Markdown解析器设计与实现

# 词法分析过程

## 定义标识符

因为“ ”空格表示的特殊性，这里将“ ”空格用单词 Space 代替

|  |  |  |
| --- | --- | --- |
| 类型 | 正则表达式定义 | 说明 |
| TEXT | [^\n\r]\* | 除换行符和回车符之外的任意字符的闭包都是TEXT |
| \n | \n | 换行符当作切分单词的分隔符 |
| \r | \r | 回车符当作切分单词的分隔符 |
| &amp; | & | HTML实体字符 |
| &lt; | < | HTML实体字符 |
| ESC | \[\`\*\_{}[]()#+-.!] | 转义字符起始标志 |
| HEAD标题组√ | | |
| H1 | # | 标题一 |
| H2 | ## | 标题二 |
| H3 | ### | 标题三 |
| H4 | #### | 标题四 |
| H5 | ##### | 标题五 |
| H6 | ###### | 标题六 |
| DIVIDELINE分隔线组√ | | |
| DIVIDELINE | -(Space-){2,}Space- | 分隔线 |
| DIVIDELINE | \\*(Space\\*){2,}Space\* | 分隔线 |
| QUOTE引用组 √ | | |
| >> | (>Space) | 引用和嵌套引用 |
| LIST列表组√ | | |
| UL | -Space | 无序列表 |
| UL | +Space | 无序列表 |
| UL | \*Space | 无序列表 |
| OL | [0-9].Space | 有序列表 |
| STYLE强调组√ | | |
| EM | \*TEXT\* | 斜体 |
| EM | \_TEXT\_ | 斜体 |
| STRONG | \*\*TEXT\*\* | 加粗 |
| STRONG | \_\_TEXT\_\_ | 加粗 |
| EMSTRONG | \*\*\*TEXT\*\*\* | 斜体加粗 |
| EMSTRONG | \_\_\_TEXT\_\_\_ | 斜体加粗 |
| CODE代码组√ | | |
| ` | `TEXT` | 行级代码 |
| LAG | javascript | java | python | c++ | c | go | php | html | css | shell | 块级代码的语言标识 |
| ``` | ```LAG \n\r[TEXT\n\r]+\n\r``` | 块级代码 |
| LINK组√ | | |
| URL | /^(https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w \.-]\*)\*\/?$/ | 定义URL的正则表示 |
| LINK | <URL> | 可跳转URL |
| ALTLINK | [TEXT](URL) | 带文字说明的可跳转URL |
| IMG | ![TEXT](URL) | 引入图片资源 |
| < | < | LINK的左符号 |
| > | > | LINK的右符号 |
| [ | [ | ALTLINK的左符号 |
| ]( | ]( | ALTLINK的中符号 |
| ) | ) | ALTLINK的右符号 |
| ! | ! | IMG的开始标识 |

## 定义状态机

为了方便表述，定义 (char)[[opt]]->state[(char)] 表示识别char经过opt操作达到状态state , opt默认为add “加入临时token序列”，这是为了后续识别的方便。除此之外，opt=emit in [type] “提交type类型到token流”；opt=transfer to [tag]“转换为tag 的html实体。其中[(char)]可以有，也可以没有，表示回流，比如(other)->state(other)表示识别other字符后回流字符到state状态重新识别。

|  |  |  |
| --- | --- | --- |
| 状态名称 | 状态入度 | 状态出度 |
| Start | 1. ()-> | 1. (#)->Head1 2. (>)->Quote 3. ([0-9])->Olist1 4. (+)->Ulist1 5. (\_)->Em 6. (\*)->EmUlistDivideline 7. (-)->UlistDivideline 8. (`)->Code1 9. (<)->LinkLeft 10. ([)->AltlinkLeft 11. (!)->ImgBegin 12. (\n)[emit]->Start 13. (\r)[emit]->Start 14. (&)[emit]->Start 15. (\)->Esc 16. (other)->Text |
| Head1 | 1. Start->(#) | 1. (#)->Head2 2. (other)[emit]->Start(other) |
| Head2 | 1. Head1->(#) | 1. (#)->Head3 2. (other)[emit]->Start(other) |
| Head3 | 1. Head2->(#) | 1. (#)->Head4 2. (other)[emit]->Start(other) |
| Head4 | 1. Head3->(#) | 1. (#)->Head5 2. (other)[emit]->Start(other) |
| Head5 | 1. Head4->(#) | 1. (#)[emit to H6]->Start 2. (other)[emit to H5]->Start(other) |
| Quote | 1. Start->(>) | 1. (Space)[emit]->Start 2. (other)[emit to TEXT]->Start(other) |
| Olist1 | 1. Start->([0-9]) | 1. (.)->Olist2 2. (other)[emit]->Start(other) |
| Olist2 | 1. Olist1->(.) | 1. (Space)[emit]->Start 2. (other)[emit]->Start(other) |
| Ulist1 | 1. Start->(+) | 1. (Space)[emit]->Start 2. (other)[emit]->Start(other) |
| Em | 1. Start->(\_) | 1. (\_)->EmStrong 2. (other)[emit to EM]->Start(other) |
| EmStrong | 1. Em->(\_) | 1. (\_)[emit] ->Start 2. (other)[emit to Em]-> Start(other) |
| EmUlistDivideline | 1. Start->(\*) | 1. (\*)->EmUlistDivideline 2. (Space) -> EmUlistDivideline 3. (other)[emit by RegExp]->Start(other) |
| UlistDivideline | 1. Start->(-) | 1. (Space)-> UlistDivideline 2. (-)->UlistDivideline 3. (other)->[emit by RegExp]->Start(other) |
| Code1 | 1. Start->(`) | 1. (`)->Code2 2. (other)[emit to INLINECODE]->Start(other) |
| Code2 | 1. Code1->(`) | 1. (`)[emit to BLOCKCODE]->Start 2. (other)[emit] ->Start(other) |
| LinkLeft | 1. Start->(<) | 1. (TEXT)->TEXT 2. (other)[emit]->Start(other) |
| ImgBegin | 1. Start->(!) | 1. ([)->AltlinkLeft 2. (other)[emit]->Start(other) |
| AltlinkLeft | 1. Start->([) 2. ImgBegin->([) | 1. (TEXT)->Text 2. (other)[emit]->Start(other) |
| AltlinkCenter | 1. Text->(]) | 1. (()->Text 2. (other)[emit]->Start(other) |
| Esc | 1. Start->(\) | 1. ([\`\*\_{}[]()#+-.!])[emit]->Start 2. (other)[emit]->Start(other) |
| Text | 1. Start->(other) 2. Altlink->(TEXT) | 1. (TEXT)->Text 2. (>)[emit by RegExp]->Start 3. (])->AltLinkCenter 4. ())[emit by RegExp]->Start 5. (other)[emit]->Start(other) |

## 程序实现思路

程序分为两个部分：状态函数和主控程序

状态函数由当前状态根据字符输入到达的下一个状态的分支条件语句构成，比如：

let tokens = [];

let token = [];

state1 = function(char){

if(char == ‘\*){

token.push(char);

return state2;

}else{

emitToken(type , token.join(‘’));

token = [] ;

start(char);

}

}

function emitToken(type , token){

tokens.push({

type : type,

token : token

})

}

主控程序就是一个while循环，负责一个字符一个字符的向后读取原Markdown文章：

let state = start;

for(let char of input.split(‘’)){

state = state(char);

}

# Token流优化

为了方便和简化语法分析构建抽象语法树的过程，词法分析产出的Token流序列可以进行一定的简化操作，使得Token流序列简洁而不失结构和逻辑的完整性。

**简化规则：**

1. 多个TEXT文本类型可以合并为一个TEXT

2. \r\n可以合并为一个分隔不同行的标识 “ NEWLINE”

3. 转义字符ESC标识可以将字符与相邻的TEXT 合并，因为转义字符本质上就是文本

4. HTML实体字符的转义后实体字符可以与相邻的TEXT合并，因为实体字符的本质就是文本

5. 多个相邻的“DIVIDELINE”逻辑上表示多行分割线，但在文章中这样的分割线是没有意义的，因此规定多个相邻的“DIVIDELINE”合并为一个”DIVIDELINE”

6. 语义修复，“DIVIDELINE”后若有“\r\n”说明是正确语义的分割线；但是若” DIVIDELINE”后跟上“TEXT”在跟上”DIVIDELINE”说明语义错误，这里修复为斜体加粗体的语义，定义为“EMSTRONG”

7. 块级CODE需特殊处理，因为块级代码片极为复杂，要想支持语法高亮，需要对代码片单独进行，编程语言识别，关键字定位等，因此暂时先搁置此功能。遇到```标识时，将第一个TEXT标识为编程语言“LAG”,将前后“NEWLINE”之间的TEXT标识为“CODE”，之后单独封装函数对参数”LAG””CODE”做语法解析

# 语法分析过程

## 定义上下文无关文法

|  |  |
| --- | --- |
| 非终结符 | 释义 |
| ARTICLE | （开始节点）表示一篇文章 |
| SECTION | 表示一个章节 |
| PARAGRAPH | 表示一个段落 |
| DIV | 表示块级元素 |
| SPAN | 表示行级元素 |
| HEAD | 表示一个标题 |
| SENTENCE | 表示一个句子 |
| QUOTE | 表示一个引述 |
| LIST | 表示一个列表 |
| OLIST | 表示一个有序列表 |
| ULIST | 表示一个无序列表 |
| STYLE | 表示带样式的文本 |
| INLINECODE | 表示行级代码片 |
| BLOCKCODE | 表示块级代码片 |
| LINK | 表示与链接有关的文本 |
| IMG | 表示图片链接 |
| ALTLINK | 表示带有说明文字的URL链接 |
| SIMLINK | 表示URL链接 |

|  |  |
| --- | --- |
| 终结符 | 释义 |
| H1 H2 H3 H4 H5 H6 | 分别表示6种级别的标题 |
| TEXT | 表示纯文本 |
| URL | 表示链接文本 |
| EM | 表示斜体开始 |
| STRONG | 表示粗体开始 |
| EMSTRONG | 表示斜体和粗体的开始 |
| DIVIDELINE | 表示分隔线 |
| OL | 表示有序列表的开始 |
| UL | 表示无序列表的开始 |
| ` | 表示行级代码片的开始 |
| ``` | 表示块级代码片的开始 |
| LAG | 表示块级代码的编程语言 |
| CODE | 表示块级代码的代码片 |
| >> | 表示引用文本的开始 |
| < | 表示URL链接的开始 |
| > | 表示URL链接的结束 |
| ! | 表示图片链接的开始 |
| [ | 前置括号 |
| ]( | 中置括号 |
| ） | 后置括号 |
| NEWLINE | 分隔行的标识，表示一个回车换行 |

**BNF文法定义：**

1. ARTICLE -> SECTION{SECTION}

2. SECTION -> [HEAD]PARAGRAPH

3. PARAGRAPH -> DIV{DIV}

4. DIV -> SENTENCE | BLOCKCODE | LINK | QUOTE | LIST | DIVIDELINE

5. HEAD -> (H1|H2|H3|H4|H5|H6) SENTENCE

6. SENTENCE -> SPAN{SPAN}

7. SPAN -> STYLE | TEXT | INLINECODE

8. INLINECODE -> ` TEXT `

9. STYLE -> EM TEXT EM | STRONG TEXT STRONG | EMSTRONG TEXT EMSTRONG

10. BLOCKCODE -> ```LAG NEWLINE CODE NEWLINE ```

11. LINK -> IMG | ALTLINK | SIMLINK

12. IMG -> ![TEXT](URL)

13. ALTLINK -> [TEXT](URL)

14. SIMLINK -> <URL>

15. QUOTE -> >> TEXT

16. LIST -> OLIST | ULIST

17. OLIST -> OL TEXT

18. ULIST -> UL TEXT

# 定义抽象语法树，简化文法

1. ARTICLE -> {SECTION}

2. SECTION -> [HEAD]PARAGRAPH

3. PARAGRAPH -> {SENTENCE | BLOCKCODE | IMG | ALTLINK | SIMLINK | QUOTE | OLIST | ULIST | DIVIDELINE}

4. HEAD -> (H1|H2|H3|H4|H5|H6) SENTENCE NEWLINE

5. SENTENCE -> {STYLE | TEXT | INLINECODE }

6. INLINECODE -> ` TEXT `

7. STYLE -> EM TEXT EM | STRONG TEXT STRONG | EMSTRONG TEXT EMSTRONG

8. BLOCKCODE -> ```LAG NEWLINE{NEWLINE} CODE NEWLINE{NEWLINE} ```

9. IMG -> ![TEXT](URL)

10. ALTLINK -> [TEXT](URL)

11. SIMLINK -> <URL>

12. QUOTE -> >> SENTENCE

13. OLIST -> OL SENTENCE

14. ULIST -> UL SENTENCE