แพกเกจหลัก (The Core Packages)

ปกติการเขียนทุกอย่างจากจุดเริ่มต้น ในโลกแห่งความเป็นจริงเราจะต้องพึ่งความสามารถของไลบรารี่ที่มีอยู่แล้ว ในบทนี้จะไปดูแพกเกจที่นิยมใช้บ่อยๆ ที่มีอยู่ในโกกัน

คำเตือน: ไลบรารี่หลายๆ ตัวจะต้องมีความรู้ในเรื่องนั้นๆ เป็นพิเศษ เช่น วิทยาการเข้ารหัส (cryptography) ซึ่งมันเกินขอบเขตของหนังสือเล่มนี้ไปในการที่อธิบายเทคโนโลยีนั้น

สตริง (Strings)

โกได้รวมฟังก์ชั่นที่เอาไว้ใช้จัดการกับสตริงอยู่ในแพกเกจที่ชื่อว่า strings

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(
        // true
        strings.Contains("test", "es"),

        // 2
        strings.Count("test", "t"),

        // true
        strings.HasPrefix("test", "te"),

        // true
        strings.HasSuffix("test", "st"),

        // 1
        strings.Index("test", "e"),

        // "a-b"
        strings.Join([]string{"a","b"}, "-"),

        // == "aaaaa"
        strings.Repeat("a", 5),

        // "bbaa"
        strings.Replace("aaaa", "a", "b", 2),

        // []string{"a","b","c","d","e"}
        strings.Split("a-b-c-d-e", "-"),

        // "test"
        strings.ToLower("TEST"),

        // "TEST"
        strings.ToUpper("test"),

    )
}
```

บางครั้งเราต้องการทำงานกับสตริงด้วยข้อมูลแบบไบนารี่(binary) การที่จะแปลงสตริงไปเป็น []byte สไลซ์ของไบท์(และจาก []byte สไลซ์ของไบท์ไปเป็นสตริง) ทำได้โดย

```
arr := []byte("test")
str := string([]byte{'t','e','s','t'})
```

อินพุต/เอาท์พุต (Input / Output)

ก่อนที่เราจะได้ดูเรื่องไฟล์เราต้องเข้าใจแพกเกจไอโอ(io) ของโกซะก่อน แพกเกจ io มีฟังก์ชั่นให้ใช้ไม่เยอะ แต่ส่วนใหญ่แพกเกจอื่นจะมาใช้อินเทอร์เฟสซะมากกว่า โดยอินเทอร์เฟสหลักๆ จะมีสองตัวคือ Reader กับ Writer โดย Reader จะสนับสนุนการอ่านผ่านการทำงานของเมธอดที่ชื่อว่า Read และ Writer จะสนับสนุนการเขียนผ่านการทำงานของเมธอดที่ชื่อว่า Write มีหลายฟังก์ชั่นในโกที่ใช้ Reader และ Writer เป็นอาร์กิวเมนท์ เช่น ฟังก์ชั่น Copy ที่เอาไว้คัดลอกข้อมูลจาก Reader ไปหา Writer

```
func Copy(dst Writer, src Reader) (written int64, err error)
```

โดยจะอ่านหรือเขียนไปที่ []byte หรือ string คุณสามารถใช้ struct ที่ชื่อว่า Buffer ซึ่งอยู่ในแพกเกจ bytes

```
package bytes
var buf bytes.Buffer
buf.Write([]byte("test"))
```

Buffer ไม่จำเป็นต้องกำหนดค่าเริ่มต้นให้มัน และสนับสนุนทั้งอินเทอร์เฟส Reader และ Writer คุณสามารถแปลงมันไปเป็น []byte โดยเรียก buf.Bytes() ถ้าคุณต้องการที่จะอ่านจาก string คุณสามารถที่จะเรียก strings.NewReader ซึ่งจะมีประสิทธิภาพดีกว่าการใช้ buffer

File & Folders

การจะเปิดไฟล์ในโกก็จะใช้ฟังก์ชั่น Open ในแพกเกจ os ในตัวอย่างนี้จะเป็นวิธีการอ่านเนื้อหาในไฟล์มาแสดงผลที่เทอร์มินัล

```
package main

import (
    "fmt"
    "os"
)
```

```
func main() {
    file, err := os.Open("test.txt")
    if err != nil {
        // handle the error here
        return
    }
    defer file.Close()

    // get the file size
    stat, err := file.Stat()
    if err != nil {
        return
    }
    // read the file
    bs := make([]byte, stat.Size())
    _, err = file.Read(bs)
    if err != nil {
        return
    }

    str := string(bs)
    fmt.Println(str)
}
```

เราใช้ defer file.Close() หลังเปิดไฟล์เพื่อที่จะทำให้มั่นใจได้ว่าจะปิดไฟล์เมื่อฟังก์ชั่นจบการทำงานลง การอ่านไฟล์เป็นอะไรที่ง่ายมากและเราสามารทำให้มันสั้นกว่าได้ด้วยการทำแบบนี้

```
package main

import (
    "fmt"
    "io/ioutil"
)

func main() {
    bs, err := ioutil.ReadFile("test.txt")
    if err != nil {
        return
    }
    str := string(bs)
    fmt.Println(str)
}
```

อันนี้เป็นวิธีการสร้างไฟล์

```
package main

import (
    "os"
)

func main() {
    file, err := os.Create("test.txt")
    if err != nil {
        // handle the error here
        return
    }
    defer file.Close()

    file.WriteString("test")
}
```

การที่จะดึงเนื้อหาของโฟลเดอร์เราจะใช้ฟังก์ชั่น os.Open เหมือนเดิม แต่จะใช้ที่อยู่ของโฟลเดอร์แทนที่อยู่ของไฟล์ และเราจะเรียกเมธอด ReadDir ได้

```
package main

import (
    "fmt"
    "os"
```

```
)

func main() {
    dir, err := os.Open(".")
    if err != nil {
        return
    }
    defer dir.Close()

    fileInfos, err := dir.Readdir(-1)
    if err != nil {
        return
    }
    for _, fi := range fileInfos {
        fmt.Println(fi.Name())
    }
}
```

หากเราต้องการที่จะเข้าไปในโฟลเดอร์ เพื่อที่จะอ่านเนื้อหาใน sub-folder หรือ sub-sub-folder โกมีวิธีการที่ง่ายมาก นั้นคือการใช้ฟังก์ชั่น Walk ซึ่งอยู่ในแพกเกจ path/filepath

```
package main

import (
    "fmt"
    "os"
    "path/filepath"
)

func main() {
    filepath.Walk(".", func(path string, info os.FileInfo, err
error) error {
        fmt.Println(path)
        return nil
    })
}
```

จากตัวอย่างด้านบน ฟังก์ชั่นที่ใส่เข้าไปใน Walk จะถูกเรียกทุกไฟล์และโฟลเดอร์ในรูทโฟลเดอร์

ข้อผิดพลาด (Errors)

โกมีไทป์ที่ถูกสร้างมาแล้วสำหรับข้อผิดพลาดซึ่งเราได้เคยเห็นมาแล้ว (type error) เราสามารถสร้าง error ของตัวเองโดยใช้ฟังก์ชั่น New ในแพกเกจ errors

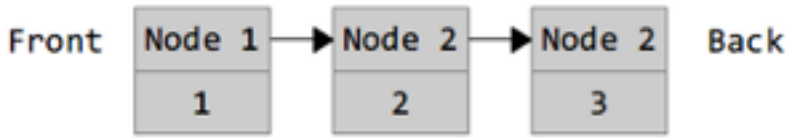
```
package main

import "errors"

func main() {
    err := errors.New("error message")
}
```

Containers & Sort

lists และ maps ของโกมี collections ให้เยอะมากภายใต้แพกเกจ container เราจะดูในแพกเกจ container/list จากตัวอย่าง

List

แพกเกจ container/list ได้อิมพลีเม้นต์ doubly-linked list ซึ่ง linked list คือโครงสร้างข้อมูลที่มีหน้าตาตามรูปด้านล่าง



ในแต่ล่ะโหนดของ list จะมีค่า (1, 2, 3 ในตัวอย่าง) และมีพอยต์เตอร์ชี้ไปหาโหนดถัดไป แต่ doubly-linked list ในแต่ล่ะโหนดจะมีพอยต์เตอร์ที่ชี้กลับไปหาโหนดก่อนหน้าด้วย ในลิสท์นี้สามารถสร้างโดยโปรแกรมจากตัวอย่างด้านล่าง

```
package main

import ("fmt" ; "container/list")

func main() {
    var x list.List
    x.PushBack(1)
    x.PushBack(2)
    x.PushBack(3)
```

```
    for e := x.Front(); e != nil; e=e.Next() {
        fmt.Println(e.Value.(int))
    }
}
```

zero value ของ List คือ list ว่าง (*List สามารถสร้างได้โดย list.New) Values จะถูกต่อเข้าไปในลิสท์ด้วยการใช้ PushBack เราวนรอบแต่ล่ะไอเท็มในลิสท์โดยดึงค่าอิลิเมนท์ตัวแรก และไปเรื่อยๆ จนกระทั่งจะพบว่าค่านั้นเป็น nil

Sort

แพกเกจ sort บรรจุฟังก์ชั่นสำหรับเรียงข้อมูล ซึ่งมีฟังก์ชั่นสำหรับ slices ints และ floats ไว้ให้ก่อนแล้ว ในตัวอย่างนี้เป็นวิธีการเรียงข้อมูลของเราเอง

```
package main

import ("fmt" ; "sort")

type Person struct {
    Name string
    Age int
}

type ByName []Person

func (this ByName) Len() int {
    return len(this)
}
func (this ByName) Less(i, j int) bool {
    return this[i].Name < this[j].Name
}
func (this ByName) Swap(i, j int) {
    this[i], this[j] = this[j], this[i]
}

func main() {
    kids := []Person{
        {"Jill",9},
        {"Jack",10},
    }
    sort.Sort(ByName(kids))
    fmt.Println(kids)
}
```

ฟังก์ชั่น Sort ใน sort ใช้ sort.Interface และเรียงมัน sort.Interface ต้องการ 3 เมธอดคือ Len, Less และ Swap การประกาศ sort ของเราเองโดยสร้าง type ที่ชื่อว่า ByName และสร้างมันเหมือนกับ slice ที่เราต้องการจะเรียง เราก็ประกาศ 3 methods ดังกล่าว

การเรียง list ของ people นั้นง่ายมากด้วยการ casting list ไปเป็น type ใหม่ หากเราต้องการเรียงตามอายุได้โดย

```
type ByAge []Person
func (this ByAge) Len() int {
    return len(this)
}
func (this ByAge) Less(i, j int) bool {
    return this[i].Age < this[j].Age
}
func (this ByAge) Swap(i, j int) {
    this[i], this[j] = this[j], this[i]
}
```

Hashes & Cryptography

ฟังก์ชั่นแฮช(hash) ช่วยทำให้เกิดกลุ่มของข้อมูลและลดขนาดให้เล็กกว่าเดิมโดยมีขนาดที่แน่นอน ซึ่งแฮชจะถูกใช้บ่อยในการเขียนโปรแกรมเพื่อที่จะตรวจสอบว่าข้อมูลมีการเปลี่ยนแปลง ในโกจะแบ่งแฮชออกเป็นสองกลุ่มหลักคือ กลุ่มที่เกี่ยวกับการเข้ารหัส (cryptographic) และกลุ่มที่ไม่เกี่ยวกับการเข้ารหัส (non-cryptographic)

ฟังก์ชั่นแฮชในกลุ่มที่ไม่เกี่ยวกับการเข้ารหัสจะอยู่ในแพกเกจที่ชื่อว่า hash และรวมถึง adler32, crc32, crc64 และ fnv เราจะแสดงให้ดูถึงตัวอย่างการใช้ crc32

```
package main

import (
    "fmt"
    "hash/crc32"
)

func main() {
    h := crc32.NewIEEE()
    h.Write([]byte("test"))
    v := h.Sum32()
    fmt.Println(v)
}
```

แฮชแบบ crc32 จะอิมพลีเม้นต์ interface Writer ทำให้เราสามารถเขียน bytes ลงไปได้เหมือน Writer ตัวอื่นๆ หลังจากที่เราเขียนสิ่งที่เราต้องการลงไป เราก็เรียก Sum32() ซึ่งจะคืน unit32 กลับมา ในการใช้งานทั่วไป crc32 จะเอาไว้ใช้เปรียบเทียบไฟล์สองไฟล์ ถ้า Sum32 มีค่าเหมือนกัน มีโอกาสสูงมาก (แต่ไม่ถึง 100%) ที่จะเป็นไฟล์เดียวกัน ถ้าค่าต่างกันก็แสดงว่าทั้งสองไฟล์นั้นไม่เหมือนกัน

```
package main

import (
    "fmt"
    "hash/crc32"
    "io/ioutil"
)

func getHash(filename string) (uint32, error) {
    bs, err := ioutil.ReadFile(filename)
    if err != nil {
        return 0, err
    }
    h := crc32.NewIEEE()
    h.Write(bs)
    return h.Sum32(), nil
}

func main() {
    h1, err := getHash("test1.txt")
    if err != nil {
        return
    }
    h2, err := getHash("test2.txt")
    if err != nil {
        return
    }
    fmt.Println(h1, h2, h1 == h2)
}
```

ฟังก์ชั่นแฮชในกลุ่มที่เกี่ยวข้องกับการเข้ารหัส จะคล้ายๆ กับกลุ่ม non-cryptographic แต่จะเพิ่มคุณสมบัติที่ทำให้คำนวณย้อนกลับได้ยาก เพื่อทำให้ยากต่อการหาข้อมูลตั้งต้น ฟังก์ชั่นแฮชในกลุ่มนี้เราจะพบเห็นใน security applications

หนึ่งในตัวอย่างที่เรารู้จักกันคือ SHA-1 โดยวิธีการใช้เป็นตามตัวอย่างด้านล่าง

```
package main

import (
    "fmt"
    "crypto/sha1"
)

func main() {
    h := sha1.New()
    h.Write([]byte("test"))
    bs := h.Sum([]byte{})
    fmt.Println(bs)
}
```

จากตัวอย่างจะเห็นว่าคล้ายกับการใช้ crc32 ในตัวอย่างก่อนหน้า เพราะทั้ง crc32 และ sha1 ได้อิมพลีเม้นต์อินเทอร์เฟส hash.Hash ความแตกต่างหลักของทั้งสอบแบบคือ crc32 นั้นจะคำนวณด้วยแฮช 32 bit ส่วน sha1 จะคำนวณด้วยแฮช 160 bit ไม่มี native type อันไหนที่เป็นตัวเลข 160 bit ดังนั้นเราเลยใช้สไลซ์ของ 20 ไบท์แทน

เซิร์ฟเวอร์ (Servers)

การเขียน network server ในโกนั้นง่ายมาก เราจะแสดงให้เห็นว่าเราจะสร้าง TCP server อย่างไร

```
package main

import (
    "encoding/gob"
    "fmt"
    "net"
)

func server() {
    // listen on a port
    ln, err := net.Listen("tcp", ":9999")
    if err != nil {
        fmt.Println(err)
        return
    }
    for {
        // accept a connection
        c, err := ln.Accept()
        if err != nil {
            fmt.Println(err)
            continue
        }
        // handle the connection
        go handleServerConnection(c)
    }
}

func handleServerConnection(c net.Conn) {
    // receive the message
    var msg string
    err := gob.NewDecoder(c).Decode(&msg)
    if err != nil {
        fmt.Println(err)
    } else {
        fmt.Println("Received", msg)
    }

    c.Close()
}
```

```
func client() {
    // connect to the server
    c, err := net.Dial("tcp", "127.0.0.1:9999")
    if err != nil {
        fmt.Println(err)
        return
    }

    // send the message
    msg := "Hello World"
    fmt.Println("Sending", msg)
    err = gob.NewEncoder(c).Encode(msg)
    if err != nil {
        fmt.Println(err)
    }

    c.Close()
}

func main() {
   โกserver()
   โกclient()

    var input string
    fmt.Scanln(&input)
}
```

ในตัวอย่างจะใช้แพกเกจ encoding/gob ซึ่งจะช่วยให้ง่ายในการเข้ารหัสค่าของโกไปที่โกโปรแกรมอื่นๆ (หรือในโกโปรแกรมเดียวกัน) สามารถอ่านได้ การ encoding จะอยู่ในแพกเกจที่อยู่ใต้แพกเกจ encoding เช่น encoding/json หรือใน 3rd party แพกเกจ (เช่นในตัวอย่างเราใช้ labix.org/v2/mgo/bson สำหรับ bson)

HTTP

HTTP server ง่ายมากในการสร้างและใช้งาน:

```
package main

import ("net/http" ; "io")

func hello(res http.ResponseWriter, req *http.Request) {
    res.Header().Set(
        "Content-Type",
        "text/html",
    )
    io.WriteString(
        res,
        `<doctype html>
<html>
    <head>
        <title>Hello World</title>
    </head>
    <body>
        Hello World!
    </body>
</html>`,
    )
}
func main() {
    http.HandleFunc("/hello", hello)
    http.ListenAndServe(":9000", nil)
}
```

HandleFunc จะ handle URL route (/hello) โดยเรียกใช้ฟังก์ชั่นที่ให้มา เราสามารถ handle static files โดยใช้ FileServer

```
http.Handle(
    "/assets/",
    http.StripPrefix(
        "/assets/",
        http.FileServer(http.Dir("assets")),
    ),
)
```

RPC

แพกเกจ net/rpc (remote procedure call) และแพคเกจ net/rpc/jsonrpc จะทำให้เมธอดถูกเรียกใช้งานข้ามเครือข่ายได้ (แทนที่จะเรียกได้เฉพาะโปรแกรมที่รันมันขึ้นมา)

```
package main

import (
    "fmt"
    "net"
    "net/rpc"
)

type Server struct {}
func (this *Server) Negate(i int64, reply *int64) error {
    *reply = -i
    return nil
}

func server() {
    rpc.Register(new(Server))
    ln, err := net.Listen("tcp", ":9999")
    if err != nil {
        fmt.Println(err)
        return
    }
    for {
        c, err := ln.Accept()
        if err != nil {
            continue
        }
        go rpc.ServeConn(c)
    }
}
func client() {
    c, err := rpc.Dial("tcp", "127.0.0.1:9999")
    if err != nil {
        fmt.Println(err)
        return
    }
    var result int64
    err = c.Call("Server.Negate", int64(999), &result)
    if err != nil {
        fmt.Println(err)
    } else {
```

```
        fmt.Println("Server.Negate(999) =", result)
    }
}
func main() {
   โกserver()
   โกclient()

    var input string
    fmt.Scanln(&input)
}
```

ในโปรแกรมนี้จะคล้ายกับตัวอย่างใน TCP ยกเว้นเราสร้างวัตถุที่มีทุกเมธอดที่ต้องการเปิดเผยและเราเรียกเมธอด Negate จาก client ดูในเอกสารของ net/rpc สำหรับข้อมูลเพิ่มเติม

Parsing Command Line Arguments

เมื่อเราต้องการเรียกคำสั่งบนเทอร์มินัลโดยการ pass อาร์กิวเมนท์ของคำสั่งนั้น เราจะเห็นได้ในคำสั่ง go:

```
go run myfile.go
```

run และ myfile.go คือ อาร์กิวเมนท์ เราสามารถส่ง flags ไปที่คำสั่งได้โดย

```
go run -v myfile.go
```

แพกเกจ flag ช่วยให้เราสามารถ parse อาร์กิวเมนท์ และ flags ที่จะส่งไปให้โปรแกรมของได้ ในตัวอย่างจะเป็นโปรแกรมที่จะสร้างตัวเลขระหว่าง 0 ถึง 6 เราสามารถเปลี่ยนค่าสูงสุดได้โดยส่ง flag -max=100 ไปที่โปรแกรม

```
package main

import ("fmt";"flag";"math/rand")

func main() {
    // Define flags
    maxp := flag.Int("max", 6, "the max value")
    // Parse
    flag.Parse()
    // Generate a number between 0 and max
    fmt.Println(rand.Intn(*maxp))
}
```

อะไรก็ตามที่ไม่ใช่ flag อาร์กิวเมนท์สามารถที่จะเอามาได้โดย flag.Args() มันจะคืน []string ออกมา

Synchronization Primitives

เราได้บอกวิธีทางที่จะจัดการการทำงานในเวลาเดียวกัน และ synchronization ในโกซึ่งอยู่ในบทที่ 10 ไปแล้ว อย่างไรก็ตามโกยังยอมให้ใช้รูปแบบเดิมๆ ในการจัดการ multithreading routines ซึ่งอยู่ในแพกเกจ sync และ sync/atomic

Mutexes

mutex จะ locks ส่วนนึงในโค้ดไปที่ single thread ในขณะนั้น และใช้ป้องกันการแย่ง shared resource จาก non-atomic operations นี่เป็นตัวอย่างของ mutex

```
package main

import (
    "fmt"
    "sync"
    "time"
)
func main() {
    m := new(sync.Mutex)

    for i := 0; i < 10; i++ {
       โกfunc(i int) {
            m.Lock()
            fmt.Println(i, "start")
            time.Sleep(time.Second)
            fmt.Println(i, "end")
            m.Unlock()
        }(i)
    }

    var input string
    fmt.Scanln(&input)
}
```

เมื่อ mutex (ตัวแปร m ในตัวอย่างด้านบน) ได้ lock มันจะบล็อคการทำงานจนกว่าเราจะสั่ง unlock จะต้องระวังมากเมื่อใช้ mutex หรือ synchronization ที่มีให้ในแพกเกจ sync/atomic

โดยทั่วไป multithreaded programming มันยาก ซึ่งสามารถสร้างความผิดพลาดได้ง่ายและยากที่จะหามัน หนึ่งในจุดแข็งของโกคือความสามารถของ concurrency ที่ง่ายต่อการเข้าใจได้และนำไปใช้งานได้อย่างเหมาะสมกว่า thread และ locks