GO 快速参考

语法、类型、并发、错误处理精要

基础

Hello World

```
package main
import "fmt"
func main() {
    fmt.Println("Hello, World!")
}
```

运行与构建

```
go run main.go        # compile and run
go build -o app .     # compile to binary
go test ./...         # run all tests
```

模块初始化

```
go mod init github.com/user/project
go mod tidy           # sync dependencies
```

变量与类型

声明

```
var name string = "Go"
age := 15             // short declaration
var x, y int = 1, 2
const Pi = 3.14159
```

基础类型

`bool`	`true`、`false`
`string`	UTF-8 不可变字节序列
`int, int8..int64`	有符号整数（平台/固定宽度）
`uint, uint8..uint64`	无符号整数
`float32, float64`	IEEE-754 浮点数
`byte`	`uint8` 的别名
`rune`	`int32` 的别名（Unicode 码点）

零值

`int, float`	`0`
`bool`	`false`
`string`	`""`（空字符串）
`pointer, slice, map`	`nil`

函数

基础函数

```
func add(a, b int) int {
    return a + b
}
```

多返回值

```
func divide(a, b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("division by zero")
    }
    return a / b, nil
}
```

可变参数与匿名函数

```
func sum(nums ...int) int {
    total := 0
    for _, n := range nums { total += n }
    return total
}
double := func(x int) int { return x * 2 }
```

Defer

```
func readFile(path string) {
    f, _ := os.Open(path)
    defer f.Close()   // runs when function returns
}
```

控制流

If / Else

```
if x > 0 {
    fmt.Println("positive")
} else if x == 0 {
    fmt.Println("zero")
} else {
    fmt.Println("negative")
}
```

For 循环

```
for i := 0; i < 10; i++ { }  // classic
for x < 100 { x *= 2 }       // while-style
for { break }                // infinite
for i, v := range slice { }   // range
```

Switch

```
switch day {
case "Mon", "Tue":
    fmt.Println("early week")
case "Fri":
    fmt.Println("TGIF")
default:
    fmt.Println("other")
}
```

结构体与方法

结构体定义

```
type User struct {
    Name  string
    Email string
    Age   int
}
u := User{Name: "Alice", Email: "a@b.com", Age: 30}
```

方法

```
func (u User) Greeting() string {
    return "Hi, " + u.Name
}
func (u *User) SetAge(age int) {
    u.Age = age   // pointer receiver mutates
}
```

嵌入

```
type Admin struct {
    User        // embedded struct
    Level string
}
a := Admin{User: User{Name: "Bob"}, Level: "super"}
fmt.Println(a.Name)  // promoted field
```

接口

定义与实现

```
type Stringer interface {
    String() string
}
// implicit implementation — no "implements" keyword
func (u User) String() string {
    return u.Name
}
```

常用接口

`io.Reader`	`Read(p []byte) (n int, err error)`
`io.Writer`	`Write(p []byte) (n int, err error)`
`fmt.Stringer`	`String() string`
`error`	`Error() string`

类型断言

```
var i interface{} = "hello"
s, ok := i.(string)   // ok == true
switch v := i.(type) {
case string: fmt.Println(v)
case int:    fmt.Println(v * 2)
}
```

Goroutine 与 Channel

Goroutine

```
go func() {
    fmt.Println("running concurrently")
}()
time.Sleep(time.Second)
```

Channel

```
ch := make(chan int)       // unbuffered
buf := make(chan int, 5)   // buffered
ch <- 42                   // send
val := <-ch                // receive
```

Select

```
select {
case msg := <-ch1:
    fmt.Println(msg)
case ch2 <- 42:
    fmt.Println("sent")
case <-time.After(time.Second):
    fmt.Println("timeout")
}
```

常用模式

`sync.WaitGroup`	等待多个 goroutine 完成
`sync.Mutex`	共享状态的互斥锁
`context.Context`	取消、超时、请求范围的值

错误处理

基本模式

```
result, err := doSomething()
if err != nil {
    return fmt.Errorf("failed: %w", err)
}
```

自定义错误

```
type NotFoundError struct {
    ID string
}
func (e *NotFoundError) Error() string {
    return "not found: " + e.ID
}
```

errors 包

`errors.New(msg)`	创建简单错误
`fmt.Errorf("%w", err)`	带上下文包装错误
`errors.Is(err, target)`	在错误链中查找匹配
`errors.As(err, &target)`	从链中提取类型化错误

Slice 与 Map

Slice

```
s := []int{1, 2, 3}
s = append(s, 4, 5)
sub := s[1:3]              // [2, 3]
cp := make([]int, len(s))
copy(cp, s)
```

Map

```
m := map[string]int{"a": 1, "b": 2}
m["c"] = 3
val, ok := m["a"]        // ok == true
delete(m, "b")
for k, v := range m { }
```

Slice 操作

`len(s)`	元素数量
`cap(s)`	底层数组容量
`append(s, elems...)`	追加元素，可能重新分配
`copy(dst, src)`	在 slice 间复制元素
`slices.Sort(s)`	排序 slice（Go 1.21+ `slices` 包）

包与导入

导入方式

```
import "fmt"
import (
    "os"
    "strings"
    "github.com/user/pkg"
)
```

可见性

首字母大写 = 导出（公开）。
首字母小写 = 未导出（包私有）。
无需 public/private 关键字。

常用标准库

`fmt`	格式化 I/O（Print、Sprintf、Errorf）
`os`	OS 函数（文件、环境变量、参数）
`io`	I/O 原语（Reader、Writer）
`net/http`	HTTP 客户端与服务器
`encoding/json`	JSON 编码/解码
`strings`	字符串处理函数
`strconv`	字符串 ⇔ 数字转换
`testing`	单元测试框架

泛型

类型参数

```
func Map[T, U any](s []T, f func(T) U) []U {
    r := make([]U, len(s))
    for i, v := range s { r[i] = f(v) }
    return r
}
```

约束

```
type Number interface {
    ~int | ~float64
}
func Sum[T Number](nums []T) T {
    var total T
    for _, n := range nums { total += n }
    return total
}
```

测试

基础测试

```
// file: math_test.go
func TestAdd(t *testing.T) {
    got := Add(2, 3)
    if got != 5 {
        t.Errorf("Add(2,3) = %d, want 5", got)
    }
}
```

测试命令

`go test`	运行当前包的测试
`go test ./...`	递归运行所有测试
`go test -v`	详细输出
`go test -run TestAdd`	按名称运行指定测试
`go test -bench .`	运行基准测试
`go test -cover`	显示覆盖率百分比