链滴

Markdown 解析原理详解和 Markdown AST 描述

作者：88250

原文链接：https://ld246.com/article/1587637426085

来源网站：链滴

概述

本文主要介绍 Markdown 引擎 Lute 的整体处理流程，并详细描述了 Markdown 抽象语法树结构。

编译原理

我们通过编译原理实现了 Lute ，大致步骤是预处理、词法分析、语法分析、代码生成这几个步骤。码结构方面我们分为两部分，解析和渲染。

```
// Markdown 将 markdown 文本字节数组处理为相应的 html 字节数组。name 参数仅用于标识文，比如可传入 id 或者标题，也可以传入 ""。
func (lute *Lute) Markdown(name string, markdown []byte) (html []byte) {
    tree := parse.Parse(name, markdown, lute.Options)
    renderer := render.NewHtmlRenderer(tree)
    html = renderer.Render()
    return
}
```

解析过程用于从 Markdown 原文构造抽象语法树。

```
// Parse 会将 markdown 原始文本字节数组解析为一颗语法树。
func Parse(name string, markdown []byte, options *Options) (tree *Tree) {
    tree = &Tree{Name: name, Context: &Context{Option: options}}
    tree.Context.Tree = tree
    tree.lexer = lex.NewLexer(markdown)
    tree.Root = &ast.Node{Type: ast.NodeDocument}
    tree.parseBlocks()
    tree.parseInlines()
    tree.lexer = nil
    return
}
```

渲染过程将遍历语法树生成 HTML 代码，本文略过。下面我们将着重介绍解析过程，从预处理阶段始。

预处理

预处理阶段主要是将输入的 Markdown 文本字节数组结尾添加换行符 \n，以方便后续解析可以统一行读取。

```
// NewLexer 创建一个词法分析器。
func NewLexer(input []byte) (ret *Lexer) {
    ret = &Lexer{}
    ret.input = input
    ret.length = len(input)
    if 0 < ret.length && ItemNewline != ret.input[ret.length-1] {
        // 以 \n 结尾预处理
        ret.input = append(ret.input, ItemNewline)
        ret.length++
    }
    return
}
```

词法分析

词法分析的目的是将 Markdown 文本转换为 token 数组。标准的编译原理中词法分析产生的 token 将带有如下这样一些属性：

- 类型（token type），比如标识符、操作符、数字、字符等
- 词素（lexeme），原始的文本字节数组
- 位置（pos），该 token 的第一个字节相对于整个文本字节数组的下标

Markdown 的词法分析进行了简化，仅返回词素作为 token，因为：

- Markdown 解析不需要类型信息，使用的标记符（比如 `#`、`*` 等）本身就是 token 类型和词素
- 大部分场景下的 Markdown 解析不需要实现 源码映射
- 提升性能

另外，编译原理教科书中是将词法分析和语法分析完全分开介绍的，即词法分析器产生 token 数组后为参数传入语法分析器，而实际工程上因为性能考虑，是在语法分析中调用词法分析来按需获得 token 数组，这样可以减少内存分配。

Markdown 词法分析的具体实现是按行进行处理的，每次处理后词法分析器会记录当前读取位置，便下次继续按行处理。

```
// Lexer 描述了词法分析器结构。
type Lexer struct {
    input  []byte // 输入的文本字节数组
    length int    // 输入的文本字节数组的长度
    offset int    // 当前读取字节位置
    width  int    // 最新一个字符的长度（字节数）
}

// NextLine 返回下一行。
func (l *Lexer) NextLine() (ret []byte) {
    if l.offset >= l.length {
        return
    }

    var b, nb byte
    i := l.offset
    for ; i < l.length; i += l.width {
        b = l.input[i]
        if ItemNewline == b {
            i++
            break
        } else if ItemCarriageReturn == b {
            // 处理 \r
            if i < l.length-1 {
                nb = l.input[i+1]
                if ItemNewline == nb {
                    l.input = append(l.input[:i], l.input[i+1:]...) // 移除 \r，依靠下一个的 \n 切行
                    l.length--                                     // 重新计算总长
                }
            }
```

```
            i++
            break
        } else if '\u0000' == b {
            // 将 \u0000 替换为 \uFFFD
            l.input = append(l.input, 0, 0)
            copy(l.input[i+2:], l.input[i:])
            // \uFFFD 的 UTF-8 编码为 \xEF\xBF\xBD 共三个字节
            l.input[i], l.input[i+1], l.input[i+2] = '\xEF', '\xBF', '\xBD'
            l.length += 2 // 重新计算总长
            l.width = 3
            continue
        }

        if utf8.RuneSelf <= b { // 说明占用多个字节
            _, l.width = utf8.DecodeRune(l.input[i:])
        } else {
            l.width = 1
        }
    }
    ret = l.input[l.offset:i]
    l.offset = i
    return
}
```

语法分析

CommonMark 规范中介绍了一种解析算法，分为两个阶段：

1. 构造所有块级节点，包括标题、块引用、代码块、分隔线、列表、段落等，还需要构造好链接引用义映射表
2. 遍历每个块级节点，构造行级节点，包括文本、链接、强调、加粗等，链接的处理可能会需要查找骤 1 中构造好的链接引用定义映射表

关于 CommonMark 规范的一些实现细节可参考我之前的笔记（CommonMark 规范要点解读、Lute 实现后记），这里就不展开了，如果感兴趣欢迎跟帖讨论。

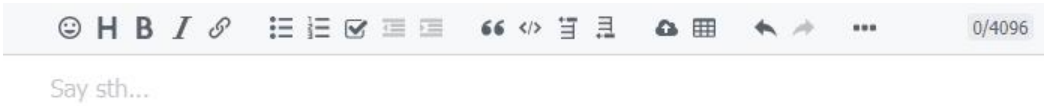
抽象语法树

Markdown 抽象语法树是由节点构成的树，从包含关系来说节点可以分为四类：

1. 根节点，可以包含所有其他任意节点
2. 块级容器节点，可以包含非根节点的其他任意节点，比如列表项包含段落
3. 块级节点，可以包含行级节点，比如段落包含强调
4. 行级节点，可以包含行级节点，比如强调包含文本

我们在实现 Lute 时做了“最细粒度”的节点结构，比如对于超链接 [foo](bar) 形成的节点结构包含左方括号 [、链接文本 foo、右方括号]、左圆括号 (、链接地址 bar 和右圆括号)。这样做的优点是便处理细致的节点操作，缺点是性能稍差，因为需要构造和遍历更多的节点。

如果你想看到较粗粒度的语法树，可以通过 Vditor Markdown 编辑器的开发者工具来查看，请到此行测试（点击编辑器工具栏上的“开发者工具”按钮就可以看到根据输入进行实时渲染的语法树了）。

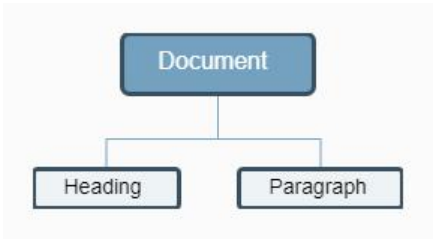


下面我们按 Lute 源码中的节点类型常量顺序来逐一描述。

根 Document

语法树的根节点。

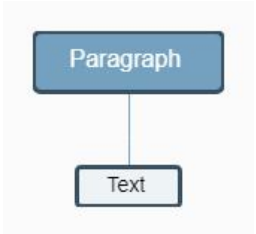
```
# foo

bar
```



段落 Paragraph

段落属于块级节点，可包含行级节点。

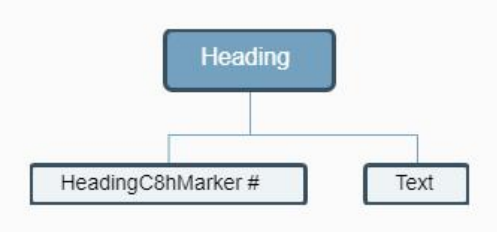
```
foo
```



标题 Heading

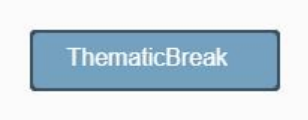
标题属于块级节点，可包含行级节点。

```
# foo
```



分隔线 ThematicBreak

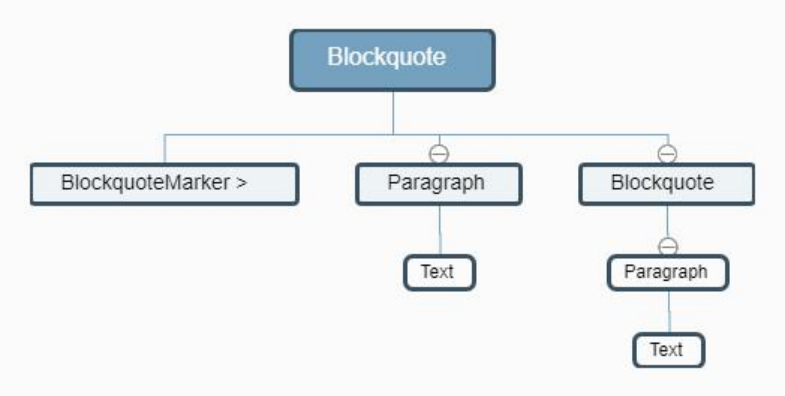
分隔线属于块级节点，没有子节点。



块引用 Blockquote

块引用属于块级容器节点，可包含任意（非根）节点。

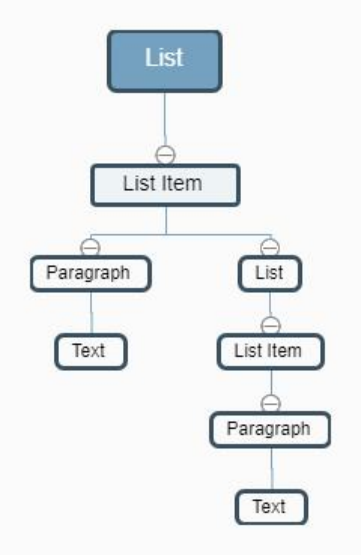
```
> foo
>
>> bar
```



列表 List

列表属于块级容器节点，但只能包含列表项节点。

```
* foo
  * bar
```



HTML 块 HTML Block

HTML 块属于块级节点，不包含其他子节点。

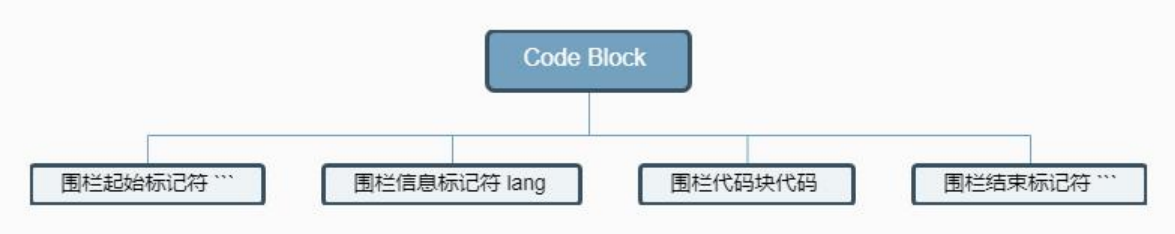
行级 HTML Inline HTML

行级 HTML 属于行级节点，不包含其他子节点。

代码块 Code Block

代码块属于块级节点，围栏代码块包含一些标记节点和代码块代码节点。

```lang
foo
```



文本 Text

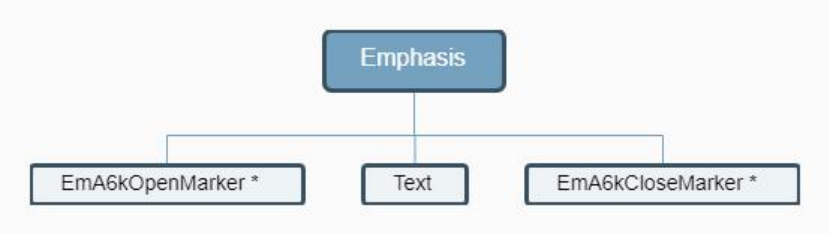
文本属于行级节点，不包含其他子节点。

强调 Emphasis

强调属于行级节点，可包含文本、加粗、链接等行级子节点。强调标识符有两种，* 和 _。

foo

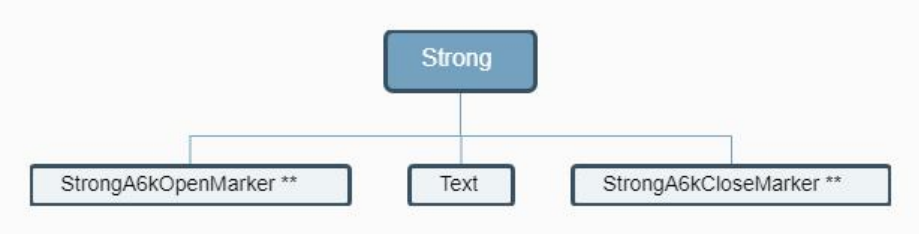
原文链接：Markdown 解析原理详解和 Markdown AST 描述



加粗 Strong

加粗属于行级节点，可包含文本、强调、链接等行级子节点。加粗标识符有两种，** 和 __。

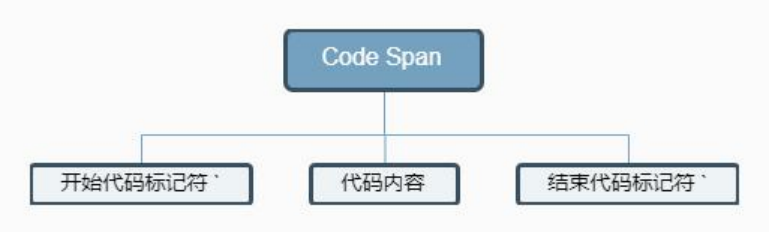
foo



代码 Code Span

代码属于行级节点，包含开始代码标记符 `、代码内容和结
代码标记符 `。

`code`



硬换行 Hard Break

硬换行属于行级节点，不包含其他子节点。

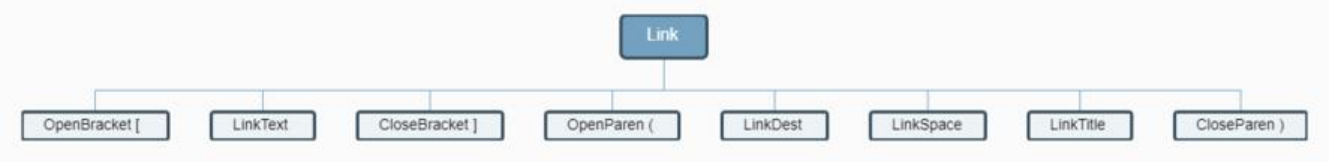
软换行 Soft Break

软换行属于行级节点，不包含其他子节点。

链接 Link

链接属于行级节点，由链接相关标识符、链接文本等构成，可包含图片、强调和加粗等行级子节点。

[foo](bar "baz")



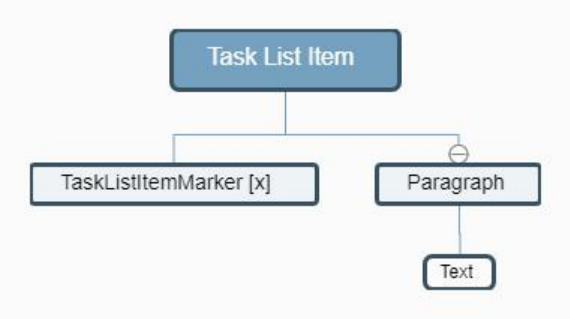
原文链接：Markdown 解析原理详解和 Markdown AST 描述

以上是 Lute 实现 CommonMark 规范时用到的全部语法树节点结构，下面继续介绍 GFM 规范扩展一些节点类型。

任务列表项 Task List Item

任务列表项属于块级容器节点，相比列表项节点多了一个任务列表项标记符。

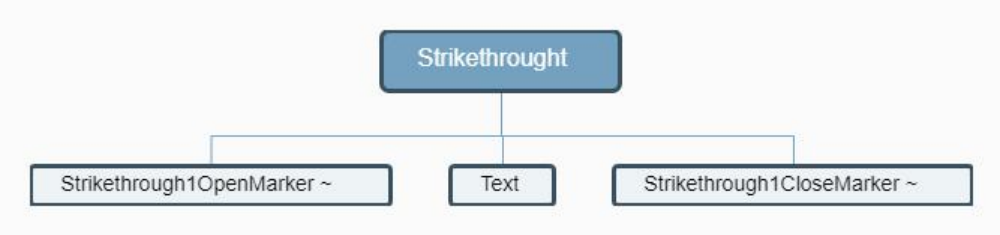
```
* [x] foo
```



删除线 Strikethrough

删除线属于行级节点，可包含文本、强调、加粗、链接等行级子节点。删除线标识符有两种，~ 和 ~~。

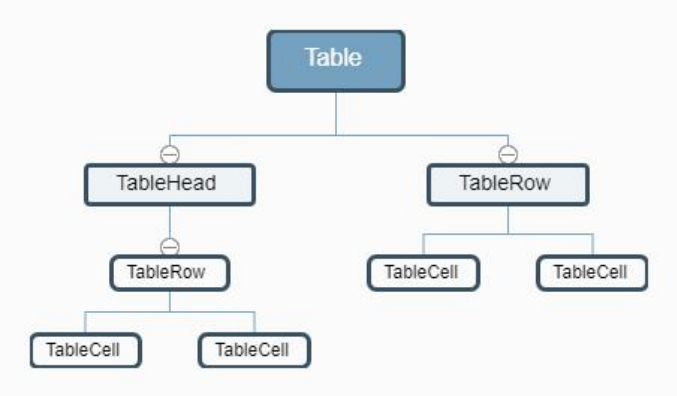
```
~foo~
```



表格 Table

表格是块级节点，可包含其他行级子节点。

```
| foo | bar |
| --- | --- |
| c1  | c2  |
```



除了以上介绍的节点结构，还有一些扩展语法比如 Emoji、数学公式、脚注和 ToC 等，这里就不展

描述了。

总结一下 Markdown AST：

- 语法树节点结构 CommonMark 规范并未定义，解析器实现时可以自由发挥
- 细粒度节点表达更丰富，但同时也会有一定性能消耗

关于 Markdown 源码映射

什么是 Source Map?

Source Map 即 Markdown 源码和 HTML 目标代码之间的字符关联信息。要实现双向映射 AST 上须要有结构来存储映射关系：

- 源码中每个字符都可以在 AST 上找到对应节点
- 从 AST 节点也可以在源码中找到对应字符位置

Source Map 有什么用?

- 为编辑器和预览视图联动提供支持
- 为 Markdown 语法高亮提供基础

本文涉及的开源项目

- CommonMark Spec
- Lute 一款对中文语境优化的 Markdown 引擎，支持 Go 和 JavaScript
- Vditor 一款浏览器端的 Markdown 编辑器，支持所见即所得、即时渲染（类似 Typora）和分屏览模式