Go语言简介

目录

Go是什么？

- Go是一款由 Google 开发的**编译型（compiled）**、**并发见长（concurrent）**、**支持垃圾回收（garbage collection）**、**静态类型（statically typed）**的语言。
- Golang的设计哲学：
  - 追求简单，少即时多
    - 语法简单，仅有25个关键字（C: 32, C++: 62; Java: 50），36个左右预定义标识符
    - 内置垃圾回收机制，无需开发人员处理内存回收功能，降低内存泄露风险
    - 没有类，继承，多态，构造/析构函数等概念，方法就是函数
    - 接口是隐式的，无需显示implements声明，属于Duck Typing（一个东西只要会像鸭子样游泳，鸭子样嘎嘎叫，那就可以看做一只鸭子）
  - 组合优于继承
    - 基于结构体嵌套，接口嵌套实现类似面向对象语言中继承的概念
  - 原生并发，高效轻量
    - 基于go关键字可以快速创建协程goroutine，其初始栈空间仅2K。runtime实现了G-M-P调度模型和work steadling算法，非常适合IO密集型应用

Go语言发展历程

- 2007年，谷歌工程师Rob Pike，Ken Thompson和Robert Griesemer开始设计一门全新的语言，用来解决Google内部使用C++的复杂性、编译构建速度慢，并发支持不便等问题，这是Go语言的最初原型。
- **2009年11月10日，Google将Go语言以开放源代码的方式向全球发布。**此后采用Weekly/Monthly Release 模式一直延续到Go1.0版本正式发布
- **2012年3月28日，Go1.0版本发布**，官方承诺只要符合G1语言规范的源代码，编译器保证向后兼容
- **2015年8月19日，Go1.5版本发布**，Go编译器和runtime全部用Go重写，原先的C实现被彻底移除，实现了自举
- **2018年8月24日，Go1.11版本发布**，引入Go module包管理模式，解决GOPATH无法管理版本的缺点
- **2022年3月15日，Go1.18版本发布**，正式引入泛型(Generics)，实现代码复用目的，提高开发效率
- **2023年2月1日，Go1.20版本发布**，这是目前最新版本，unsafe包新增了三个函数，拓展了切片到数组的转换支持

Go语言特点

- 同C语言一样，属于静态类型(Statically typed)，编译型(compiled)语言
- 支持自动内存垃圾回收(Garbage Collection)，无需手动进行内存回收处理。
- 语法简洁明(Simple Syntax)，继承了C语言中许多理念，比如基础数值类型，支持参数传值，指针等，但弱化了指针操作功能
- 基于包进行代码组织管理，包中变量或方法首字母大小写决定其可见性
- 天生支持并发编程，充分利用到多核CPU性能，通过协程(goroutine)和通道(channel)简化并发编程模式，内置sync/atomic等包支持并发操作
- 支持延迟函数，函数多个返回值，动态数组(切片)，哈希表(映射）、泛型编程等特性
- 不支持try-catch进行异常处理，而是通过返回错误来进行异常处理
- 不支持面向对象的语言中的类，没有构造器、继承，多态概念
- 跨平台运行支持，本地编译二进制可执行代码。内置丰富的工具链生态，可以简化编译，格式化，单元测试，基准测试，调度分析，性能优化等工作

使用Go语言的知名项目

- Docker: 开源的应用容器引擎，用于构建、部署和管理容器化应用程序。
- Kubernetes: 自动化部署、伸缩和操作应用程序容器的开源平台，其在容器编排领域占据了主导地位
- CockroachDB: 可伸缩的、跨地域复制的、支持事务的、高可用、强一致性的分布式 SQL 数据库
- InfluxDB: 分布式、高性能、可扩展的时间序列数据库。
- Etcd: 开源的分布式键值存储系统，用于管理和存储集群状态数据
- Caddy: 开源的 HTTP/2 Web 服务器，一键支持 HTTPS
- Hugo: 静态网站生成工具
- Prometheus: 开源的服务监控系统和时序数据库，一般和Grafana搭配使用
- Grafana: 监控信息的看板系统，支持Elasticsearch，InfluxDB，Prometheus等数据源

基本数据类型

- 数值类型
- 布尔类型
- 字符串
- 常量
- 指针

基本数据类型

- 数值类型
  - 整形
    - int int8 int16 int32 int64
    - uint uint8 uint16 uint32 uint64
  - 浮点型
    - float32
    - float64
  - 复数
    - complex64
    - complex128
    - complex

基本数据类型

- 布尔类型
  - true
  - flase
- 字符串
- 常量
  - 声明关键字为const
- 指针
  - 取地址操作符为&
  - 解引用操作符为*
  - 无法直接进行指针运算，需要引入unsafe包转换成uintptr后运算

复合类型

- 数组 array
- 切片 slice
- 映射 map
- 结构体 struct
- 通道 channel
- 接口 interface

复合类型

- 数组 array
  - 具有相同类型元素的集合，存储在连续内存空间类型
  - 数组大小也是类型的一部分，只有相同大小，且元素类型的一样的数组才能够比较
  - 不同于C语言中数组，Go语言数组是值传递的，传递给函数的是个拷贝，故函数内部改变该数组内容，不会影响外面的数组

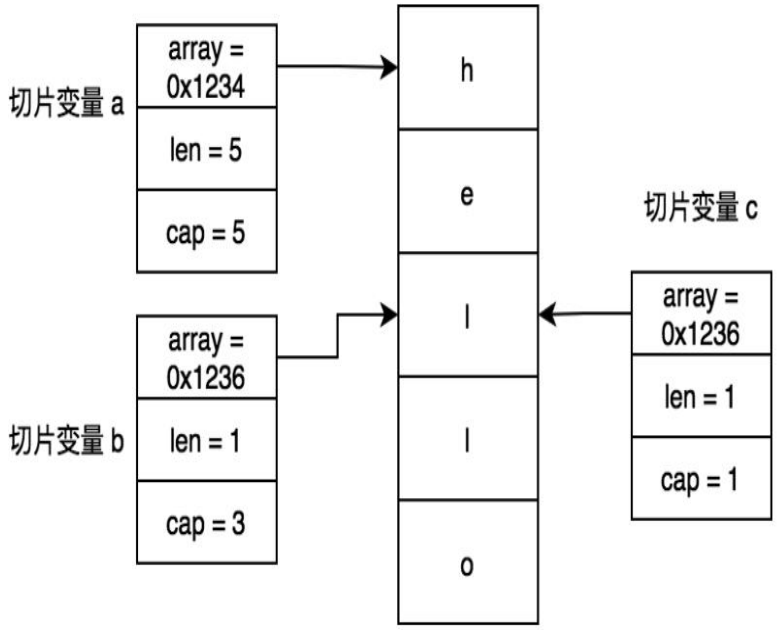
```go
// 数组定义
var a [5]int // a = [0, 0, 0, 0, 0]
b := [...]int{1, 2, 3, 4, 5} // 编译器会推断出数组大小
c := [5]int{1, 2, 3, 4, 5}
// 遍历
for i, v := range a {
    fmt.Printf("数组下标:%d, 数组元素值:%d\n", i, v)
}
// 数组是值传递
var arr1 [5]int = [5]int{1, 2, 3, 4, 5}
dobuleValue(arr1)
fmt.Println(arr1) //输出[1 2 3 4 5]

// 如果想像C那样，可以传递一个指向该数组的指针
doubleValue2(&arr1)
fmt.Println(arr1) // 输出[2 4 6 8 10]

func dobuleValue(arr [5]int) { // 值传递
    for i := 0; i < 5; i++ {
        arr[i] = arr[i] * 2
    }
    fmt.Println(arr) // 输出[2 4 6 8 10]
}

func doubleValue2(arr *[5]int) { // 指针传递
    for i := 0; i < 5; i++ {
        arr[i] = arr[i] * 2
    }
    fmt.Println(*arr) // 输出[2 4 6 8 10]
}
```

复合类型

- 切片 slice
  - 内置make函数用于创建slice: myslice = make([]T, length, capacity)
  - 动态数组，支持重组(reslice)操作: newslice = []T[low: high: max]
    - max可以不写，默认max=cap([]T)
    - len(newslice) = high-low ; cap(newslice) = max - low
  - 底层结构包含cap, len, pointer三个字段的结构体，pointer指针切片实际存储的地址，属于胖指针概念
  - 由于底层拥有指针，属于引用类型，作为参数传递时候，函数会修改原始值
  - 切片使用过程中时刻注意“副作用”，也许注意内存泄露问题
  - 切片进行扩容时候(append函数调用时)，当len小于1024时候，会扩容一倍，之后会每次扩容1/4

```
type slice struct {// slice底层结构
        array unsafe.Pointer // 底层数据数组的指针
        len   int // 切片长度
        cap   int // 切片容量
}
```

```
func main() {
        a := []byte{'h', 'e', 'l', 'l', 'o'} // 创建一个切片
        b := a[2:3] // reslice派生出一个切片
        c := a[2:3:3]
        fmt.Println(string(a), string(b), string(c)) // 输出 hello l l
}
```



复合类型

- 映射 map
  - 类似java语言中哈希表，能够在O(1)时间复杂度内get/set key-value数据。底层基于链表法实现的
  - 对于零值map，可以读取和删除，但不能写入（会panic）。**不支持并发操作**，另外**删除元素时候，并不会释放内存，存在内存泄露风险**。
  - 若需要并发操作，可以使用sync包中map数据结构

```go
// 类型格式:  map[K]V
// 创建
m1 := map[string]int{"a":1, "b":2}
m2 := make(map[string]int)
// 读取与更新
m2["a"] = 1
m2["b"] = 2 // 写入
fmt.Println(m2["c"]) // 读取

// 删除
delete(m2, "d")
delete(m2, "a") // 即使删除了，该内存空间也不会释放掉

// 迭代
for k, v := range m2 {
    fmt.Printf("key: %s, value: %d", k, v)
}
```

复合类型

- 结构体
  - 类似C语言中的结构体
  - 结构体字段间会有内存对齐，有时候为了减少false share，可以手动进行padding，这是某些场景下的可以采用的优化手段
  - 结构体是可以内嵌到其他结构体内，实现组合模式
  - 可比较性：字段类型，顺序一致才能够比较
  - 一般都是通过type关键字定义一个结构体，而不是直接使用strcut

```go
type baseStruct struct {
    f int
}

type base2Struct struct {
    h string
}

type exampleStruct struct {
    a          int8
    b          int16
    c          string
    d          bool
    baseStruct // 内嵌一个结构体
    base2      base2Struct // 内嵌一个结构体，并指定名字
}

// 定义结构体exampleStruct类型变量a
var a exampleStruct
fmt.Println(&a.a, &a.b) // 打印a和b字段对应的内存地址，
// 由于padding作用，两者相差不是8而是16

fmt.Println(a.baseStruct.f) // 访问f字段，也可以直接a.f访问
fmt.Println(a.base2.h)      // 访问h字段，只能通过base2访问
```

复合数据

- 通道
  - 通道也称信道，类似队列，FIFO
  - 通道是并发安全的，与goroutine一起简化了并发编程模式
  - 通道分为有缓存通道，和无缓冲通道，有缓冲通道底层有个ringbuffer数据结构
  - 按照是否支持读写，通道可分为只读通道，只写通道，以及可读可写通道
  - 对于零值通道，读取会一直阻塞，写入会panic
  - 遍历通道数据可以使用for-range控制结构，多通道同时操作，可以使用select控制结构体
  - 使用通道过程需要注意内存泄露问题
  - 从已关闭通道读取数据，会一直读取到通道元素类型的零值，一般用此作广播。
  - 可比较性：只要通道元素类型一致，即可以比较（不管通道缓冲大小）

```go
// 通道定义
// 创建整数类型的无缓冲信道
ci := make(chan int)
// 整数类型的无缓冲信道
cj := make(chan int, 0)
// 指向文件指针的带缓冲信道
cs := make(chan *os.File,5)

// 通道往往配合goroutine使用
c := make(chan int)  // 分配一个信道
// 在Go程中启动排序。当它完成后，在信道上发送信号。
go func() {
    list.Sort()
    c <- 1  // 发送信号，什么值无所谓。
}()
doSomethingForAWhile()
<-c   // 等待排序结束，丢弃发来的值。
```

复合类型

- 接口
  - 面向对象编程中，接口用来对行为进行抽象，即定义对象需要支持的操作，操作对应的就是接口中列出的方法
  - 空接口即没有定义任何方法的接口，Go语言中为interface{}。反之为非空接口。Go语言空接口类似C语言中void*，属于增强版void*
  - 接口类似结构体支持嵌套，践行组合优于继承的理念
  - 接口底层结构由两个字段组成，一个指向具体类型的类型定义，一直指向具体类型的值
  - 具体类型转换成接口属于装箱过程，反之为拆箱过程。任意类型都可以转换成空接口类型
  - 接口是隐式的，无需显示implements声明，属于Duck Typing
  - 类型的值只能实现值接收者的接口；指向类型的指针，既可以实现值接收者的接口，也可以实现指针接收者的接口
  - 通过类型断言可以判断是否属于某个接口，或者实现了某个接口，或者得到具体类型，断言有四种情况：
    - E to 具体类型 （E代表空接口）
    - E to I (空接口到非空接口)
    - I to 具体类型
    - I to I

复合类型—接口

```go
// 任意类型都可以隐式转换成空接口
var a int = 1
var b string = "hello"
var c interface{} = a
c = b

// 定义一个接口Handler
// 只要实现ServeHTTP方法的变量都可以隐式转换成Handler接口
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}

// 类型断言
var value interface{}
switch str := value.(type) {
case string: // E to 具体类型
    return str
case fmt.Stringer: // E to I
    return str.String()
}
```

```go
// 接口嵌套
type Reader interface {
    Read(p []byte) (n int, err error)
}

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

//接口嵌套 ReadWriter 接口结合了 Reader 和 Writer 接口。
type ReadWriter interface {
    Reader
    Writer
}

// 为强制保证某类型实现了某接口，可以这样
var _ Reader = (*UserdefineType)(nil)
```

控制结构

- for
  - 没有while关键字，都是通过for关键字实现
  - 和range关键字一起，用于字符串、切片、通道等遍历
- if/else
  - 条件表达式的值必须是bool类型值，不能是数字类型
- switch/case
  - 每个case分支不需要break关键字，默认达到该分支后不会向后执行了
  - 每个case分支支持多个值
  - 可以用于类型选择

```
// 如同C的for循环
for init; condition; post { }

// 如同C的while循环
for condition { }

// 如同C的for(;;)循环
for { }

// if 控制结构
if x > 0 {
    return y
}

// switch/case控制结构
func unhex(c byte) byte {
    switch {
    case '0' <= c && c <= '9':
        return c - '0'
    case 'a' <= c && c <= 'f':
        return c - 'a' + 10
    case 'A' <= c && c <= 'F':
        return c - 'A' + 10
    }
    return 0
}
```

控制结构

- select
  - 通道选择器，用于多个通道读取或写入处理，实现通道的多路复用
  - 类似switch/case结构，每个case语句都必须要待要处理的通道
  - 当所有case分支的通道都处于未准备就绪状态，那么会执行default代码(如果有的话)
  - **如果没有任何分支的话，那么当前gorutine会处于挂起状态**
  - 多个case分支的通道处于就绪状态，那么会**随机选择一个分支进行处理**（基于洗牌算法实现随机）

```go
// c1/c2通道处于读就绪，或者c3处于写就绪状态，那么会随机选择一个处理
select {
case msg1 := <-c1: // 从通道c1读取消息
    fmt.Println("received", msg1)
case msg2 := <-c2: // 从通道c2读取消息
    fmt.Println("received", msg2)
case c3 <- msg3: // 写入消息到c3通道
    fmt.Println("sended", msg3)
default:
    fmt.Println("当c1/c2/c3未就绪，那么会执行此处代码")
}

select {//没有任何case分支
    // 当前goroutine会挂起，不消耗cpu资源
}

for {
    // 死循环，会不停消耗cpu资源
}
```

函数

- Go中函数是一等公民(first class vlaue)
  - 可以赋值给一个变量
  - 可以作为返回值
  - 可以作为函数参数传递
  - 可以作为匿名函数使用
  - 可以作为闭包使用
- 支持多返回值
  - 一般最后一个返回值作为error错误，不同于c语言中使用线程局部变量errno，Go一般使用最后一个返回表示有没有错误发生
- 支持不固定参数(变参)
- 和go关键字一起创建协程，和defer关键字一起创建延迟执行函数
- 内置函数有：make/new/len/cap/append/copy/close/delete/panic/recover

函数

- init函数：每个包/文件可以定义一个或多个init函数，用于包加载时候初始化操作。

```go
// 支持多返回值
f, err := os.Open(filename) // 返回两个值，第二个值是错误类型
if err != nil {
    return "", err
}
defer f.Close() // defer关键字会创建一个延迟执行函数

// 函数赋值给一个变量
func add(a, b int) int {
    return a + b
}
fn := add
fmt.Println(fn(1, 2)) // 3

// 支持不固定参数
func Println(v ...interface{}) {
    std.Output(2, fmt.Sprintln(v...))
}

// go关键字用来创建协程
go func() {
    fmt.Println("这是一个gorutine")
}()
```

协程：goroutine

- Go语言中协程称为Gorutine = go + coroutine(协程)
- 轻量级，初始化栈空间仅2KB左右，创建简单：go关键字+匿名函数或者具名函数
- 同C语言一样，main函数时入口函数，Go语言中main是入口函数，也是一个协程函数
- Go协程的栈伸缩机制，保证栈不存在栈溢出风险
  - runtime会在协程函数最开始插入栈检查代码，如果发现栈空间不够，那就分配2的空间，把旧栈空间copy过去，然后再执行协程函数
  - runtime在执行gc期间，扫描到gorouine栈空间，只使用不到分配的1/4时候，会进行栈收缩处理，跟上面操作相反

方法

- Go方法类似OOP的中的方法，只不过方法的持有者不叫对象，而是叫接收者
- 方法本质就是普通的函数，方法的接收者就是隐含的第一个参数
- 将接受者的方法赋值给一个变量，这种变量叫方法值Method Value。通过该变量调用方法只需传递参数即可
- 除了接口类型外，任意类型变量都可以作为方法的接受者(receiver)
- 值类型和指针类型变量通过语法糖，可以调用指针类型或值类型接受者的方法
- 方法接收者分为指值接收者和指针接收者，当接收者占用空间较大时，推荐使用指针接收者：
  - func (receiver T) method()
  - func (recevier *T) method()

方法

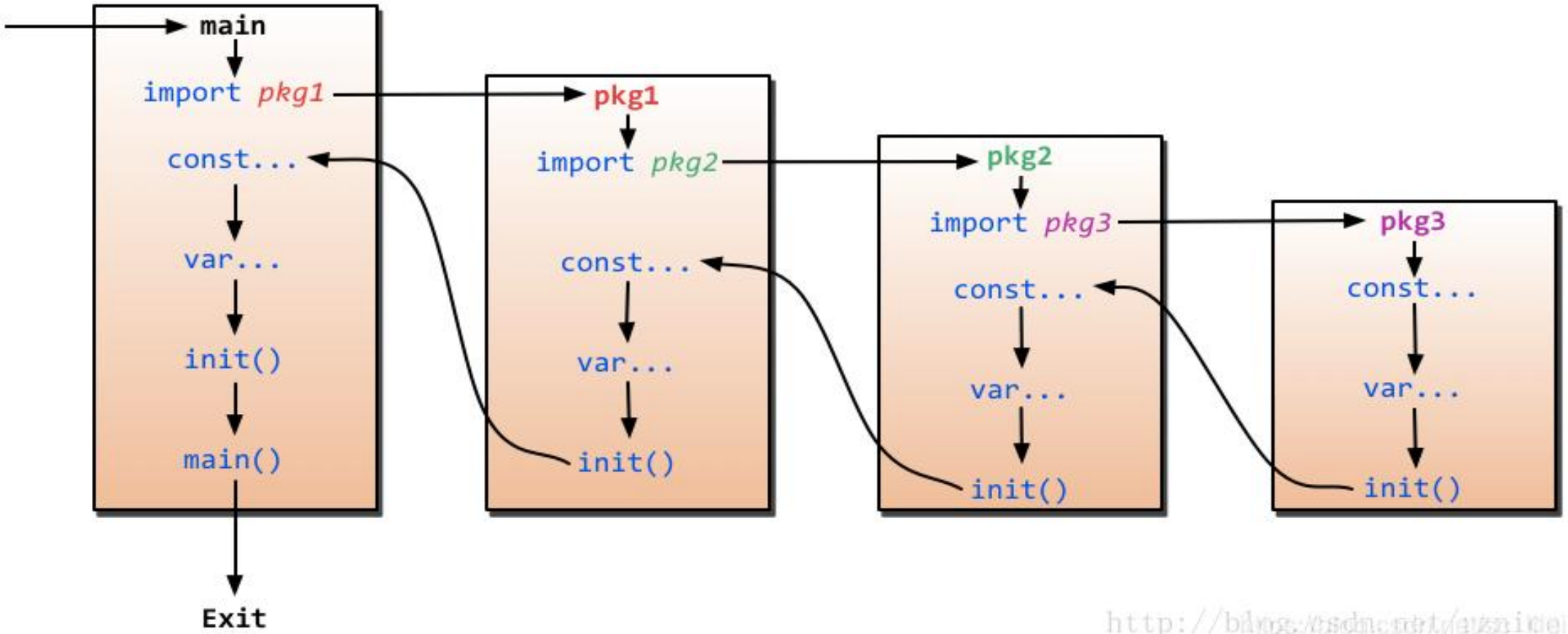
```go
type T struct {
    a int
}

func (tv T) Mv(a int) int           { return 0 } //  值接收者方法
func (tp *T) Mp(f float32) float32 { return 1 } // 指针接收者方法

func funcMv(t T, a int) int { // 普通函数，第一个参数是T类型变量
    return t.Mv(a)
}

var t T
fmt.Println(t.Mv(1)) // 输出 0
fmt.Println(t.Mp(0)) // 输出 1，这是个语法糖等效于(&t).Mp1()

// https://go.godbolt.org/z/PrYqcd13z 通过汇编看方法如何传参
t1 := reflect.TypeOf(T.Mv)
t2 := reflect.TypeOf(funcMv)
fmt.Println(t1 == t2) // 输出true，说明方法就是一个普通函数
```

包

- Go中通过包进行代码模块化管理，Go1.11之前使用GOPATH模式管理包，之后使用Go module模式管理
- 包可以避免名称冲突、隐藏未导出变量、函数等
- 包中首字母大写的变量、函数、接口等属于可导出的，可以被外界访问的
- main包的main函数是整个程序的入口
- 包可以有一个或多个init函数，用来进行包初始化相关操作
- 将包重命名为空白标识符，可以启动包的副作用

```go
// 单个导入
import "fmt"
import "bytes"

// 多个导入
import (
    "fmt"
    "bytes"
)

// 使用别名
import (
    "fmt"
    "bytes"
    // 将包重名为mybytes防止冲突
    mybytes "github.com/cyub/bytes"
)

import _ "net/http/pprof" // 引入副作用
```

包——包导入流程



代码示例

- 示例内容：计算从1到N自然数之和
  - 为了达到演示并发处理任务的目的，计算1到N之和，采用批量分块计算模式，每一批次使用独立线程或者协程并发处理。
  - 示例：假如N为10，每批次只处理4个数，那就需分三批次：[1,2,3,4]，[5,6, 7, 8], [9, 10]，演示程序会启动3个线程或者协程同时处理。
  - 示例代码：https://github.com/cyub/open
- 关注点：
  - 批量并发计算时，多线程/协程如何协同处理?
  - 多线程/协程并发计算过程中如何保证结果准确性?
  - C语言与Go语言处理有什么不一样之处?

学习资料

- Go官网：https://go.dev
- Go语言使用用户：https://github.com/golang/go/wiki/GoUsers
- Go语言圣经：https://golang-china.github.io/gopl-zh
- Go语言在线练习场：https://goplay.tools
- Go语言官方语法指南：https://go.dev/ref/spec
- Go语言101：https://gfw.go101.org/article/101.html
- Effective Go：https://go.dev/doc/effective_go
- Go by Example：https://gobyexample.com