的列表。否则，如果没有任何匹配项时它返回一个空列表。结合Grunt的 `--verbore` 标志, 这个选项可以帮助用来调试文件路径的问题。
- `dot` 它允许模式模式匹配句点开头的文件名，即使模式并不明确文件名开头部分是否有句点。
- `matchBase` 如果设置这个属性，缺少斜线的模式(意味着模式中不能使用斜线进行文件路径的匹配)将不会匹配包含在斜线中的文件名。 例如，a?b将匹配 `/xyz/123/acb` 但不匹配 `/xyz/acb/123` 。
- `expand` 处理动态的 `src-dest` 文件映射，更多的信息请查看"动态构建文件对象" (http://gruntjs.com/configuring-tasks#building-the-files-object-dynamically) 。
- 其他的属性将作为匹配项传递给底层的库。在node-glob (https://github.com/isaacs/node-glob) 和minimatch (https://github.com/isaacs/minimatch) 文档中可以查看更多选项。

简洁格式

这种形式允许每个目标对应一个src-dest文件映射。通常情况下它用于只读任务，比如grunt-contrib-jshint (https://github.com/gruntjs/grunt-contrib-jshint) , 它就值需要一个单一的 `src` 属性，而不需要关联的 `dest` 选项. 这种格式还支持为每个 `src-dest` 文件映射指定附加属性。

```
grunt.initConfig({
  jshint: {
    foo: {
      src: ['src/aa.js', 'src/aaa.js']
    }
  },
  concat: {
    bar: {
```

```
        src: ['src/bb.js', 'src/bbb.js'],
        dest: 'dest/b.js'
      }
    }
});
```

文件对象格式

这种形式支持每个任务目标对应多个 `src-dest` 形式的文件映射，属性名就是目标文件，源文件就是它的值(源文件列表则使用数组格式声明)。可以使用这种方式指定数个 `src-dest` 文件映射， 但是不能够给每个映射指定附加的属性。

```
grunt.initConfig({
  concat: {
    foo: {
      files: {
        'dest/a.js': ['src/aa.js', 'src/aaa.js'],
        'dest/a1.js': ['src/aa1.js', 'src/aaa1.js']
      }
    },
    bar: {
      files: {
        'dest/b.js': ['src/bb.js', 'src/bbb.js'],
        'dest/b1.js': ['src/bb1.js', 'src/bbb1.js']
      }
    }
  }
});
```

文件数组格式

这种形式支持每个任务目标对应多个 `src-dest` 文件映射，同时也允许每个映射拥有附加属性：

```
grunt.initConfig({
  concat: {
    foo: {
      files: [
        {src: ['src/aa.js', 'src/aaa.js'], dest: 'dest/a.js'},
        {src: ['src/aa1.js', 'src/aaa1.js'], dest: 'dest/a1.js'}
      ]
    },
    bar: {
      files: [
```

```
            {src: ['src/bb.js', 'src/bbb.js'], dest: 'dest/b/', nonull: true},
            {src: ['src/bb1.js', 'src/bbb1.js'], dest: 'dest/b1/', filter: 'isFile'}
        ]
    }
  }
});
```

较老的格式

dest-as-target文件格式在多任务和目标形式出现之前是一个过渡形式，目标文件路径实际上就是目标名称。遗憾的是, 由于目标名称是文件路径，那么运行 `grunt task:target` 可能不合适。此外，你也不能指定一个目标级的 `options` 或者给每个 `src-dest` 文件映射指定附加属性。

```
grunt.initConfig({
  concat: {
    'dest/a.js': ['src/aa.js', 'src/aaa.js'],
    'dest/b.js': ['src/bb.js', 'src/bbb.js']
  }
});
```

自定义过滤函数

`filter` 属性可以给你的目标文件提供一个更高级的详细帮助信息。只需要使用一个有效的fs.Stats方法名 (http://nodejs.org/docs/latest/api/fs.html#fs_class_fs_stats) 。下面的配置仅仅清理一个与模式匹配的真实的文件：

```
grunt.initConfig({
  clean: {
    foo: {
      src: ['temp/**/*'],
      filter: 'isFile'
    }
  }
});
```

或者创建你自己的 `filter` 函数根据文件是否匹配来返回 `true` 或者 `false` 。下面的例子将仅仅清理一个空目录：

```
grunt.initConfig({
  clean: {
    foo: {
      src: ['temp/**/*'],
      filter: function(filepath){
        return (grunt.file.isDir(filepath) && require('fs').readdirSync(filepath).length === 0);
      }
```

```
    }
  }
});
```

通配符模式

> 原文档标题为Globbing patterns，大意是指使用一些通配符形式的匹配模式快速的匹配文件。

通常分别指定所有源文件路径的是不切实际的(也就是将源文件-目标文件一一对应的关系列出来)，因此Grunt支持通过内置的node-glob (https://github.com/isaacs/node-glob) 和minimatch (https://github.com/isaacs/minimatch) 库来匹配文件名(又叫作 globbing)。

当然这并不是一个综合的匹配模式方面的教程，你只需要知道如何在文件路径匹配过程中使用它们即可：

- * 匹配任意数量的字符，但不匹配 /
- ? 匹配单个字符，但不匹配 /
- ** 匹配任意数量的字符，包括 / ，只要它是路径中唯一的一部分
- {} 允许使用一个逗号分割的列表或者表达式
- ! 在模式的开头用于否定一个匹配模式(即排除与模式匹配的信息)

大多数的人都知道 foo/*.js 将匹配位于 foo/ 目录下的所有的 .js 结尾的文件, 而 foo/**/*.js 将匹配 foo/ 目录以及其子目录中所有以 .js 结尾的文件。

此外, 为了简化原本复杂的通配符模式，Grunt允许指定一个数组形式的文件路径或者一个通配符模式。模式处理的过程中，带有 ! 前缀模式不包含结果集中与模式相配的文件。 而且其结果集也是唯一的。

示例：

```
//可以指定单个文件
{src: 'foo/this.js', dest: …}
//或者指定一个文件数组
{src: ['foo/this.js', 'foo/that.js', 'foo/the-other.js'], dest: …}
//或者使用一个匹配模式
{src: 'foo/th*.js', dest: …}

//一个独立的node-glob模式
{src: 'foo/{a,b}*.js', dest: …}
//也可以这样编写
{src: ['foo/a*.js', 'foo/b*.js'], dest: …}
```

```
//foo目录中所有的.js文件，按字母排序
{src: ['foo/*js'], dest: …}
//这里首先是bar.js，接着是剩下的.js文件按字母排序
{src: ['foo/bar.js', 'foo/*.js'], dest: …}

//除bar.js之外的所有的.js文件，按字母排序
{src: ['foo/*.js', '!foo/bar.js'], dest: …}
//所有.js文件按字母排序, 但是bar.js在最后.
{src: ['foo/*.js', '!foo/bar.js', 'foo/bar.js'], dest: …}

//模板也可以用于文件路径或者匹配模式中
{src: ['src/<%= basename %>.js'], dest: 'build/<%= basename %>.min.js'}
//它们也可以引用在配置中定义的其他文件列表
{src: ['foo/*.js', '<%= jshint.all.src %>'], dest: …}
```

可以在node-glob (https://github.com/isaacs/node-glob) 和minimatch (https://github.com/isaacs/minimatch) 文档中查看更多的关于通配符模式的语法。

构建动态文件对象

当你希望处理大量的单个文件时，这里有一些附加的属性可以用来动态的构建一个文件. 这些属性都可以指定在 Compact 和 Files Array 映射格式中(这两种格式都可以使用)。

- expand 设置 true 用于启用下面的选项:
- cwd 相对于当前路径所匹配的所有 src 路径(但不包括当前路径。)
- src 相对于 cwd 路径的匹配模式。
- dest 目标文件路径前缀。
- ext 使用这个属性值替换生成的 dest 路径中所有实际存在文件的扩展名(比如我们通常将压缩后的文件命名为 .min.js)。
- flatten 从生成的 dest 路径中移除所有的路径部分。
- rename 对每个匹配的 src 文件调用这个函数(在执行 ext 和 flatten 之后)。传递 dest 和匹配的 src 路径给它，这个函数应该返回一个新的 dest 值。 如果相同的 dest 返回不止一次，每个使用它的 src 来源都将被添加到一个数组中。

在下面的例子中， minify 任务将在 static_mappings 和 dynamic_mappings 两个目标中查看相同的 src-dest 文件映射列表, 这是因为任务运行时Grunt会自动展开 dynamic_mappings 文件对象为4个单独的静态 src-dest 文件映射--假设这4个文件能够找到。

可以指定任意结合的静态 src-dest 和动态的 src-dest 文件映射。

```
grunt.initConfig({
  minify: {
    static_mappings: {
      //由于这里的src-dest文件映射时手动指定的, 每一次新的文件添加或者删除文件时，Gruntfile都需要更新
      files: [
        {src: 'lib/a.js', dest: 'build/a.min.js'},
        {src: 'lib/b.js', dest: 'build/b.min.js'},
        {src: 'lib/subdir/c.js', dest: 'build/subdir/c.min.js'},
        {src: 'lib/subdir/d.js', dest: 'build/subdir/d.min.js'}
      ]
    },
    dynamic_mappings: {
      //当'minify'任务运行时Grunt将自动在"lib/"下搜索"**/*.js", 然后构建适当的src-dest文件映射，因此你不需要在文件添加或者
      files: [
        {
          expand: true, //启用动态扩展
          cwd: 'lib/', //批匹配相对lib目录的src来源
          src: '**/*.js', //实际的匹配模式
          dest: 'build/', //目标路径前缀
          ext: '.min.js' //目标文件路径中文件的扩展名.
        }
      ]
    }
  }
});
```

模板

使用 <% %> 分隔符指定的模板会在任务从配置中读取相应的数据时将自动填充。模板会以递归的方式填充，直到配置中不再存在遗留模板相关的信息(与模板匹配的)。

整个配置对象决定了属性上下文(模板中的属性)。此外，在模板中使用 grunt 以及它的方法都是有效的，例如：<%= grunt.template.today('yyyy-mm-dd') %> 。

- <%= prop.subprop %> 将会自动展开配置信息中的 prop.subprop 的值，不管是什么类型。像这样的模板不仅可以用来引用字符串值，还可以引用数组或者其他对象类型的值。
- <% %> 执行任意内联的JavaScript代码，对于控制流或者循环来说是非常有用的。

下面提供了一个 concat 任务配置示例，运行 grunt concat:sample 时将通过banner中的 /* abcde */ 连同 foo/*.js + bar/*.js + bar/*.js 匹配的所有文件来生成一个名为 build/abcde.js 的文件。

```
grunt.initConfig({
    concat: {
        sample: {
            options: {
                banner: '/* <%= baz %> */\n' // '/* abcde */\n'
            },
            src: ['<%= qux %>', 'baz/*.js'], // [['foo/*js', 'bar/*.js'], 'baz/*.js']
            dest: 'build/<%= baz %>.js'
        }
    },
    //用于任务配置模板的任意属性
    foo: 'c',
    bar: 'b<%= foo %>d', //'bcd'
    baz: 'a<%= bar %>e', //'abcde'
    qux: ['foo/*.js', 'bar/*.js']
});
```

导入外部数据

在下面的Gruntfile中，项目的元数据是从 package.json 文件中导入到Grunt配置中的，并且grunt-contrib-uglify插件 (http://github.com/gruntjs/grunt-contrib-uglify) 的 uglify 任务被配置用于压缩一个源文件以及使用该元数据动态的生成一个banner注释。

Grunt有 grunt.file.readJSON 和 grunt.file.readYAML 两个方法分别用于引入JSON和YAML数据。

```
grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    uglify: {
        options: {
            banner: '/* <%= pkg.name %> <%= grunt.template.today("yyyy-mm-dd") %> */\n'
        },
        dist: {
            src: 'src/<%= pkg.name %>.js',
            dest: 'dist/<%= pkg.name %>.min.js'
        }
    }
});
```

Gruntfile 实例

下面就针对一个 Gruntfile 案例做简单分析，也可以作为一个实例使用：

```
module.exports = function(grunt) {

  grunt.initConfig({
    jshint: {
      files: ['Gruntfile.js', 'src/**/*.js', 'test/**/*.js'],
      options: {
        globals: {
          jQuery: true
        }
      }
    },
    watch: {
      files: ['<%= jshint.files %>'],
      tasks: ['jshint']
    }
  });

  grunt.loadNpmTasks('grunt-contrib-jshint');
  grunt.loadNpmTasks('grunt-contrib-watch');

  grunt.registerTask('default', ['jshint']);

};
```

在页面底部是这个 Gruntfile 实例的完整内容，如果你按顺序阅读本文的话，可以跟随我们一步步分析这个文件中的每一部分。我们会用到以下5个 Grunt 插件：

第一部分是"wrapper" 函数，它包含了整个Grunt配置信息。

```
module.exports = function(grunt) {
}
```

在这个函数中，我们可以初始化 configuration 对象：

```
grunt.initConfig({
});
```

接下来可以从 package.json 文件读入项目配置信息，并存入 pkg 属性内。这样就可以让我们访问到 packag e.json 文件中列出的属性了，如下：

```
pkg: grunt.file.readJSON('package.json')
```

到目前为止我们就可以看到如下配置：

```
module.exports = function(grunt) {
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json')
  });
};
```

现在我们就可以为我们的每个任务来定义相应的配置(逐个定义我们为项目定义的任务配置)，然后每个任务的配置对象作为Grunt配置对象(即grunt.initConfig({})所接受的配置对象)的属性，并且这个属性名称与任务名相同。因此"concat"任务就是我们的配置对象中的"concat"键(属性)。下面便是我的"concat"任务的配置对象。

```
concat: {
  options: {

    separator: ';'
  },
  dist: {

    src: ['src/**/*.js'],

    dest: 'dist/<%= pkg.name %>.js'
  }
}
```

注意我是如何引用JSON文件(也就是我们在配置对象顶部引入的配置文件)中的 name 属性的。这里我使用 pkg.name 来访问我们刚才引入并存储在 pkg 属性中的 package.json 文件信息，它会被解析为一个JavaScript对象。Grunt自带的有一个简单的模板引擎用于输出配置对象(这里是指 package.json 中的配置对象)属性值，这里我让 concat 任务将所有存在于 src/ 目录下以 .js 结尾的文件合并起来，然后存储在 dist 目录中，并以项目名来命名。

现在我们来配置uglify插件，它的作用是压缩（ minify ）JavaScript文件：

```
uglify: {
  options: {

    banner: '/*! <%= pkg.name %> <%= grunt.template.today("dd-mm-yyyy") %> */n'
  },
  dist: {
    files: {
      'dist/<%= pkg.name %>.min.js': ['<%= concat.dist.dest %>']
    }
```

```
  }
}
```

这里我们让 uglify 在 dist/ 目录中创建了一个包含压缩结果的JavaScript文件。注意这里我使用了 <%= concat.dist.dest> ，因此uglify会自动压缩concat任务中生成的文件。

QUnit插件的设置非常简单。 你只需要给它提供用于测试运行的文件的位置，注意这里的QUnit是运行在HTML文件上的。

```
qunit: {
  files: ['test/**/*.html']
},
```

JSHint插件的配置也很简单：

```
jshint: {

  files: ['gruntfile.js', 'src/**/*.js', 'test/**/*.js'],

  options: {

    globals: {
      jQuery: true,
      console: true,
      module: true
    }
  }
}
```

JSHint只需要一个文件数组(也就是你需要检测的文件数组)， 然后是一个 options 对象(这个对象用于重写JSHint提供的默认检测规则)。你可以到[JSHint官方文档][1]站点中查看完整的文档。如果你乐于使用JSHint提供的默认配置，那么在Gruntfile中就不需要重新定义它们了.

最后，我们来看看watch插件：

```
watch: {
  files: ['<%= jshint.files %>'],
  tasks: ['jshint', 'qunit']
}
```

你可以在命令行使用 grunt watch 来运行这个任务。当它检测到任何你所指定的文件(在这里我使用了JSHint任务中需要检测的相同的文件)发生变化时，它就会按照你所指定的顺序执行指定的任务(在这里我指定了jshint和qunit任务)。

最后, 我们还要加载所需要的Grunt插件。 它们应该已经全部通过npm安装好了。

```
grunt.loadNpmTasks('grunt-contrib-uglify');
grunt.loadNpmTasks('grunt-contrib-jshint');
grunt.loadNpmTasks('grunt-contrib-qunit');
grunt.loadNpmTasks('grunt-contrib-watch');
grunt.loadNpmTasks('grunt-contrib-concat');
```

最后设置了一些task。最重要的是default任务：

```
grunt.registerTask('test', ['jshint', 'qunit']);

grunt.registerTask('default', ['jshint', 'qunit', 'concat', 'uglify']);
```

下面便是最终完成的 Gruntfile 文件：

```
module.exports = function(grunt) {

  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    concat: {
      options: {
        separator: ';'
      },
      dist: {
        src: ['src/**/*.js'],
        dest: 'dist/<%= pkg.name %>.js'
      }
    },
    uglify: {
      options: {
        banner: '/*! <%= pkg.name %> <%= grunt.template.today("dd-mm-yyyy") %> */n'
      },
      dist: {
        files: {
          'dist/<%= pkg.name %>.min.js': ['<%= concat.dist.dest %>']
        }
      }
    },
    qunit: {
      files: ['test/**/*.html']
    },
    jshint: {
      files: ['Gruntfile.js', 'src/**/*.js', 'test/**/*.js'],
      options: {

        globals: {
          jQuery: true,
```

```
        console: true,
        module: true,
        document: true
      }
    }
  },
  watch: {
    files: ['<%= jshint.files %>'],
    tasks: ['jshint', 'qunit']
  }
});

grunt.loadNpmTasks('grunt-contrib-uglify');
grunt.loadNpmTasks('grunt-contrib-jshint');
grunt.loadNpmTasks('grunt-contrib-qunit');
grunt.loadNpmTasks('grunt-contrib-watch');
grunt.loadNpmTasks('grunt-contrib-concat');

grunt.registerTask('test', ['jshint', 'qunit']);

grunt.registerTask('default', ['jshint', 'qunit', 'concat', 'uglify']);

};
```

创建任务

任务是Grunt的基本组成。例如你常用的工具： `jshint` 和 `nodeunit` 。当你每次运行Grunt时，你可以指定它(Grunt)运行一个或者多个任务，通过这些任务来告诉Grunt你想要它做什么事情。

如果你没有指定任务，但是定义了一个名为"default"的任务，那么该任务(default任务)将默认运行(不要惊讶，顾名思义它代表默认会运行哪些定义好的任务)。

别名任务

如果指定了一个任务列表，新任务(名)便是其他一个或多个任务的别名。每当运行'别名任务'时，指定在 `taskList` 中的每个任务(指在 `grunt.initConfig()` 中定义的任务)都会按照指定它们的顺序运行。 `taskList` 参数必须是一个任务数组。其语法如下：

```
grunt.registerTask(taskName, [description, ] taskList);
```

这里有一个例子，它定义了一个名为'default'别名任务，如果运行Grunt时没有指定任何任务，它将自动运行'jshint'，'qunit'，'concat'和'uglify'任务。

```
grunt.registerTask('default', ['jshint','qunit','concat','uglify']);
```

可以给任务指定参数。在下面的例子中，别名任务'dist'会运行'concat'和'uglify'这两个任务，并且每个任务都带有一个'dist'参数：

```
grunt.registerTask('dist', ['concat:dist', 'uglify:dist']);
```

多任务

当运行一个多任务时，Grunt会自动从项目的配置对象中查找同名属性。多任务可以有多个配置，并且可以使用任意命名的'targets'。

同时指定像 `grunt concat:foo` 或者 `grunt concat:bar` 这样的任务和目标，在运行时Grunt只会处理指定目标的配置；然而如果运行 `grunt concat` ，将会遍历所有的目标, 并按任务指定的顺序处理每个目标。注意，如果一个任务已经使用grunt.renameTask (https://github.com/gruntjs/grunt/wiki/grunt#wiki-grunt-renameTask) 重命名过，Grunt将会自动在配置对象中查找新任务名称属性。

大部分的contrib任务(主要是指官方提供的任务)，包括grunt-contrib-jshint插件的jshint任务 (https://github.com/gruntjs/grunt-contrib-jshint) ，以及grunt-contrib-concat插件的concat任务 (https://github.com/gruntjs/grunt-contrib-concat) 都是多任务形式的。

```
grunt.registerMultiTask(taskName, [description, ] taskFunction);
```

鉴于指定的配置，这里有一个示例演示了如果通过 `grunt log:foo` 运行Grunt，它会输出 `foo: 1,2,3` ；如果通过 `grunt log:bar` 来运行Grunt， 它会输出 `bar: hello world` 。然而如果通过 `grunt log` 运行Grunt, 它会输出 `foo: 1,2,3` ，然后是 `bar: hello world` ，最后是 `baz: false` (任务目标会按照指定的顺序进行处理)。

```
grunt.initConfig({
  log: {
    foo: [1,2,3],
    bar: 'hello world',
    baz: false
  }
});

grunt.registerTask('log','log stuff.', function(){
  grunt.log.writeln(this.target + ': ' + this.data);
});
```

'基本'任务

当运行一个基本任务时，Grunt并不会查找配置和检查运行环境--它仅仅运行指定的任务函数，可以传递任意使用冒号分割的参数作为任务函数的参数(注意多任务中的冒号并不是传递参数，而是指定具体任务的目标)。

```
grunt.registerTask(taskName, [description, ] taskFunction);
```

这里有一个例子演示了如果通过 `grunt foo:testing:123` 运行Grunt将输出 `foo, testing 123` 。如果运行这个任务时不传递参数，只运行 `grunt foo` ，那么这个任务会输出 `foo, no args` 。

```
grunt.registerTask('foo', 'A sample task that logs stuff.', function(arg1, arg2) {
  if (arguments.length === 0) {
    grunt.log.writeln(this.name + ", no args");
  } else {
    grunt.log.writeln(this.name + ", " + arg1 + " " + arg2);
  }
});
```

自定义任务

你可能会着迷于任务。但是如果你的任务并没有遵循多任务结构，那么你可以使用自定义任务。

```
grunt.registerTask('default', 'My "default" task description.', function(){
  grunt.log.writeln('Currently running the "default" task.');
});
```

在任务的内部，你还可以运行其他的任务。

```
grunt.registerTask('foo', 'My "foo" task.', function() {
  //在foo任务完成之后一次运行队列中的bar和baz任务
  grunt.task.run('bar', 'baz');
  // Or:
  grunt.task.run(['bar', 'baz']);
});
```

任务还可以是异步的.

```
grunt.registerTask('asyncfoo', 'My "asyncfoo" task.', function() {
  //将任务转变为异步模式并交给done函数处理
  var done = this.async();
  //同步任务
  grunt.log.writeln('Processing task...');
  //异步任务
  setTimeout(function() {
    grunt.log.writeln('All done!');
    done();
  }, 1000);
});
```

任务也可以访问它们自身名称和参数.

```
grunt.registerTask('foo', 'My "foo" task.', function(a, b) {
  grunt.log.writeln(this.name, a, b);
});

// 用法:
// grunt foo foo:bar
//   logs: "foo", undefined, undefined
//   logs: "foo", "bar", undefined
// grunt foo:bar:baz
//   logs: "foo", "bar", "baz"
```

如果任务记录到错误信息，还可以终止任务执行(通过标记任务失败的方式)。

```
grunt.registerTask('foo', 'My "foo" task.', function() {
  if (failureOfSomeKind) {
    grunt.log.error('This is an error message.');
  }

  //如果这个任务抛出错误则返回false
  if (ifErrors) { return false; }

  grunt.log.writeln('This is the success message');
});
```

当任务失败时，所有后续的除了指定 `--force` 标志的任务都会终止。

```
grunt.registerTask('foo', 'My "foo" task.', function() {
  // Fail synchronously.
  return false;
});

grunt.registerTask('bar', 'My "bar" task.', function() {
  var done = this.async();
  setTimeout(function() {
    // Fail asynchronously
    done(false);
  }, 1000);
});
```

任务还可以依赖于其他任务的成功执行。注意 `grunt.task.requires` 并不会运行其他任务(比如参数中指定的任务)。它仅仅检查那些任务(其他任务)的运行并没有失败(即其他任务，也就是所依赖的任务是否失败)。

```
grunt.registerTask('foo', 'My "foo" task.', function() {
  return false;
});

grunt.registerTask('bar', 'My "bar" task.', function() {
  //如果foo任务运行失败或者没有运行则任务失败
  grunt.task.requires('foo');
  //如果foo任务运行成功则执行这里的代码
  grunt.log.writeln('Hello, world.');
});

// 用法
// grunt foo bar
// 没有输出，因为foo失败
```

```
// grunt bar
// 没有输出，因为foo从未运行
```

如果任务需要的配置属性不存在，任务也可能失败。

```
grunt.registerTask('foo', 'My "foo" task', function(){
  //如果缺省"meta.name"配置属性则任务失败
  grunt.config.requires('meta.name');
  //与上一句相同，如果缺省"meta.name"配置属性则任务失败
  grunt.config.requires(['meta', 'name']);
  //附加记录
  grunt.log.writeln('This will only log if meta.name is defined in the config');
});
```

任务还可以访问配置属性。

```
grunt.registerTask('foo', 'My "foo" task.', function(){
  // 记录属性值，如果属性未定义则返回null
  grunt.log.writeln('The meta.name property is:' + grunt.config('meta.name'));
  // 同样的记录属性值，如果属性未定义则返回null
  grunt.log.writeln('The meta.name property is:' + grunt.config(['meta', 'name']));
});
```

在contrib tasks (https://github.com/gruntjs/) 中可以查看更多的例子。

CLI选项和环境

TODO(从FAQ拉取，推荐process.env)

为什么我的异步任务没有完成?

可能由于你忘记调用this.async (http://gruntjs.com/api/grunt.task#wiki-this-async) 方法来告诉Grunt你的任务是异步的，那么就可能会发生这种情况(异步任务失败)。为了简单起见，Grunt使用同步的编码风格，可以在任务体中通过调用 this.async 将该任务(调用这个方法的任务)转换为异步的。

注意传递 false 给 done 函数就会告诉Grunt任务已经失败。

例如:

```
grunt.registerTask('asyncme', 'My asynchronous task.', function(){
  var done = this.async();
  doSomethingAsync(done);
});
```

创建插件

1. 首先运行 `npm install -g grunt-init` 安装grunt-init (https://github.com/gruntjs/grunt-init) (这是一个Grunt插件构建模块)；
2. 运行 `git clone git://github.com/gruntjs/grunt-init-gruntplugin.git ~/.grunt-init/gruntplugin` 安装grunt插件模板；
3. 在一个空目录运行 `grunt-init gruntplugin` (这样就会将该目录初始化为Grunt插件构建目录，构建插件的文件最终存储在这个目录中， `grunt-init` 会自动生成相关配置文件)；
4. 运行 `npm install` 准备开发环境(自动安装插件构建所需的依赖)；
5. 编写你的插件(插件开发)；
6. 运行 `npm publish` 发布你的Grunt插件到npm(发布插件)；

注意

给你的任务命名

'grunt-contrib'命名空间是保留给Grunt团队[官方]维护的任务，请适当的给你的任务命名以避免命名冲突。

调试

默认情况下Grunt隐藏了错误堆栈跟踪信息，但是可以用 `--stack` 选项启用它以方便任务调试。 如果你希望Grunt始终都记录错误堆栈跟踪信息， 那么你就需要在你的shell中创建一个别名。 比如，在bash中你可以通过 `alias grunt="grunt ---stack"` 命令做到。

存储任务文件

只在项目根目录的.grunt/[npm-module-name]目录中存储数据，并且在适当的时候清除它。对于临时文件这并不是一个好的解决方案，在这种情况下可以使用常用的npm模块(如:temporary (https://npmjs.org/package/temporary) ， tmp (https://npmjs.org/package/tmp))来充分利用操作系统级的临时目录。

避免改变当前工作目录：process.cwd()

默认情况下，当前工作目录被设置为包含Gruntfile的目录。用户可以在它们的Gruntfile中使用 grunt.file.setBase 去改变它(改变当前工作目录)，但是插件应该当心点不要去改变它。

path.resolve('foo') 可以用于获取相对于 Gruntfile 所在目录的 foo 文件路径的绝对路径。

使用命令行工具

安装命令行工具

执行 `sudo npm install -g grunt-cli` 。

`grunt` 命令行接口提供了一系列选项。你可以在你的终端中使用 `grunt -h` 查看这个选项。

--help, -h

显示帮助信息。

--base, -b

指定一个基本路径。默认情况下，所有文件路径都是相对于 `Gruntfile` 的。

还可以调用 `grunt.file.setBase(...)`

--no-color

禁用彩色输出。

--gruntfile

指定 `Gruntfile` 文件。

默认情况下，grunt会从当前目录或者父目录中寻找最近的 `Gruntfile.js` 或者 `Gruntfile.coffee` 文件。

--debug, -d

对支持调试的任务启用调试模式。

--stack

因警告或者致命错误退出时打印堆栈跟踪信息。

--force, -f

一种强制跳过警告信息的方式。

如果像从警告中得到提示，就不要使用这个选项，可以根据提示信息修正代码。

--tasks

指定一个包含可加载的任务和"额外"文件的目录。

还可以调用 `grunt.loadTasks(...)`

--npm

在通过npm安装的插件中检查可加载的任何以及额外文件。

还可以调用 `grunt.loadNpmTasks(...)`

--no-write

禁用写文件操作（可用于演示）。

--verbose, -v

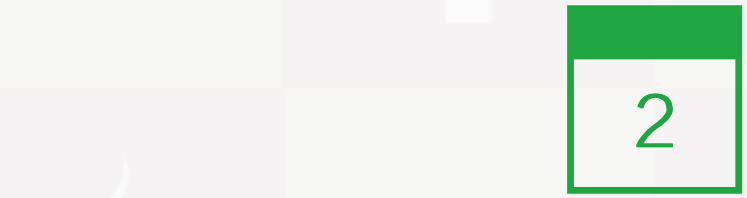
冗长模式（Verbose mode）。会输出很多的信息。

--version, -V

打印grunt版本。结合 --verbose 一起使用可以获取更多信息。

--completion

输出shell的自动补全规则。更多信息参考grunt-cli相关的文档。



2

进阶知识

API

安装Grunt

本文档说明了如何安装指定版本的Grunt和Grunt插件。

概览

Grunt和Grunt插件应该作为项目依赖定义在你项目的 package.json 中。这样就允许你使用一个单独的命令: `npm install` 安装你项目中的所有依赖(在 `package.json` 中定义好的grunt和grunt插件在使用 `npm install` 时会自动安装相关依赖，正如我们已经了解到的，这些依赖都定义在 `package.json` 中了)。当前稳定的和开发中的Grunt版本始终都列在wiki页面上 (https://github.com/gruntjs/grunt/wiki/) 。

安装指定版本的Grunt

如果你要安装指定版本的Grunt或者Grunt插件，只需要运行 `npm install grunt@VERSION --save-dev` 命令，其中 `VERSION` 就是你所需要的版本(指定版本号即可)。这样会安装指定版本的Grunt或者插件，并将它作为你的项目依赖添加到 `package.json` 。

注意，当你给 `npm install` 添加 `--save-dev` 标志时，一个波浪线范围将被用于你的 `package.json` 中。通常这么做是很好的，因为如果指定版本的Grunt或者插件有更新补丁出现时，它将自动升级到开发中的版本，与 semver 对应。

安装已发布的开发版本的Grunt

通常当新功能开发完成后，Grunt构建都可能会定期被发布到npm中。没有显式指定的版本号，这些构建**不可能**安装到依赖中，通常它会有一个内部版本号或者alpha/beta/指定候选版本号。

与安装指定版本的Grunt一样，运行 `npm install grunt@VERSION --save-dev` 其中 `VERSION` 就是你所需要的版本，同时npm将会安装那个版本(所指定版本的模块)的Grunt到你的项目目录中(通常都会安装到nodemodule中)，并把它添加到 `package.json` 依赖中。

注意，如果你没有注意指定版本号，都会有一个波浪线的版本范围将被指定到 `package.json` 中。**这是非常糟糕的**，因为指定开发版本的模块都是新的，可能是不兼容的，如果开发发布的补丁通过 `npm` 被安装到你的 `package.json` 中会有可能破坏你的构建工作。

在这种情况下手动的编辑你的 package.json 是**非常重要的**，并且你应该从 package.json 中的版本号中移除~(波浪线)。这样就会锁定你所指定的精确的开发版本(稳定并安装好的依赖模块)。

这种方式同样也可以用于安装已发布的开发版本的Grunt插件。

> 译注：通常发布的开发版都只是作为源代码提交到指定的仓库如Github等，可能并没有作为npm模块正式发布。在使用的过程中，建议按需添加稳定版本的依赖模块。如果你有足够的信心，也可以尝试使用最新的未正式发布的模块来满足工作需求。

从Github上直接安装

如果你想安装一个最新版的，未正式发布的Grunt或者Grunt插件，按照说明你可以指定一个 Git URL 作为依赖，注意这里一定要指定一个实际提交的SHA(而不是一个分支名)作为 commit-ish 。这样就会保证你的项目总是使用明确版本的Grunt。

指定的Git URL可能来自于Grunt官方或者分支。

> 也可以将github上托管的源码文件下载到本地来安装。

常见问题

如何安装Grunt?

对于一般安装说明，请阅读新手入门 (http://gruntjs.com/getting-started/) 指南。如果你在阅完读新手入门指南之后想要了解更多具体的信息，请阅读详细的安装Grunt (http://gruntjs.com/installing-grunt/) 指南。

我什么时候能够使用开发中的某个特性?

依据安装Grunt (http://gruntjs.com/installing-grunt/) 指南中的说明可以了解如何安装发布的和未发布的开发版本的Grunt(对于插件，你也可以使用npm安装并添加到依赖中使用；对于新的特性可以查阅相关说明安装使用)。

Grunt能在Windows上工作吗?

Grunt可以很好的工作在Windows平台，因为Node.js (http://nodejs.org/) 和npm (http://npmjs.org/) 在Windows上都工作得很好(Grunt是基于Node.js的，因而是能够跨平台工作，对于特定平台不支持的特性一般都会有相关说明或者有兼容性的选择来支持)。通常情况下问题在于Cygwin (http://www.cygwin.com/) ，因为它捆绑了一个过时版本的Node.js。

避免这个问题最好的方式是使用msysGit安装器 (http://msysgit.github.com/) 安装二进制的 `git` ，然后使用Node.js安装器 (http://nodejs.org/#download) 安装二进制的 `node` 和 `npm` ，并且使用内置的Window命令提示符 (http://www.cs.princeton.edu/courses/archive/spr05/cos126/cmd-prompt.html) 或者PowerShell (http://support.microsoft.com/kb/968929) 替代Cygwin(或者说我们首可以先安装好Git环境，然后将托管在Github上的Grunt库克隆到本地来安装使用Grunt，在工具方面Windows对Node.js的支持或许有欠缺，但是在对Node.js本身的支持上已经做得很好了)。

为什么我的异步任务不能完成?

如果发生这种情况可能是由于你忘记调用this.async (http://gruntjs.com/grunt.task#wiki-this-async) 方法来告诉Grunt，你的任务是异步的。为了简单起见，Grunt使用了同步编码的风格，可以通过在任务体内调用 `this.async()` 方法将该任务转换为异步任务。

注意，可以传递 `false` 给 `done()` 函数来告诉Grunt该任务失败。

例子:

```
grunt.registerTask('asyncme', 'My asynchronous task.', function() {
   var done = this.async();
   doSomethingAsync(done);
});
```

如何启用tab命令来自动补全?

添加下面的代码到你的 `~/.bashrc` 文件中来在Grunt中来启用bash tab自动补全功能:

```
eval "$(grunt --completion=bash)"
```

当然前提是你已经使用 `npm install -g grunt` 在全局范围内安装好了Grunt。当前，自动补全唯一支持的shell脚本是bash。

如何跨多任务共享参数?

虽然每个任务都可以接受它自己的参数，但是这里有几个选项可以在多个任务之间进行参数共享。

"动态"别名任务

这是在多任务间共享参数的首选方法。

别名任务 (http://gruntjs.com/grunt#wiki-grunt-registertask) 是非常简单的，一个常规的任务可以使用grunt.task.run (http://gruntjs.com/grunt.task#wiki-grunt-task-run) 方法让它编程一个有效的"动态的"别名任务函数。在下面的例子中，在命令行运行 `grunt build:001` 的结果就是执行 `foo:001` , `bar:001` 和 `baz:001` 这三个任务。

```
grunt.registerTask('build', 'Run all my build tasks.', function(n) {
  if (n == null) {
    grunt.warn('Build num must be specified, like build:001.');
  }
  grunt.task.run('foo:' + n, 'bar:' + n, 'baz:' + n);
});
```

-- options

另一种跨多任务共享参数的方式就是使用grunt.option (http://gruntjs.com/grunt#wiki-grunt-option) 。在下面的例子中，在命令行中运行 `grunt deploy --target=staging` 将导致 `grunt.option('target')` 返回 `"staging"` 。

```
grunt.registerTask('upload','Upload code to specified target.', function(n){
    var target = grunt.option('target');
    //在这里使用target做一些有用的事情
});
grunt.registerTask('deploy', ['validate', 'upload']);
```

注意布尔类型的选项(options)可以只指定使用一个没有值的键。例如，在命行中运行 `grunt deploy --staging` 将导致 `grunt.option('staging')` 返回 `true` .

全局和配置

在其他情况下，你可能希望暴露一个设置配置或者全局的值方法。 在这种情况下，可以在注册任务时设置其参数作为一个全局对象的或者项目配置的值。

在下面的例子中，在命令行运行 `grunt set_global:name:peter set_config:target:staging deploy` 会导致 `global.name` 的值为 `"peter"` 以及 `grunt.config('target')` 将会返回 `"staging"` 。由此推断， `deploy` 任务就可以使用这些值。

```
grunt.registerTask('set_global', 'Set a global variable.', function(name, val){
    global[name] = val;
});

grunt.registerTask('set_config', 'Set a config property.', function(name, val){
    grunt.config.set(name, val);
});
```

Grunt 0.3相关问题

在Windows的Grunt 0.3中, 为什么当我尝试运行grunt时会打开我的JS编辑器

如果你在Gruntfile (http://gruntjs.com/getting-started) 所在目录中(在命令行进入到这个目录)，当你输入grunt时，Windows中会尝试执行该文件(实际上Windows中会尝试执行grunt.js，因为在0.4.x之前的版本中Gruntfile.js名为grunt.js)。因此你应该输入 `grunt.cmd` 来替代。

另一种方式是使用 `DOSKEY` 命令创建一个Grunt宏, 可以参考也读这篇文章 (http://devblog.point2.com/2010/05/14/setup-persistent-aliases-macros-in-windows-command-prompt-cmd-exe-using-doskey/) 。这样就会允许你使用 `grunt` 来替代 `grunt.cmd` 。

下面是你可以使用的 `DOSKEY` 命令:

```
DOSKEY grunt=grunt.cmd $*
```

项目脚手架

grunt-init

`grunt-init` 是一个用于自动创建项目的脚手架工具。它会基于当前工作环境和几个问题的答案，构建一个完整的目录结构。但是这依赖于模板的选择，它会根据所提问题的答案，来创建更精确的文件和内容。

注意：这个独立的程序曾经是作为Grunt内置的 `init` 任务而存在的。在从0.3升级到0.4 (https://github.com/gruntjs/grunt/wiki/Upgrading-from-0.3-to-0.4) 指南中可以查看更多关于它演变的信息。

安装

为了能够使用grunt-init，你应该先在全局安装它：

> 对于常用的功能或者构建插件建议都在全局安装，从而可以避免在每个项目中都安装一次而占用更多的存储空间。当然在具体项目中安装，读取文件的速度会更快。

```
npm install -g grunt-init
```

这样就会把 `grunt-init` 命令植入到你的系统路径，从而允许你在任何目录中都可以运行它。

注意：你可能需要使用sudo权限或者作为超级管理员运行shell命令来执行这个操作。

用法

- 使用 `grunt-init --help` 来获取程序帮助以及列出可用模板清单；
- 使用 `grunt-init TEMPLATE` 基于可用模板创建一个项目；
- 使用 `grunt-init /path/to/TEMPLATE` 基于任意其他目录中可用的模板创建一个项目。

注意，大多数的模板都应该在当前目录(执行命令的目录)中生成它们的文件(自动生成的项目相关的文件)，因此，如果你不想覆盖现有的文件，注意一定要切换到一个新目录中来保证文件生成到其他目录。

安装模板

一旦模板被安装到你的 `~/.grunt-init/` 目录中(在Windows平台是 `%USERPROFILE%\.grunt-init\` 目录)，那么就可以通过 `grunt-init` 命令来使用它们了。建议你使用git将模板克隆到项目目录中。例如, grunt-init-jquery (https://github.com/gruntjs/grunt-init-jquery) 模板可以像下面这样安装：

```
git clone https://github.com/gruntjs/grunt-init-jquery.git ~/.grunt-init/jquery
```

注意：如果你希望在本地像"foobarbaz"这样使用模板，你应该指定 `~/.grunt-init/foobarbaz` 之后再克隆。 `grunt-init` 会使用实际在于 `~/.grunt-init/` 目录中的实际的目录名。

下面是一些有Grunt官方维护的grunt-init模板：

- grunt-init-commonjs (https://github.com/gruntjs/grunt-init-commonjs) – 创建一个包含Nodeunit单元测试的commonjs模块。(sample "generated" repo (https://github.com/gruntjs/grunt-init-commonjs-sample/tree/generated) | creation transcript (https://github.com/gruntjs/grunt-init-commonjs-sample#project-creation-transcript))
- grunt-init-gruntfile (https://github.com/gruntjs/grunt-init-gruntfile) – 创建一个基本的Gruntfile。(sample "generated" repo (https://github.com/gruntjs/grunt-init-gruntfile-sample/tree/generated) | creation transcript (https://github.com/gruntjs/grunt-init-gruntfile-sample#project-creation-transcript))
- grunt-init-gruntplugin (https://github.com/gruntjs/grunt-init-gruntplugin) – 创建一个包含Nodeunit单元测试的Grunt插件。(sample "generated" repo (https://github.com/gruntjs/grunt-init-gruntplugin-sample/tree/generated) | creation transcript (https://github.com/gruntjs/grunt-init-gruntplugin-sample#project-creation-transcript))
- grunt-init-jquery (https://github.com/gruntjs/grunt-init-jquery) – 创建一个包含QUnit单元测试的jQuery插件。(sample "generated" repo (https://github.com/gruntjs/grunt-init-jquery-sample/tree/generated) | creation transcript (https://github.com/gruntjs/grunt-init-jquery-sample#project-creation-transcript))
- grunt-init-node (https://github.com/gruntjs/grunt-init-node) – 创建一个包含Nodeunit单元测试的Node.js模块。(sample "generated" repo (https://github.com/gruntjs/grunt-init-node-sample/tree/generated) | creation transcript (https://github.com/gruntjs/grunt-init-node-sample#project-creation-transcript))

自定义模板

你可以创建和使用自定义模板。但是你的模板必须遵循与上述模板相同的文件结构。

下面是一个名为 `my-template` 的模板示例，它必须遵循下面这样的常规文件结构：

- `my-template/template.js` – 主模板文件；
- `my-template/rename.json` – 模板特定的重命名规则，用于处理模板；
- `my-template/root/` – 要复制到目标位置的文件。

假设这些文件存储在 `/path/to/my-template` 目录中，那么命令 `grunt-init /path/to/my-template` 就用于处理这些模板。这个目录中可能存在多个命名唯一的模板(多个不重名的模板)。

此外，如果你把这个自定义模板放在你的 `~/.grunt-init/` 目录中(在Windows上是 `%USERPROFILE%\.grunt-init\` 目录)，那么只需要使用 `grunt-init my-template` 命令它就会自动变得可用。

复制文件

当执行初始化模板时, 只要模板使用 `init.filesToCopy` 和 `init.copyAndProcess` 方法，任何位于 `root/` 子目录中的文件都将被复制到当前目录。

注意所有被复制的文件都会被处理为模板，并且所有暴露在 `props` 数据对象集合中的 `{% %}` 模板都会被处理，除非设置了 `noProcess` 选项。可以看看jquery template (https://github.com/gruntjs/grunt-init-jquery) 的例子。

重命名或者排除模板文件

`rename.json` 用于描述 `sourcepath` 到 `destpath` 的重命名映射关系。 `sourcepath` 必须是相对于 `root/` 目录要被复制的文件路径，但是 `destpath` 值可以包含 `{% %}` 模板，用于描述目标路径是什么。

如果 `destpath` 被指定为 `false` ，那么文件就不会被复制。此外， `srcpath` 还支持通配符匹配模式。

指定默认提示信息

每个初始化提示都会有一个硬编码的默认值或者它会根据当前环境来尝试确定该缺省值。如果你想覆盖某个特定提示信息的默认值，你可以在OS X或者Linux的 `~/.grunt-init/defaults.json` 或者Windows的 `%USERPROFILE%\.grunt-init\defaults.json` 文件中选择性的进行处理。

例如，由于我希望使用一个与众不同的名字来替代默认的名字，并且我还希望排除我的邮箱地址，同时我还希望自动指定一个作者的url，那么我的 `defaults.json` 看起来就可能像下面这样。

```
{
  "author_name": "\"Cowboy\" Ben Alman",
  "author_email": "none",
  "author_url": "http://benalman.com/"
}
```

注意: 即使记录了所有的内置提示信息都被重新定义了，你仍然可以在源代码 (https://github.com/gruntjs/grunt-init/blob/master/tasks/init.js) 中找到他们的名字和默认值。

定义一个初始化模板

exports.description

当用户运行 `grunt init` 或者 `grunt-init` 来显示所有的有效初始化模板时，这个简短的模板描述也会和模板名一起显示。

```
exports.description = descriptionString;
```

exports.notes

如果指定了这个选项，这个可选的扩展描述将会在任何提示信息显示之前显示出来。这是一个给用户提供一些解释命名空间相关帮助信息的很好的地方。这些提示可能是必选的也可能是可选的，等等。

```
exports.notes = noteString;
```

exports.warnOn

如果这个(推荐指定)可选的通模式或者通配模式数组匹配了，Grunt将终止并生成一个警告信息，用户可以使用 `--force` 来覆盖这个默认行为。这对于初始化模板可能覆盖现有文件的情况来说是非常有用的。

```
exports.warnOn = wildcardPattern;
```

然而最常见的值是 '*'，它能够匹配任意文件或者目录。使用minimatch (https://github.com/isaacs/minimatch) 通配符模式具有很大的灵活性。例如：

```
exports.warnOn = 'Gruntfile.js'; // 警告Gruntfile.js文件.
exports.warnOn = '*.js'; //警告任意.js文件
exports.warnOn = '*'; //警告任意非点文件或目录
exports.warnOn = '.*'; //警告任意点文件或目录
exports.warnOn = '{.*, *}'; //警告任意文件或目录
exports.warnOn = '!*/**'; //警告任意目录中的文件
exports.warnOn = '*.{png,gif,jpg}'; //警告任意图片文件
//最后一个例子的另一种形式
exports.warnOn = ['*.png','*.gif','*.jpg'];
```

exports.template

虽然 exports 属性定义在该函数的外面，然而所有实际的初始化代码指定在它内部。这个函数接受三个参数，grunt 参数是一个grunt的引用，它包含所有的grunt方法和库 (http://gruntjs.com/api/grunt) 。init 参数是一个包含特定于这个初始化模板而存在的方法和属性的对象。done 参数是在初始化模板执行完成时必须调用的函数。

```
exports.template = function(grunt, init, done){
  //查看"Inside an init template"一节
};
```

初始化模板的内部

init.addLicenseFiles

可以给files对象添加适当命名的许可协议证书文件。

```
var files = {};
var licenses = ['MIT'];
init.addLicenseFiles(files, licenses);
// files === {'LICENSE-MIT': 'licenses/LICENSE-MIT'}
```

init.availableLicenses

返回一个可用许可协议证书的数组：

```
var licenses = init.availableLicenses();
// licenses = ['Apache-2.0', 'GPL-2.0', 'MIT', 'MPL-2.0']
```

init.copy

它提供一份绝对或者相对源文件路径，以及一个可选的相对的目标文件路径，复制一个文件时，可以通过传递的回调函数来选择性的处理它。

```
init.copy(srcpath[, destpath], options);
```

init.copyAndProcess

遍历所传递对象中的所有文件，将源文件复制到目标路径，并处理相关内容。

```
init.copyAndProcess(files, props[, options]);
```

init.defaults

用户在 `defaults.json` 中指定的默认初始值。

```
init.defaults
```

init.destpath

目标文件绝对路径。

```
init.destpath();
```

init.expand

与grunt.file.expand (https://github.com/gruntjs/grunt/wiki/grunt.file#wiki-grunt-file-expand) 相同。

返回一个独一无二的与给定通配符模式所匹配的所有文件或目录路径数组。这个方法接收一个逗号分割的通配符模式或者数组形式的通配符模式参数。如果路径匹配模式以 `!` 开头，与模式所匹配的结果就会从返回的数组中排除。模式是按顺序处理的，所以包含和排除的顺序是很明显的。

```
init.expand([options, ] patterns);
```

init.filesToCopy

返回一个包含待复制文件源文件的绝对路径和相对的目标路径的对象，并按照 rename.json (如果存在)中的规则重命名(或者忽略)。

```
var files = init.filesToCopy(props);
/* files === { '.gitignore': 'template/root/.gitignore,
  'jshintrc': 'template/root/.jshintrc',
  'Gruntfile.js': 'template/root/Gruntfile.js',
  'README.md': 'template/root/README.md',
  'test/test_test.js': 'template/root/test/name_test.js'
  } */
```

init.getFile

获取单一的任务文件路径。

```
init.getFile(filepath[, ...]);
```

init.getTemplates

返回一个包含所有可用模板的对象。

```
init.getTemplates()
```

init.initSearchDirs

在初始化目录中搜索初始化模板。 template 是指模板的位置。还包括 ~/.grunt-init 和grunt-init中的核心初始化任务。

```
init.initSearchDirs([filename]);
```

init.process

启动程序并提示开始输入。

```
init.process(options, prompts, done);

init.process({}, [
```

```
    //Prompt for these values
    init.prompt('name'),
    init.prompt('description'),
    init.prompt('version')
], function(err, props){
    // All finished, do something with the properties
});
```

init.prompt

给用户一个提示值。

```
init.prompt(name[, default])
```

init.prompts

包含所有提示信息对象。

```
var prompts = init.prompts();
```

init.readDefaults

读取任务文件中JSON默认值(如果存在)，并将它们合并到一个数据对象中。

```
init.readDefaults(filepath[, ...])
```

init.renames

模板的重命名规则。

```
var renames = init.renames;
//renames = {'test/name_test.js': 'test/{%= name%}_test.js' }
```

init.searchDirs

搜索模板的目录数组。

```
var dirs = init.searchDirs;
/* dirs = ['/Users/shama/.grunt-init',
'/usr/local/lib/node_modules/grunt-init/templates' ] */
```

init.srcpath

根据文件名搜索初始化模板路径并返回一个绝对路径。

```
init.srcpath(filepath[, ...]);
```

init.userDir

返回用户模板目录的绝对路径。

```
var dir = init.userDir();
//dir === '/Users/shama/.grunt-init'
```

init.writePackageJSON

在目标目录中保存一个 `package.json` 文件。回调函数可以用于后置处理属性的添加/移除/其他操作。

```
init.writePackageJSON(filename, props[, callback]);
```

内置提示

author_email

用于 `package.json` 中的作者邮箱地址。默认情况下会尝试从用户的git配置中找到一个默认值。

author_name

用于 `package.json` 中的作者全名和版权信息。也会尝试从用户的git配置中找到一个默认值。

author_url

`package.json` 中的用于公开作者个人网站(博客)的URL。

bin

项目根目录中cli脚本的相对路径。

bugs

用于项目问题跟踪的公开URL。如果项目有一个Github仓库，将自动指向项目Github的问题跟踪模块(issue)。

description

项目的描述。通常在 package.json 或者README文件中。

grunt_version

项目所需的有效Grunt版本范围描述符。

homepage

指向项目首页的公开URL。如果是Github仓库将默认指向Github上的url。

jquery_version

如果是jQuery项目，它表示项目所需的jQuery版本。必须是一个有效的版本范围描述符。

licenses

项目许可协议证书。多个许可协议证书使用空格分割，内置的许可协议有： MIT ， MPL-2.0 ， GPL-2.0 和 Apache-2.0 。可以使用init.addLicenseFiles (http://gruntjs.com/#initaddlicensefiles) 方法添加自定义许可协议证书。

main

项目主入口点，默认情况下是 lib 目录中的项目名称(通常是项目名称目录)。

name

项目名称。在项目模版中将会大量使用，默认指向当前工作目录。

node_version

项目所需的Node.js版本。必须是一个有效的版本范围描述符。

npm_test

项目中运行测试的命令，默认情况下是 `grunt` 。

repository

项目的git仓库，默认猜测为一个github上的url。

title

一个用户可读的项目名称，默认是修改过的让更多人可读的实际项目名称。

version

项目的版本，默认指向第一个有效的语义版本， `0.1.0` 。