

You said:

```
how to scrape url var urls = []string{ "https://en.wikipedia.org/wiki/Robotics",
"https://en.wikipedia.org/wiki/Robot", "https://en.wikipedia.org/wiki/Reinforcement_learning",
"https://en.wikipedia.org/wiki/Robot_Operating_System",
"https://en.wikipedia.org/wiki/Intelligent_agent",
"https://en.wikipedia.org/wiki/Software_agent",
"https://en.wikipedia.org/wiki/Robotic_process_automation",
"https://en.wikipedia.org/wiki/Chatbot",
"https://en.wikipedia.org/wiki/Applications_of_artificial_intelligence",
"https://en.wikipedia.org/wiki/Android_(robot)", } in go and make a jl file out of it with Go programming
```

ChatGPT said:

You can use Go's `net/http` package to fetch web pages and the `goquery` package to parse and extract content. Here's how you can scrape the Wikipedia pages from your `urls` list, extract their text content, and save the results in a `.jl` (JSON Lines) file.

Steps:

1. Fetch each URL's content using `http.Get`.
2. Parse the HTML using `goquery` to extract the main text.
3. Store the extracted text in a struct and write it to a `.jl` file in JSON Lines format.

Install Dependencies:

Make sure you have `goquery` installed:

```
sh
```

CopyEdit

```
go get github.com/PuerkitoBio/goquery
```

Go Code:

```
go
```

CopyEdit

```
package main import ( "encoding/json" "fmt" "log" "os" "strings" "github.com/PuerkitoBio/goquery"
"net/http" ) type PageData struct { URL string `json:"url"` Title string `json:"title"` Text string `json:"text" }
func scrapePage(url string) (*PageData, error) { // Fetch the webpage resp, err := http.Get(url) if err != nil {
return nil, fmt.Errorf("failed to fetch URL %s: %w", url, err) } defer resp.Body.Close() // Check for HTTP
errors if resp.StatusCode != 200 { return nil, fmt.Errorf("error fetching page: %s returned status code %d",
url, resp.StatusCode) } // Parse the page with goquery doc, err :=
goquery.NewDocumentFromReader(resp.Body) if err != nil { return nil, fmt.Errorf("failed to parse HTML:
%w", err) } // Extract the title title := doc.Find("h1").First().Text() // Extract the main content text var
textContent []string doc.Find("p").Each(func(i int, s *goquery.Selection) { textContent = append(textContent,
strings.TrimSpace(s.Text())) }) // Combine extracted paragraphs into a single text block fullText :=
strings.Join(textContent, " ") return &PageData{ URL: url, Title: title, Text: fullText, }, nil } func main() { urls :=
[]string{ "https://en.wikipedia.org/wiki/Robotics", "https://en.wikipedia.org/wiki/Robot",
"https://en.wikipedia.org/wiki/Reinforcement_learning",
"https://en.wikipedia.org/wiki/Robot_Operating_System",
"https://en.wikipedia.org/wiki/Intelligent_agent", "https://en.wikipedia.org/wiki/Software_agent",
"https://en.wikipedia.org/wiki/Robotic_process_automation", "https://en.wikipedia.org/wiki/Chatbot",
"https://en.wikipedia.org/wiki/Applications_of_artificial_intelligence",
"https://en.wikipedia.org/wiki/Android_(robot)", } // Open the .jl file for writing file, err :=
os.Create("scraped_data.jl") if err != nil { log.Fatalf("Failed to create file: %v", err) } defer file.Close() // Scrape
each URL and write to the .jl file for _, url := range urls { fmt.Printf("Scraping: %s\n", url) pageData, err :=
scrapePage(url) if err != nil { log.Printf("Skipping URL due to error: %v\n", err) continue } // Convert struct to
JSON jsonData, err := json.Marshal(pageData) if err != nil { log.Printf("Failed to encode JSON for %s: %v\n",
url, err) continue } // Write JSON to file as a new line _, err = file.WriteString(string(jsonData) + "\n") if err !=
nil { log.Printf("Failed to write to file: %v\n", err) continue } } fmt.Println("Scraping completed. Data saved in
scraped_data.jl") }
```

Explanation:

- `scrapePage(url)`: Fetches the page, extracts the title and main text content, and returns it as a `PageData` struct.
- `main()`:
 - Loops through the URLs.
 - Scrapes each page and converts the extracted data into JSON.
 - Writes each JSON object to a `.jl` file (one JSON object per line).

Output Format (`scraped_data.jl`):

Each line in the file will contain a JSON object:

json

CopyEdit

```
{"url":"https://en.wikipedia.org/wiki/Robotics","title":"Robotics","text":"Robotics is a field of engineering ..."} {"url":"https://en.wikipedia.org/wiki/Robot","title":"Robot","text":"A robot is a machine ..."} ...
```

This format allows easy parsing with JSON tools or scripts.

Let me know if you need modifications or explanations! 🚀

You said:

use sync.WaitGroup function

ChatGPT said:

Using `sync.WaitGroup`, we can parallelize the scraping process by launching goroutines for each URL. This improves efficiency by making multiple HTTP requests concurrently rather than sequentially.

Steps to Use sync.WaitGroup:

1. Initialize a `sync.WaitGroup`.
2. Launch a goroutine for each URL.
3. Each goroutine scrapes the page and writes to the `.json` file.
4. Use a `sync.Mutex` to ensure safe concurrent writes to the file.
5. Wait for all goroutines to finish before exiting.

Updated Go Code with Concurrency:

go

CopyEdit

```
package main import ( "encoding/json" "fmt" "log" "os" "strings" "sync" "github.com/PuerkitoBio/goquery" "net/http" ) type PageData struct { URL string `json:"url"` Title string `json:"title"` Text string `json:"text" } // Function to scrape a single page func scrapePage(url string) (*PageData, error) { resp, err := http.Get(url) if err != nil { return nil, fmt.Errorf("failed to fetch URL %s: %w", url, err) } defer resp.Body.Close() if
```

```
resp.StatusCode != 200 { return nil, fmt.Errorf("error fetching page: %s returned status code %d", url,
resp.StatusCode) } doc, err := goquery.NewDocumentFromReader(resp.Body) if err != nil { return nil,
fmt.Errorf("failed to parse HTML: %w", err) } title := doc.Find("h1").First().Text() var textContent []string
doc.Find("p").Each(func(i int, s *goquery.Selection) { textContent = append(textContent,
strings.TrimSpace(s.Text())) }) fullText := strings.Join(textContent, " ") return &PageData{ URL: url, Title:
title, Text: fullText, }, nil } func main() { urls := []string{ "https://en.wikipedia.org/wiki/Robotics",
"https://en.wikipedia.org/wiki/Robot", "https://en.wikipedia.org/wiki/Reinforcement_learning",
"https://en.wikipedia.org/wiki/Robot_Operating_System",
"https://en.wikipedia.org/wiki/Intelligent_agent", "https://en.wikipedia.org/wiki/Software_agent",
"https://en.wikipedia.org/wiki/Robotic_process_automation", "https://en.wikipedia.org/wiki/Chatbot",
"https://en.wikipedia.org/wiki/Applications_of_artificial_intelligence",
"https://en.wikipedia.org/wiki/Android_(robot)", } var wg sync.WaitGroup var mutex sync.Mutex // Open
the .jl file for writing file, err := os.Create("scraped_data.jl") if err != nil { log.Fatalf("Failed to create file: %v",
err) } defer file.Close() // Start scraping in parallel for _, url := range urls { wg.Add(1) // Increment WaitGroup
counter go func(url string) { defer wg.Done() // Decrement counter when goroutine completes
fmt.Printf("Scraping: %s\n", url) pageData, err := scrapePage(url) if err != nil { log.Printf("Skipping URL due
to error: %v\n", err) return } jsonData, err := json.Marshal(pageData) if err != nil { log.Printf("Failed to
encode JSON for %s: %v\n", url, err) return } // Ensure only one goroutine writes to the file at a time
mutex.Lock() _, err = file.WriteString(string(jsonData) + "\n") mutex.Unlock() if err != nil { log.Printf("Failed
to write to file: %v\n", err) } }(url) // Pass URL as argument to avoid closure issues } wg.Wait() // Wait for all
goroutines to finish fmt.Println("Scraping completed. Data saved in scraped_data.jl") }
```

Key Changes:

- ✅ **Added sync.WaitGroup:** Allows multiple goroutines to run concurrently and ensures the program waits for all scraping operations to finish.
- ✅ **Used sync.Mutex:** Ensures only one goroutine writes to the `.jl` file at a time, preventing race conditions.
- ✅ **Improved Error Handling:** Each goroutine logs errors individually instead of stopping the whole process.

Why This is Better?

- **Faster Execution:** Scrapes all URLs concurrently instead of one by one.
- **Safe File Writing:** Uses a `sync.Mutex` to prevent data corruption.
- **Scalability:** Can handle a larger list of URLs efficiently.

Let me know if you need further tweaks! 🚀