Go クイックリファレンス

構文、型、並行処理、エラーハンドリングの基礎

基本

Hello World

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

実行とビルド

```
go run main.go        # コンパイルして実行
go build -o app .     # バイナリにコンパイル
go test ./...         # すべてのテストを実行
```

モジュールの初期化

```
go mod init github.com/user/project
go mod tidy           # 依存関係を同期
```

変数と型

宣言

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

基本型

bool	**true**、**false**
string	UTF-8 の不変バイト列
int, int8..int64	符号付き整数（プラットフォーム / 固定幅）
uint, uint8..uint64	符号なし整数
float32, float64	IEEE-754 浮動小数点
byte	**uint8** の別名
rune	**int32** の別名（Unicode コードポイント）

ゼロ値

int, float	**0**
bool	**false**
string	**""**（空文字列）
pointer, slice, map	**nil**

関数

基本関数

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

複数の戻り値

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

可変長引数と無名関数

```
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()   // 関数が返るときに実行
}
```

制御フロー

if / else

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

for ループ

```
for i := 0; i < 10; i++ { }  // 従来型
for x < 100 { x *= 2 }       // while スタイル
for { break }                 // 無限ループ
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")
}
```

構造体とメソッド

構造体の定義

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

メソッド

```
func (u User) Greeting() string {
    return "Hi, " + u.Name
}
func (u *User) SetAge(age int) {
    u.Age = age   // ポインタレシーバーで変更
}
```

埋め込み

```
type Admin struct {
    User          // 埋め込み構造体
    Level string
}
a := Admin{User: User{Name: "Bob"}, Level: "super"}
fmt.Println(a.Name)  // 昇格フィールド
```

インターフェース

定義と実装

```
type Stringer interface {
    String() string
}
// 暗黙の実装 — "implements" キーワードなし
func (u User) String() string {
    return u.Name
}
```

よく使うインターフェース

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**

型アサーション

```
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)
}
```

ゴルーチンとチャンネル

ゴルーチン

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

チャンネル

```
ch := make(chan int)       // バッファなし
buf := make(chan int, 5)   // バッファ付き
ch <- 42                   // 送信
val := <-ch                // 受信
```

select

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

パターン

sync.WaitGroup	複数のゴルーチンの完了を待つ
sync.Mutex	共有状態のための相互排除ロック
context.Context	キャンセル、デッドライン、リクエストスコープの値

エラーハンドリング

基本パターン

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

Go クイックリファレンス

カスタムエラー

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

errors パッケージ

errors.New(msg)	シンプルなエラーを作成
fmt.Errorf("%w", err)	コンテキスト付きでエラーをラップ
errors.Is(err, target)	エラーチェーンで一致を確認
errors.As(err, &target)	チェーンから型付きエラーを取り出す

スライスとマップ

スライス

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

マップ

```
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 { }
```

スライス操作

len(s)	要素数
cap(s)	内部配列の容量
append(s, elems...)	要素を追加（再割り当てが発生する場合あり）
copy(dst, src)	スライス間で要素をコピー
slices.Sort(s)	スライスをソート（Go 1.21+ の **slices** パッケージ）

パッケージとインポート

インポートスタイル

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

可視性

先頭が大文字 = エクスポート済み（公開）。
先頭が小文字 = エクスポートなし（パッケージプライベート）。
public/private などのキーワードは不要。

よく使う標準ライブラリ

fmt	フォーマット I/O（Print、Sprintf、Errorf）
os	OS 関数（ファイル、環境変数、引数）
io	I/O プリミティブ（Reader、Writer）
net/http	HTTP クライアントとサーバー
encoding/json	JSON エンコード/デコード
strings	文字列操作関数
strconv	文字列と数値の変換
testing	ユニットテストフレームワーク

ジェネリクス

型パラメーター

```
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
}
```

制約

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

テスト

基本テスト

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

テストコマンド

go test	カレントパッケージのテストを実行
go test ./...	すべてのテストを再帰的に実行
go test -v	詳細な出力
go test -run TestAdd	名前で特定のテストを実行
go test -bench .	ベンチマークを実行
go test -cover	カバレッジのパーセンテージを表示