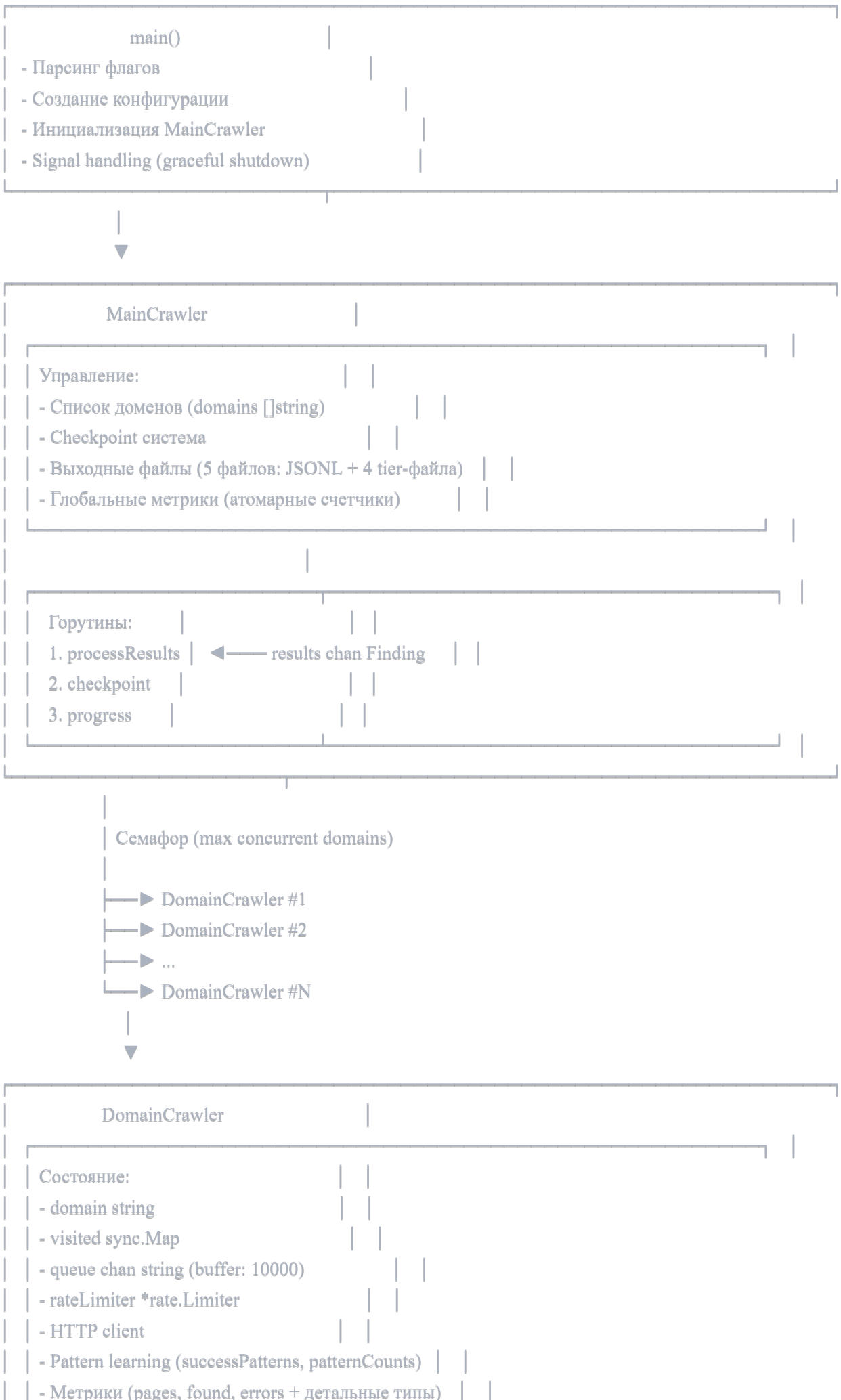
АРХИТЕКТУРНАЯ КАРТА: Web Crawler для поиска комментариев

Версия: 1.0

Дата: 12 октября 2025

Файл: pavuk5.go

📐 Общая архитектура системы


main()
- Парсинг флагов
- Создание конфигурации
- Инициализация MainCrawler
- Signal handling (graceful shutdown)
MainCrawler
Управление:
- Список доменов (domains []string)
- Checkpoint система
- Выходные файлы (5 файлов: JSONL + 4 tier-файла)
- Глобальные метрики (атомарные счетчики)
Горутины:
1. processResults
results chan Finding
2. checkpoint
3. progress
Семафор (max concurrent domains)
DomainCrawler #1
DomainCrawler #2
...
DomainCrawler #N
DomainCrawler
Состояние:
- domain string
- visited sync.Map
- queue chan string (buffer: 10000)
- rateLimiter *rate.Limiter
- HTTP client
- Pattern learning (successPatterns, patternCounts)
- Метрики (pages, found, errors + детальные типы)

🧩 Детальное описание модулей

1. CONFIGURATION MODULE

Файл: Строки 42-92

Тип: Структуры данных

Структура Config

```go
type Config struct {
    WorkersPerDomain   int           // Количество воркеров на домен
    MaxTotalWorkers    int           // Лимит общего числа воркеров
    MaxPagesPerDomain  int           // Лимит страниц на домен
    RequestTimeout     time.Duration // Таймаут HTTP запроса
    DomainTimeout      time.Duration // Таймаут обработки домена
    RatePerDomain      float64       // Rate limit (запросов/сек)
    OutputFile         string        // Основной JSONL output
    CheckpointFile     string        // Файл для сохранения прогресса
    CheckpointInterval time.Duration // Частота сохранения
    RetryAttempts      int           // Количество повторных попыток
    UserAgents         []string      // Список User-Agent'ов
}
```

Назначение:

- Централизованное хранение всех параметров работы
- DefaultConfig() возвращает разумные значения по умолчанию

Зависимости:

- Используется: MainCrawler, DomainCrawler
- Не зависит ни от чего

Проблемы:

- ❌ Нет валидации значений (можно установить негативные числа)
- ❌ Жестко закодированные UserAgent'ы
- ⚠️ Нет возможности загрузки из файла

Связи:

```
Config ──────> MainCrawler
    └──────> DomainCrawler
```

---

2. DATA STRUCTURES MODULE

Файл: Строки 94-210

Тип: Core types

Finding (основная единица результата)

```go
type Finding struct {
    Domain        string
    URL           string
    Method        string    // modern_system | native_form | etc.
    System        string    // disqus | wordpress | native | etc.
    Confidence    float64   // 0.0 - 1.0
    Tier          int       // 1-4 (1=definite, 4=ambiguous)
    HasTextarea   bool
    TextareaCount int
    Signals       []string  // Список обнаруженных сигналов
    Timestamp     time.Time
    PageContext   float64   // NEW: контекст страницы
    URLWeight     float64   // NEW: вес URL
    FormSemantics float64   // NEW: семантика формы
}
```

Назначение:

- Хранение информации о найденной системе комментариев
- Включает метрики для классификации

DomainCrawler (воркер для одного домена)

```go
type DomainCrawler struct {
    // Конфигурация
    domain      string
    config      *Config
    client      *http.Client
    rateLimiter *rate.Limiter

    // Состояние
    visited     sync.Map        // URL → bool (посещенные)
    queue       chan string     // Очередь URL для обработки
    results     chan *Finding   // Канал результатов (shared)

    // Метрики (atomic counters)
    errors          int64
    pages           int64
    found           int64
    timeoutErrors   int64
    networkErrors   int64
    parseErrors     int64
    notFoundErrors  int64

    // Pattern Learning
    successPatterns sync.Map  // URL → bool (успешные)
    patternCounts   sync.Map  // pattern → int (счетчики)

    // Управление жизненным циклом
    ctx    context.Context
    cancel context.CancelFunc
    wg     sync.WaitGroup
}
```

Назначение:

- Полная автономная обработка одного домена
- Управление воркерами и очередью
- Обучение на паттернах успешных находок

MainCrawler (оркестратор)

```go
type MainCrawler struct {
    config        *Config
    domains       []string
    domainCrawlers sync.Map      // domain → *DomainCrawler
    results       chan *Finding // Shared channel
    checkpoint    *Checkpoint
    checkpointMu  sync.RWMutex

    // Файлы вывода
    outputFile    *os.File
    tier1File     *os.File
    tier2File     *os.File
    tier3File     *os.File
    tier4File     *os.File

    // Метрики
    tierCounts    [5]int64
    startTime     time.Time
    domainsTotal  int64
    domainsActive int64
    domainsDone   int64
    domainsWithComments int64
    pagesTotal    int64
    findingsTotal int64
    errorsTotal   int64
    errorTypes    sync.Map

    // Управление
    ctx    context.Context
    cancel context.CancelFunc
    wg     sync.WaitGroup
}
```

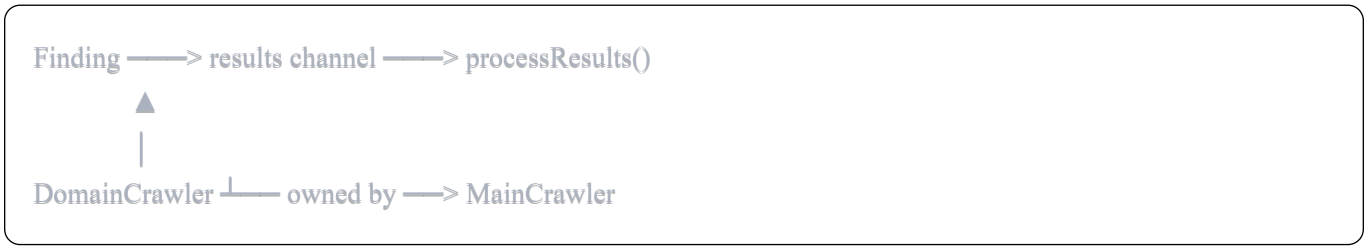
Назначение:

- Координация всех DomainCrawler'ов
- Агрегация результатов
- Checkpoint и прогресс

Checkpoint (сохранение состояния)

```go
type Checkpoint struct {
    ProcessedDomains map[string]bool   // domain → processed
    Statistics       map[string]int64  // key → value
    Timestamp        time.Time
}
```

Связи:



Проблемы в структурах:

- ❌ sync.Map используется без type safety (count.(int) может паниковать)
- ❌ Unbounded channels и maps (memory growth)
- ⚠️ Дублирование метрик между DomainCrawler и MainCrawler

---

3. CONTEXT ANALYSIS MODULE

Файл: Строки 212-495

Тип: Аналитические функции

3.1 analyzeURLContext()

Строки: 214-280

Входы: URL string

Выходы: contextType string, weight float64

Логика:

1. Сканирует URL на наличие ключевых паттернов
2. Каждый паттерн имеет вес (+ или -)
3. Положительные: /blog/, /post/, /article/, даты в URL
4. Отрицательные: /contact, /admin, /search, /cart
5. Веса накапливаются без нормализации

Результат:

- "content" если weight > 0.2
- "system" если weight < -0.2
- "neutral" иначе

Причинно-следственная связь:

```
URL паттерны → weight → contextType → PageContext → Tier classification
                                    → URL prioritization
```

Проблемы:

- ⚠️ Регулярки компилируются каждый раз (performance)
- ⚠️ Веса подобраны эмпирически без обоснования
- ✅ Логика правильная, работает

3.2 analyzePageStructure()

Строки: 283-380

Входы: *goquery.Document

Выходы: score float64 (0.0-1.0)

Паттерны для поиска:

1. **Repeating blocks** (findRepeatingStructures) - +0.15-0.25
2. **Timestamps** рядом с текстом - +0.15-0.30
3. **Nested structures** (.reply, .children) - +0.20-0.35
4. **User indicators** (.avatar, .author) - +0.20
5. **Comment counters** - +0.15
6. **Reply buttons** - +0.25

Причинно-следственная связь:

```
DOM структура → паттерны → score → PageContext → влияет на:
                                    - Form scoring
                                    - Tier classification
                                    - URL filtering
```

Проблемы:

- ⚠️ Множественные проходы по DOM (performance)
- ⚠️ Магические числа без комментариев
- ✅ Логика умная и корректная

3.3 findRepeatingStructures()

Строки: 383-430
Вспомогательная для analyzePageStructure

Логика:

1. Собирает все классы элементов
2. Подсчитывает повторения
3. Игнорирует служебные классы (wrapper, container)
4. Возвращает максимум повторений одного класса

Связь:

```
DOM → classCount map → maxRepeat → contributes to PageContext
```

3.4 analyzeFormSemantics()

Строки: 433-558
Входы: form *goquery.Selection, pageDoc *goquery.Document
Выходы: score float64 (-1.0 to 1.0)

Сложная эвристика:

1. Анализ текста ПЕРЕД формой (commentKeywords vs contactKeywords)
2. Анализ полей формы (email, name, phone, subject, company)
3. Pattern matching:
   - Classic comment: name + email + [website] → +0.35
   - Minimal comment: email OR name → +0.2
   - Contact form: phone OR subject OR company → -0.3-0.4
4. Проверка кнопки submit (текст)
5. Контекст формы (внутри article?)
6. Наличие существующих комментариев рядом

Причинно-следственная связь:

```
Form HTML → Fields + Context → Semantic score → FormSemantics →
                                    ↓
                                Tier classification
                                Confidence adjustment
```

Проблемы:

- ⚠️ Очень длинная функция (~125 строк)
- ⚠️ Магические коэффициенты
- ⚠️ Может давать false positives на сложных формах
- ✅ В целом логика правильная

3.5 categorizeSignals()

Строки: 561-593
Вспомогательная для классификации

Логика: Разбивает signals на категории (strong/weak/negative)

Связь:

```
signals []string → categorize → counts → используется в classifyFindingTierV2
```

---

4. TIER CLASSIFICATION MODULE

Файл: Строки 602-655
Тип: Decision logic

classifyFindingTierV2()

Назначение: Финальная классификация находки в Tier 1-4

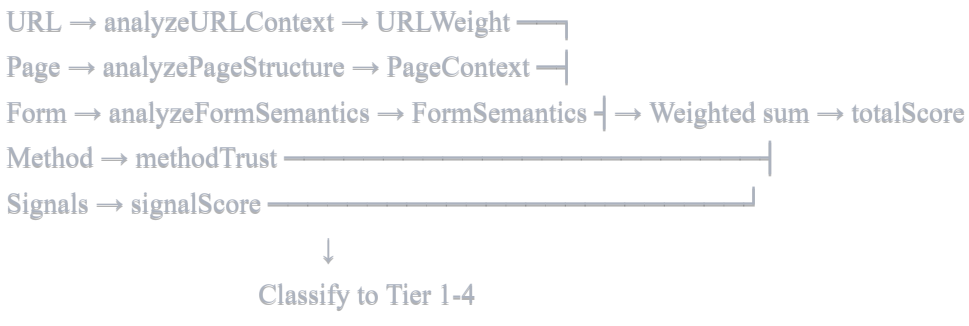
Входы:

- finding *Finding (с заполненными метриками)
- doc *goquery.Document (для дополнительного анализа если нужно)

Алгоритм:

```
1. Извлечение метрик:
   - URLWeight (from finding)
   - PageContext (from finding)
   - FormSemantics (from finding)

2. Доверие к методу:
   modern_system   → 0.9
   dynamic_system  → 0.7
   native_form     → 0.5
   placeholder     → 0.3

3. Анализ сигналов:
   categorizeSignals(finding.Signals) →
   signalScore = strong*0.4 + weak*0.15 - negative*0.5

4. Взвешенная сумма:
   totalScore = urlWeight*0.15 + pageContext*0.25 +
                formSemantics*0.20 + methodTrust*0.25 +
                signalScore*0.15

5. Дополнительное усреднение с confidence:
   totalScore = (totalScore + finding.Confidence) / 2

6. Классификация:
   ≥ 0.75 → Tier 1 (definite)
   ≥ 0.50 → Tier 2 (probable)
   ≥ 0.25 → Tier 3 (possible)
   < 0.25 → Tier 4 (ambiguous)
```

Причинно-следственная связь:



Веса компонентов:

- PageContext: 25% (самый важный)
- MethodTrust: 25% (метод детектирования)
- FormSemantics: 20%
- URLWeight: 15%
- SignalScore: 15%

Проблемы:

- ⚠️ Магические пороги (0.75, 0.50, 0.25)
- ⚠️ Веса не обоснованы
- ⚠️ Усреднение с confidence может быть избыточным
- ✅ Логика в целом разумная

---

5. DOMAIN CRAWLER MODULE

Файл: Строки 657-880

Тип: Core worker

NewDomainCrawler()

Строки: 657-703

Создание:

1. Context с таймаутом (DomainTimeout)
2. HTTP Transport с настройками:
   - MaxIdleConns: 10
   - MaxIdleConnsPerHost: WorkersPerDomain
   - IdleConnTimeout: 90s
   - TLS InsecureSkipVerify: true ⚠️
   - DialTimeout: 30s
3. HTTP Client с redirect limit (5)
4. Rate Limiter (config.RatePerDomain req/sec)
5. Queue channel (buffer: 10000)

Проблемы:

- ❌ InsecureSkipVerify = true (security issue для production)
- ⚠️ Hardcoded значения таймаутов
- ✅ Хорошая настройка connection pool

Start()

Строки: 705-712

Логика:

1. Запускает N воркеров (config.WorkersPerDomain)
2. Запускает seedInitialURLs() (в основной горутине)
3. Запускает монитор

Причинно-следственная связь:

```
Start() → spawn workers → seedInitialURLs → fill queue
                ↘ monitor → track progress

workers consume queue → process URLs → add new URLs to queue
                      → send findings to results
```

seedInitialURLs()

Строки: 714-757

Логика:

1. Пробует HTTPS, потом HTTP для главной страницы
2. Если не удалось - пробует sitemap
3. Обрабатывает главную страницу:
   - detectComments()
   - extractLinks()
   - prioritizeLinks()
   - Добавляет в queue
4. Вызывает processSitemap()

Проблемы:

- ⚠️ Если главная страница 404, полагается только на sitemap
- ✅ Разумная fallback стратегия

worker()

Строки: 759-825

Основной цикл обработки:

```
go

for {
    select {
    case <-ctx.Done():
        return
    case url := <-queue:
        // 1. Check visited
        // 2. Check page limit
        // 3. Rate limit
        // 4. processURL(url)
        // 5. Check stop conditions
    }
}
```

Условия останова:

- found >= 50 комментариев
- pages >= MaxPagesPerDomain
- context cancelled

⚠ КРИТИЧЕСКИЙ БАГ:

```
go

defer func() {
    if r := recover(); r != nil {
        dc.wg.Add(1)
        go dc.worker(id)  // ❌ Создает дубликат!
    }
}()
```

Проблема: При panic создается новый worker, но старый продолжает работу после recover.

Результат: удвоение воркеров при каждом panic.

Причинно-следственная связь:

```
Queue URL → rate limit → processURL → detect → send to results
                                    → extract links → prioritize → queue
```

---

6. URL PROCESSING MODULE

Файл: Строки 827-950

Тип: Core processing pipeline

processURL()

Строки: 827-908

Pipeline:

```
1. Increment pages counter
2. fetchAndParse(url) → doc, finalURL
   ├─ success → continue
   └─ error → classify error type, log, return

3. detectComments(doc, finalURL) → finding
   ├─ nil → just extract links
   └─ *Finding → classify tier, filter, send

4. classifyFindingTierV2(finding, doc) → tier

5. Filter low confidence:
   - confidence < 0.1 → skip all
   - tier 4 && confidence < 0.3 → skip

6. Send to results channel

7. If tier ≤ 2 && confidence ≥ 0.7:
   learnFromSuccess(finalURL)  // ⚠ Has race condition

8. Log findings by tier

9. Extract and prioritize links:
   extractLinks(doc, finalURL) →
   prioritizeLinksByFreshnessEnhanced(links, doc) →
   Add to queue (if not visited, under limit)
```

Классификация ошибок:

- Timeout → timeoutErrors++
- 404 → notFoundErrors++ (не логируется)
- Network → networkErrors++
- 403/401 → логируется
- Parse → parseErrors++

Проблемы:

- ⚠️ Ошибки подсчитываются, но не используются для адаптации
- ⚠️ Фильтрация по confidence может быть слишком агрессивной
- ✅ Логика обработки правильная

Причинно-следственная связь:

```
URL → fetch → parse → detect → classify → filter → results
                    → learn patterns
                    → extract links → prioritize → queue (loop)
```

fetchAndParse()

Строки: 910-988

Логика с retry:

```
for attempt := 1 to RetryAttempts:
   1. Create HTTP request with context
   2. Set random User-Agent
   3. Set headers (Accept, Accept-Language)
   4. Execute request
   5. Check status code (fail on >= 400)
   6. Handle charset encoding
   7. Read body (limit: 10MB)
   8. Parse with goquery
   9. Return doc + final URL

   On error:
      if timeout && attempt < max:
         sleep(attempt * 2 seconds)  // ⚠️ Doesn't check context!
         retry
      else:
         return error
```

⚠️ КРИТИЧЕСКИЙ БАГ:

```
go

time.Sleep(time.Second * time.Duration(attempt*2))
```

Блокирует без проверки context. При graceful shutdown worker будет висеть до конца sleep.

Проблемы:

- ❌ Context не проверяется при sleep
- ⚠️ Exponential backoff слишком простой
- ⚠️ 10MB limit может быть мало для больших страниц
- ✅ Charset handling правильный

---

7. DETECTION MODULE

Файл: Строки 990-1485

Тип: Core detection logic

Это **самый важный модуль** - сердце системы.

7.1 detectStructuredData()

Строки: 990-1095

Методы детектирования:

1. **Schema.org микроразметка:**

```
html

  <div itemtype="http://schema.org/Comment">
```

Score: +0.9

2. **JSON-LD:**

```
json

  {
   "@type": "Comment",
   "@type": "UserComments",
   "@type": "DiscussionForumPosting"
  }
```

Score: +0.85

3. **Open Graph meta tags:**

```
html

  <meta property="article:published_time" content="...">
```

Score: +0.2

4. **Microformats:**

```html
<div class="h-entry">
  <div class="p-comment">
```

Score: +0.3

5. **WordPress classes:**

```
.wp-comment, .comment-metadata, .comment-reply-link
```

Score: +0.15 each, max +0.6

6. **Drupal indicators:**

```
.comment-wrapper, .comment-submitted
```

Score: +0.1 each, max +0.4

7. **Data attributes:**

```html
<div data-comment-id="123">
```

Score: +0.1

Return: Finding если score > 0.5

Проблемы:

- ⚠️ Может давать false positives на сайтах с микроразметкой без реальных комментариев
- ✅ Очень надежный метод для современных CMS

7.2 detectComments() - ГЛАВНАЯ ФУНКЦИЯ

Строки: 1098-1153

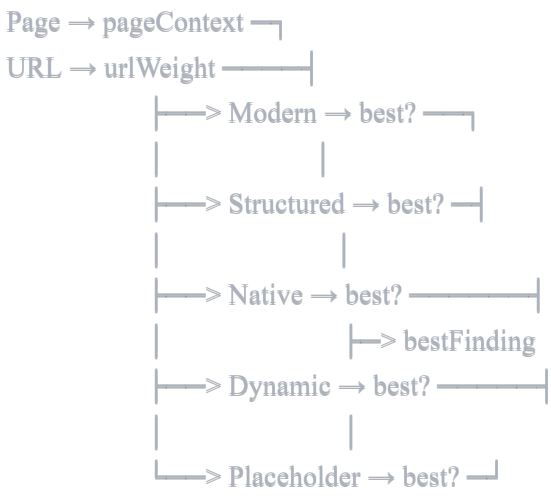
Стратегия каскадного детектирования:

```
1. Предварительный анализ:
   analyzePageStructure(doc) → pageContext
   analyzeURLContext(url) → urlWeight

2. Проверка modern systems (если maxConfidence < 0.9):
   detectModernSystems() → Finding или nil
   Update bestFinding if better

3. Проверка structured data (если maxConfidence < 0.9):
   detectStructuredData() → Finding или nil
   Update bestFinding if better

4. Проверка native forms (если maxConfidence < 0.8):
   detectNativeFormsEnhanced() → Finding или nil
   Update bestFinding if better

5. Проверка dynamic systems (если maxConfidence < 0.7):
   detectDynamicSystems() → Finding или nil
   Update bestFinding if better

6. Проверка placeholders (если maxConfidence < 0.5 AND context good):
   detectPlaceholdersEnhanced() → Finding или nil
   Update bestFinding if better

Return: bestFinding (или nil)
```

Логика "early exit":

- Если нашли modern system с confidence 0.95 - не проверяем остальное
- Экономит время на очевидных случаях

Причинно-следственная связь:



Проблемы:

- ⚠️ Пороги (0.9, 0.8, 0.7, 0.5) не обоснованы
- ⚠️ Может пропустить хорошие находки если нашел раньше
- ✅ Логика правильная и эффективная

7.3 detectModernSystems()

Строки: 1155-1270

База данных систем (40+ систем):

- Популярные: Disqus, Facebook, VK, WordPress
- Self-hosted: Commento, Utterances, Giscus, Remark42
- Региональные: Livere, Hypercomments, Yandex
- CMS-specific: Drupal, Joomla, Ghost
- Frameworks: React Comments, Vue Comments
- Другие: Telegram, Coral, Cackle, Muut

Паттерны поиска:

```
go

systems := map[string][]string{
    "disqus": {
        "#disqus_thread",
        ".disqus",
        "disqus.com/embed",
        "disqus_config",
    },
    // ... 40+ систем
}
```

Детектирование:

1. Получить HTML всей страницы
2. ToLower для case-insensitive поиска
3. Для каждой системы и каждого паттерна:
    - Проверить в HTML (strings.Contains)
    - ИЛИ найти элемент в DOM (doc.Find)
4. При первом совпадении:
    - Return Finding с confidence 0.95

Проблемы:

- ⚠️ ToLower всего HTML - тяжелая операция (1MB → 1MB copy)
- ⚠️ Может давать false positives если паттерн упомянут в тексте
- ⚠️ 200+ строк хардкода систем
- ✅ Очень эффективен для известных систем

7.4 detectNativeFormsEnhanced() - САМАЯ СЛОЖНАЯ

Строки: 1273-1432

~160 строк сложной эвристики!

Алгоритм:

```
FOR EACH <form> on page:
  1. Базовая фильтрация:
    - Пропустить если нет textarea
    - Пропустить поисковые формы

  2. Семантический анализ:
    formSemantics = analyzeFormSemantics(form, doc)

  3. Начальный score:
    score = pageContext * 0.3 +
          urlWeight * 0.2 +
          formSemantics * 0.5

  4. Анализ textarea (расширенные паттерны):
    Patterns:
      ["comment", "reply"] → +0.35
      ["message", "text"] → +0.25
      ["review", "feedback"] → +0.2
      ["post", "entry"] → +0.15
      ["description"] → +0.1

  5. Анализ других полей:
    Собрать signature:
      - hasEmail (email field or type="email")
      - hasName (name field)
      - hasWebsite (url field)
      - hasPhone (tel field)
      - hasSubject (subject field)
      - hasCompany (company field)

  6. Pattern matching полей:
    IF name + email + !phone + !subject + !company:
       score += 0.35  // Classic comment pattern
       IF website: score += 0.1

    IF (email OR name) + !phone + !subject + !company:
       score += 0.2  // Minimal comment

    IF no fields + 1 textarea:
       score += 0.15  // Anonymous comment

    IF phone: score -= 0.4  // Contact form
    IF subject: score -= 0.25
    IF company: score -= 0.3

  7. Анализ submit button:
```

```
    Patterns:
      "post comment" ⟶ +0.3
      "reply" ⟶ +0.25
      "send" ⟶ +0.1
      "contact" ⟶ -0.3

  8. Контекстные проверки:
    IF form inside <article>: score += 0.15
    IF comments section nearby: score += 0.2

  9. Complexity check:
    IF total fields ≤ 4: score += 0.1
    IF total fields > 7: score -= 0.15

  10. Добавить context signals

  11. IF score > minThreshold (0.3 или 0.25):
        Update bestFinding

RETURN: bestFinding
```

Адаптивные пороги:

```go
minThreshold := 0.3
if pageContext > 0.3 || urlWeight > 0.2 {
   minThreshold = 0.25  // Снижаем для хорошего контекста
}
```

Проблемы:

- ❌ Монолитная функция (160 строк)
- ⚠️ Множество магических чисел
- ⚠️ Сложная логика без тестов
- ✅ Но работает хорошо на практике!

Причинно-следственная связь:

```
Form fields → signature → pattern matching → score adjustment
Form context → nearby elements → score adjustment
Textarea names → pattern matching → score adjustment
Submit button → text analysis → score adjustment
Page context → weight → initial score
                    ↓
              Total score → Finding
```

7.5 detectDynamicSystems()

Строки: 1434-1464

Поиск AJAX-загрузчиков:

Паттерны в HTML:

```
- loadComments
- fetchComments
- getComments
- ajax.*comment
- comment.*ajax
- comments-container
- data-comments
- discussionUrl
```

Метод: Regex search в HTML

Return: Finding с confidence 0.7 при совпадении

Проблемы:

- ⚠️ Может давать false positives
- ⚠️ Регулярки компилируются каждый раз
- ✅ Ловит React/Vue компоненты

7.6 detectPlaceholdersEnhanced()

Строки: 1467-1520

Логика:

```
1. Требования:
   - pageContext ≥ 0.1 (минимальный контекст)
   - hasCommentArea (форма или #comments существует)

2. Поиск фраз:
   "no comments yet" → 0.5
   "be the first to comment" → 0.5
   "0 comments" → 0.4
   "leave a comment" → 0.4
   [+ русские варианты]

3. Коррекция confidence:
   baseConfidence + pageContext * 0.2
   (max 0.8)

RETURN: Finding если найдена фраза
```

Проблемы:

- ⚠️ Может ловить ссылки на другие статьи с "0 comments"
- ⚠️ Требует наличия формы, но форма может быть динамической
- ✅ Консервативный подход - хорошо

---

8. LINK EXTRACTION & PRIORITIZATION MODULE

Файл: Строки 1522-1750

Тип: Graph traversal logic

8.1 extractLinks()

Строки: 1522-1582

Логика:

```
1. Parse baseURL
2. FOR EACH <a href> on page:
   - Parse href
   - Resolve to absolute URL
   - Filter by domain (только свой домен)
   - Remove fragment (#)
   - Clean query params (utm_* и т.д.)

3. Filter by extension:
   Skip: .jpg, .pdf, .zip, .mp3, .css, .js, etc.

4. Filter by path:
   Skip: /wp-admin, /admin, /api/, /feed, etc.

5. Deduplicate (via map)
6. RETURN: []string
```

Проблемы:

- ⚠️ Может пропустить комментарии на /api/comments endpoint
- ✅ Разумные фильтры

8.2 prioritizeLinks() - СТАРАЯ ВЕРСИЯ

Строки: 1584-1633

Простая эвристика:

```
FOR EACH link:
   score = 50  // Base

   IF contains "comment"|"discuss"|"review"|"forum": +40
   IF contains "blog"|"article"|"post"|"news": +30
   IF matches /202[34]/: +20

   IF contains "contact"|"about"|"privacy"|"login": -20

SORT by score descending
RETURN sorted links
```

Проблема: Эта функция **НЕ ИСПОЛЬЗУЕТСЯ**!
Вместо нее вызывается prioritizeLinksByFreshnessEnhanced()

8.3 prioritizeLinksByFreshnessEnhanced() - ИСПОЛЬЗУЕТСЯ

Строки: 1670-1750

Сложная приоритизация с обучением:

```
FOR EACH link:
  score = 50

  1. Pattern learning:
    patternScore = scoreByPattern(link)  // 0-100
    score += patternScore

  2. Extract date from URL:
    IF /YYYY/MM/: confidence = 0.7
    IF /YYYY/: confidence = 0.5

  3. Freshness scoring:
    currentYear: +100 (+50 if last 2 months)
    1 year ago: +70
    2 years: +50
    3-5 years: +20
    6-10 years: +5
    >10 years: +0 (НЕ штрафуем старый контент!)

  4. Keyword bonus:
    "comment"|"discuss": +30
    "blog"|"article"|"post": +20

  5. Adaptive penalty (после 20 находок):
    Снижаем штраф для aggregator pages
    "/page/"|"/tag/"|"/category/": -30 → -20

  6. Fallback to page date if URL date unknown

SORT by: score → patternScore → year → month
RETURN sorted links
```

Pattern learning integration:

```
extractURLPattern(url) → pattern (обобщенный)
getPatternCount(pattern) → count успешных находок
scoreByPattern(url):
  count ≥ 10: +100
  count ≥ 5: +70
  count ≥ 2: +40
  count = 1: +20
  count = 0: +0
```

Причинно-следственная связь:

```
URL → extractURLPattern → pattern
                ↓
Successful finding → learnFromSuccess → patternCounts++
                          ↓
Future URL → scoreByPattern → high priority if pattern known
        ↓
    Higher chance to find more comments
```

Проблемы:

- ⚠️ Pattern learning имеет race condition
- ⚠️ Старый контент не штрафуется - можно упустить свежие статьи
- ⚠️ Логируется только если patternScore > 0 (debug затруднен)
- ✅ Очень умная система!

8.4 extractDateFromPage()

Строки: 1600-1668

Извлечение даты публикации:

```
Priority order:
1. Meta tags (confidence 0.9):
  - article:published_time
  - article:modified_time
  - name="publish_date"
  Parse: RFC3339, YYYY-MM-DD

2. <time datetime> (confidence 0.8):
  Parse: RFC3339, YYYY-MM-DD

3. Text in date elements (confidence 0.5-0.6):
  .date, .post-date, .entry-date, .published
  Regex patterns:
  - YYYY-MM-DD
  - MM/DD/YYYY
  - Month DD, YYYY

RETURN: year, month, confidence
```

⚠️ КРИТИЧЕСКИЙ БАГ:

```
go

doc.Find(".date").Each(func(i int, s *goquery.Selection) {
    // ...
    return  // ❌ Выходит только из Each, не из функции!
})
```

Результат: Если нашли дату в meta, потом ищем в text и перезаписываем менее надежной датой.

Правильно:

```
go

doc.Find(".date").EachWithBreak(func(i int, s *goquery.Selection) bool {
    // ...
    return false  // Break из Each и из функции
})
```

8.5 extractURLPattern()

Строки: 1650-1668

Обобщение URL для pattern learning:

```
/blog/2024/12/my-article-name-123
    ↓
/blog/{year}/{month}/my-article-name-{id}

Rules:
- /\d{4}/ → /{year}/
- /\d{1,2}/ → /{num}/
- -\d+ → -{id}
- /\d+$ → /{id}
```

Причинно-следственная связь:

```
Concrete URL → Pattern → Count successful → Priority boost for similar URLs
```

Проблемы:

- ⚠️ Может быть слишком общим (ложные срабатывания)
- ⚠️ Или слишком специфичным (не улавливает вариации)
- ✅ Баланс достигнут хорошо

8.6 learnFromSuccess()

Строки: 1670-1690

⚠ КРИТИЧЕСКИЙ БАГ - RACE CONDITION:

```
go

if count, ok := dc.patternCounts.Load(pattern); ok {
    dc.patternCounts.Store(pattern, count.(int)+1)  // ✗ RACE!
} else {
    dc.patternCounts.Store(pattern, 1)
}
```

Проблема:

```
Thread 1: Load(pattern) → 5
Thread 2: Load(pattern) → 5
Thread 1: Store(pattern, 6)
Thread 2: Store(pattern, 6)  // Lost increment!
```

Правильное решение:

```
go

// Option 1: Use atomic.Int64 in a wrapper
type atomicCounter struct {
    value atomic.Int64
}

// Option 2: LoadOrStore + CAS loop
for {
    oldVal, loaded := dc.patternCounts.LoadOrStore(pattern, &atomicCounter{})
    if !loaded {
        oldVal.(*atomicCounter).value.Store(1)
        return
    }
    counter := oldVal.(*atomicCounter)
    counter.value.Add(1)
    return
}
```

8.7 getPatternCount() & scoreByPattern()

Строки: 1692-1710

Безопасные читающие функции (но читают данные с race condition)

---

9. SITEMAP & ROBOTS.TXT MODULE

Файл: Строки 1752-1920

Тип: Discovery helpers

9.1 processSitemap()

Строки: 1752-1808

Стратегия:

```
1. Try multiple sitemap URLs:
  - /sitemap.xml
  - /sitemap_index.xml
  - /sitemap1.xml
  - /post-sitemap.xml
  - /page-sitemap.xml
  - http:// variants

2. FOR EACH sitemap:
  - Fetch (limit 5MB)
  - IF sitemapindex:
      Extract nested sitemaps → process recursively
  - ELSE:
      extractLinksFromSitemap()
  - STOP if links > 100

3. Also check robots.txt:
  processRobotsTxt()

4. Prioritize and queue (max 200 links)
```

Проблемы:

- ⚠️ 5MB limit может быть мало для больших sitemap
- ⚠️ Рекурсия без ограничения глубины
- ✅ Хорошая стратегия поиска

9.2 processSitemapURL()

Строки: 1810-1835

Вспомогательная для загрузки вложенного sitemap

9.3 extractLinksFromSitemap()

Строки: 1837-1890

XML парсинг:

```
Regex: <url>.*?<loc>([^<]+)</loc>.*?<lastmod>...?<priority>...?</url>

Extract:
- URL (required)
- lastmod (optional)
- priority (optional, default 0.5)

Sort by:
1. Priority (высокий первым)
2. Lastmod (свежие первые)

Return: sorted []string
```

Проблемы:

- ⚠️ Regex для XML - может быть ненадежно
- ⚠️ Лучше использовать xml.Decoder
- ✅ На практике работает

9.4 processRobotsTxt()

Строки: 1892-1920

Извлечение info:

```
1. Fetch /robots.txt (limit 100KB)

2. Parse line by line:
  - Find "Sitemap:" directives
    → processSitemapURL()

  - For our User-Agent (* or bot):
    Find "Allow:" directives
    → Add to allLinks
```

Проблемы:

- ⚠️ Простой парсинг (может пропустить сложные robots.txt)
- ✅ Достаточно для большинства случаев

---

10. MONITORING MODULE

Файл: Строки 1922-1985

Тип: Health checker

monitor()

Строки: 1922-1985

Задачи:

1. Отслеживать прогресс обработки домена
2. Определять условия останова
3. Предотвращать зависания

Логика:

```
ticker := 5 seconds

FOR EACH tick:
   pages = atomic pages count
   found = atomic found count
   queueLen = len(queue)

   IF queueLen == 0:
      emptyQueueCount++

      IF emptyQueueCount >= 3:
         IF pages >= 50 OR found > 0:
            cancel()  // Normal completion
         ELSE IF pages > 0:
            cancel()  // Insufficient pages
         // ⚠ ELSE: никогда не выйдет при pages == 0

   ELSE:
      emptyQueueCount = 0

   IF pages >= MaxPagesPerDomain:
      cancel()
```

⚠ КРИТИЧЕСКИЙ БАГ:

```go
if emptyQueueCount >= 3 {
    if pages >= 50 || found > 0 {
        dc.cancel()
    } else if pages > 0 {
        dc.cancel()
    }
    // Что если pages == 0? Зависнет навсегда!
}
```

Сценарий зависания:

1. seedInitialURLs() не смог загрузить главную страницу
2. sitemap не найден или пустой
3. queue пустая, pages = 0
4. emptyQueueCount растет, но никогда не выходим

Правильное решение:

```go
if emptyQueueCount >= 3 {
    log.Printf("[%s] Queue empty for too long, stopping", dc.domain)
    dc.cancel()
    return
}
```

Причинно-следственная связь:

```
Queue empty → emptyQueueCount++ → >= 3 → check conditions → cancel
Workers process → queue not empty → emptyQueueCount = 0
Pages limit → cancel
```

---

11. DOMAIN CRAWLER HELPERS

Файл: Строки 1987-2015

Тип: Utility methods

Wait(), Stats()

Строки: 1987-1995

Простые методы:

- Wait(): ждет завершения всех воркеров
- Stats(): возвращает атомарные счетчики

---

12. MAIN CRAWLER MODULE

Файл: Строки 2017-2300

Тип: Orchestration

12.1 NewMainCrawler()

Строки: 2017-2061

Инициализация:

```
1. Create root context (cancellable)

2. Open output files:
   - comments_found.jsonl (JSONL format)
   - tier1_definite_comments.txt
   - tier2_probable_comments.txt
   - tier3_possible_comments.txt
   - tier4_ambiguous.txt

3. Create MainCrawler with:
   - Config
   - results channel (buffer 1000)
   - outputFile handles
   - Empty checkpoint

4. Load checkpoint from file (if exists)

RETURN: *MainCrawler
```

Проблемы:

- ⚠️ Если open file fails, продолжает с nil handles (panic потом)
- ⚠️ Channel buffer 1000 может быть мало при высокой скорости
- ✅ Checkpoint система хорошая

12.2 loadCheckpoint()

Строки: 2063-2081

Восстановление состояния:

```
1. Read checkpoint.json
2. Unmarshal to Checkpoint struct
3. IF error: create empty checkpoint
```

Причинно-следственная связь:

```
Previous run → saveCheckpoint() → checkpoint.json
                                        ↓
New run → loadCheckpoint() → skip processed domains
                    → resume from where stopped
```

12.3 saveCheckpoint()

Строки: 2083-2096

Сохранение состояния:

```
1. Lock checkpointMu (write lock)
2. Update Statistics with atomic counters
3. Set Timestamp
4. Marshal to JSON (indented)
5. Write to file
```

Периодичность: Каждые config.CheckpointInterval (60 sec)

12.4 Run() - ГЛАВНАЯ ФУНКЦИЯ

Строки: 2098-2155

Оркестрация:

```
1. Load domains from file

2. Log startup info

3. Start background goroutines:
   - processResults() (1)
   - checkpointRoutine() (1)
   - progressReporter() (1)

4. Create semaphore (MaxTotalWorkers / WorkersPerDomain)

5. FOR EACH domain:
   - Skip if in checkpoint
   - Acquire semaphore
   - Spawn processDomain() goroutine

6. Wait for all to complete

7. Save final checkpoint

8. Print final stats
```

Управление параллелизмом:

```
semaphore := make(chan struct{}, maxConcurrentDomains)

Acquire: semaphore <- struct{}{}
Release: <-semaphore
```

Причинно-следственная связь:

```
Domains list → filter by checkpoint → spawn DomainCrawler
                                          ↓
                                    results channel
                                          ↓
                                    processResults() → write files
                                                     → update tiers
```

12.5 processDomain()

Строки: 2157-2187

Обработка одного домена:

```
1. Increment domainsActive
2. Create DomainCrawler
3. Store in sync.Map (для graceful shutdown)
4. Start crawler
5. Wait for completion
6. Get stats
7. Update global counters
8. Mark as processed in checkpoint
9. Decrement domainsActive
```

Проблемы:

- ✅ Хорошая изоляция
- ✅ Правильная синхронизация

12.6 processResults() - АГРЕГАЦИЯ

Строки: 2189-2287

Главный обработчик результатов:

```
encoder := json.NewEncoder(outputFile)
uniqueURLs := make(map[string]bool)  // ⚠️ Unbounded!
txtFile := urls_with_comments.txt

LOOP:
   SELECT:
      CASE finding := <-results:
         1. Skip if duplicate URL
         2. Validate tier (1-4)
         3. Increment tier counter
         4. Write to tier file
         5. Write to txt file (if tier ≤ 3)
         6. Write to JSONL

      CASE <-ctx.Done():
         1. Drain remaining results
         2. Process same way
         3. Log tier distribution
         4. Return
```

⚠️ КРИТИЧЕСКАЯ ПРОБЛЕМА - MEMORY LEAK:

```go
uniqueURLs := make(map[string]bool)  // Растет бесконечно!
```

Для 100k доменов × 1000 страниц = 100M entries.

При 100 байт на URL = 10GB памяти!

Решение - bounded LRU cache:

```go
type BoundedSet struct {
    items map[string]time.Time
    max   int
}

func (bs *BoundedSet) Add(url string) bool {
    if _, exists := bs.items[url]; exists {
        return false // duplicate
    }

    if len(bs.items) >= bs.max {
        // Evict oldest
        var oldest string
        var oldestTime time.Time = time.Now()
        for url, t := range bs.items {
            if t.Before(oldestTime) {
                oldest = url
                oldestTime = t
            }
        }
        delete(bs.items, oldest)
    }

    bs.items[url] = time.Now()
    return true // new
}
```

Tier distribution logging:

```
На ctx.Done() выводит финальную статистику:
- Tier 1: N URLs
- Tier 2: N URLs
- Tier 3: N URLs
- Tier 4: N URLs
- Total: N URLs
```

12.7 checkpointRoutine()

Строки: 2289-2303

Периодическое сохранение:

```
ticker := config.CheckpointInterval

LOOP:
  SELECT:
    CASE <-ticker.C:
      saveCheckpoint()
    CASE <-ctx.Done():
      return
```

Просто и эффективно ✅

12.8 progressReporter()

Строки: 2305-2320

Вывод прогресса:

```
ticker := 30 seconds

LOOP:
  SELECT:
    CASE <-ticker.C:
      printStats()
    CASE <-ctx.Done():
      return
```

12.9 printStats()

Строки: 2322-2348

Текущая статистика:

```
elapsed = time since start
domainsProgress = done / total * 100
pagesPerSec = total pages / elapsed seconds

Log:
[XX.X%] Active: N | Done: N/N |
Pages: N (X.X/s) | Pages with comments: N |
T1:N T2:N | Errors: N
```

Причинно-следственная связь:

```
Atomic counters → read every 30s → format → log
                                 → user sees progress
                                 → can estimate completion
```

12.10 printFinalStats()

Строки: 2350-2371

Итоговая статистика:

```
Success rate = domains with comments / domains done * 100

Output:
================================================
FINAL RESULTS
================================================
Runtime: Xs
Domains processed: N/N
Total pages crawled: N
Domains with comments: N (XX.X%)
Pages with comments: N
Total errors: N
Output files: (list)
================================================
```

12.11 loadDomains()

Строки: 2373-2405

Загрузка списка доменов:

```
1. Open file
2. Scanner line by line:
   - Trim whitespace
   - Skip empty lines
   - Skip comments (#)
   - Strip http://, https://, www.
   - Strip trailing /
   - ToLower()
   - Add to list
3. Return []string
```

Проблемы:

- ⚠️ Не проверяет валидность домена
- ⚠️ Может пропустить невалидные домены молча
- ✅ Простой и понятный код

12.12 Shutdown()

Строки: 2407-2421

Graceful shutdown:

```
1. Log "Shutting down..."
2. Cancel main context
3. For each DomainCrawler:
   - Cancel its context
4. Wait for all goroutines
5. Close output files
6. Save checkpoint
7. Print final stats
```

Причинно-следственная связь:

```
SIGINT/SIGTERM → Shutdown() → cancel contexts
                            → wait for goroutines
                            → save state
                            → exit cleanly
```

Проблемы:

- ⚠️ Workers могут висеть на sleep без проверки context
- ⚠️ Нет таймаута на shutdown (может висеть вечно)
- ✅ В целом правильный подход

---

13. UTILITY FUNCTIONS

Файл: Строки 2423-2440

Тип: Helpers

cleanQuery()

Строки: 2423-2437

Удаление tracking параметров:

```
Remove from URL query:
- utm_source
- utm_medium
- utm_campaign
- utm_term
- utm_content
- gclid (Google Click ID)
- fbclid (Facebook Click ID)
- yclid (Yandex Click ID)
- _ga (Google Analytics)
```

Назначение: Деупликация URL'ов

Проблемы:

- ⚠️ Может не учитывать другие tracking системы
- ✅ Покрывает основные

maxFloat()

Строки: 2439-2440

Простой helper для max(a, b)

---

14. MAIN FUNCTION

Файл: Строки 2442-2502

Тип: Entry point

main()

Логика:

```
1. Set GOMAXPROCS to use all CPUs

2. Parse flags:
   -domains (required)
   -workers (default: 3)
   -max-workers (default: 300)
   -max-pages (default: 1000)
   -output (default: comments_found.jsonl)
   -checkpoint (default: checkpoint.json)

3. Validate: domains file required

4. Create config from flags + defaults

5. Create MainCrawler

6. Setup signal handler:
   SIGINT/SIGTERM → Shutdown() → os.Exit(0)

7. Run crawler:
   crawler.Run(domainFile)

8. Exit
```

Причинно-следственная связь:

```
main() → parse flags → create config → create crawler
                              ↓
                    signal handler → graceful shutdown
                         ↓
                    Run() → spawn workers → process domains
                                  → write results
```

Проблемы:

- ⚠️ os.Exit(0) в signal handler - не дает defer'ам отработать
- ⚠️ Нет логирования в файл
- ⚠️ Нет версии в выводе
- ✅ Базовая функциональность правильная

🔗 КРИТИЧЕСКИЕ ЗАВИСИМОСТИ МЕЖДУ МОДУЛЯМИ

Граф зависимостей (упрощенный):

```
main()
  └─> MainCrawler
      ├─> Config
      ├─> Checkpoint
      ├─> DomainCrawler (N instances)
      │   ├─> Config
      │   ├─> HTTP Client
      │   ├─> Rate Limiter
      │   ├─> Detection Module:
      │   │   ├─> Modern Systems
      │   │   ├─> Structured Data
      │   │   ├─> Native Forms
      │   │   │   └─> Form Semantics
      │   │   ├─> Dynamic Systems
      │   │   └─> Placeholders
      │   ├─> Context Analysis:
      │   │   ├─> URL Context
      │   │   ├─> Page Structure
      │   │   └─> Form Semantics
      │   ├─> Tier Classification
      │   ├─> Link Extraction
      │   ├─> Prioritization:
      │   │   ├─> Pattern Learning
      │   │   └─> Freshness
      │   └─> Sitemap Processing
      ├─> Results Processing
      ├─> Checkpoint Routine
      └─> Progress Reporter
```

Потоки данных:

```
Domain → DomainCrawler → fetch → parse → detect → classify
                                    ↓
                                 Finding
                                    ↓
                              results channel
                                    ↓
                             processResults()
                                    ↓
                    ┌───────────────┴───────────────┐
                    ↓                               ↓
              Write to files                  Update metrics
              (JSONL + tier txt)              (atomic counters)
```

Критические точки синхронизации:

1. **results channel** - single point of collection
2. **sync.Map visited** - concurrent access
3. **sync.Map patternCounts** - RACE CONDITION ⚠️
4. **Atomic counters** - thread-safe
5. **checkpointMu** - RWLock для checkpoint

📈 МЕТРИКИ И МОНИТОРИНГ

Собираемые метрики:

Per Domain:

- pages (обработано страниц)
- found (найдено комментариев)
- errors (всего ошибок)
  - timeoutErrors
  - networkErrors
  - parseErrors
  - notFoundErrors

Global:

- domainsTotal
- domainsActive
- domainsDone
- domainsWithComments
- pagesTotal
- findingsTotal
- errorsTotal
- tierCounts[1-4]

Pattern Learning:

- successPatterns (map[url]bool)
- patternCounts (map[pattern]int) ⚠️ Race condition

Недостающие метрики:

❌ **Latency metrics** (время обработки страницы)
❌ **Memory usage** (текущее потребление)
❌ **Queue depth over time** (динамика очереди)
❌ **False positive rate** (качество детектирования)
❌ **Cache hit rate** (если бы был кэш)

🎯 КЛЮЧЕВЫЕ ИНСАЙТЫ

Что работает хорошо:

1. ✅ **Cascading detection** - умная приоритизация методов
2. ✅ **Pattern learning** - адаптация на основе успеха
3. ✅ **Context-aware scoring** - учет URL + Page + Form
4. ✅ **Tier classification** - четкое разделение уверенности
5. ✅ **Checkpoint system** - устойчивость к сбоям
6. ✅ **Rate limiting** - защита от блокировок
7. ✅ **Sitemap discovery** - максимальный охват

Что требует исправления:

1. ❌ **Race conditions** в pattern learning
2. ❌ **Goroutine leaks** при panic recovery
3. ❌ **Context ignoring** при retry sleep
4. ❌ **Unbounded memory** в uniqueURLs map
5. ❌ **Early return bug** в extractDateFromPage
6. ❌ **Monitor hang** при pages == 0
7. ❌ **Unsafe type assertions** везде

Что можно оптимизировать:

1. ⚠️ **Regex compilation** - предкомпилировать
2. ⚠️ **DOM traversal** - единый проход
3. ⚠️ **String operations** - использовать strings.Builder
4. ⚠️ **ToLower HTML** - избегать больших копий
5. ⚠️ **Duplicate code** - выделить общие паттерны

📝 ВЫВОДЫ

Общая оценка архитектуры: 7/10

Сильные стороны:

- Модульная структура
- Хорошее разделение ответственностей
- Умная логика детектирования
- Продуманная приоритизация

Слабые стороны:

- Критические баги в concurrency
- Отсутствие абстракций (интерфейсов)
- Монолитные функции
- Хардкод конфигураций
- Нет тестов

Рекомендации:

1. **Немедленно** исправить критические баги
2. **Добавить** bounded collections
3. **Рефакторить** монолитные функции
4. **Выделить** интерфейсы детекторов
5. **Добавить** comprehensive тесты
6. **Оптимизировать** hot paths
7. **Документировать** магические числа

Конец архитектурной карты

Продолжение в документах:

- `02_PROBLEMS_MATRIX.md` - детальная матрица проблем
- `03_REFACTORING_PLAN.md` - пошаговый план исправлений