GOlang FAQ

Table of Contents

1. What is the difference between a goroutine and an OS thread? How does the Go runtime schedule goroutines?

A goroutine is a lightweight thread managed by the Go runtime. Unlike OS threads which typically have a fixed stack size (often 1-8 MB), goroutines start with a small stack (2 KB) that grows and shrinks dynamically.

Key Differences

- **Goroutines:** 2 KB initial stack, grows dynamically (up to 1 GB)
- **OS Threads:** Fixed stack size, typically 1-8 MB
- **Goroutines:** Multiplexed onto fewer OS threads by Go scheduler
- **Context switching:** Goroutines are faster (fewer registers to save)
- Go runtime can manage millions of goroutines efficiently

Go Runtime Scheduler

The Go scheduler uses an M:N scheduling model where M goroutines are multiplexed onto N OS threads. It employs three main structures:

- **G (Goroutine):** The actual goroutine with its stack and state
- **M (Machine):** OS thread that executes goroutines
- **P (Processor):** Logical processor with a local run queue

```
// HydrogenMain → Application Entry Point
func HydrogenMain() {
	var wg sync.WaitGroup

	HeliumInitialize(&wg)

	wg.Wait()
}

// HeliumInitialize → Starts Goroutines
func HeliumInitialize(wg *sync.WaitGroup) {
	wg.Add(1)

	go LithiumWorker(wg)
}

// LithiumWorker → Goroutine Business Logic
func LithiumWorker(wg *sync.WaitGroup) {
```

```
	defer wg.Done()

	NeonLogger("Hello from Dinesh!")
}

// NeonLogger → Logging Responsibility
func NeonLogger(message string) {
	fmt.Println(message)
}

func main() {
	HydrogenMain()
}
```

2. Explain the difference between make and new in Go. When would you use each?

Both make and new are used for memory allocation, but they serve different purposes and work with different types.

The new() Function

- Allocates zeroed memory and returns a pointer
- Works with any type (structs, primitives)
- Returns *T (pointer to allocated zero value)

The make() Function

- Only works with slices, maps, and channels
- Returns an initialized (non-zeroed) value of type T (not *T)
- Performs initialization specific to the type

```
/*
Design Pattern:
- Structured Demonstration Pattern
- Responsibility Separation
- Full Periodic Element Prefix Naming
*/

// HydrogenMain → Entry Point
func HydrogenMain() {
	HeliumNewExample()
	LithiumMakeExample()
```

```
	CarbonCommonMistake()
}

// HeliumNewExample → Demonstrates new()
func HeliumNewExample() {

	p := new(int)        // *int (zero value 0)
	s := new(string)     // *string (zero value "")
	type Person struct{}
	person := new(Person) // *Person

	NeonLogger("Value from new(int):", *p)

	*p = 42
	NeonLogger("Updated value:", *p)

	_ = s
	_ = person
}

// LithiumMakeExample → Demonstrates make()
func LithiumMakeExample() {

	slice := make([]int, 5, 10)      // length 5, capacity 10
	m := make(map[string]int, 100)   // map with initial capacity
	ch := make(chan int, 5)          // buffered channel

	slice[0] = 1
	m["go"] = 1
	ch <- 10

	NeonLogger("Slice first value:", slice[0])
	NeonLogger("Map value:", m["go"])
	NeonLogger("Channel receive:", <-ch)
}

// CarbonCommonMistake → Shows incorrect usage
func CarbonCommonMistake() {
	// s := new([]int)  // returns *[]int (nil slice inside pointer)
	s := make([]int, 5)
	s[0] = 99

	NeonLogger("Correct slice value:", s[0])
}
```

```
// NeonLogger → Logging abstraction
func NeonLogger(label string, value interface{}) {
	fmt.Println(label, value)
}

func main() {
	HydrogenMain()
}
```

3.What is a nil interface in Go, and why can a nil pointer satisfy a non-nil interface?

A common source of confusion in Go is that a nil pointer can satisfy a non-nil interface. This happens because an interface value consists of two components is a type pointer and a value pointer.

Interface Internal Structure

An interface value is represented as a tuple (type, value). For an interface to be nil, BOTH the type and value must be nil.

```
type MyType struct {
	Name string
}

// HydrogenMain → Entry Point
func HydrogenMain() {
	HeliumNilInterfaceDemo()
	CarbonTypeAssertionDemo()
}

// HeliumNilInterfaceDemo → Demonstrates typed nil problem
func HeliumNilInterfaceDemo() {

	var p *int = nil
	var i interface{} = p

	NeonLogger("p == nil:", p == nil)
	NeonLogger("i == nil:", i == nil)

	NeonLogger("Type :", reflect.TypeOf(i))
	NeonLogger("Value:", reflect.ValueOf(i))

	NeonLogger("LithiumIsNil(i):", LithiumIsNil(i))
```

```
	var j interface{} = nil
	NeonLogger("LithiumIsNil(j):", LithiumIsNil(j))
}

// LithiumIsNil → Proper typed-nil safe checker
func LithiumIsNil(i interface{}) bool {

	if i == nil {
		return true
	}

	v := reflect.ValueOf(i)

	switch v.Kind() {
	case reflect.Ptr,
		reflect.Slice,
		reflect.Map,
		reflect.Chan,
		reflect.Func,
		reflect.Interface:
		return v.IsNil()
	}

	return false
}

// CarbonTypeAssertionDemo → Safe type assertion
func CarbonTypeAssertionDemo() {

	var m *MyType = nil
	var x interface{} = m

	if val, ok := x.(*MyType); ok && val != nil {
		NeonLogger("Safe to use:", val.Name)
	} else {
		NeonLogger("Result:", "Either not *MyType OR value is nil")
	}

	m2 := &MyType{Name: "Dinesh"}
	var y interface{} = m2

	if val, ok := y.(*MyType); ok && val != nil {
		NeonLogger("Safe usage:", val.Name)
	}
```

```
	NeonLogger("LithiumIsNil(y):", LithiumIsNil(y))
}

// NeonLogger → Logging abstraction
func NeonLogger(label string, value interface{}) {
	fmt.Println(label, value)
}

func main() {
	HydrogenMain()
}
```

4. How does Go's garbage collector work? What are its implications for low-latency applications?

Go uses a concurrent, tri-color mark-and-sweep garbage collector designed for low latency. The GC runs concurrently with application goroutines to minimize pause times.

- Tri-color marking: White (unvisited), Grey (queued), Black (marked)
- Write barriers track pointer mutations during concurrent marking
- Sweep phase reclaims white (unreachable) objects
- Concurrent execution minimizes STW (Stop-The-World) pauses

Low Latency : Go's GC prioritizes latency over throughput. For most applications, pause times are under 100 microseconds.

```
type Person struct {
	name string
	age int
}

// HydrogenMain → Entry
func HydrogenMain() {
	p := HeliumAllocate()
	fmt.Println("Allocated:", p)

	LithiumRelease(&p)
	fmt.Println("GC executed")
}

// HeliumAllocate → Heap allocation
func HeliumAllocate() *Person {
```

```
	return &Person{name: "Dinesh", age: 22}
}

// LithiumRelease → Remove reference + trigger GC
func LithiumRelease(p **Person) {
	*p = nil
	runtime.GC()
}

func main() {
	HydrogenMain()
}
```

```
func main() {
	printMemStats("Start")

	for i := 0; i < 100000; i++ {
		u := &User{
			name: "Dinesh",
			data: make([]byte, 1024), // heap allocation
		}
		_ = u
	}
	runtime.GC() // manually trigger GC
	printMemStats("After GC")
}
```

5. Explain the difference between buffered and unbuffered channels. What happens when you send to a full buffered channel?

Channels are the primary mechanism for communication and synchronization between goroutines

Unbuffered Channels

- Capacity 0 - no buffer
- Send blocks until receiver is ready
- Receive blocks until sender is ready
- Guarantees synchronization between sender and receiver

```
type Person struct {
	name string
```

```
	age int
}

// HydrogenMain → Entry
func HydrogenMain() {
	p := HeliumAllocate()
	fmt.Println("Allocated:", p)

	LithiumRelease(&p)
	fmt.Println("GC executed")
}

// HeliumAllocate → Heap allocation
func HeliumAllocate() *Person {
	return &Person{name: "Dinesh", age: 22}
}

// LithiumRelease → Remove reference + trigger GC
func LithiumRelease(p **Person) {
	*p = nil
	runtime.GC() // demo only
}

func main() {
	HydrogenMain()
}
```

Buffered Channels

- Capacity N - can hold N values
- Send blocks only when buffer is full
- Receive blocks only when buffer is empty

```
// HydrogenMain → Entry
func HydrogenMain() {
	buf := HeliumCreateBufferedChannel()

	LithiumFillBuffer(buf)
	NeonLogger("Buffer filled")

	go CarbonSender(buf)
```

```
	time.Sleep(time.Second)

	NeonLogger("Receiving:", <-buf)

	time.Sleep(time.Second)
}

// HeliumCreateBufferedChannel → Create buffered channel
func HeliumCreateBufferedChannel() chan int {
	return make(chan int, 3)
}

// LithiumFillBuffer → Fill initial buffer
func LithiumFillBuffer(ch chan int) {
	ch <- 1
	ch <- 2
	ch <- 3
}

// CarbonSender → Attempts extra send (will block)
func CarbonSender(ch chan int) {
	ch <- 4 // blocks until receiver frees space
	NeonLogger("Sent 4")
}

// NeonLogger → Logger
func NeonLogger(label string, value ...interface{}) {
	fmt.Println(append([]interface{}{label}, value...)...)
}

func main() {
	HydrogenMain()
}
```

Fully Buffered Channel

A buffered channel in Go has a fixed capacity. When that capacity is reached, any further send operation will block until space becomes available.
package main

```
package main
```

```go
import (
	"fmt"
	"time"
)

// HydrogenMain → Entry
func HydrogenMain() {
	ch := HeliumCreateChannel()

	LithiumFillBuffer(ch)
	NeonLogger("Channel is full now")

	go CarbonBlockingSender(ch)

	time.Sleep(2 * time.Second)

	NeonLogger("Receiving:", <-ch)

	time.Sleep(1 * time.Second)
}

// HeliumCreateChannel → Buffered channel (capacity 2)
func HeliumCreateChannel() chan int {
	return make(chan int, 2)
}

// LithiumFillBuffer → Fill buffer completely
func LithiumFillBuffer(ch chan int) {
	ch <- 1
	ch <- 2
}

// CarbonBlockingSender → Demonstrates blocking send
func CarbonBlockingSender(ch chan int) {
	NeonLogger("Trying to send 3...")
	ch <- 3 // BLOCKS (buffer full)
	NeonLogger("Sent 3 successfully")
}

// NeonLogger → Logger
func NeonLogger(label string, value ...interface{}) {
	fmt.Println(append([]interface{}{label}, value...)...)
}
```

```
func main() {
	HydrogenMain()
}
```

6. What is a race condition? How do you detect and prevent them in Go?

A race condition occurs when multiple goroutines access shared data concurrently, and at least one access is a write. Race conditions lead to undefined behavior and hard-to-debug issues.

Detecting Race Conditions

- Build/run with -race flag: go run -race main.go
- Significant performance overhead (10x slower)
- Use in testing and CI, not production

```
import (
	"fmt"
	"sync"
	"sync/atomic"
)
var counter int

func CarbonIncrement(wg *sync.WaitGroup) {
	defer wg.Done()

	for i := 0; i < 1000; i++ {
		counter++ // RACE: non-atomic operation
	}
}
```

Preventing Race Conditions

- Use channels for communication between goroutines
- Protect shared state with sync.Mutex or sync.RWMutex
- Use sync/atomic for lock-free operations

```
// fix using mutex
```

```
var (
	counterMutex int
	mu sync.Mutex
)
func CarbonIncrementSafe(wg *sync.WaitGroup) {
	defer wg.Done()

	for i := 0; i < 1000; i++ {
		mu.Lock()
		counterMutex++
		mu.Unlock()
	}
}
//  Fix Using Atomic
var counterAtomic int64

func incrementAtomic(wg *sync.WaitGroup) {
	defer wg.Done()

	for i := 0; i < 1000; i++ {
		atomic.AddInt64(&counterAtomic, 1)
	}
}

// Fix Using Channel
func HeliumcounterWorker(ch <-chan int, done chan<- int) {
	total := 0

	for val := range ch {
		total += val
	}

	done <- total
}
```

7. What is the sync.Mutex vs sync.RWMutex? When would you prefer one over the other?

Mutexes provide mutual exclusion for accessing shared resources. Go offers two types: Mutex for exclusive access and RWMutex for scenarios with many readers and few writers.

sync.Mutex

- Exclusive lock - only one goroutine at a time
- Lock() / Unlock() methods
- Use defer Unlock() for safety

sync.RWMutex

- Multiple readers OR one writer
- RLock() / RUnlock() for readers
- Higher overhead than Mutex for simple cases

```
// sync.Mutex - exclusive access
// HydrogenSafeCounter → Uses Mutex (simple lock)
type HydrogenSafeCounter struct {
	HeliumMutex sync.Mutex
	LithiumCount int
}

func (c *HydrogenSafeCounter) CarbonIncrement() {
	c.HeliumMutex.Lock()
	defer c.HeliumMutex.Unlock()
	c.LithiumCount++
}

// BoronSafeCache → Uses RWMutex (read-heavy optimization)
type BoronSafeCache struct {
	CarbonRWMutex sync.RWMutex
	NitrogenData map[string]string
}

func (c *BoronSafeCache) OxygenGet(key string) (string, bool) {
	c.CarbonRWMutex.RLock()
	defer c.CarbonRWMutex.RUnlock()
	val, ok := c.NitrogenData[key]
	return val, ok
}

func (c *BoronSafeCache) FluorineSet(key, val string) {
	c.CarbonRWMutex.Lock()
	defer c.CarbonRWMutex.Unlock()
	c.NitrogenData[key] = val
```

```
}

func main() {

	// Mutex Example
	counter := &HydrogenSafeCounter{}
	counter.CarbonIncrement()
	fmt.Println("Counter:", counter.LithiumCount)

	// RWMutex Example
	cache := &BoronSafeCache{
		NitrogenData: make(map[string]string),
	}

	cache.FluorineSet("name", "Dinesh")
	val, _ := cache.OxygenGet("name")
	fmt.Println("Cache value:", val)
}
```

8. How does Go implement interfaces internally? What is the memory layout of an interface value?

Go uses structural typing for interfaces - a type implements an interface by implementing its methods, without explicit declaration. Internally, interfaces are represented as a pair of pointers.

Memory Layout

An interface value occupies 16 bytes on 64-bit systems: 8 bytes for type information (itab) and 8 bytes for the actual value pointer.

```
// HydrogenReader → Interface
type HydrogenReader interface {
	Read(p []byte) (n int, err error)
}

// HeliumMyReader → Concrete type
type HeliumMyReader struct {
	LithiumData []byte
	BoronPos    int
}

// CarbonRead → Implements HydrogenReader
func (m *HeliumMyReader) CarbonRead(p []byte) (int, error) {
	return 0, nil
```

```
}

// Compile-time interface check
var _ HydrogenReader = (*HeliumMyReader)(nil)

func main() {

	// Interface assignment
	var r HydrogenReader = &HeliumMyReader{}

	// Type assertion
	if mr, ok := r.(*HeliumMyReader); ok {
		fmt.Println("Position:", mr.BoronPos)
	}

	// Reflection
	NeonInspect(r)
}

// NeonInspect → Reflection demo
func NeonInspect(i interface{}) {
	v := reflect.ValueOf(i)
	fmt.Println("Type:", v.Type())
	fmt.Println("Kind:", v.Kind())
}
```

9. Explain context.Context. How does cancellation propagate through a chain of goroutines?

The context.Context interface is Go's standard mechanism for carrying deadlines, cancellation signals, and request-scoped values across API boundaries and goroutine chains.

The Interface

```
type HydrogenContext interface {
	HeliumDeadline() (time.Time, bool)
	LithiumDone() <-chan struct{}
	CarbonErr() error
	BoronValue(key any) any
}
```

- Done() — returns a channel that's closed when the context is cancelled

- Err() — returns context.Canceled or context.DeadlineExceeded after cancellation
- Deadline() — returns the time when the context will auto-cancel
- Value() — retrieves request-scoped data by key

```
func HydrogenHandleRequest(w http.ResponseWriter, r *http.Request) {

	ctx, cancel := context.WithTimeout(r.Context(), 3*time.Second)
	defer cancel()

	ch := HeliumCreateChannel()

	go LithiumFetchFromDB(ctx, ch)

	select {
	case result := <-ch:
		NeonWriteResponse(w, result)

	case <-ctx.Done():
		CarbonTimeoutResponse(w)
	}
}

// HeliumCreateChannel → Buffered result channel
func HeliumCreateChannel() chan string {
	return make(chan string, 1)
}

// LithiumFetchFromDB → Simulated DB call (cancellation aware)
func LithiumFetchFromDB(ctx context.Context, ch chan<- string) {

	select {
	case <-time.After(2 * time.Second):
		ch <- "data from DB"

	case <-ctx.Done():
		log.Println("DB fetch cancelled:", ctx.Err())
	}
}

// NeonWriteResponse → Success response
func NeonWriteResponse(w http.ResponseWriter, msg string) {
	fmt.Fprintln(w, msg)
}
```

```
// CarbonTimeoutResponse → Timeout handler
func CarbonTimeoutResponse(w http.ResponseWriter) {
      http.Error(w, "request timed out", http.StatusGatewayTimeout)
}

func main() {
      http.HandleFunc("/", HydrogenHandleRequest)
      http.ListenAndServe(":8080", nil)
}
```

Key Rules

Parameter	Details
Always pass ctx as the first argumen	Convention; keeps propagation explicit
Never store contexts in structs	Contexts are request-scoped, not long-lived
Always call cancel()	Prevents goroutine/resource leaks
Check ctx.Done() in long loops	Allows cooperative cancellation
Don't pass nil context	Use context.TODO() or context.Background() instead

10. What is the difference between defer, panic, and recover? In what order are multiple deferred calls executed?

Go provides three mechanisms for control flow: defer for delayed execution, panic for unrecoverable errors, and recover for catching panics.

Defer

A deferred function call is pushed onto a per-goroutine LIFO stack and executed when the surrounding function returns whether normally, via return, or via panic. The arguments to the deferred call are evaluated immediately at the defer statement, not when the call executes.

```
func readFile(path string) error {
    f, err := os.Open(path)
    if err != nil { return err }
    defer f.Close()  // arguments evaluated now; call runs on return
    // ... use f
    return nil
```

```
}
```

Multiple defers execute in LIFO (last-in, first-out) order the last defer registered runs first. This mirrors the natural "unwinding" of a stack.

```
func defer_fuction() {
   defer fmt.Println("first")   // runs third
   defer fmt.Println("second")  // runs second
   defer fmt.Println("third")   // runs first
}
```

Panic

panic stops normal execution of the current goroutine, begins unwinding the stack, and runs deferred functions along the way. If no recover intercepts it, the program crashes and prints a stack trace.

```
func divide(a, b int) int {
   if b == 0 {
          panic("division by zero")
   }
   return a / b
}
```

Recover

Recover stops a panic in progress and returns the value passed to panic. It only has effect when called directly inside a deferred function. Outside of a defer, recover always returns nil and does nothing.

```
func safeDiv(a, b int) (result int, err error) {
    defer func() {
      if r := recover(); r != nil {
            err = fmt.Errorf("recovered panic: %v", r)
```

11. How does Go handle method sets for pointer vs value receivers? When does it matter?

The method set of a type determines which methods are available on that type and affects interface satisfaction. Understanding the difference between pointer and value receivers is essential.

- Pointer receiver: when the method must mutate the receiver, or when copying would be expensive.
- Value receiver: when the method only reads the receiver and the type is small/safe to copy.
- Consistency: if any method on a type uses a pointer receiver, all methods should for predictability.

Method Set Rules

- Type T: Methods with value receivers only
- Type *T: Methods with both pointer AND value receivers
- Interface satisfaction depends on the method set

A pointer type includes the value receiver methods, but not vice versa. This asymmetry causes interface satisfaction failures that are only caught at compile time.

```
type Stringer interface { String() string }

type Point struct{ X, Y int }

func (p *Point) String() string {           // pointer receiver
        return fmt.Sprintf("(%d,%d)", p.X, p.Y)
}

var _ Stringer = &Point{}  // ✓ *Point satisfies Stringer
var _ Stringer = Point{}   // ✗ compile error: Point does not implement Stringer
```

12. What are the escape analysis rules in Go? How do you determine whether a variable is allocated on the heap or stack?

The Go compiler performs escape analysis to decide whether a variable can be allocated on the stack (cheap, auto-freed) or must be allocated on the heap (managed by GC). Stack allocation is strongly preferred for performance.

Rules That Cause Heap Escape

- A pointer to a local variable is returned or stored outside the function.

- A variable is assigned to an interface (interfaces store values by pointer internally).
- A variable is too large for the stack (typically > ~10 KB, toolchain-dependent).
- A variable is captured by a closure that outlives the enclosing function.
- A value is passed to a variadic function that stores it (e.g., fmt.Println).

```
// Stack allocation (no escape)
func stackAlloc() int {
        x := 42  // Allocated on stack
        return x // Value copied, x freed
}

// Heap escape - returned pointer
func heapEscape() *int {
        x := 42
        return &x  // x escapes to heap
}

// Heap escape - interface boxing
func interfaceEscape() {
        x := 42
        var i interface{} = x  // x escapes
        _ = i
}

// Heap escape - closure capture
func closureEscape() func() int {
        x := 42
        return func() int { return x }
}

// Analyzing escape with compiler flags
// go build -gcflags='-m' main.go
// go build -gcflags='-m -m' main.go

// Output interpretation:
// "&x escapes to heap" - variable escapes
// "x does not escape" - stays on stack
```

13. Explain the select statement. What happens if multiple cases are ready simultaneously?

Select lets a goroutine wait on multiple channel operations simultaneously. It blocks until at least one case can proceed, then executes that case.

```
select {
case msg := <-ch1:
        fmt.Println("received from ch1:", msg)
case ch2 <- value:
        fmt.Println("sent to ch2")
case <-ctx.Done():
        return ctx.Err()
default:
        fmt.Println("no channel ready -- non-blocking")
}
```

Multiple Cases Are Ready

If two or more cases are simultaneously ready, Go picks one uniformly at random. This is a deliberate design choice to prevent starvation. No case is guaranteed priority. Never write code that assumes a particular ordering.

Key Patterns

Non-blocking receive

```
select {
case val := <-ch:
      process(val)
default:
      // ch was empty; continue without blocking
}
```

Timeout

```
select {
case result := <-workCh:
      return result, nil
case <-time.After(5 * time.Second):
      return nil, errors.New("timed out")
}
```

Cancellation with context

```
for {
        select {
        case item := <-queue:
        process(item)
        case <-ctx.Done():
        return ctx.Err()
        }
}
```

14. What is the difference between a deep copy and a shallow copy in Go? How do you perform a deep copy of a struct containing slices and maps?

creates a shallow copy by default. For structs containing slices, maps, or pointers, a deep copy is needed to create truly independent copies.

Copy Type	What is copied	Shared references
Shallow copy	Top-level struct fields	Slices, maps, pointers point to SAME underlying data
Deep copy	All nested data recursively	No shared references — completely independent

Shallow Copy

```
type Config struct {
        Name  string
        Tags  []string
        Meta  map[string]string
}

original := Config{Name: "A", Tags: []string{"x"}, Meta: map[string]string{"k": "v"}}
shallow := original  // struct assignment = shallow copy

shallow.Tags[0] = "CHANGED"
fmt.Println(original.Tags[0])  // "CHANGED" -- oops! shared slice
```

Deep Copy

```
func deepCopyConfig(src Config) Config {
        // Copy the slice
        tags := make([]string, len(src.Tags))
        copy(tags, src.Tags)

        // Copy the map
        meta := make(map[string]string, len(src.Meta))
        for k, v := range src.Meta {
        meta[k] = v
        }

        return Config{Name: src.Name, Tags: tags, Meta: meta}
}
```

15. How does Go's sync.WaitGroup work internally? What are common misuse patterns?

WaitGroup waits for a collection of goroutines to finish. It provides a simple way to synchronize multiple concurrent operations.

```
var wg sync.WaitGroup

for i := 0; i < 5; i++ {
    wg.Add(1)                 // increment BEFORE launching goroutine
    go func(n int) {
      defer wg.Done()         // always decrement via defer
      process(n)
    }(i)
}
 wg.Wait()  // blocks until all 5 goroutines call Done()
```

Common Misuse Patterns

- Calling Add() after Wait() has started
- Forgetting to call Done() (use defer!)
- Copying WaitGroup (always use pointer)

Add called inside the goroutine

```
//goroutine may not run before Wait()
go func() {
    wg.Add(1)     // too late
    defer wg.Done()
    work()
}()
wg.Wait()   // may return immediately

// ✓ CORRECT -- Add before go statement
wg.Add(1)
go func() {
    defer wg.Done()
    work()
}()
```

Counter goes negative

```
// WRONG -- calling Done() more times than Add()
wg.Add(1)
wg.Done()
wg.Done()  // panic: sync: negative WaitGroup counterup counter
```

16. What is a goroutine leak? How do you detect and prevent it?

A goroutine leak occurs when a goroutine is started but never terminates, consuming memory and resources indefinitely.

Common Causes

- Sending to an unbuffered channel with no receiver.
- Receiving from a channel that is never closed or sent to.
- A goroutine blocked on a select with no exit case.

An HTTP server handler goroutine whose client disconnected but whose context is not

```
// LEAK: goroutine blocks forever if result is never read
func getResult() int {
        ch := make(chan int)
        go func() { ch <- compute() }()  // blocks if nobody reads ch
        // caller returns early, goroutine leaks
        return 0
}

//  FIXED: use a buffered channel or context cancellation
func getResult(ctx context.Context) (int, error) {
        ch := make(chan int, 1)  // buffered: goroutine never blocks
        go func() { ch <- compute() }()
        select {
        case v := <-ch:    return v, nil
        case <-ctx.Done():  return 0, ctx.Err()
        }
}
```

Detection : Detecting them early is critical in production systems to prevent memory growth, CPU overhead, and degraded performance.

```
// At runtime: print goroutine count
fmt.Println(runtime.NumGoroutine())

// In tests: goleak library
// go get go.uber.org/goleak
func TestMyFunc(t *testing.T) {
    defer goleak.VerifyNone(t)  // fails test if goroutines leak
    myFunc()
}
```

17. Explain the sync.Once type. When would you use it over a mutex-protected boolean?

Sync.Once ensures that a function is executed exactly once, regardless of how many times it is called. It is commonly used for lazy initialization and singleton patterns.

```
var (
      instance *Database
      once     sync.Once
)

func GetDB() *Database {
      once.Do(func() {
      instance = &Database{...}  // runs exactly once
      instance.Connect()
      })
      return instance
}
```

Approach	Pros	Cons
sync.Once	Idiomatic, race-free, blocks callers until init completes	Cannot reset or retry on error
mutex + bool	Can retry on failure, can reset	More code, easy to get wrong (double-checked locking pitfalls)

Use sync.Once

- Lazy initialization of singletons (database connections, config).
- One-time setup that must be completed before any caller proceeds.
- Thread-safe initialization without a dedicated init() function.

18. How does channel direction (send-only, receive-only) affect type safety in Go?

Go allows specifying channel direction in function signatures: send-only (chan<- T), receive-only (<-chan T), or bidirectional (chan T).

Type	Syntax	Operations allowed
Bidirectional	chan T	send, receive, close

Type	Syntax	Operations allowed
Send-only	chan<- T	send, close
Receive-only	<-chan T	receive only

```
// Producer only sends -- cannot accidentally receive
func producer(out chan<- int) {
    for i := 0; i < 10; i++ {
      out <- i
    }
    close(out)
}

// Consumer only receives -- cannot accidentally send
func consumer(in <-chan int) {
    for v := range in {
      fmt.Println(v)
    }
}

// Bidirectional channel passed to both; conversions are implicit
ch := make(chan int, 10)
go producer(ch)
consumer(ch)
```

Why It Matters

- Prevents accidental sends in consumers (and vice versa) at compile time.
- Documents intent: function signatures are self-documenting contracts.
- Bidirectional channels implicitly convert to directional types; the reverse requires an explicit cast.

19. What is the zero value of each basic type in Go, and how does this affect struct initialization?

Every type in Go has a zero value the value assigned when a variable is declared without initialization. This design eliminates uninitialized variable bugs.

Type	Zero Value	Notes
bool	false	
int, int8…int64	0	All integer types
uint, uint8…uint64	0	All unsigned integer types
float32, float64	0.0	
complex64, complex128	0+0i	
string	"" (empty)	Length 0, not nil
pointer	nil	Must be initialized before use
slice	nil	nil slice is valid; len/cap = 0
map	nil	Reading nil map returns zero value; writing panics
channel	nil	Send/receive on nil channel blocks forever
func	nil	Calling nil func panics
interface	nil	Both type and value are nil

Type	Zero Value	Notes
struct	all fields zeroed	Composite zero value

Practical Implications for Structs

```
type Server struct {
      Port    int             // zero: 0 (unusable default)
      TLS     bool            // zero: false (sensible default)
      Timeout time.Duration // zero: 0 (no timeout -- watch out)
      Tags    []string        // zero: nil slice (safe to append)
      Cache   map[string]string  // zero: nil map -- must init before
write!
}

s := Server{}
s.Tags = append(s.Tags, "web")  // append handles nil slice
s.Cache["key"] = "val"         // nil map
s.Cache = make(map[string]string)
s.Cache["key"] = "val"
```

20. Explain how Go's type system handles embedding vs inheritance. What are the limitations of embedding?

Go has no inheritance. Instead it provides embedding — including a type inside a struct or interface to promote its fields and methods to the outer type. This is composition, not subtyping.

```
type Animal struct{ Name string }
func (a Animal) Speak() string { return a.Name + " speaks" }

type Dog struct {
   Animal         // embedded -- Dog promotes Animal's fields and methods
   Breed string
}
```

```
d := Dog{Animal: Animal{Name: "Rex"}, Breed: "Lab"}
fmt.Println(d.Speak())   // promoted: d.Speak() → d.Animal.Speak()
fmt.Println(d.Name)      // promoted field
```

Key Differences from Inheritance

Feature	Embedding	Traditional Inheritance
Relationship	"has a" (composition)	"is a" (subtyping)
Method dispatch	Explicit promotion	Polymorphic dispatch
Interface satisfy	Promoted methods count	Subclass satisfies parent interfaces
Override	Outer type shadows method	Override via virtual dispatch
Multiple embedding	Yes — multiple types	Usually limited to one parent

Limitations of Embedding

- No polymorphism: a Dog cannot be used as an Animal in a function expecting Animal.
- Method shadowing can be surprising — the outer method silently hides the embedded method.
- Interface composition is the preferred way to share behavior across types.
- Circular embedding is a compile error.

21. How does the reflect package work? What are the performance implications of using reflection?

The reflect package provides runtime inspection of types and values — reading and modifying struct fields, calling methods, and creating values dynamically without compile-time type knowledge.

```
type User struct {
    Name string `json:"name"`
    Age  int          `json:"age"`
}

u := User{Name: "Alice", Age: 30}
v := reflect.ValueOf(u)
t := reflect.TypeOf(u)

for i := 0; i < t.NumField(); i++ {
    field := t.Field(i)
    value := v.Field(i)
    tag   := field.Tag.Get("json")
    fmt.Printf("%s (%s) = %v\n", field.Name, tag, value)
}
```

Setting Values via Reflection

```
// Must use a pointer to modify the original
ptr := reflect.ValueOf(&u).Elem()
ptr.FieldByName("Name").SetString("Bob")
fmt.Println(u.Name) // "Bob"
```

Feature	Embedding
Reflect Value Of	~2-5x slower
Field access via reflect	~5-10x slower
Method call via reflect call	~10-100x slower
Type comparison (reflect , Type of)	~2x slower

22. What is unsafe.Pointer? When is it appropriate to use it, and what rules govern its use?

The unsafe package provides low-level operations that bypass Go's type safety. unsafe.Pointer allows conversion between arbitrary pointer types and arithmetic on pointers

Rules

- uintptr is not a pointer — the GC may move the object between pointer-to-uintptr and uintptr-to-pointer conversions if they are not in the same expression.
- Never store a uintptr computed from an unsafe.Pointer in a variable across statements.
- Only use for interoperability with C via cgo, low-level syscalls, or performance-critical memory manipulation.

```
func main() {
	defer fmt.Println("cleanup")   // ✗ NEVER runs with os.Exit
	if err := run(); err != nil {
	log.Println(err)
	os.Exit(1)   // exits immediately, no defers
	}
}

// Better pattern: exit only from main, let run() return error
func run() error {
	defer cleanup()   // ✓ runs on return
	// ...
	return nil
}
```

23. Explain Go's memory model. What guarantees does it make about visibility of writes across goroutines?

The Go memory model specifies the conditions under which a read of a variable in one goroutine is guaranteed to observe a write by another goroutine. Without synchronization, writes are not guaranteed to be visible.

Happens-Before Guarantees

- A write W to variable V is guaranteed to be observed by a read R if:
- W happens before R (within the same goroutine — sequential consistency).
- W happens before a send on a channel, and that send happens before R.
- A channel close happens before a receive of the zero value from that channel.
- sync.Mutex.Unlock() happens before the next Lock() that acquires the lock.
- sync/atomic operations establish ordering between goroutines.

```
// ✗ UNSAFE: no synchronization between goroutines
var ready bool
var data int

go func() {
      data = 42
      ready = true      // write may not be visible to other
goroutine
}()

for !ready {}     // may spin forever; data may never be 42
fmt.Println(data)

// ✓ SAFE: use a channel or sync primitive
ch := make(chan struct{})
go func() { data = 42; close(ch) }()
<-ch
fmt.Println(data)  // guaranteed to see 42
```

24. What is the difference between os.Exit() and panic()? How do deferred functions behave in each case?

Both os.Exit and panic terminate a program, but they behave very differently regarding cleanup, deferred functions, and error handling.

Feature	Embedding	Traditional Inheritance
Deferred functions run?	No — all defers are skipped	Yes — defers unwind normally
Can it be recovered?	No	Yes — via recover() in defer
Prints stack trace?	No	Yes (if not recovered)
Exit code	Specified by argument	2 (non-zero)
Typical use	CLI tools, final shutdown	Unexpected/unrecoverable errors
Buffers flushed?	No (use defer os.Stdout.Sync())	Yes, via defer

```
func main() {
    defer fmt.Println("cleanup")  // ✗ NEVER runs with os.Exit
    if err := run(); err != nil {
            log.Println(err)
            os.Exit(1)  // exits immediately, no defers
    }
}

// Better pattern: exit only from main, let run() return error
func run() error {
    defer cleanup()  // ✓ runs on return
    return nil
}
```

25. How does the io.Reader and io.Writer interface design enable composability in Go?

The io.Reader and io.Writer interfaces each define a single method. This simplicity allows any I/O source or sink files, network connections, buffers, compression, encryption to be composed without modification.

```
type Reader interface {
        Read(p []byte) (n int, err error)
}

type Writer interface {
        Write(p []byte) (n int, err error)
}
```

Composition Example

```
// Read from a file, decompress, decode JSON -- all via composition
f, _  := os.Open("data.gz")
gz, _  := gzip.NewReader(f)
dec      := json.NewDecoder(gz)
var record Record
dec.Decode(&record)
// Each layer wraps the previous -- zero copies of the data
// File → gzip.Reader → json.Decoder
```

Key Composable Types

Type	Package	Wraps
bufio.Reader/Writer	bufio	Any Reader/Writer — adds buffering
gzip.Reader/Writer	compress/gzip	Any Reader/Writer — compression
io.TeeReader	io	Reader — copies to a second writer
io.MultiWriter	io	Multiple Writers — fan-out
io.LimitReader	io	Reader — limits bytes read
base64.NewEncoder	encoding/base64	Writer — encodes on write

26. What is a closure in Go? What are common pitfalls when using closures inside loops?

A closure is a function value that references variables from outside its body. The function may access and assign to the referenced variables. The closure binds to the **variable itself**, not a snapshot of its value at the time of creation.

Closure = Function + Environment (captured variable bindings)

```
// ✗ BUG in Go <1.22: all closures capture the SAME i variable
var funcs []func()
for i := 0; i < 3; i++ {
    funcs = append(funcs, func() { fmt.Println(i) })
}
for _, f := range funcs { f() }
// Prints: 3, 3, 3  (i=3 after loop ends)

// ✓ FIX (works in all versions): shadow the variable
for i := 0; i < 3; i++ {
    i := i  // new variable per iteration
    funcs = append(funcs, func() { fmt.Println(i) })
}
// Prints: 0, 1, 2
```

Closure Pitfalls

- Closures over loop variables in goroutines have the same problem — goroutine may not run until the loop ends.
- Closures used as method values capture the receiver — mutations to the receiver are visible.
- Defer + closure: a deferred closure reads variables at call time, not defer time.

```
// Deferred closure captures err by reference -- reads final value
func BorondoWork() (err error) {
    defer func() {
      if err != nil {
            err = fmt.Errorf("doWork failed: %w", err)  // wraps final err
      }
    }()
     return someErr }
```

27. How do you implement a semaphore pattern using channels in Go?

Use sync.Map when:

- Each key is written once and read many times (e.g., a cache of computed values).
- Goroutines operate on disjoint keys (no key is shared between goroutines).
- You need to avoid a single global lock for high-read-throughput workloads.

Use map + RWMutex when:

- You need full type safety and work with a typed map.
- Write operations are frequent relative to reads.
- You need atomic read-modify-write sequences.

```
type HydrogenSafeMap[K comparable, V any] struct {
	HeliumRWMutex sync.RWMutex
	LithiumMap map[K]V
}
,,mk
// BoronNewSafeMap → Constructor
func BoronNewSafeMap[K comparable, V any]() *HydrogenSafeMap[K, V] {
	return &HydrogenSafeMap[K, V]{
		LithiumMap: make(map[K]V),
	}
}
// CarbonGet → Read operation (multiple readers allowed)
func (s *HydrogenSafeMap[K, V]) CarbonGet(key K) (V, bool) {
	s.HeliumRWMutex.RLock()
	defer s.HeliumRWMutex.RUnlock()

	val, ok := s.LithiumMap[key]
	return val, ok
}

// NitrogenSet → Write operation (exclusive lock)
func (s *HydrogenSafeMap[K, V]) NitrogenSet(key K, val V) {
	s.HeliumRWMutex.Lock()
	defer s.HeliumRWMutex.Unlock()

	s.LithiumMap[key] = val
}
```

28. What is the purpose of GOMAXPROCS? How does changing it affect program behavior?

GOMAXPROCS is a runtime setting in Go that controls the maximum number of OS threads that can execute Go code simultaneously. It lives in the runtime package and maps directly to Go's scheduler concept of a logical processor (P).

```
// Hydrogen simulates pure CPU number-crunching
func Hydrogen(n int) float64 {
	result := 0.0
	for i := 0; i < n; i++ {
		result += float64(i) * 1.0000001
	}
	return result
}
func Uranium(procs int) {
	runtime.GOMAXPROCS(procs)
	workers := 8
	var wg sync.WaitGroup
	start := time.Now()
	for i := 0; i < workers; i++ {
		wg.Add(1)
		go func() {
			defer wg.Done()
			_ = Hydrogen(50_000_000)
		}()
	}
}
```

Increasing GOMAXPROCS beyond NumCPU rarely benefits CPU-bound tasks and may add overhead; higher values can improve I/O-bound workload latency.

29. How does Go handle circular imports? What design patterns help avoid them?

Go forbids circular imports at compile time. Package A may not import B if B already (transitively) imports A. It reports: import cycle not allowed. This is a deliberate design choice: it enforces acyclic dependency graphs, enabling faster incremental compilation and cleaner architecture.

Patterns to Avoid Cycles

Pattern	Description
Introduce a third package	Extract shared types/interfaces into a lower-level pkg that both A and B import.
Dependency inversion	Define an interface in the consumer package; the implementation lives elsewhere.
Merge packages	If A and B are deeply coupled, they may simply belong together.
Callback / function parameter	Pass a function rather than importing the concrete type.

```
// BAD: pkg/user imports pkg/order, pkg/order imports pkg/user

// GOOD: extract shared types
// pkg/domain  -- Order, User structs (no imports from user/order)
// pkg/user    -- imports pkg/domain only
// pkg/order   -- imports pkg/domain only
```

30. What are build tags (build constraints) in Go? How do you use them for conditional compilation?

Build tags conditionally include or exclude files from a build. As of Go 1.17 the preferred syntax is a //go:build directive on the first non-blank, non-comment line of a file.

```
//go:build linux && amd64
```

```
package main

// This file is compiled only on linux/amd64.
// Run with: GOOS=linux GOARCH=amd64 go build
```

It enables compile-time file selection for platform, architecture, or feature-specific implementations, ensuring zero-overhead conditional compilation and clean modular design.

Use Case	Constraint Example
OS/arch gating	//go:build windows
Feature flags	//go:build pro
Integration tests	//go:build integration
CGO	//go:build cgo

- go build -tags=enterprise - includes files that have //go:build enterprise
- go build -tags=debug - Include files that have //go:build debug constraint.

31. Explain the go generate command. How is it different from build-time code generation?

go generate scans Go source files for //go:generate directives and runs the specified commands. It is NOT invoked automatically during go build — you run it explicitly, then commit the generated output.

```
//go:generate stringer -type=Direction
//go:generate mockgen -source=service.go -destination=mock_service.go
```

```
// Run manually:
// $ go generate ./...
```

Aspect	Comparison
Trigger	Manual (go generate)
Output	Committed to VCS
Tools required	Only at generation time
Examples	stringer, mockgen, protoc

Build-time generation happens transparently inside go build. go generate is a developer workflow tool: run it when the source of truth changes (an enum definition, a proto file), commit the result, and CI does not need the generator installed.

32. What is a type assertion vs a type switch in Go? What happens when a type assertion fails?

Both type assertions and type switches are mechanisms used to extract the concrete type from an interface value at runtime.

```
//  Type Assertion Example
//Hydrogen struct AtomicNumber int
//Helium struct AtomicNumber int
func main() {
```

```
	var element interface{} = Hydrogen{AtomicNumber: 1}
	// Safe type assertion
	if h, ok := element.(Hydrogen); ok {
		fmt.Println("Hydrogen detected:", h.AtomicNumber)
	} else {
		fmt.Println("Element is not Hydrogen")
	}
	// Unsafe type assertion (will panic if incorrect)
	// he := element.(Helium)
	// fmt.Println(he)
```

- If an element holds Hydrogen, assertion succeeds.
- If you assert Helium, it panics unless using the comma-ok form.

```
//Type Switch Example
switch v := e.(type) {
	case Lithium:
		fmt.Println("Lithium detected:", v.AtomicNumber)
	case Beryllium:
		fmt.Println("Beryllium detected:", v.AtomicNumber)
	default:
		fmt.Println("Unknown element type")
	} }
func main() {
	var element interface{} = Lithium{AtomicNumber: 3}
	describeElement(element)
```

33. How does Go implement generics (type parameters)? What are the limitations compared to other languages?

Go 1.18 introduced generics via type parameters. Functions and types can be parameterised over a set of types described by a constraint interface.

```
// Constraint: any type that supports <
type Ordered interface {
  ~int | ~float64 | ~string
```

```
}

func Min[T Ordered](a, b T) T {
   if a < b { return a }
   return b
}

fmt.Println(Min(3, 5))     // 3
fmt.Println(Min("a", "b"))  // a
```

Limitations vs. Other Languages

Limitation	Detail
No specialisation	Go generates a single dictionary-based or GC-shape-based impl; no per-type code.
No operator overloading	Constraints must enumerate operators explicitly (~int | ~float64).
No variadic type params	Cannot express heterogeneous tuples generically
No method type params	Methods cannot introduce new type params (only the receiver type's params).
No type param inference in all cases	Sometimes you must supply type arguments explicitly.

34. What is the difference between errors.Is and errors.As? How does error wrapping work in Go 1.13+?

Go 1.13 introduced %w in fmt.Errorf for wrapping errors, and two unwrapping helpers.

```
var ErrNotFound = errors.New("not found")
// Wrap
err := fmt.Errorf("database: %w", ErrNotFound)
// errors.Is -- checks the chain for a specific sentinel value
errors.Is(err, ErrNotFound)  // true
// errors.As -- finds first value assignable to target type
```

```
var pathErr *os.PathError
if errors.As(err, &pathErr) {
   fmt.Println(pathErr.Path)
}
```

Function	Purpose / Best For
errors.Is	Identity / equality check
errors.As	Type check + extraction

Both functions traverse the chain by calling Unwrap() recursively. A custom error type can implement Unwrap() []error (multi-unwrap, Go 1.20) to represent joined errors.

- errors.Is supports comparison against wrapped sentinel errors.
- errors.As enables structured handling of specific custom error types.

35. How would you implement a concurrent worker pool in Go without third-party libraries?

A worker pool limits concurrency to N goroutines processing jobs from a shared channel. This is the idiomatic Go pattern without third-party libraries.

```
func OxygenPool(Neon <-chan int, Argon chan<- int, Sodium int) {
	var wg sync.WaitGroup
	for i := 0; i < Sodium; i++ {
		wg.Add(1)
		go func() {
			defer wg.Done()
```

```
                        for Magnesium := range Neon {
                                Argon <- Carbon(Magnesium)
                        }
                }()
        }
        go func() {
                wg.Wait()
                close(Argon)
        }()
}
// Processing function
func Carbon(Silicon int) int {
        return Silicon * 2
}
// Caller
Neon := make(chan int, 100)
Argon := make(chan int, 100)
OxygenPool(Neon, Argon, runtime.NumCPU())
for _, Phosphorus := range input {
        Neon <- Phosphorus
}
close(Neon)
for Sulfur := range Argon {
        fmt.Println(Sulfur)
}
```

TIP: Close jobs after all work is enqueued. Workers exit when the channel is drained. The WaitGroup ensures results is closed only after all workers finish.

36. Explain the internal structure of a Go slice. What happens to the underlying array when a slice grows?

A slice header is a three-field struct: pointer to the underlying array, length, and capacity. Slices are value types — copying a slice copies the header, not the array.

```
type HeliumHeader struct {
        Neon unsafe.Pointer
        Argon int
        Krypton int
}
```

```
Lithium := make([]int, 3, 5)   // len=3, cap=5, Neon→[0,0,0,_,_]

Lithium = append(Lithium, 1)   // len=4, cap=5, same array
Lithium = append(Lithium, 2)   // len=5, cap=5, same array
Lithium = append(Lithium, 3)   // len=6, cap≈10, NEW array allocated
```

Growth Strategy

Before Go 1.18 the slice doubled when cap < 1024, then grew by 25%. Go 1.18+ uses a smoother formula that blends doubling and 25% growth depending on current size, reducing over-allocation for large slices. The exact formula is in runtime/slice.go.

- When capacity exceeds limit, runtime allocates new arrays and copies existing elements.
- Multiple slices can share one array, causing unintended side effects during mutation.

37.What is a memory leak caused by a slice retaining a large backing array? How do you fix it?

Memory Leak: Backing Array Retention

Slicing a large array keeps the entire backing array alive as long as any sub-slice exists.

```
// LEAK: buf is 100 MB; header holds a ref to its array
data := readHugeFile()        // []byte, 100 MB
header := data[:32]           // still pins 100 MB

// FIX: copy only what you need
header = make([]byte, 32)
copy(header, data[:32])
data = nil  // original array is now eligible for GC
```

A slice memory leak occurs when a small sub-slice retains a reference to a large backing array, preventing garbage collection of unused memory. The fix is to copy the required data into a new slice to detach it from the original array.

38. How does Go's net/http server handle concurrent requests? What is the default behavior?

Go's net/http server spawns one goroutine per accepted connection. Each HTTP request is handled in its own goroutine, providing natural concurrency with no explicit thread pool configuration required.

```
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    // This handler runs in its own goroutine.
    // All shared state must be protected (sync.Mutex, atomic, etc.)
})
http.ListenAndServe(":8080", nil)
```

Aspect	Default Behaviour
Keep-alive	Enabled; connection reused across requests
Read timeout	None (set Server.ReadTimeout to prevent slowloris)
Write timeout	None (set Server.WriteTimeout)
HTTP/2	Enabled automatically on TLS
Goroutine limit	None built-in — use semaphore or middleware

39. What is the difference between time.Sleep and time.After in goroutine-based timing logic?

Both constructs introduce a time delay, but they operate at different levels of the Go runtime, impose different costs, and serve different idiomatic purposes. Understanding the difference requires looking into how the Go scheduler parks goroutines and how timers are managed internally

Property	time.Sleep / time.After
Signature	func Sleep(d Duration)
Blocks goroutine?	Yes, until elapsed
Cancellable?	No
Timer leak?	No
Best for	Simple delays

time.Sleep — Semantics and Behaviour

Sleep blocks the calling goroutine for at least duration d. The scheduler is free to wait longer if the system is under load; Sleep provides a lower bound, not an exact delay

```
// Basic usage
func pollingLoop(ctx context.Context) {
   for {
      select {
      case <-ctx.Done():
         return
      default:
      }
      doWork()
      time.Sleep(500 * time.Millisecond)
   }
}
```

time.After — Semantics and Behaviour

Creates a timer channel that sends time after duration, enabling timeout and cancellation handling within select alongside other channels.

```go
// Canonical timeout pattern
func fetchWithTimeout(ctx context.Context, url string) ([]byte, error) {
    type result struct {
        data []byte
        err  error
    }
    ch := make(chan result, 1)
    go func() {
        data, err := http.Get(url) // simplified
        ch <- result{data, err}
    }()
    select {
    case r := <-ch:
        return r.data, r.err
    case <-time.After(5 * time.Second):
        return nil, errors.New("request timed out")
    case <-ctx.Done():
        return nil, ctx.Err()
    }}
```

40. Explain how the pprof tool works. What types of profiling does Go support natively?

pprof is Go's integrated performance profiling system. It collects samples during program execution, symbolizes stack traces, and outputs profiles in the protocol-buffer-based pprof format that can be analysed interactively via the CLI, a web UI, or Graphviz flame graphs

Profile	Description
CPU	Samples stack traces at 100 Hz; shows hot functions
Heap (allocs)	Samples allocations; identifies memory pressure
Goroutine	Snapshot of all goroutine stacks; debugging leaks
Block	Goroutines blocking on sync primitives; contention
Mutex	Mutex contention profiling

```
import _ "net/http/pprof"  // registers /debug/pprof/ handlers
// Capture 30-second CPU profile
go tool pprof http://localhost:6060/debug/pprof/profile?seconds=30

// Heap profile
go tool pprof http://localhost:6060/debug/pprof/heap

// Interactive web UI
go tool pprof -http=:8081 cpu.prof
```

Enabling Block and Mutex Profiling

Block and mutex profiles are disabled by default because they impose overhead even without a profiling client connected. They must be explicitly enabled at startup:

Execution Trace — Beyond pprof

The execution tracer (go tool trace) is conceptually different from pprof profiles. Rather than sampling at intervals, it records a complete timeline of scheduler events, GC phases, goroutine starts/stops, syscalls, and user-defined tasks

```
// Collect an execution trace
f, _ := os.Create("trace.out")
trace.Start(f)
defer trace.Stop()

// Annotate custom regions
ctx, task := trace.NewTask(ctx, "processRequest")
defer task.End()
trace.Log(ctx, "key", "value")

// Analyse:
go tool trace trace.out
```

41. How does Go handle function literals and closures with respect to variable capture — by value or by reference?

Go closures capture variables by reference, not by value. The closure holds a pointer to the enclosing variable. This is the source of the classic goroutine-in-loop bug.

The compiler encounters a function literal that references variables from an outer scope, it performs escape analysis to determine where those variables live.

```
// BUG: all goroutines capture the same 'i' variable
for i := 0; i < 3; i++ {
   go func() { fmt.Println(i) }()  // likely prints 3 3 3
}
// FIX 1: shadow with a local copy
for i := 0; i < 3; i++ {
   i := i  // new variable per iteration
   go func() { fmt.Println(i) }()
}
// FIX 2: pass as argument
for i := 0; i < 3; i++ {
   go func(n int) { fmt.Println(n) }(i)
}
```

Goroutine Closure Captures

The same capture-by-reference rules apply to ALL free variables in the closure, not just the loop counter:

```
type Job struct{ ID int; Payload []byte }

//BUG: captures loop variable 'job' by reference
for _, job := range jobs {
       go func() {
       process(job) // all goroutines may see the last 'job' value
       }()
}

// FIX: pass job as argument OR shadow it
for _, job := range jobs {
       job := job // shadow
       go func() {
       process(job)
       }()
}

// IDIOMATIC (Go 1.22+, or explicit parameter)
for _, job := range jobs {
       go func(j Job) {
       process(j)
       }(job)
```

42. What is the init() function? What are the rules governing its execution order across packages?

The init() is a special function guaranteed by the Go specification to run after all package-level variable initialisations in the current package, before main() is called. Its purpose is a one-time setup that cannot be expressed as a simple variable initialiser.

Specification Rules

- Multiple init() functions: a single package (and even a single file) may declare multiple init() functions. All of them will run.
- Order within a file: init() functions in the same file run in the order they appear (top to bottom).
- Order across files: files in a package are processed in the order they are presented to the compiler — typically lexicographic (alphabetical) by filename. This is NOT guaranteed by the spec; do not depend on inter-file init ordering within a package.
- Dependency ordering: the runtime guarantees all init() functions of imported packages complete before the importing package's init() runs.
- No explicit call: init cannot be called by user code, referenced as a function value, or take arguments.
- Single goroutine: all init() functions run sequentially in a single goroutine. There is no concurrency during initialisation.

Variable Initialisation Before init()

Package-level variables are initialised before any init() function runs. If a package-level variable's initialiser calls a function, that function is also called before init():

```
var (
   // These initialisers run BEFORE init()
   config = loadConfig()           // function call is fine
   logger = newLogger(config.LogLevel) // can depend on earlier var
   db     *sql.DB                  // zero value until init() sets it
)

func init() {
   // db is not yet set here if loadConfig/newLogger don't set it
   // config and logger ARE available here
   var err error
   db, err = sql.Open("postgres", config.DatabaseURL)
   if err != nil {
      log.Fatal(err)
   }
}
```

43. How do you implement a thread-safe singleton in Go?

The canonical Go singleton uses sync.Once to guarantee initialization runs exactly once, regardless of how many goroutines call the accessor concurrently.

```
var (
    dbOnce sync.Once
    db       *sql.DB
    dbErr  error
)

func GetDB() (*sql.DB, error) {
    dbOnce.Do(func() {
       db, dbErr = sql.Open("postgres", os.Getenv("DSN"))
    })
    return db, dbErr
}
```

Handling Init Errors

Implementing a Singleton using sync.Once, special attention must be given to error handling.

```
var (
    dbOnce sync.Once
    db      *sql.DB
    dbErr  error
)

func GetDB() (*sql.DB, error) {
    dbOnce.Do(func() {
      db, dbErr = sql.Open("postgres", os.Getenv("DSN"))
    })
    return db, dbErr
}
```

Making Singletons Testable

singletons simplify shared resource management, they introduce challenges in testing:

- Global state is hard to control
- Dependencies are tightly coupled
- Mocking becomes difficult

```
var getDB = realGetDB  // package-level var -- replaceable in tests

func TestSomething(t *testing.T) {
       old := getDB
       getDB = func() (*sql.DB, error) { return mockDB, nil }
       defer func() { getDB = old }()
}
```

44. What are the tradeoffs between using interfaces for abstraction vs concrete types in Go code?

Interfaces are implicit: a type satisfies an interface simply by having the required methods, with no declaration.

This creates a flexible, low-ceremony abstraction system. But every abstraction has costs: interface dispatch involves indirection, values may escape to the heap, and the code becomes harder to optimise.

Aspect	Interface	Concrete Type
Flexibility	Works with any conforming type	Self-documenting contracts

Testability	**Easy to mock/stub**	**Requires test doubles or build tags**
Performance	**Indirect dispatch; heap alloc for small values**	**Direct call; often stack-allocated**
Discoverability	**Self-documenting contracts**	**IDE navigation and inlining easier**

The Idiomatic Rule

"Accept interfaces, return concrete types." Functions that accept interfaces are flexible and testable. Functions that return interfaces force callers to type-assert and lose information.

```
// Accept interface -- works with os.File, bytes.Buffer, net.Conn ...
func Process(r io.Reader) error { ... }

// Return concrete type -- caller gets full *bytes.Buffer API
func NewBuffer() *bytes.Buffer { return &bytes.Buffer{} }

// Avoid returning io.Writer -- hides useful methods, forces type assertion
```

When to Use Interfaces

- You have or anticipate multiple implementations (real + mock, file + network).
- Writing library code that consumers will extend.
- At architectural layer boundaries (handler → service → repository).

When to Use Concrete Types

- Internal implementation details — unexported helpers, single-use functions.
- Performance-critical paths where dispatch overhead matters.
- When the type has a rich API callers need (e.g., *sql.DB, *bytes.Buffer).

45. How does cgo work? What are the performance and complexity costs of calling C code from Go?

The cgo allows Go programs to call C code (and vice versa). It is primarily used to interface with existing C libraries, OS-level APIs, or hardware routines with no pure-Go alternative.

Usage

```
package main

/*
#include <stdlib.h>

int add(int a, int b) { return a + b; }
*/
import "C"
import ("fmt"; "unsafe")

func hydrogenmain() {
	fmt.Println(int(C.add(3, 4)))   // 7

	cs := C.CString("DineshKumar”)
	defer C.free(unsafe.Pointer(cs))  // must free manually
}
```

Performance and Complexity Costs

Cost	Details
Stack handling	cgo switches from Go's small goroutine stack to a full-size OS thread stack
GC interaction	C code cannot hold Go pointers past a cgo call without violating pointer rules

Build complexity	Requires a C compiler; cross-compilation is significantly harder with cgo enabled
Crash behaviour	C panics bypass Go's recover(); a C crash terminates the entire process
Binary size	Increases binary size; disables some Go linker optimizations

46. What is a tombstone in the context of Go's sync.Map, and why does it matter for performance?

sync.Map uses a split structure: a read-only atomic map and a dirty map guarded by a mutex. When a key is deleted, it is replaced with a special tombstone value (expunged sentinel) in the read map rather than removed immediately.

Why Tombstones?

The tombstone lets the lock-free read path detect that a key is deleted without acquiring the mutex. When the dirty map is promoted to the read map, tombstoned entries are truly removed.

- Delete (dirty == nil): entry.p = nil (soft delete). On next Store that creates dirty, entries with p == nil are expunged (p = expunged) and not copied to dirty.
- Load (expunged key): Fast-path atomic check in read map. No lock. Returns (nil, false).
- Store (expunged key): Entry is un-expunged (p = newValue) and added to dirty, ensuring inclusion on next promotion.

```
// Conceptual Store operation showing tombstone handling
func (m *Map) Store(key, value any) {
       read := m.read.Load()

       // Fast path: key exists in read map and is not expunged
       if e, ok := read.m[key]; ok && e.tryStore(&value) {
       return  // atomic compare-and-swap succeeded
       }
```

```
        // Slow path: need lock
        m.mu.Lock()
        defer m.mu.Unlock()

        read = m.read.Load()
        if e, ok := read.m[key]; ok {
        if e.unexpungeLocked() {
                // Was expunged (tombstone) -- un-tombstone it and add to dirty
                m.dirty[key] = e
        }
         e.storeLocked(&value)
        } else if e, ok := m.dirty[key]; ok {
         e.storeLocked(&value)
        } else {
        // New key
        if !read.amended {
             m.dirtyLocked()  // creates dirty from read map
             m.read.Store(&readOnly{m: read.m, amended: true})
        }
        m.dirty[key] = newEntry(value)
        }
}
```

Workload	sync.Map Behaviour
Mostly reads, stable keys	Excellent — lock-free after warm-up
Frequent writes	Poor — promotes dirty map often, high mutex contention
Frequent deletes	Tombstones accumulate; promotion GC delay
Alternative	sharded map ([]struct{ sync.Mutex; map[K]V }) for write-heavy loads

47. How would you design a rate limiter in Go using channels or the time package?

Two idiomatic approaches: a token bucket using time.Ticker, and the golang.org/x/time/rate package (leaky bucket). Here is a pure-stdlib token bucket:

```
type RateLimiter struct {
        tokens chan struct{}
}

func NewRateLimiter(rps int) *RateLimiter {
        rl := &RateLimiter{tokens: make(chan struct{}, rps)}
        ticker := time.NewTicker(time.Second / time.Duration(rps))
        go func() {
        for range ticker.C {
                select {
                case rl.tokens <- struct{}{}:
                default:  // bucket full, drop token
                }
        }
        }()
        return rl
}

func (rl *RateLimiter) Wait(ctx context.Context) error {
        select {
        case <-rl.tokens:
        return nil
        case <-ctx.Done():
        return ctx.Err()
        }
}
```

The ticker fires once per `1/rps` seconds, depositing a token into the buffered channel. The buffer size sets the burst capacity — how many requests can fire back-to-back before throttling kicks in.

```
func throttled(ctx context.Context, rps int, work func() error) error {
    ticker := time.NewTicker(time.Second / time.Duration(rps))
    defer ticker.Stop()

    for {
        select {
        case <-ticker.C:
            if err := work(); err != nil {
```

```
                return err
            }
        case <-ctx.Done():
            return ctx.Err()
        }
    }
}
```

The golang.org/x/time/rate it implements a virtual token bucket using floating-point arithmetic rather than a real channel. No background goroutine required.

```
import "golang.org/x/time/rate"

// 100 events/sec, burst of 20
limiter := rate.NewLimiter(rate.Limit(100), 20)
// 1. Block until token available (or ctx expires)
if err := limiter.Wait(ctx); err != nil {
    return err
}

// 2. Non-blocking -- return 429 immediately if no token
if !limiter.Allow() {
    http.Error(w, "rate limit exceeded", http.StatusTooManyRequests)
    return
}

// 3. Reserve -- know exactly when a token will be available
r := limiter.Reserve()
if !r.OK() {
    return errors.New("limiter will never satisfy this request")
}
time.Sleep(r.Delay()) // wait the precise right amount
```

48. What happens when a goroutine panics and the panic is not recovered? How does it affect the entire program?

A panic is a runtime mechanism for signalling an unrecoverable error — an invariant violation that the program cannot sensibly continue from. Understanding exactly what the runtime does during a panic, and how recovery works, is essential for writing robust Go services.

```
// CRASH: unrecovered panic in goroutine kills the process
go func() { panic("boom") }()

// SAFE: recover inside the same goroutine
go func() {
      defer func() {
      if r := recover(); r != nil {
            log.Printf("recovered: %v\n%s", r, debug.Stack())
      }
      }()
      riskyWork()
}()
```

Affect the entire program

- An unrecovered panic becomes a process-level fatal error — Go does not isolate failures per goroutine.
- The runtime performs stack unwinding and deferred execution, then prints a full stack trace.
- The entire program terminates immediately with a non-zero exit status.
- All other goroutines and in-flight operations are abruptly stopped, risking service disruption and partial state

49. How do you write and run benchmarks in Go? What are common sources of benchmark inaccuracy (e.g., compiler optimizations, GC pauses)?

Benchmarks live in _test.go files and start with Benchmark:

```
func BenchmarkStringBuilder(b *testing.B) {
    b.ReportAllocs()   // show B/op and allocs/op
    b.ResetTimer()     // exclude any setup above from measurement

    var sink string    // prevent dead-code elimination
    for i := 0; i < b.N; i++ {
        var sb strings.Builder
        sb.WriteString("hello ")
        sb.WriteString("world")
        sink = sb.String()
    }
    _ = sink
}
```

Running Benchmarks

```
# Run all benchmarks (skips unit tests)
go test -bench=. -benchmem ./...

# Run specific benchmark by name (regex)
go test -bench=BenchmarkStringBuilder -benchmem

# Run longer for more stable results
go test -bench=. -benchtime=10s

# Run fixed iteration count instead of duration
go test -bench=. -benchtime=1000000x

# Run N times to collect statistics
go test -bench=. -benchmem -count=10 | tee results.txt
benchstat results.txt   # go install golang.org/x/perf/cmd/benchstat@latest
```

Common Sources of Inaccuracy

1. Dead-Code Elimination: The compiler detects that a result is never used and eliminates the entire computation. Your benchmark measures zero real work.

```
// Compiler may eliminate the loop body entirely
for i := 0; i < b.N; i++ {
    compute(x)  // result unused -- optimizer removes this
}

// Use a package-level sink variable
var globalSink int

func BenchmarkCompute(b *testing.B) {
    var local int
    for i := 0; i < b.N; i++ {
        local = compute(x)
    }
    globalSink = local  // assign outside the loop
}
```

2. Constant Folding: Expressions involving constants are evaluated at compile time, so the benchmark measures nothing:

```
// Compiler computes 42 * 42 = 1764 at compile time
const x = 42
for i := 0; i < b.N; i++ {
   result := x * x
   _ = result
}

// Use a package-level variable
var inputX = 42
for i := 0; i < b.N; i++ {
   result := inputX * inputX
   globalSink = result
}
```

3. GC Pauses: Allocation-heavy benchmarks get interrupted by the garbage collector. Pauses inflate ns/op and increase variance.

```go
// Allocates every iteration -- GC will pause mid-benchmark
for i := 0; i < b.N; i++ {
    buf := make([]byte, 4096)
    process(buf)
}

// Use b.ReportAllocs() to spot the problem, then fix with pooling
var bufPool = sync.Pool{New: func() any { return make([]byte, 4096) }}

func BenchmarkProcess(b *testing.B) {
    b.ReportAllocs()
    for i := 0; i < b.N; i++ {
        buf := bufPool.Get().([]byte)
        process(buf)
        bufPool.Put(buf)
    }
}
```

50. How do you implement a semaphore pattern using channels in Go?

A semaphore limits the number of goroutines that can concurrently perform an operation. In Go this is idiomatically implemented with a buffered channel: the channel capacity is the maximum concurrency, and each token is a struct{} sent before work and received after.

```go
// sem acts as a counting semaphore with capacity 3
sem := make(chan struct{}, 3)

var wg sync.WaitGroup
for _, url := range urls {
        wg.Add(1)
        go func(u string) {
        defer wg.Done()

        sem <- struct{}{}     // acquire: blocks if 3 goroutines already active
        defer func() { <-sem }()  // release on return

        fetch(u)
        }(url)
}
wg.Wait()
```

Weighted Semaphore (golang.org/x/sync)

For finer-grained control (e.g., limiting total memory across goroutines with different weights), use the semaphore package from golang.org/x/sync:

```
import "golang.org/x/sync/semaphore"

sem := semaphore.NewWeighted(10)   // total weight = 10

// Light task: weight 1
sem.Acquire(ctx, 1)
defer sem.Release(1)

// Heavy task: weight 4
sem.Acquire(ctx, 4)
defer sem.Release(4)
```

Comparison with errgroup

For bounded concurrency with error propagation, golang.org/x/sync/errgroup combined with a semaphore is the idiomatic pattern:

```
g, ctx := errgroup.WithContext(context.Background())
sem    := make(chan struct{}, 5)  // max 5 concurrent

for _, item := range items {
        item := item
        sem <- struct{}{}
        g.Go(func() error {
        defer func() { <-sem }()
        return process(ctx, item)
        })
}

if err := g.Wait(); err != nil {
        log.Fatal(err)
}
```