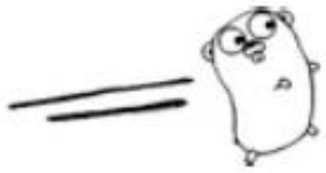
Lecture 4: Golang Programming

Lecturer: Prof. Zichen Xu

Outline

- The history
- The reason
- The code



Sophistication

If more than one function is selected, any function template specializations in the set are eliminated if the set also contains a non-template function, and any given function template specialization F1 is eliminated if the set contains a second function template specialization whose function template is more specialized than the function template of F1 according to the partial ordering rules of 14.5.6.2. After such eliminations, if any, there shall remain exactly one selected function.

(C++0x, §13.4 [4])

Sophistication

- Which Boost templated pointer type should I use?

- linked_ptr
- scoped_ptr
- shared_ptr
- smart_ptr
- weak_ptr
- intrusive_ptr
- exception_ptr

Noise

```
public static <I, O> ListenableFuture<O> chain(ListenableFuture<I>
input, Function<? super I, ? extends ListenableFuture<? extends O>>
function)
```

- a recently observed chat status

```
foo::Foo *myFoo = new foo::Foo(foo::FOO_INIT)
```

- but in the original Foo was a longer word

How did we get here?

1) C and Unix became dominant in research.

2) The desire for a higher-level language led to C++, which grafted the Simula style of object-oriented programming onto C. It was a poor fit but since it
compiled to C it brought high-level programming to Unix.

3) C++ became the language of choice in parts of industry and in many research universities.

4) Java arose as a clearer, stripped-down C++.

5) By the late 1990s, a teaching language was needed that
seemed relevant, and Java was chosen.

Programming becomes too hard

- These languages are hard to use.
- They are subtle, intricate, and verbose.
- Their standard model is oversold, and we respond with add-on models such as "patterns".

(Norvig: patterns are a demonstration of weakness in a language.)

- Yet these languages are successful and vital

Reaction

- The inherent clumsiness of the main languages has caused a reaction.
- A number of successful simpler languages (Python, Ruby, Lua, JavaScript, Erlang, ...) have become popular, in part as a rejection of the standard languages.
- Some beautiful and rigorous languages designed by domain experts (Scala, Haskell, ...) have also arisen, although they are not as widely adopted.
- So despite the standard model, other approaches are popular and there are signs of a growth in "outsider“ languages, a renaissance of language invention

Confusion

- The standard languages (Java, C++) are statically typed.
- Most outsider languages (Ruby, Python, JavaScript) are interpreted and dynamically typed.
- Perhaps as a result, non-expert programmers have confused "ease of use" with interpretation and dynamic typing.
- This confusion arose because of how we got here: grafting an orthodoxy onto a language that couldn't support it cleanly.

The Good

- The standard languages are very strong: type-safe, effective, efficient.
- In the hands of experts, they are great.
- Huge systems and huge companies are built on them.
- In practice they work well for large scale programming: big programs, many programmers.

The bad

- The standard languages are hard to use.
- Compilers are slow and fussy. Binaries are huge.
- Effective work needs language-aware tools, distributed compilation farms, ...
- Many programmers prefer to avoid them.
- The languages are at least 10 years old and poorly adapted to the current computing environment: clouds of networked multicore CPUs.

And Python

- This is partly why Python et al. have become so popular:

- They don't have much of the "bad".
  - dynamically typed (fewer noisy keystrokes)
  - interpreted (no compiler to wait for)
  - good tools (interpreters make things easier)
- But they also don't have the "good":
  - slow
  - not type-safe (static errors occur at runtime)
  - very poor at scale

- And they're also not very modern.

A niche

- There is a niche to be filled: a language that has the good, avoids the bad, and is suitable to modern computing infrastructure:

  - comprehensible
  - statically typed
  - light on the page
  - fast to work in
  - scales well
  - doesn't require tools, but supports them well
  - good at networking and multiprocessing

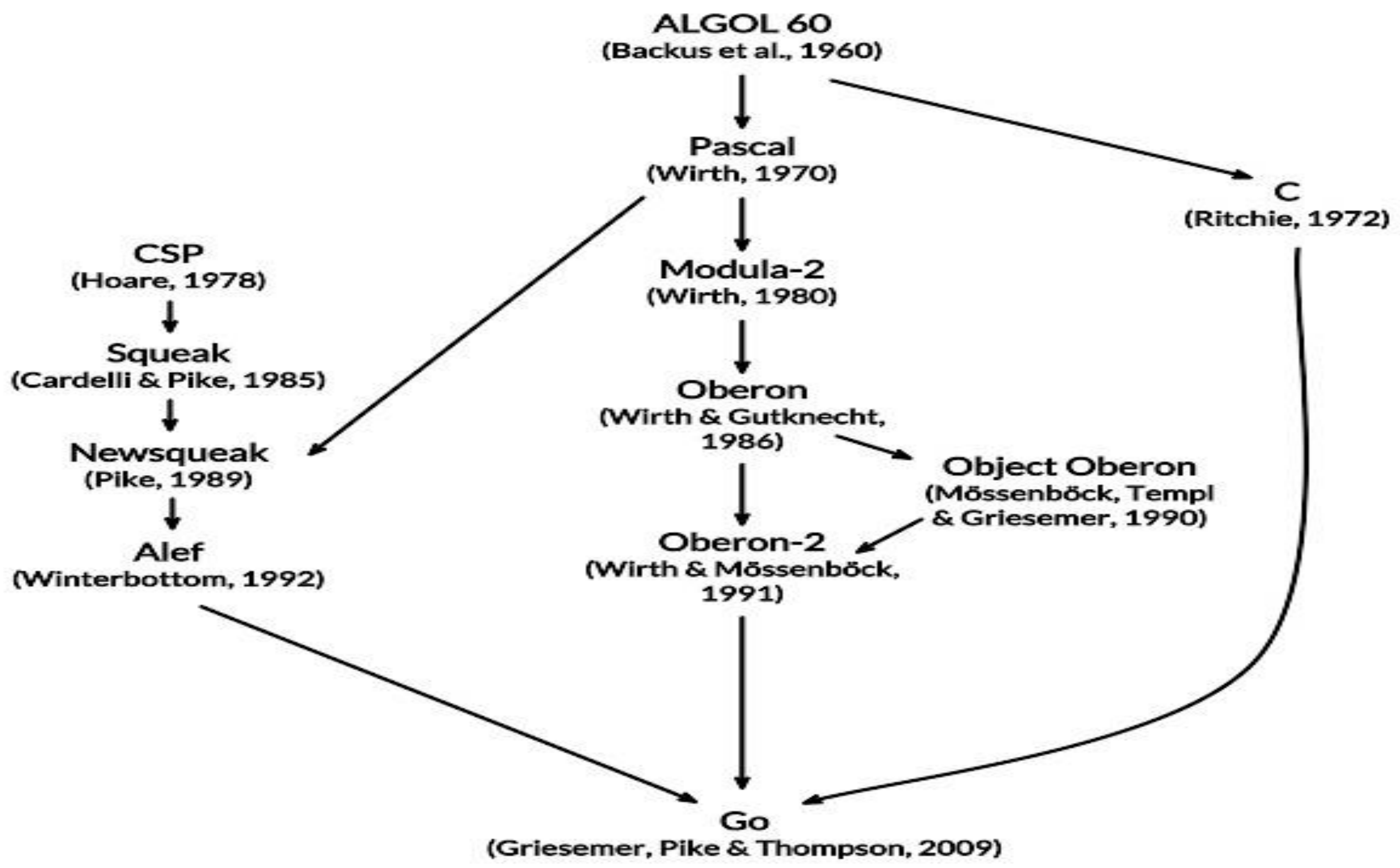
Go

- developed ~2007 at Google by

  Robert Griesemer, Rob Pike, Ken Thompson
- open source

- compiled, statically typed
  - very fast compilation
- C-like syntax
- garbage collection
- built-in concurrency
- no classes or type inheritance or overloading or generics
  - unusual interface mechanism instead of inheritance

Influences


ALGOL 60
(Backus et al., 1960)
Pascal
(Wirth, 1970)
C
(Ritchie, 1972)
CSP
(Hoare, 1978)
Modula-2
(Wirth, 1980)
Squeak
(Cardelli & Pike, 1985)
Oberon
(Wirth & Gutknecht, 1986)
Newsqueak
(Pike, 1989)
Object Oberon
(Mössenböck, Templ & Griesemer, 1990)
Alef
(Winterbottom, 1992)
Oberon-2
(Wirth & Mössenböck, 1991)
Go
(Griesemer, Pike & Thompson, 2009)

Negative influences

"When the three of us got started, it was pure research. The three of us got together and decided that we hated C++. We started off with the idea that all three of us had to be talked into every feature in the language, so there was no extraneous garbage put into the language for any reason." (Ken Thompson)

Hello world in Go

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

```
$ go run hello.go    # to compile and run
$ go build hello.go  # to create a binary

$ go help            # for more
```

Hello, world 2.0

```
package main
import (
	"fmt"
	"http"
)
func handler(c *http.Conn, r *http.Request) {
	fmt.Fprintf(c, "Hello, %s.", r.URL.Path[1:])
}
func main() {
	http.ListenAndServe(":8080",
	http.HandlerFunc(handler))
}
```

What does Go do?

- Fast compilation
- Expressive type system
- Concurrency
- Garbage collection
- Systems programming capabilities
- Clarity and orthogonality

Go’s Performance

- New clean compiler worth ~5X compared to gcc.
- We want a millionX for large programs, so we need to fix the dependency problem.
- In Go, programs compile into packages and each compiled package file imports transitive dependency info.
- If A.go depends on B.go depends on C.go:
  - compile C.go, B.go, then A.go.
  - to compile A.go, compiler reads B.o but not C.o.
- At scale, this can be a huge speedup.

Trim the tree

- Large C++ programs (Firefox, OpenOffice, Chromium) have huge build times. On a Mac (OS X 10.5.7, gcc 4.0.1):

- C: #include <stdio.h>
  - reads 360 lines from 9 files

- C++: #include <iostream>
  - reads 25,326 lines from 131 files

- Objective-C: #include <Cocoa/Cocoa.h>
  - reads 112,047 lines from 689 files

But we haven't done any real work yet!

- In Go, import "fmt" reads one file:
  - 195 lines summarizing 6 dependent packages.
- As we scale, the improvement becomes exponential.

Go is different

- Go is object-oriented not type-oriented
  - inheritance is not primary
  - methods on any type, but no classes or subclasses
- Go is (mostly) implicit not explicit
  - types are inferred not declared
  - objects have interfaces but they are derived, not specified
- Go is concurrent not parallel
  - intended for program structure, not max performance
  - but still can keep all the cores humming nicely
  - ... and many programs are more nicely expressed with concurrent ideas even if not parallel at all

Types, constants, variables

- basic types

  `bool string  int8 int16 int32 int64  uint8 … int uint`

  `float32 float64 complex64 complex128`

  quotes: `‘世’, “UTF-8 string”, ` `` `raw string` ``

- variables

```
var c1, c2 rune
var x, y, z = 0, 1.23, false   // variable decls

x := 0; y := 1.23; z := false  // short variable decl
```

  Go infers the type from the type of the initializer

  assignment between items of different type requires an explicit conversion, e.g., int(float_expression)

- operators
  - mostly like C, but **`++`** and **`--`** are postfix only and not expressions
  - assignment is not an expression
  - no **`?:`** operator
    -

Echo command:

```
// Echo prints its command-line arguments.
package main

import (
  "fmt"
  "os"
)

func main() {
      var s, sep string
      for i := 1; i < len(os.Args); i++ {
           s += sep + os.Args[i]
           sep = " "
      }
      fmt.Println(s)
}
```

Echo command (version 2):

```
// Echo prints its command-line arguments.
package main

import (
  "fmt"
  "os"
)

func main() {
   s, sep := "", ""
   for _, arg := range os.Args[1:] {
      s += sep + arg
      sep = " "
   }
   fmt.Println(s)
}
```

Arrays and slices

- an array is a fixed-length sequence of same-type items

```
months := [...]string {1:"Jan", 2:"Feb",
/*...,*/ 12:"Dec"}
```

- a slice is a subsequence of an array

```
summer := months[6:9]; Q2 := months[4:7]
```

- elements accessed as `slice[index]`
  - indices from 0 to **`len(slice)-1`** inclusive

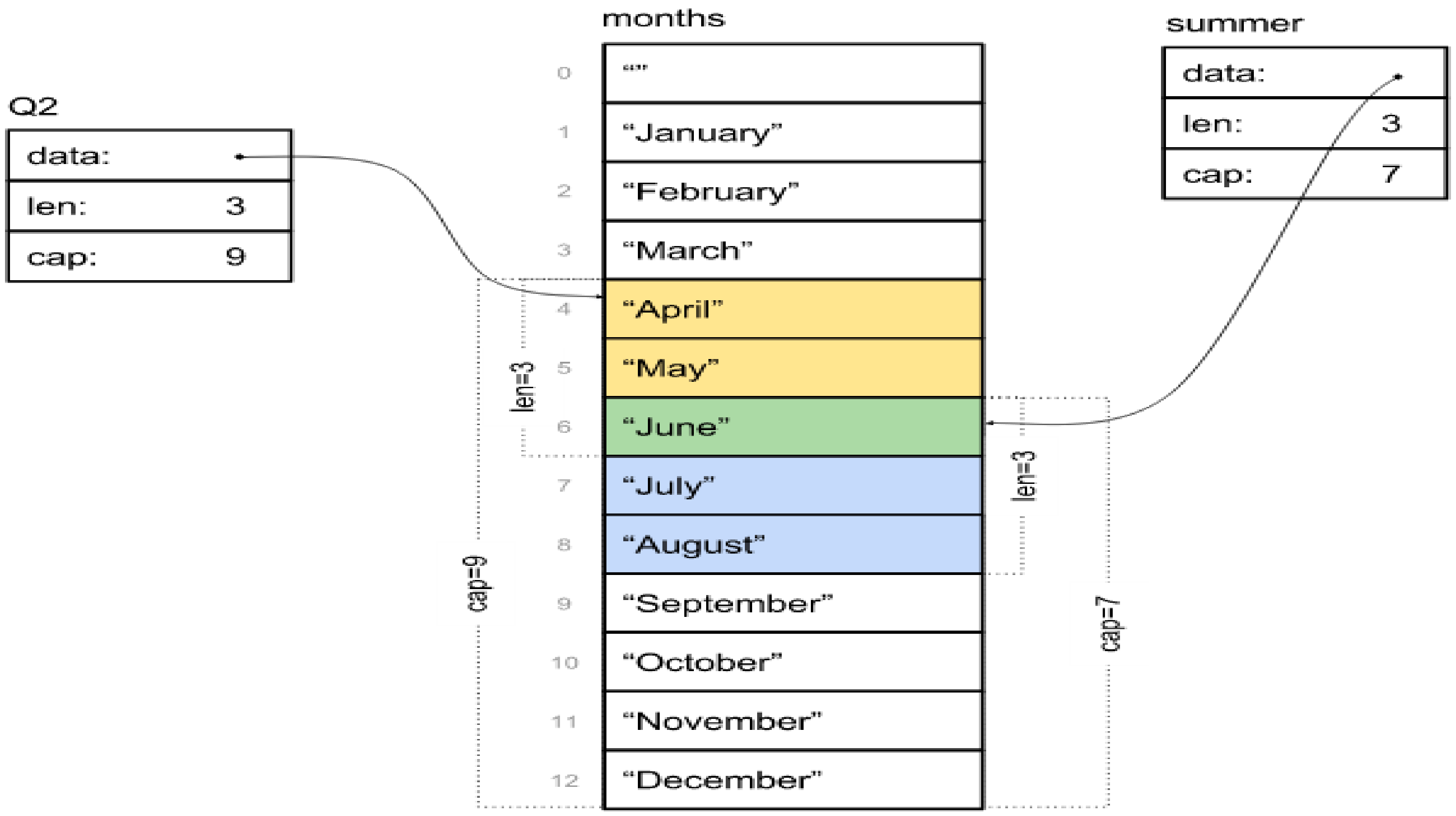
  `summer[0:3]` is elements **`months[6:9]`**

```
summer[0] = "Jun"
```

- loop over a slice with for range

```
for i, v := range summer {
  fmt.Println(i, v)
}
```

- slices are very efficient (represented as small structures)
- most library functions work on slices


months
0 “”
1 “January”
2 “February”
3 “March”
4 “April”
5 “May”
6 “June”
7 “July”
8 “August”
9 “September”
10 “October”
11 “November”
12 “December”
Q2
data:
len: 3
cap: 9
summer
data:
len: 3
cap: 7
len=3
cap=9
len=3
cap=7

Maps (== associative arrays)

- unordered collection of key-value pairs
  - keys are any type that supports **==** and **!=** operators
  - values are any type

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

Methods and pointers

- can define methods that work on any type, including your own:

```
type Vertex struct {
    X, Y float64
}
func (v *Vertex) Scale(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}
func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
func main() {
    v := &Vertex{3, 4}
    v.Scale(5)
    fmt.Println(v, v.Abs())
}
```

Interfaces

- an interface is satisfied by any type that implements all the methods of the interface
- completely abstract: can't instantiate one
- can have a variable with an interface type
- then assign to it a value of any type that has the methods the interface requires

- a type implements an interface merely by defining the required methods
  - it doesn't declare that it implements them

- Writer: the most common interface

```
type Writer interface {
   Write(p []byte) (n int, err error)
}
```

Sort interface

- sort interface defines three methods
- any type that implements those three methods can sort
- algorithms are inside the soft package, invisible outside

```
package sort

type Interface interface {
    Len() int
    Less(i, j int) bool
    Swap(i, j int)
}
```

Sort interface (adapted from Go Tour)

```
type Person struct {
  Name string
  Age  int
}
func (p Person) String() string {
  return fmt.Sprintf("%s: %d", p.Name, p.Age)
}
type ByAge []Person

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

func main() {
  people := []Person{{"Bob",31}, {"Sue",42}, {"Ed",17}, {"Jen",26},}
  fmt.Println(people)
  sort.Sort(ByAge(people))
  fmt.Println(people)
}
```

Tiny version of curl

```
func main() {
 url := os.Args[1]
 resp, err := http.Get(url)
 if err != nil {
     fmt.Fprintf(os.Stderr, "curl: %v\n", err)
     os.Exit(1)
 }
 _, err = io.Copy(os.Stdout, resp.Body)
 if err != nil {
     fmt.Fprintf(os.Stderr, "curl: copying %s:
 %v\n",
                                     url, err)
     os.Exit(1)
 }
}
```

Tiny web server

```
func main() {
 http.HandleFunc("/", handler)
 http.ListenAndServe("localhost:8000", nil)
}

// handler echoes Path component of the request URL r.
func handler(w http.ResponseWriter, r *http.Request) {
 fmt.Fprintf(w, "URL.Path = %q\n", r.URL.Path)
}
```

- `http.ResponseWriter` implements Writer interface

Concurrency: goroutines & channels

- channel: a type-safe generalization of Unix pipes
  - inspired by Hoare's Communicating Sequential Processes (1978)

- goroutine: a function executing concurrently with other goroutines in the same address space
  - run multiple parallel computations simultaneously
  - loosely like threads but much lighter weight

- channels coordinate computations by explicit communication
  - locks, semaphores, mutexes, etc., are much less often used

Example: web crawler

- want to crawl a bunch of web pages to do something
  - e.g., figure out how big they are
- problem: network communication takes relatively long time
  - program does nothing useful while waiting for a response
- solution: access pages in parallel
  - send requests asynchronously
  - display results as they arrive
  - needs some kind of threading or other parallel process mechanism
- takes less time than doing them sequentially

Practical Use Cases

- The Really Good: Network and Web Servers
  - Network applications live and die by concurrency
  - Go's native concurrency features: (goroutines and channels)
  - Networking, distributed functions, or services (APIs, Web Servers, minimal frameworks for Web Apps)
  - Items are either a part of language (goroutines for threadlike behavior) or available in the standard library http package.
    - ”Batteries included" philosophy like Python
- Stand-alone command-line apps or scripts
  - Stand-alone executables with no external dependencies
    - Versus Python which requires a copy of the interpreter on the target machine
  - Can talk to underlying system, external C libraries, native system calls

➢*Caution*: Desktop/GUI-based apps & System-Level Programming

Go source materials

- official web site:

  golang.org

- Go tutorial, playground

- Rob Pike on why it is the way it is:

  http://www.youtube.com/watch?v=rKnDgT73v8s

- Russ Cox on interfaces, reflection, concurrency

  http://research.swtch.com/gotour

Conclusion

- We have started on Golang programming
- We have shared the history of a relatively new programming language
- We have discussed the syntax of Golang programming
- We will start on System Functionality next week

Version 1: no parallelism

```
func main() {
  start := time.Now()
  for _, site := range os.Args[1:] {
    count("http://" + site)
  }
  fmt.Printf("%.2fs total\n", time.Since(start).Seconds())
}

func count(url string) {
  start := time.Now()
  r, err := http.Get(url)
  if err != nil {
    fmt.Printf("%s: %s\n", url, err)
    return
  }
  n, _ := io.Copy(ioutil.Discard, r.Body)
  r.Body.Close()
  dt := time.Since(start).Seconds()
  fmt.Printf("%s %d [%.2fs]\n", url, n, dt)
}
```

Version 2: parallelism with goroutines

```
func main() {
  start := time.Now()
  c := make(chan string)
  n := 0
  for _, site := range os.Args[1:] {
    n++
    go count("http://" + site, c)
  }
  for i := 0; i < n; i++ {
    fmt.Print(<-c)
  }
  fmt.Printf("%.2fs total\n", time.Since(start).Seconds())
}
func count(url string, c chan<- string) {
  start := time.Now()
  r, err := http.Get(url)
  if err != nil {
    c <- fmt.Sprintf("%s: %s\n", url, err)
    return
  }
  n, _ := io.Copy(ioutil.Discard, r.Body)
  r.Body.Close()
  dt := time.Since(start).Seconds()
  c <- fmt.Sprintf("%s %d [%.2fs]\n", url, n, dt)
}
```

Version 2: main() for parallelism with goroutines

```
func main() {
  start := time.Now()
  c := make(chan string)
  n := 0
  for _, site := range os.Args[1:] {
    n++
    go count("http://" + site, c)
  }
  for i := 0; i < n; i++ {
    fmt.Print(<-c)
  }
  fmt.Printf("%.2fs total\n",
 time.Since(start).Seconds())
}
```

Version 2: count() for parallelism with goroutines

```
func count(url string, c chan<- string) {
  start := time.Now()
  r, err := http.Get(url)
  if err != nil {
    c <- fmt.Sprintf("%s: %s\n", url, err)
    return
  }
  n, _ := io.Copy(ioutil.Discard, r.Body)
  r.Body.Close()
  dt := time.Since(start).Seconds()
  c <- fmt.Sprintf("%s %d [%.2fs]\n", url,
 n, dt)
}
```