Introduction to Golang

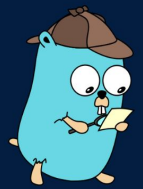
WHAT IS GO GOOD FOR?

WHY CHOOSE GO?

1

IT'S QUICKLY

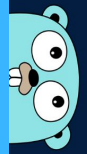
Golang is a really quick language. It compiles fast, runs fast, and it's also fast to team.



3

IT'S SIMPLE

Go is very clean and easy to learn. It has a minimalistic design and easy maintained code.



5

IT'S CROSS-PLATFORM

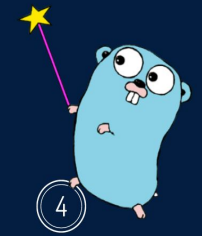
Go can be used with various platforms like UNIX, Linux, Windows, and other operating systems that work on mobile devices.

GO

2

IT'S POPULAR

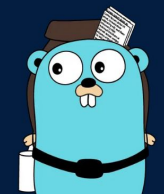
Go is a fast-growing language with an excellent combination of concurrency, safety, and simplicity of programming.



4

IT'S SECURE

Go's compiler and garbage collector cleans up all errors and bugs, what makes the entire framework more secure.



The Go programming language was conceived in late 2007 as an answer to some of the problems we were seeing developing software infrastructure at Google. The computing landscape today is almost unrelated to the environment in which the languages being used, mostly C++, Java, and Python, had been created. The problems introduced by multicore processors, networked systems, massive computation clusters, and the web programming model were being worked around rather than addressed head-on. Moreover, the scale has changed: today's server programs comprise tens of millions of lines of code, are worked on by hundreds or even thousands of programmers, and are updated literally every day. To make matters worse, build times, even on large compilation clusters, have stretched to many minutes, even hours.

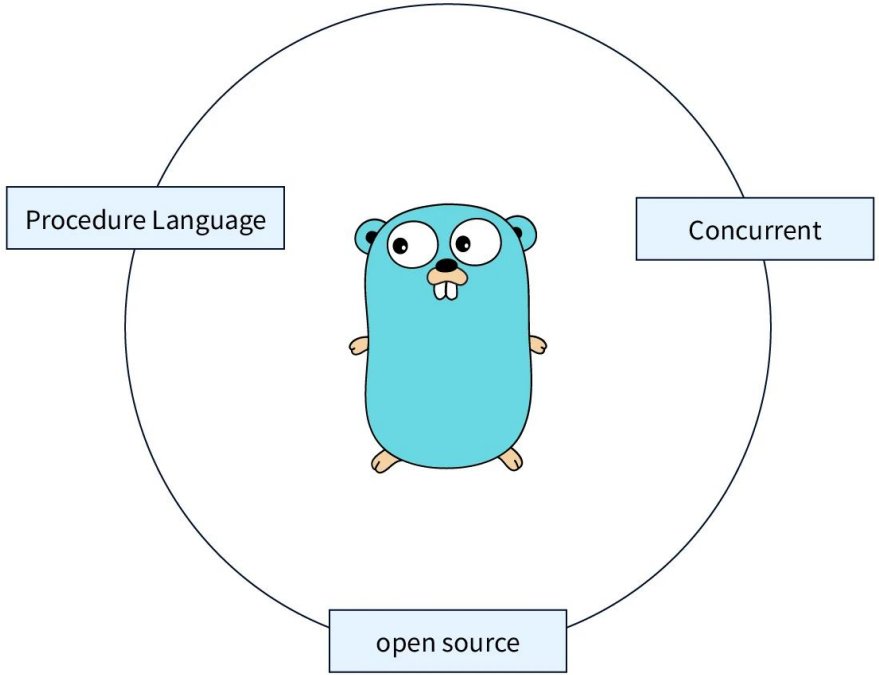
Go was designed and developed to make working in this environment more productive. Besides its better-known aspects such as built-in concurrency and garbage collection, Go's design considerations include rigorous dependency management, the adaptability of software architecture as systems grow, and robustness across the boundaries between components.

Advantages

- Simplicity and Ease of Use
- Strong Concurrency Support
- Efficient Runtime
- Extensive Standard Library
- Active Community

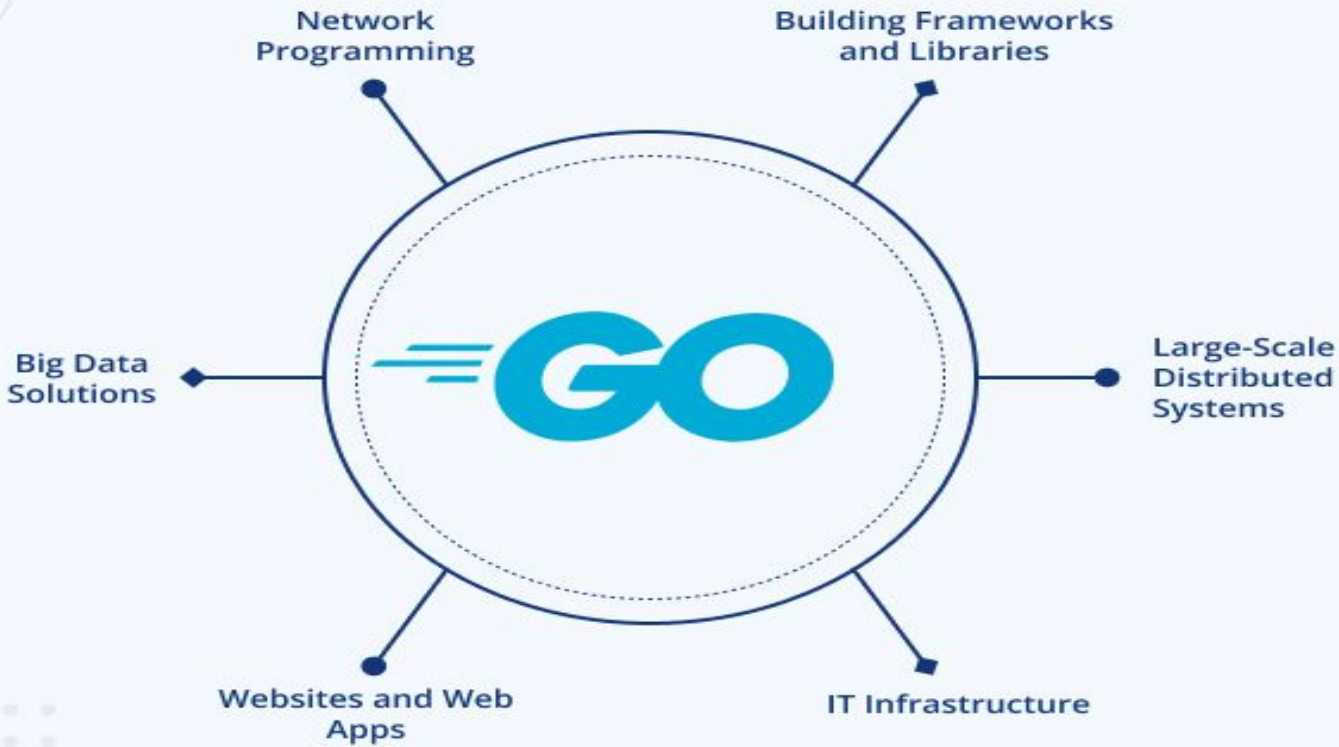
Disadvantages

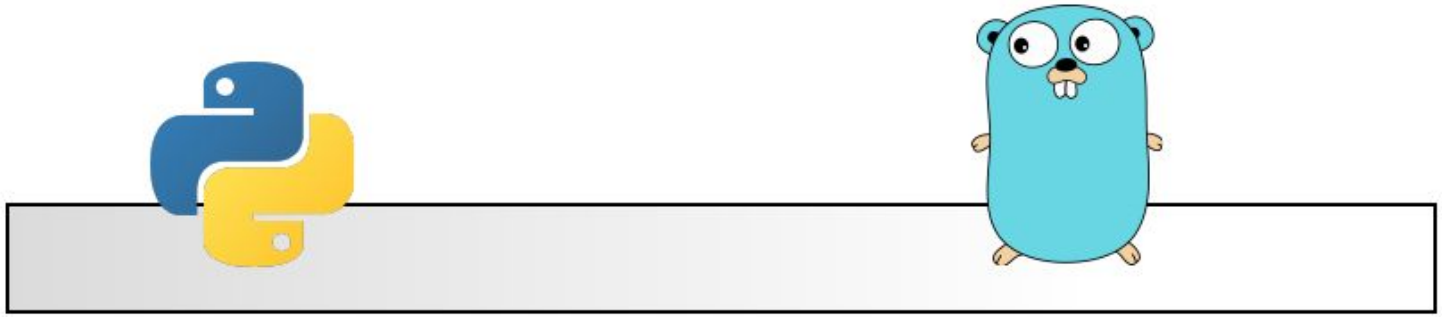
- Limited Generics Support
- Fewer Advanced Language Features
- Smaller Standard Library
- Limited Low-Level Systems Programming
- Complex Dependency Management



SCALER Topics

USE CASES OF GOLANG

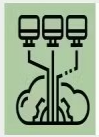




Flexibility and
Expressiveness

Stability and
Performance

Key Features of Golang



Concurrency support through goroutines and channels

Automatic memory management

Rich standard library

Simple, clean, and efficient syntax

Support for multiple programming paradigms

Turing

Pros

High Performance

Being a compiled language, the Go code is translated into machine code before execution. This results in faster speed for high-demand applications.

Advanced Security

The standard library of Go tools reduces dependence on third-party libraries and the possibility of security breaches. Go is also a statically typed language, which enhances code reliability and reduces runtime errors.

Efficiency

Go utilizes garbage collection for automatic memory management. This leads to lower system resource consumption and cost-effectiveness.

Scalability

Go's ability to handle multiple tasks concurrently enables applications to grow seamlessly with increasing user demands.

Cons

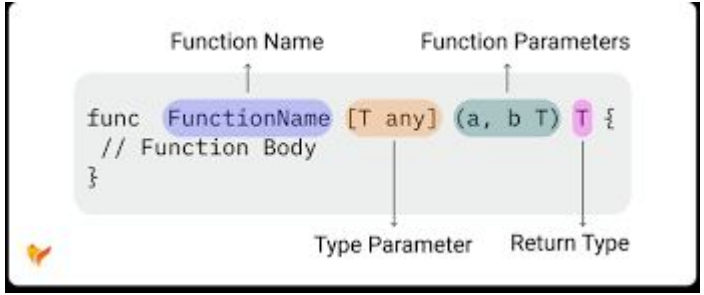
Limited GUI Capabilities

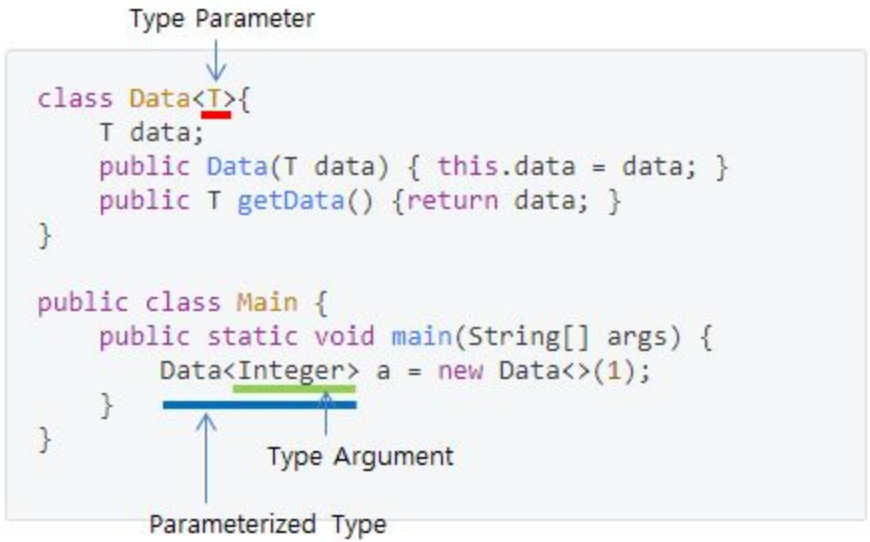
Go is primarily a server-side technology. The one way to use Go for the front end is by compiling it into WebAssembly

Smaller Developer Pool

While Go's syntax is simple, it is a relatively new programming language compared to Java, Python, or JavaScript.

Jay.devs






```
enerics > math.go > SumInts
7   func SumInts(m map[string]int64) int64 {
8       var s int64
9       for _, v := range m {
10          s += v
11      }
12      return s
13  }
14
15  func SumFloats(m map[string]float64) float64 {
16      var s float64
17      for _, v := range m {
18          s += v
19      }
20      return s
21  }
22
23  func SumIntsOrFloats[K comparable, V int64 | float64](m map[K]V) V {
24      var s V
25      for _, v := range m {
26          s += v
27      }
28      return s
```

Cloud-Native and Infrastructure:

- **Kubernetes (k8s):** The widely adopted container orchestration platform that automates deployment, scaling, and management of containerized applications. Go's concurrency and performance were crucial for building its distributed architecture.
- **Docker:** The platform that revolutionized containerization. Go's ease of use, strong standard library, and ability to produce single, portable binaries made it an excellent choice.
- **Prometheus:** A popular open-source monitoring and alerting system. Go's efficiency and concurrency handle the collection and processing of time-series data effectively.
- **etcd:** A distributed key-value store used by Kubernetes and other distributed systems for configuration management, service discovery, and distributed consensus. Go's strong networking and concurrency features are vital here.
- **Traefik:** A modern HTTP reverse proxy and load balancer that is cloud-native aware. Go's performance and integration capabilities with container orchestration systems are key.
- **Consul:** A service networking solution to connect and secure services across any runtime platform. Built by HashiCorp, it leverages Go's networking and concurrency strengths.
- **Terraform:** An infrastructure-as-code tool that allows you to define and provision infrastructure across various cloud providers. Go's extensibility and ability to interact with APIs efficiently are important.

Networking and Web:

- **Caddy:** A powerful, enterprise-ready, open-source web server with automatic HTTPS. Go's strong standard library for networking and its focus on security made it a good fit.
- **Hugo:** A fast and flexible static site generator. Go's speed and efficiency in processing files and templates are significant advantages.
- **Gin:** A high-performance HTTP web framework known for its speed and developer-friendliness.
- **Echo:** Another fast and extensible HTTP framework for building web applications and APIs.
- **NATS:** A lightweight, high-performance messaging system (message broker) for cloud-native systems, IoT messaging, and more. Go's concurrency and networking capabilities are essential.

Databases and Data Processing:

- **CockroachDB:** A distributed SQL database designed for resilience and scalability. Go's concurrency and low-level control contribute to its performance and fault tolerance.
- **InfluxDB:** A time-series database often used for monitoring and analytics. Go's efficiency in handling high volumes of data is crucial.
- **BadgerDB:** An embeddable, persistent key-value store written in Go. It's used as the underlying storage engine in some other Go projects.

Command-Line Tools:

- **Go itself (the `go` tool):** The command-line interface for the Go programming language is, of course, written in Go.
- **Docker CLI:** The command-line interface for interacting with the Docker daemon.
- **Kubectl:** The command-line tool for interacting with Kubernetes clusters.
- **Helm:** A package manager for Kubernetes.
- **Vault:** A secrets management tool by HashiCorp.
- **Nomad:** A cluster scheduler by HashiCorp.

In each of these examples, the choice of Go was often driven by a combination of the factors I mentioned earlier, but some specific reasons stand out:

- **For infrastructure tools (Kubernetes, Docker, Terraform):** The need for high concurrency to manage many resources simultaneously, efficient resource utilization for running on potentially large clusters, and the ability to produce portable, self-contained binaries for easy deployment.
- **For networking and web servers (Caddy, NATS):** The requirement for high performance to handle numerous concurrent connections and requests, and Go's strong networking libraries.
- **For databases (CockroachDB, InfluxDB):** The necessity for efficient memory management and concurrency to handle large datasets and concurrent queries.
- **For command-line tools:** Fast compilation times for quick iteration during development and the ability to create single, easily distributable executables.

These specific examples illustrate how Go's strengths in concurrency, performance, networking, and ease of deployment make it a compelling choice for building a wide range of modern software systems.

Go

```go
package main

import (
        "fmt"
        "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "Hello, World!")
}

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

Go

```go
package main

import (
	"fmt"
	"time"
)

func task(name string) {
	fmt.Printf("Task %s started\n", name)
	time.Sleep(time.Second) // Simulate work
	fmt.Printf("Task %s finished\n", name)
}

func main() {
	go task("A") // Start task A in a goroutine
	go task("B") // Start task B in a goroutine
	task("C")    // Run task C in the main goroutine

	time.Sleep(2 * time.Second) // Wait for goroutines to finish
	fmt.Println("All tasks completed")
}
```

Go

```go
package main

import "fmt"

// Address struct
type Address struct {
	Street  string
	City    string
	State   string
	ZipCode string
}

// Person struct
type Person struct {
	FirstName string
	LastName  string
	Age       int
	Address   Address // Embedding the Address struct
}

func main() {
	// Creating an Address instance
	address := Address{
		Street:  "123 Main St",
		City:    "Anytown",
		State:   "CA",
		ZipCode: "91234",
	}

	// Creating a Person instance, embedding the Address
	person := Person{
		FirstName: "John",
		LastName:  "Doe",
		Age:       30,
		Address:   address, // Assigning the Address instance
	}
```

Go

```go
package main

import (
	"encoding/json"
	"fmt"
	"os"
)

// Define a struct that matches the JSON structure
type Person struct {
	Name string `json:"name"`
	Age  int    `json:"age"`
	City string `json:"city"`
}

func main() {
	// Open the JSON file
	jsonFile, err := os.Open("data.json")
	if err != nil {
		fmt.Println(err)
		return
	}
	defer jsonFile.Close()

	// Create a decoder to read the JSON stream
	decoder := json.NewDecoder(jsonFile)

	// Read the opening bracket of the JSON array
	_, err = decoder.Token()
	if err != nil {
		fmt.Println(err)
		return
	}

	// Loop through the JSON array
	for decoder.More() {
		var p Person
		err := decoder.Decode(&p) // Decode each JSON object into
		if err != nil {
			fmt.Println(err)
			return
		}
		fmt.Printf("Person: %+v\n", p)
	}

	// Read the closing bracket of the JSON array
	_, err = decoder.Token()
	if err != nil {
		fmt.Println(err)
		return
	}
}
```

```go
package main

import "fmt"

func intSeq() func() int {
    i := 0
    return func() int {
        i++
        return i
    }
}

func main() {

    nextInt := intSeq()



    fmt.Println(nextInt())
    fmt.Println(nextInt())
    fmt.Println(nextInt())

    newInts := intSeq()
    fmt.Println(newInts())
}
```

```go
package main

import "fmt"

func main() {
    messages := make(chan string)
    signals := make(chan bool)

    select {
    case msg := <-messages:
        fmt.Println("received message", msg)
    default:
        fmt.Println("no message received")
    }

    msg := "hi"
    select {
    case messages <- msg:
        fmt.Println("sent message", msg)
    default:
        fmt.Println("no message sent")
    }

    select {
    case msg := <-messages:
        fmt.Println("received message", msg)
    case sig := <-signals:
        fmt.Println("received signal", sig)
    default:
        fmt.Println("no activity")
    }
}
```

```
$ go run non-blocking-channel-operations.go
no message received
no message sent
no activity
```