

What?

- Godaddy (去你爹, X)
- Go a head (去个头, X)
- Golang (Go语言, YES)

Go a head...

Golang is

一个在语言层面实现了并发机制的类C
通用型编程语言

Why Golang?

- 云计算时代，多核化和集群化是趋势
- 传统软件不能充分利用硬件资源
- 传统编程语言多核并发支持比较繁琐
- 生产效率，少即是多

Go前世今生

- 1995 Bell Labs, Plan9 -> Inferno (Limbo)
- 2007/09 Google's 20% project
- 2008/05 Google full-time project
- 2009/11 officially announced
- 2012/03 Go1 Released

Go语言特性

- 语言层面支持并发编程 (go)
- 优雅的错误处理机制 (defer)
- 简洁而又强大的面向对象表达 (OOP)
- 非侵入式接口 (Interface)
- 可扩展 (Cgo)

特性 (1)

- Go在语言层面支持并发

普通并发

// in Java (简化, 用标准库中的线程模拟并发)

```
public class MyThread implements Runnable {  
  
    String arg;  
  
    public MyThread(String a) {  
        arg = a;  
    }  
  
    public void run() {  
        // ...  
    }  
  
    public static void main(String[] args) {  
        new Thread(new MyThread("test")).start();  
        // ...  
    }  
}
```


Go语言的并发

```
func run(arg string) {  
    // ...  
}
```

```
func main() {  
    go run("test")  
    ...  
}
```


goroutine

启动一个异步过程

```
func foo(arg1 T1, arg2 T2) {  
    // ...  
}  
go foo(arg1, arg2)
```


goroutine 交互

// 等待结束

```
func foo(arg1 T1, arg2 T2, done chan bool) {  
    // ...  
    done <- true  
}
```

```
done := make(chan bool)  
go foo(arg1, arg2, done)  
// ...  
<-done  
// 得到结果
```


CSP

- 没有共享内存,更没有内存锁
- 通信靠channels来传递消息

特性 (2)

- 优雅的错误处理机制

Go错误处理范式

```
// 文件操作
file, err := os.Open(fileName)
if err != nil {
    return
}
defer file.Close()
... // 操作已经打开的 f 文件
```

```
// 锁操作
var mutex sync.Mutex
// ...
mutex.Lock()
defer mutex.Unlock()
... // 正常代码
```


内建error类型

```
type error interface {  
    Error() string  
}
```


普通资源释放

// In Java

```
Connection conn = ...;
try {
    Statement stmt = ...;
    try {
        ResultSet rset = ...;
        try {
            ... // 正常代码
        }
        finally {
            rset.close();
        }
    }
    finally {
        stmt.close();
    }
}
finally {
    conn.close();
}
```


Go的资源释放

```
// In Golang  
conn := ...  
defer conn.Close()
```

```
stmt := ...  
defer stmt.Close()
```

```
rset := ...  
defer rset.Close()
```

```
... // 正常代码
```


特性 (3)

- 简洁而又强大的面向对象表达

结构体 (Struct)

// 是类，不只是结构体

```
type Foo struct {  
    a int  
    b string  
}
```

```
func (this *Foo) Bar(arg1 T1, arg2 T2, ...) (out1 RetT1, ...) {  
    // ...  
}
```


面向对象

```
type Point struct {  
    x, y int  
}  
  
func (p *Point) Get() (int, int) { // Public  
    return p.x, p.y  
}  
  
func (p *Point) Put(x, y int) { // Public  
    p.x = x  
    p.y = y  
}  
  
func (p *Point) add(x, y int) int { // private  
    return p.x + p.y  
}
```


模拟继承

```
type YetAnotherPointer struct {  
    Point // 匿名字段  
    z int  
}
```

```
func (p *YetAnotherPointer) Get() (int, int, int) {  
    return p.x, p.y, p.z  
}
```

```
m := YetAnotherPointer{Pointer{1,2}, 3}  
fmt.Println(m.Get())
```


特性 (4)

- 非侵入式接口 (Interface)

普通接口实现

```
class Foo implements IFoo { // Java文法
    ...
}
```

```
class Foo : public IFoo { // C++文法
    ...
}
```

```
IFoo* foo = new Foo;
```


Go语言接口实现

```
type IBar interface {  
    Bar(arg1 T1, arg2 T2, ...) (out1 RetT1, ...)  
}
```

```
type Foo struct {  
    ...  
}
```

```
var foo IFoo = new(Foo)
```


非侵入式接口

```
type Pointer interface {  
    Get() (int, int)  
    Put(x, y int)  
}
```

```
type Point struct {  
    x, y int  
}
```

```
func (p *Point) Get() (int, int) {  
    return p.x, p.y  
}
```

```
func (p *Point) Put(x, y int) {  
    p.x = x  
    p.y = y  
}
```


接口查询

```
var a interface{} = ...
```

```
if w, ok := a.(io.Writer); ok {  
    // ...  
}
```

```
if foo, ok := a.(*Foo); ok {  
    // ...  
}
```


特性 (5)

- 可扩展 (Cgo)
- 与 C 的交互，是除了C++、Objective-C这两个以兼容C为前提的语言外中最简单的。

C字符串转Go

```
/*  
#include <stdlib.h>  
char* GetString() { ... }  
*/  
  
import "C"  
import "unsafe"  
  
func GetString() string {  
    cstr := C.GetString()  
    str := C.GoString(cstr)  
    C.free(unsafe.Pointer(cstr))  
    return str  
}
```


更多特性

- 模块化
- 反射
- Unicode
- 跨平台
-

内建类型

切片(slice)

```
arr := make([]T, n) // make([]T, len, cap)
```

```
arr := []T{t1, t2, ...}
```

```
slice := arr[i:j] // arr[i:], arr[:j]
```

字符串(string)

```
str := "Hello, world"
```

```
substr := str[i:j]
```

字典(map)

```
dict := make(map[KeyT]ValT)
```

```
dict := map[KeyT] ValT{k1: v1, k2: v2, ...}
```

```
dict[k] = v
```


切片 (slice)

// slice append

```
var arr []int
arr = append(arr, 1)
arr = append(arr, 2, 3, 4)
arr2 := []int{5, 6, 7, 8}
arr = append(arr, arr2...)
```

// slice copy

```
var a = [...]int{0, 1, 2, 3}
var s = make([]int, 2)
```

```
n1 := copy(s, a[0:]) // n1 == 2, s == []int{0,1}
n2 := copy(s, a[2:]) // n2 == 2, s == []int{2,3}
```


字典(map)

插入

`dict[k] = v`

删除

`delete(dict, k)`

查询

`v, ok := dict[k]`

字符串 (string)

```
var s string = "hello"
```

```
s[0] = 'a' // Error
```

```
// OK
```

```
s1 := []byte(s)
```

```
s1[0] = 'a'
```

```
s2 := string(s1)
```

字符串一旦定义，不可修改。字符串是字符的序列，不是字节的序列。

基本类型

- `bool` (`true`, `false`)
- 数字内型 (有符号/无符号, 有长度/无长度)
- `string` (内建“UTF-8”支持)
- `array` (`[n]<type>`)
- `slice` (`array[i:j]`)
- `map` (`map[<from_type>]<to_type>`)
- `chan`
- `error`

数字类型

- 无长度
 - int, uint
- 有长度
 - int8, int16, int32, int64
 - byte/uint8, uint16, uint32, uint64
 - float32, float64

控制语句

- if
- switch
- for
- 支持 break, continue 关键字

if

```
if x > 0 {  
    return y  
} else {  
    return x  
}
```

```
if err := os.Open(file); err != nil {  
    return err  
}  
doSomething(f)
```


switch

```
switch <val> {  
    case <expr1>[, <expr2>, <expr3>]:  
        // ...  
    case <exprN>:  
        // ...  
    default:  
        // ...  
}
```


fallthrough

```
switch i {  
    case 0:  
    case 1:  
        do() // 如果i == 0该函数不被调用  
}
```

```
switch i {  
    case 0: fallthrough  
    case 1:  
        do() // 如果i == 0该函数会被调用  
}
```


for

for init; condition; post { } // 计数循环

for condition { } // while 循环

for { } // for {;;} 死循环

for

```
sum := 0
```

```
for i := 0; i < 10; i++ {  
    sum += i  
}
```

```
list := []string{"a", "b", "c"}  
for k, v := range list {  
    // ...  
}
```


函数

```
func Name(arg1 T1, arg2 T2, ...) RetT {  
    // ...  
}  
func Name(arg1 T1, arg2 T2, ...) (out1 RetT1, out2 RetT2, ...) {  
    // ...  
}  
func Name(arg1 T1, arg2 T2, ...) (out1 RetT1, ..., err error) {  
    // ...  
}
```

优势

- 输入输出清晰形象
- 错误处理规范

goto

```
func myfunc() {
```

```
    i := 0
```

```
    Here:
```

```
        fmt.Println(i)
```

```
        i++
```

```
        goto Here
```

```
}
```


闭包(closure)

```
foo := func(arg1 T1, arg2 T2, ...) RetT {  
    // ...  
}  
out := foo(arg1, arg2, ...)
```

```
bar := func(arg1 T1, arg2 T2, ...) (out1 RetT1, out2 RetT2, ...) {  
    // ...  
}  
out1, out2, ... := bar(arg1, arg2, ...)
```


Panic and Recover

```
func check(err error) {  
    if err != nil {  
        panic(err)  
    }  
}
```

```
func safeHandler(fn http.HandlerFunc) http.HandlerFunc {  
    return func(w http.ResponseWriter, r *http.Request) {  
        defer func() {  
            if e, ok := recover().(error); ok {  
                http.Error(w, e.Error(), http.StatusInternalServerError)  
                log.Panic("panic fired in %v.panic - %v", fn, e)  
                log.Panic(string(debug.Stack()))  
            }  
        }()  
        fn(w, r)  
    }  
}
```

```
mux.HandleFunc("/", safeHandler(func(w http.ResponseWriter, r *http.Request)  
{ app.pageHandler(w, r) })))
```


Hello World

```
package main  
import "fmt"
```

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

```
export GOROOT=$HOME/go  
export PATH=$PATH:$GOROOT/bin
```

```
$ go build hello.go  
$ ./hello  
Hello, 世界
```




大道至简!

Q & A

@飞天急速徐倒立 | @why404
awhy.xu@gmail.com