链滴

Lute 实现后记

作者：88250

原文链接：https://ld246.com/article/1567062979327

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

CommonMark 规范实现难点

Markdown 解析没有“报错”一说：无论是词法分析还是语法分析阶段，都不可能出现 error 退出。句话说，对于编程语言而已，其在设计时就是准结构化的，甚至为了解析方便而特设语法（比如 golang 数据类型后置、模板中的表达式使用前缀表达式等），而 CM 则是为兼容各种写法、将各种写法规化而设计的，所以规则非常多，目的是为了兼容而不是报错。

整体解析方面比较难的是计算缩进宽度。缩进在 Markdown 中主要用于缩进代码块、列表项对齐。为列表项作为块容器是可以容纳任何元素的（包括块级元素和行级元素），所以当列表项出现子块时进就要根据列表的定义来计算。比如要考虑列表标记宽度，标记后到第一个非空字符的空格数等。最烦的是块引用嵌套列表的场景，因为块引用也是一种块容器。

除了缩进，还有个比较难的是“延续”判断。比如换行以后需要判断是否需要延续之前的行所在的节。不同节点类型有不同的打断规则，最麻烦的也还是列表项和块引用，因为它们是块容器，可能出现状态有点多。

最后还有个难点是判断列表是紧凑模式还是松散模式。因为列表是块级容器，所以需要考虑嵌套时列模式的传播。官方参考实现的算法是在下降解析过程中把父节点方向上都赋值一遍，这样应该是代价小的。

总之，除非你已经精通 CM 规范，否则强烈建议不要自己设计解析算法。要独立实现 CM 规范真的难。CM 规范文末的解析算法应该已经是最优算法了，可以参考官方的参考实现项目进行实现。当然如果你有更好的算法，请不吝赐教。

换行处理

回车和换行

块级解析算法是基于行的，每次读取一行进行处理，断行的依据是 \r\n 和 \n，因为规范中定义了行束符就是这两者。Lute 是在词法分析阶段做了预处理，将 \r\n 替换为 \n，后续统一使用 \n 进行断。

软硬换行

规范定义了两种换行节点：软换行 \n 和硬换行
，HTML 渲染时这两者会有明显的区别。规范义的硬换行规则是行结尾用 \ 或者结尾两个空格，这个规则我相信很少人知道。

规范定义硬换行需要结尾用\
或者结尾用两个空格
这样渲染结果才会有
 以达到换行的目的

鉴于大部分 Markdown 用户习惯了软换行自动转硬换行，所以建议实现时考虑默认开启这个支持：户只需一个换行就可以渲染出
。

开启软换行自动转硬换行后结尾不需要标记
就可以达到渲染
 的效果

渲染时的换行

规范中对渲染结果 HTML 中的 \n 是有明确要求的，有的情况下标签结尾需要带换行，有点情况下不要。比如下面渲染结果中的 <li> 标签：

```
<ul>
<li>
<p>foo</p>
<p>bar</p>
<ul>
<li>baz</li>
</ul>
</li>
</ul>
```

这类换行不是用户输入的，所以不会解析生成软换行节点，只能靠渲染时进行处理。总体的处理逻辑某些标签后需要跟换行，但是换行不能重复。可通过记录最后一个输出字符，如果是换行的话则下次再重复输出。

```
// newline 会在最新内容不是换行符 \n 时输出一个换行符。
func (r *Renderer) newline() {
    if itemNewline != r.lastOut {
        r.writer.WriteByte(itemNewline)
        r.lastOut = itemNewline
    }
}
```

性能优化

能不构造对象就不要构造对象

比如词法分析时不要定义 token 对象，而是直接用 byte 类型。这样可以减少构造 token 对象，但需在某些迭代 token 的地方处理字符解码（utf8.decodeRune），从而跳过 size 个字节继续迭代。

另外，还有个优化点是减少嵌套结构体的构造。比如出现最多的节点是 Text ，整颗语法树上可能 95% 都是文本节点。此时如果文本节点嵌套了基础节点 BaseNode 的话（嵌套的目的是为了复用 BaseNode 的字段和接口实现），构造文本节点就需要创建两次对象，将会重复调用 runtime.newobject 降性能。优化方案就是去掉嵌套的 BaseNode，将 BaseNode 的结构在 Text 中再做一次。

总的来说，不构造对象可以换来巨大的性能提升，但代价就是降低代码可读性，并且看上去会显得有僵硬。

减少 slice 增长，使用 make 提前分配好大小

靠自动增长会增加内存分配次数，这是 slice、map 的常规优化手段。

快速的 []byte~string 转换优化

直接上代码吧（items 是 []byte 类型）：

```
// fromItems 快速转换 items 为 string。
func fromItems(items items) string {
    return *(*string)(unsafe.Pointer(&items))
}

// toItems 快速转换 str 为 items。
func toItems(str string) items {
```

```
    x := (*[2]uintptr)(unsafe.Pointer(&str))
    h := [3]uintptr{x[0], x[1], x[1]}
    return *(*items)(unsafe.Pointer(&h))
}
```

需要注意的是转换后如果要进行写操作，则转换前要进行一次动态构造：

```
// 动态构造一次，因为后续有可能会对字节数组进行赋值
// 不构造的话会报错 fatal error: fault
builder := strings.Builder{}
builder.Write(input)
input = items(builder.String())
// 后续会对 input 中的 byte 进行赋值
// 并且可以放心地使用 toItems 和 fromItems 互转
```

并行解析

在 CM 规范的优先级一节中提到了块级元素解析要按行顺序解析，但行级元素可并行解析，因为块级点下的行级解析不会相互影响。

> Note that the first step requires processing lines in sequence, but the second can be parallelized, since the inline parsing of one block element does not affect the inline parsing of any other.

Lute 基于该提示进行了行级并行解析实现：

```
// 遍历处理子节点，通过并行处理提升性能
cwg := &sync.WaitGroup{}
for child := node.FirstChild(); nil != child; child = child.Next() {
    cwg.Add(1)
    go t.walkParseInline(child, cwg)
}
cwg.Wait()
```

基准测试结果也印证了并行解析在超过 2 核的情况下性能提升很显著：

```
BenchmarkLute-2   300   5132936 ns/op   2994687 B/op   21333 allocs/op
BenchmarkLute-4   300   4351804 ns/op   2995676 B/op   21345 allocs/op
BenchmarkLute-8   300   4168959 ns/op   2996725 B/op   21355 allocs/op
```

其他选手

在 CM 规范实现这条赛道上的选手并不多，甚至可以说是非常少。我整理了下 GitHub 上相关的项目目前实现了 CM 规范的项目有这几个：

项目	语言	Stars
commonmark/cmark（RI）	C	K
atlassian/commonmark-java	Java	K
vsch/flexmark-java	Java	1K
commonmark/commonmark.js（RI）	JavaScript	

K

markdown-it/markdown-it	JavaScript	.6K
Knagis/CommonMark.NET	C#	K
lunet-io/markdig	C#	1.6K
readthedocs/commonmark.py	Python	.2K
thephpleague/commonmark	PHP	.1K
kivikakk/comrak	Rust	0.2K
yuin/goldmark	Go	0.2K
b3log/lute	Go	

更多实现可参考这里。

后记

前车之鉴

一开始走了一段弯路，我打算“一遍”就处理完（尽量减少回溯和遍历），并沿用 Go 模板引擎的解策略，递归下降+词法分析和语法分析并行处理，输入文本一遍过完后就得到语法树。后来实在是编下去了，因为规范细节的情况太繁琐，有的情况需要向前读取很多字符才能确定当前节点（或者说当字符可能会改变已经解析的节点），所以用递归下降分析法是实现不了的，只能考虑通过多次迭代进解析。

后来我还是老老实实按照规范附录部分介绍的解析策略进行实现。总之，心太急、忽略别人的经验之、自作聪明容易自讨苦吃。

参考实现

自编解析策略翻车后，我基本有时间就看规范，反复看以确认细节并加深理解，特别是附录的解析策部分。

块级元素、行级元素解析的思路还是比较好理解的，粗枝大叶的适合我这样的糙老爷们。但附录最后的调链接解析算法开始看的时候基本就是完全靠猜了，因为这部分讲得太抽象了。这期间我试图去看了参考实现 cmark（C 语言写的），帮助不大，毕竟数据结构差别有点大。后来又看了 JavaScript 的考实现 commonmark.js，这个项目核心解析代码只有几百行，结合规范文档看会有很大帮助的。

编程带来的快乐

Markdown 处理整个过程其实就是大量的 if else 和 for 构成的，万物皆为 CRUD。不过和业务的 CRD 相比还是稍有区别：

1. 业务上一个请求一条线理下来，情况不会太多，实在不行还能 try-catch 兜底
2. 业务在非功能性需求方面有较多成熟的解决方案，比如各类分布式中间件
3. 业务处理性能瓶颈大多出现在数据库和网络 IO 上

同是 CRUD，但撸一个 Markdown 引擎带来的快感远远超过撸业务代码。对我来说，业务代码要撸同样感觉的话除非这个业务系统是给上百万人日常使用的。

也许我是太久没有写这样的代码了，几乎忘记了编程带来的快乐。希望大家能经常写写这样的代码，有实际需求可以刷下类似力扣的编程题，找回初学编程时的快乐。

潜意识和睡眠

我感觉我的潜意识应该是拒绝实现 Lute 的，因为拒绝产生的动能居然打败了睡眠障碍。之前睡眠不么好，自从开始写 Lute，躺下后一想解析策略和性能优化基本就睡着了，结果就是算法没想出来但治了失眠。如果你也失眠，不妨试着写个 CommonMark 实现，亲测有效！