

Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной
математики

Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу «Информационный поиск»

Студент: В. М. Филиппов
Преподаватель: А. А. Кухтичев
Группа: М8О-410Б
Дата:
Оценка:
Подпись:

Москва, 2025

Лабораторная работа №1 «Добыча корпуса документов»

Необходимо написать парсер на любом языке программирования.

Написать поисковый робот — компоненты обкачки документов, используя любой язык программирования; Единственным аргументом поисковому роботу подаётся путь до yaml-конфига, содержащий: Данные для базы данные в секции db; Данные для робота в секции logic: задержка между обкачкой страницы; Любые другие данные, необходимые для реализации логики поискового робота. Сохранять в базе данных (например, MongoDB) документы со следующими полями: url, нормализованный; «сырой» html-текст документа; название источника; Дата обкачки документа в формате Unix time stamp. Поисковый робот можно остановить в любой момент и при повторном запуске робот должен начать с того документа, с которого он остановился; Периодически он должен уметь переобкачивать документы, которые уже есть в базе, но только в том случае, если они изменились.

1 Описание

Для реализации поискового робота был выбран язык программирования Go, так как его средства конкурентного программирования позволяют эффективно распараллелить процесс сбора данных. В качестве хранилища документов используется база данных MongoDB.

Сбор данных осуществляется с помощью библиотеки `gocolly`. Она позволяет гибко настраивать параметры краулера: устанавливать задержку между запросами (включая случайное смещение (Jitter) для эмуляции поведения реального пользователя), регулировать количество одновременных потоков и настраивать `User-Agent`.

1 Алгоритм работы

Глобально логика работы робота выглядит следующим образом:

1. Загрузка конфигурации из аргументов командной строки, инициализация и настройка экземпляра краулера.
2. Проверка наличия устаревших документов в базе (требующих повторного сбора). Если таковые есть — они добавляются в очередь.
3. Запуск основного цикла поиска и загрузки: получение страницы, валидация (проверка на «мусорный» контент или служебные разделы), добавление в очередь.

Процесс обработки различается в зависимости от источника:

Wikipedia. Если URL содержит домен `wikipedia.org`, проверяется тип страницы. Наличие подстроки `/Category:` в пути указывает на то, что это категория. В противном случае страница считается статьей.

- **Для категорий:** производится поиск всех ссылок по селекторам `#mw-pages a[href]` и `#mw-subcategories a[href]`, найденные ссылки добавляются в очередь.
- **Для статей:** страница сохраняется в БД. Перед сохранением производится проверка на дубликаты путем сравнения хэш-суммы текущей версии статьи с версией, уже находящейся в базе.

The Guardian. Если URL содержит домен `theguardian.com`, тип страницы определяется по структуре ссылки:

- **Статьи:** как правило, содержат в пути дату публикации (год, месяц, день), например: `/2023/oct/25`.
- **Категории:** имеют малую вложенность, например: `/sport/tennis`.

Если страница не удовлетворяет ни одному из условий, она помечается как имеющая неизвестный тип (вероятно, служебная) и игнорируется.

2 База данных

В базе данных я храню следующие поля.

- ID документа
- URL страницы
- Нормализованный URL
- Источник страницы (`wikipedia/theguardian`)
- HTML-код страницы
- Заголовок страницы
- Хэш HTML-кода
- Время первой обкачки
- Время последней обкачки
- Размер HTML-кода страницы в байтах

Итого, при помощи этого робота, у меня получилось загрузить 305135 документов, из которых 254904 - это статьи с Википедии, а 50251 с The Guardian. База данных получилась размером в 10,78920657 ГБ. Не пропорциональность в загруженных документах объясняется тем, что Википедия даёт очень много ссылок с одной категориальной страницы, тем самым забивая очередь на обработку.

2 Исходный код

```
1 package main
2
3 import (
4     "flag"
5     "fmt"
6     "web_spider/internal/app"
7     "web_spider/internal/config"
8 )
9
10 func generateArchiveURLs(base string, fin_page int) []string {
11     var urls []string
12     for page := 0; page < fin_page; page++ {
13         urls = append(urls, fmt.Sprintf("%s?page=%d", base, page))
14     }
15     return urls
16 }
17
18 func main() {
19     configPath := flag.String("config", "config.yaml", "Path to configuration file")
20
21     flag.Parse()
22
23     cfg, err := config.LoadConfig(*configPath)
24
25     if err != nil {
26         panic(err)
27     }
28
29     crawler := app.NewNewsCrawler(cfg)
30
31     startURLGuardian := "https://www.theguardian.com/sport"
32     startURLWiki := "https://en.wikipedia.org/wiki/Category:Sports"
33     urls := generateArchiveURLs(startURLGuardian, 100);
34     urls = append(urls, startURLGuardian)
35     urls = append(urls, startURLWiki)
36
37
38     crawler.Start(urls, 1000000000)
39 }

```

```
1 package models
2
3 type ExtractedArticle struct {
4     Title string
5     Text string
6     HTML string
7     Excerpt string

```

```

8 | }
9 |
10 | type Document struct {
11 |     ID string 'bson: "_id" '
12 |     URL string 'bson: "url" '
13 |     NormalizedURL string 'bson: "normalized_url" '
14 |     Source string 'bson: "source" '
15 |     HTMLContent string 'bson: "html_content" '
16 |     Title string 'bson: "title" '
17 |     ContentHash string 'bson: "content_hash" '
18 |     FirstScraped int64 'bson: "first_scraped" '
19 |     LastScraped int64 'bson: "last_scraped" '
20 |     ContentLength int 'bson: "content_length" '
21 | }

1 | package config
2 |
3 | import (
4 |     "os"
5 |
6 |     "gopkg.in/yaml.v2"
7 | )
8 |
9 | type SourceConfig struct {
10 |     Name string 'yaml: "name" '
11 |     BaseURLs []string 'yaml: "base_urls" '
12 |     StartURLs []string 'yaml: "start_urls" '
13 |     FollowPatterns []string 'yaml: "follow_patterns" '
14 |     ExcludePatterns []string 'yaml: "exclude_patterns" '
15 |     MaxPages int 'yaml: "max_pages" '
16 |     Priority int 'yaml: "priority" '
17 | }
18 |
19 | type DBConfig struct {
20 |     Connection string 'yaml: "connection" '
21 |     Database string 'yaml: "database" '
22 |     Collections struct {
23 |         Documents string 'yaml: "documents" '
24 |         SpiderState string 'yaml: "spider_state" '
25 |         SpiderHistory string 'yaml: "spider_history" '
26 |     } 'yaml: "collections" '
27 | }
28 |
29 | type LogicConfig struct {
30 |     DelayMS int 'yaml: "delay_ms" '
31 |     TimeoutSec int 'yaml: "timeout_sec" '
32 |     MaxRetries int 'yaml: "max_retries" '
33 |     MaxDepth int 'yaml: "max_depth" '
34 |     RecrawlThresholdHours int 'yaml: "recrawl_threshold_hours" '
35 |     MaxConcurrentWorkers int 'yaml: "max_concurrent_workers" '

```

```

36 }
37
38 type SpiderConfig struct {
39     DB DBConfig `yaml:"db"`
40     Logic LogicConfig `yaml:"logic"`
41     Sources map[string]SourceConfig `yaml:"sources"`
42 }
43
44 func LoadConfig(path string) (*SpiderConfig, error) {
45     data, err := os.ReadFile(path)
46     if err != nil {
47         return nil, err
48     }
49     var cfg SpiderConfig
50     if err := yaml.Unmarshal(data, &cfg); err != nil {
51         return nil, err
52     }
53     return &cfg, nil
54 }
55
56 package utils
57
58 import (
59     "crypto/md5"
60     "fmt"
61     "net/url"
62     "regexp"
63     "strings"
64 )
65
66 func NormalizeURL(urlStr string) string {
67     parsed, err := url.Parse(urlStr)
68     if err != nil {
69         return urlStr
70     }
71
72     parsed.Fragment = ""
73
74     parsed.Host = strings.TrimPrefix(parsed.Host, "www.")
75
76     if parsed.Scheme == "" {
77         parsed.Scheme = "https"
78     }
79
80     return parsed.String()
81 }
82
83 func ExtractLinksFromHTML(htmlContent, baseURL string) []string {
84     var links []string

```

```

31 re := regexp.MustCompile(`href=["']([~"']+)["']`)
32 matches := re.FindAllStringSubmatch(htmlContent, -1)
33
34 for _, match := range matches {
35     if len(match) > 1 {
36         link := match[1]
37         if strings.HasPrefix(link, "http") {
38             links = append(links, link)
39         } else if strings.HasPrefix(link, "/") {
40             parsed, _ := url.Parse(baseURL)
41             absURL := parsed.Scheme + "://" + parsed.Host + link
42             links = append(links, absURL)
43         }
44     }
45 }
46 return links
47 }
48
49 func ComputeContentHash(content string) string {
50     hash := md5.Sum([]byte(content))
51     return fmt.Sprintf("%x", hash)
52 }
53
54 func URLShouldBeFollowed(urlStr string, followPatterns, excludePatterns []string) bool
55 {
56     for _, pattern := range excludePatterns {
57         if URLMatchesPattern(urlStr, pattern) {
58             return false
59         }
60     }
61     if len(followPatterns) == 0 {
62         return true
63     }
64     for _, pattern := range followPatterns {
65         if URLMatchesPattern(urlStr, pattern) {
66             return true
67         }
68     }
69 }
70
71 return false
72 }
73
74 func URLMatchesPattern(urlStr string, pattern string) bool {
75     re, err := regexp.Compile(pattern)
76     if err != nil {
77         return false
78     }

```



```

79 |     return re.MatchString(urlStr)
80 | }

1 | package app
2 |
3 | import (
4 |     "context"
5 |     "fmt"
6 |     "log"
7 |     "net/url"
8 |     "regexp"
9 |     "strconv"
10 |    "strings"
11 |    "sync"
12 |    "sync/atomic"
13 |    "time"
14 |    "web_spider/internal/config"
15 |    "web_spider/internal/db"
16 |    "web_spider/internal/models"
17 |    "web_spider/internal/utls"
18 |
19 |    "github.com/PuerkitoBio/goquery"
20 |    "github.com/go-shiori/go-readability"
21 |    "github.com/gocolly/colly"
22 |    "github.com/gocolly/colly/extensions"
23 | )
24 |
25 | var (
26 |     reWhitespace = regexp.MustCompile(`\s+`)
27 |     reArticleDate = regexp.MustCompile(`/\d{4}/[a-z]{3}/\d{1,2}/`)
28 | )
29 |
30 | type PageType int
31 |
32 | const (
33 |     PageTypeUnknown PageType = iota
34 |     PageTypeCategory
35 |     PageTypeArticle
36 | )
37 |
38 | // String representation for debugging
39 | func (p PageType) String() string {
40 |     switch p {
41 |     case PageTypeCategory:
42 |         return "Category"
43 |     case PageTypeArticle:
44 |         return "Article"
45 |     default:
46 |         return "Unknown"
47 |     }

```

```

48 }
49
50 type Spider struct {
51     collector *colly.Collector
52     db *db.MongoDB
53     visited sync.Map
54
55     pageCount int64
56     articleCount int64
57
58     maxPages int64
59     maxDepth int
60     recrawlHours int
61
62     ctx context.Context
63     cancelFunc context.CancelFunc
64     paused int32
65
66     saveChan chan *models.Document
67     wgSave sync.WaitGroup
68
69     doneChan chan struct{}
70 }
71
72 func NewNewsCrawler(cfg *config.SpiderConfig) *Spider {
73     mongoDB, err := db.NewMongoDB(cfg.DB)
74     if err != nil {
75         log.Fatalf("Failed to connect to DB: %v", err)
76         return nil
77     }
78
79     ctx, cancel := context.WithCancel(context.Background())
80
81     c := colly.NewCollector(
82         colly.AllowedDomains("www.theguardian.com", "theguardian.com", "en.wikipedia.org"),
83         colly.UserAgent("Mozilla/5.0 (NewsBot/1.0)"),
84         colly.Async(true),
85     )
86
87     c.IgnoreRobotsTxt = false
88
89     extensions.RandomUserAgent(c)
90
91     c.Limit(&colly.LimitRule{
92         DomainGlob: "*",
93         Parallelism: cfg.Logic.MaxConcurrentWorkers,
94         Delay: time.Duration(cfg.Logic.DelayMS) * time.Millisecond,
95         RandomDelay: 500 * time.Millisecond,
96     })

```

```

97
98     return &Spider{
99         collector: c,
100         db: mongoDB,
101         ctx: ctx,
102         cancelFunc: cancel,
103         maxDepth: cfg.Logic.MaxDepth,
104         recrawlHours: cfg.Logic.RecrawlThresholdHours,
105         saveChan: make(chan *models.Document, 1000),
106         doneChan: make(chan struct{}),
107     }
108 }
109
110 func (w *Spider) Start(startURLs []string, maxPages int64) {
111     w.maxPages = maxPages
112
113     w.setupCollector()
114     w.startWriter()
115
116     go w.runPhases(startURLs)
117
118     w.cliControlLoop()
119 }
120
121 func (w *Spider) runPhases(startURLs []string) {
122     defer close(w.doneChan)
123
124     fmt.Println("PHASE 1: Recrawl started...")
125
126     staleURLs, err := w.db.GetStaleDocuments(w.recrawlHours, 200)
127     if err == nil && len(staleURLs) > 0 {
128         fmt.Printf("Found %d stale pages. Processing...\n", len(staleURLs))
129
130         for _, u := range staleURLs {
131             w.waitForPaused()
132             if w.isStopped() {
133                 return
134             }
135
136             ctx := colly.NewContext()
137             ctx.Put("is_recrawl", "true")
138             ctx.Put("depth", "0")
139
140             w.collector.Request("GET", u, nil, ctx, nil)
141         }
142
143         w.collector.Wait()
144         fmt.Println("PHASE 1: Recrawl finished.")
145     } else {

```

```

146     fmt.Println("No stale pages found. Skipping Phase 1.")
147 }
148
149 if w.isStopped() {
150     return
151 }
152
153 fmt.Println("PHASE 2: Discovery started...")
154
155 startCtx := colly.NewContext()
156 startCtx.Put("depth", "0")
157 startCtx.Put("is_recrawl", "false")
158
159 for _, u := range startURLs {
160     if w.isStopped() {
161         return
162     }
163     w.collector.Request("GET", u, nil, startCtx, nil)
164 }
165
166 w.collector.Wait()
167 fmt.Println("All phases completed.")
168 }
169
170 func (w *Spider) isStopped() bool {
171     select {
172     case <-w.ctx.Done():
173         return true
174     default:
175         return false
176     }
177 }
178
179 func (w *Spider) setupCollector() {
180     w.collector.OnRequest(func(r *colly.Request) {
181         select {
182         case <-w.ctx.Done():
183             r.Abort()
184             return
185         default:
186             }
187
188         for atomic.LoadInt32(&w.paused) == 1 {
189             select {
190             case <-w.ctx.Done():
191                 r.Abort()
192                 return
193             default:
194                 time.Sleep(500 * time.Millisecond)

```

```

195     }
196 }
197
198 isRecrawl := r.Ctx.Get("is_recrawl") == "true"
199 if !isRecrawl && atomic.LoadInt64(&w.pageCount) >= w.maxPages {
200     r.Abort()
201     return
202 }
203 })
204
205 w.collector.OnHTML("html", func(e *colly.HTMLElement) {
206     urlStr := e.Request.URL.String()
207     depth, _ := strconv.Atoi(e.Request.Ctx.Get("depth"))
208     isRecrawl := e.Request.Ctx.Get("is_recrawl") == "true"
209
210     var pageType PageType
211     var source string
212     if strings.Contains(urlStr, "en.wikipedia.org") {
213         pageType = w.determinePageTypeWiki(urlStr)
214         source = "wikipedia"
215     } else if strings.Contains(urlStr, "theguardian.com") {
216         pageType = w.determinePageTypeGuardian(urlStr)
217         source = "theguardian"
218     } else {
219         pageType = PageTypeUnknown
220     }
221
222     if pageType == PageTypeArticle {
223         article, err := w.extractContent(string(e.Response.Body), urlStr)
224         if err == nil && len(article.Text) > 200 {
225             doc := models.Document{
226                 Title: article.Title,
227                 HTMLContent: article.HTML,
228                 Source: source,
229                 ContentHash: utils.ComputeContentHash(article.HTML),
230                 NormalizedURL: utils.NormalizeURL(urlStr),
231                 LastScraped: time.Now().Unix(),
232                 ContentLength: len(article.HTML),
233             }
234             select {
235             case w.saveChan <- &doc:
236             default:
237                 log.Println("Save queue full, dropping document")
238             }
239         }
240     }
241
242     if !isRecrawl {
243         w.processLinks(e, depth, pageType)

```

```

244     }
245 })
246
247 w.collector.OnError(func(r *colly.Response, err error) {
248     if err == colly.ErrRobotsTxtBlocked {
249         log.Printf("Skipped (Robots.txt): %s", r.Request.URL)
250         return
251     }
252
253     if r.StatusCode != 429 {
254         log.Printf("Error on %s: %v", r.Request.URL, err)
255     }
256 })
257 }
258
259 func getURLPath(href string) string {
260     u, err := url.Parse(href)
261     if err != nil {
262         return ""
263     }
264     return u.Path
265 }
266
267 func (w *Spider) determinePageTypeGuardian(href string) PageType {
268     path := getURLPath(href)
269     if path == "" {
270         return PageTypeUnknown
271     }
272
273     if strings.Contains(href, "/video/") || strings.Contains(href, "/audio/") ||
274        strings.Contains(href, "/gallery/") || strings.Contains(href, "/crosswords/") {
275         return PageTypeUnknown
276     }
277
278     if reArticleDate.MatchString(path) {
279         return PageTypeArticle
280     }
281
282     segments := strings.Split(strings.Trim(path, "/"), "/")
283     if len(segments) >= 1 && len(segments) <= 3 {
284         return PageTypeCategory
285     }
286
287     return PageTypeUnknown
288 }
289
290 func (w *Spider) determinePageTypeWiki(href string) PageType {
291     path := getURLPath(href)
292     if path == "" {

```

```

293     return PageTypeUnknown
294 }
295
296 excluded := []string{
297     "File:", "Special:", "Talk:", "User:", "Template:",
298     "Help:", "Wikipedia:", "Draft:", "Portal:", "Main_Page",
299     "Template_talk:", "Category_talk:",
300     "action=", "diff=", "oldid=",
301 }
302 for _, exc := range excluded {
303     if strings.Contains(path, exc) {
304         return PageTypeUnknown
305     }
306 }
307 if strings.Contains(path, "/Category:") {
308     return PageTypeCategory
309 }
310
311 if strings.HasPrefix(path, "/wiki/") {
312     return PageTypeArticle
313 }
314 return PageTypeUnknown
315 }
316
317 func (sc *Spider) extractContent(rawHTML string, pageURL string) (*models.
    ExtractedArticle, error) {
318     parsedURL, err := url.Parse(pageURL)
319     if err != nil {
320         return nil, err
321     }
322
323     reader := strings.NewReader(rawHTML)
324     article, err := readability.FromReader(reader, parsedURL)
325     if err != nil {
326         return nil, err
327     }
328
329     doc, err := goquery.NewDocumentFromReader(strings.NewReader(article.Content))
330     if err != nil {
331         return nil, err
332     }
333
334     doc.Find("figure, .dcr-citation, aside, .element-atom, script, style, .mw-
        editsection").Remove()
335
336     cleanText := doc.Text()
337     cleanText = reWhitespace.ReplaceAllString(cleanText, " ")
338     cleanText = strings.TrimSpace(strings.Split(cleanText, "Reuse this content")[0])
339

```

```

340     return &models.ExtractedArticle{
341         Title: article.Title,
342         Text: cleanText,
343         HTML: article.Content,
344         Excerpt: article.Excerpt,
345     }, nil
346 }
347
348 func (w *Spider) processLinks(e *colly.HTML_Element, currentDepth int, currentPageType
    PageType) {
349     if currentDepth >= w.maxDepth {
350         return
351     }
352
353     currentURL := e.Request.URL
354
355     if strings.Contains(currentURL.Host, "wikipedia.org") {
356         if currentPageType == PageTypeArticle {
357             return
358         }
359     }
360
361     targetSelector := "main a[href], section a[href], .fc-container a[href], #mw-pages a
        [href], #mw-subcategories a[href]"
362
363     e.ForEach(targetSelector, func(_ int, el *colly.HTML_Element) {
364         href := el.Attr("href")
365         absoluteURL := el.Request.AbsoluteURL(href)
366
367         u, err := url.Parse(absoluteURL)
368         if err != nil {
369             return
370         }
371
372         u.RawQuery = ""
373         u.Fragment = ""
374         cleanURL := u.String()
375
376         if _, loaded := w.visited.LoadOrStore(cleanURL, true); loaded {
377             return
378         }
379
380         var targetPageType PageType
381         if strings.Contains(cleanURL, "en.wikipedia.org") {
382             targetPageType = w.determinePageTypeWiki(cleanURL)
383         } else if strings.Contains(cleanURL, "theguardian.com") {
384             targetPageType = w.determinePageTypeGuardian(cleanURL)
385         } else {
386             targetPageType = PageTypeUnknown

```



```

387     }
388
389     if targetPageType == PageTypeUnknown {
390         return
391     }
392
393     ctx := colly.NewContext()
394     ctx.Put("depth", strconv.Itoa(currentDepth+1))
395
396     w.collector.Request("GET", cleanURL, nil, ctx, nil)
397 }
398 }
399
400 func (w *Spider) cliControlLoop() {
401     fmt.Println("Controls: [p]ause, [r]esume, [s]top")
402
403     go func() {
404         <-w.doneChan
405         fmt.Println("\nCrawling finished. Press Enter to exit report...")
406     }()
407
408     var input string
409     for {
410         select {
411             case <-w.ctx.Done():
412                 w.shutdown()
413                 return
414             default:
415             }
416             _, err := fmt.Scanln(&input)
417             if err != nil {
418                 select {
419                     case <-w.doneChan:
420                         w.shutdown()
421                         return
422                     default:
423                         time.Sleep(100 * time.Millisecond)
424                         continue
425                 }
426             }
427
428             switch strings.ToLower(input) {
429             case "p":
430                 if atomic.CompareAndSwapInt32(&w.paused, 0, 1) {
431                     fmt.Println("PAUSED. Waiting for workers to sleep...")
432                 }
433             case "r":
434                 if atomic.CompareAndSwapInt32(&w.paused, 1, 0) {
435                     fmt.Println("RESUMED")

```

```

436     }
437     case "s", "q":
438         fmt.Println("STOPPING initiated...")
439         w.cancelFunc()
440         return
441     }
442 }
443 }
444
445 func (w *Spider) shutdown() {
446     if w.saveChan != nil {
447         close(w.saveChan)
448     }
449     w.wgSave.Wait()
450     fmt.Printf("\n REPORT: Scraped Pages: %d | Articles Saved: %d\n", w.pageCount, w.
         articleCount)
451     fmt.Println(" Bye!")
452 }
453
454 func (w *Spider) startWriter() {
455     w.wgSave.Add(1)
456     go func() {
457         defer w.wgSave.Done()
458         for doc := range w.saveChan {
459             w.waitForPaused()
460             if err := w.db.SaveDocument(doc); err != nil {
461                 log.Printf("Error saving to DB: %v", err)
462             } else {
463                 atomic.AddInt64(&w.articleCount, 1)
464                 count := atomic.AddInt64(&w.pageCount, 1)
465                 if count%5 == 0 {
466                     fmt.Printf("Saved: %d ... \r", count)
467                 }
468             }
469         }
470     }()
471 }
472
473 func (w *Spider) waitForPaused() {
474     if atomic.LoadInt32(&w.paused) == 0 {
475         return
476     }
477
478     for atomic.LoadInt32(&w.paused) == 1 {
479         select {
480             case <-w.ctx.Done():
481                 return
482             case <-time.After(200 * time.Millisecond):
483                 continue

```

484		}
485		}
486		}

3 Выводы

При написании поискового робота я научился обкачивать документы с сайтов с использованием языка Go. Узнал про существование robots.txt и принципах вежливого скрапинга, научился обходить капчи, редиректы и прочие препятствия на пути. На практике познакомился с тем, как устроены HTML-страницы, научился их разбирать. Ранее я никогда не занимался web-скрапингом, для меня это был интересный опыт, который обязательно пригодится мне в будущем!

Список литературы

- [1] Маннинг, Рагхаван, Шютце *Введение в информационный поиск* — Издательский дом «Вильямс», 2011. Перевод с английского: доктор физ.-мат. наук Д. А. Ключина — 528 с. (ISBN 978-5-8459-1623-4 (рус.))
- [2] Веб-скрапинг при помощи Go | Полное руководство 2024 // Nstbrowser: [сайт]. URL: <https://www.nstbrowser.io/ru/blog/golang-web-scraping> (дата обращения: 12.12.2025).