Web Crawler Project

Learning Objectives

Build a production-ready web crawler to learn:

- Bounded parallelism with worker pools
- Domain-specific rate limiting (politeness)
- Deduplication with sync.Map
- robots.txt respect
- Graceful cancellation with context

Requirements

Functional Requirements

1. **URL Fetching**
    - Download web pages concurrently
    - Parse HTML for links
    - Follow links to specified depth
2. **Politeness**
    - Respect robots.txt rules
    - Rate limit per domain (1 req/sec)
    - Configurable User-Agent
    - Delay between requests to same domain
3. **Deduplication**
    - Track visited URLs (no duplicates)
    - Thread-safe visited set
    - Canonical URL normalization
4. **Concurrency Control**
    - Bounded workers (prevent resource exhaustion)
    - Configurable max concurrent fetches
    - Graceful shutdown (wait for in-flight)

Non-Functional Requirements

1. **Performance**
    - Crawl 1000+ pages/minute
    - Efficient memory usage
    - Minimal blocking
2. **Correctness**
    - No race conditions
    - All pages visited exactly once
    - Respects crawl depth limit
3. **Observability**

    - Pages fetched, errors, duplicates
    - Crawl progress tracking
    - Per-domain request counts

Three Implementations

1. Naive Implementation (naive/crawler.go)

Approach:

- Unbounded goroutines (one per URL)
- No politeness (hammers servers)
- Simple map for visited (race conditions)
- No robots.txt support

Problems (Intentional):

- **Resource exhaustion:** 10,000 URLs = 10,000 goroutines
- **Rate problems:** Hundreds of requests/sec to same domain
- **Race conditions:** Concurrent map access panics
- **Poor citizenship:** Ignores robots.txt, overloads servers

Expected Issues:

- OOM with large crawls
- IP banned for aggressive crawling
- Panics from concurrent map writes
- Wasted bandwidth on duplicates

2. Improved Implementation (improved/crawler.go)

Fixes:

- **Worker pool:** Fixed number of fetchers (bound parallelism)
- **sync.Map:** Thread-safe visited tracking
- **Per-domain queues:** Rate limit each domain independently
- **Context cancellation:** Graceful shutdown

Improvements:

- Memory bounded (fixed worker count)
- Concurrent-safe (sync.Map)
- Polite crawling (rate limiting)
- Can cancel mid-crawl

Remaining Issues:

- No robots.txt parsing
- Fixed rate limit (should be configurable per domain)
- No retry logic for transient failures
- Basic metrics only

3. Final Implementation (final/crawler.go)

Production-Ready:

- **Robots.txt support:** Parse and respect rules

- **Adaptive rate limiting:** Honor Crawl-Delay directive
- **Pipeline architecture:** Fetch → Parse → Queue (3 stages)
- **Comprehensive metrics:** Per-domain stats
- **Retry logic:** Exponential backoff for failures
- **Context tree:** Cascade cancellation properly

Key Features:

```
crawler := final.NewCrawler(final.Config{
    Workers:    20,
    MaxDepth:   3,
    UserAgent:  "MyBot/1.0",
    Politeness: &final.PolitenessConfig{
        DefaultDelay:   time.Second,
        RespectRobots:  true,
        MaxPerDomain:   10,
    },
})

results := crawler.Crawl(ctx, "https://example.com")
for result := range results {
    fmt.Printf("Fetched: %s (depth %d)\n", result.URL, result.Depth)
}
```

Performance:

- Throughput: ~60 pages/sec (20 workers, avg 300ms/page)
- Memory: Fixed overhead + visited set (~100 bytes/URL)
- Politeness: Configurable per-domain delays

Success Criteria

1. **Understand bounded parallelism:**
    - Why worker pool vs unbounded goroutines?
    - How to choose worker count?
    - When does more workers hurt performance?
2. **Master politeness:**
    - Why respect robots.txt?
    - How to rate limit per domain?
    - Ethical crawling practices
3. **Handle deduplication:**
    - Canonical URL normalization
    - Thread-safe visited set
    - Memory vs accuracy trade-offs
4. **Implement pipeline:**
    - Fetch workers (I/O-bound, many)
    - Parse workers (CPU-bound, NumCPU)

- Queue coordinator (single)

Usage Examples

Naive Implementation

```
crawler := naive.NewCrawler()
urls := crawler.Crawl("https://example.com", 2) // depth 2
for url := range urls {
    fmt.Println(url)
}
```

Final Implementation

```
ctx, cancel := context.WithTimeout(context.Background(), 5*time.Minute)
defer cancel()

crawler := final.NewCrawler(final.Config{
    Workers:   20,
    MaxDepth:  3,
    UserAgent: "MyBot/1.0 (+https://mysite.com/bot)",
    Politeness: &final.PolitenessConfig{
        RespectRobots: true,
        DefaultDelay:  time.Second,
        MaxPerDomain:  10, // Max concurrent requests per domain
    },
})

results := crawler.Crawl(ctx, "https://example.com")
fetched := 0
for result := range results {
    if result.Error != nil {
        log.Printf("Error fetching %s: %v", result.URL, result.Error)
        continue
    }
    fetched++
    fmt.Printf("[Depth %d] %s (%d links)\n",
        result.Depth, result.URL, len(result.Links))
}

stats := crawler.Stats()
fmt.Printf("Fetched: %d, Errors: %d, Duplicates: %d\n",
    stats.Fetched, stats.Errors, stats.Duplicates)
```

Testing Strategy

Unit Tests

```
cd final
go test -v
go test -race
go test -cover
```

Test Cases

1. **TestBoundedWorkers** - Verify worker count never exceeds limit
2. **TestDedup** - Same URL not fetched twice
3. **TestDepthLimit** - Respects max depth
4. **TestPoliteness** - Rate limiting per domain
5. **TestRobotsTxt** - Respects disallow rules
6. **TestCancellation** - Context cancel stops crawl
7. **TestRetry** - Retries transient failures

Real-World Scenarios

1. Sitemap Generator

```
// Crawl own website to generate sitemap
crawler := final.NewCrawler(final.Config{
    Workers:  10,
    MaxDepth: 5,
    Politeness: &final.PolitenessConfig{
        RespectRobots: false, // Own site
        DefaultDelay:  100 * time.Millisecond,
    },
})

var urls []string
results := crawler.Crawl(ctx, "https://mysite.com")
for result := range results {
    if result.Error == nil {
        urls = append(urls, result.URL)
    }
}

generateSitemap(urls)
```

2. Link Checker

```
// Check for broken links on website
crawler := final.NewCrawler(final.Config{
    Workers:  20,
    MaxDepth: -1, // No depth limit
})

broken := []string{}
results := crawler.Crawl(ctx, "https://mysite.com")
```

```
for result := range results {
    if result.StatusCode >= 400 {
        broken = append(broken, result.URL)
    }
}

fmt.Printf("Found %d broken links\n", len(broken))
```

3. News Aggregator

```
// Crawl multiple news sites
sites := []string{
    "https://news1.com",
    "https://news2.com",
    "https://news3.com",
}

crawler := final.NewCrawler(final.Config{
    Workers:  50,
    MaxDepth: 2,
    Politeness: &final.PolitenessConfig{
        RespectRobots: true,
        DefaultDelay:  2 * time.Second,
    },
})

for _, site := range sites {
    go func(url string) {
        results := crawler.Crawl(ctx, url)
        for result := range results {
            if isArticle(result) {
                saveArticle(result)
            }
        }
    }(site)
}
```

Common Pitfalls

Pitfall 1: Unbounded Goroutines

```
// ❌ WRONG: One goroutine per URL
for _, url := range urls {
    go fetch(url) // 10,000 URLs = 10,000 goroutines = OOM
}

// ✅ CORRECT: Bounded worker pool
for i := 0; i < 20; i++ {
```

```
    go worker(urlQueue)
}
```

Pitfall 2: Hammering Servers

```
// ❌ WRONG: No rate limiting
for _, url := range urls {
    fetch(url) // 100 requests/sec to same domain
}

// ✅ CORRECT: Per-domain rate limiting
limiter := rate.NewLimiter(rate.Every(time.Second), 1)
limiter.Wait(ctx)
fetch(url)
```

Pitfall 3: Concurrent Map Panic

```
// ❌ WRONG: Regular map with concurrent access
visited := make(map[string]bool)
go func() { visited[url] = true }() // Panic!

// ✅ CORRECT: sync.Map or mutex-protected map
visited := &sync.Map{}
visited.Store(url, true)
```

Pitfall 4: Ignoring robots.txt

```
// ❌ WRONG: Ignore robots.txt
fetch("https://example.com/admin") // Might be disallowed

// ✅ CORRECT: Check robots.txt first
robots := parseRobotsTxt("https://example.com/robots.txt")
if !robots.Allowed("/admin") {
    return // Respect disallow
}
```

Interview Discussion Points

1. **Why worker pool over unbounded goroutines?**
    - Bounded resources (memory, file descriptors)
    - Predictable behavior
    - Avoid overwhelming target servers
2. **How to rate limit per domain?**
    - Option 1: sync.Map of domain → rate.Limiter
    - Option 2: Separate queue per domain

- Option 3: Token bucket with domain key

3. **Why respect robots.txt?**
   - Ethical crawling
   - Legal requirements (some jurisdictions)
   - Avoid IP bans
   - Reduce server load
4. **How to deduplicate efficiently?**
   - sync.Map for concurrent access
   - Bloom filter for memory efficiency (false positives OK)
   - Normalize URLs (http vs https, trailing slash, query params)
5. **Pipeline vs single-stage?**
   - Fetch (I/O-bound) → many workers
   - Parse (CPU-bound) → NumCPU workers
   - Queue (coordination) → single goroutine
   - Separates concerns, better performance

Next Steps

After mastering web-crawler:

1. **connection-pool/** - Database pool with circuit breaker
2. **pub-sub/** - Topic-based messaging with fan-out

Build ethical, scalable web crawlers! 🕷️