BEST SELLER

MASTER GO LANGUAGE LIBRARIES

ESSENTIAL TOOLS FOR RAPID LEARNING

Explore 100 Essential Libraries In Just 1 Hour
With Our Best Comprehensive Guide

KANRO TOMOYA

Index

43. urfave/cli
44. fsnotify
45. gocql
46. goquery
47. gorilla/pat
48. GoDotEnv
49. golang.org/x/net/websocket
50. github.com/spf13/viper
51. olivere/elastic
52. fsnotify/fsnotify
53. github.com/Shopify/sarama
54. github.com/lib/pq
55. goroutinepool
56. gota
57. mgo
58. bleve
59. gobuffalo/pop
60. go-redis/redis

Chapter 1 Introduction

1. Purpose

Welcome to an exciting journey through the world of Go programming! This book is designed to offer a comprehensive dive into Go's standard library, showcasing a hundred different pac kages through practical examples and concise explanations.

Whether you are a beginner looking to get a solid foundation or an intermediate programmer aiming to enhance your Go toolkit, this guide is tailored to help you achieve mastery in just o ne hour per topic.

Here, you will learn to leverage Go’s robust standard packages to build more efficient and rel iable applications. Each chapter focuses on a unique package, exploring its functionalities, c ommon uses, and some tips and tricks to get the most out of it.

Dive in and start enhancing your Go programming skills today!

Chapter 2 standard library

1. net/http

The net/http package in Go provides HTTP client and server implementations.

Ex:net/http

```
package main
import (
	"fmt"
	"net/http"
)
func main() {
	http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
		fmt.Fprintln(w, "Hello, HTTP!")
	})
	http.ListenAndServe(":8080", nil)
}
```

When you run this server and access http://localhost:8080 in your browser, you will see the message: "Hello, HTTP!"

This code snippet demonstrates a basic HTTP server in Go. Here’s a breakdown:Import fmt and net/http packages: fmt for formatting and output, net/http for handling HTTP requests.http.HandleFunc("/") sets up a route handler for the root URL ("/"). Whenever this route is accessed, the function specified as the second argument is called.Inside the function, fmt.Fprintln(w, "Hello, HTTP!") sends a string back to the client. w is an http.ResponseWriter, which is used to write the HTTP response.http.ListenAndServe(":8080", nil) starts the server on port 8080. nil indicates that the default server multiplexer, http.DefaultServeMux, is used for routing. This example creates a very basic web server that can be accessed by visiting localhost:8080 in a web browser, where it simply displays a greeting message.

2. os

The os package in Go provides a platform-independent interface to operating system functio nality.

Ex:os

```
package main
import (
	"fmt"
	"os"
)
func main() {
	file, err := os.Create("example.txt")
	if err != nil {
		fmt.Println("Error creating file:", err)
		return
	}
	defer file.Close()
	_, err = file.WriteString("Hello, file handling!")
	if err != nil {
		fmt.Println("Error writing to file:", err)
		return
	}
}
```

The file "example.txt" is created in the current directory with the text "Hello, file handling!" ins ide it.

This code snippet demonstrates basic file handling using the os package:Import fmt and os: f mt for output and error handling, os for interacting with the operating system.os.Create("exa mple.txt") attempts to create a file named "example.txt". It returns a file handle and an error. If an error occurs, the error handling block will print it and stop further execution.defer file.Clo se() schedules the file.Close() method to be called when the main function completes, ensuri ng that the file handle is properly closed after the operations are done.file.WriteString("Hello, file handling!") writes a string to the file. It returns the number of bytes written and an error if any.Error handling is present to manage potential issues during file creation and writing.This example provides a simple demonstration of creating and writing to a file, showcasing basic error handling and file operations.

3. log

The "log" package in Go provides a simple logging facility, allowing you to output formatted a nd timestamped log entries.

Ex:log

```
package main
import (
    "log"
    "os"
)
func main() {
    // Creating a log file
    file, err := os.Create("example.log")
    if err != nil {
        log.Fatal(err)  // Logs a message and then calls os.Exit(1)
    }
    defer file.Close()
    // Setting the output of the logger to the file
    log.SetOutput(file)
    // Print info log with date and time
    log.Println("This is an informational message")
}
```

The log file example.log will contain a timestamped entry like:csharpCopy code2009/11/10 2 3:00:00 This is an informational message
(Note: Timestamp will vary based on when the code is run.)

In the code above, we start by importing the necessary packages. The log package is used f or logging, and os is used for file handling operations such as creating a file. Here's what ha ppens in the code:File Creation: We attempt to create a file named "example.log". If an error occurs during file creation (e.g., due to permission issues), log.Fatal(err) will log the error an d then terminate the program with an exit code of 1.Set Logger Output: By default, the log pa ckage writes to standard error (stderr). We change this behavior by setting the logger's outp ut destination to the file we just created.Logging a Message: We then log an informational m essage. This message is automatically prefixed with the current date and time, thanks to the default logger setup.This basic setup is very useful for adding logs to your application which can help in debugging or monitoring the software behavior over time.

4. html/template

The "html/template" package in Go is used for data-driven templates that generate HTML ou tput safe against code injection.

Ex:html/template

```
package main
import (
    "html/template"
    "os"
)
func main() {
    // Define a template with a placeholder
    tmpl := `<!DOCTYPE html>
<html>
<head>
    <title>{{.Title}}</title>
</head>
<body>
    <h1>{{.Header}}</h1>
    <p>{{.Message}}</p>
</body>
</html>`
    // Parse the template string
    t, err := template.New("webpage").Parse(tmpl)
    if err != nil {
        panic(err)
    }
    // Data to fill the template
    data := struct {
        Title   string
        Header  string
        Message string
    }{
        Title:   "Test Page",
        Header:  "Welcome to the Test Page",
        Message: "This is a test message.",
    }
    // Execute the template and output to os.Stdout
    t.Execute(os.Stdout, data)
}
```

```
<!DOCTYPE html>
```

```
<html>
<head>
    <title>Test Page</title>
</head>
<body>
    <h1>Welcome to the Test Page</h1>
    <p>This is a test message.</p>
</body>
</html>
```

This example demonstrates how to use the html/template package to create and execute HTML templates. Here’s a breakdown:Template Definition: We start by defining a string that acts as our HTML template. It includes placeholders ({{.Title}}, {{.Header}}, {{.Message}}) that will be replaced by actual values.Parsing the Template: The template string is then parsed into a template object. Errors in parsing (such as syntax errors in the template) would cause the program to panic, though in a production setting, you'd likely handle this more gracefully.Data Structure: We define a data structure that matches the placeholders in our template. This data is passed to the template during execution.Executing the Template: Finally, the template is executed with the provided data, and the output is sent directly to os.Stdout (standard output), but it could also be written to a file or an HTTP response body.This process ensures that any data used in the template is appropriately escaped, preventing Cross-Site Scripting (XSS) attacks, making it a safer choice for generating dynamic HTML content.

5. crypto/hmac

This package implements key-hashed message authentication code (HMAC) support. HMA C is a mechanism for message authentication using cryptographic hash functions.

Ex:crypto/hmac

```
package main
import (
	"crypto/hmac"
	"crypto/sha256"
	"encoding/hex"
	"fmt"
)
func main() {
	key := []byte("secret")
	message := []byte("Message to hash")
	hash := hmac.New(sha256.New, key)
	hash.Write(message)
	signature := hash.Sum(nil)
	fmt.Println(hex.EncodeToString(signature))
}
```

```
9a0d60c9b4f4b04e97f616157069c47b227a1f03124a000747b6ef805fd5d1bb
```

In the provided Go code, we first import necessary packages for HMAC with SHA256 hashin g. We define a 'key' and a 'message' that we want to secure. The hmac.New(sha256.New, k ey) function initializes a new HMAC object using SHA256 for hashing and the secret key. Th e Write method is used to input the message into the hash object, and Sum(nil) calculates th e HMAC value. Finally, we convert the HMAC byte slice to a hex string to make it readable, which is printed as the output.

6. crypto/cipher

The package provides interfaces to generic block cipher implementations. This package is ty pically used to implement encryption and decryption of data using various symmetric key alg orithms.

Ex:crypto/cipher

```
package main
import (
	"crypto/aes"
	"crypto/cipher"
	"crypto/rand"
	"fmt"
	"io"
)
func main() {
	key := []byte("examplekey123456") // 16 bytes for AES-128
	block, err := aes.NewCipher(key)
	if err != nil {
		panic(err)
	}
	plaintext := []byte("This is some text that needs encryption")
	ciphertext := make([]byte, aes.BlockSize+len(plaintext))
	iv := ciphertext[:aes.BlockSize] // initialization vector
	if _, err := io.ReadFull(rand.Reader, iv); err != nil {
		panic(err)
	}
	stream := cipher.NewCFBEncrypter(block, iv)
	stream.XORKeyStream(ciphertext[aes.BlockSize:], plaintext)
	fmt.Printf("Encrypted: %x\n", ciphertext)
}
```

```
Encrypted: 16byteIVfollowedbyencrypteddata
```

This code demonstrates the usage of the crypto/cipher package to encrypt plaintext using th e AES algorithm in CFB mode. Initially, we generate a new AES cipher block with a 16-byte key suitable for AES-128. The aes.NewCipher function initializes the AES cipher with the pro vided key. The ciphertext slice is prepared to include space for the initialization vector (IV) an d the actual ciphertext. We then populate the IV with random bytes, ensuring that each encry ption session is unique. The cipher.NewCFBEncrypter creates a stream cipher from the AES block cipher in CFB mode, and XORKeyStream encrypts the plaintext by XORing it with the keyed pseudo-random stream generated from the IV and key. The resulting encrypted data i s printed in a hexadecimal format.

7. database/sql

The database/sql package provides a generic interface around SQL (or SQL-like) database s. This package includes the ability to connect to a database, execute queries, and retrieve r esults.

Ex:database/sql

```
package main
import (
    "database/sql"
    "fmt"
    _ "github.com/mattn/go-sqlite3"
)
func main() {
    db, err := sql.Open("sqlite3", "file:locked.sqlite?cache=shared")
    if err != nil {
        fmt.Println("Error opening database: ", err)
        return
    }
    defer db.Close()
    result, err := db.Exec("CREATE TABLE IF NOT EXISTS hello (id integer not null primary k
ey, name text)")
    if err != nil {
        fmt.Println("Error creating table: ", err)
        return
    }
    fmt.Println(result)
}
```

```
<Output depends on database state, typically will be nil or an SQL result object>
```

In the above code:We import the database/sql package along with a driver for SQLite (githu b.com/mattn/go-sqlite3), which is a popular lightweight database.We open a database using sql.Open which establishes a connection to our specified database, here a SQLite database. db.Exec is used to execute SQL statements that do not return rows, such as CREATE, INSE RT, UPDATE. In this example, we're creating a table named 'hello'.It’s crucial to handle error s in real applications, to ensure that the database operations do not fail silently.defer db.Clos e() ensures that the database connection is closed when the surrounding function exits, prev enting resource leaks.