



GIAC

全球互联网架构大会

GLOBAL INTERNET ARCHITECTURE CONFERENCE

2018 - Go语言将走向何方?

柴树杉(chai2010) @青云QingCloud



自我介绍(chai2010)

- Go语言代码贡献者, 国内第一批Gopher
- <Go高级编程><WebAssembly标准入门>作者
- 青云QingCloud从事OpenPitrix开发工作
- <https://github.com/chai2010>

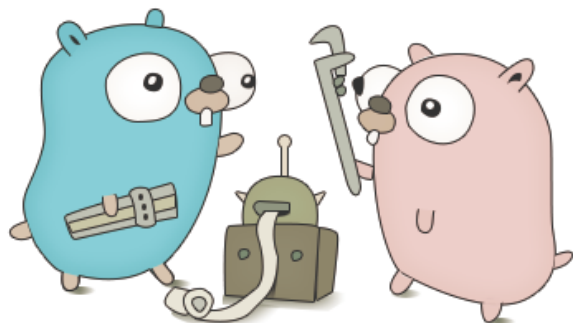
-
- 当歌曲和传说都已经哑巴的时候, 只有代码还在说话!
 - Less is more!



GO语言中国社区

GO高级编程

Advanced Go Programming



出版社

柴树杉 曹春晖 著



一切可编译为 WebAssembly 的，
终将会被编译为 WebAssembly。



WebAssembly 标准入门

柴树杉 丁尔男 著

中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS



内容大纲

1. 模块化(Go1.11+)
2. 错误处理草案(Go2)
3. 泛型草案(Go2)
4. WebAssembly(Go1.11+)

个人预测：2020年正式开始Go2开发，2030年终止Go1支持！



第1部分 模块化(Go1.11+)



1.1 模块的设计目标

同目录Go源文件的集合构成package；

同目录下子目录对应的package的集合构成module。

模块的设计目标：

- 可重现的构建：**同样代码，不同时间不同环境构建的输出必须是一样的**
- 依赖package版本的升级、降级管理

go get的问题：

- 如果\$GOPATH已有依赖包，则使用本地的，不同环境有差异
- go get -u 获取最新版本的package，不同时间最新版本不同

vendor的问题：

- vendor中每个package的版本是独立的，很难关联管理
- 嵌套的vendor导致诸多问题



1.2 模块快速入门 - 01

```
$ export G0111MODULE=on  
$ mkdir myapp
```

```
// myapp/hello.go  
package main
```

```
import "fmt"
```

```
func main() {  
    fmt.Println("hello module")  
}
```

```
$ go run hello.go  
go: cannot find main module; see 'go help modules'
```



1.2 模块快速入门 - 02

```
// myapp/go.mod  
module myapp
```

```
$ go run hello.Go  
hello module
```

含有go.mod文件的目录下的全部子包构成模块



1.3 go.mod 和 go.sum文件

```
module rsc.io/quote
```

```
require (  
    rsc.io/quote/v3 v3.0.0  
    rsc.io/sampler v1.3.0  
)
```

```
golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c
```

```
h1:qg0Y6WgZ0aTkIIMiVjBQcw93ERBE4m30iBm00nkL0i8=
```

```
golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c/go.mod
```

```
h1:NqM8EU0U14njkJ3fqMW+pc6Ldnwhi/IjpwHt7yyuw0Q=
```

```
rsc.io/quote/v2 v2.0.1 h1:DF8hmGbDhgiIa2tpqLjHLIKkJx6WjCtLEqZBAU+hACI=
```

```
rsc.io/quote/v2 v2.0.1/go.mod h1:EgjjEkPoRlzZbvGiUV/6yo8qd6yeDd/CP/9lRtfg4PU=
```

```
rsc.io/sampler v1.3.0 h1:7uVkiFmeBqHfdjD+gZwtXXI+R0DJ2Wc407MPEh/QiW4=
```

```
rsc.io/sampler v1.3.0/go.mod h1:T1hPZKmBbMNahiBKfy5HrXp6adAjACjK9JXDnKaTXpA=
```



1.4 go.mod文件深入

```
module my/thing
```

```
require (  
    other/thing v1.0.2  
    new/thing v2.3.4  
)
```

```
exclude (  
    old/thing v1.2.3  
)
```

```
replace (  
    bad/thing v1.4.5 => github.com/good/thing v1.4.5  
    bad/thing => /home/user/good/thing  
    bad/thing => ./good/thing  
)
```

replace不能是仓库镜像，必须是固定的版本



1.5 go get重新入门

```
go get -u          # to use the latest minor or patch releases
```

```
go get -u=patch   # to use the latest patch releases
```

```
go get github.com/gorilla/mux@master    # records current meaning of master
```

```
go get github.com/gorilla/mux@latest    # same (@latest is default)
```

```
go get github.com/gorilla/mux@v1.6.2    # records v1.6.2
```

```
go get github.com/gorilla/mux@'<v1.6.2' # records < v1.6.2
```

```
go get github.com/gorilla/mux@'>v1.6.2' # records > v1.6.2
```

```
go get github.com/gorilla/mux@e3702bed2 # records v1.6.2
```

```
go get github.com/gorilla/mux@c856192   # records v0.0.0-20180517173623-c856..
```



1.6 语义化版本号

v(major).(minor).(patch), such as v0.1.0, v1.2.3, or v3.0.1

major – 表示API完全不同

minor – 较大的版本兼容较小版本的API

patch – API没有变化, 仅仅用于修复BUG

v0.x.y – 属于开发版本, 可以出现API不兼容的改动

v1.x.y – 进入稳定版本, 导入路径不包含v1信息

v2.x.y – 大于v1的稳定版本, 导入路径包含v2信息

v0.0.0-20180523231146-b3f5c0f6e5f1

v1.2.0-alpha



1.7 v1/v2/v3版本共存

```
import "github.com/my/module/pkg/foo" // v0 or v1
import "github.com/my/module/v2/pkg/foo" // v2
import "github.com/my/module/v3/pkg/foo" // v3
```

v0/v1对应根目录

根目录必须有go.mod文件，表示这是一个模块

v0/v的版本对于v0.x.y和v1.x.y模式的tag

v2版对于v2子目录

v2子目录必须有go.mod文件，表示这是一个模块

v2的版本对于v2.x.y模式的tag



1.8 非v2/v3版本模式的子模块

```
import "github.com/my/module/pkg/foo" // v0 or v1  
import "github.com/my/module/submod/pkg" // submod
```

v0/v1对应根目录

根目录必须有go.mod文件，表示这是一个模块

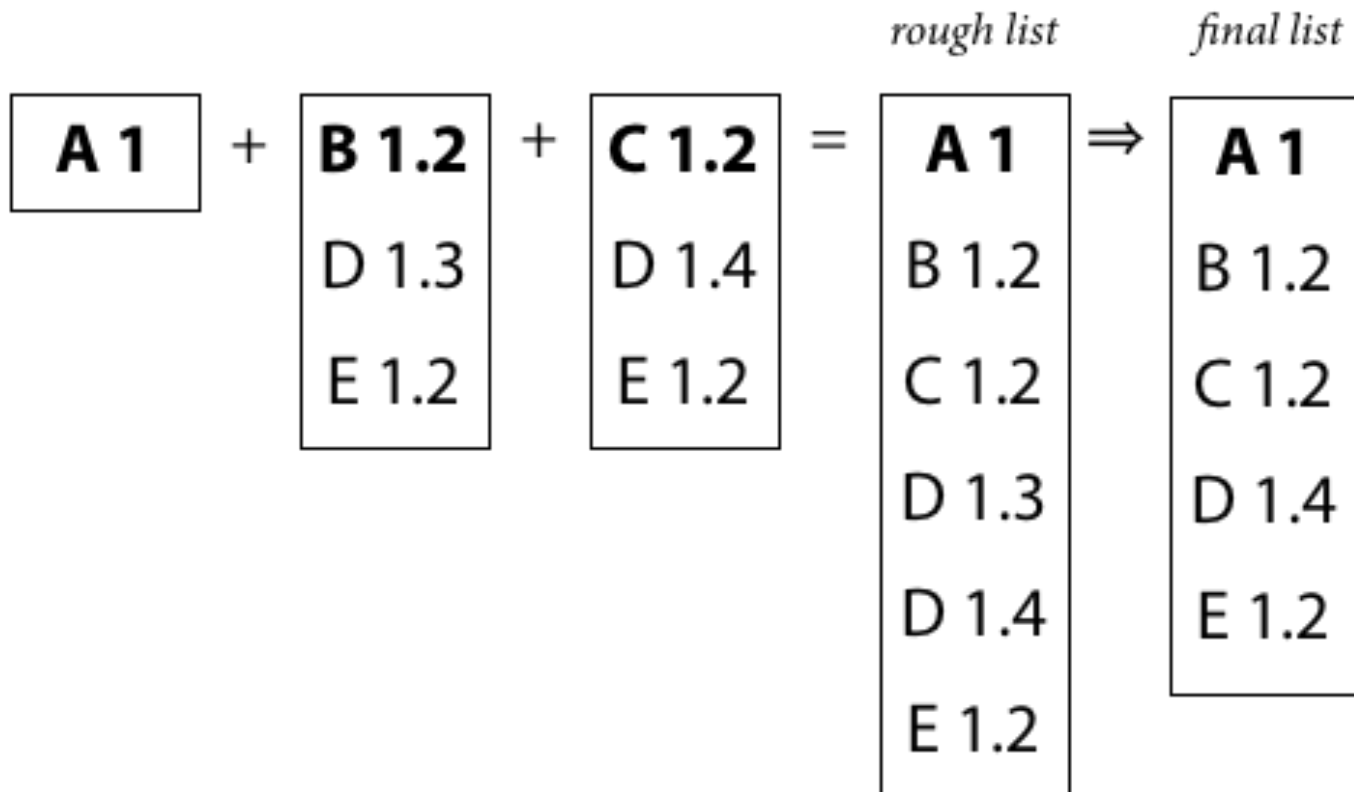
submod对应子目录，但是不是v2/v3模式

submod子目录含有go.mod文件，表示这是一个子模块

submod子模块的版本对于submod/v1.2.3模式的tag



1.9 最小化版本选择





1.10 版本不相容和间接依赖

```
require (  
    github.com/coreos/etcd v3.3.10+incompatible  
    github.com/golang/protobuf v1.2.0  
    golang.org/x/net v0.0.0-20181029044818-c440 // indirect  
    golang.org/x/sync v0.0.0-20180314180146-1d6 // indirect  
)
```

- etcd的v3版本和v1/v2是相同的导入路径，违背了Go模块的规则
- x/net虽然没有直接使用，但是protobuf模块的grpc插件间接引入了



1.11 go mod命令

download	下载依赖的module到本地cache
edit	编辑go.mod文件
graph	打印模块依赖图
init	在当前文件夹下初始化一个新的模块，创建go.mod文件
tidy	增加缺少的module，删除未用的module
vendor	将依赖复制到vendor下
verify	校验依赖的HASH码
why	解释为什么需要依赖



1.12 私有仓库/镜像

- replace只能定位到本地固定目录或某个tag
- replace目前不支持从镜像获取tag列表
- HTTPS_PROXY可以指定go get代理服务器
- GOPROXY环境变量指定模块代理服务器

```
HTTPS_PROXY=socks5://127.0.0.1:1080  
GOPROXY=file:///filesystem/path
```



第2部分 错误处理草案(Go2)



2.1 常见的错误处理流程

```
func CopyFile(src, dst string) error {
    r, err := os.Open(src)
    if err != nil {
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }
    defer r.Close()

    w, err := os.Create(dst)
    if err != nil {
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }

    if _, err := io.Copy(w, r); err != nil {
        w.Close()
        os.Remove(dst)
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }

    if err := w.Close(); err != nil {
        os.Remove(dst)
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }
    return nil
}
```



2.2 错误处理的问题

大量重复的错误代码，下面代码出现4次：

```
if err != nil {  
    return fmt.Errorf("copy %s %s: %v", src, dst, err)  
}
```

打断了表达式流程，无法链式操作：

```
v0, err := strconv.Atoi("123")  
if err != nil {  
    return err  
}  
v1 := v0*2
```



2.3 错误的捕获 - check & handle

```
func CopyFile(src, dst string) error {  
    handle err {  
        return fmt.Errorf("copy %s %s: %v", src, dst, err)  
    }  
  
    r := check os.Open(src)  
    defer r.Close()  
  
    w := check os.Create(dst)  
    handle err {  
        w.Close()  
        os.Remove(dst) // (only if a check fails)  
    }  
  
    check io.Copy(w, r)  
    check w.Close()  
    return nil  
}
```



2.4 单返回值的 check 是一个表达式

```
func printSum(a, b string) error {  
    handle err { return err }  
    fmt.Println("result:", check strconv.Atoi(x) + check strconv.Atoi(y))  
    return nil  
}
```



2.5 handle & check 处理流程

```
func process(user string, files chan string) (n int, err error) {
    handle err { return 0, fmt.Errorf("process: %v", err) } // handler A
    for i := 0; i < 3; i++ {
        handle err { err = fmt.Errorf("attempt %d: %v", i, err) } // handler B
        handle err { err = moreWrapping(err) } // handler C

        check do(something()) // check 1: handler chain C, B, A
    }
    check do(somethingElse()) // check 2: handler chain A
}

func printSum(a, b string) error {
    x := check strconv.Atoi(a)
    y := check strconv.Atoi(b)
    fmt.Println("result:", x + y)
    return nil
}
```




2.6 错误的上下文和再包装

- 错误的上下文主要包含发生错误的位置和函数调用帧信息
- 错误如果需要携带额外的上下文信息，需要再次包装
- 底层的错误类型在上层希望被包装为更友好的信息
- 但是错误的再包装可能导致原始错误信息的丢失



2.7 新的errors包

```
package errors
```

```
// A Wrapper is an error implementation  
// wrapping context around another error.  
type Wrapper interface {  
    // Unwrap returns the next error in the error chain.  
    // If there is no next error, Unwrap returns nil.  
    Unwrap() error  
}  
  
func Is(err, target error) bool  
func As(type E)(err error) (e E, ok bool)
```

再包装的错误必须实现 Wrapper 接口



2.8 错误的Is和As区别

Is是判断值相等：

```
// err == io.ErrUnexpectedEOF  
if errors.Is(err, io.ErrUnexpectedEOF) { ... }
```

As是判断类型断言：

```
// pe, ok := err.(*os.PathError)  
if pe, ok := errors.As(*os.PathError)(err); ok { ... }
```

对于被 `errors.Wrapper` 的错误，只要包装链满足一个即可！



2.9 定制错误输出格式

```
package errors
```

```
type Formatter interface {  
    Format(p Printer) (next error)  
}
```

```
type Printer interface {  
    Print(args ...interface{})  
    Printf(format string, args ...interface{})  
  
    Detail() bool  
}
```



第3部分 泛型草案(Go2)



3.1 缺少泛型的问题 - 大量重复代码

```
func MaxInt8 (a, b int8) int8 { if a > b { return a }; return b }  
func MaxInt16(a, b int16) int16 { if a > b { return a }; return b }  
func MaxInt32(a, b int32) int32 { if a > b { return a }; return b }
```

```
func sort.Float64s(a []float64)  
func sort.Ints(a []int)  
func sort.Strings(a []string)
```



3.2 缺少泛型的问题 - API不安全

```
var m1 = new(sync.Map)

m1.Store("key", "value")
m1.Store("key", 123)

v, ok := m1.Load("key")
println(v.(string)) // panic
```



3.3 泛型草案预览 - 安全的API

```
type List(type T) []T
func Keys(type K, V)(m map[K]V) []K

var ints List(int)
var keys = Keys(int, string)(
    map[int]string{1:"one", 2: "two"},
)
```




3.4 泛型草案预览 - 跨类型的代码复用

```
contract Addable(t T) { t + t }  
// or  
type Addable(t T) contract{ t + t }  
  
func Sum(type T Addable)(x []T) T {  
    var total T  
    for _, v := range x {  
        total += v  
    }  
    return total  
}  
  
var s1 := Sum(float32)(x)  
var s2 := Sum([]int{1, 2, 3})
```



3.5 泛型和接口 - 01

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

```
type ReaderB = interface {  
    Read(p []byte) (n int, err error)  
}
```

```
type ReaderC(r T) contract {  
    T: {  
        Read(p []byte) (n int, err error)  
    }  
}
```

// 泛型是编译时约束

// 接口运行时，泛型特化将转化为接口(别名还是新类型?)



3.5 泛型和接口 - 02

```
type Person(r T) contract {  
    T: {  
        Age int  
        Name string  
    }  
}
```

// 接口只有方法，接口无法定义成员
// 泛型是否可以约束成员？



3.6 泛型带来的影响

- Go将进军运算型领域(结合AVX512的支持), 比如机器学习
- 泛型和接口是两个不同的维度, 标准库需要针对泛型重新设计
- 跨类型的代码复用
- 设计更安全的API



第4部分 WebAssembly



4.1 WebAssembly是什么？

WebAssembly是一种新兴的网页虚拟机标准。

它的设计目标包括：

- 高可移植性
- 高安全性
- 高效率（包括载入效率和运行效率）
- 尽可能小的程序体积



Ending定律：一切可被编译为WebAssembly的，终将被编译为WebAssembly。

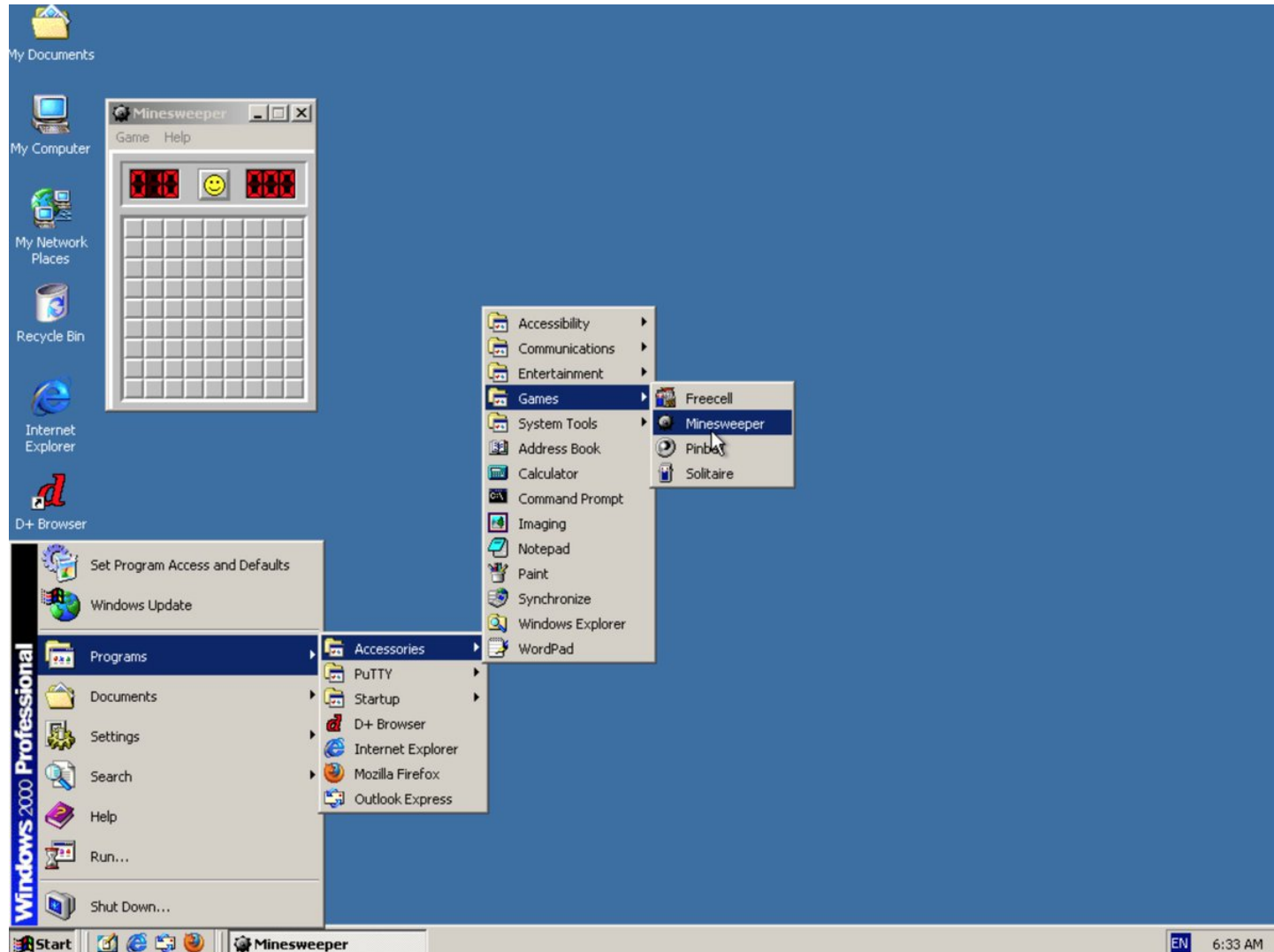


4.2 WASM最新动态

- 2018.07 - WebAssembly 发布1.0规范
- 2018.08 - Go1.11开始试验性地支持WebAssembly
- 2018.08 - Windows2000 运行在浏览器中
- 2018年诞生了诸多WASM开源项目
- 2018下半年, 4本WebAssembly书
- WASM的各种新闻井喷.....

Google Earth
AutoCAD

.....



Windows 2000 on WebAssembly: <https://bellard.org/jslinux/>₄₀



4.3 Ending定律

Ending's law: "Any application that can be compiled to WebAssembly, will be compiled to WebAssembly eventually."

Ending在2016年Emscripten技术交流会上给出断言：
所有可以用WebAssembly实现的终将会用
WebAssembly实现。



4.4 你好，WebAssembly

```
package main
```

```
import (  
    "fmt"  
)
```

```
func main() {  
    fmt.Println("你好，WebAssembly")  
}
```

```
$ export GOOS=js
```

```
$ export GOARCH=wasm
```

```
$ go run -exec="$(GOROOT)/misc/wasm/go_js_wasm_exec" hello.go
```

你好，Go语言



4.5 使用JS函数

```
package main
```

```
import (  
    "syscall/js"  
)
```

```
func main() {  
    alert := js.Global().Get("alert")  
    alert.Invoke("Hello wasm!")  
  
    js.Global().Call("eval", `  
        console.log("hello, wasm!");  
    `)  
}
```



4.6 JS回调Go函数

```
func main() {  
    var cb = js.NewCallback(func(args []js.Value) {  
        println("hello callback")  
    })  
    js.Global().Set("println", cb)  
  
    println := js.Global().Get("println")  
    println.Invoke()  
}
```



4.7 Go语言WASM虚拟机执行 - 01

```
import (  
    "github.com/go-interpreter/wagon/exec"  
    "github.com/go-interpreter/wagon/wasm"  
)  
  
func main() {  
    f, _ := os.Open("add.wasm")  
    m, _ := wasm.ReadModule(f, nil)  
    vm, _ := exec.NewVM(m)  
  
    // result = add(100, 20)  
    result, _ := vm.ExecCode(0, 100, 20)  
    fmt.Println(result)  
}
```



4.7 Go语言WASM虚拟机执行 - 02

```
;; add.wat
(module $modname
  (func $add (export "add") (param i32 i32) (result i32)
    get_local 0
    get_local 1
    i32.add
  )
)

$ wat2wasm add.wat
```



4.8 WebAssembly带来的影响

- 1.借助WASM平台，Go语言可以开发Web应用
- 2.Go内置WASM虚拟机，吸收其它语言的成果



4.9 WebAssembly中文资料



<https://github.com/chai2010/awesome-wasm-zh>



参考资料

1. <https://github.com/golang/go/wiki/Modules>
2. <https://github.com/golang/proposal/blob/master/design/go2draft.md>
3. <https://github.com/chai2010/awesome-wasm-zh>
4. <https://github.com/chai2010/awesome-go-zh>

GIAC

全球互联网架构大会

GLOBAL INTERNET ARCHITECTURE CONFERENCE



关注公众号获得
更多案例实践