目錄

介紹 0
基础环境 1
Node安装 1.1
GitBook安装 1.2
[GitBook配置] 2
package.json配置 2.1
GitBook基本配置 2.2

学习gitbook

1. 安装 node.js 不必多说
2. 安装 Gitbook

```
npm install gitbook -g
npm install gitbook-cli -g
```

3. gitbook文件

```
# book.json //gitbook配置
# README.md//介绍
# SUMMARY.md //目录
```

前言

介绍前端模块系统的演进历史，以及 Webpack 出现的背景及其特点。

Node.js

> Node.js 是一个基于Chrome JavaScript 运行时建立的一个平台， 用来方便地搭建快速的， 易于扩展的网络应用 · Node.js 借助事件驱动， 非阻塞 I/O 模型变得轻量和高效， 非常适合 run across distributed devices 的 data-intensive 的实时应用。

1.win下安装Node.js

> 直接到nodejs官网下载，简单安装即可,此处不再细述。

GitBook

> GitBook 是一个基于 Node.js 的命令行工具，可使用 Github/Git 和 Markdown 来制作精美的电子书。通过Node.js命令安装GitBook

1、NMP安装Gitbook

```
npm install gitbook -g
```

2、安装gitbook CLI

> 想在系统上的任何地方的gitbook命令，需要安装“gitbook CLI”，执行以下命令

```
//安装命令
npm install -g gitbook-cli
//卸载命令
npm uninstall -g gitbook
```

使用

首先创建一个静态页面 index.html 和一个 JS 入口文件 entry.js：

```
<!-- index.html -->
<html>
<head>
  <meta charset="utf-8">
</head>
<body>
  <script src="bundle.js"></script>
</body>
</html>
```

```
// entry.js
document.write('It works.')
```

然后编译 entry.js 并打包到 bundle.js：

```
$ webpack entry.js bundle.js
```

打包过程会显示日志：

```
Hash: e964f90ec65eb2c29bb9
Version: webpack 1.12.2
Time: 54ms
    Asset     Size  Chunks             Chunk Names
bundle.js  1.42 kB       0  [emitted]  main
   [0] ./entry.js 27 bytes {0} [built]
```

用浏览器打开 `index.html` 将会看到 `It works.` 。

接下来添加一个模块 `module.js` 并修改入口 `entry.js` ：

```
// module.js
module.exports = 'It works from module.js.'
```

```
// entry.js
document.write('It works.')
document.write(require('./module.js')) // 添加模块
```

重新打包 `webpack entry.js bundle.js` 后刷新页面看到变化 `It works.It works from module.js.`

```
Hash: 279c7601d5d08396e751
Version: webpack 1.12.2
Time: 63ms
    Asset     Size  Chunks             Chunk Names
bundle.js  1.57 kB       0  [emitted]  main
   [0] ./entry.js 66 bytes {0} [built]
   [1] ./module.js 43 bytes {0} [built]
```

Webpack 会分析入口文件，解析包含依赖关系的各个文件。这些文件（模块）都打包到 bundle.js 。Webpack 会给每个模块分配一个唯一的 id 并通过这个 id 索引和访问模块。在页面启动时，会先执行 entry.js 中的代码，其它模块会在运行 `require` 的时候再执行。

Loader

Webpack 本身只能处理 JavaScript 模块，如果要处理其他类型的文件，就需要使用 loader 进行转换。

Loader 可以理解为是模块和资源的转换器，它本身是一个函数，接受源文件作为参数，返回转换的结果。这样，我们就可以通过 `require` 来加载任何类型的模块或文件，比如 CoffeeScript、 JSX、 LESS 或图片。

先来看看 loader 有哪些特性？

- Loader 可以通过管道方式链式调用，每个 loader 可以把资源转换成任意格式并传递给下一个 loader ，但是最后一个 loader 必须返回 JavaScript。
- Loader 可以同步或异步执行。
- Loader 运行在 node.js 环境中，所以可以做任何可能的事情。
- Loader 可以接受参数，以此来传递配置项给 loader。
- Loader 可以通过文件扩展名（或正则表达式）绑定给不同类型的文件。
- Loader 可以通过 `npm` 发布和安装。
- 除了通过 `package.json` 的 `main` 指定，通常的模块也可以导出一个 loader 来使用。
- Loader 可以访问配置。
- 插件可以让 loader 拥有更多特性。
- Loader 可以分发出附加的任意文件。

Loader 本身也是运行在 node.js 环境中的 JavaScript 模块，它通常会返回一个函数。大多数情况下，我们通过 npm 来管理 loader，但是你也可以在项目中自己写 loader 模块。

按照惯例，而非必须，loader 一般以 `xxx-loader` 的方式命名， `xxx` 代表了这个 loader 要做的转换功能，比如 `json-loader` 。

在引用 loader 的时候可以使用全名 `json-loader` ，或者使用短名 `json` 。这个命名规则和搜索优先级顺序在 webpack 的 `resolveLoader.moduleTemplates` api 中定义。

```
Default: ["*-webpack-loader", "*-web-loader", "*-loader", "*"]
```

Loader 可以在 `require()` 引用模块的时候添加，也可以在 webpack 全局配置中进行绑定，还可以通过命令行的方式使用。

接上一节的例子，我们要在页面中引入一个 CSS 文件 style.css，首页将 style.css 也看成是一个模块，然后用 `css-loader` 来读取它，再用 `style-loader` 把它插入到页面中。

```
/* style.css */
body { background: yellow; }
```

修改 entry.js：

```
require("!style!css!./style.css") // 载入 style.css
document.write('It works.')
document.write(require('./module.js'))
```

安装 loader：

```
npm install css-loader style-loader
```

重新编译打包，刷新页面，就可以看到黄色的页面背景了。

如果每次 `require` CSS 文件的时候都要写 loader 前缀，是一件很繁琐的事情。我们可以根据模块类型（扩展名）来自动绑定需要的 loader。

将 entry.js 中的 `require("!style!css!./style.css")` 修改为 `require("./style.css")` ，然后执行：

```
$ webpack entry.js bundle.js --module-bind 'css=style!css'

# 有些环境下可能需要使用双引号
$ webpack entry.js bundle.js --module-bind "css=style!css"
```

显然，这两种使用 loader 的方式，效果是一样的。