Hitchhiker's Guide to Golang Development

Huseyin BABAL

Lead Cloud Engineer @ namecheap

About me

- Software Development since 2007
- Currently working for Namecheap as cloud engineer, responsible for thousands of servers as a team
- You can see me writing code, reading boot, doing mock interviews on Twitch (huseyinbabal)
- More 👇

What is Go?

Fast and Efficient

Go is known for its speed and efficiency, making it ideal for performance-critical applications.

Simple and Concise

Go's syntax is clean and straightforward, promoting code readability and maintainability.

Concurrency-Focused

Go's built-in concurrency features allow for efficient handling of parallel tasks, making it suitable for complex applications.

Cross-Platform Compatibility

Go compiles to native executables, enabling it to run seamlessly on various operating systems and architectures.

Go's history

1

Go's Public Announcement (2009)

Go was publicly announced in 2009, introducing a new systems programming language focused on simplicity, concurrency, and performance.

2

Go 1.0 (2012)

The first major release of Go, establishing its core features like garbage collection, concurrency support, and a robust standard library.

3

Go 1.11 (2018)

This release introduced experimental support for Go modules, a significant step towards improving dependency management and code organization.

4

Go 1.18 (2022)

A landmark release featuring the introduction of generics, bringing enhanced type safety and code reusability to Go.

5

Go 1.20 (2023)

Go 1.20 also included improvements to error handling, allowing for more concise and informative error messages, which aids in debugging and troubleshooting.

Building a web crawler app with Go

1

Fetch URL

Retrieve the HTML content of a given web page.

2

Parse HTML

Extract relevant data, such as links, text, and images.

3

Store Data

Save the extracted data in a database or other storage mechanism.

4

Process Data

Analyze and manipulate the collected data based on the specific requirements of the web crawler.

```go
func crawl(url string) {
    resp, err := http.Get(url)
    if err != nil {
        fmt.Println("Error fetching URL:", err)
        return
    }
    defer resp.Body.Close()

    tokenizer := html.NewTokenizer(resp.Body)
    for {
        tokenType := tokenizer.Next()
        if tokenType == html.ErrorToken {
            break
        }

        token := tokenizer.Token()
        if tokenType == html.StartTagToken && token.Data == "a" {
            for _, attr := range token.Attr {
                if attr.Key == "href" && strings.HasPrefix(attr.Val, "http")
{                   fmt.Println("Found link:", attr.Val)
                }
            }
        } else if tokenType == html.TextToken {
            fmt.Println("Text:", strings.TrimSpace(token.Data))
        }
    }
}

func main() {
    url := "https://example.com"
    crawl(url)
}
```

Async web crawling with Go's concurrency

1

Go Routines

Lightweight threads that allow for concurrent execution of multiple tasks.

2

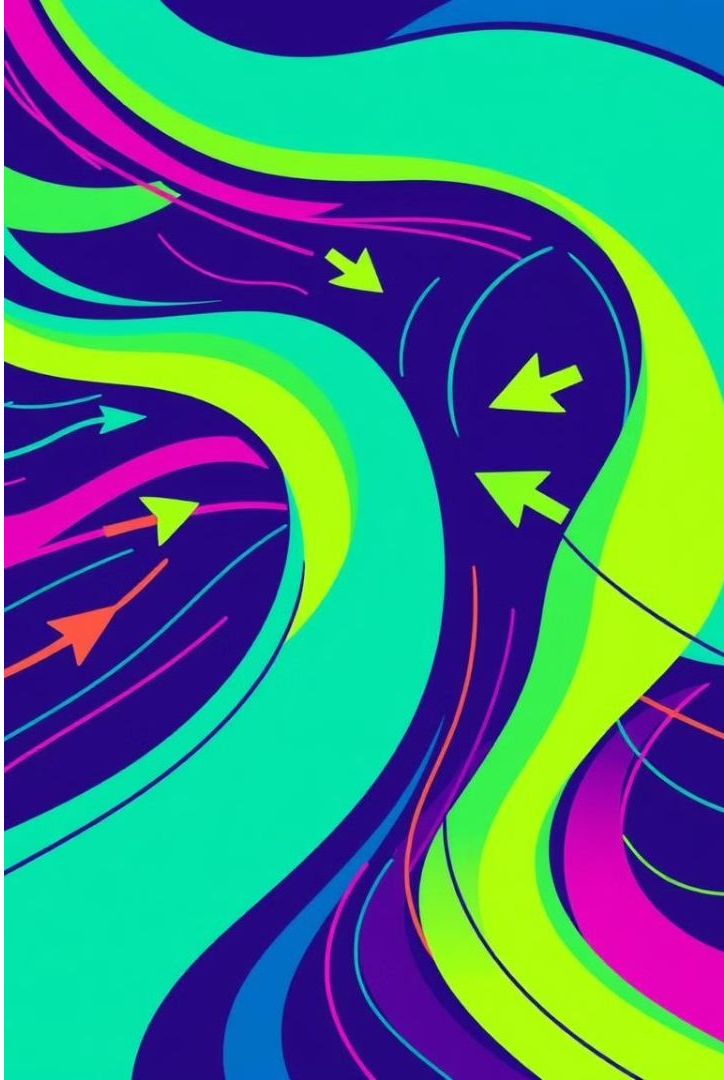
Channels

Communication channels between goroutines, enabling data exchange and synchronization.

3

Simplicity of `go` keyword

The `go` keyword simplifies the process of launching concurrent tasks, promoting code readability and maintainability.



```go
func main() {
    urls := []string{
        "https://example.com",
        "https://example.net",
        "https://example.org",
    }

    for _, url := range urls {
        go crawl(url)
    }
}
```

Concurrent Web Crawling with Go

```go
func crawl(url string, wg *sync.WaitGroup) {
  defer wg.Done()
  .....
}

func main() {
    urls := []string{
        "https://example.com",
        "https://example.net",
        "https://example.org",
    }

    var wg sync.WaitGroup
    for _, url := range urls {
        wg.Add(1)
        go crawl(url, &wg)
    }
    wg.Wait()
}
```

Converting Crawl Function to a REST Service

```go
func crawlHandler(w http.ResponseWriter, r *http.Request) {
    url := r.URL.Query().Get("url")
    if url == "" {
        http.Error(w, "URL parameter 'url' is required", http.StatusBadRequest)
        return
    }

    links, err := crawl(url)
    if err != nil {
        http.Error(w, fmt.Sprintf("Error crawling URL: %v", err), http.StatusInternalServerError)
        return
    }

    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(map[string]interface{}{"links": links})
}


func main() {
    http.HandleFunc("/crawl", crawlHandler)
    fmt.Println("Server listening on :8080")
    http.ListenAndServe(":8080", nil)
}
```

More on web frameworks

Fiber: A fast and minimalist web framework for building efficient and scalable web applications in Go, known for its simplicity and performance.

Gin: A high-performance, feature-rich web framework that provides a smooth and efficient development experience, with a focus on developer productivity.

Gorilla Mux: A powerful and flexible HTTP router and URL matcher for building web services in Go, offering advanced routing capabilities and middleware support.

Building Go executables

Compile and Build

Use the `go build` command to compile your Go code and create a standalone executable file.

Executable File

The resulting executable file can be run directly on any system that supports Go, without the need for a Go runtime environment.

Cross-Compilation

Go allows for cross-compilation, enabling you to build executables for different operating systems and architectures, even if your development machine is different.

OS/architecture support for Go

Operating Systems

- aix
- android
- darwin
- dragonfly
- freebsd
- illumos
- ios
- js
- linux
- netbsd
- openbsd
- plan9
- solaris
- windows

Architectures

- 386
- amd64
- arm
- arm64
- mips
- mips64
- mips64le
- mipsle
- ppc64
- ppc64le
- riscv64
- s390x
- wasm

Packaging and distributing your Go app

Github Action Matrix Build

```yaml
name: Build and Push Docker Image

on:
  push:
    branches:
      - main

jobs:
  build-and-push:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        go-version: [1.18, 1.19]
        os: [ubuntu-latest, macos-latest, windows-latest]

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Go
        uses: actions/setup-go@v3
        with:
          go-version: ${{ matrix.go-version }}

      - name: Build Docker image
        run: docker build -t your-docker-registry/your-image:${{ matrix.os }}-go${{ matrix.go-version }} .

      - name: Log in to Docker registry
        run: echo "${{ secrets.DOCKER_PASSWORD }}" | docker login your-docker-registry -u ${{ secrets.DOCKER_USERNAME }} --password-stdin

      - name: Push Docker image
        run: docker push your-docker-registry/your-image:${{ matrix.os }}-go${{ matrix.go-version }}
```

```dockerfile
FROM golang:alpine AS builder

WORKDIR /app

COPY go.mod ./
COPY go.sum ./

RUN go mod download

COPY . .

RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o main .


FROM alpine:latest

WORKDIR /app

COPY --from=builder /app/main .


EXPOSE 8080

CMD ["/app/main]
```

Testing

1 Unit Tests

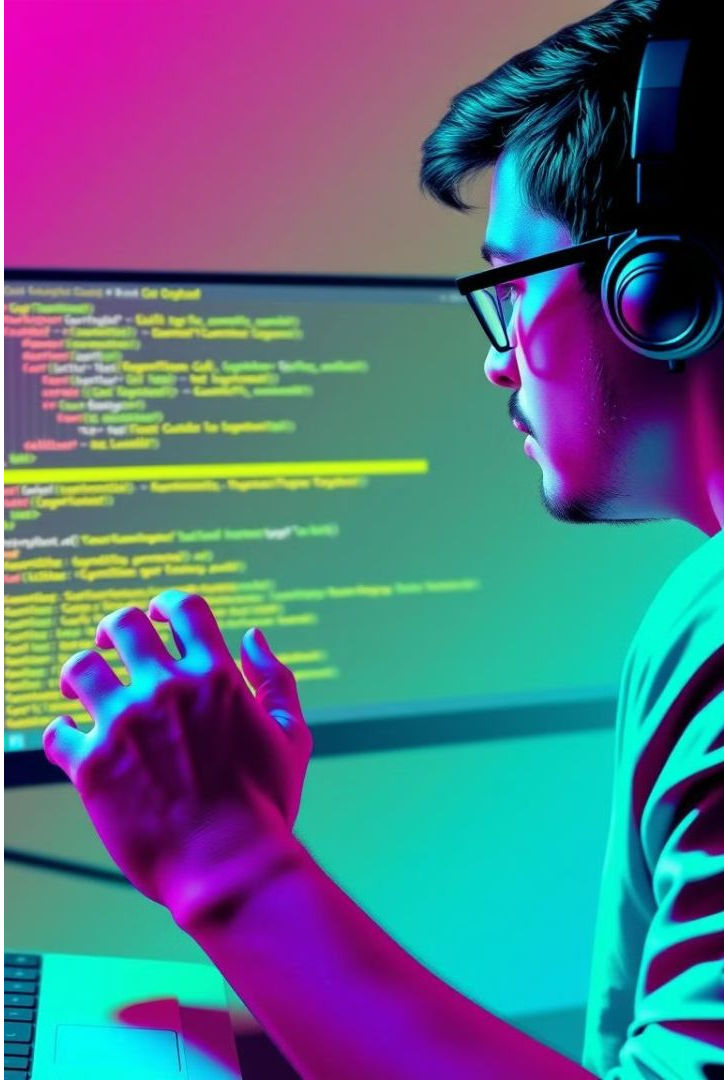
Write tests for individual functions or components, ensuring that each piece of code works as expected.

2 Integration Tests

Test how different parts of your application interact with each other, validating the overall functionality.

3 Go's Testing Framework

Go's built-in testing framework provides tools for writing, running, and reporting tests, simplifying the testing process.



```go
package main

func TestCrawl(t *testing.T) {
    testCases := []struct {
        url string
                expected string
    }{
        {url: "https://example.com", expected: "Hello world"},
        {url: "https://www.google.com", expected: "Search"},
    }

    for _, testCase := range testCases {
        if crawl(testCase.url) != testCase.expected {
                    t.Errorf("Failed for %s", testCase.url)
                }
    }
}
```

More on testing

TestContainers: A powerful testing framework for creating and managing Docker containers, enabling integration testing of your Go applications in a realistic environment.

Mockery: A mocking library that allows you to create mock implementations of your Go interfaces, simplifying the testing of complex dependencies.

Static Code Analysis with golangcilint

Automated Code Linting: golangcilint is a powerful static code analysis tool that automatically checks Go code for common programming errors, stylistic issues, and potential bugs.

Comprehensive Checks: golangcilint runs a wide range of linters, ensuring your code adheres to best practices and maintainability standards.

Continuous Integration: Integrate golangcilint into your CI/CD pipeline to catch issues early and maintain code quality throughout the development lifecycle.

Customizable Configuration: Easily configure golangcilint to fit your project's needs, enabling seamless integration with your development workflow.

```yaml
linters-settings:
  errcheck:
    check-blank: true
  govet:
    check-shadowing: true
  golint:
    min-confidence: 0
  godox:
    keywords:
      - BUG
      - FIXME
      - HACK
  gofmt:
    simplify: true

linters:
  enable-all: true
  disable:
    - dupl
    - gocyclo
    - gocognit
    - lll
    - interfacer
    - maligned
    - scopelint
    - structcheck
    - varcheck
    - wsl

issues:
  exclude-rules:
    - path: _test\.go
      linters:
        - errcheck
        - gosec
run:
  issues-exit-code: 1
  timeout: 10m

output:
  format: colored-line-number # colored-line-number|line-number|tab|checkstyle|code-climate
```

Performance Analysis with pprof in Go

1

Profiling with pprof

`pprof` is a powerful tool for analyzing CPU, memory, and other performance characteristics of your Go applications. It helps identify bottlenecks and optimize resource usage.

2

CPU Profiling

Identify functions consuming the most CPU time using `go tool pprof cpu.prof`. This helps pinpoint areas for optimization to improve overall application speed.

3

Memory Profiling

Detect memory leaks and excessive memory allocation using `go tool pprof heap.prof`. Optimize memory management to reduce resource consumption and enhance performance.

4

Web Interface

Visualize profiles in a web browser using `go tool pprof -http=:8080 cpu.prof`. The interactive interface aids in exploring call graphs and identifying performance bottlenecks.

Profiling Go Applications with pprof

1

Enable pprof Endpoint

Import the **net/http/pprof** package to enable the pprof debugging endpoints in your Go application.

2

Collect Profiles

Use **curl** to fetch the relevant profiles from the **/debug/pprof/** endpoint, such as **cpu.prof** and **heap.prof**.

3

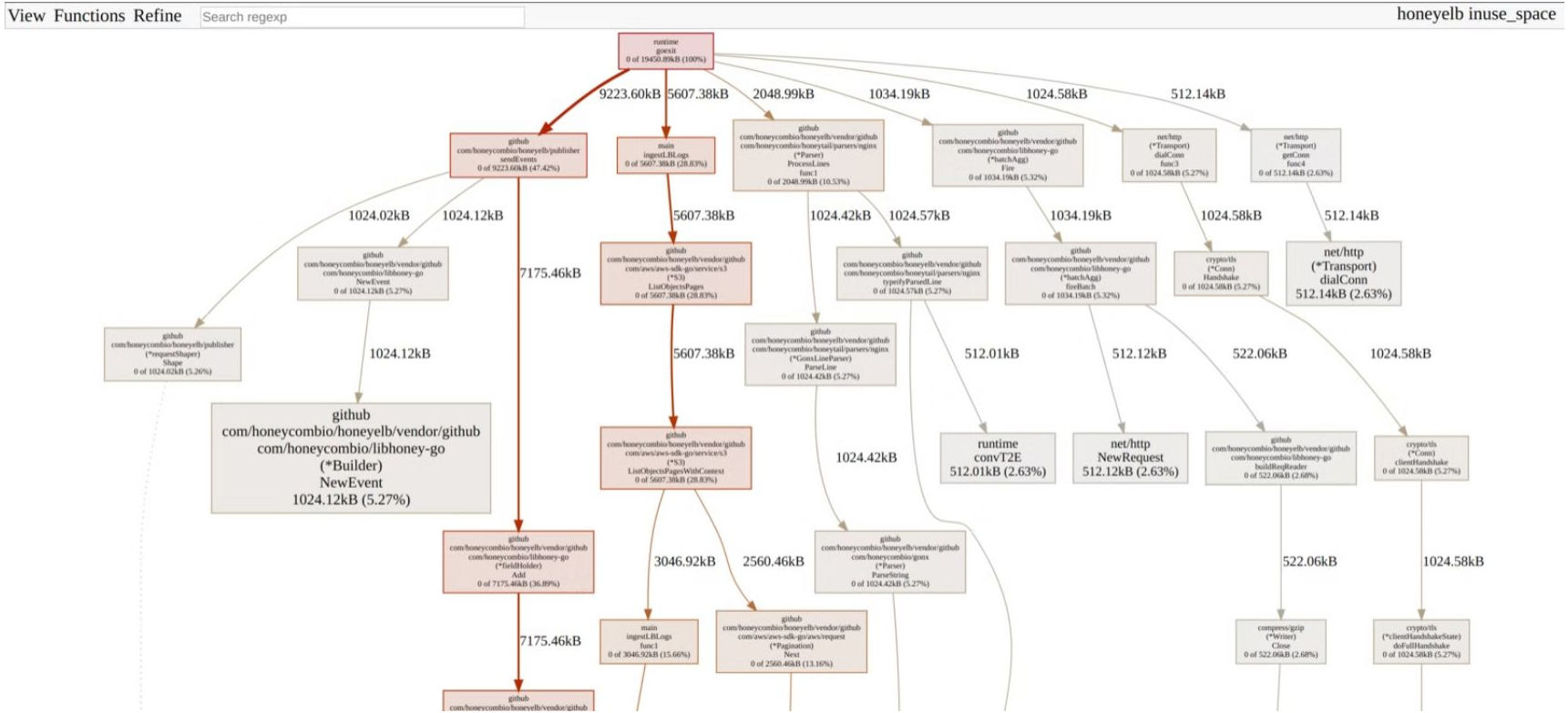
Analyze with pprof

Run **go tool pprof** to explore the profiles and identify performance bottlenecks in your application.

4

Visualize with Web UI

Start the pprof web server to interactively visualize the profiles in a browser and better understand your application's performance.



https://www.honeycomb.io/wp-content/uploads/2017/08/screen1.png

Monitoring and Observability with Go OpenTelemetry

Tracking Application Performance

Go OpenTelemetry provides a comprehensive suite of tools to monitor the performance of your Go applications, including capturing metrics, traces, and logs.

Database Connection Insights

OpenTelemetry can track the health and performance of your database connections, helping you identify and resolve issues with data access.

End-to-End Visibility

By integrating OpenTelemetry across your services, you gain end-to-end visibility into your application's behavior, enabling you to troubleshoot problems more effectively.

Vendor-Agnostic Instrumentation

OpenTelemetry's vendor-neutral approach allows you to use the same instrumentation across various cloud providers and observability platforms.

Opentelemetry with Code Examples

```go
import (
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/exporters/stdout/stdouttrace"
    "go.opentelemetry.io/otel/trace"
)

func setupTracing() {
    exporter, _ := stdouttrace.New()
    tracerProvider := trace.NewTracerProvider(trace.WithBatcher(exporter))
    otel.SetTracerProvider(tracerProvider)
}

import (
    "net/http"
    "go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp"
)

func main() {
    handler := http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hello, World!"))
    })
    wrappedHandler := otelhttp.NewHandler(handler, "hello-handler")

    http.ListenAndServe(":8080", wrappedHandler)
}
```

Opentelemetry with Code Examples

```go
import (
    "database/sql"
    "go.opentelemetry.io/contrib/instrumentation/database/sql/otelsql"
    _ "github.com/lib/pq" // PostgreSQL driver
)

func main() {
    driverName, _ := otelsql.Register("postgres", otelsql.WithAttributes())
    db, _ := sql.Open(driverName, "postgres://user:pass@localhost/dbname")
    defer db.Close()

    db.Query("SELECT * FROM users")
}
```

How many PRs in 2024?

# Ranking	Programming Language	Percentage (YoY Change)	YoY Trend
1	Python	16.925% (-0.284%)	
2	Java	11.708% (+0.393%)	
3	Go	10.262% (-0.162%)	
4	JavaScript	9.859% (+0.306%)	^
5	C++	9.459% (-0.624%)	v

https://madnight.github.io/githut/#/pull_requests/2024/1

How many issues in 2024?

# Ranking	Programming Language	Percentage (YoY Change)	YoY Trend
1	Python	16.278% (-0.052%)	
2	Java	11.453% (+0.642%)	
3	C++	10.370% (-0.385%)	
4	TypeScript	10.064% (-0.283%)	
5	JavaScript	9.939% (-0.104%)	
6	Go	8.987% (-0.569%)	
7	PHP	5.765% (-0.343%)	
8	C	5.691% (+0.357%)	
9	C#	4.688% (+0.118%)	
10	Dart	2.413% (-0.141%)	

https://madnight.github.io/githut/#/issues/2024/1

How many stars in 2024?

# Ranking	Programming Language	Percentage (YoY Change)	YoY Trend
1	Python	18.172% (+0.129%)	
2	JavaScript	15.278% (-1.341%)	
3	Go	12.275% (+0.151%)	
4	C++	9.750% (+0.785%)	
5	Java	7.959% (-0.074%)	

https://madnight.github.io/githut/#/stars/2024/1

How many pushes in 2024?

# Ranking	Programming Language	Percentage (YoY Change)	YoY Trend
1	Python	16.219% (-0.368%)	
2	Java	11.851% (-0.109%)	
3	JavaScript	11.003% (+0.241%)	
4	C++	10.069% (+0.119%)	
5	TypeScript	7.694% (-0.264%)	
6	PHP	7.692% (+0.426%)	
7	Go	6.809% (-0.039%)	
8	C	4.865% (+0.145%)	
9	Ruby	4.567% (-0.029%)	
10	C#	3.329% (+0.032%)	

https://madnight.github.io/githut/#/pushes/2024/1

What do you waiting for?

Thank you