Gorgo Go for Java Programmers

June 11, 2026

Contents

Chapter 11

Synchronization

Chapter 10 introduced goroutines and channels — Go's preferred way to share work. But channels are not always the right tool: sometimes you need to protect a shared data structure, initialize something exactly once, or coordinate goroutines that don't exchange messages. The `sync` package and `sync/atomic` cover those cases, offering primitives that Java programmers will recognize under new names.

The Go Memory Model

Before reaching for a mutex, you need to understand what the Go memory model guarantees. The model defines **happens-before** relationships: when a write in one goroutine is guaranteed to be visible to a read in another.

Without synchronization, goroutines are allowed to observe memory in any order. The compiler and CPU can reorder instructions as long as the reordering is invisible within a single goroutine — but that reordering **is** visible across goroutines.

The key rules:

- A send on a channel happens-before the corresponding receive from that channel.
- A `sync.Mutex` `Unlock` happens-before any subsequent `Lock` that succeeds.
- `sync.Once.Do` completion happens-before any call to `Do` returns.
- Program initialization (all `init` functions) happens-before `main`.

Wut: Java programmers expect that writing a variable in one thread and reading it in another is safe as long as no two threads write at the same time. In Go (and in Java under the JMM) that is **not** safe unless there is a synchronization action between the write and the read. Without one, the reading goroutine may see a stale or partially written value.

The practical rule: any time two goroutines access the same memory and at least one of them is writing, you must use a channel, a mutex, or an atomic to establish a happens-before relationship. The race detector (`go test -race`) enforces this mechanically — see the end of this chapter.

`sync.Mutex` and `sync.RWMutex`

`sync.Mutex` is Go's equivalent of Java's `synchronized` block. It provides exclusive access to a critical section.

```
import "sync"

type Playlist struct {
```

```
    mu     sync.Mutex // protects tracks
    tracks []string
}

func (p *Playlist) Add(track string) {
    p.mu.Lock()
    defer p.mu.Unlock()
    p.tracks = append(p.tracks, track)
}

func (p *Playlist) Tracks() []string {
    p.mu.Lock()
    defer p.mu.Unlock()
    result := make([]string, len(p.tracks))
    copy(result, p.tracks)
    return result
}
```

`Lock` blocks until the mutex is available. `Unlock` releases it. Using `defer p.mu.Unlock()` immediately after `Lock` ensures the mutex is always released, even if the function panics.

Tip: Always use `defer mu.Unlock()` on the very next line after `mu.Lock()`. This eliminates the risk of forgetting to unlock on every return path.

Trap: A `sync.Mutex` must not be copied after first use. If you embed one in a struct, always pass the struct by pointer (`*Playlist`), never by value. Copying a locked mutex is undefined behavior. [*pointer-receiver-for-mutex*]

Comparison with Java `synchronized`

Java's `synchronized` keyword takes an **object monitor** as its lock:

```
synchronized (this) {
    tracks.add(track);
}
```

Go has no per-object monitor. You declare an explicit `sync.Mutex` field and lock it by name. This is more verbose but also more precise: you can have multiple independent mutexes protecting different fields in the same struct.

`sync.RWMutex`

`sync.RWMutex` is an independent type (not a subtype of `sync.Mutex`) that offers a reader/writer lock with separate read and write modes. Many goroutines can hold the read lock at once, **or** a single goroutine can hold the write lock exclusively — never both. Readers do not block each other; a writer blocks everyone. Use it when reads far outnumber writes.

```
type Catalog struct {
    mu     sync.RWMutex
    songs  map[string]string // title -> artist
}

func (c *Catalog) Lookup(title string) (string, bool) {
    c.mu.RLock()         // multiple goroutines can hold RLock at once
```

```
    defer c.mu.RUnlock()
    artist, ok := c.songs[title]
    return artist, ok
}

func (c *Catalog) Add(title, artist string) {
    c.mu.Lock()          // exclusive write lock
    defer c.mu.Unlock()
    c.songs[title] = artist
}
```

`RLock` and `RUnlock` are the read-side pair. `Lock` and `Unlock` are the write-side pair, identical to `sync.Mutex`.

Tip: Only reach for `RWMutex` when you have measured a contention problem. A plain `Mutex` is faster for workloads with balanced reads and writes because `RWMutex` has higher overhead to track readers.

Tip: Yes, `sync.Map` exists, and yes, it is the answer to the "where did `ConcurrentHashMap` go?" question. No, you usually do not want it. A mutex-guarded map like `Catalog` above is the idiomatic default: type-safe, simple, and easy to reason about. `sync.Map` trades type safety (`any` keys and values) for better performance in one niche — append-mostly caches where many goroutines read, store, and overwrite **disjoint** sets of keys.

```
func (m *Map) Load(key any) (value any, ok bool)                  // lookup
func (m *Map) Store(key, value any)                               // upsert
func (m *Map) LoadOrStore(key, value any) (actual any, loaded bool) // get/add
func (m *Map) Range(f func(key, value any) bool)                  // iterate
```

`LoadOrStore` returns the existing value (`loaded` is `true`) when the key is already present, and `Range` stops as soon as `f` returns `false`.

sync.WaitGroup

`sync.WaitGroup` lets a goroutine wait for a collection of other goroutines to finish. Java programmers typically use `CountDownLatch` or `CompletableFuture.allOf` for the same pattern; `WaitGroup` is simpler.

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    songs := []string{"Escape", "$100 Bills", "Legend"} // Jaroslav Beck
    var wg sync.WaitGroup

    for _, song := range songs {
        wg.Add(1)
        go func(s string) {
            defer wg.Done()
            fmt.Printf("playing: %s\n", s)
        }(song)
    }
```

```
    wg.Wait() // blocks until all three goroutines call Done
    fmt.Println("all songs finished")
}
```

The three operations are:

```
wg.Add(n)  // increment the counter by n --- call before launching goroutines
wg.Done()  // decrement the counter by 1 --- call when a goroutine finishes
wg.Wait()  // block until the counter reaches zero
```

Trap: Call `wg.Add(n)` **before** launching the goroutines, not inside them. If the goroutine calls `Add` and the main goroutine calls `Wait` before the goroutine starts, `Wait` will return immediately with a zero counter.

Trap: Pass the `WaitGroup` by pointer or use a closure that captures it. Copying a `WaitGroup` after first use is a bug.

`WaitGroup.Go` (Go 1.25+)

The `Add(1)` + `go func() { defer wg.Done() ... }()` dance is so common that Go 1.25 added a helper that bundles all three steps:

```
func (wg *WaitGroup) Go(f func()) // Add(1), run f in a new goroutine, Done when f returns
```

`Go` increments the counter, launches `f` in a new goroutine, and calls `Done` automatically when `f` returns — so you cannot forget the `Done`, and there is no chance of misplacing the `Add` (the Trap above simply cannot happen). The earlier example shrinks to this:

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    songs := []string{"Escape", "$100 Bills", "Legend"} // Jaroslav Beck
    var wg sync.WaitGroup

    for _, song := range songs {
        wg.Go(func() {
            fmt.Printf("playing: %s\n", song)
        })
    }

    wg.Wait() // blocks until all goroutines finish
    fmt.Println("all songs finished")
}
```

Note that `song` is captured directly: since Go 1.22 each loop iteration gets its own copy, so you no longer need the `func(s string){...}(song)` trick to avoid sharing one variable across goroutines.

Tip: Prefer `wg.Go(f)` over the manual `Add/go/defer Done` pattern in new code (Go 1.25+). It is shorter and structurally rules out the misplaced-`Add` and forgotten-`Done` bugs.

Fan-out / Fan-in Without Channels

`WaitGroup` is the idiomatic way to fire off N goroutines and wait for all of them without the ceremony of a results channel. If you need return values, combine `WaitGroup` with a pre-allocated slice (one slot per goroutine, no contention) or use `errgroup` from `golang.org/x/sync` (covered in Chapter 12).

`sync.Once`

`sync.Once` runs a function exactly once, no matter how many goroutines call it concurrently. It is the safe, idiomatic replacement for the double-checked locking pattern that Java programmers used before `volatile` was fixed in Java 5.

```
package main

import (
    "fmt"
    "sync"
)

var (
    once     sync.Once
    catalog  map[string]string
)

func loadCatalog() {
    once.Do(func() {
        // expensive initialization runs exactly once
        catalog = map[string]string{
            "Escape":               "Jaroslav Beck",
            "J'ai pas vingt ans !": "Alizée",
            "J'en ai marre !":      "Alizée",
        }
        fmt.Println("catalog loaded")
    })
}

func main() {
    var wg sync.WaitGroup
    for i := 0; i < 5; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            loadCatalog()
        }()
    }
    wg.Wait()
    fmt.Println(catalog["Escape"]) // Jaroslav Beck
}
```

Output:

```
catalog loaded
Jaroslav Beck
```

The message "catalog loaded" appears exactly once even though five goroutines called `loadCatalog` concurrently. Subsequent calls to `once.Do` return immediately.

Wut: `sync.Once` caches the **first** call's completion, not its result. If the function passed to `Do` panics, the once is still considered done — subsequent callers see the (partial) side effects and `Do` never runs again. Panic inside `Do` is almost always a bug.

The Java equivalent that `sync.Once` replaces:

```java
// Java double-checked locking (error-prone before Java 5)
private volatile Map<String,String> catalog;

public Map<String,String> getCatalog() {
    if (catalog == null) {
        synchronized (this) {
            if (catalog == null) {
                catalog = loadExpensive();
            }
        }
    }
    return catalog;
}
```

Go's `sync.Once` does this correctly with zero boilerplate.

Since Go 1.21 you often do not even need the `sync.Once` variable: the `sync` package wraps the whole pattern in `OnceFunc`, `OnceValue`, and `OnceValues`.

```
func OnceFunc(f func()) func()                                  // runs f exactly once
func OnceValue[T any](f func() T) func() T                      // runs once, caches value
func OnceValues[T1, T2 any](f func() (T1, T2)) func() (T1, T2) // two values, read: value+error
```

The lazy-value idiom becomes a one-liner:

```
var getCatalog = sync.OnceValue(loadExpensive) // loadExpensive runs on the first call
```

Every call to `getCatalog()` after the first returns the cached map — the entire Java method above, in one line, with no `volatile` archaeology.

sync.Cond

`sync.Cond` is a condition variable — a way for goroutines to wait for a predicate to become true and for other goroutines to signal when state changes. Java's equivalent is `Object.wait()` / `Object.notify()` / `Object.notifyAll()`, or the more modern `java.util.concurrent.locks.Condition`. In Go, channels are usually the preferred tool for this kind of coordination; reach for `sync.Cond` mainly when you need to broadcast a wakeup to many waiters at once (something a single channel send cannot do).

A `sync.Cond` is always associated with a `sync.Locker` (usually a `*sync.Mutex`):

```
cond := sync.NewCond(&mu) // create a Cond associated with mu
```

The three operations are:

```
cond.Wait()      // atomically unlock mu and suspend; re-lock mu on wake
cond.Signal()    // wake one waiting goroutine
cond.Broadcast() // wake all waiting goroutines
```

Here is a producer/consumer example with a bounded queue:

```
package main

import (
    "fmt"
    "sync"
)

type Queue struct {
    mu    sync.Mutex
    cond  *sync.Cond
    items []string
    cap   int
}

func NewQueue(cap int) *Queue {
    q := &Queue{cap: cap}
    q.cond = sync.NewCond(&q.mu)
    return q
}

func (q *Queue) Push(item string) {
    q.mu.Lock()
    for len(q.items) == q.cap {
        q.cond.Wait() // releases lock, waits for signal, re-acquires lock
    }
    q.items = append(q.items, item)
    q.cond.Broadcast()
    q.mu.Unlock()
}

func (q *Queue) Pop() string {
    q.mu.Lock()
    for len(q.items) == 0 {
        q.cond.Wait()
    }
    item := q.items[0]
    q.items = q.items[1:]
    q.cond.Broadcast()
    q.mu.Unlock()
    return item
}

func main() {
    q := NewQueue(2)

    go func() {
        for _, song := range []string{"$100 Bills", "Legend", "Escape"} {
            q.Push(song)
            fmt.Println("pushed:", song)
```

```
        }
    }()

    for i := 0; i < 3; i++ {
        fmt.Println("popped:", q.Pop())
    }
}
```

Trap: Always check the wait condition in a `for` loop, not an `if`. Unlike Java's `Object.wait()`, Go's `Cond.Wait` never wakes spuriously — but another goroutine may have consumed the item between the wakeup and re-acquiring the lock (and `Broadcast` wakes waiters whose predicate may already be false), so you must re-check the predicate before proceeding. The discipline is the same as Java's `while (condition) { lock.wait(); }`.

Tip: Prefer `Broadcast` over `Signal` when multiple goroutines could each satisfy the predicate. A spurious `Signal` that wakes the wrong waiter wastes a cycle; `Broadcast` wakes them all and lets them re-check.

Wut: `sync.Cond` cannot be used with Go's `select` statement. If you need to select between "condition is met" and a timeout or another channel, restructure using channels instead.

sync/atomic

The `sync/atomic` package provides low-level atomic memory operations. In Go 1.19, typed atomic types were added that are much safer than the old function-based API.

Typed Atomics (Go 1.19+)

```
import "sync/atomic"
```

The typed atomic types:

```
atomic.Bool                     // atomic bool
atomic.Int32, atomic.Int64      // atomic signed integers
atomic.Uint32, atomic.Uint64    // atomic unsigned integers
atomic.Uintptr                  // atomic uintptr
atomic.Pointer[T]               // atomic pointer to T (generic)
```

Each type provides the same set of methods:

```
func (x *Int64) Load() int64                        // read atomically
func (x *Int64) Store(val int64)                    // write atomically
func (x *Int64) Add(delta int64) (new int64)        // add and return new value
func (x *Int64) Swap(new int64) (old int64)         // set and return old value
func (x *Int64) CompareAndSwap(old, new int64) bool  // CAS: swap if current == old
```

`atomic.Pointer[T]` uses generics so you get type safety without a cast:

```
func (x *Pointer[T]) Load() *T          // load atomically
func (x *Pointer[T]) Store(val *T)      // store atomically
func (x *Pointer[T]) Swap(new *T) *T    // swap atomically
func (x *Pointer[T]) CompareAndSwap(old, new *T) bool // CAS
```

Here is a play-count tracker using atomic integers:

```go
package main

import (
    "fmt"
    "sync"
    "sync/atomic"
)

func main() {
    var plays atomic.Int64
    var wg sync.WaitGroup

    songs := []string{
        "Escape", "$100 Bills", "Legend",
        "J'ai pas vingt ans !", "J'en ai marre !",
    }

    for _, song := range songs {
        wg.Add(1)
        go func(s string) {
            defer wg.Done()
            plays.Add(1) // no mutex needed
            fmt.Printf("played: %s\n", s)
        }(song)
    }

    wg.Wait()
    fmt.Println("total plays:", plays.Load()) // 5
}
```

When to Use Atomics vs Mutexes

Use atomics for:

- Simple counters (hits, errors, bytes transferred).
- A single flag that goroutines read and one goroutine writes.
- Lock-free data structures where you understand the ABA problem.

The **ABA problem** is a subtle hazard in lock-free code built on `CompareAndSwap`: a value changes from A to B and back to A between your read and your CAS, so the CAS succeeds even though the underlying state was modified and restored in between. The successful swap hides the fact that other goroutines touched the data, which can corrupt structures like lock-free stacks.

Use a mutex when:

- You are protecting more than one variable together (invariant maintenance).
- The critical section does more than a single load/store/add.

Tip: Java's `java.util.concurrent.atomic.AtomicLong` maps directly to `atomic.Int64`. The semantics are the same: both behave as sequentially consistent atomics, and an atomic store observed by an atomic load establishes a happens-before edge for surrounding memory too (Go 1.19 memory model; Java `volatile` semantics). Still prefer a mutex when several variables must change together — atomics order memory but cannot make a multi-variable update atomic.

The Race Detector

Go ships a built-in race detector based on ThreadSanitizer. Enable it with the `-race` flag:

```
go test -race ./...
go run -race main.go
go build -race -o myapp .
```

When a data race is detected at runtime, the race detector prints a detailed report showing both the racing accesses and their goroutine stack traces:

```
==================
WARNING: DATA RACE
Write at 0x00c0000b4010 by goroutine 7:
  main.main.func1()
      /tmp/race.go:12 +0x2c

Previous read at 0x00c0000b4010 by goroutine 6:
  main.main.func1()
      /tmp/race.go:9 +0x30
==================
```

The race detector adds roughly 2–20x runtime overhead and 5–10x memory overhead. It is not suitable for production, but it should run in CI on every pull request and on every test suite.

Tip: Run `go test -race ./...` in CI on every push. Data races are undefined behavior: the program may produce wrong results, crash, or appear to work correctly on your machine while failing in production. The race detector is the only reliable way to find them.

Here is an example the race detector catches immediately:

```go
package main

import (
    "fmt"
    "sync"
)

func main() {
    counter := 0 // shared without synchronization
    var wg sync.WaitGroup
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            counter++ // DATA RACE: concurrent read-modify-write
        }()
    }
    wg.Wait()
    fmt.Println(counter) // result is unpredictable
}
```

The fix: replace `counter` with `atomic.Int64` or protect it with a `sync.Mutex`.

Try It

Type this in and run it twice with `go run .` and once with `go run -race .` to convince yourself it stays correct under the detector. It exercises four primitives at once: a `sync.Mutex` guarding a map, `sync.WaitGroup` to fan out and wait, `sync.Once` for one-time setup, and `atomic.Int64` for a lock-free counter.

```
package main

import (
    "fmt"
    "sort"
    "sync"
    "sync/atomic"
)

type Charts struct {
    mu    sync.Mutex     // protects spins
    spins map[string]int // title -> play count
}

func (c *Charts) Play(title string) {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.spins[title]++
}

func main() {
    charts := &Charts{spins: make(map[string]int)}
    var total atomic.Int64
    var ready sync.Once
    var wg sync.WaitGroup

    queue := []string{"Espresso", "Espresso", "Birds of a Feather", "Houdini"}
    for _, song := range queue {
        wg.Add(1)
        go func(s string) {
            defer wg.Done()
            ready.Do(func() { fmt.Println("dj booth online") }) // runs once
            charts.Play(s)
            total.Add(1)
        }(song)
    }

    wg.Wait()
    fmt.Println("total spins:", total.Load())

    titles := make([]string, 0, len(charts.spins))
    for t := range charts.spins {
        titles = append(titles, t)
    }
    sort.Strings(titles)
    for _, t := range titles {
        fmt.Printf("%s: %d\n", t, charts.spins[t])
    }
```

```
}
```

The "dj booth online" line prints exactly once and the total is always `4`, no matter how the goroutines interleave.

Try these modifications:

- Drop the `c.mu.Lock()` / `defer c.mu.Unlock()` from `Play` and run with `-race` — watch the detector flag the concurrent map writes.
- Swap the `atomic.Int64` for a plain `int` incremented with `total++` and confirm `-race` catches that too.
- Replace `charts.mu` with a `sync.RWMutex` and add a `Spins(title string) int` reader method that takes `RLock`, then call it concurrently with the writers.

Key Points

- The Go memory model defines **happens-before** relationships; without synchronization, goroutines may observe stale memory.
- `sync.Mutex` provides exclusive access; use `defer mu.Unlock()` immediately after `mu.Lock()`.
- `sync.RWMutex` allows concurrent readers; reach for it only when reads dominate and you have measured a contention problem.
- `sync.WaitGroup` is the idiomatic fan-out/fan-in primitive when you do not need to pass results back through a channel.
- `sync.Once` runs a function exactly once; it is the correct, simple replacement for double-checked locking.
- `sync.Cond` is a condition variable; always check the predicate in a `for` loop, not an `if`.
- `atomic.Int64`, `atomic.Pointer[T]`, and friends (Go 1.19+) are the type-safe atomic primitives; use them for simple counters and flags.
- Run `go test -race ./...` in CI; a data race is undefined behavior and may be silent on your laptop.

Exercises

1. **Think about it:** Java's `synchronized` keyword locks an object's monitor, which is built into every Java object. Go has no per-object monitor; instead you declare explicit `sync.Mutex` fields. What are the practical advantages and disadvantages of each approach? Consider: what happens when you need to protect two independent fields in the same struct, and how would you do it with each language's mechanism?

2. **What does this print?**

```go
package main

import (
    "fmt"
    "sync"
)

func main() {
    var once sync.Once
    var wg sync.WaitGroup
    results := make([]string, 3)

    for i := 0; i < 3; i++ {
        wg.Add(1)
        go func(n int) {
            defer wg.Done()
            once.Do(func() {
```

```
                results[n] = "loaded"
            })
            if results[n] == "" {
                results[n] = "skipped"
            }
        }(i)
    }

    wg.Wait()
    loaded := 0
    skipped := 0
    for _, r := range results {
        if r == "loaded" {
            loaded++
        } else if r == "skipped" {
            skipped++
        }
    }
    fmt.Printf("loaded=%d skipped=%d\n", loaded, skipped)
}
```

3. **Calculation:** Consider the following program fragment:

```
var counter atomic.Int64
var wg sync.WaitGroup

for i := 0; i < 4; i++ {
    wg.Add(1)
    go func() {
        defer wg.Done()
        counter.Add(10)
    }()
}
wg.Wait()
fmt.Println(counter.Load())
```

   (a) What value does `counter.Load()` always print, regardless of goroutine scheduling order?
   (b) If `counter.Add(10)` were replaced by `counter.Add(int64(i))` (capturing `i` from the loop), what value would always be printed? Would your answer have differed in Go 1.21 or earlier, and if so, why?

4. **Where is the bug?**

```
package main

import (
    "fmt"
    "sync"
)

type SafeMap struct {
    mu sync.Mutex
    m  map[string]int
}

func NewSafeMap() SafeMap {
```

```
    return SafeMap{m: make(map[string]int)}
}

func (s SafeMap) Inc(key string) {
    s.mu.Lock()
    defer s.mu.Unlock()
    s.m[key]++
}

func (s SafeMap) Get(key string) int {
    s.mu.Lock()
    defer s.mu.Unlock()
    return s.m[key]
}

func main() {
    sm := NewSafeMap()
    var wg sync.WaitGroup
    for i := 0; i < 100; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            sm.Inc("Escape")
        }()
    }
    wg.Wait()
    fmt.Println(sm.Get("Escape"))
}
```

5. **Write a program:** Implement a concurrent-safe `RateLimiter` struct that uses a `sync.Mutex` to protect a counter and a `time.Time` field tracking when the window resets. The struct should have a method `Allow(n int) bool` that returns `true` if `n` tokens are available in the current one-second window, deducting them if so, and `false` otherwise (without deducting). Write a `main` function that launches 10 goroutines, each calling `Allow(1)` in a loop 5 times, and prints how many calls were allowed versus denied across all goroutines combined. Use `sync.WaitGroup` to wait for all goroutines to finish.

6. **Where is the bug?**

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    var wg sync.WaitGroup
    results := make([]int, 5)
    for i := 0; i < 5; i++ {
        go func(n int) {
            wg.Add(1)
            defer wg.Done()
            results[n] = n * n
        }(i)
    }
```

```
    wg.Wait()
    fmt.Println(results)
}
```

The author expects this to print `[0 1 4 9 16]`, but it usually prints something like `[0 0 0 0 0]` or a partial result, and `go run -race` reports a data race on `results`. What is wrong, and how would you fix it? (Hint: where is `wg.Add(1)` called, and what does `go vet` say about it?)