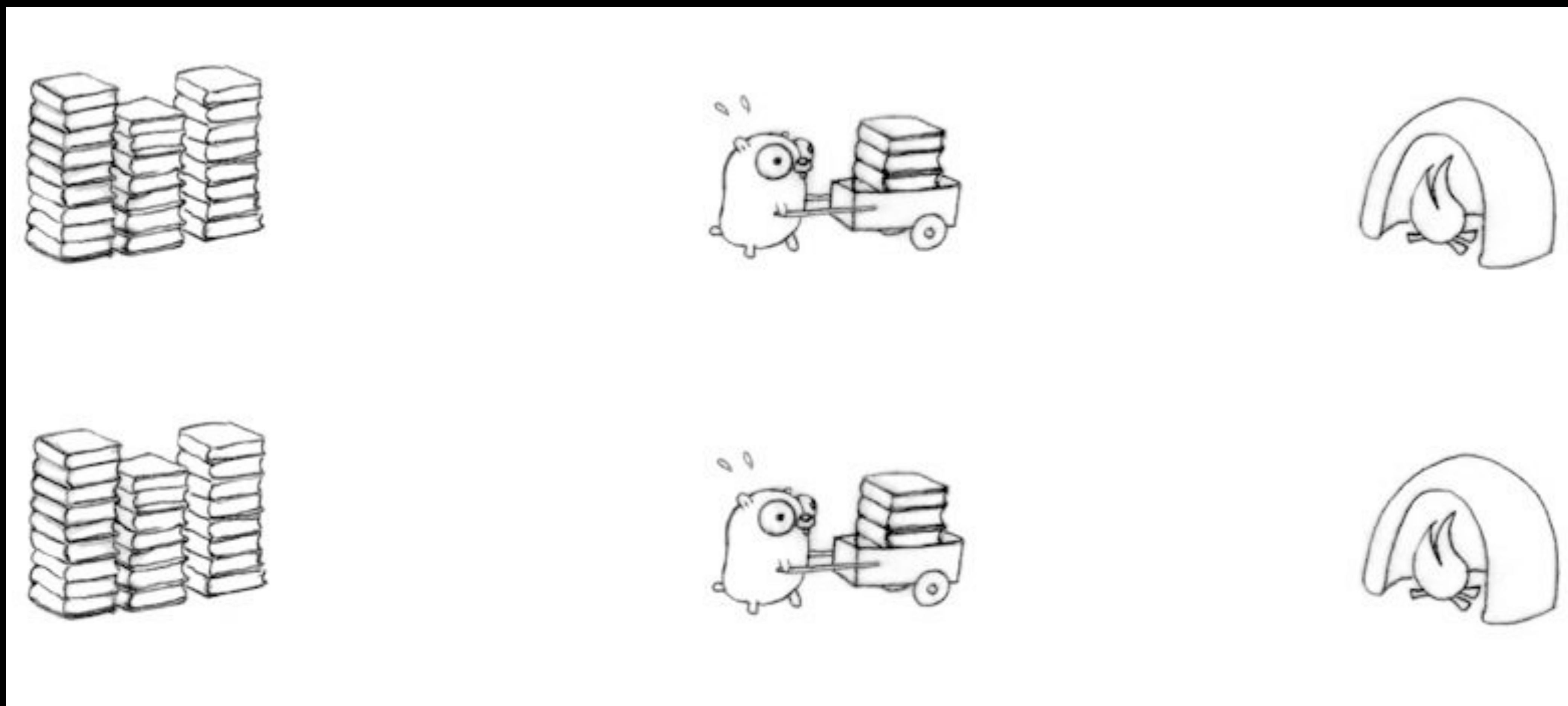
A Very Short Introduction to

# Concurrent Programming in Go

工程效率部门 马波

2015.3



Go is burning C books concurrently

# Concurrency is NOT Parallelism


一个老师：

讲课、板书、擦黑板、照顾学生

concurrent but not parallel


一个老师

讲课、板书

一个助教

擦黑板、照顾学生

both concurrent and parallel

所有学生

一起做手工

parallel but not concurrent

# Concurrency is NOT Parallelism

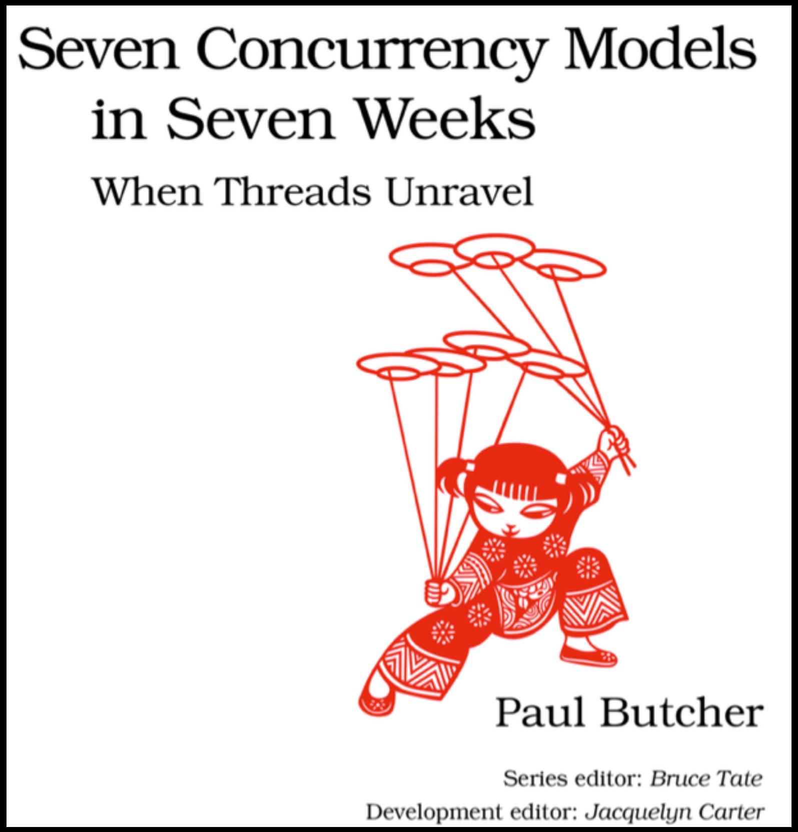
Structure Execution

OS: concurrent but not necessary parallel

Concurrency: construct program with independent pieces & coordination

Concurrency requires communication

Concurrent Model




Thread and Locks

Functional Parallelism

Software Transactional Memory (STM)

Actor

Communicating Sequential Process (CSP)

Data Parallelism

Lambda Architecture

# Tony Horne 1978
## *Communicating Sequential Process* (CSP)

Let $x$ be an event and let $P$ be a process. Then

$(x \rightarrow P)$ (pronounced "$x$ then $P$")

describes an object which first engages in the event $x$ and then behaves exactly as described by $P$. The process $(x \rightarrow P)$ is defined to have the same alphabet as $P$, so this notation must not be used unless $x$ is in that alphabet; more formally,

$\alpha(x \rightarrow P) = \alpha P$ provided $x \in \alpha P$

Examples

X1 A simple vending machine which consumes one coin before breaking

$(coin \rightarrow STOP_{\alpha VMS})$

□

X2 A simple vending machine that successfully serves two customers before breaking

$(coin \rightarrow (choc \rightarrow (coin \rightarrow (choc \rightarrow STOP_{\alpha VMS}))))$

进程之间只能通过 一对通信原语实现协作：

Q->x表示从进程Q输入一个值到变量x中

P<-e表示把表达式e的值发送给进程P

当P进程执行Q->x， 同时Q进程执行P<-e时，发生通信，

e的值从Q进程传送给P进程的变量x

## Go’s Concurrency Principle

Do not communicate by sharing memory;
instead, share memory by communicating

shared values are passed around on *channels*
and, in fact, never actively shared by
separate threads of execution.

Only one *goroutine* has access to
the value at any given time

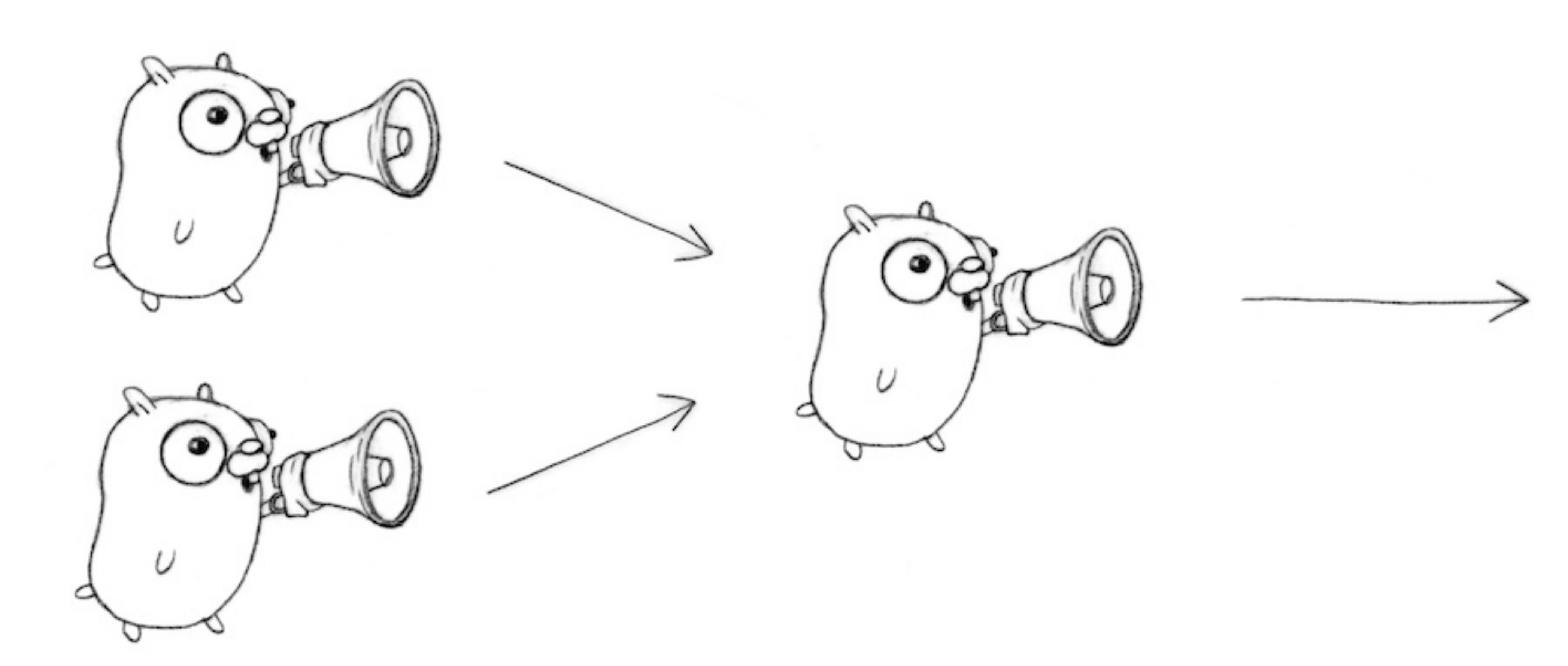
```go
package main

import (
  "fmt"
  "time"
)

func say(s string) {
  for i := 0; i < 5; i++ {
    time.Sleep(100 * time.Millisecond)
    fmt.Println(s)
  }
}

func main() {
  go say("world")
  say("hello")
}
```

A *goroutine* is a lightweight thread managed by the Go runtime.

it is a function executing concurrently with other goroutines in the same address space.

Goroutines are multiplexed onto multiple OS threads so if one should block, such as while waiting for I/O, others continue to run.

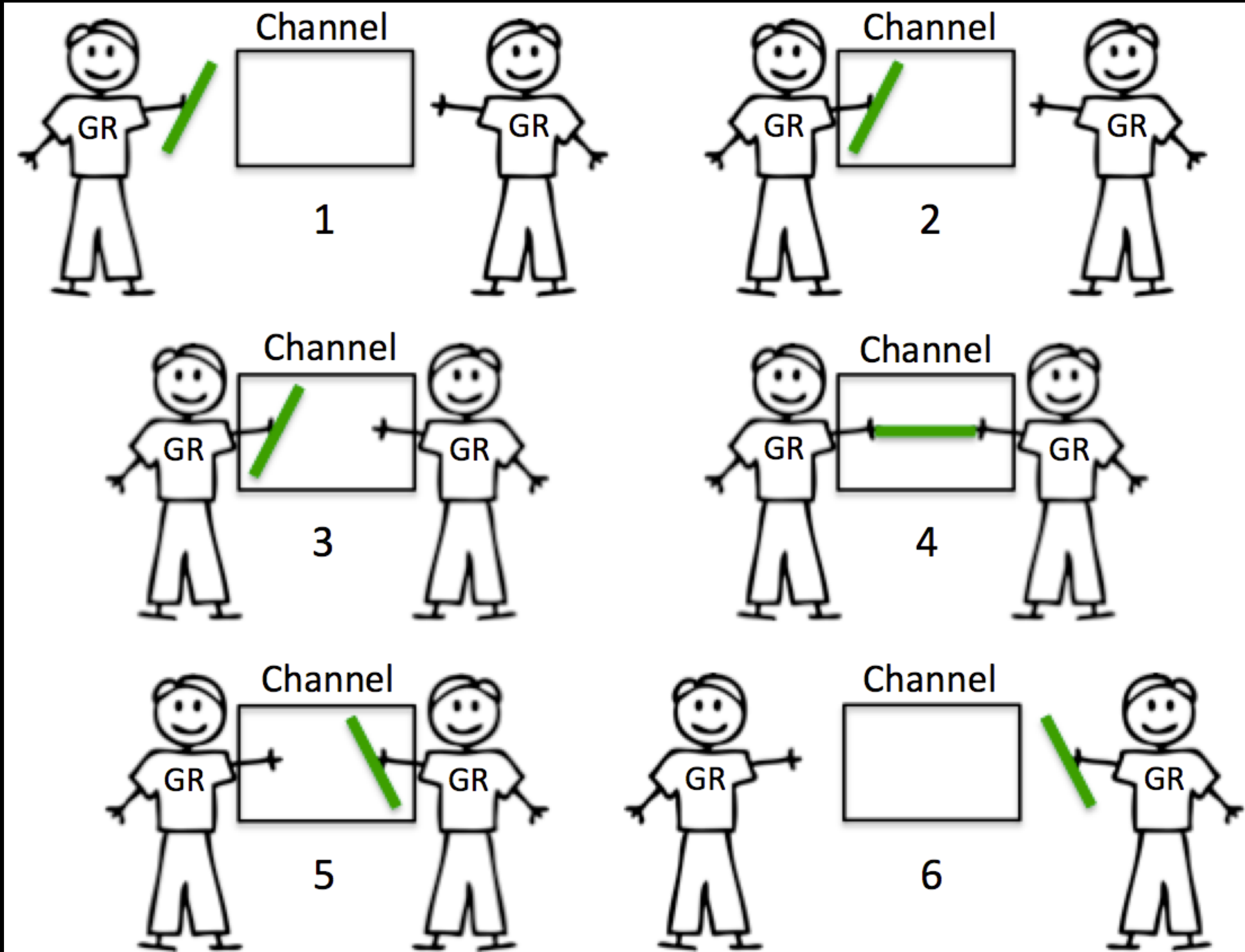
```go
package main

import "fmt"

func sum(a []int, c chan int) {
  sum := 0
  for _, v := range a {
    sum += v
  }
  c <- sum // send sum to c
}

func main() {
  a := []int{7, 2, 8, -9, 4, 0}

  c := make(chan int)
  go sum(a[:len(a)/2], c)
  go sum(a[len(a)/2:], c)
  x, y := <-c, <-c // receive from c

  fmt.Println(x, y, x+y)
}
```



How to Wait for All Goroutines to Finish Executing Before Continuing

```go
package main

import (
  "fmt"
  "sync"
  "net/http"
)

var wg sync.WaitGroup
var urls = []string{
  "http://www.baidu.com/",
  "http://www.weibo.com/",
  "http://www.163.com/",
}

func main() {
  for _, url := range urls {
    wg.Add(1)  // Increment the WaitGroup counter.
    go func(url string) {
      defer wg.Done()
      http.Get(url)
      fmt.Printf("Visited %s\n", url)
    }(url)
  }

  wg.Wait()
}
```

```go
package main

import (
  "fmt"
  "sync"
  "time"
)

func f(wg *sync.WaitGroup, val string) {
  time.Sleep(3 * time.Second)
  fmt.Printf("Finished: %v - %v\n", val, time.Now())
  wg.Done()
}

func main() {
  var wg sync.WaitGroup
  wg.Add(3)  // wait for 3 calls to 'done' on this wait group
  go f(&wg, "goroutine A")

  go func(wg *sync.WaitGroup, val string) {
    time.Sleep(3 * time.Second)
    fmt.Printf("Finished: %v - %v\n", val, time.Now())
    wg.Done()
  }(&wg, "goroutine B")

  go f(&wg, "goroutine C")
  wg.Wait()

  fmt.Printf("Finished all goroutines: %v\n", time.Now())
}
```


Channel
GR
GR
1
Channel
GR
GR
2
Channel
GR
GR
3
Channel
GR
GR
4
Channel
GR
GR
5
Channel
GR
GR
6

# Channel

unbuffered
buffered
dropping
sliding

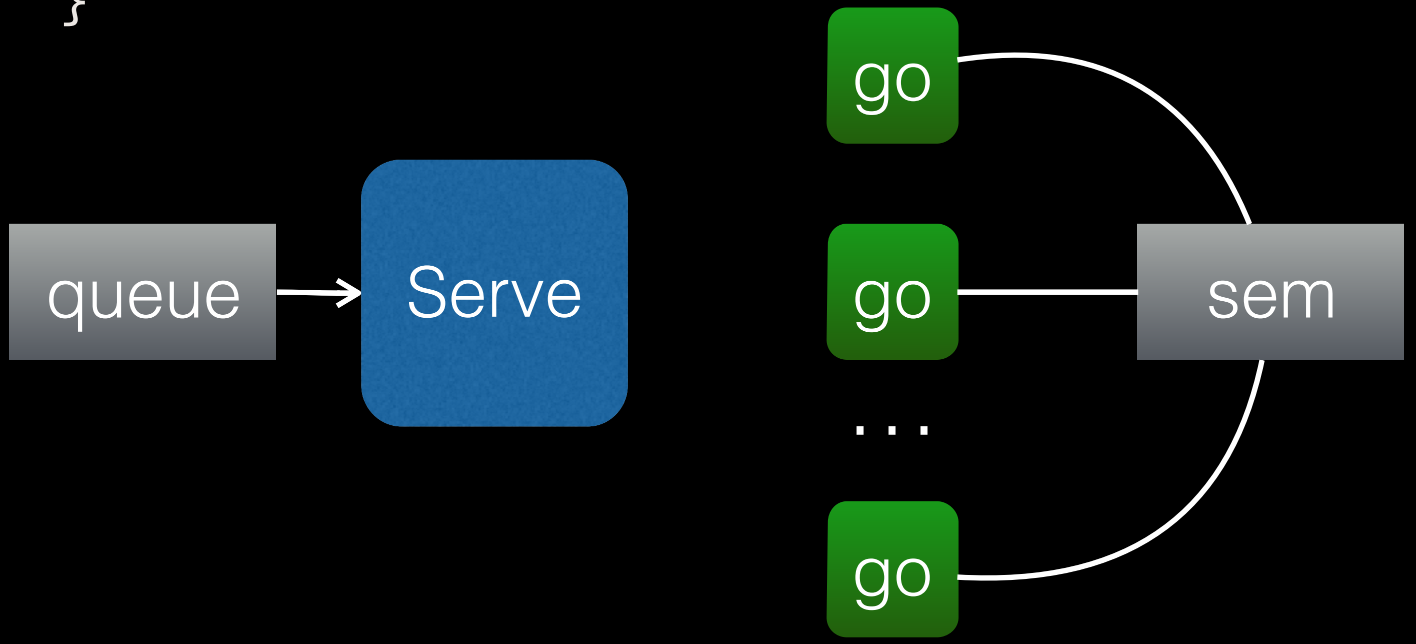
```
ci := make(chan int)              // unbuffered channel of integers
cj := make(chan int, 0)           // unbuffered channel of integers
cs := make(chan *os.File, 100)  // buffered channel of pointers to Files

; Dropping channel
(def dc (chan (dropping-buffer 5)))
(onto-chan dc (range 0 10))
(<!! (async/into [] dc))
; output: [0 1 2 3 4]

; Sliding channel
(def sc (chan (sliding-buffer 5)))
(onto-chan sc (range 0 10))
(<!! (async/into [] sc))
; output: [5 6 7 8 9]
```

```go
package main

import "fmt"

func sum(a []int, c chan int) {
  sum := 0
  for _, v := range a {
    sum += v
  }
  c <- sum // send sum to c
}

func main() {
  a := []int{7, 2, 8, -9, 4, 0}

  c := make(chan int)
  go sum(a[:len(a)/2], c)
  go sum(a[len(a)/2:], c)
  x, y := <-c, <-c // receive from c

  fmt.Println(x, y, x+y)
}
```

```go
package main

import (
	"fmt"
	"sync"
	"time"
)

func main() {
	messages := make(chan int)
	var wg sync.WaitGroup

	wg.Add(3)
	go func() {
		defer wg.Done()
		time.Sleep(time.Second * 3)
		messages <- 1
	}()
	go func() {
		defer wg.Done()
		time.Sleep(time.Second * 2)
		messages <- 2
	}()
	go func() {
		defer wg.Done()
		time.Sleep(time.Second * 1)
		messages <- 3
	}()
	go func() {
		for i := range messages {
			fmt.Println(i)
		}
	}()

	wg.Wait()
}
```

# channel as semaphore

```
var sem = make(chan int, MaxOutstanding)

func Serve(queue chan *Request) {
  for req := range queue {
    sem <- 1
    go func(req *Request) {
      process(req)
      <-sem
    }(req)
  }
}
```



```go
type Request struct {
  args        []int
  f           func([]int) int
  resultChan  chan int
}

// Client
func sum(a []int) (s int) {
  for _, v := range a {
      s += v
  }
  return
}

request := &Request{
  []int{3, 4, 5},
  sum,
  make(chan int)
}

// Send request
clientRequests <- request

// Wait for response.
fmt.Printf("answer: %d\n", <-request.resultChan)
```

channel is a first-class value that can be allocated and passed around like any other.

A common use of this property is to implement safe, parallel demultiplexing.

```go
// Server
func handle(queue chan *Request) {
  for req := range queue {
      req.resultChan <- req.f(req.args)
  }
}
```

concurrent composition

# Concurrency Patterns

# Generator: function that returns a channel

```
// Function returning a channel
c := boring("boring!")
for i := 0; i < 5; i++ {
  fmt.Printf("You say: %q\n", <-c)
}
fmt.Println("You're boring; I'm leaving")


func boring(msg String) <- chan string {
  c := make(chan string)
  go func() {
    for i := 0; ; i++ {
      c <- fmt.Sprintf("%s %d", msg i)
      time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
  }()
  return c
}
```

# Channels as a handle on a service

```go
// channel as a handle on a service
func main() {
  joe := boring("Joe")
  ann := boring("Ann")
  for i :=0; i < 5; i++ {
    fmt.Println(<-joe)
    fmt.Println(<-ann)
  }
  fmt.Println("You're both boring: I'm leaving")
}
```

# Multiplexing

```go
// Multiplexing
func fanIn(input1, input2 <- chan string) <-chan string {
  c := make(chan string)
  go func() { for { c <- <-input1 } }()
  go func() { for { c <- <-input2 } }()
  return c
}

func main() {
  c := fanIn(boring("Joe"), boring("Ann"))
  for i := 0; i < 10; i++ {
    fmt.Println(<-c)
  }
  fmt.Println("You're both boring; I'm leaving")
}
```

```go
// Multiplexing 完整代码
package main

import (
  "fmt"
  "time"
  "math/rand"
)

func boring(msg string) <- chan string {
  c := make(chan string)
  go func() {
    for i := 0; ; i++ {
      c <- fmt.Sprintf("Boring msg: %s %d", msg, i)
      time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
  }()
  return c
}

func fanIn(input1, input2 <- chan string) <-chan string {
  c := make(chan string)
  go func() { for { c <- <-input1 } }()
  go func() { for { c <- <-input2 } }()
  return c
}

func main() {
  c := fanIn(boring("Joe"), boring("Ann"))
  for i := 0; i < 10; i++ {
    fmt.Println(<-c)
  }
  fmt.Println("You're both boring; I'm leaving")
}
```

# Select

```go
// Select
select {
case v1 := <-c1:
  fmt.Printf("received %v from c1\n", v1)
case v2 := <-c2:
  fmt.Printf("received %v from c2\n", v2)
case v3 <- 23:
  fmt.Printf("received %v from c3\n", 23)
default:
  fmt.Printf("no one was ready to communicate\n")
}
```

# Fan-in use Select

```go
// Fan-in use select
func fanIn(input1, input2 <-chan string) <-chan string {
  c := make(chan string)
  go func() {
    for {
      select {
      case s := <-input1: c <- s
      case s := <-input2: c <- s
      }
    }
  }()
  return c
}
```

# Timeout using Select

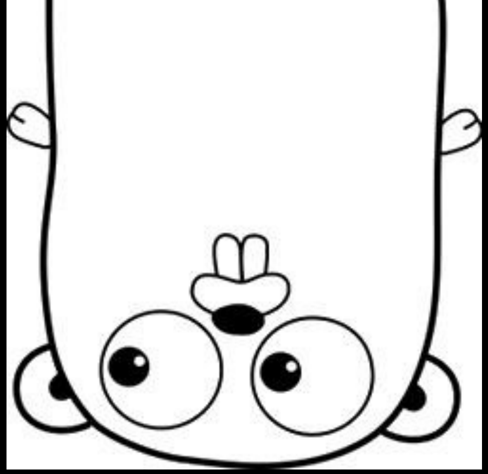
```go
// timeout using select
func main() {
  c := boring("Joe")
  for {
    select {
    case s := <-c:
      fmt.Println(s)
    case <- time.After(1 * time.Second):
      fmt.Println("You're too slow.")
      return
    }
  }
}
```

# Timeout for whole conversation using Select

```go
// timeout for whole conversation
func main() {
  c := boring("Joe")
  timeout := time.After(5 * time.Second)
  for {
    select {
    case s := <-c:
      fmt.Println(s)
    case <- timeout:
      fmt.Println("You talk too much.")
      return
    }
  }
}
```

# Receive on quite channel

```go
// receive on quit channel
quit := make(chan string)
c := boring("Joe", quit)
for i := rand.Intn(10); i >=0; i-- {
  fmt.Println(<-c)
}
quit <- "Bye!"
fmt.Printf("Joe says: %q\n", <- quit)

// sever
select {
case c <- fmt.Sprintf("%s: %d", msg, i):
  // do nothing
case <- quit:
  cleanup()
  quit <- "See you!"
  return
}
```



No locks, no conditional variables, no callbacks.

# A Very Short Introduction to
# Concurrent Programming in Go

工程效率部门 马波

2015.3

*~end~*