Distributed Systems

Go tutorial

Dixie State University—Computing and Design

Spring 2022

Finding duplicate lines

```
// Dup1 prints "count: line" for duplicated lines.
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    counts := make(map[string]int)
    input := bufio.NewScanner(os.Stdin)
    for input.Scan() {
        counts[input.Text()]++
    }
    // clipped: check input.Err() for errors
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
```

Printf

The most common printf verbs:

```
%d          decimal integer
%x, %o, %b  integer in hexadecimal, octal, binary
%f, %g, %e  floating-point number: 3.141593 3.141592653589793 3.141593e+00
%t          boolean: true or false
%c          rune (Unicode code point)
%s          string
%q          quoted string "abs" or rune 'c'
%v          any value in a natural format
%T          type of any value
%%          literal percent sign (no operand)
```

Finding duplicate lines part 2

```go
func main() {
    counts := make(map[string]int)
    files := os.Args[1:]
    if len(files) == 0 {
        countLines(os.Stdin, counts)
    } else {
        for _, arg := range files {
            f, err := os.Open(arg)
            if err != nil {
                fmt.Fprintf(os.Stderr, "dup2: %v\n", err)
                continue
            }
            countLines(f, counts)
            f.Close()
        }
    }
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
```

```go
func countLines(f *os.File, counts map[string]int) {
    input := bufio.NewScanner(f)
    for input.Scan() {
        counts[input.Text()]++
    }
    // ignoring potential errors from input.Err()
}

// this version "streams" its input, processing each
// file as it reads it
```

Finding duplicate lines part 3

```
// package and imports skipped... added import "io/ioutil"

func main() {
    counts := make(map[string]int)
    for _, filename := range os.Args[1:] {
        data, err := ioutil.ReadFile(filename)
        if err != nil {
            fmt.Fprintf(os.Stderr, "dup3: %v\n", err)
            continue
        }
        for _, line := range strings.Split(string(data), "\n") {
            counts[line]++
        }
    }
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
```

Fetch a URL

```
// note: new package "net/http" imported

func main() {
    for _, url := range os.Args[1:] {
        resp, err := http.Get(url)
        if err != nil {
            fmt.Fprintf(os.Stderr, "fetch: %v\n", err)
            os.Exit(1)
        }
        b, err := ioutil.ReadAll(resp.Body)
        resp.Body.Close()
        if err != nil {
            fmt.Fprintf(os.Stderr, "fetch: reading %s: %v\n", url, err)
            os.Exit(1)
        }
        fmt.Printf("%s", b)
    }
}
```

Fetching URLs concurrently

```go
package main

import (
    "fmt"
    "io"
    "io/ioutil"
    "net/http"
    "os"
    "time"
)
func main() {
    start := time.Now()
    ch := make(chan string)
    for _, url := range os.Args[1:] {
        go fetch(url, ch) // start a goroutine
    }
    for range os.Args[1:] {
        fmt.Println(<-ch) // receive from chan ch
    }
    fmt.Printf("%.2fs elapsed\n",
        time.Since(start).Seconds())
}
```

```go
func fetch(url string, ch chan<- string) {
    start := time.Now()
    resp, err := http.Get(url)
    if err != nil {
        ch <- fmt.Sprint(err) // send to channel ch
        return
    }

    nbytes, err :=
        io.Copy(ioutil.Discard, resp.Body)
    resp.Body.Close() // don't leak resources
    if err != nil {
        ch <- fmt.Sprintf("while reading %s: %v",
                  url, err)
        return
    }
    secs := time.Since(start).Seconds()
    ch <- fmt.Sprintf("%.2fs  %7d  %s",
                  secs, nbytes, url)
}
```

Fetch URLs example

```
$ go build
$ ./fetchall https://golang.org http://gopl.io https://godoc.org
0.14s     6852  https://godoc.org
0.16s     7261  https://golang.org
0.48s     2475  http://gopl.io
0.48s elapsed
```

Switch

The `switch` statement is a multi-way branch:

```
switch coinflip() {
case "heads":
    heads++
case "tails":
    tails++
default:
    fmt.Println("landed on edge!")
}
```

Notes:

- Cases are evaluated top to bottom, first match is taken
- Optional `default` case is taken if no other branch matches
- Cases do **not** fall through. If you want that, use `fallthrough`

Tagless switch

Another form of `switch`, called a *tagless switch*, is a tidier way to express chains of `if...else` statements

```
func Signum(x int) int {
    switch {
    case x > 0:
        return +1
    default:
        return 0
    case x < 0:
        return -1
    }
}
```

Names

Go has the following reserved words:

```
break       default     func        interface       select
case        defer       go          map             struct
chan        else        goto        package         switch
const       fallthrough if          range           type
continue    for         import      return          var
```

And the following predefined names:

```
Constants:      true false iota nil
Types:          int int8 int16 int32 int64
                uint uint8 uint16 uint32 uint64 uintptr
                float32 float64 complex128 complex64
                bool byte rune string error
Functions:      make len cap new append copy close delete
                complex real imag
                panic recover
```

Strings

A string is an immutable sequence of bytes, typically interpreted as UTF-8-encoded Unicode code points (runes).

The built-in `len` function returns the number of bytes (not runes) and the *index* operation retrieves the *i*-th byte of a string:

```
s := "hello, world"
fmt.Println(len(s))      // "12"
fmt.Println(s[0], s[7]) // "104 119" ('h' and 'w')

c := s[len(s)]           // panic: index out of range
```

You can get a substring using slice notation:

```
fmt.Println(s[0:5])      // "hello"
fmt.Println(s[:5])       // "hello"
fmt.Println(s[7:])       // "world"
fmt.Println(s[:])        // "hello, world"
```

Strings

The + operator makes a new string by concatenating two strings:

```
fmt.Println("goodbye" + s[5:])  // "goodbye, world"
```

Strings can be compared with operations like == and $<$. The comparison is done byte by byte.

Strings are immutable. The byte sequence can never be changed. However, a string variable can be reassigned:

```
s := "left foot"
t := s
t += ", right foot"
```

Immutability means that it is safe for two strings to share underlying memory, making it cheap to copy strings and create substrings with shared memory.

String literals

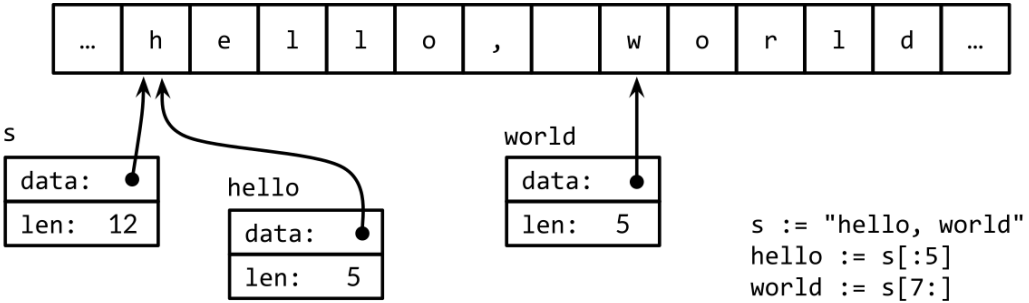

Figure 1: Memory layout of a string with substrings

UTF-8

UTF-8 is a variable-length encoding for text created by Ken Thompson and Rob Pike.

```
0xxxxxxx                            runes 0-127     (ASCII)
110xxxxx 10xxxxxx                   128-2047        (values <128 unused)
1110xxxx 10xxxxxx 10xxxxxx          2048-65535      (values <2048 unused)
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx 65536-0x10fff   (other values unused)
```

Nice properties:

- Zero is zero
- The first byte tells you how many bytes in the rune
- Can look at any byte and tell if it is a middle byte
- ASCII is still ASCII

Use `unicode/utf8` package to decode and work with runes. Or use a `for` loop with `range`:

```
s := "Hello, world™"

for i, r := range s {
    fmt.Println("index", i, "rune", r)
}
```