Go coding in go way

GopherChina 2017
15 April 2017

Tony Bai
Neusoft

人类语言与思维：萨丕尔-沃夫假说

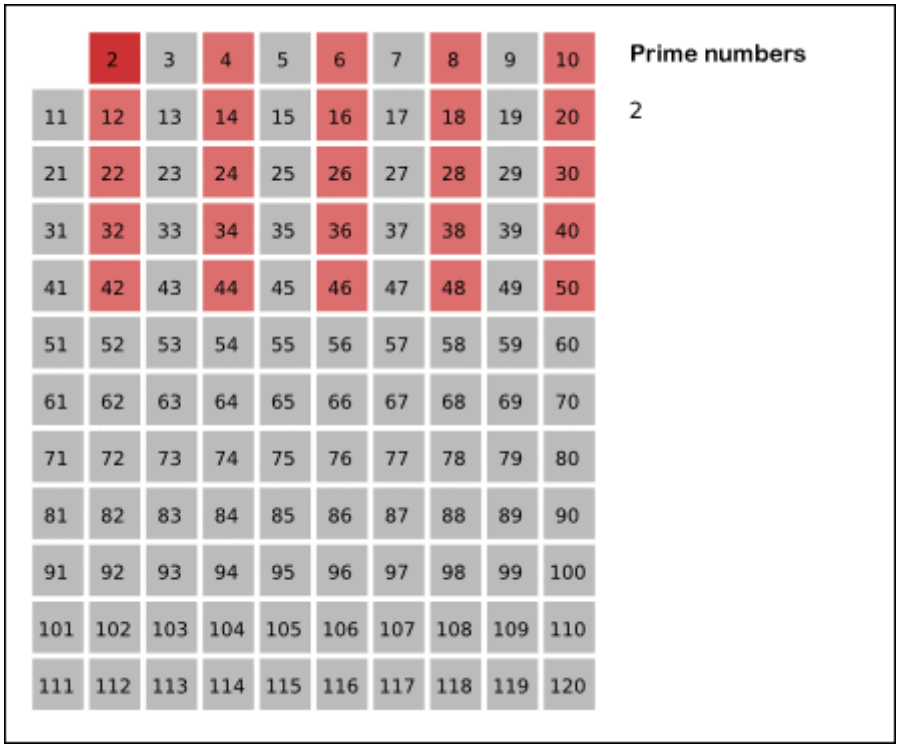
"语言影响/决定思维" - 萨丕尔-沃夫假说

("Language influences/determines thought" - Sapir-Whorf hypothesis)

编程语言与思维：图灵奖得主的认知

"A language that doesn't affect the way you think about programming is not worth knowing." - Alan J. Perlis(艾伦·佩利)，首届图灵奖得主

问题: 素数筛



素数是一个自然数，它具有两个截然不同的自然数除数：1和它本身。 要找到小于或等于给定整数n的素数，我们可以使用Eratosthenes' sieve(埃拉托斯特尼)算法(https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes)。

算法描述：先用最小的素数2去筛，把2的倍数剔除掉；下一个未筛除的数就是素数(这里是3)。再用这个素数3去筛，筛除掉3的倍数... 这样不断重复下去，直到筛完为止。

sieve.c //基于数组的内存操作

```
void sieve() {
    int c, i,j,numbers[LIMIT], primes[PRIMES];

    for (i=0;i<LIMIT;i++){
        numbers[i]=i+2; /*fill the array with natural numbers*/
    }

    for (i=0;i<LIMIT;i++){
        if (numbers[i]!=-1){
            for (j=2*numbers[i]-2;j<LIMIT;j+=numbers[i])
                numbers[j]=-1; /*sieve the non-primes*/
        }
    }

    c = j = 0;
    for (i=0;i<LIMIT&&j<PRIMES;i++) {
        if (numbers[i]!=-1) {
            primes[j++] = numbers[i]; /*transfer the primes to their own array*/
            c++;
        }
    }

    for (i=0;i<c;i++) printf("%d\n",primes[i]);
}
```

sieve.hs //函数递归

```
sieve [] = []
sieve (x:xs) = x : sieve (filter (\a -> not $ a `mod` x == 0) xs)

n = 100
main = print $ sieve [2..n]
```

[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97]

sieve.go //并发组合 (borrowed from Rob Pike's slide)

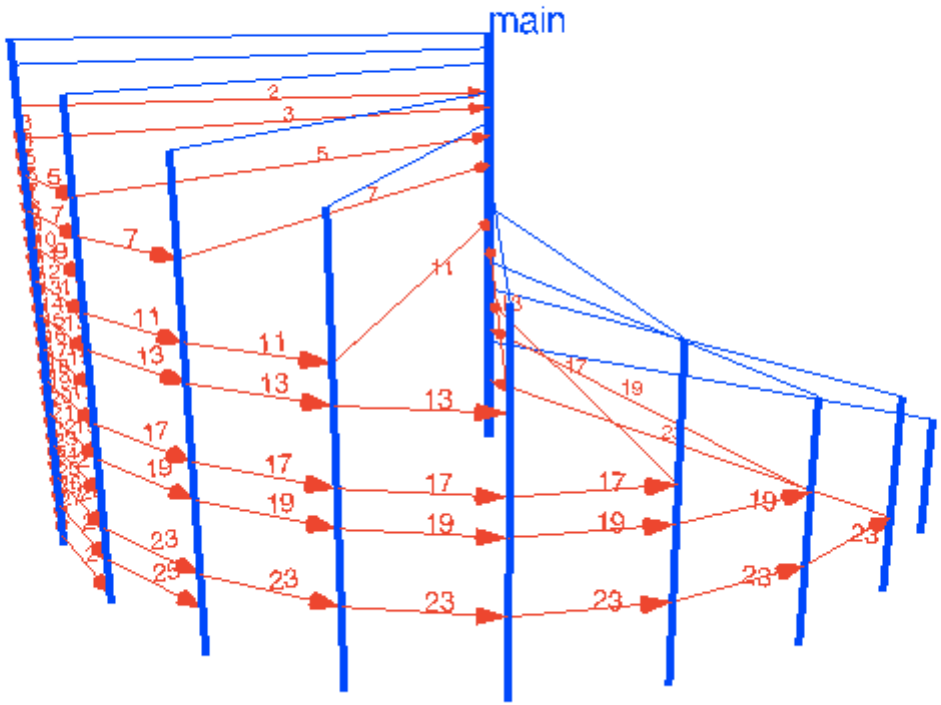
```
func generate(ch chan<- int) {
    for i := 2; ; i++ {
        ch <- i // Send 'i' to channel 'ch'.
    }
}

func filter(src <-chan int, dst chan<- int, prime int) {
    for i := range src { // Loop over values received from 'src'.
        if i%prime != 0 {
            dst <- i // Send 'i' to channel 'dst'.
        }
    }
}

func sieve() {
    ch := make(chan int) // Create a new channel.
    go generate(ch)      // Start generate() as a subprocess.
    for {
        prime := <-ch
        fmt.Print(prime, "\n")
        ch1 := make(chan int)
        go filter(ch, ch1, prime)
        ch = ch1
    }
}
```

sieve.go (cont.)

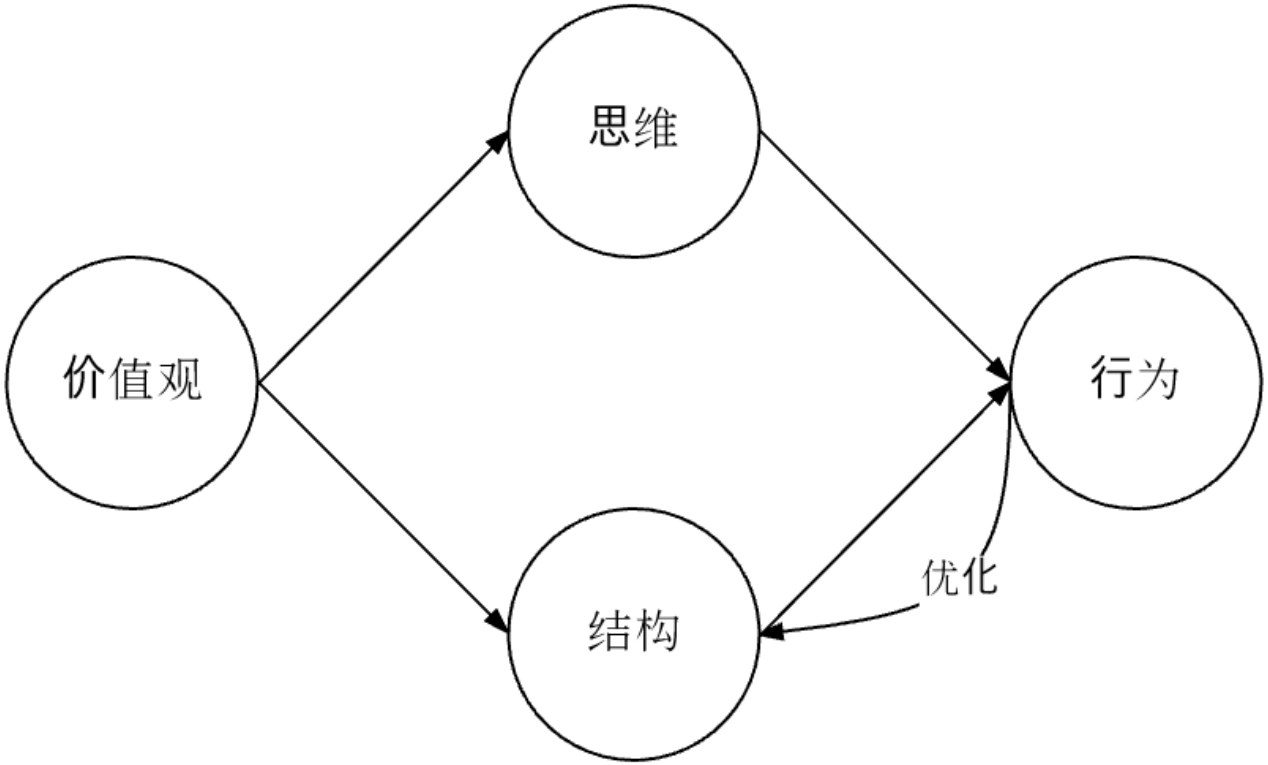
- From divan's blog (http://divan.github.io/posts/go_concurrency_visualize/)



思考

- 面对同一个问题，来自不同编程语言的程序员给出了思维方式截然不同的解决方法
- 一定程度上印证了前面的假说：编程语言影响编程思维
- 避免Go coding in c way/in java way/in python way...
- 目标：**Go coding in go way**

编程语言思维的形成



- 根本：价值观决定思维和语言结构
- 核心：思维和语言结构影响语言的应用行为
- 反馈：语言的应用行为反过来持续影响/优化语言结构

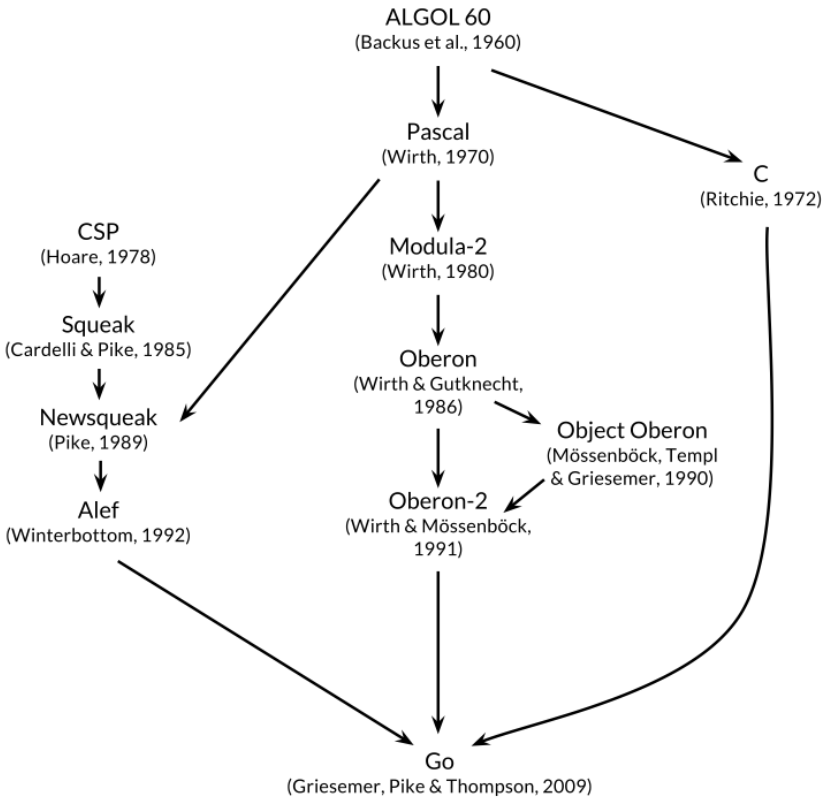
Go语言价值观

- Go编程思维的形成深受Go语言价值观的影响
- Go语言价值观的形成和内容
- Go语言价值观主导下的Go编程思维

Go语言价值观形成 - 语言设计者&Unix文化(主导)

(从左到右) Robert Griesemer, Rob Pike和Ken Thompson

Go语言价值观形成 - Go语言的遗传基因



The task of the programming language designer **" is consolidation not innovation "**. (Tony Hoare, 1973, Go并发模型最初CSP的提出者).

Go语言价值观形成 - 面向新的基础设施环境和大规模软件开发的诸多问题(初衷)

新的基础设施环境:

- 大规模云计算数据中心
- 多核以及多处理器硬件体系

大规模软件开发的难题(Google内部）：

- slow builds
- uncontrolled dependencies
- each programmer using a different subset of the language
- poor program understanding (documentation, etc.)
- duplication of effort
- cost of updates
- cross-language builds

Go语言的价值观

- Overall Simplicity
- Orthogonal Composition
- Preference in Concurrency

一句话概括Go的价值观：**" orthogonal composition of simple concepts with preference in concurrency "**.

Go语言的价值观(cont.)

- 价值观
- (价值观下的)语言设计
- 编程思维
- 该编程思维在语言设计、标准库实现以及主流Go开源项目中的应用体现

Overall Simplicity

语言设计

"Simplicity & elegance are unpopular because they require hard work & discipline to achieve & education to be appreciated." - 图灵奖获得者Dijkstra(迪杰斯特拉)

"Overall Simplicity"价值观更多地体现在了Go语言设计层面：

- clean and regular(mostly) syntax
- only 25 keywords
- one way to write a piece of code and minimize programmer's effort
- garbage collection
- goroutines
- constants
- interfaces
- packages

short naming thought

- 短小
- 一致
- 在上下文环境中用最短的名字携带足够的信息

```
java    vs. go

"index" vs. "i"
"value" vs. "v"
```

变量命名统计

```
cat $(find $GOROOT -name '*.go') | indents | sort | uniq -c | sort -nr | sed 30q
```

```
60224 v
42444 err
38012 t
33386 x
33302 i
33277 b
27943 p
25121 s
21228 n
20831 r
20634 _
19236 c
17056 y
12850 f
12834 a
... ...
```

常见的变量短命名含义

```
[v, k, i]

// loop varible
for i, v := range s {
for k, v := range m {
for v := range r { // channel

// if、switch/case clause varible
if v := mimeTypes[ext]; v != "" {
switch v := ptr.Elem(); v.Kind() {
case v := <-c:

v := reflect.ValueOf(x) // result of reflect.Value()

[t]

t := time.Now() // time
t := &Timer{ // timer
if t := md.typemap[off]; t != nil { // type

[b]

b := make([]byte, n) // bytes slice
b := new(bytes.Buffer) // bytes.Buffer
```

minimal thought

- "一种"代码写法，或提供最少的写法、更简单的写法

```
Go is not a “TMTOWTDI—There’s More Than One Way To Do It”
```

- 显而易见的代码，而不是"聪明"的代码
- minimize programmer's effort

```
Least concern about coding style:
   - Gofmt's style is no one's favorite, yet gofmt is everyone's favorite.
```

"一种"循环: for

- 常规

```
for i := 0; i < count; i++ {}
```

- "while"

```
for condition { }
```

- "do-while"

```
for { // use "for-break" instead
      doSomething()
      if condition { break }
}
```

- iterator loop

```
for k, v := range f.Value {}
```

- dead loop

```
for {}
```

"一种"constants

- constants只是数字，无需显式赋予类型

```
const incomingQueueLength = 25

const (
    http2minMaxFrameSize = 1 << 14
    http2maxFrameSize    = 1<<24 - 1
)

const PI = 3.1415928
const e = 1E6
```

- 无需算术转换

```
const a = 10080
var c int32 = 99
d := c + a
fmt.Printf("%T\n", d) //int32
fmt.Printf("%T\n", a) //int
```

"一种"错误处理方法

There are only two kinds of language: the ones people complain about, and the ones nobody uses. - Bjarne Stroustrup

- 基于比较的错误处理, 没有针对exception的"try-catch"的控制结构
- 让使用者更聚焦于错误本身，显式地处理每一个error，让你对代码更加自信
- 不增加语言复杂性：错误值和其他类型的值地位是一样的，错误处理代码也是普通代码，并无特殊之处。

```
"We believe that coupling exceptions to a control structure,
    as in the try-catch-finally idiom, results in convoluted code.(go faq)"
```

错误处理模式

使用error类型作为错误码返回类型，而不是其他类型（比如int）

- 最常见的

```
callee:
  return errors.New("something error")

  or

  return fmt.Errorf("something error: %s", "error reason")

caller:
  if err != nil {... }
```

- 导出的error变量

```
// io/io.go
var ErrShortWrite = errors.New("short write")
var ErrShortBuffer = errors.New("short buffer")

if err := doSomeIO(); err == io.ErrShortWrite { ... }
```

错误处理模式(cont.)

- 定义新的错误类型实现定制化错误上下文

```
// encoding/json/decode.go
type UnmarshalTypeError struct {
    Value  string       // description of JSON value - "bool", "array", "number -5"
    Type   reflect.Type // type of Go value it could not be assigned to
    Offset int64        // error occurred after reading Offset bytes
    Struct string       // name of the struct type containing the field
    Field  string       // name of the field holding the Go value
}

func (e *UnmarshalTypeError) Error() string {
    ... ...
    return "json: cannot unmarshal " + e.Value + " into Go value of type " + e.Type.String()
}

if serr, ok := err.(*UnmarshalTypeError); ok {
   //use serr to access context in UnmarshalTypeError
   ... ...
}
```

错误处理模式(cont.)

- 将某个包中的Error Type归类，统一提取出一些公共行为特征
- 将这些公共行为特征(behaviour)放入一个公开的interface中

以stdlib的net package为例：

```
//net/net.go
type Error interface {
    error
    Timeout() bool   // Is the error a timeout?
    Temporary() bool // Is the error temporary?
}
```

net/http/server.go中的使用举例：

```
rw, e := l.Accept()
if e != nil {
    if ne, ok := e.(net.Error); ok && ne.Temporary() {
       ... ..
    }
    ... ...
}
```

错误处理模式(cont.)

- 一个实现了net.Error的Error type的实现

```
//net/net.go
type OpError struct {
    ... ...
    // Err is the error that occurred during the operation.
    Err error
}

type temporary interface {
    Temporary() bool
}

func (e *OpError) Temporary() bool {
  if ne, ok := e.Err.(*os.SyscallError); ok {
      t, ok := ne.Err.(temporary)
      return ok && t.Temporary()
  }
  t, ok := e.Err.(temporary)
  return ok && t.Temporary()
}
```

错误处理模式(cont.)

- 还可以通过一些公开的error behaviour function对error behaviour进行判断

```
//os/error.go

func IsExist(err error) bool {
    return isExist(err)
}
func IsNotExist(err error) bool { ... }
func IsPermission(err error) bool { ... }
```

例子:

```
f, err := ioutil.TempFile("", "_Go_ErrIsExist")
f2, err := os.OpenFile(f.Name(), os.O_RDWR|os.O_CREATE|os.O_EXCL, 0600)
if os.IsExist(err) {
    fmt.Println("file exist")
    return
}
```

error naming

- 错误类型: xxxError

```
//net/net.go
type OpError struct { ... }
type ParseError struct { ... }
type timeoutError struct{}
```

- 导出的错误变量: ErrXxx

```
//io/io.go
var ErrShortWrite = errors.New("short write")
var ErrNoProgress = errors.New("multiple Read calls return no data or error")
```

消除重复

问题：下面的代码片段重复多次 - 不优雅！

```
var err error
err = doSomethingA()
if err != nil {
    return err
}

err = doSomethingB()
if err = nil {
    return err
}

err = doSomethingC()
if err = nil {
    return err
}

... ...
```

消除重复(cont.)

- 将error作为内部状态

```
//bufio/bufio.go
type Writer struct {
    err error
    buf []byte
    n   int
    wr  io.Writer
}
func (b *Writer) Write(p []byte) (nn int, err error) {
    if b.err != nil {
        return nn, b.err
    }
    ... ...
}

//writer_demo.go
buf := bufio.NewWriter(fd)
buf.WriteString("hello, ")
buf.WriteString("gopherchina ")
buf.WriteString("2017")
if err := buf.Flush() ; err != nil {
    return err
}
```

消除重复(cont.)

- 可能包含太多职责
- 重构!

Orthogonal Composition

语言设计

- 正交性

```
- 无类型体系，类型定义正交独立
- 方法和类型的正交性：每种类型都可以拥有自己的method
- interface与其实现之间无"显式关联"
- 正交性为"组合"策略的实施奠定了基础
```

- 组合

"If C++ and Java are about type hierarchies and the taxonomy of types, Go is about composition." - Rob Pike

```
- 类型间的耦合方式直接影响到程序的结构
- Go语言通过“组合”方式构架程序结构
- 垂直组合(类型组合)：type embedding
- 水平组合：通过interface“连接”
```

- 更大概念上的组合：goroutines和channels

vertical composition thought

- vertical composition by type embedding
- 组合而不是继承，没有父子类型的概念，没有向上、向下转型(type casting)
- 被嵌入的类型并不知道将其嵌入的外部类型的存在
- Method匹配取决于方法名字，而不是类型

vertical composition thought (cont.)

- construct interface by embedding interface

```
type ReadWriter interface {
    Reader
    Writer
}
```

- construct struct by embedding interface

```
type MyReader struct {
    io.Reader // underlying reader
    N int64   // max bytes remaining
}
```

- construct struct by embedding struct

```
// sync/pool.go
type poolLocal struct {
    private interface{}   // Can be used only by the respective P.
    shared  []interface{} // Can be used by any P.
    Mutex                 // Protects shared.
    pad     [128]byte     // Prevents false sharing.
}
```

interface - Go语言真正的魔法

- **interface决定Go语言中类型的水平组合方式**
- interface与其实现者之间的关系是隐式的，无需显式的"implements"声明(但编译器会做静态检查)
- interface仅仅是method集合（没有数据）
- method和普通function一样声明，无需在特定位置

small interface thought

- 小接口（通常1-3个方法）
- 典型接口定义：

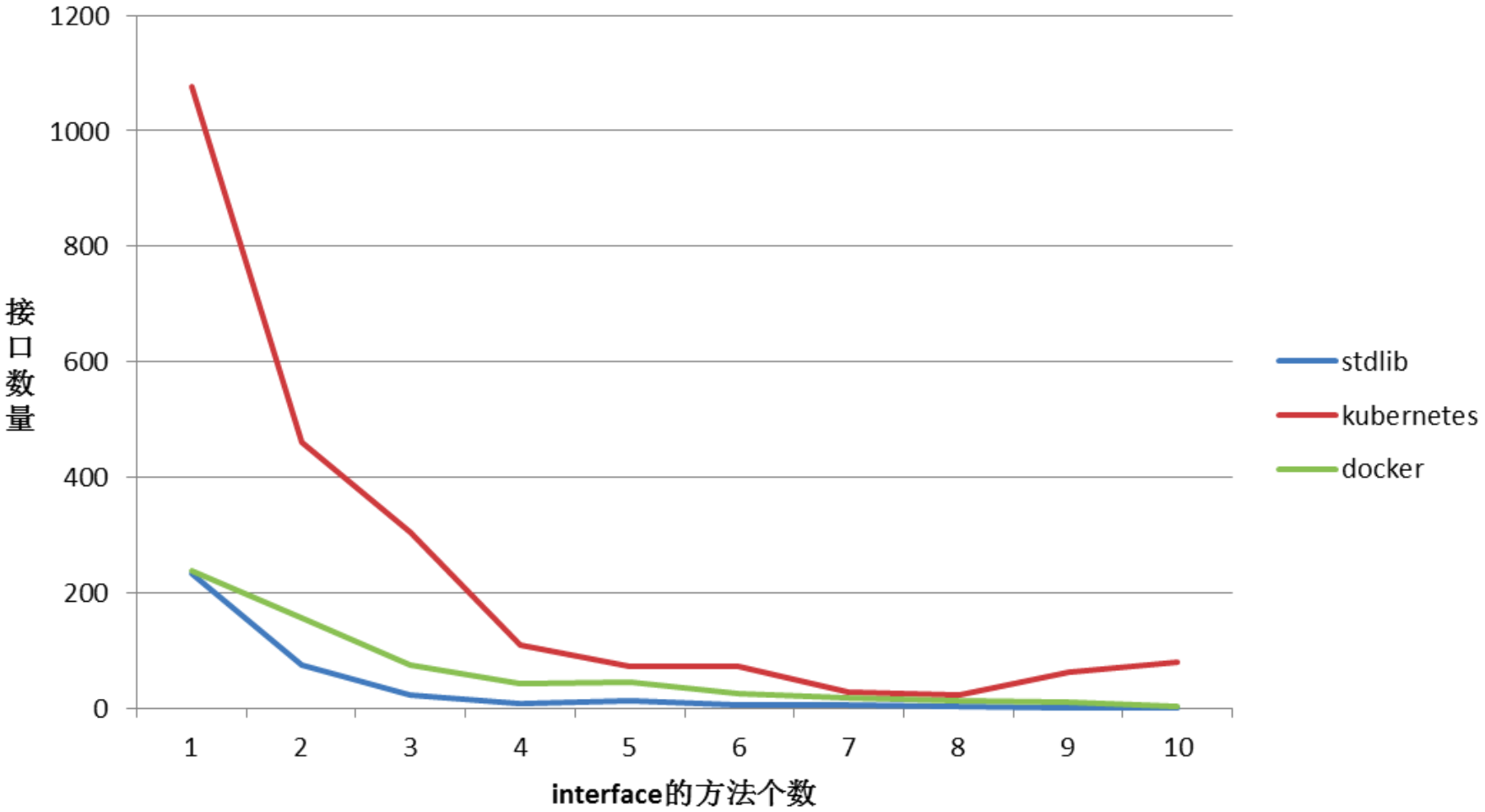
```
// builtin/builtin.go
type error interface {
    Error() string
}

// io/io.go
type Reader interface {
    Read(p []byte) (n int, err error)
}

// net/http/server.go
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}

type ResponseWriter interface {
    Header() Header
    Write([]byte) (int, error)
    WriteHeader(int)
}
```

方法个数统计


1200
1000
800
600
400
200
0
接口数量
1
2
3
4
5
6
7
8
9
10
interface的方法个数
stdlib
kubernetes
docker

定义小接口的考量

小接口的优势:

- 方法少，职责单一
- 易于实现和测试
- 通用性强
- 易于组合(如:io.Reader)

如何定义小接口?

- 领域理解
- 聚焦抽象

horizontal composition thought

- 关注如何通过接口进行“连接”的方式实现水平组合，以解决大问题、复杂的问题
- 通过function进行组合：接受interface类型参数

```
- basic form
- wrapper function (chain)
- adapter function type
- middleware
```

基本形式

- 接受interface类型参数

```
func ReadAll(r io.Reader) ([]byte, error)
func Copy(dst Writer, src Reader) (written int64, err error)
```

- 形式：someFunc(interface value parameter)

wrapper function

- 接受interface类型参数，并返回与其参数类型相同的返回值

wrapper function:

```
func LimitReader(r Reader, n int64) Reader { return &LimitedReader{r, n} }
```

interface implementation:

```
type LimitedReader struct {
    R Reader // underlying reader
    N int64  // max bytes remaining
}
func (l *LimitedReader) Read(p []byte) (n int, err error) {}
```

usage:

```
r := strings.NewReader("some io.Reader stream to be read\n")
lr := io.LimitReader(r, 4)
if _, err := io.Copy(os.Stdout, lr); err != nil {
    log.Fatal(err)
}
// Output: some
```

wrapper function chain

- CapReader

```
func CapReader(r io.Reader) io.Reader {
    return &capitalizedReader{r: r}
}

type capitalizedReader struct {
    r io.Reader
}

func (r *capitalizedReader) Read(p []byte) (int, error) {
    n, err := r.r.Read(p)
    if err != nil {
        return 0, err
    }

    q := bytes.ToUpper(p)
    for i, v := range q {
        p[i] = v
    }
    return n, err
}
```

wrapper function chain(cont.)

- 形式：wrapperFunc(wrapperFunc(wrapperFunc(...)))

```
s := strings.NewReader("some io.Reader stream to be read\n")
r := io.LimitReader(s, 4)
r = CapReader(r)
r = io.TeeReader(r, os.Stdout)
b, _ := ioutil.ReadAll(r) //SOME
fmt.Println(len(b))       //4
```

- 更为直观的用法：

```
s := strings.NewReader("some io.Reader stream to be read\n")
r := io.TeeReader(CapReader(io.LimitReader(s, 4)), os.Stdout)
```

adapter function type

- adapter function type将一个function转换为自己的类型，同时实现了某个interface
- 辅助快速实现某个"one-method" interface
- 例子: HandlerFunc

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}

type HandlerFunc func(ResponseWriter, *Request)

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}
```

- HandlerFunc adapts functions

```
func index(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Welcome!")
}

http.ListenAndServe(":8080", http.HandlerFunc(index))
```

middleware chain

- middleware = wrapper function + adapter function type

```
func logHandler(h http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        t := time.Now()
        log.Printf("[%s] %q %v\n", r.Method, r.URL.String(), t)
        h.ServeHTTP(w, r)
    })
}

func authHandler(h http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        err := validateAuth(r.URL.Query().Get("auth"))
        if err != nil {
            http.Error(w, "bad auth param", http.StatusUnauthorized)
            return
        }
        h.ServeHTTP(w, r)
    })
}

func main() {
    http.ListenAndServe(":8080", logHandler(authHandler(http.HandlerFunc(index))))
}
```

Preference in Concurrency

Concurrency

- 并发不是并行(https://golang.org/s/concurrency-is-not-parallelism)，并发不是关于性能的。
- 并发是关于程序结构的。
- 对于程序结构来说，concurrency是一个比interface组合更大的概念
- concurrency是一种在程序执行层面上的组合：goroutines各自执行特定的工作，通过channels+select将goroutines连接起来
- 适应现代计算环境
- 鼓励独立计算的分解

从某种意义上说，Go语言就是关于conconcurrency和interface的设计!

语言设计

- goroutines提供 **并发执行**, 它是Go runtime调度的基本单元

```
- goroutine实现了异步编程模型的高效，但却允许你使用同步范式编写代码，降低心智负担。
- goroutines被动态地多路复用到系统核心线程上以保证所有goroutines的正常运行。
```

- channels用于goroutines之间的 **通信和同步**

```
- channel是goroutines间建立联系的主要途径
- channel的功用类似Unix pipe
```

- select可以让goroutine同时 **协调处理** 多个channel操作

concurrency thought

本质：组合! 组合! 组合!

- 识别和分解出独立的计算单元放入Goroutines执行
- 使用channel/select建立Goroutines之间的联系

计算单元的拆解

- 因业务域不同而异
- 例如：素数筛实现为每个素数建立一个goroutine以筛除素数的倍数。

建立Goroutines间的“联系”

```
退出机制:
   "detached"
   "parent-child" relationships on quit

通信联系:
   service handle
   service handles aggregation
   dispatch-and-mix
```

"detached" goroutines

- 这类goroutines启动后与其创建者彻底分离(detached)，生命周期与程序生命周期相同
- 在后台执行一些特定任务，如：monitor、watcher等
- usually a "for-select" snippet
- timer or event driven
- 例子：Go GC goroutine

```
// runtime/mgc.go
go gcBgMarkWorker(p) // each P has a background GC G.

func gcBgMarkWorker(_p_ *p) {
    gp := getg()

    for {
      ... ...
    }
}
```

"parent-child" goroutines

- 通知并等待child goroutine退出

parent:

```
quit := make(chan string)
go child(quit)
```

child:

```
select {
  case c := <-workCh:
  // do something
  case <-quit:
  // do some cleanup
  quit<-"done"
}
```

parent:

```
quit<-"quit"
<-quit
```

"parent-child" goroutines (cont.)

- 当需要同时通知多个child goroutine quit时
- 通知并等待，直到timeout

parent:

```
quit := make(chan struct{})
for ... {
  go child(quit) // several child goroutines
}
```

child:

```
select {
  case c := <-workCh: // do something
  case <-quit: // do some cleanup
  return
}
```

parent:

```
close(quit)
time.Sleep(time.Second * 30)
```

"parent-child" goroutines (cont.)

- 如果parent要获得child的退出状态值，可以自定义quit channel中的元素类型

```
type ExitStatus interface {
    Status() int
}

type IntStatus int // an adapter
func (n IntStatus)Status() int {
  return int(n)
}

quit := make(chan ExitStatus) // for each child goroutine
```

child:

```
quit <- IntStatus(2017)
```

parent:

```
s := <-quit
fmt.Println(s.Status()) //2017
```

service handle

- 一些goroutine在程序内部提供特定service
- 这些goroutine使用channel作为service handle，其他goroutine通过service handle与其通信

```
//time/sleep.go
func After(d Duration) <-chan Time {
    return NewTimer(d).C
}
```

The timer service routine:

```
// runtime/time.go
func timerproc() {
  timers.gp = getg()
  for {
    ... ...
  }
}
```

- 考虑对于“慢消费者”，service goroutine应该如何处置：阻塞、丢弃...

service handles aggregation

- 消息来自多个不同service handle时

```
type msg struct {
    content string
    source  string
}

func wechatReceiver() <-chan *msg {
    c := make(chan *msg)
    go func() {
        c <- &msg{"wechat1", sourceWechat}
        c <- &msg{"wechat2", sourceWechat}
        c <- &msg{"wechat3", sourceWechat}
    }()

    return c
}

func weiboReceiver() <-chan *msg {...}
func textmessiageReceiver() <-chan *msg {...}
```

service handles aggregation (cont.)

- 对于不固定数量的聚合，用goroutine(非常类似于unix pipe chain)

```
func serviceAggregation(ins ...<-chan *msg) <-chan *msg {
    out := make(chan *msg)
    for _, c := range ins {
        go func(c <-chan *msg) {
            for v := range c {
                out <- v
            }
        }(c)
    }
    return out
}

c := serviceAggregation(weiboReceiver(), wechatReceiver(), textmessageReceiver())
m := <-c // 获取message并处理
```

service handles aggregation (cont.)

- 对于固定数量聚合，可以用select

```
func serviceAggregation(weibo, wechat, textmessage <-chan *msg) <-chan *msg {
    out := make(chan *msg)

    go func(out chan<- *msg) {
        for {
            select {
            case m := <-weibo:
                out <- m
            case m := <-wechat:
                out <- m
            case m := <-textmessage:
                out <- m
            }
        }
    }(out)
    return out
}
c := serviceAggregation(weiboReceiver(), wechatReceiver(), textmessageReceiver())
m := <-c // 获取message并处理
```

dispatch-and-mix goroutines

- 在“微服务”时代，我们在处理一个请求时经常调用多个外部微服务并综合处理返回结果：

```
func handleRequestClassic() {
  r1 := invokeService1()
  //handle result1
  r2 := invokeService2()
  //handle result2
  r3 := invokeService3()
  //handle result3
}
```

- 不足：

```
- 顺序调用
- 慢
- 不可预知
```

dispatch-and-mix goroutines(cont.)

- 将处理请求时对外部的服务调用分发到goroutine中，再汇总返回结果

```
func handleRequestByDAM() {
  c1, c2, c3 := make(chan Result1), make(chan Result2), make(chan Result3)
  go func() { c1 <- invokeService1() } ()
  go func() { c2 <- invokeService2() } ()
  go func() { c3 <- invokeService3() } ()
  timeout := time.After(200 * time.Millisecond)
  for i := 0; i < 3; i++ {
      select {
      case r := <-c1: //handle result1
          c1 = nil
      case r := <-c2: //handle result2
          c2 = nil
      case r := <-c3: //handle result3
          c3 = nil
      case <-timeout:
          fmt.Println("timed out")
          return
      }
  }
  return
}
```

dispatch-and-mix goroutines with context

- 通过Context可以cancel掉已经向外发起的在途请求，释放占用资源

```
type service func() result
func invokeService(ctx content.Content, s service) chan result {
  c := make(chan result)
  go func() {
    c1 := make(chan result)
    go func() {
      c1 <-s()
    }
    select {
      case v := <-c1:
           c <-v
      case <-ctx.Done():
      // cancel this in-flight request by closing its connection.
    }
  }()
  return c
}
```

dispatch-and-mix goroutines with context (cont.)

```
func handleRequestByDAM() {
  ctx, cf := context.WithCancel(context.Background())
  c1, c2, c3 := invokeService(ctx, service1), invokeService(ctx, service2),
                invokeService(ctx, service3)
  timeout := time.After(200 * time.Millisecond)
  for i := 0; i < 3; i++ {
      select {
      case r := <-c1: //handle result1
      case r := <-c2: //handle result2
      case r := <-c3: //handle result3
      case <-timeout:
          cf() // cancel all service invoke requests
          return
      }
  }
  return
}
```

- 优点:

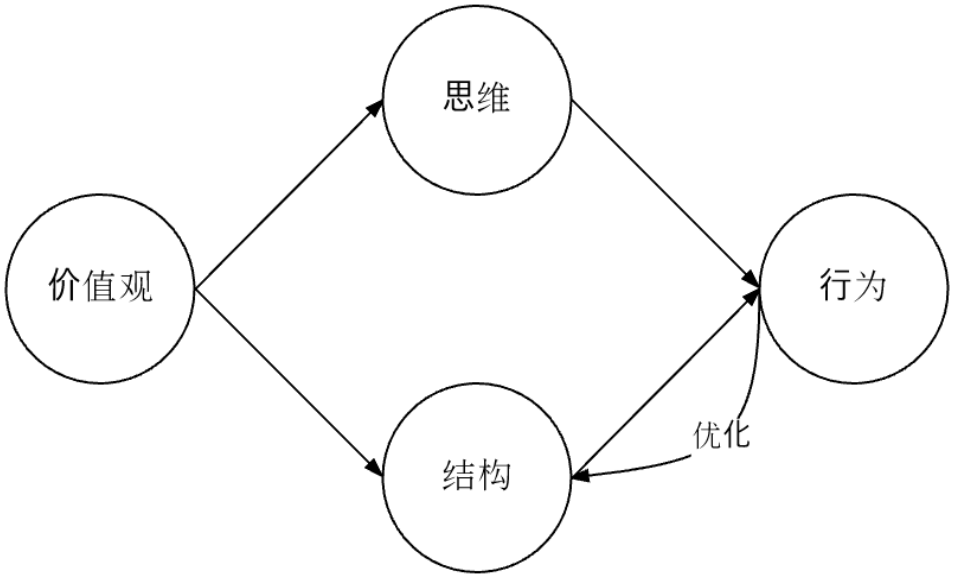
```
- 并发
- 快
- 显式timeout/cannel，实现可预知
```

closing thoughts

- make programming fun when coding in go way
- "less is more"

Go 2.0?

- Go 1.9: 2017.8 Release!
- Then Go 1.10 or Go 2.0?
- 回顾"编程语言思维的形成"模型：行为对结构的反馈，导致结构的持续改变和优化，促进语言演化
- 价值观不变



Thank you

Tony Bai
Neusoft
Weibo: @tonybai_cn (#ZgotmplZ)
Weixin: tonybai_cn (#ZgotmplZ)
Weixin Official Account: iamtonybai (#ZgotmplZ)
Blog: http://tonybai.com (#ZgotmplZ)