使用 Angular2+ 开发 Markdown 编辑器

前言

一直想写一个 Angular2+ 的分享，但是没有一个好的切入点。前段时间我向分享 Chat 的白宦成老师请教 Markdown 的问题，他向我推荐 Typora 编辑器，我觉着这就是我一直想要的 Markdown 编辑器，于是我就想到了这个主题。当然，我的水平一时是难于写出 Typora 那样的编辑器的，但人生已经如此艰难，大家就不要拆穿了，我主要是想通过一个实际应用分享一下 Angular（文中提到的 Angular 指的都是Angular2+，实际版本是 Angular5）的开发过程，主要内容包括：

- Angular 项目建立
- Angular 中的服务
- Angular 中使用第三方传统库
- 打包桌面版本
- 制作一个安装程序

Angular 项目的默认语言是 TypeScript，如果你不太熟悉请看我的另外一个Chat：TypeScript 快速入门

项目初始化

说了一大堆，我们是要使用 Angular 来开发一个新的项目，如果你以前没安装过，那么你需要先安装 Node 和 npm。然后使用下面的命令来安装 Angular/Cli，我们是通过 Angular/Cli 进行项目的管理。

```
npm install -g @angular/cli
```

如果你已经安装过老版本的 Angular/Cli，建议你使用下面的命令，升级到最新的版本。

```
npm uninstall -g @angular/cli
npm cache clean
# if npm version is > 5 then use `npm cache verify` to avoid
errors (or to avoid using --force)
npm install -g @angular/cli@latest
```

升级过程中可能会报错误，类似于：

> npm WARN ajv-keywords@3.1.0 requires a peer of ajv@^6.0.0 but none is installed. You must install peer dependencies your
> self.
>
> npm ERR! code EINTEGRITY
> npm ERR! sha1-Ua99YUrZqfYQ6huvu5idaxxWiQ8= integrity checksum failed when using sha1: wanted sha1-Ua99YUrZqfYQ6huvu5idaxxWiQ8= but got sha1-D6HriBx1DgZRGWjwAGT4GuPfJE4=. (988 bytes)
>
> npm ERR! A complete log of this run can be found in:
> npm ERR! C:\Users\sunjipeng\AppData\Roaming\npm-cache_logs\2018-02-21T06_26_16_216Z-debug.log

我们可以手动删除解决：

1. 删除用户目录下./npmrc。
2. 删除 Roaming 目录下 npm 和 npm_cache。

接下来我们要用下面的命令，新建一个项目:

```
ng new ngMDEditor
```


新建项目后，有可能因为大局域网的原因，部分依赖未成功安装，可以通过 npm i 完成安装，安装之后，可以通过如下命令，运行预览调试：

```
ng serve
```

先添加个依赖

首先，我是一个前端渣渣；其次，开源世界有很多现成的东西供我们使用，不需要我们再花精力在我们不熟悉的领域。

> 现在我们依赖别人，以后依赖人工智能，最后会不会替代？

要做一个 Markdown 编辑器，首先需要解析 Markdown，根据 Github 上 most stars 排序，JavaScript 类的 marked 排第一，网上用的人也比较多，但是我喜欢另外一个 markdown-it，主观感觉特性丰富，而且插件比较多，有 MathJax 的插件，挺好用的。

我们先安装 markdown-it，使用 npm 命令如下：

```
npm install markdown-it --save
```

因为 Angular 默认使用 Typescript，通常我们需要编写声明文件告诉 TypeScript 我们的 JavaScript 库的存在，很多库已经有现成的声明文件，所以，使用 npm 安装一下：

```
npm install --save @types/markdown-it
```

另外，我需要界面布局，前面说了，我是前端渣渣，所以我常常使用其他工具来布局和写界面，我用得最多的就是 Bootstrap，虽然这很不 Angular，还非常的 jQuery，但不得不承认如今这几种框架混用的现象是非常普遍的。我们先安装 Bootstrap，同样使用 npm 命令：

```
npm install bootstrap
```

由于 Bootstrap 还依赖 jQuery 和 popper.js，所以我们用同样的方法安装这两个：

```
npm i jquery popper.js
```


我们在根目录下，打开 .angular-cli.json，编辑 scripts 项如下：

```
  "scripts": [
    "../node_modules/markdown-it/dist/markdown-it.min.js",
    "../node_modules/jquery/dist/jquery.min.js",
    "../node_modules/popper.js/dist/umd/popper.min.js",
    "../node_modules/bootstrap/dist/js/bootstrap.min.js"
  ],
```

这样就算把这些第三方库加入到框架中了，注意那个 poper 使用的是 umd 库，用错了不行，我们也可以在 index.html 中使用传统方式引入，但这样更专业一些，Bootstrap 还包含了样式，我们修改上述文件的 styles 项如下：

```
"styles": [
    "styles.css",
    "../node_modules/bootstrap/dist/css/bootstrap.min.css"
],
```

类似的，我们可以在 styles.css 中引入项目文件，例如:

```
@import "~bootstrap/dist/css/bootstrap.min.css";
```

主界面开发

接下来，我们就正式开始编写一个 Markdown 语法的编辑器，借用流行的左右两栏模式，左边是 Markdown 语法，右边是解析渲染后的样子。我们用 Angular/Cli 命令行工具生成一个 component，也就是这个编辑器：

```
ng g c page/Editor
```

上面 ng 就是 Angular/Cli 的命令行工具，参数 g 是 generator 的简写，c 是 component 的简写，合起来意思就是生成一个组件，最后的参数就是组件的名称和位置（相对于源码）。好了，这是自动生成四个文件：

- editor.component.css
- editor.component.html
- editor.component.spec.ts
- editor.component.ts

看后缀我们知道 css 是样式，html 是页面，ts 是业务逻辑。我们先来把页面写出来：

```
<div class="container d-flex h-100 p-3 mx-auto flex-column">
  <div class="row h-100">
    <div class="col-6">
      <textarea class="h-100 w-100">这是左边</textarea>
    </div>
    <div class="col-6">
      <p>
        这是右边
      </p>
    </div>
  </div>
</div>
```


我不想详细讲太多 HTML 页面知识，总之就是左边一个文本输入框，右边在 p 元素内将页面渲染出来。

双向绑定

我们要把文本区域的内容获取，传统的做法就是通过 getElementById 等方法获取元素，然后根据不同元素的属性获取值，大多数时候使用 getElmentById(id).getValue()。但是，我们现在不是传统做法，用的是 Angular，可以使用双向绑定。

双向绑定说白了就是元素值和业务代码里的某个变量或方法是双向流通的，详细的内容可以参考官方文档，但常用的方法我要说一下，方括号 [] 就是代码值传到页面，圆括号

() 就是页面值传到代码，事件就是页面传到代码。当只有一种括号时是单向绑定，两种括号一起用时就是双向绑定。例如：

```
<input type="text" [(value)]="srcTxt"/>
```

对于 form 控件，更常用的是 ngModel，使用 ngModel 前我们需要在 module 中引入 FormsModule：

```
  imports: [
    BrowserModule,
    FormsModule
  ],
```

基于我们前面的需求，现在代码中新增一个属性 srcTxt 保存文本区域输入的 Markdown 代码：

```
srcTxt:string = "源文件"
```

然后，在页面上通过 ngModel 绑定此属性：


```
<textarea class="h-100 w-100" [(ngModel)]="srcTxt">这是左边
</textarea>
```

解析 Markdown 并显示

我们现在通过 srcTxt 获取了 Markdown 源码，接下来我们通过 Markdown-it 解析出结果并显示出来。前面我们已经添加了 Markdown-it 的类型库，现在我们引入它就可以使用：

```
import * as MarkdownIt from 'markdown-it'
```

当 Markdown 源码改变时，我们需要重新解析源码，因此在页面添加监听 ngModel 的改变：

```
<textarea class="h-100 w-100" [(ngModel)]="srcTxt"
(ngModelChange)="srcChanged($event)">这是左边</textarea>
```

在逻辑代码中实现这个监听，解析源码为 HTML 格式：

```
  srcChanged($event) {
    var md = new MarkdownIt();
    this.outTxt = md.render($event)
  }
```

这里的 outTxt 是我们定义的一个结果属性，通过页面将这个结果展现出去，页面就完成了。

```
<div [innerHTML]="outTxt"></div>
```

另外一种方式是使用 pipe，Angular 里面 pipe 的目的就是将输入的值转换为目标值，比如时间、货币、数字的格式化等。我们先写一个 Markdown 解析转换的 pipe，先生成一个：

```
ng g pipe pipe/Md
```

Angular 中内置了一些 pipe，也可以自定义 pipe，我们修改生成的 pipe，这里的代码很简单：


```
  transform(value: any, args?: any): any {
    var md = new MarkdownIt();
    return md.render(value);
  }
```

相应的，界面上需要根据 pipe 的用法改变：

```
<div class="col-6">
    <span>
        <!-- 这是右边 -->

        <!-- 通过监听动态转换 -->
        <!-- <div [innerHTML]="outTxt"></div> -->

        <!-- 通过pipe转换 -->
        <div [innerHTML]="srcTxt | md"></div>
    </span>
</div>
```

支持数学公式

首先，我们从这里下载一个 MathJax 的库，本来 Markdown-it 有 MathJax 的扩展，但是使用起来也有各种问题，就只好直接使用 MathJax 官网库，但编译后提示有文件找不到，最后就找了这么一个单文件库，总算可以了。直接加载文件是这样配置 .angular-cli.json 的：

```
        "scripts": [
          ...
          "./assets/MathJax.min.js"
        ],
```

导入库后需要配置，由于我们不是用 npm 导入的，也没有类型库，所以我们先定义以下，然后再配置：

```
declare var MathJax:any
  ngOnInit() {
    MathJax.Hub.Config({tex2jax: {inlineMath: [['$','$'], ['\\
(','\\)']]}});
  }
```

这次我们编写一个 directive 来实现这个功能，本来我打算通过 pipe，或者监听文本变化用 MathJax 解析公式，但是由于 MathJax 解析方法不太好实现，所以我们用 directive：


```
ng g directive directive/MathJax
```

directive 代码如下：

```
import { Directive, Input, ElementRef } from '@angular/core';
declare var MathJax:any
@Directive({
  selector: '[MathJax]'
})
export class MathJaxDirective {
  @Input('MathJax') fractionString: string;

  constructor(private el: ElementRef) { }
  ngOnChanges() {
    this.el.nativeElement.innerHTML = this.fractionString;
    MathJax.Hub.Queue(["Typeset", MathJax.Hub,
this.el.nativeElement]);
  }
}
```

最后，我们在标签上使用 directive 来处理公式：

```
<div [MathJax]="outTxt"></div>
```

这是我们输出公式就可以正常显示了。

保存编辑到数据库

这一节我们主要是想讲 Angular 使用 HttpClient 与远程服务器通讯的功能，这是技术上的，从功能上我们还可以将各种功能进行设计，在 Angular 中可以将它们抽象成服务，新建一个服务如下：

```
ng g s service/Md
```

通过上面的命令，我们在 Service 目录下新建了一个 Md 服务，我们要在这个服务中，添加一个将 Markdown 文档保存到服务器的功能。

Angular 通过 HttpClient 来和远程服务通讯，HttpClient 也是一个服务，要是用这个服务我们首先要在 app.module.ts 中导入 HttpClientModule，app.module.ts 变成了如下内容:


```
@NgModule({
  declarations: [
    ...
  ],
  imports: [
    ...
    HttpClientModule,
    ...
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

然后，我们将 HttpClient 注入到要使用的地方，这里具体就是 Md 服务：

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable()
export class MdService {

  constructor(private http:HttpClient) { }

}
```

现在，我们就可以通过 HttpClient 来保存数据了，新建一个服务方法：

```
save(id:number, title:string, content:string) {
    return this.http.post("http://localhost:8080/post", {id: id,
title:title, content:content})
}
```

有了服务之后，我们需要在界面中调用它，首先，需要在 app.module.ts 的 providers 中定义：

```
@NgModule({
  declarations: [
    AppComponent,
    EditorComponent,
    MdPipe
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [MdService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```


同样和 HttpClient 一样，我们需要在使用的地方注入它，我们在 Editor 页面新建一个按钮事件：

```
  <div class="row">
      <button type="button" class="btn btn-primary"
(click)="doSave()">保存</button>
  </div>
  <div class="row">
      <input type="text" [(ngModel)]="title">
  </div>
```

按钮点击通过圆括号绑定到 doSave 方法，doSave 方法通过注入的 Md 服务保存内容到服务器：

```
  constructor(private mdSvc:MdService) { }

  doSave() {
    this.mdSvc.save(null, this.title,
this.srcTxt).subscribe(data=> {
```

```
        alert("保存成功")
      }, err => {
        alert("保存失败")
      })
    }
```

我这里仅演示一下，服务使用的是我前面的 Chat：Kotlin开发SpringBoot之Data JPA，有请兴趣的朋友可以阅读该 Chat，另外一个编辑器的功能不单是保存，还有打开等功能，就留作大家的作业吧。

Electron 打包成桌面应用

现在前端的发展其实是很快的，网页、桌面、手机一网打尽了，Angular 的主页写的就是一种框架，适于多个平台。把 JavaScript 项目打包成桌面，现在主流有两种方案 NW.js 和 Electron，我查找学习的资料大多是 Electron，所以这里我们来看看如何用 Electron 把 Angular 项目打包成桌面应用。

Electron 打包桌面应用说白了就是网页套个壳，高级点可以提供 API 访问本地原生功能，基于此，我们甚至可以给一个网站做个桌面应用，我们先初始化一个 npm 的空项目：


```
npm init
```

然后编辑 index.js:

```
const { app, BrowserWindow, Menu, ipcMain, shell } =
require('electron')
const path = require('path')
const url = require('url')

let win

function createWindow() {
    //新建一个浏览器窗口作为应用的壳
    win = new BrowserWindow({ width: 800, height: 600 })
    //不要菜单
    Menu.setApplicationMenu(null)

    // GitChat的入口
    win.loadURL("http://www.gitbook.cn")

    //注册关闭事件
    win.on('closed', () => {
        win = null
    })
    isReady = true
```

```
}

/**
 * 应用就绪创建窗口
 */
app.on('ready', createWindow)


app.on('window-all-closed', () => {
    if (process.platform !== 'darwin') {
        app.quit()
    }
})

app.on('activate', () => {
    if (win === null) {
        createWindow()
    }
})
```

上面的脚本就是调用 Electron 打开一个浏览器窗口，并加载 GitChat 作为主界面。是不是很简单？运行一下看看效果：

```
electron .
```


好了，原理了解就好，我们只是应用而已，针对我们的编辑器，我们从头一步一步来。首先，我们要全局安装 Electron，因为后面需要 Electron 命令行运行：

```
npm i -g electron
```

接着，我们需要在项目中安装 Electron：

```
npm install electron --save-dev
```

同时，index.html 中的 base 路径为 / ，根目录可能会引起错误，所以我们修改为相对的当前路径：

```
<base href="./">
```

前面类似，对于 Electron 的应用有一个 js 文件作为入口，默认是 index.js，这里我们设置为 main.js，入口文件主要是初始化和启动应用。

```
const {app, BrowserWindow} = require('electron')
const path = require('path')
```

```
const url = require('url')

let win

function createWindow () {
  win = new BrowserWindow({width: 800, height: 600})

  // load the dist folder from Angular
  win.loadURL(url.format({
    pathname: path.join(__dirname, 'dist/index.html'),
    protocol: 'file:',
    slashes: true
  }))

  // Open the DevTools optionally:
  // win.webContents.openDevTools()

  win.on('closed', () => {
    win = null
  })
}

app.on('ready', createWindow)


app.on('window-all-closed', () => {
  if (process.platform !== 'darwin') {
    app.quit()
  }
})

app.on('activate', () => {
  if (win === null) {
    createWindow()
  }
})
```


上面这段代码就是 Electron 的入口程序，大意就是加载 `dist/index.html` 页面，然后对各种事件进行处理。接下来我们要让 npm 和 Electron 知道这个文件，就是配置 package.json 文件，修改内容如下：

```
{
  "main": "main.js",        // 新增，说明入口文件

  "scripts": {
    // npm 命令脚步

    "electron": "ng build && electron .",
    "electron-aot": "ng build --prod && electron .",
```

```
  },
}
```

上面的配置省略了其他配置，且 JSON 不能注释，只是为了示意，意思就是当我们运行 npm run electron 或者 npm run electron-aot 的时候会先用 ng build 输出到 dist，然后再用 Electron 执行当前目录，当前目录的入口文件是 main.js，main.js 又会去调用刚输出到 dist 的 index.html。

现在，我们运行看看效果：

```
npm run electron
```

现在虽然不是在浏览器中运行了，但还不是一个独立的应用，和我们平时的 exe 执行文件不同，为了打包成独立的可执行文件式的桌面应用，我们要安装 Electron 的打包工具：

```
npm install electron-packager -g
```

安装成功后就可以用打包工具生成桌面可执行文件了：


```
electron-packager . --platform=win32
```

上面的命令把当前目录作为一个 Electron 项目，打包成 Win32 平台的可执行程序，输出可执行行文件到 ng-mdeditor-win32-x64 目录下，双击 exe 即可执行。类似的，可以在 MacOs 上这样打包：

```
electron-packager . --platform=darwin
```

与 Electron 功能结合

前面我们给基于 Angular 的 Markdown 编辑器套了个壳，可以运行在桌面端，但有时我们需要深度集成，比如和 Electron 的菜单交互，访问本地文件，系统托盘的功能时，需要用到 Electron 的一些功能，这一节我们就尝试深度的集成一下 Electron。

首先看到运行的程序界面中是无关的菜单，我们需要自定义菜单，首先自定义一个菜单模版，如下所示：

```
var template = [
    {
        label: '文件',
```

```
            submenu: [
                {
                    label: '保存到服务器',
                    click: () => saveToSvr()
                },
                {
                    label: '保存到本地',
                    click: function ()
{winOpen('http://www.baidu.com')}
                }
                ,
                {
                    type: 'separator'
                },
                {
                    label: '退出',
                    click: function()
{winOpen('http://www.baidu.com')}
                }
            ]
        },
        {
            label: '帮助',
            submenu: [
                {
                    label: '我的github',
                    click: () =>
winOpen('https://github.com/chengang4505')
                },
                {
                    label: '我的博客',
                    click: () =>  winOpen('http://blog.techcave.cn')
                },
                {
                    label: '关于',
                }
            ]
        }
];
```


解析该模版，设置为应用菜单，修改 main.js 相关代码：

```
const menu = Menu.buildFromTemplate(template)
Menu.setApplicationMenu(menu)
```

上面菜单中，我们设计了菜单项“保存到服务器”，但是业务逻辑是实现在 Angular 中，怎么通知 Angular 执行呢？我们通过主进程发送一个消息，表明菜单被点击了：

```
function saveToSvr() {
    win.webContents.send('save-to-serve')
}
```

然后，我们需要在 Angualr，对于 Electron 来说就是渲染进程中接收这个消息。Electron 也提供了相应的 API 来实现该功能，确实也存在 `@types/electron` 包让 TypeScript 开发者来使用 Electron，但实际上 Electron 依赖于 window.require() 方法，而这个方法在 Electron 的渲染进程中不存在（这里主要说 Angular）。直接访问 Electron 的 API 会导致一些错误，通过搜索引擎可以查到大量开发者在寻求 Angular 中访问 Electron API 的方案。所以就产生了一个组件 ngx-electron 来解决这个问题。和其他 npm 安装一样：

```
npm install ngx-electron --save
```

安装完成后在 AppModule 中引入 NgxElectronModule：

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {NgxElectronModule} from 'ngx-electron';
import {AppComponent} from './app.component';

@NgModule({
    declarations: [],
    imports: [
      BrowserModule,
      NgxElectronModule
    ],
    bootstrap: [AppComponent]
})
export class AppModule {

}
```


NgxElectronModule 中提供了服务组件 ElectronService，该服务封装了 Electron API，我们通过该服务来访问 Electron API 功能：

```
  constructor(private es: ElectronService, private
mdSvc:MdService) {
    this.es.ipcRenderer.on('save-to-server',
this.doSave.bind(this));
  }
```

上面这段代码注入了 ElectronService，通过该服务实现监听绑定 save-to-server 消息。

打开 Electron 的 DevTools 界面

说到底，Electron 也是套了个 Chrome 的壳，整个程序跑起来后调试用 DevTools 会方便很多。打开此界面也很容易，语句如下：

```
win.webContents.openDevTools()
```

打开和保存本地文件

Electron 可以访问原生功能，比如打开保存本地文件，思路是先在 Electron 主进程操作原生功能，然后通过 Electron 与渲染进程通讯，将操作结果返回给渲染进程（这里指 Angular）。

```
function openFileDlg() {
    dialog.showOpenDialog(win, {
        properties: ['openFile']
    }, function (files) {
        if (files) {
            fs.readFile(files[0], 'utf-8', function (err, data) {
                event.sender.send('selected-file', data)
            })
        }
    })
}
```


上面这段代码在打开文件后读取内容，然后将内容通过 selected-file 消息发送出去，接下来就是接收了，我们在 Angular 程序中代码如下：

```
this.es.ipcRenderer.on('selected-file',
this.onFileSelected.bind(this))

onFileSelected(event, files) {
    this.srcTxt = files
}
```

这段代码现在初始化时注册事件回调，当文件打开后接收来自 Electron 的文件内容，然后在回调中，将文件内容赋值给本地变量，该变量绑定到了界面的编辑框。

理想是美好的，但是残酷的现实来了，通过消息获取打开的本地文件后，ngModel 并没有变化，也就是 Angular 没有侦测到属性值的变化，需要点击一下界面文件内容才会显示出来，这显然是一个比较糟糕的体验，百思不得其解，谷歌一下，有以下几种方法可以尝试：

- ApplicationRef.tick() - similar to AngularJS's `$rootScope.$digest()` – i.e., check the full component tree
- NgZone.run(callback) - similar to `$rootScope.$apply(callback)` – i.e., evaluate the callback function inside the Angular zone. I think, but I'm not sure, that this ends up checking the full component tree after executing the callback function.
- ChangeDetectorRef.detectChanges() - similar to `$scope.$digest()` – i.e., check only this component and its children

上面的方法会需要注入 ApplicationRef、NgZone 或 ChangeDetectorRef 到编辑器组件中。

上面几种方法我都尝试了个遍，但不能完全解决问题，貌似 this 无法完全指向当前组件，最后我把 this 和 zone 保存到全局竟然可行，最后的方案如下：

```
// 初始化通过全局变量保存环境信息
window['mdEditor'] = { component: this, zone: zone };
// 通过全局变量操作赋值
  onFileSelected(event, files) {
    let that = window['mdEditor'].component;
    let zone = window['mdEditor'].zone;
    zone.run(() => {
      that.srcTxt = files;
    })
  }
```


文件可以打开了，反过来我们编辑完文档需要保存，流程是反的应该是这个样子：

> Anular -> 保存消息 -> Electron -> 保存操作

Angular，也就是界面渲染进程发送一个消息：

```
this.es.ipcRenderer.send("save-file", this.srcTxt);
```

这个消息的名称是 save-file，同时把文件内容作为消息参数传了过去，Electron 主进程是这样接收消息的：

```
    ipcMain.on('save-file', (event, arg) => {
        dialog.showSaveDialog(function (fileName) {
            if (fileName === undefined) {
                console.log("You didn't save the file");
                return;
            }
            saveChanges(fileName, arg);
        });
    })
```

代码先打开对话框，当选择了保存文件时将收到的文件内容保存到文件，保存逻辑调用的 fs 模块，具体逻辑如下：

```
function saveChanges(filepath, content) {
    fs.writeFile(filepath, content, function (err) {
        if (err) {
            console.log(err);
            return;
        }
    });
}
```

注册快捷键

通常，Windows 下面我们会使用 Ctrl+s 来保存文件，现在，我们注册一下刚才的保存功能演示一下：

```
const {app, globalShortcut} = require('electron')

  app.on('ready', () => {
    // Register a 'CommandOrControl+S' shortcut listener.
    globalShortcut.register('CommandOrControl+Y', () => {
      win.webContents.send('to-save-file')
    })
  })
```


首先，我们引入 globalShortcut，然后在 App Ready 的时候，注册 CommandOrControl+Y，响应的逻辑代码就是刚才的保存文件。当然你也可以写别的逻辑，具体就根据业务需求来了，我们这里仅演示一下，功能设计不一定合理。

Typora技术初探

虽然写不出 Typora 那样的编辑器，但是内心还是有那样的向往，心里一直在琢磨应该怎样实现呢？通过 Typora 的视图菜单，可以看见一个打开 DevTools 的菜单项，类似于 Chrome 浏览器的开发者工具。打开 source，有下面这样一个路径：

> resource/app/lib.asar/codemirror/mode.min.js

可以猜测，Typora 是为 Markdown 语法写了一个 CodeMirror 的 mode，CodeMirror 是一个语法高亮的代码编辑器，不同的代码通过 mode 来扩展，如果你会解析语法你也可以

为 Markdown 写一个这样的 mode，可能 Typora 就是写了一个，只不过把高亮样式改成了 Markdown 的样式而已。

NSIS 制作安装包

专业的软件都有安装程序，我们今天就是要用 NSIS 给我们编写的 Markdown 编辑器做一个安装程序。

NSIS 是“Nullsoft 脚本安装系统（Nullsoft Scriptable Installation System）”的缩写，它是一个免费的 Win32 安装、卸载系统，它的特点是脚本简洁高效、系统开销小，当然进行安装、卸载、设置系统设置、解压文件等等更不在话下。

NSIS 做安装程序，说简单也简单，说复杂那就深了去了，我们本着够用就行的原则写一个最小实践，更深的东西大家可以在此基础上扩展。

NSIS 制作安装程序主要是编写脚本，然后编译打包，网上还有一些向导类工具用可视化的方式形成脚本，可能更容易，我没有用过，觉得一般的手写脚本也够用了。好了，我们先给出脚本，一个最小实践：


```
; 这就是我们的Markdown编辑器的安装脚步
; 通过分号来注释。

;--------------------------------
;--------------------------------
; 可以通过include 引入外部第三方组件，我们简单例子就不管了
; 这里注释了，演示一下就好
;!include "EnvVarUpdate.nsh"

; 这个是安装程序的名字，如果其他系统，照着改就好了。
Name "我自己的Markdown编辑器"

; 安装程序的输出文件，如果其他项目照猫画虎修改即可。
OutFile "ngMDEditor_setup.exe"

; 设置默认的安装目录
InstallDir $PROGRAMFILES\ngMDEditor
; InstallDir D:\test\install

; 请求相关的安装权限，有时候需要
; RequestExecutionLevel user
; 下面这句以管理员身份运行
RequestExecutionLevel admin
; 授权文本，假装重视版权，从其他地方抄了个软件协议过来。
LicenseData Eula.txt
; 使用中文显示
LoadLanguageFile "${NSISDIR}\Contrib\Language
files\SimpChinese.nlf"
```

```
;--------------------------------
;好了从这里进入整体，前面都是参数设置
;NSIS 把安装程序分为几个部分，首先是页面或者界面


; 页面
; 授权界面
Page license
; 安装目录的界面
Page directory
; 安装文件界面
Page instfiles
; 卸载确认页
UninstPage uninstConfirm
; 卸载文件
UninstPage instfiles

;--------------------------------

; 这里就是安装的内容，素材？还是怎么说，小节
; 没有多余的组件，所以这个名字不是很重要
Section "" ;

  ; 设置输出到安装目录
  SetOutPath $INSTDIR
  ; 你还可以在过程中执行脚步文件
  ; 但是我们没有，这里注释了，你懂的就行
  ; ExecWait "install.bat"

  ; 可以包含什么文件
  ; File Main.java
  ; /r就是包含子路径
  ; File /r pkg\Claz.java
  ; /x 意思是忽略不包含，有点类似gitignore，
  ; 下面语句包含子路径，不包含ngMdEditor.nsi，不包含data.rar，不包含
Data，复制*.*
  File /r /x ngMdEditor.nsi /x data.rar /x data /x msdata /x
line.mdb *.*

  ; 同样是执行批处理，和前面那个可能是不同版本的语法
  nsExec::Exec "$INSTDIR\install.bat $INSTDIR"

  ; 创建桌面快捷方式
  CreateShortCut "$DESKTOP\我的Markdown编辑器.lnk"
"$INSTDIR\ngMDEditor.exe"

  ; 设置环境变量PATH，这是通过前面导入的模块，已经注释
  ; ${EnvVarUpdate} $0 "PATH" "A" "HKLM" "$INSTDIR\bin"
SectionEnd ; end the section
```

脚本已经够详细了，简不简单？说实话再复杂我也不会了。但基本够用了，可以做出一个装模作样的安装程序，如果你要添加更多功能，这个脚步也能帮你理解，够你扩展了。

总结

本次分享按整个开发过程捋了一下，说了一些关键的地方，其余的内容都在代码里了。如果你想看某个点，阅读文章，如果你想看看全貌，阅读代码，如果你想操作运行，请动手。

完整代码请移步：gitee。