2015年12月10日

realtime-markdown富文本编辑器

一个单行转换markdown格式为html格式的AngularJs组件

一、 引文

AngularJs的学习主要集中在10月中旬到11月中旬，主要阅读的书籍包括《用AngularJS开发下一代Web应用》、《精通AngularJs》、《AngularJs Text Driven Development》。AngularJs部分的学习内容主要包括：双向数据绑定、服务、指令、过滤器、模板、路由和与服务器通讯等内容。当时和导师沟通的时候，也就定了做一个AngularJs组件，作为AngularJs学习内容的考核标的。

一次偶然的机会，我发现了Typora Markdown编辑器，它和传统的Markdown编辑器最大的不同点就是，其是所见及所得，并且实时转化markdown格式为html格式，并替换markdown文本显示到编辑器上。传统的markdown编辑器有两个部分，一个部分是书写markdown的编辑器，另一个部分是实时预览的显示框。这样的一个弊端就是一边编辑还得一边看转换后的格式，显然不够方便和简洁。正如Typora官网（http://www.typora.io/）这样介绍到Typora：

『Typora will give you a seamless experience as both a reader and a writer. It removes the preview window, mode switcher, syntax symbols of markdown source code, and all other unnecessary distractions. Replace them with a real live preview feature to help you concentrate the content itself.』

不过遗憾的是，Typora现在仍然处于beta阶段，在编辑一些简单的短小的文本还是挺好用的，如果使用Typora编辑大段文本，甚至是写作书籍，难免会出现一些无法调节的bug，而且Typora现阶段只有Mac 版、Window和Linix版本。并没有网页版，也就是说Typora无法整合到Web 应用中，对于一个前端工程师而言，不能够整合到Web 应用中，这不就像鸡肋吗？食之无味，弃之可惜。

于是便产生了自己写一个类似于Typora的markdown编辑器，并且可以使用到Web应用中。我将这个编辑器命名为：realtime-markdown。现在编辑器还处于beta的beta阶段，欢迎大家体验，提交issue。

项目地址：https://github.com/Jocs/realtime-markdown

演示地址：http://rtmarkdown.duapp.com/（现阶段仅支持Chrome浏览器）

二、realtime-markdown的现有功能及实现原理

2.1 realtime-markdown 现有功能及对markdown语法的一些取舍

进入演示地址，左边就是realtime-markdown编辑器，如果有上传本地图片，本地图片会同步发送到简单服务器上一个某个特定位置。同时realtime-markdown会把编辑器内的html中的img元素的src替换为服务器上图片的相对路径，并把html发送到简单服务器上，服务器再把html发回到客户端，并显示到显示地址网页的显示框内。其实也就是一个客户端和服务端数据绑定的过程。只要客户端数据有更新，服务端的数据也就相应发生变化。上面是演示页面的一个简单工作原理。

现在realtime-markdown支持如下一些markdown语法：

1）支持atx形式的标题

类Atx形式则在行首插入1到6各#。对应到标题1到6阶，例如：

这是H1

这是H2

这是H3

2） 区块引用Blockquotes

Markdown的区块引用类似于email中的区块引用，就是在每行行首添加”>”的引用方式，在realtime-markdown中输入如下内容敲回车就会生成一个引用块；

> 您输入的内容

不过现在realtime-markdown还不支持引用嵌套，也就是在引用区块内还不支持有序列表、无序列表、标题等。

3）列表

realtime-markdown现支持有序列表和无序列表。

无序列表使用星号、加号、减号作为列表标记：

* hello
+ world
- foo

有序列表使用一个数字紧接一个英文句点和空格作为有序列表标记：

1. hello
2. world

3. bar

4）代码区块

和程序相关的写作或是标签语言原始码通常会有已经排版好的代码区块，通常这些区块我们并不希望它以一般段落文件的方式去排版，而是照原来的样子显示，Markdown会用 <pre> 和 <code> 标签来把代码区块包起来。

还在realtime-markdown中生成一个代码区块相当简洁，只需要在行首输入三个「`」，敲击回车，就会生成一个CodeMirror代码块，并且光标会出现在代码块中。为了实现代码实时高亮，realtime-markdown中的代码块并不一定在<pre> 和 <code>标签中，而是CodeMirror代码块，CodeMirror将在后面介绍到。CodeMirror会对输入的代码进行实时高亮显示。现阶段，realtime-markdown仅支持javascript代码高亮。

5）分割线

你可以在一行中用三个以上的星号、减号、底线来建立一个分隔线，行内不能有其他东西。下面每种写法都可以建立分隔线(输入后记得敲击回车):

***

___

- - -

6）区段元素

连接

realtime-markdown支持行内式的连接。要建立一个行内式的链接，只要在方块括号后面紧接着圆括号并插入网址链接即可，如果你还想要加上链接的 title 文字，只要在网址后面，用双引号把 title 文字包起来即可，例如:

This is [an example](http://example.com/ "Title") inline link.

转换后html如下:

This is

an example inline link.

强调

在realtime-markdown中，使用星号和底线作为标记来强调文字，被一边一个星号（底线）包围会被转化成标签包围，被一边两个星号（底线）包起来的话，则会被转化为。例如:

single asterisks

single underscores

double asterisks

__double underscores__

会被转化为：

single asterisks

single underscores

double asterisks

double underscores

在realtime中，你可以使用任意一种强调的语法，唯一的限制就是，用什么符号开启强调，就需要用什么符号来结束。

代码

在realtime-markdown中，通过两个「`」包裹一个单词，就会产生一个代码小段。例如：

Use the `printf()` function.

会被转化为：

Use the <code>printf()</code> function.

如果需要在代码片段内插入反引号「`」，则可以使用两个反引号来开启和结束代码区段。（注意：不要使用三个或者超过三个的反引号，这样敲击回车会产生一个代码块，而不是代码区段）。

图片

在realtime-markdown中可以使用行内式的语法来插入网络图片，语法如下：

![Alt text](/path/to/img.jpg "Optional title”)

realtime-markdown有个比较强大的功能就是可以插入本地图片，通过组合键Control + Alt + i，可以弹出一个文件选择框，你可以选择本地图片插入到编辑器中，你也可以使用拖拽，把图片拖拽到编辑器中，realtime-markdown可以自动帮你把图片上传到服务器上。并且替换图片的src为服务器上的图片相对路径。

2.2 realtime-markdown的实现原理

在realtime-markdown中引用了两个外部库，一个是CodeMirror（http://codemirror.net/），另外一个是showdown。CodeMirror可以对CodeMirror中的代码进行实时高亮显示，并且支持多种语言和主题。现在有些代码编辑器内部使用的就是CodeMirror库。showdown的作用就是对markdown格式的文本转化为html文本。

realtime-markdown编辑器实际上就是一个带有contenteditable属性的div元素，然后通过AngularJs的指令，对contenteditable指令进行了功能增强。当每输入一行代码，敲击回车，就会通过showdown库把当前行的markdown 文本转化为html，然后进行替

换。对于一些多行的markdown语法可能就没法通过showdown来进行转换了，这个时候就需要手动进行模式匹配，比如列表等。然后替换为相应的html。比如输入三个「```」敲击回车，会生成一个代码块，这就不是通过showdown转化的，而是手动将『```』所在元素替换为CodeMirror代码块。

realtime-markdown具体实现细节将在第三部分详细叙述。

三、在实现realtime-markdown过程中遇到的问题

在写realtime-markdown时，没有参考别人的代码，踩了很多坑，遇到了很多问题，其中最大的问题就是控制光标的位置。当然还包括其他一系列问题，比如，在生成列表的时候，怎么使后面输入的内容整合到前面的UL列表或者OL列表中，也就是说，后面输入的列表项，它不会单独生成一个UL元素或者OL元素。而是转化为前面一个UL或者OL元素中的一个LI元素。在Markdown语法格式高亮、CodeMirror代码块中都遇到了相当多的问题。在本部分将按照realtime-markdown的实现过程，详细论述实现细节及遇到的问题。

3.1 contenteditable指令的Main Code

```
angular.module('realtime-markdown', ['realtime-markdown.service'])
.directive('contenteditable', function(){
    return {
        restrict : 'A',
        controller: contenteditableController,
        require :['contenteditable'],
        compile: function(element){
            element.html('<div><br></div>');
            replaceInput();
            return {
                pre : function(scope, element, attrs, ctrls){},
                post : postLinkFn
            }
        }
    }
});
```

这也是贯穿整个realtime-markdown的代码。contenteditableController提供contenteditable指令所需的一些API。在compile函数中，首先为了兼容各大主流浏览器，在div[contenteditable]中，通过innerHTML插入「'
 '」。这也保证了

div[contenteditable]元素内不会出现文本子节点，这因为元素节点相对于文本节点更方便转化。

replaceInput();的作用是生成一个input[type = 'file']的不可见输入框，用来插入图片所用。最后compile函数返回一个preLink函数和postLink函数。preLink函数暂时留空，重点就是postLinkFn了。在postLinkFn中绑定了各种事件监听和DOM操作。realtime-markdown的核心内容也就是监听各种键盘事件，判断keyCode值，进行相应操作，或者DOM替换等。

3.2 监听Enter键，进行Markdown格式转化为Html格式所遇到的问题

在realtime-markdown中，通过「md-dirty」的className来标记该行文本已经被转换了。但是在div[contenteditable]中，敲击回车，生成的新的空行也可能自动带有「md-dirty」的className，如果带有md-dirty，该行文本也就没法被转换了，因此在敲击回车时，需要把最后空行的「md-dirty」className去掉。通过如下代码去掉。

```
if(activeNode.nodeName !== 'OL'
&& activeNode.nodeName !== 'UL'
&& !activeNode.classList.contains('CodeMirror')
&& $(activeNode).hasClass('md-dirty')){
        $(activeNode).removeClass('md-dirty');
};
```

当然，如果光标所在行是OL UL CodeMirror元素，就不删除md-dirty的类名了，这样可以避免列表或者代码块被错误转化了。

通过/^\d\.\s/来匹配有序列表，通过/^[*\-\+]{1}\s{1}/来匹配无序列表。这样如果检测到一行文本是列表项，再判断其前面一行是否是UL 或者 OL。如果是，就将当前列表项转化为html，在和前面的列表项合并成一个UL或者OL。实现代码如下：

```
if((preToTransfor && /^\d\.\s/.test(preToTransfor.textContent)
&& prePreNode && prePreNode.nodeName == 'OL' )
||(preToTransfor && /^[\*\-\+]{1}\s{1}/.test(preToTransfor.textContent)
&& prePreNode && prePreNode.nodeName == 'UL' )){
        text = preToTransfor.textContent;
        html = $(ctrl.makeHtml(text));
        $(prePreNode).append(html[0].firstElementChild);
        $(preToTransfor).remove();
}
```

通过/^`{3}/来匹配代码块，如果匹配成功，就通过CodeMirror来替换当前行为一个CodeMirror代码块。并通过focus（）方法使得代码块自动获取光标，添加count属性，用于删除代码块时计数，添加md-dirty类名，保证CodeMirror代码块不会被误删除或者错误转化。实现代码如下：

```
else if(preToTransfor && /^`{3}/.test(preToTransfor.textContent)){
    //将prePreNode替换为CodeMirror文本编辑器，并删除preToTransfor元素
    ctrl.CodeMirror(preToTransfor);
    var activeElement = getActiveElement();
    var codeMirror = activeElement.previousElementSibling;
    var CodeMirrorLines =
activeElement.previousElementSibling.querySelector('textarea');
    codeMirror.count = 0;
    codeMirror.classList.add('md-dirty');
    CodeMirrorLines.focus();
}
```

如果以上情况都不匹配的话，那么就直接可以通过showdown进行转化了，如果转化后元素内包含img标签，也就是说插入了图片，那么就向该元素添加一个md-image类名。

3.3 当敲击Backspace按键时所进行的操作和所遇到的问题

Backspace的keyCode为8.

首先，如果div[contenteditable] 内的内容为空，也就是说编辑器内没有内容的时候，如果按了BackSpace按键，这样会有一个副作用就是把最初的一个’
 ’删除掉。好不容易放进去的一个兼容各大浏览器的元素就被删除了，这样就需要再次向div[contenteditable] 里面添加一个’
 ’。并且使得该添加的元素获取光标。整个代码如下：

```
if(!/\S+/.test(element.text()) && $(element)[0].getElementsByTagName('img').length == 0) {
    var range, userSelection;
    $(element)[0].innerHTML = '<div><br></div>';
    userSelection = window.getSelection();
    range = userSelection.createRange? userSelection.createRange():
userSelection.getRangeAt(0);
    range.setStartAfter($(element)[0].firstChild.firstChild);
```

```
        range.setEndAfter($(element)[0].firstChild.firstChild);
        userSelection.removeAllRanges();
        userSelection.addRange(range);
};
```

其次，如果按下BackSpace时，使得该行代码为空了，也就是

!/\S+/.test(activeNode.textContent)为true时，需要删除该行md-dirty类名，使得该行恢复可以转化的状态，当然该行元素不能够包含img标签，如果元素内有img标签，就不能够删除md-dirty类名了，否者会造成图片误删的问题。也就是说图片会被showdown转化为空字符串了。

按下BackSpace遇到的最大问题就是，删除CodeMirror代码块的问题，在没有处理前，按BackSpace是没法删除CodeMirror代码块的。这就造成了生成代码块但是没有办法删除的问题。在按下BackSpace时，如果当前行为CodeMirror代码块，并且判断CodeMirror代码块内容是否为空。如果为空的话就通过

activeNode.parentNode.removeChild(activeNode);

来删除该代码块，并且使代码块下一行开始位置获取光标。在这里还涉及到了一个删除代码块计数的问题，当代码块第一次出现为空时，不删除代码块，count ++。当再次按下BackSpace时，count为1，删除代码块，这样体验会更好。而删除代码块的难点不是控制光标，也不是删除计数。而是判断代码块为空。因为即使代码块内没有代码，代码块也不是空的。最里层也会有一个零宽空格。而难点就是匹配这个零宽空格。也就是comfirmCodeMirrorEmpty函数。comfirmCodeMirrorEmpty整个代码如下：

```
/**
 * [comfirmCodeMirrorEmpty 用于判断CodeMirror是否为空]
 * @return {[Boolean]} [为空返回true， 不为空返回false]
 */
function comfirmCodeMirrorEmpty(){
        var activeElement = getActiveElement();
        var code = activeElement.querySelector('.CodeMirror-code');
        var codeCounts = code.children.length;
        var text = code.firstElementChild.querySelector('.CodeMirror-line').textContent;
        // /\u200b/匹配零宽空格
        return codeCounts == 1 && /\u200b/.test(text);
}
```

通过getActiveElement方法获取当前光标所在行，也就是CodeMirror代码块，判断现在代码块只有一行，通过/\u200b/.test(text);来判断text是否是零宽空格。如果都为true的话，说明当前代码块为空。

3.4 监听keydown事件时遇到的问题

有时需要监听keydown事件，而不是keyup事件，比如在判断使用了组合键的时候就需要监听keydown事件了。比如通过组合键control ＋i来弹出图片选择框。实现代码如下：

```
//图片上传input输入框的显示
if(e.target.hasAttribute('contenteditable') && e.ctrlKey == true && e.keyCode == 73){
    var input = document.querySelector('.md-hiddenInput');
    input.click();
}
```

在监听keydown事件遇到的难题：如果光标是代码块最后一行时，按向下箭头（keyCode = 40）时，光标就需要跳出代码块，移动到下一行的开始位置。而难点就是记录光标的最后高度，用于判断是否是在最后一行。通过一个变量lastCursorHeight来记录光标的最后位置，通过如下代码来对该变量值进行更新。

```
var activeElement = getActiveElement(), userSelection = window.getSelection();
if(e.keyCode !== 40 && /^CodeMirror/.test(activeElement.className)) {
    var lastCursor = activeElement.querySelector('.CodeMirror-cursor');
    lastCursorHeight = parseInt(lastCursor.style.top) + parseInt(lastCursor.style.height);
}
```

也就是说，只要不是按向下箭头时，都会记录lastCursorHeight的高度。如果按下向下箭头，计算一个当前cursorHeight，如果cursorHeight与lastCursorHeight相等，就说明光标在最后一行了，然后通过创建range，是下一行元素获取光标。代码如下：

```
if(e.keyCode === 40 && /^CodeMirror/.test(activeElement.className)){
//如果pre是最后一个元素，那么就向pre后面添加一个换行元素。
    if(!activeElement.nextElementSibling){
        $(element)[0].appendChild(getBrElement());
    }
    var cursor = activeElement.querySelector('.CodeMirror-cursor');
    var cursorHeight = parseInt(cursor.style.top) + parseInt(cursor.style.height);
    if(cursorHeight == lastCursorHeight) {
```

```
        var range;
        range = userSelection.createRange? userSelection.createRange():
        userSelection.getRangeAt(0);
        range.setStartAfter(activeElement.nextElementSibling.firstChild);
        range.setEndAfter(activeElement.nextElementSibling.firstChild);
        userSelection.removeAllRanges();
        userSelection.addRange(range);
} else {
        lastCursorHeight = cursorHeight;
        }
};
```

如果光标所在行是在代码块下，按向上箭头时，代码块需要获取光标。代码块获取光标也是通过focus()方法，这儿就不赘述。

3.5 监听drop事件时需要注意的地方

监听drop事件的目的主要是实现图片拖拽上传的功能。通过confirmInEditor方法判断拖拽的target是否在编辑器内。confirmInEditor实现原理很简单，就是判断e.target的祖先元素是否有contenteditable属性。如果有就说明其在编辑器内，如果没有这说明drop的target没有在编辑器内部。

当drop事件发生时，会遍历e.dataTransfer.files。发送图片到服务器，同时通过FileReader对象，通过readAsDataURL API把图片读成一个dataUrl，这个url可以直接作为img元素的src属性值。这样，reader的load事件发生时，图片也就能够在编辑器中显示出来了。sendImage 和readFile方法都写在了contenteditable指令的控制器中。realtime-markdown是通过socket来发送图片的。sendImage代码如下：

```
        this.sendImage = function(file){
                var data = {
                        type : file.type,
                        name : file.name,
                        file : file
                };
                socket.emit('uploadImage', data);
        };
```

而readFile方法主要通过FileReader对象，因为read File是一个异步的过程，因此在readFile中使用到了promise对象。readFile代码如下：

```
this.readFile = function(file){
        var reader = new FileReader();
        reader.readAsDataURL(file);
        var promise = new Promise(function(resolve, reject){
                reader.addEventListener('load', function(e){
                        resolve(e.target.result);
                });
        });
        return promise;
};
```

在监听drop事件最后一个注意点就是，图片是放在e.target前面。而不是在编辑器最后，在监听图片输入框change事件时，图片是插入到编辑器内最后位置。

3.6 发送输入框中html的300ms延迟

发送数据到服务器的300ms延迟的目的主要就是减少和服务器的交互，节约资源，减少服务器压力。之所以选择300ms是因为，300ms内的延迟，用户一般是感受卡顿的。

实现代码如下：

```
        var timer = null;
        $(element).bind('keyup', function(){
                if(timer) clearTimeout(timer);
                timer = setTimeout(function(){
                        ctrl.sendArticle($(element)[0].innerHTML);
                }, 300);
        });
```

3.7 realtime-markdown遇到的最大问题：markdown语法高亮和光标的移动

markdown语法格式高亮的实现原理是，监听textInput事件，然后通过正则匹配markdown语法特殊字符，如果匹配到了字符，就在字符外面包裹一个带有特定样式的span标签，这样特殊字符就显示除了语法高亮，而实现的难点就是加上包含span标签后，需要把光标设置到原来替换前光标的位置，最麻烦的事就是计算光标的lastOffset。

markdown语法高亮第一步，书写正则表达式：

```
var HEADER_RPG      = /(^#{1,6})(?=[^#])/g,
QUOTE_RPG           = /(^|[^\\])(`)([^`]+)(`)(?=[^`])/g,
LINK_BRACKET_RPG    = /(^|[^\\])(\[)([^\[\]]*)(\]\()([^\(\)]*)(\))(?=[^\)])/g,
PICTURE_BRACKET_RPG = /(^|[^\\])(\!\[)([^\[\]]*)(\]\()([^\(\)]*)(\))(?=[^\)])/g,
EM_RPG              = /(^|[^\\\*])([*]{1})([^*]+)(\2)(?=[^\*])/g,
STRONG_RPG          = /(^|[^\\])([*]{2})([^*]+)(\2)(?=[^\*])/g,
DISORDER_RPG        = /(^[*-]\s)(?=[^\s])/g,
ORDER_RPG           = /(^\d\.\s)(?=[^\s])/g,
ESC_RPG             = /(\\)(?!\\)/g,
KAN_JI_RPG          = /(<\/span\>)([^\u4e00-\u9fa5]+)([\u4e00-\u9fa5]+)$/g;
```

从上往下依次是匹配markdown 标题正则，markdown的反引号正则（用于生产代码区段），markdown的链接语法正则，markdown的图片引用语法正则，markdown斜体的正则，markdown文字强调的正则、无序列表正则、有序列表正则、转义字符的正则、以及最后一个匹配汉字的正则（暂时没有使用到这个正则表达式）；

通过testText方法，来判断输入的文本是否匹配上面的正则表达式，如果匹配就实现包裹span标签的转化。在获取输入文本时有个小技巧，这个小技巧的作用就是消除在进行语法高亮时，拼音和汉字同时显示的bug。技巧代码如下：

```
var promise = new Promise(function(resolve, reject){
    setTimeout(function(){
        var text = activeElement.textContent || activeElement.nodeValue;
        resolve(text);
    }, 0);
});
```

如果不通过0秒延迟，那么在输入汉字的时候直接获取到的就是输入的拼音，而不是汉字本身，因为在使用拼音输入法的时候，拼音是先输入到输入框的，敲击空格，拼音才会被替换为汉字。因此不通过0秒延迟，通过textContent或者nodeValue可能就只能够获取到拼音，最后转换后拼音和汉字就会同时显示在输入框内。通过0秒延迟就会等待拼音转化为汉字后在获取汉字，五笔等其他输入法应该也类似吧，其他输入法还没有测试过。（其实这儿有个好玩的，虽然是0秒延迟，其实真正延迟时间可能不止0秒，各大浏览器都对最小间隔做了一个限制，大概在4ms，而且浏览器是单线程，因此异步代码可能会等待其他代码运行完成后才能执行，因此延迟应该在4ms以上，而在NodeJs中，没有4ms限制）。

在进行代码高亮时另外一个困扰了我好久的问题就是，如果一个符号比如说「*」进行高亮了，外面包裹了一个带有特殊样式的span标签，在紧接着这个星号后面输入，输入的文字也会进行自动高亮，也就是说输入的文字在前面那个span标签内，而不是标签外，而我们需要高亮的文字就只有星号，而后面的文字也被高亮了，怎么解决怎么问题呢？一开始我是想着设置光标位置，把光标位置移动到span标签外，试了几种方法也不行，后来灵机一动，输入的文字会自动高亮吗？我把高亮的span标签个去掉，这样不就不会自动高亮了吗？等下次高亮转化的时候，符号还是会被高亮，而文字就不会高亮了，最后再设置一下光标的位置，就不完美的解决了上述问题，不完美的原因是符号也取消了高亮，只能够等待下次转化才能高亮了。实现代码如下：

```
if(userSelection.focusNode.parentNode.classList.contains('rt-grey')
&& /`|\*|\)|\]|#|\s|\(|\[/.test(userSelection.focusNode.nodeValue)){
        var range = document.createRange();
        var offset = userSelection.focusOffset;
        var node = userSelection.focusNode.parentNode;
        var textNode = document.createTextNode(userSelection.focusNode.nodeValue);
        node.parentNode.insertBefore(textNode, node);
        node.parentNode.removeChild(node);

        range.setStart(textNode, offset);
        range.setEnd(textNode, offset);
        range.collapse(true);
        userSelection.removeAllRanges();
        userSelection.addRange(range);
}
```

markdown语法高亮最烦的地方，也是困扰我最久的一个问题，就是记录高亮前光标的位置，到文本被加上span标签后，光标还能够移动到原来高亮前的位置。移动光标很多坑，这也可能是CodeMirror为啥放弃了原生光标，自己画一个光标的原因之一吧。首先我通过getLastOffset来获取光标的最后位置：实现代码如下：

```
function getLastOffset(){
        var count = 0;
        var focusNode = userSelection.focusNode;
        while(focusNode.previousSibling){
```

```
            focusNode = focusNode.previousSibling;
            var plus = focusNode.textContent?
            focusNode.textContent.length : focusNode.nodeValue.length;
            count += plus;
        }
        return userSelection.focusOffset + count;
}
```

实现原理就是遍历光标所在元素之前的所有元素，取出其textContent或者nodeValue，计算其长度，然后加上光标所在元素的偏移值，这就是光标最初的字符偏移值了。

移动光标的第二步就是把光标移动到替换格式后的原来位置，找到替换后光标应该在的位置和移动光标是通过findCurrentNode方法来完成了。整个实现代码如下：

```
function findCurrentNode(range, element, offset){
        var count = 0, childNodes = element.childNodes;
        for(var i = 0; i < childNodes.length; i ++){
            count += childNodes[i].textContent.length;
            if(count >= offset) {
                count -= childNodes[i].textContent.length;
                break;
            }
        }
        var pos = { position: i, offset: offset - count };

        var currentActiveNode = element.childNodes[pos.position] || element.lastChild;

        range.setStart(currentActiveNode, pos.offset );
        range.setEnd(currentActiveNode, pos.offset );
        range.collapse(true);
        userSelection.removeAllRanges();
        userSelection.addRange(range);
}
```

range一个比较恶心的地方就是，其不能够设置整个的偏移值，只能够设置在某个元素内的偏移值，这样我们就还需要计算光标需要移动到哪个元素中，并且在元素中的位置。pos对象就记录了光标需要移动到的位置，position表示元素的一个index，offset是

光标在该元素的偏移。最后通过setStart和setEnd来设置光标，通过removeAllRanges来移除之前的所有光标，通过addRange来添加上面设置的光标。

3.8 关于contenteditable指令的控制器和指令所需服务

content editable指令的控制器和服务基本没有遇到太大的问题。那就说说代码思路。所有的服务都写在了realtime-markdown.service.js文件中，服务也不多，就三个，一个是对showdown库进行包装，用来实现单行的markdown文本转化为html，转化的方法使用的showdown库内的makeHtml。contenteditable直接使用了这个api，也为postLink函数提供了makeHtml方法。

第二个服务是socket，socket使用的是socket.io。代码比较简单，没有对socket进行封装，也是就直接返回了socket。因此在使用这个服务时，不会自动触发digest循环，需要手动调用scope.digest(),来触发digest循环，毕竟现在项目中没有涉及到需要双向数据绑定的地方，自动调用digest循环难免会对性能有一点点影响。在整个realtime-markdown编辑器中，所有和服务器的交互都是通过socket的来完成的。

第三个服务是CodeMirror。CodeMirror的作用就是替换编辑器中的某个元素为CodeMirror代码块，在CodeMirror代码块中进行输入，代码可以实时高亮。并且CodeMirror服务，我提供了一个setOption的API，可以对CodeMirror进行一些基本配置，配置项如下，当然配置发生变化，相应的外部引用文件也需要变化。

```
var options = {
        value: '',
        lineNumbers: true, // lineNumbers：是是否显示行号。
        mode: "javascript", // mode：是高亮代码的语言
        keyMap: "sublime", // keyMap：快捷键类型
        autoCloseBrackets: true, // 自动补全括号
        matchBrackets: true,
        showCursorWhenSelecting: true,
        theme: "base16-light",
        tabSize: 2 // tab键两个空格
};
```

四、realtime-markdown未来发展方向及展望

4.1 抛弃AngularJS,拥抱更多的框架或库。

毕竟一个编辑器，更加通用才会更加实用，只有放弃AngularJS才能够更加通用，而且由于个人原因，不太喜欢AnguLarJS中对DOM元素操作的API,(也就是类jQuery 的语法)，原因有二，一是现在原生的JS 对元素的选择，以及对DOM的操作已经不逊色于jQuery了。第二个原因就是选择恐惧症，当有两个API可以同时使用的时候，我不知道选择哪个了？比如jQuery中的addClass可以用classList.add来替换，replaceWith可以使用insertBefore来实现，原生JS选择器querySelect和querySelectAll来选择DOM元素的性能已经甩开jQuery选择器好远了。

4.2 realtime-markdown高度可配置，按需加载代码做一个轻量级的编辑器

realtime-markdown高度可配置主要是正对CodeMirror吧，可以选择需要高亮的语言，甚至可以直接在编辑器内更改代码块的主题，这不是很帅吗？按需加载代码的意思就是需要高亮什么语言，使用什么主题，再去下载相应的js或css代码，减少整个编辑器的臃肿。

4.3 编辑器内文本可以输入markdown格式和html格式

如果一个markdown编辑器只能够输入html格式文本，会不会太单调呢？如果可以选择性的输入markdown格式和html格式会不会更好呢?

4.4 编辑器配套的一些工作

如果要把realtime-markdown做好的话，希望别人贡献代码，同时也希望别人试用编辑器的话，GitHub开源是一定要的，让别人快速了解代码，至少需要一个简易文档的README，和一个包含基本文档和编辑器展示的官网。展望得有点远了，就这么多吧。