Dmitrii Kumancev · Follow

7 min read · Nov 18, 2023

Hi friends!
Today I suggest you to familiarize yourself with recommendations on writing clean code in Go. We will understand the peculiarities of the language by examples and apply the main syntactic constructions in practice.

We're off!

Working with Data

The Distinction between make and new

Make and new are built-in mechanisms for memory allocation. They are used in different situations and have their own characteristics.

- **new** initializes a zero value for the given type and returns a pointer to that type.
- **make** is exclusively used for creating and initializing slices, maps, and channels, returning a non-zero instance of the specified type.
- The main difference between them lies in the fact that **make** returns an initialized type ready for use after creation, while **new** returns a pointer to the type with its zero value.

```
a := new(chan int) // a has type *chan int
b := make(chan int) // b has type chan int
```

Hidden Data in Slices

A slice is a variable-length array that can store elements of a single type, internally represented as a reference to the underlying array.

When working with slices, there is often a need to “cut” them into smaller pieces. As a result, the resulting slice will reference the original array. It is crucial not to forget about this, as otherwise, the program may experience unpredictable memory consumption.

```
// Poor practice - unpredictable memory consumption
func cutSlice() []byte {
 slice := make([]byte, 256)
 fmt.Println(len(slice), cap(slice), &slice[0]) // 256 256 <0x…>
 return slice[:10]
}
func main() {
 res := cutSlice()
 fmt.Println(len(res), cap(res), &res[0]) // 10 256 <0x…>
}
```

Let's examine this characteristic through specific examples:
To prevent this error, it is essential to ensure that the copy is made from a temporary slice:

```
// Good practice - data copied from a temporary slice
func cutSlice() []byte {
 slice := make([]byte, 256)
 fmt.Println(len(slice), cap(slice), &slice[0]) // 256 256 <0x…>
 copyOfSlice := make([]byte, 10)
 copy(copyOfSlice, slice[:10])
 return slice[:10]
}
func main() {
 res := cutSlice()
 fmt.Println(len(res), cap(res), &res[0]) // 10 256 <0x…>
}
```

Functions

Functions with Multiple Returns

In the Go programming language, functions can return multiple values, a feature known as "multiple returns." This language feature allows functions to not only return a result but also additional values, such as errors or other necessary data.

Here is an example of declaring a function with multiple returns in Go:

```
package main
import "fmt"
func swap(a, b int) (int, int) {
 return b, a
}
func main() {
 x, y := swap(1, 2)
 fmt.Println(x, y) // 2 1
a, _ := swap(3, 4)
 fmt.Println(a) // 4
}
```

In the provided example, the `*swap*` function takes two arguments of type `*int*` and returns two values of the same type, swapping the positions of the input variables.

It is also possible to ignore one or more of the returned values using the blank identifier (`_`).

Functions with multiple returns are particularly useful when there is a need to return multiple results, such as when working with errors or parallel data processing.

The following `***openFile***` function returns two values, one of which is an error or ***nil*** in case of its absence:

```
func openFile(name string) (*File, error) {
 file, err := os.Open(name)
 if err != nil {
 return nil, err
 }
 return file, nil
}
```

Interfaces

In Go, interfaces represent a set of methods that define an object's behavior. They allow abstraction from a specific implementation and enable working with various data types. In other words, interfaces only define a certain functionality but do not implement it themselves.

> ***Using Interfaces Correctly***
> 💡*Remember this important rule:*
> *Avoid defining interfaces before their usage. Without a real example, it's challenging to determine whether they are genuinely necessary, not to mention the methods they should contain.*

```
package worker // worker.go

type Worker interface { Work() bool }

func Foo(w Worker) string { … }
```

```
package worker // worker_test.go

type secondWorker struct{ … }
func (w secondWorker) Work() bool { … }
```

```
…
if Foo(secondWorker{ … }) == "value" { … }
```

Below is an example of an incorrect approach when working with interfaces:

```
// Poor practice
package employer

type Worker interface { Worker() bool }

type defaultWorker struct{ … }
func (t defaultWorker) Work() bool { … }

func NewWorker() Worker { return defaultWorker{ … } }
```

The correct solution from the Go perspective is to return the specific type and allow *`Worker`* to mimic the *`employer's`* implementation:

```
// Good practice
package employer

type Worker struct { … }
func (w Worker) Work() bool { … }

func NewWorker() Worker {
 return Worker{
 …
 }
}
```

Concurrency and Parallelism

Tracking Goroutines

Goroutines are inexpensive to launch and operate, but they come with a finite cost in terms of memory usage — you cannot create an infinite number of them. Unlike variables, the Go runtime cannot detect when a goroutine will no longer be used.

A detailed exploration of goroutines I'll tell you about it next time — for now you can look up other sources about it.

> 💡 ***Remember this important rule:***
> *Every time you use the `go` keyword in your program to launch a goroutine, you must know how and when it will finish. Not knowing the answers to these two questions can lead to memory leaks.*

Let’s consider an example to illustrate this mistake:

```
func leakGoroutine() {
 ch := make(chan int)
 go func() {
   received := <-ch
   fmt.Println("Received value:", received)
 }
}
```

Here, the ***`leakGoroutine`*** function launches a goroutine that blocks reading from the ***`ch`*** channel. As a result, nothing will be sent to it, and it will never close. The goroutine will be blocked indefinitely, and the call to the ***`fmt.Println`*** function will never happen.

Detecting Leaks

Engineers at Uber, actively involved in Go development, created a goroutine leak detector — the `goleak` package, designed to integrate with modular tests. Let’s look at an example of using this tool in practice.

Suppose there is a function ***`leakGoroutine`*** with a goroutine leak:

```
func leakGoroutine() {
 go func() {
   time.Sleep(time.Minute)
 }()
  return nil
}
```

And a test for this function:

```go
func TestLeakGoroutine(t *testing.T) {
  defer goleak.VerifyNone(t)

  if err := leakGoroutine(); err != nil {
    t.Fatal("Fatal message")
 }
}
```

When running the tests, an error message appears, indicating *"found unexpected goroutines,"* along with the stack trace of the problematic goroutine, its state, and identifier.

This tool can be valuable in program development as it helps reduce the time spent identifying and resolving memory leaks.

Error Handling and Recovery

Errors in Go are represented by the `error` interface, which defines the `Error() string` method. Any type implementing this method can be used as an error.

```go
type error interface {
 Error() string
}
```

Handling Errors Correctly

Ignoring errors can lead to undefined behavior and complicate code debugging. Let's consider the correct way to handle errors using a file operation as an example:

```go
// Poor practice
file, err := os.Open("filename.txt")
if err == nil {
 // file operations
}
```

```go
// Good practice
file, err := os.Open("filename.txt")
if err != nil {
 log.Fatal(err) // error handling
```

```
}
defer file.Close() // deferred function call to close the file
```

Without Panicking, but with Recovery

The classic way to report an error is to return the *`error`* type. However, what should be done in cases where quick recovery is not possible? In such situations, the built-in *`panic`* function comes to the rescue. It terminates the program and outputs a customizable error message.

Here's an example of a simple function with panic:

```go
package main

import "fmt"

func examplePanic() {
 panic("Panic - program terminated")
 fmt.Println("Function examplePanic successfully completed")
}

func main() {
 examplePanic()
 fmt.Println("Function main successfully completed")
}
```

When a panic occurs, the function terminates, and any remaining deferred functions are executed using *`defer`*, along with unwinding the stack of goroutines. In real-world development, situations leading to panics should be avoided as they jeopardize the smooth operation of the program. Fortunately, the Go authors anticipated this drawback and created a panic recovery mechanism — *`recover`*. It allows halting the unwinding of the stack and returning control to the developer.

To demonstrate the operation of this mechanism, let's refer to the example:

```go
package main

import "fmt"

func Recovery() {
 if recoveryResult := recover(); recoveryResult != nil {
 fmt.Println(recoveryResult)
 }
```

```
  fmt.Println("Recovery…")
 }

 func Panic() {
  defer Recovery()
  panic("Panic")
  fmt.Println("Function Panic successfully completed")
 }

 func main() {
  Panic()
  fmt.Println("Function main successfully completed")
 }
```

Upon code execution, we get the following output:

```
Panic
Recovery…
Function main successfully completed
```

Note that the `*Panic*` function does not complete after the panic. This is because the deferred function `*Recovery*` is called via `*defer*`, which restores the program's operation. Subsequently, execution is handed back to `*main*`, where the entire code is successfully completed.

Conclusion

Code quality relies not just on the programming language but also on the developer's skills. By applying the discussed examples and adhering to general principles, you can enhance the quality of your software.

This article aims to inspire readers to implement these practices in Go development, creating programs that are easily understandable, even for beginners. Always remember, clean code is the path to a successful project!

> **That's it! I hope my article was interesting and informative for you 😎😜**

Don't forget about my github: https://github.com/DmitriiKumancev

See you soon!❤️

Golang Programming DevOps Development Computer Science

Written by Dmitrii Kumancev

15 Followers · 1 Following

Follow

🚀 Passionate Backend AI & Web Developer 🖥️ My github: https://github.com/DmitriiKumancev 🚀✨

Responses (3)

What are your thoughts?

Respond

Tomáš Mráz
about 1 year ago

Probably copyOfSlice should be returned in cutSlice()?

3 1 reply

Reply

Arton D.
about 1 year ago

Very helpful!

Reply

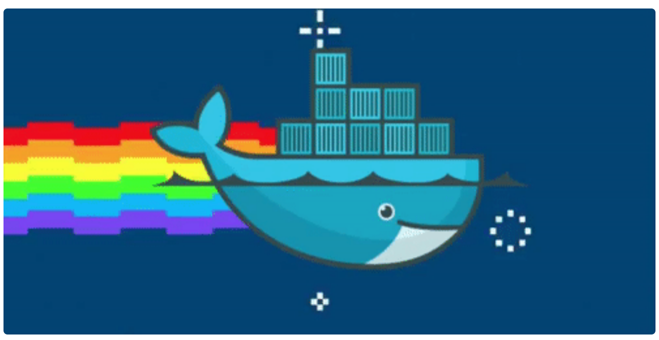
1

Alexandr Kumancev
about 1 year ago

Great 👍

5 Reply

More from Dmitrii Kumancev



In DevOps.dev by Dmitrii Kumancev

Docker Security: Best Practices, Configurations, and Real-Life...

Hi, dudes! Docker security is essential in protecting containerized applications from...

Nov 5, 2024 12

Dmitrii Kumancev

Build a Scalable Library Management System in Go: Desig...

In this article, we'll dive into designing a Library Management System (LMS) using G...

Dec 26, 2024

Dmitrii Kumancev

Diving into Go: Implementing Classic Elevator Scheduling...

Hi, dudes! Having spent a significant amount of time working with Go, I decided to take on...

Dec 24, 2024

Dmitrii Kumancev

MongoDB-based REST API with Go and integration testing

Developing a REST API that harmonizes well with MongoDB poses a frequent challenge in...

Nov 25, 2023 1

See all from Dmitrii Kumancev

Recommended from Medium

Harendra

How I Am Using a Lifetime 100% Free Server

Get a server with 24 GB RAM + 4 CPU + 200 GB Storage + Always Free

Oct 26, 2024 · 8.5K · 129

Smart byte labs

Organize Like a Pro: A Simple Guide to Go Project Folder...

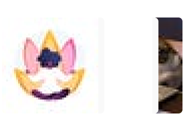
When we talk about folder structure in Golang (or really any programming language), we're...

Oct 25, 2024 · 406 · 3

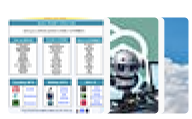
Lists

General Coding Knowledge
20 stories · 1853 saves

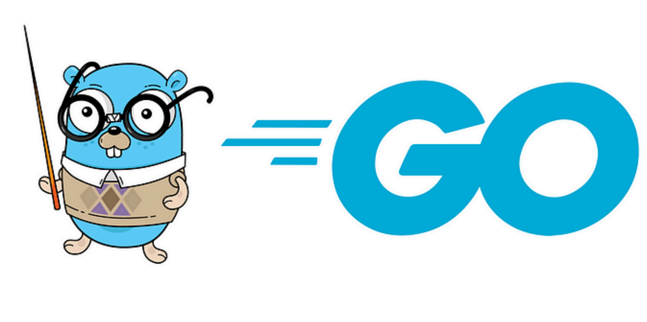
Coding & Development
11 stories · 963 saves



Stories to Help You Grow as a Software Developer
19 stories · 1543 saves



ChatGPT
21 stories · 938 saves



In Go Golang by Yash

8 Golang Performance Tips I Discovered After Years of Coding

These have saved me a lot of headaches, and I think they'll help you too. Don't forget to...

Oct 17, 2024 · 816 · 8

Furkan Türkal

How does Docker actually work? The Hard Way: A Technical Deep...

Unveiling the power of Docker: What is Docker? How does Docker work? Explore th...

Jun 4, 2024 · 732 · 9

In InfoSec Write-ups by Deon van Zyl

You're Not Paranoid: Your Devices Are Always Listening—and Here'...

Think your phone is eavesdropping? You're not imagining things! From smartphones to...

Oct 25, 2024 · 30 · 1

In Level Up Coding by Matt Bentley

My Top 3 Tips for Being a Great Software Architect

My top tips for being a great Software Architect and making the best decisions for...

6d ago · 1.2K · 16

See more recommendations