Sync Package: What Are New Features in Golang sync.Once?

Jan 10, 2024 | Huy Nguyen | Software Development

sync.Once is a synchronization primitive provided by the Golang sync package. 3 new features of Go sync.Once are: OnceFunc, OnceValue, and OnceValues.

Suppose you have some initialization code that should only be executed once, regardless of how many times it's called from different parts of your program or by different goroutines. If you don't have a mechanism to control this, you might end up with race conditions where multiple goroutines are attempting to initialize the same resource concurrently, leading to unpredictable and potentially incorrect behavior.

The Go language provides a simple and efficient way to guarantee that a given function is executed only once. It uses a combination of a boolean flag and a mutex to ensure that only one goroutine executes the specified function, while other goroutines that attempt to execute the same function will wait until the initialization is complete. It call `Sync.Once` .

>> Read more about Golang:

- *Go Tutorial: Golang Basics of Knowledge for Beginners*
- *API Development in Go with Gin Framework*
- *A Comprehensive Guide To Dockerize A Golang Application*
- *Detailed Guide for Simplifying Testing with Golang Testify*
- *Type Conversion in Golang*
- *Understanding Golang Ordered Map with Code Examples*

3 Common Scenarios for Using sync.Once

sync.Once is particularly useful in scenarios where you want to perform a certain initialization or setup operation only once, regardless of how many times that operation is requested. It's commonly used in scenarios like this:

Lazy Initialization

When you want to initialize a resource or perform a setup operation only when it is first needed, and you want to avoid the overhead of repeated initialization.

clike Copy

```
var once sync.Once
var expensiveResource *SomeType

func getExpensiveResource() *SomeType {
    once.Do(func() {
        expensiveResource = initializeExpensiveResource()
    })
    return expensiveResource
}
```

Singleton Pattern

Ensuring that a certain operation, like creating a singleton instance, is performed only once, even in a concurrent environment.

clike Copy

```
var once sync.Once
var instance *Singleton

func GetInstance() *Singleton {
    once.Do(func() {
        instance = createSingletonInstance()
    })
    return instance
}
```

Package-Level Initialization

In package-level variables or initialization functions, where you want to ensure that certain setup code is executed only once when the package is used.

clike Copy

```
package mypackage
```

Table of Contents

Recent Blogs

10 Best Kotlin App Development Companies in Vietnam 2025

Relia Software, SOTATEK JSC,...

Implement Golang Graceful Shutdown: A Hands-on Guide

```
func initialize() {
    // Initialization code here
    initialized = true
}
```

In these cases, `sync.Once` provides a clean and efficient way to handle the one-time initialization, and it ensures that the initialization is safe for concurrent use. It helps avoid race conditions and guarantees that the initialization code is executed exact once.

3 New Features of sync.Once in Go 1.21

The recently introduced `OnceFunc` , `OnceValue` , and `OnceValues` functions encapsulate a common usage of Once, designed for the deferred initialization of a value upon its initial use.

OnceFunc

clike Copy

```
func OnceFunc(f func()) func()
```

`OnceFunc` return a function that supports concurrent calls and can be invoked multiple times.

The following code, `onceVoid` , is only executed once.

clike Copy

```
package main

import (
    "log"
    "sync"
)

func main() {
    onceVoid := func() {
        log.Println("Only once")
    }
    call := sync.OnceFunc(onceVoid)
    for i := 0; i < 10; i++ {
        call()
    }
}
```

clike Copy

```
func OnceValue[T any](f func() T) func() T
```

`OnceValue` the returned function retrieves the result of the initial function call, ensuring that subsequent invocations yield the same value.

In the code below, `randNum` only be executed once, returning the result as `n`, and each call to the bar function will return `n`. bar can be called concurrently.

clike Copy

```
package main

import (
    "log"
    "math/rand"
    "sync"
)

func main() {
    randNum := func() int {
        return rand.Int() % 10
    }
    getNum := sync.OnceValue(randNum)
    for i := 0; i < 10; i++ {
        log.Println(getNum())
    }
```

Result:

clike Copy

```
2023/12/18 22:48:48 4
2023/12/18 22:48:48 4
2023/12/18 22:48:48 4
2023/12/18 22:48:48 4
2023/12/18 22:48:48 4
2023/12/18 22:48:48 4
2023/12/18 22:48:48 4
2023/12/18 22:48:48 4
2023/12/18 22:48:48 4
```

`randNum` return one random value no matter how many `getNum` called that the reason why call many time but still receive value 4.

OnceValues

clike Copy

- The three functions yield 0, 1, and 2 return values, respectively, upon invocation of the associated function.
- The returned functions can be called concurrently.
- Should the execution of function f result in a panic, the returned function will also panic upon subsequent calls, preserving the same panic value as encountered during the execution of f.

clike Copy

```
import (
    "log"
    "math/rand"
    "sync"
)

func main() {
    randTwoNum := func() (int, int) {
        return rand.Int() % 10, rand.Int() % 10
    }
    getTwoNum := sync.OnceValues(randTwoNum)
    for i := 0; i < 10; i++ {
        log.Println(getTwoNum())
    }
}
```

Result:

clike Copy

```
2023/12/18 22:56:52 4 6
2023/12/18 22:56:52 4 6
2023/12/18 22:56:52 4 6
2023/12/18 22:56:52 4 6
2023/12/18 22:56:52 4 6
2023/12/18 22:56:52 4 6
2023/12/18 22:56:52 4 6
2023/12/18 22:56:52 4 6
2023/12/18 22:56:52 4 6
2023/12/18 22:56:52 4 6
```

Same result with `OnceValue` but the different is that return two value only once.

Keep going with `OnceValues` we can receive once more than two value

clike Copy

```
package main

import (
```

```
            return rand.Int() % 10, rand.Int() % 10
        })
    })

    for i := 0; i < 10; i++ {
```

Result:

clike Copy

```
2023/12/18 23:09:28 8 4 9
2023/12/18 23:09:28 8 4 9
2023/12/18 23:09:28 8 4 9
2023/12/18 23:09:28 8 4 9
2023/12/18 23:09:28 8 4 9
2023/12/18 23:09:28 8 4 9
2023/12/18 23:09:28 8 4 9
2023/12/18 23:09:28 8 4 9
2023/12/18 23:09:28 8 4 9
2023/12/18 23:09:28 8 4 9
```

`OnceValues` returns a function that invokes `f` only once and returns the values returned by `f`. The returned function may be called concurrently. If `f` panics, the returned function will panic with the same value on every call.

>> You may be interested in these Golang-related blogs:

- *Best Practices For Dependency Inversion in Golang*
- *Detailed Code Examples of Dependency Inversion in Golang*
- *Hands-On Implementation for Dependency Injection in Go*
- *Practical SOLID in Golang: Single Responsibility Principle*

Conclusion

In conclusion, the introduction of the new functions, **`OnceFunc`**, **`OnceValue`**, and **`OnceValues`**, in Go's **`sync.Once`** package is a significant enhancement that adds valuable flexibility and convenience to the already powerful synchronization primitive.

Understanding and leveraging these new functions is crucial for Go developers, as they provide elegant solutions for deferred initialization scenarios with varying degrees of complexity.

- The ability to use **`OnceFunc`** enables developers to encapsulate operations that need to be executed only once, creating a function that supports concurrent calls and can be invoked multiple times. This feature is particularly useful in scenarios where an operation should be performed exactly once, but the result is not needed immediately.

as they address common challenges in concurrent programming, enhance code readability, and provide a more expressive and efficient way to handle deferred initialization. As the Go language evolves, staying informed and adopting these improvements will contribute to writing more robust, scalable, and maintainable Go code.

>>> Follow and Contact Relia Software for more information!

The Author

Huy Nguyen - Golang Developer

A Golang Developer with over 5 years of experience.

Hello! I'm Huy Nguyen, a digital craftsman turning lines of code into engaging, high-performance systems. With over 5 years of experience, I've had the privilege of contributing to major projects at companies like VNG Corporation, Ninja Van, and Topebox. My expertise lies in backend development, and I'm proficient in technologies such as Golang, GCP, AWS, Kubernetes, and event-driven architecture. My work has consistently resulted in significant improvements in system performance and scalability. I hold a B.S.E in Computer Science Engineering from the Industrial University of Ho Chi Minh and am passionate about leveraging technology to solve complex problems and create seamless digital experiences.

golang coding development

Share

Understanding Golang Ordered Map with Code Examples

Next Post

Related Blogs

Software Development Jun 27, 2025

golang-migrate Tutorial: Implementing Database Migration in Go

golang-migrate is a database migration tool for Go applications, providing both a command-line interface and a library for managing database schema changes.

Software Development Jun 26, 2025

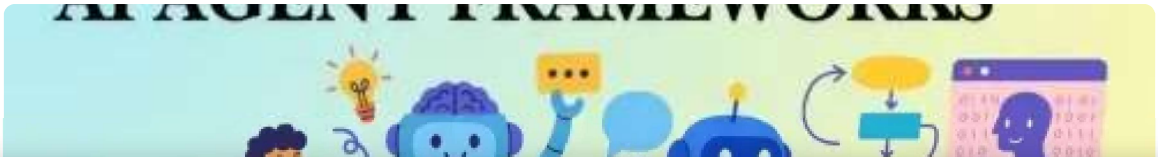
10+ JavaScript Project Ideas for Beginners to Advanced Coders

Digital clock, simple image slider, age calculator, recipe book app, e-commerce product page, etc, are popular Javascript project ideas for developers.

Software Development Jun 16, 2025

A Complete Flutter Firebase Tutorial: Build a Photo Sharing App

Learn how to integrate Firebase services into a Flutter app (a Photo Sharer), focusing on Firebase Authentication, Cloud Firestore, and Firebase Storage.



Software Development

Jun 13, 2025

9 Key Logistics Technology Trends for Businesses 2025

AI & ML, IoT, blockchain, automation and robotics, digital twins, advanced data analytics, cloud computing & SaaS, etc, are 9 logistic technology trends in 2025