**Golang Best Practices (Top 20)**

[[](https://medium.com/@golangda?source=post_page-----c0cea6a43f20--------------------------------)](https://medium.com/@golangda?source=post_page-----c0cea6a43f20--------------------------------)

[Golang Da](https://medium.com/@golangda?source=post_page-----c0cea6a43f20--------------------------------)

·

Follow

12 min read

·

Nov 9

556

6

**Introduction**

Spend just 12 minutes of your time on this article and it will help you to write an effective Go code.

**#20: Use Proper Indentation**

Good indentation makes your code readable. Use tabs or spaces consistently (preferably tabs) and follow the Go standard convention for indentation.

package main  
  
import "fmt"  
  
func main() {  
 for i := 0; i < 5; i++ {  
 fmt.Println("Hello, World!")  
 }  
}

Run gofmt to automatically format(indent) your code according to the Go standard.

$ gofmt -w your\_file.go

**#19: Import Packages Properly**

Import only the packages you need, and format the import section to group standard library packages, third-party packages, and your own packages.

package main  
  
import (  
 "fmt"  
 "math/rand"  
 "time"  
)

**#18: Use Descriptive Variable and Function Names**

1. ***Meaningful Names***: Use names that convey the purpose of the variable.
2. ***CamelCase***: Start with a lowercase letter and capitalize the first letter of each subsequent word within the name.
3. ***Short Names***: Short, concise names are acceptable for short-lived variables with a small scope.
4. ***No Abbreviations***: Avoid cryptic abbreviations and acronyms in favor of descriptive names.
5. ***Consistency***: Maintain naming consistency throughout your codebase.

package main  
  
import "fmt"  
  
func main() {  
 // Declare variables with meaningful names  
 userName := "John Doe" // CamelCase: Start with lowercase and capitalize subsequent words.  
 itemCount := 10 // Short Names: Short and concise for small-scoped variables.  
 isReady := true // No Abbreviations: Avoid cryptic abbreviations or acronyms.  
  
 // Display variable values  
 fmt.Println("User Name:", userName)  
 fmt.Println("Item Count:", itemCount)  
 fmt.Println("Is Ready:", isReady)  
}  
  
// Use mixedCase for package-level variables  
var exportedVariable int = 42  
  
// Function names should be descriptive  
func calculateSumOfNumbers(a, b int) int {  
 return a + b  
}  
  
// Consistency: Maintain naming consistency throughout your codebase.

**#17: Limit Line Length**

Keep your lines of code under 80 characters when possible to improve readability.

package main  
  
import (  
 "fmt"  
 "math"  
)  
  
func main() {  
 result := calculateHypotenuse(3, 4)  
 fmt.Println("Hypotenuse:", result)  
}  
  
func calculateHypotenuse(a, b float64) float64 {  
 return math.Sqrt(a\*a + b\*b)  
}

**#16: Use Constants for Magic Values**

Avoid magic values in your code. Magic values are hard-coded numbers or strings scattered throughout your code that lack context, making it challenging to understand their purpose. Define constants for them to make your code more maintainable.

package main  
  
import "fmt"  
  
const (  
 // Define a constant for a maximum number of retries  
 MaxRetries = 3  
  
 // Define a constant for a default timeout in seconds  
 DefaultTimeout = 30  
)  
  
func main() {  
 retries := 0  
 timeout := DefaultTimeout  
  
 for retries < MaxRetries {  
 fmt.Printf("Attempting operation (Retry %d) with timeout: %d seconds\n", retries+1, timeout)  
   
 // ... Your code logic here ...  
  
 retries++  
 }  
}

**#15: Error Handling**

Go encourages developers to handle errors explicitly for the following reasons:

1. ***Safety***: Error handling ensures that unexpected issues don’t cause the program to panic or crash abruptly.
2. ***Clarity***: Explicit error handling makes your code more readable and helps identify where errors may occur.
3. ***Debugging***: Handling errors provides valuable information for debugging and troubleshooting.

Let’s create a simple program that reads a file and handles errors properly:

package main  
  
import (  
 "fmt"  
 "os"  
)  
  
func main() {  
 // Open a file  
 file, err := os.Open("example.txt")  
 if err != nil {  
 // Handle the error  
 fmt.Println("Error opening the file:", err)  
 return  
 }  
 defer file.Close() // Close the file when done  
  
 // Read from the file  
 buffer := make([]byte, 1024)  
 \_, err = file.Read(buffer)  
 if err != nil {  
 // Handle the error  
 fmt.Println("Error reading the file:", err)  
 return  
 }  
  
 // Print the file content  
 fmt.Println("File content:", string(buffer))  
}

**#14: Avoid Global Variables**

Minimize the use of global variables. Global variables can lead to unpredictable behavior, make debugging challenging, and hinder code reuse. They can also introduce unnecessary dependencies between different parts of your program. Instead, pass data through function parameters and return values.

Let’s write a simple Go program to illustrate the concept of avoiding global variables:

package main  
  
import (  
 "fmt"  
)  
  
func main() {  
 // Declare and initialize a variable within the main function  
 message := "Hello, Go!"  
  
 // Call a function that uses the local variable  
 printMessage(message)  
}  
  
// printMessage is a function that takes a parameter  
func printMessage(msg string) {  
 fmt.Println(msg)  
}

**#13: Use Structs for Complex Data**

Use structs to group related data fields and methods together. They allow you to group related variables together, making your code more organized and readable.

Here’s a complete sample program that demonstrates the use of structs in Go:

package main  
  
import (  
 "fmt"  
)  
  
// Define a struct named Person to represent a person's information.  
type Person struct {  
 FirstName string // First name of the person  
 LastName string // Last name of the person  
 Age int // Age of the person  
}  
  
func main() {  
 // Create an instance of the Person struct and initialize its fields.  
 person := Person{  
 FirstName: "John",  
 LastName: "Doe",  
 Age: 30,  
 }  
  
 // Access and print the values of the struct's fields.  
 fmt.Println("First Name:", person.FirstName) // Print first name  
 fmt.Println("Last Name:", person.LastName) // Print last name  
 fmt.Println("Age:", person.Age) // Print age  
}

**#12: Comment Your Code**

Add comments to explain your code’s functionality, especially for complex or non-obvious parts.

**Single-Line Comments**

Single-line comments begin with //. Use them to explain specific lines of code.

package main  
  
import "fmt"  
  
func main() {  
 // This is a single-line comment  
 fmt.Println("Hello, World!") // Print a greeting  
}

**Multi-Line Comments**

Multi-line comments are enclosed in /\* \*/. Use them for longer explanations or comments spanning multiple lines.

package main  
  
import "fmt"  
  
func main() {  
 /\*  
 This is a multi-line comment.  
 It can span several lines.  
 \*/  
 fmt.Println("Hello, World!") // Print a greeting  
}

**Function Comments**

Add comments to functions to explain their purpose, parameters, and return values. Use the `*godoc`*style for function comments.

package main  
  
import "fmt"  
  
// greetUser greets a user by name.  
// Parameters:  
// name (string): The name of the user to greet.  
// Returns:  
// string: The greeting message.  
func greetUser(name string) string {  
 return "Hello, " + name + "!"  
}  
  
func main() {  
 userName := "Alice"  
 greeting := greetUser(userName)  
 fmt.Println(greeting)  
}

**Package Comments**

Add comments at the top of your Go files to describe the package’s purpose. Use the same `*godoc`*style.

package main  
  
import "fmt"  
  
// This is the main package of our Go program.  
// It contains the entry point (main) function.  
func main() {  
 fmt.Println("Hello, World!")  
}

**#11: Use goroutines for Concurrency**

Utilize goroutines to perform concurrent operations efficiently. Goroutines are lightweight, concurrent threads of execution in Go. They enable you to run functions concurrently without the overhead of traditional threads. This allows you to write highly concurrent and efficient programs.

Let’s demonstrate this with a simple example:

package main  
  
import (  
 "fmt"  
 "time"  
)  
  
// Function that runs concurrently  
func printNumbers() {  
 for i := 1; i <= 5; i++ {  
 fmt.Printf("%d ", i)  
 time.Sleep(100 \* time.Millisecond)  
 }  
}  
  
// Function that runs in the main goroutine  
func main() {  
 // Start the goroutine  
 go printNumbers()  
  
 // Continue executing main  
 for i := 0; i < 2; i++ {  
 fmt.Println("Hello")  
 time.Sleep(200 \* time.Millisecond)  
 }  
 // Ensure the goroutine completes before exiting  
 time.Sleep(1 \* time.Second)  
}

**#10: Handle panics with Recover**

Use the recover function to gracefully handle panics and prevent program crashes. In Go, panics are unexpected runtime errors that can crash your program. However, Go provides a mechanism called recover to gracefully handle panics.

Let’s demonstrate this with a simple example:

package main  
  
import "fmt"  
  
// Function that might panic  
func riskyOperation() {  
 defer func() {  
 if r := recover(); r != nil {  
 // Recover from the panic and handle it gracefully  
 fmt.Println("Recovered from panic:", r)  
 }  
 }()  
  
 // Simulate a panic condition  
 panic("Oops! Something went wrong.")  
}  
  
func main() {  
 fmt.Println("Start of the program.")  
  
 // Call the risky operation within a function that recovers from panics  
 riskyOperation()  
  
 fmt.Println("End of the program.")  
}

**#9: Avoid Using `init` Functions**

Avoid using init functions unless necessary, as they can make code harder to understand and maintain.

A better approach is to move your initialization logic into regular functions that you call explicitly, typically from your main function. This gives you better control, enhances code readability, and simplifies testing.

Here’s a simple Go program that demonstrates avoiding init functions:

package main  
  
import (  
 "fmt"  
)  
  
// InitializeConfig initializes configuration.  
func InitializeConfig() {  
 // Initialize configuration parameters here.  
 fmt.Println("Initializing configuration...")  
}  
  
// InitializeDatabase initializes the database connection.  
func InitializeDatabase() {  
 // Initialize database connection here.  
 fmt.Println("Initializing database...")  
}  
  
func main() {  
 // Call initialization functions explicitly.  
 InitializeConfig()  
 InitializeDatabase()  
  
 // Your main program logic goes here.  
 fmt.Println("Main program logic...")  
}

**#8: Use Defer for Resource Cleanup**

defer allows you to delay the execution of a function until the surrounding function returns. It’s commonly used for tasks like closing files, unlocking mutexes, or releasing other resources.

This ensures that cleanup actions are performed, even in the presence of errors.

Let’s create a simple program that reads data from a file, and we’ll use defer to ensure the file is properly closed, regardless of any errors that may occur:

package main  
  
import (  
 "fmt"  
 "os"  
)  
  
func main() {  
 // Open the file (Replace "example.txt" with your file's name)  
 file, err := os.Open("example.txt")  
 if err != nil {  
 fmt.Println("Error opening the file:", err)  
 return // Exit the program on error  
 }  
 defer file.Close() // Ensure the file is closed when the function exits  
  
 // Read and print the contents of the file  
 data := make([]byte, 100)  
 n, err := file.Read(data)  
 if err != nil {  
 fmt.Println("Error reading the file:", err)  
 return // Exit the program on error  
 }  
  
 fmt.Printf("Read %d bytes: %s\n", n, data[:n])  
}

**#7: Prefer Composite Literal over Constructor Functions**

Use composite literals to create instances of structs instead of constructor functions.

**Why Use Composite Literals?**

Composite literals provide several advantages:

1. Conciseness
2. Readability
3. Flexibility

Let’s demonstrate this with a simple example:

package main  
  
import (  
 "fmt"  
)  
  
// Define a struct type representing a person  
type Person struct {  
 FirstName string // First name of the person  
 LastName string // Last name of the person  
 Age int // Age of the person  
}  
  
func main() {  
 // Using a composite literal to create a Person instance  
 person := Person{  
 FirstName: "John", // Initialize the FirstName field  
 LastName: "Doe", // Initialize the LastName field  
 Age: 30, // Initialize the Age field  
 }  
  
 // Printing the person's information  
 fmt.Println("Person Details:")  
 fmt.Println("First Name:", person.FirstName) // Access and print the First Name field  
 fmt.Println("Last Name:", person.LastName) // Access and print the Last Name field  
 fmt.Println("Age:", person.Age) // Access and print the Age field  
}

**#6: Minimize Function Parameters**

In Go, it’s essential to write clean and efficient code. One way to do this is by minimizing the number of function parameters, which can lead to more maintainable and readable code.

Let’s explore this concept with a simple example:

package main  
  
import "fmt"  
  
// Option struct to hold configuration options  
type Option struct {  
 Port int  
 Timeout int  
}  
  
// ServerConfig is a function that accepts an Option struct  
func ServerConfig(opt Option) {  
 fmt.Printf("Server configuration - Port: %d, Timeout: %d seconds\n", opt.Port, opt.Timeout)  
}  
  
func main() {  
 // Creating an Option struct with default values  
 defaultConfig := Option{  
 Port: 8080,  
 Timeout: 30,  
 }  
  
 // Configuring the server with default options  
 ServerConfig(defaultConfig)  
  
 // Modifying the Port using a new Option struct  
 customConfig := Option{  
 Port: 9090,  
 }  
  
 // Configuring the server with custom Port value and default Timeout  
 ServerConfig(customConfig)  
}

In this example, we define an Option struct to hold configuration parameters for a server. Instead of passing multiple parameters to the ServerConfig function, we use a single Option struct, which makes the code more maintainable and extensible. This approach is especially useful when you have functions with numerous configuration parameters.

**#5: Use Explicit Return Values Instead of Named Return Values for Clarity**

Named return values are commonly used in Go, but they can sometimes make code less clear, especially in larger codebases.

Let’s see the difference using a simple example.

package main  
  
import "fmt"  
  
// namedReturn demonstrates named return values.  
func namedReturn(x, y int) (result int) {  
 result = x + y  
 return  
}  
  
// explicitReturn demonstrates explicit return values.  
func explicitReturn(x, y int) int {  
 return x + y  
}  
  
func main() {  
 // Named return values  
 sum1 := namedReturn(3, 5)  
 fmt.Println("Named Return:", sum1)  
  
 // Explicit return values  
 sum2 := explicitReturn(3, 5)  
 fmt.Println("Explicit Return:", sum2)  
}

In the sample program above, we have two functions, namedReturn and explicitReturn. Here's how they differ:

* namedReturn uses a named return value result. Although it's clear what the function is returning, it may not be immediately obvious in more complex functions.
* explicitReturn returns the result directly. This is simpler and more explicit.

**#4: Keep function complexity to the minimum**

Function complexity refers to the degree of intricacy, nesting, and branching within a function’s code. Keeping function complexity low makes your code more readable, maintainable, and less prone to errors.

Let’s explore this concept with a simple example:

package main  
  
import (  
 "fmt"  
)  
  
// CalculateSum returns the sum of two numbers.  
func CalculateSum(a, b int) int {  
 return a + b  
}  
  
// PrintSum prints the sum of two numbers.  
func PrintSum() {  
 x := 5  
 y := 3  
 sum := CalculateSum(x, y)  
 fmt.Printf("Sum of %d and %d is %d\n", x, y, sum)  
}  
  
func main() {  
 // Call the PrintSum function to demonstrate minimal function complexity.  
 PrintSum()  
}

In above sample program:

1. We define two functions, CalculateSum and PrintSum, with specific responsibilities.
2. CalculateSum is a simple function that calculates the sum of two numbers.
3. PrintSum uses CalculateSum to calculate and print the sum of 5 and 3.
4. By keeping functions concise and focused on a single task, we maintain low function complexity, improving code readability and maintainability.

**#3: Avoid Shadowing of Variables**

Shadowing of variables occurs when a new variable with the same name is declared within a narrower scope, which can lead to unexpected behavior. It hides the outer variable with the same name, making it inaccessible within that scope. Avoid shadowing variables within nested scopes to prevent confusion.

Let’s look at a sample program:

package main  
  
import "fmt"  
  
func main() {  
 // Declare and initialize an outer variable 'x' with the value 10.  
 x := 10  
 fmt.Println("Outer x:", x)  
  
 // Enter an inner scope with a new variable 'x' shadowing the outer 'x'.  
 if true {  
 x := 5 // Shadowing occurs here  
 fmt.Println("Inner x:", x) // Print the inner 'x', which is 5.  
 }  
  
 // The outer 'x' remains unchanged and is still accessible.  
 fmt.Println("Outer x after inner scope:", x) // Print the outer 'x', which is 10.  
}

**#2: Use Interfaces for Abstraction**

**Abstraction**

Abstraction is a fundamental concept in Go, allowing us to define behavior without specifying implementation details.

**Interfaces**

In Go, an interface is a collection of method signatures.

Any type that implements all the methods of an interface implicitly satisfies that interface.

This enables us to write code that can work with different types as long as they adhere to the same interface.

Here’s a single sample program in Go that demonstrates the concept of using interfaces for abstraction:

package main  
  
import (  
 "fmt"  
 "math"  
)  
  
// Define the Shape interface  
type Shape interface {  
 Area() float64  
}  
  
// Rectangle struct  
type Rectangle struct {  
 Width float64  
 Height float64  
}  
  
// Circle struct  
type Circle struct {  
 Radius float64  
}  
  
// Implement the Area method for Rectangle  
func (r Rectangle) Area() float64 {  
 return r.Width \* r.Height  
}  
  
// Implement the Area method for Circle  
func (c Circle) Area() float64 {  
 return math.Pi \* c.Radius \* c.Radius  
}  
  
// Function to print the area of any Shape  
func PrintArea(s Shape) {  
 fmt.Printf("Area: %.2f\n", s.Area())  
}  
  
func main() {  
 rectangle := Rectangle{Width: 5, Height: 3}  
 circle := Circle{Radius: 2.5}  
  
 // Call PrintArea on rectangle and circle, both of which implement the Shape interface  
 PrintArea(rectangle) // Prints the area of the rectangle  
 PrintArea(circle) // Prints the area of the circle  
}

In this single program, we define the Shape interface, create two structs Rectangle and Circle, each implementing the Area() method, and use the PrintArea function to print the area of any shape that satisfies the Shape interface.

This demonstrates how you can use interfaces for abstraction in Go to work with different types using a common interface.

**#1: Avoid Mixing Library Packages and Executable Files**

In Go, it’s crucial to maintain a clear separation between packages and executable files to ensure clean and maintainable code.

Here’s sample project structure which demonstrates the separation of library and executable file:

myproject/  
 ├── main.go  
 ├── myutils/  
 └── myutils.go

myutils/myutils.go:

// Package declaration - Create a separate package for utility functions  
package myutils  
  
import "fmt"  
  
// Exported function to print a message  
func PrintMessage(message string) {  
 fmt.Println("Message from myutils:", message)  
}

main.go:

// Main program  
package main  
  
import (  
 "fmt"  
 "myproject/myutils" // Import the custom package  
)  
  
func main() {  
 message := "Hello, Golang!"  
  
 // Call the exported function from the custom package  
 myutils.PrintMessage(message)  
  
 // Demonstrate the main program logic  
 fmt.Println("Message from main:", message)  
}

1. In the above example, we have two separate files: myutils.go and main.go.
2. myutils.go defines a custom package named myutils. It contains an exported function PrintMessage that prints a message.
3. main.go is the executable file that imports the custom package myutils using its relative path ("myproject/myutils").
4. The main function in main.go calls the PrintMessage function from the myutils package and prints a message. This separation of concerns keeps the code organized and maintainable.

Happy coding!

Clap 10 times to show support!!

***Note: This article is part of my Golang: Quick Reference series for budding gophers.***