JOB-READY SERIES · JOB-SWITCH EDITION

BASICS TO REDIS CLONE

Golang

The Job-Switch Guide · From Basics to a Redis Clone

Everything a Go backend interview asks about, in simple language: real code that compiles, real output, goroutines and channels, a tiny Redis you build yourself, and 73 interview questions with answers.




```
you@new-job: ~/goredis

$ go run ./cmd/goredis            ready to accept offers
$ redis-cli SET offer accepted     OK
$ go test -race ./...             ok  no races, no regrets
```

Syntax · Slices & Maps · Interfaces · Errors · Generics · Goroutines · Channels · Context · net/http · Testing · goredis

WRITTEN BY

Shreyash Tripathi

Contents

1 Getting Started with Go 3

Installing Go and your first program · The go command and project layout

2 Syntax and Control Flow 5

Packages, variables, constants and iota · Strings, bytes and runes · if, for and switch

3 Functions, defer and Errors 9

Functions · defer, panic and recover · Errors are values

4 Slices, Maps and Structs 13

Arrays and slices · Maps · Structs and embedding

5 Pointers, Methods, Interfaces and Generics 17

Pointers and methods · Interfaces, io.Reader and io.Writer · Type switches, any and the nil interface trap · Generics

6 Concurrency 21

Goroutines and WaitGroup · Channels and select · Mutex, RWMutex and atomic · context, leaks and data races

7 The Standard Library for Servers 27

net: listeners, connections and deadlines · bufio and io · time, log/slog and flag · net/http and encoding/json

8 Testing 31

Table-driven tests and t.Run · Helpers, temp dirs, examples and benchmarks

9 Mini Project: A Tiny Key-Value Server 34

Lab: the store and one client · Lab: accept loop and shutdown · Lab: run it, talk to it, test it · From toy to goredis

10 Gotchas 38

11 Go Interview Question Bank 40

Go fundamentals · Go concurrency · Networking and protocol · goredis design · Persistence and durability · Testing · DevOps and SRE · Telling the goredis story

APPENDIX A Answer Key 50

APPENDIX B Cheat Sheet 51

APPENDIX C Glossary and Where to Go Next 52

🎯 **INTERVIEW CORNER**
Questions interviewers really ask, with model answers.

✅ **CHECK YOUR PROGRESS**
Short questions after topics; exercise answers in Appendix A.

💼 **AT WORK**
How each idea shows up in goredis and on the job.

💡 **DO YOU KNOW?**
Real facts from the Go world.

😄 **FUNNY SIDE**
Because `if err != nil` needs jokes.

🧪 **HANDS-ON LAB**
Every program here compiles. Run them with `bash run_examples.sh`.

All code was compiled, checked with `go vet` and run with Go 1.27.4 on windows/amd64. Every output block is the real output, copied in by the build script from `go-basics-src/out/`.

CHAPTER 1

Getting Started with Go

Go was dreamt up while its creators waited for a big C++ build to finish. Some people make tea while compiling. They made a language.

After studying this chapter, you will be able to

- Explain what Go is and why so much server software is written in it.
- Install Go and use `go mod init`, `go run`, `go build`, `go test`, `go vet` and `gofmt`.
- Read the standard project layout with `cmd/` and `internal/`.

Think of a pressure cooker. It has very few parts, everybody in the house knows how to use it, and it gets dinner ready fast. Go is the pressure cooker of programming languages: a small, compiled, statically typed language with garbage collection, a fast compiler and concurrency built in. A Go program compiles into one self-contained binary with nothing extra to install, which is why Docker, Kubernetes and Prometheus are written in it.

1.1 Installing Go and your first program

Download Go from `go.dev/dl` and check it with `go version`. Every Go file starts with a `package` line, and a program starts at `func main` in package `main`. Code lives in a **module**: a folder with a `go.mod` file.

examples/hello/main.go

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, Go!")
}
```

TERMINAL: CREATE, RUN, BUILD

```
$ go mod init example.com/hello
go: creating new go.mod: module example.com/hello
$ go run .
Hello, Go!
$ go build
$ ls
go.mod
hello.exe
main.go
$ ./hello.exe
Hello, Go!
$ go vet ./...
$ gofmt -l .
$ cat go.mod
module example.com/hello

go 1.27.4
```

1.2 The go command and project layout

Command	What it does
`go mod init NAME`	Create `go.mod`. NAME is usually a repository path.
`go run .` / `go build`	Compile and run / compile a binary (`-o bin/app ./cmd/app`).
`go test ./...`	Run tests in this folder and all folders below it.
`go vet ./...`	Find likely bugs the compiler allows (wrong `Printf` verb, copied mutex).
`gofmt -l .` / `-w .`	List / fix unformatted files. Most editors run it on save.
`go mod tidy`	Add missing and remove unused dependencies.

Bigger projects put each binary under `cmd/` and the real code in packages under `internal/`. The compiler enforces `internal/`: only code in the same module can import it.

```
goredis/
  go.mod                    # module goredis
  cmd/goredis/main.go       # package main: flags, signals, wiring
  internal/protocol/        # RESP parser and reply writer
  internal/command/         # command table and handlers
  internal/store/           # the data: map + RWMutex + TTLs
  internal/server/          # TCP accept loop, one goroutine per client
  internal/persistence/     # append-only file
```

💼 **AT WORK: GOREDIS**

This is the real goredis layout. Each folder is one package that can be tested alone: the parser without a network, the store without a parser. Import paths are the module name plus the folder, such as `goredis/internal/store`.

💡 **DO YOU KNOW?**

Go was designed at Google by Robert Griesemer, Rob Pike and Ken Thompson, announced in 2009, and reached 1.0 in 2012 with a promise that Go 1 code keeps compiling. That promise still holds.

😄 **FUNNY SIDE**

"Gofmt's style is no one's favourite, yet gofmt is everyone's favourite." Tabs versus spaces? Go settled it for you, and nobody had to leave the team.

✅ **CHECK YOUR PROGRESS 1.1**

1. **What does `./...` mean in `go test ./...`?**
   Answer: every package in the current folder and all folders below it.
2. **Who can import a package under `internal/`?**
   Answer: only code inside the same module (strictly, inside the parent of `internal`).

Exercises

1. Which command finds a wrong `Printf` verb that still compiles? (a) gofmt (b) go vet (c) go build (d) go mod tidy
2. Short answer: why is a Go program easy to ship in a container?

CHAPTER 2

Syntax and Control Flow

Go has exactly one loop keyword. It is called for. For everything. For real.

After studying this chapter, you will be able to

- Declare variables and constants, use `iota`, and rely on zero values.
- Convert between types and tell strings, bytes and runes apart.
- Use every form of `if`, `for` and `switch`, including labeled `break`.

A kirana shop ledger has a fixed column for every item, and an empty row still means something: zero sold. Go variables work the same way. Every variable has a type, and a declared variable is never garbage: it starts at a sensible **zero value**.

2.1 Packages, variables, constants and iota

A name that starts with a capital letter (`Store`) is **exported** and visible to other packages; a lower-case name is private. `:=` declares and assigns inside functions; `var` works everywhere. `iota` counts up inside a `const` block, which is Go's way to write enums. Go never converts types for you, and unused variables or imports are compile errors.

examples/variables/main.go

```go
package main

import (
    "fmt"
    "strconv"
)

// Package-level declarations use var or const (no := outside functions).
var serverName = "goredis"

const maxClients = 1000 // an untyped constant

type Role int

const (
    Leader    Role = iota // 0
    Follower              // 1 (the expression repeats, iota grows)
    Candidate             // 2
)

const (
    _  = iota             // skip 0
    KB = 1 << (10 * iota) // 1 << 10
    MB                    // 1 << 20
    GB                    // 1 << 30
)

func main() {
    // Three ways to declare a variable.
    var port int = 6379
    var host = "localhost" // type inferred from the value
    debug := false         // short form, only inside functions

    // Zero values: every variable starts with a usable default.
    var (
        n   int
        f   float64
        s   string
        ok  bool
```

```
        p   *int
        buf []byte
        m   map[string]int
    )
    fmt.Println(serverName, host, port, debug, maxClients)
    fmt.Printf("%d %v %q %v %v %v %v\n", n, f, s, ok, p, buf == nil, m == nil)
    fmt.Println(Leader, Follower, Candidate, KB, MB, GB)

    // Conversions are always explicit.
    total, count := 10, 4
    avg := float64(total) / float64(count)
    fmt.Println(total/count, avg) // integer division truncates

    // Numbers <-> text go through strconv.
    fmt.Println(strconv.Itoa(port) + "!")
    v, err := strconv.Atoi("42x")
    fmt.Println(v, err)
    fmt.Println(string(rune(65))) // string(rune) gives a character: "A"
}
```

OUTPUT

```
go run ./examples/variables
```

```
goredis localhost 6379 false 1000
0 0 "" false <nil> true true
0 1 2 1024 1048576 1073741824
2 2.5
6379!
0 strconv.Atoi: parsing "42x": invalid syntax
A
```

2.2 Strings, bytes and runes

A string is a read-only sequence of bytes, usually UTF-8. `len` counts bytes, indexing returns a byte, and `range` decodes runes (Unicode code points). Sockets read and write `[]byte`.

examples/stringsbytes/main.go — package and imports omitted

```
func main() {
    s := "héllo, 世界"
    fmt.Println(len(s), utf8.RuneCountInString(s)) // bytes vs characters
    fmt.Println(s[0], s[1])                        // indexing gives bytes

    for i, r := range "héllo" { // range decodes UTF-8 into runes
        fmt.Printf("%d:%c ", i, r)
    }
    fmt.Println()

    b := []byte("SET") // string -> []byte copies the bytes
    b[0] = 'G'
    fmt.Println(string(b), []byte("hi"), []rune("世界"))

    // Strings are immutable. Build them up with strings.Builder.
    var sb strings.Builder
    for i := range 3 {
        fmt.Fprintf(&sb, "k%d ", i)
    }
    fmt.Printf("%q\n", sb.String())

    fields := strings.Fields("  SET   user:1  alice ")
    fmt.Println(len(fields), fields)
    fmt.Println(strings.HasPrefix("user:1", "user:"), strings.Split("a,b,c", ","))
    fmt.Println(strings.EqualFold("get", "GET"), strings.Repeat("=", 5))
    key, val, found := strings.Cut("name=gopher", "=")
    fmt.Println(key, val, found)
}
```

OUTPUT
```
go run ./examples/stringsbytes
```

```
14 9
104 195
0:h 1:é 3:l 4:l 5:o
GET [104 105] [19990 30028]
"k0 k1 k2 "
3 [SET user:1 alice]
true [a b c]
true =====
name gopher true
```

> ⚠ **COMMON MISTAKE**
>
> `"héllo"` is 6 bytes, not 5: `é` takes two bytes in UTF-8, which is why the range index jumps from 1 to 3. Use `utf8.RuneCountInString` to count characters.

2.3 if, for and switch

No parentheses around conditions, braces always required. `if v, err := f(); err != nil` keeps `v` and `err` inside the if. `for` has four shapes, including `for i := range 10` (Go 1.22+). `switch` cases do not fall through. A labeled `break outer` leaves the outer loop.

examples/control/main.go — package and imports omitted

```go
func main() {
    // if with an init statement: n and err exist only inside if/else.
    if n, err := strconv.Atoi("6379"); err != nil {
        fmt.Println("bad port:", err)
    } else {
        fmt.Println("port", n)
    }

    // for is Go's only loop keyword. It has several shapes.
    for i := 0; i < 3; i++ { // classic three-part loop
        fmt.Print(i, " ")
    }
    n := 1
    for n < 100 { // condition only: a "while" loop
        n *= 3
    }
    fmt.Println("| n =", n)

    for i := range 3 { // range over an integer (Go 1.22+)
        fmt.Print(i, " ")
    }
    for i, cmd := range []string{"GET", "SET"} { // index and value
        fmt.Print(i, "=", cmd, " ")
    }
    fmt.Println()

    retries := 0
    for { // infinite loop: leave with break or return
        retries++
        if retries == 3 {
            break
        }
    }
    fmt.Println("retries:", retries)

    // switch: no fallthrough by default, and a case can list values.
    for _, cmd := range []string{"get", "SET", "ping", "FLUSHALL"} {
        switch strings.ToUpper(cmd) {
        case "GET", "SET":
            fmt.Println(cmd, "-> key command")
        case "PING":
            fmt.Println(cmd, "-> PONG")
        default:
            fmt.Println(cmd, "-> unknown")
        }
```

OUTPUT
```
go run ./examples/control
```

```
port 6379
0 1 2 | n = 243
0 1 2 0=GET 1=SET
retries: 3
get -> key command
SET -> key command
ping -> PONG
FLUSHALL -> unknown
medium
negative at 1 1
```

```
    }

    // A switch with no expression is a tidy if/else-if chain.
    size := 2048
    switch {
    case size < 1024:
        fmt.Println("small")
    case size < 1<<20:
        fmt.Println("medium")
    default:
        fmt.Println("large")
    }

    // A labeled break leaves the outer loop, not just the inner one.
    grid := [][]int{{1, 2}, {3, -1}, {5, 6}}
outer:
    for r, row := range grid {
        for c, v := range row {
            if v < 0 {
                fmt.Println("negative at", r, c)
                break outer
            }
        }
    }
}
```

INTERVIEW CORNER

Q: How do you write a while loop in Go? **A:** With `for` and only a condition: `for n < 100 { ... }`. A bare `for { }` loops forever until `break` or `return`.

Q: Inside a `for` with a `select`, why does `break` not stop the loop? **A:** It only leaves the `select`. Use a label (`break loop`) or `return`.

CHECK YOUR PROGRESS 2.1

1. **What is the zero value of a map, and can you write to it?**
   Answer: `nil`. Reading works, writing panics.
2. **What does `7/2` give for integers?**
   Answer: `3`. Integer division truncates; use `float64(7)/2` for 3.5.

Exercises

1. What does `len("héllo")` return? (a) 5 (b) 6 (c) 4 (d) it does not compile
2. What does `const ( A = iota; B; C )` make C equal to? (a) 0 (b) 1 (c) 2 (d) 3
3. Short answer: when do you use `var` instead of `:=`?