How Hugo handles your pages (technically speaking)
Lexer, Parser and Goldmark

Duccio Marco Gasparri @dgasparri

https://hugoconf.io/

How Hugo handles your pages (technically speaking)
Lexer, Parser and Goldmark

Duccio Marco Gasparri

https://gasparri.org

@dgasparri

https://github.com/dgasparri

https://hugoconf.io/

Duccio Marco Gasparri

https://gasparri.org

@dgasparri

https://github.com/dgasparri

Why reverse engineering Hugo?

- I’m no Hugo expert
  - I’ve been using Hugo for a couple of months now
  - I had to understand how Hugo was processing Markdown pages (At the low level)

- I tried the direct approach - open the source files and read the code
  - but it was too obfuscated
  - some functions are injected or chosen at run time
  - No clear distinction among different components

- So I stepped back and did some reverse engineering

How was Hugo RE?

- the old-fashioned way
  - fmt.Println(“We are in function xy”)

- Stack trace dump
  - Regex to find functions in the code, and trace dump to a file for later review

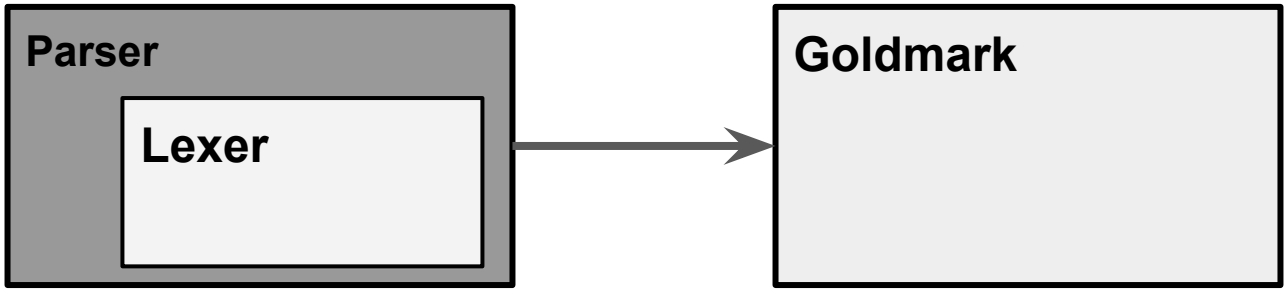
Duccio Gasparri @dgasparri

How is Hugo inside?

- Ignoring the file structure, templates, and so on (for another time)

- I focused on what was of interest to me, specifically on the way Hugo parses and handles Markdown files
  - Relevant for Issue #3606 on GitHub “Support wiki-style internal page links” https://github.com/gohugoio/hugo/issues/3606

Three main blocks for processing .md files



Those three blocks combined transform a Markdown file into a html file

Goldmark

- It is a native Markdown processor written by Yusuke Inuzuka (independent from Hugo)
  - https://github.com/yuin/goldmark
- Fully compliant with CommonMark 0.30b

It can process main md items:

- Headings #
- Italic
- Bold
- Lists
- Links

But it cannot process other items:

- Hugo templates
- Shortcodes
- Links (rewriting, since it has no idea of the internal structure of the website)

Parser

First step of the whole process

- Two jobs:
  1. it invokes the Lexer
  2. Receives the page content as “digested” from the Lexer (chunks of the original file, each chunk containing either Front Matter, Shortcodes, Links..) and handles it

Specifically, it handles:

- Front Matter
- Summary Dividers
- Shortcodes
- Emojis

  (but only if returned by the Lexer)

Plain text or standard Markdown are not processed (left to Goldmark)

Lexer

- The function of the Lexer is:
  1. to find hooks in the page (for example, short code opening “{{<”)
  2. to divide the page in chunks according to the content function (and send them to the parser)

- Among others, it separates:
  - Front Matter
  - Shortcodes
  - Emojis

Parser and Lexer - source of uncertainty

An example of something that was obscure at the beginning:

- The **Lexer** finds hooks in the file content for shortcodes' ***openings***
- The **Parser** finds hooks in the file content for the shortcodes' ***closings***

But…

- The Lexer finds the hooks for the Front Matter *opening* and *closing*
- The Parser doesn't search for front matter in the file content, it just digests what the lexer has found

Once you know it, it's not anymore confusing… once you know it (and they had reasons for doing it like that)

To recap

- The **Lexer** divides the page in chunks of Front Matter, shortcodes, emojis, plain Markdown text
- The **Parser** takes the list of chunks from the Lexer and processes them (but not the plain Markdown, which is left to Goldmark)
- **Goldmark** processes the remaining Markdown in the page (headings, bold, italic, etc.)

Let’s now look at the low level

Parser - low level

- Package **"github.com/gohugoio/hugo/parser/pageparser"**
- Called (for each page) by the function:
  - hugolib/content__map_page.go:
    - (m *pageMap).newPageFromContentNode():149

```
149     parseResult, err := pageparser.Parse(
150         r,
151         pageparser.Config{EnableEmoji: s.siteCfg.enableEmoji},
152     )
```


Duccio Gasparri @dgasparri

Parser - low level (cont)

- The Parse(...) function (parser/pageparser/pageparser.go:39) calls the parseSection() function **passing the lexIntroSection function** as an argument
  - This is relevant! The Lexer is based on functions chosen at runtime according to the content presented, this gives its entry point
- The order is:
  - Parse()
  - parseSection()
    - Read the io.Reader content
  - parseBytes
    - Instantiates the Lexer and runs it
    - Returns the Lexer with all the content

```
110 func parseBytes(b []byte, cfg Config, start stateFunc) (Result, error) {
111     lexer := newPageLexer(b, start, cfg)
112     lexer.run()
113     return lexer, nil
114 }
```

Parser - low level (cont)

- … after some passages …
- The lexer result is returned to the **hugolib/page.go:mapContentForResult()** function that iterates through the pieces of the page (as divided by the Lexer) and handles each piece accordingly to its type. Cases are:
  - Case it.IsFrontMatter()
  - Case it.Type == pageparser.TypeLeadSummaryDivider
  - Case it.IsLeftShortcodeDelim()
  - case it.Type == pageparser.TypeEmoji
  - case it.IsError()
  - Default (plain text to write to output file)

Lexer - low level

- Called by: **parser/pageparser.go:parseBytes()** (entry point)
  - pagelexer.go:newPageLexer()
  - lexer.run()

- **pagelexer.go:newPageLexer()**
  - instantiates a new pageLexer
  - initializes it with the initial “state” (i.e., a Lexer function for parsing the given content, that at this stage is **lexIntroSection())**
  - creates the **Section Handlers**, that is, functions and codes for handling shortcodes, page summary and emoji

Lexer - low level (cont)

- **pagelexer.go:run()**
  - Runs the first lexer function lexIntroSection()
  - Receives - from the lexer function - the next lexer function and runs it

```
92  // main loop
93  func (l *pageLexer) run() *pageLexer {
94      for l.state = l.stateStart; l.state != nil; {
95          l.state = l.state(l)
96      }
97      return l
98  }
```

Lexer - low level (cont)

- This setup is not trivial
- The lexer function not only has to parse its part of the content (for example, front matter setting) but it must also decide which lexer function should run next
- It must:
  - be aware of the available lexer functions
  - make a decision on what to run next
- No separation of concerns
- Main lexer functions:
  - lexIntroSection() - Front matter
    - -> lexMainSection() always
  - lexMainSection()
    - -> lex section handlers
    - return lexDone()

Goldmark - low level

- It's an independent library, so it is just imported (with its extensions) and instantiated
- Instantiated by **hugolib/hugo_sites.go:applyDeps()** during config loading through:
  - helpers/content.go:NewContentSpec()
  - markup/goldmark/convert.go:(provide p struct)New()
  - markup/goldmark/convert.go:newMarkdown()
    - Return goldmarkConverter struct with goldmark:Markdown interface (Convert() function implemented)

```
93      } else {
94          // Use Goldmark's sanitizer
95          p := converterProvider.Get("goldmark")
96          conv, err := p.New(converter.DocumentContext{})
97          if err != nil {
98              return nil, err
99          }
100         spec.anchorNameSanitizer = conv.(converter.AnchorNameSanitizer)
101     }
```

Goldmark - low level

- Called by hugolib/content_map_page.go/**newPageFromContentNode()** that calls:
- -> hugolib/page__per_output.go:newPageContentOutput()
- -> hugolib/page__per_output.go:renderContent()
- -> hugolib/page__per_output.go:renderContentWithConverter()
- -> Convert()

Duccio Gasparri @dgasparri

To sum up

- The function **newPageFromContentNode**() is responsible of calling:
  - the Parser->Lexer
  - and then, the Goldmark processor
- The **Lexer** splits the page in chunks based on the content type of what it is reading (front matter, shortcodes, markup/text)
- The **Parser** takes each chunk that is not markup/text and processes it
- In the end, the content is passed to the **Goldmark** processor for final markup parsing

You can find my notes on Hugo's code at

https://gasparri.org/hugo-techical/