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 реализовации 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

Архитектурно приложение разделено на три ключевых слоя:

Слой сбора. Это основная логика в пакете app (Spider). Краулер запускает асинхронные HTTP-воркеры (через colly), которые обходят страницы. Внутри коллбэков (OnHTML) происходит эвристический анализ: URL классифицируется как «Статья» или «Категория». Если это статья, контент очищается через go-readability и отправляется в буферизированный канал saveChan. Если категория — извлекаются новые ссылки для рекурсивного обхода (до заданной глубины MaxDepth).

Слой сохранения (Consumer): Отдельная горутина (startWriter) постоянно слушает канал saveChan и записывает обработанные документы (models.Document) в MongoDB. Это позволяет не блокировать быстрый процесс скачивания страниц медленными операциями записи в БД.

Слой управления и конфигурации: Приложение управляется через CLI с использованием атомарных флагов и context для синхронизации. Логика запуска разделена на две фазы: обновление устаревших записей из БД и поиск новых страниц по стартовым URL. Все параметры (лимиты, таймауты, селекторы) вынесены в YAML-конфигурацию.

```
package main

import (
  "flag"
  "fmt"
  "web_spider/internal/app"
  "web_spider/internal/config"
)

func generateArchiveURLs(base string, fin_page int) []string {
    var urls []string
    for page := 0; page < fin_page; page++ {
    urls = append(urls, fmt.Sprintf("%s?page=%d", base, page))
    }
    return urls
}

func main() {
  configPath := flag.String("config", "config.yaml", "Path to configuration file")

    flag.Parse()

    cfg, err := config.LoadConfig(*configPath)

  if err != nil {
    panic(err)
  }
```

```

  crawler := app.NewNewsCrawler(cfg)

  startURLGuardian := "https://www.theguardian.com/sport"
  startURLWiki := "https://en.wikipedia.org/wiki/Category:Sports"
  urls := generateArchiveURLs(startURLGuardian, 100);
  urls = append(urls, startURLGuardian)
  urls = append(urls, startURLWiki)


  crawler.Start(urls, 1000000000)
}
```

```
package models

type ExtractedArticle struct {
  Title string
  Text string
  HTML string
  Excerpt string
}

type Document struct {
  ID string `bson:"_id"`
  URL string `bson:"url"`
  NormalizedURL string `bson:"normalized_url"`
  Source string `bson:"source"`
  HTMLContent string `bson:"html_content"`
  Title string `bson:"title"`
  ContentHash string `bson:"content_hash"`
  FirstScraped int64 `bson:"first_scraped"`
  LastScraped int64 `bson:"last_scraped"`
  ContentLength int `bson:"content_length"`
}
```

```
package config

import (
  "os"

  "gopkg.in/yaml.v2"
)

type SourceConfig struct {
  Name string `yaml:"name"`
  BaseURLs []string `yaml:"base_urls"`
  StartURLs []string `yaml:"start_urls"`
  FollowPatterns []string `yaml:"follow_patterns"`
  ExcludePatterns []string `yaml:"exclude_patterns"`
  MaxPages int `yaml:"max_pages"`
```

```
  Priority int `yaml:"priority"`
}

type DBConfig struct {
  Connection string `yaml:"connection"`
  Database string `yaml:"database"`
  Collections struct {
    Documents string `yaml:"documents"`
    SpiderState string `yaml:"spider_state"`
    SpiderHistory string `yaml:"spider_history"`
  } `yaml:"collections"`
}

type LogicConfig struct {
  DelayMS int `yaml:"delay_ms"`
  TimeoutSec int `yaml:"timeout_sec"`
  MaxRetries int `yaml:"max_retries"`
  MaxDepth    int `yaml:"max_depth"`
  RecrawlThresholdHours int `yaml:"recrawl_threshold_hours"`
  MaxConcurrentWorkers int `yaml:"max_concurrent_workers"`
}

type SpiderConfig struct {
  DB DBConfig `yaml:"db"`
  Logic LogicConfig `yaml:"logic"`
  Sources map[string]SourceConfig `yaml:"sources"`
}

func LoadConfig(path string) (*SpiderConfig, error) {
  data, err := os.ReadFile(path)
  if err != nil {
    return nil, err
  }
  var cfg SpiderConfig
  if err := yaml.Unmarshal(data, &cfg); err != nil {
    return nil, err
  }
  return &cfg, nil
}
```

```
package utils

import (
  "crypto/md5"
  "fmt"
  "net/url"
  "regexp"
  "strings"
)

```

```
11
12 func NormalizeURL(urlStr string) string {
13   parsed, err := url.Parse(urlStr)
14   if err != nil {
15     return urlStr
16   }
17
18   parsed.Fragment = ""
19
20   parsed.Host = strings.TrimPrefix(parsed.Host, "www.")
21
22   if parsed.Scheme == "" {
23     parsed.Scheme = "https"
24   }
25
26   return parsed.String()
27 }
28
29 func ExtractLinksFromHTML(htmlContent, baseURL string) []string {
30   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   for _, pattern := range excludePatterns {
56     if URLMatchesPattern(urlStr, pattern) {
57       return false
58     }
```

```
  }

  if len(followPatterns) == 0 {
    return true
  }

  for _, pattern := range followPatterns {
    if URLMatchesPattern(urlStr, pattern) {
      return true
    }
  }

  return false
}

func URLMatchesPattern(urlStr string, pattern string) bool {
  re, err := regexp.Compile(pattern)
  if err != nil {
    return false
  }
  return re.MatchString(urlStr)
}
```

```
package app

import (
  "context"
  "fmt"
  "log"
  "net/url"
  "regexp"
  "strconv"
  "strings"
  "sync"
  "sync/atomic"
  "time"
  "web_spider/internal/config"
  "web_spider/internal/db"
  "web_spider/internal/models"
  "web_spider/internal/utils"

  "github.com/PuerkitoBio/goquery"
  "github.com/go-shiori/go-readability"
  "github.com/gocolly/colly"
  "github.com/gocolly/colly/extensions"
)

var (
  reWhitespace = regexp.MustCompile(`\s+`)
  reArticleDate = regexp.MustCompile(`/\d{4}/[a-z]{3}/\d{1,2}/`)
```

```
)

type PageType int

const (
  PageTypeUnknown PageType = iota
  PageTypeCategory
  PageTypeArticle
)

// String representation for debugging
func (p PageType) String() string {
  switch p {
  case PageTypeCategory:
    return "Category"
  case PageTypeArticle:
    return "Article"
  default:
    return "Unknown"
  }
}

type Spider struct {
  collector *colly.Collector
  db *db.MongoDB
  visited sync.Map

  pageCount int64
  articleCount int64

  maxPages int64
  maxDepth int
  recrawlHours int

  ctx context.Context
  cancelFunc context.CancelFunc
  paused int32

  saveChan chan *models.Document
  wgSave sync.WaitGroup

  doneChan chan struct{}
}

func NewNewsCrawler(cfg *config.SpiderConfig) *Spider {
  mongoDB, err := db.NewMongoDB(cfg.DB)
  if err != nil {
    log.Fatalf("Failed to connect to DB: %v", err)
    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
```

```
  staleURLs, err := w.db.GetStaleDocuments(w.recrawlHours, 200)
  if err == nil && len(staleURLs) > 0 {
    fmt.Printf("Found %d stale pages. Processing...\n", len(staleURLs))

    for _, u := range staleURLs {
      w.waitIfPaused()
      if w.isStopped() {
        return
      }

      ctx := colly.NewContext()
      ctx.Put("is_recrawl", "true")
      ctx.Put("depth", "0")

      w.collector.Request("GET", u, nil, ctx, nil)
    }

    w.collector.Wait()
    fmt.Println("PHASE 1: Recrawl finished.")
  } else {
    fmt.Println("No stale pages found. Skipping Phase 1.")
  }

  if w.isStopped() {
    return
  }

  fmt.Println("PHASE 2: Discovery started...")

  startCtx := colly.NewContext()
  startCtx.Put("depth", "0")
  startCtx.Put("is_recrawl", "false")

  for _, u := range startURLs {
    if w.isStopped() {
      return
    }
    w.collector.Request("GET", u, nil, startCtx, nil)
  }

  w.collector.Wait()
  fmt.Println("All phases completed.")
}

func (w *Spider) isStopped() bool {
  select {
  case <-w.ctx.Done():
    return true
  default:
```

```
    return false
  }
}

func (w *Spider) setupCollector() {
  w.collector.OnRequest(func(r *colly.Request) {
    select {
    case <-w.ctx.Done():
      r.Abort()
      return
    default:
    }

    for atomic.LoadInt32(&w.paused) == 1 {
      select {
      case <-w.ctx.Done():
        r.Abort()
        return
      default:
        time.Sleep(500 * time.Millisecond)
      }
    }

    isRecrawl := r.Ctx.Get("is_recrawl") == "true"
    if !isRecrawl && atomic.LoadInt64(&w.pageCount) >= w.maxPages {
      r.Abort()
      return
    }
  })

  w.collector.OnHTML("html", func(e *colly.HTMLElement) {
    urlStr := e.Request.URL.String()
    depth, _ := strconv.Atoi(e.Request.Ctx.Get("depth"))
    isRecrawl := e.Request.Ctx.Get("is_recrawl") == "true"

    var pageType PageType
    var source string
    if strings.Contains(urlStr, "en.wikipedia.org") {
      pageType = w.determinePageTypeWiki(urlStr)
      source = "wikipedia"
    } else if strings.Contains(urlStr, "theguardian.com") {
      pageType = w.determinePageTypeGuardian(urlStr)
      source = "theguardian"
    } else {
      pageType = PageTypeUnknown
    }

    if pageType == PageTypeArticle {
      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
```

```
  }

  reader := strings.NewReader(rawHTML)
  article, err := readability.FromReader(reader, parsedURL)
  if err != nil {
    return nil, err
  }

  doc, err := goquery.NewDocumentFromReader(strings.NewReader(article.Content))
  if err != nil {
    return nil, err
  }

  doc.Find("figure, .dcr-citation, aside, .element-atom, script, style, .mw-
      editsection").Remove()

  cleanText := doc.Text()
  cleanText = reWhitespace.ReplaceAllString(cleanText, " ")
  cleanText = strings.TrimSpace(strings.Split(cleanText, "Reuse this content")[0])

  return &models.ExtractedArticle{
    Title: article.Title,
    Text: cleanText,
    HTML: article.Content,
    Excerpt: article.Excerpt,
  }, nil
}

func (w *Spider) processLinks(e *colly.HTMLElement, currentDepth int, currentPageType
    PageType) {
  if currentDepth >= w.maxDepth {
    return
  }

  currentURL := e.Request.URL

  if strings.Contains(currentURL.Host, "wikipedia.org") {
    if currentPageType == PageTypeArticle {
      return
    }
  }

  targetSelector := "main a[href], section a[href], .fc-container a[href], #mw-pages a
      [href], #mw-subcategories a[href]"

  e.ForEach(targetSelector, func(_ int, el *colly.HTMLElement) {
    href := el.Attr("href")
    absoluteURL := el.Request.AbsoluteURL(href)

```

```
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("\nREPORT: 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.waitIfPaused()
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) waitIfPaused() {
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. Клюшина — 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).