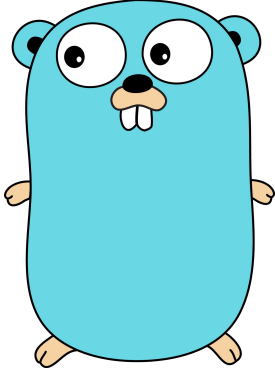
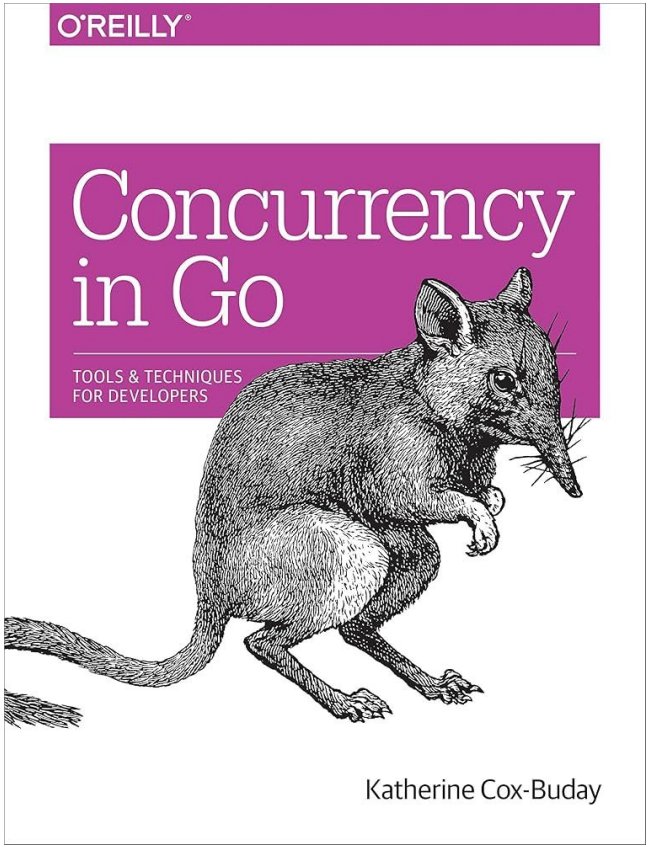


Concurrency in Golang

Persuasion to use Go in your next project

Zach Croft

Credits


O'REILLY
Concurrency in Go
TOOLS & TECHNIQUES FOR DEVELOPERS
Katherine Cox-Buday

Overview

- Intro to Golang structure and syntax
- Concurrency vs. parallelism
- Communicating Sequential Processes (CSP)
- Threads, coroutines, and goroutines *
- Go's **sync** package (traditional memory synchronization)
- Go's **channel** and **channel patterns** (CSP-style memory synchronization) *
- Go's **context** package
- Go's **synctest** package (new August 2025) *
- Why should a Ruby developer care?

Not planning on directly discussing (but some info is in the book for 1-3):

- Details of the "Communicating Sequential Processes" paper or CSP language syntax
- Examples and solutions to deadlocks, livelocks, and starvation
- Go runtime internals (particularly how goroutines are scheduled and channels work)
- Ruby's threads or Global Interpreter Lock (feel free to add input, but I didn't research these much)

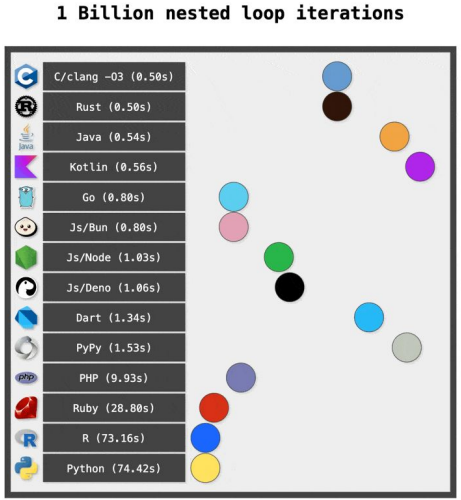
Intro to Go

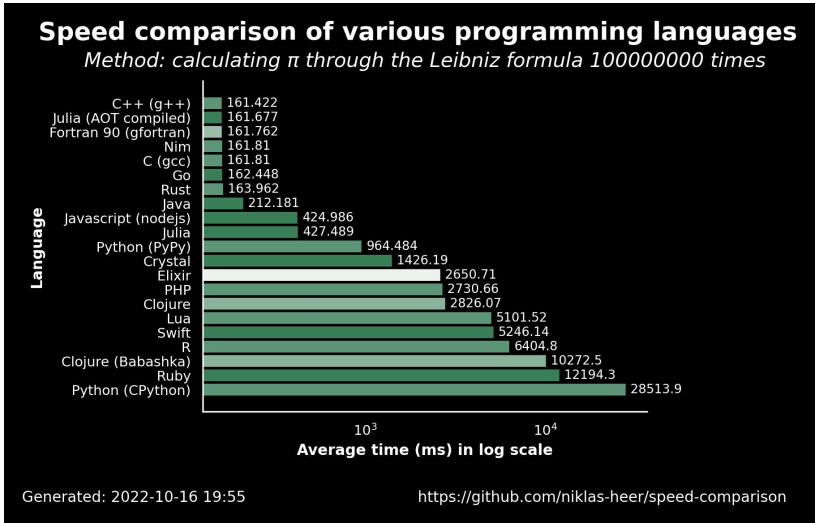
- Go is compiled, statically typed, and is exclusively garbage collected.
  - Unless you use the “unsafe” package in Go stdlib, to use malloc and free through invoking C.
- Go reverses typical syntax (int x = 3) to (var x int = 3)
- Go is not object oriented. Structs like C.
- To note in VSCode:
  - packages/modules
  - Building/executing a binary
  - Syntax:
    - Variables, pointers, functions, receivers, arrays/slices
    - Access modifiers, first class functions, duck typing/interfaces

… To VSCode

Speed

- While many benchmarks are often iffy because of unfair comparisons or unusual circumstances, Go tends to land slightly slower than Rust/C/C++, but much faster than Java, Python, Ruby, JavaScript, etc.





Concurrency vs Parallelism

“Concurrency is a property of the code; parallelism is a property of the running program.”

- Parallelism: tasks being run *at the same time*, across multiple cores/processors.
- Concurrency: tasks being structured to run simultaneously, whether they do or not.
  - Concurrent tasks on one core are alternated between rapidly, appearing simultaneous, but are really sequential (time-slicing: preemptive multitasking by CPU, giving each process fixed-length turns).
  - Concurrent code *can* be parallel, but doesn’t have to be.

“Communicating Sequential Processes” (CSP)

- Title of an ACM paper written in 1978 by Charles Antony Richard Hoare.
- Formal language for describing patterns of interaction in concurrent systems.
  - Emphasizes that concurrent processes (or functions) should be pure–no synchronized global state, all data is managed through I/O passed between processes.
  - Based on message passing via **channels**, which became popular in distributed systems.

Why should you care?

- CSP-style code, while sometimes long, is easy to reason about and typically safe.
  - * We will come back to this at the end, to handle hungry philosophers.
- CSP directly inspired Go’s **channels** and **select** statement.
- “Don't communicate by sharing memory; share memory by communicating.”

```
PHIL = *[... during ith lifetime ... →
        THINK;
        room!enter( );
        fork(i)!pickup( ); fork((i + 1) mod 5)!pickup( );
        EAT;
        fork(i)!putdown( ); fork((i + 1) mod 5)!putdown( );
        room!exit( )
        ]
```

<- Snippet of (intentionally buggy) CSP code from the paper

Threads, coroutines, and goroutines

- OS/kernel threads are 1:1 with hardware cores.
  - 12 cores means 12 tasks can be worked on in parallel.
- Many programming languages support threads. *These are not necessarily 1:1 with kernel threads.*
  - **Native threads** are 1:1 with OS threads. (e.g. POSIX threads in C).
    - Heavy memory consumption, OS manages scheduling, and slow context switches.
  - **Green threads** are M:N (M language threads mapped to N OS threads), and the language runtime manages scheduling between language threads.
- Coroutines:
  - *Concurrency in Go* defines coroutines as: “Concurrent subroutines that are non-preemptive…with multiple points throughout which allow for suspension or reentry.”
  - Similar to green threads: lightweight user-space constructs managed by runtime.
  - Difference lies in scheduling:
    - Green threads are preempted by the runtime.
    - Coroutines are based on cooperative scheduling, voluntarily giving up control at specific yield points.
      - e.g. async/await in JavaScript.

Goroutines

- Goroutines are unique to Go. They are a hybrid between green threads and coroutines.
  - Goroutines don’t define their own suspension or reentry points; Go’s runtime observes the runtime behavior and automatically suspends and resumes when blocked and unblocked.
- Very lightweight (10k empty goroutines ~ 2.8 kb in memory).
  - Native threads would be GBs of memory.
  - Context switch in Go runtime is ~92% faster than OS context switch (benchmark in book).
- There is always at least one goroutine (the main goroutine).

… To VSCode

Memory (GB)	Goroutines (#/100,000)	Order of magnitude
2^0	3.718	3
2^1	7.436	3
2^2	14.873	6
2^3	29.746	6
2^4	59.492	6
2^5	118.983	6
2^6	237.967	6
2^7	475.934	6
2^8	951.867	6
2^9	1903.735	9

sync package

- WaitGroup (seen during goroutines)
  - Concurrent-safe counter
    - Calls to *Add(n)* increment the counter by n, and calls to *Done* decrement by 1.
    - Calls to *Wait* block until the counter is zero.
- Mutex (Mutual exclusion) and RWMutex (Read/Write mutex)
  - Both Mutex and RWMutex the ‘sync.Locker’ interface: Lock(), Unlock()
    - defer lock.Unlock() to protect from deadlocking when panicking in a critical section.
  - RWMutex allows any # of goroutines to acquire a read lock, only while no write lock is held.
  - *“Regarding mutexes, the sync package implements them, but we hope Go programming style will encourage people to try higher-level techniques. In particular, consider structuring your program so that only one goroutine at a time is ever responsible for a particular piece of data.”*

sync package (continued)

- Cond
  - A rendezvous point for goroutines waiting for or announcing the occurrence of an event.
  - Goroutines can efficiently sleep until signaled to wake and check their condition.

… To VSCode

  - Not going to show a code example for time’s sake, but there is also a Broadcast() function on Cond. It allows a publish-subscribe pattern where all listeners can unblock.
  - This may prompt you to ask, if multiple goroutines are blocking on c.Wait() and another goroutine calls c.Signal(), what happens?
    - The runtime has a FIFO list of wait times and notifies the longest-waiting goroutine.

sync package (continued, again)

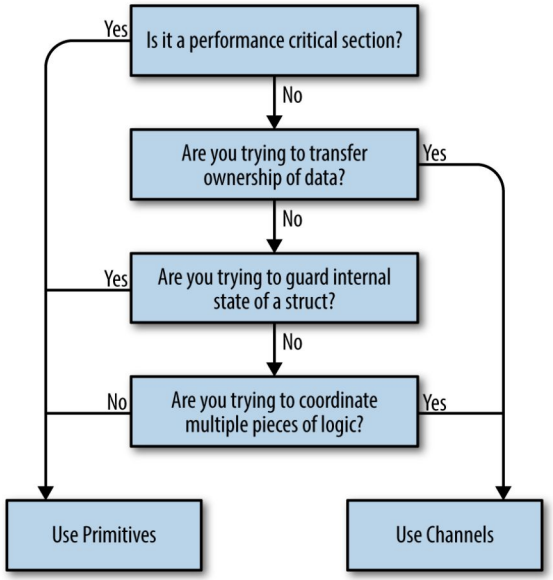
- sync.Once
- sync.Pool
  - “concurrent-safe implementation of the object pool pattern”

… To VSCode

Channels

- Derived from CSP, to communicate info between goroutines, *like a stream*.
- For sake of time, skipping details of channels.
- Send values with ‘channelVariable <- value’ operator.
- Read values with ‘value <- channelVariable’ operator.

… To VSCode until patterns



Channels and Channel Patterns

- Channels seem frightening–they can cause blocking, compilation errors, and panics all over.
  - **Assign channel ownership to limit this (confinement).**
- Confinement/Ownership:
- 1. Instantiate the channel.
- 2. Perform writes, or pass ownership to another goroutine.
- 3. Close the channel.
- 4. Encapsulate the previous three things in this list and expose them via a reader channel.

Operation	Channel state	Result
Read	`nil`	Block
	Open and Not Empty	Value
	Open and Empty	Block
	Closed	<default value>, false
	Write Only	Compilation Error
Write	`nil`	Block
	Open and Full	Block
	Open and Not Full	Write Value
	Closed	**panic**
	Receive Only	Compilation Error
`close`	`nil`	**panic**
	Open and Not Empty	Closes Channel; reads succeed until channel is drained, then reads produce default value
	Open and Empty	Closes Channel; reads produces default value
	Closed	**panic**
	Receive Only	Compilation Error

Channel Patterns

- Confinement (ownership) pattern
- For-select pattern
- Done channel pattern
  - Read-only channel to signal goroutines to stop their work.
  - Convention is to have ‘<- done’ as the last channel in a select statement.
- The or-channel (the coolest part of this presentation)
  - Multiplex data from n channels onto a single *notification* channel.
    - If you can’t know how many channels you will be reading from at runtime, pass a variadic slice of channels to the or function to return an or-channel.
- Fan-out, fan-in (omitting example code)
  - Multiplex data from n channels onto a single *data* channel, similar to or-channel.
- Tee-channel
  - Like tee command in bash (split input into two outputs):
  - alias testsuite='RAILS_ENV=test bundle exec rake test:parallel | tee testrun.log'

Channels and context package

- Wraps a ‘done’ channel in additional information, for the life of a request.
- Controller -> Model -> Repository layer
- Pass context through each layer, creating sub-contexts for differing behavior.
  - Select on **ctx.Done()** (Done() returns a done channel) in all layers
- Example:
  - Controller: ctx := context.Background() // new, empty context, pass into model
  - Model: ctxWithDeadline := context.WithDeadline(ctx, 2 * time.Second)
    - Pass the context with a deadline to the repository layer.
    - The repository layer selects on the done channel, and will abort after 2 second timeout.

```
func Background() Context
func TODO() Context
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc)
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
func WithValue(parent Context, key, val interface{}) Context
```

Channels and context package (continued)

- Controversial: context.WithValue(parent Context, key any , val any) Context
- An arbitrary grab-bag of **untyped** data (ahhhh scary).

```
func main() {
    ProcessRequest("jane", "abc123")
}

func ProcessRequest(userID, authToken string) {
    ctx := context.WithValue(context.Background(), "userID", userID)
    ctx = context.WithValue(ctx, "authToken", authToken)
    HandleResponse(ctx)
}

func HandleResponse(ctx context.Context) {
    fmt.Printf(
        "handling response for %v (%v)",
        ctx.Value("userID"),
        ctx.Value("authToken"),
    )
}
```

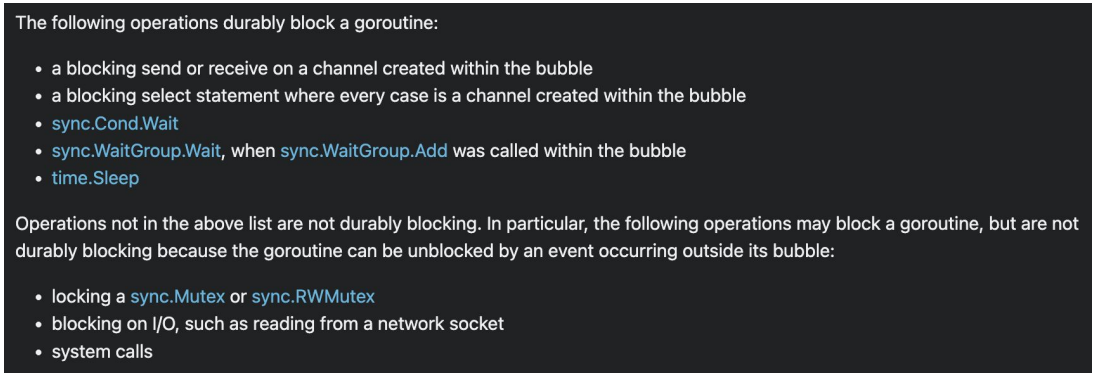
synctest (New in Go 1.25, 9/3/2025)

- Ok, big deal. We can write concurrent code, but can we test it?
  - synctest creates “bubbles” and any goroutines started within the bubble are also part of the it.
  - Within a bubble, the time package uses a fake clock. Each bubble has its own clock. The initial time is midnight UTC 2000-01-01.

```
func TestTime(t *testing.T) {
    synctest.Test(t, func(t *testing.T) {
        start := time.Now() // always midnight UTC 2000-01-01
        go func() {
            time.Sleep(1 * time.Second)
            t.Log(time.Since(start)) // always logs "1s"
        }()
        time.Sleep(2 * time.Second) // the goroutine above will run before this Sleep returns
        t.Log(time.Since(start))    // always logs "2s"
    })
}
```

synctest (New in Go 1.25, 9/3/2025) (continued)

- A goroutine in a bubble is "durably blocked" when it is blocked and can only be unblocked by another goroutine in the same bubble.
  - A goroutine which can be unblocked by an event from outside its bubble is not durably blocked.
- The Wait function blocks until all other goroutines in the bubble are durably blocked.
- A few specific operations unblock a durably blocked goroutine.



The following operations durably block a goroutine:

- a blocking send or receive on a channel created within the bubble
- a blocking select statement where every case is a channel created within the bubble
- sync.Cond.Wait
- sync.WaitGroup.Wait, when sync.WaitGroup.Add was called within the bubble
- time.Sleep

Operations not in the above list are not durably blocking. In particular, the following operations may block a goroutine, but are not durably blocking because the goroutine can be unblocked by an event occurring outside its bubble:

- locking a sync.Mutex or sync.RWMutex
- blocking on I/O, such as reading from a network socket
- system calls

synctest (New in Go 1.25, 9/3/2025) (continued)

- https://pkg.go.dev/testing/synctest
- https://victoriametrics.com/blog/go-synctest/
- https://www.youtube.com/watch?v=1ZIcsgkOvCk

```
func TestContextAfterFunc(t *testing.T) {
    synctest.Test(t, func(t *testing.T) {
        // Create a context.Context which can be canceled.
        ctx, cancel := context.WithCancel(t.Context())

        // context.AfterFunc registers a function to be called
        // when a context is canceled.
        afterFuncCalled := false
        context.AfterFunc(ctx, func() {
            afterFuncCalled = true
        })

        // The context has not been canceled, so the AfterFunc is not called.
        synctest.Wait()
        if afterFuncCalled {
            t.Fatalf("before context is canceled: AfterFunc called")
        }

        // Cancel the context and wait for the AfterFunc to finish executing.
        // Verify that the AfterFunc ran.
        cancel()
        synctest.Wait()
        if !afterFuncCalled {
            t.Fatalf("before context is canceled: AfterFunc not called")
        }
    })
}
```

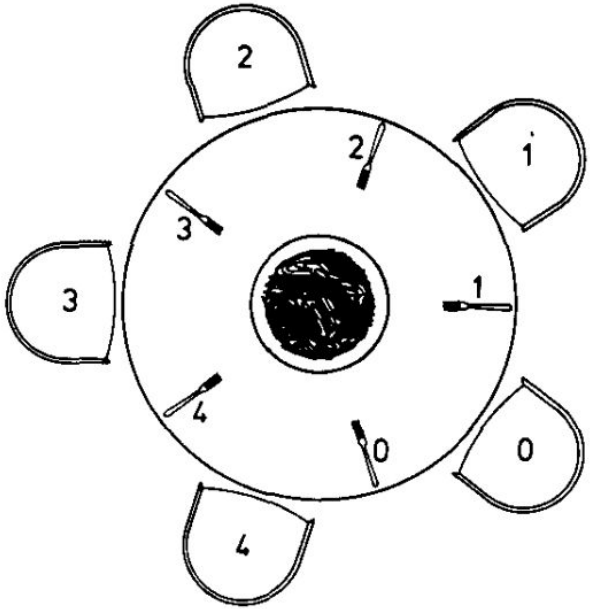
Dining Philosophers: The Return of CSP

- 5 philosophers sit around a table with 5 forks (one between each pair). Each philosopher alternates between thinking and eating.
- Philosophers are very hungry and want two forks to eat!
- To eat, a philosopher needs both adjacent forks (left and right).
- The Challenge: How do they coordinate without:
  - Deadlock - all grab left fork, wait forever for right fork?
  - Starvation - some philosopher never gets to eat?
- Simple representation:
  - Have 5 total forks, and grant 2 at a time to a philosopher.

… To VSCode

Naive implementation with mutexes can easily cause deadlocks.

CSP-style implementation with channels is easy to reason about and is safe!



Ok, so what? We are Ruby developers...

- Confinement (ownership) is a pattern that can still be applied, even without channels.
- Ruby has a Queue class:
  - Thread-safe FIFO queue for inter-thread communication. Like a channel!!
- Channels have also been recreated in gems.
- Go is cool.