

Go

xbZhong

2025-10-11

[本页PDF](#)

Golang的优势

由 `Google` 开发的语言，是一门编译型语言，编译出来的可执行文件（机器码）是单独的二进制文件，无需安装Go环境，不需要任何依赖（特殊情况除外）即可直接运行！！！

`docker` 和 `k8s` 都是基于go编写的

- 极简单的部署方式
 - 可直接编译成机器码
 - 不依赖其它库
 - 直接运行即可部署
- 静态类型语言（动态语言无编译器）
 - 编译的时候可以检查出来隐藏的大多数问题
 - 语言层面的并发
 - 天生的基因支持
 - 可以充分利用CPU多核
- 强大的标准库
 - `runtime` 系统调度机制
 - 可以帮助做垃圾回收，资源调度等
 - 高效的GC垃圾回收
 - 用了三色标记、混合回收等
 - 拥有丰富的标准库
- 简单易学
 - 25个关键字
 - 内嵌C语法支持
 - 具有面向对象特征
 - 跨平台

编译、执行时间对比

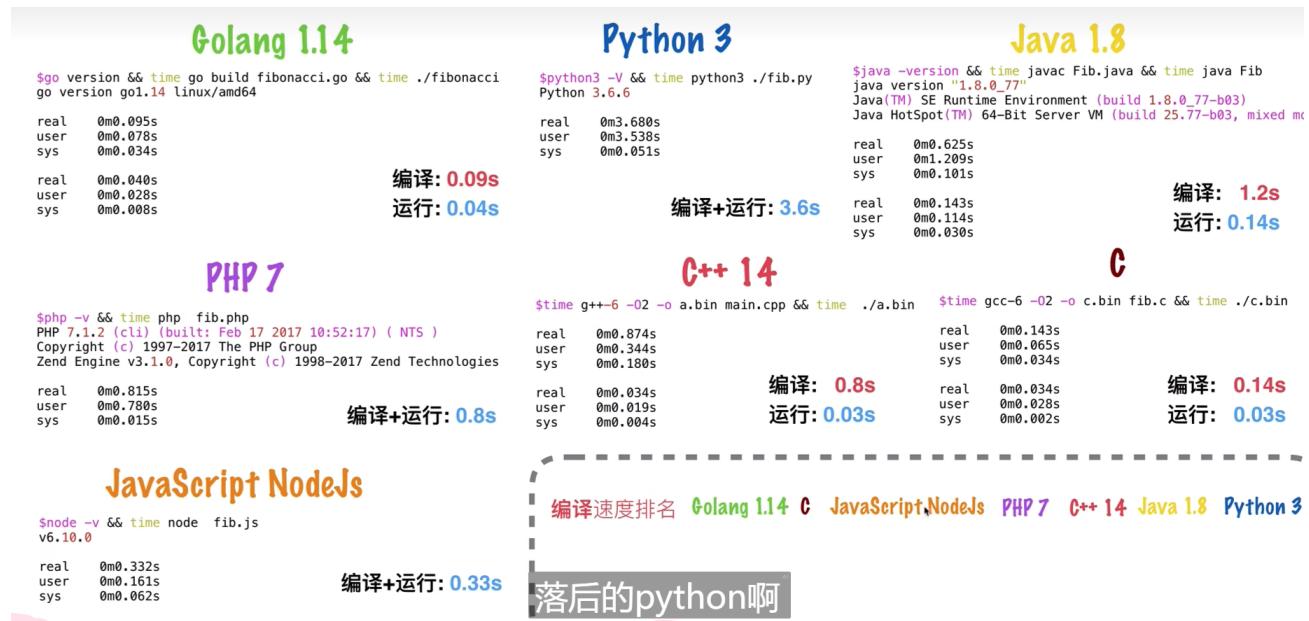


image-20251011122440219

Golang基本语法

命令行

目前先掌握这些即可

- `go run` : 编译并运行go程序
 - 只能运行可执行程序（有 `main()` 函数的程序）
- `go build` : 编译go程序，可针对任意包
 - `-o` : 后面跟输出文件名
- `go version` : 查看go版本
- `go get path` : 从远程仓库下载G模块或包到本地
- `go env` : 查看环境变量

程序结构

Hello World

```

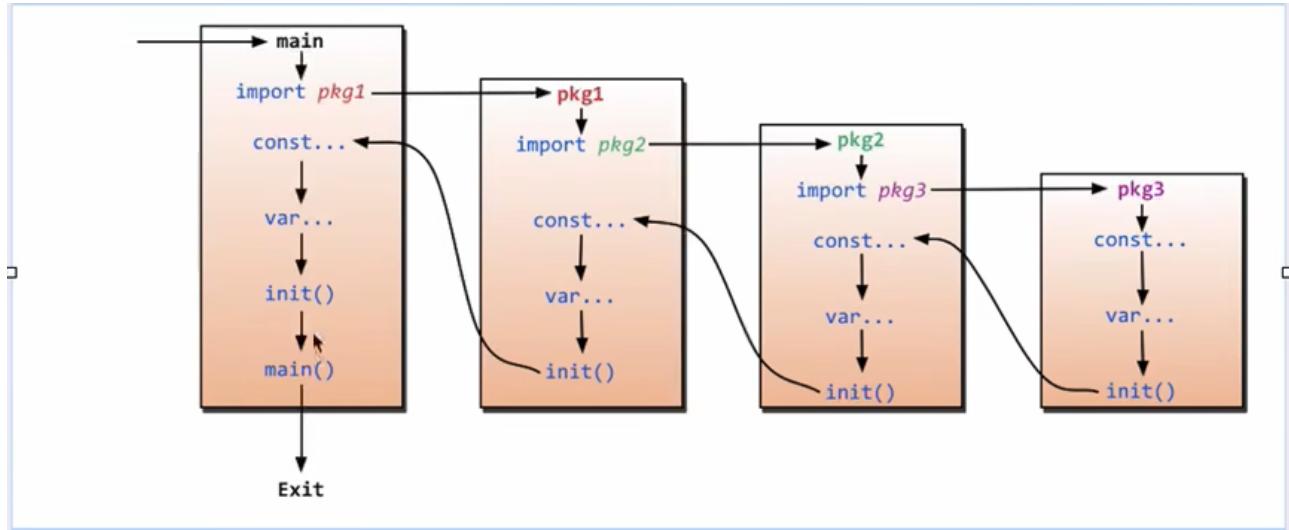
1 package main // 声明包
2
3 import "fmt" // 导入包
4
5 // 导多个包
6 import(
7     "fmt"
8     "time"
9 )
10
11 // 主函数
12 func main(){
13     fmt.Println("Hello World")
14 }
```

包的声明

- `package main` 表明这是一个**可执行程序**（而不是库）
 - 只有包含 `package main` 的程序才能编译为可执行文件
 - 普通包编译后生成的是库文件（`.a` 文件）
 - `main` 包编译后生成的是**可执行二进制文件**
- 包名通常与**源文件所在目录的最后一级目录名一致**
- 一个子文件夹内的所有源文件的`package`声明必须一致

包的导入

- `import` 导入了一个标准库 `fmt`，这个包主要用于往屏幕输入输出字符串，格式化字符串
 - `import` 后面可以接一个括号，导入多个包
 - `import` 语句导入的是**文件系统的目录路径**，而不是包名
- 在 `go` 中，**大写开头**的功能是可以公用的（公有），**小写开头**的功能只能在包里面使用（私有）
 - 功能包括**函数、方法、变量等**
- 导包的时候会先执行要导的包的 `init()` 函数，形成层级调用



- 可以使用 `_` 对已导入但不使用的包起别名，防止程序报错，但是会执行这个包的 `init()` 方法
 - 也可以在路径前指定别名
 - 可以使用 `.` 把导入的包里的方法全部引入，在当前源文件直接调用

```

1 package main
2
3 import(
4     _ "./lib1"
5     mylib2 "./lib2"
6     . "./lib3"
7 )
8
9 func main(){
10    // 用别名启用方法
11    mylib2.Lib2Test()
12
13    // 直接调用方法
14    Lib3Test()
15 }
```

语法

- 函数的主左括号一定要和函数名同一行，否则编译不通过

常见API

- `fmt.Println()` : 按顺序输出参数，但不会自动添加空格或换行
- `fmt.Println()` : 会在每个参数之间自动添加空格，并在结尾自动换行
- `fmt.Printf()` : 可以用 `%d`、`%s`、`%v` 等占位符来自定义输出，不会自动换行
- `fmt.Sprint()` : 拼接字符串并返回
- `fmt.Sprintf()` : 按照规则拼接字符串
- `fmt.Sprintf()` : 拼接并加空格 + 换行
- `strconv.ParseInt` : 将字符串转换为整型，返回值为**转换结果和错误信息**
 - `s` : 要转换的字符串
 - `base` : 进制
 - `10` : 十进制（最常用）
 - `2` : 二进制
 - `8` : 八进制
 - `16` : 十六进制
 - `bitSize` : 目标整数位数
- `math.MaxInt64` : 有符号最大值
- `math.MinInt64` : 有符号最小值
- 变量一，变量二 = 变量二，变量一 : 变量交换
- Go内置了 `Error` 接口，实现类的时候若有要求可以直接实现 `Error` 接口

排序

```
1 // 整数切片排序
2 sort.Ints(nums []int) // 升序
3 sort.IntsAreSorted(nums []int) bool // 检查是否已排序
4 sort.SearchInts(a []int, x int) int // 二分查找
5
6 // 浮点数切片排序
7 sort.Float64s(f []float64) // 升序
8 sort.Float64sAreSorted(f []float64) bool
9
10 // 字符串切片排序
11 sort.Strings(s []string) // 字典序
12 sort.StringsAreSorted(s []string) bool
```

基本数据类型

```
1 // 布尔类型
2 bool
3
4 // 字符串类型
5 string
6
7 // 整数类型
8 int int8 int16 int32 int64
9 uint uint8 uint16 uint32 uint64 uintptr
10
11 // uint8的别名，即一个字节
12 byte
13
14 // int32的别名，四个字节，表示一个Unicode字符，常用来表示单个字符
15 rune
16
17 // 浮点类型
18 float32 float64
19
20 // 复数类型
21 complex64 complex128
22
23 // 字符串类型，使用""或者``表示
24 string
```

格式化

- %v : 默认格式

- `%T` : 类型的字符串表示
- `%t` : 布尔值, 显示为 true 或 false
- `%d` : 十进制整数
- `%x` : 十六进制整数
- `%b` : 二进制整数
- `%c` : 对应的 Unicode 字符
- `%s` : 字符串
- `%q` : 双引号括起来的字符串, 适合打印 JSON
- `%p` : 指针的地址
- `%f` : 浮点数
- `%e` : 科学记数法的浮点数
- `%g` : 根据数值大小选择 `%e` 或 `%f`
- `%#v` : Go 语法表示的值
- `%#x` : 带有前缀 0x 的十六进制整数

变量声明

四种变量的声明方式

```

1 func main(){
2     // 方式一
3     var a int
4     // 方式二
5     var b int = 100
6     // 声明多个变量
7     var b,c int = 1,2
8     var bb,cc = 100,"zxb"
9     var(
10        bbb int = 100
11        ccc string = "zxb"
12    )
13    // 方式三
14    var c = 100
15    // 方法四: 省略var关键字
16    e := 100
17 }
```

`var`

- 基本语法: `var 变量名 变量类型`

- 使用 `var` 可以自动推导变量类型
- 用 `var` 声明的变量值默认值为0
- 使用 `fmt` 标准库的 `Printf` 方法打印数据类型
 - 打印数据类型: `fmt.Printf("type of a = %T", a)` , 占位符用 `%T`
- 可以声明全局变量
- 可以声明多个变量, 可以进行多行的多变量声明 (需要用括号括起来)

`:=`

- 使用 `:=` 可以实现变量的声明和初始化
- 可以自动推导变量类型
- 无法在函数外使用, 即无法声明全局变量
- 可以使用 `:=` 快速重新赋值, 而不是再声明一个

常量的定义

```

1 const(
2     BEIJING = 10*iota // iota=0
3     SHANGHAI          // iota=1
4     SHENZHEN           // iota=2
5 )
6 func main(){
7     const length int = 10
8 }
```

`const`

- 具有只读属性, 声明后不能再次修改
- 可以自动推断类型
- 可以声明全局变量, 可以使用大括号声明多个变量
 - 使用大括号声明的时候可以使用关键字 `iota` , 每行的 `iota` 都会累加1, 第一行的 `iota` 的值是0

函数

```
1 import "fmt"
2
3 // 示例
4 func fool(a string,b int) int{
5     fmt.Println("a = ",a)
6     fmt.Println("b = ",b)
7
8     c := 100
9     return c
10 }
11
12 // 返回多个返回值
13 func fool(a string,b int) (int,int){
14     return 666,777
15 }
```

常见说明

- go 的函数支持多返回值
 - 声明函数返回类型需要用括号指定多个返回值的类型
- 函数的参数类型写在参数名后面，返回类型写在函数名后面
- 函数名推荐驼峰命名法
- 对于多个相同类型的参数，可以只写一个参数类型
- 可以使用 `_` 对返回值进行忽略

带名称的返回值

- 函数值的返回值可以被命名
 - 作用域为当前函数范围！！！
- 使用空的return语句直接返回已命名的返回值

```
1 func split(sum int) (x,y int){
2     x = sum*4/9
3     y = sum-x
4     return
5 }
```

指针

和c语言的类似，在这给个例子看一下区别即可

- 接收指针类型参数的时候用 `*` 声明，也用 `*` 进行地址的解引用
- 用 `&` 获取变量的地址

```
1 package main
2
3 import "fmt"
4
5 func add(n int) {
6     n += 2
7 }
8
9 func addptr(n *int){
10    // 此时n里面存的是p的地址
11    *n += 2
12 }
13
14 func main(){
15     p := 5
16     add(p)
17     fmt.Println(p) // p = 5
18     addptr(&p)
19     fmt.Println(p) // p = 7
20 }
```

异常

在 Go 中的异常有三种级别：

- `error`：正常的流程出错，需要处理，直接忽略掉不处理程序也不会崩溃
- `panic`：很严重的问题，程序应该在处理完问题后立即退出
- `fatal`：非常致命的问题，程序应该立即退出

panic 和 recover

- 可以使用 `panic` 抛出异常，异常会逐层向上传播，如果没有 `recover`，程序会崩溃
- 可以使用 `recover` 捕获异常

```
1 func checkTemperature(temp float64) {
2     // defer 函数会在 checkTemperature 返回前执行,无论是因为正常 return 还是因为 panic
3     defer func() {
4         // 在函数结束前检查是否有 panic
5         if r := recover(); r != nil {
6             fmt.Println("\\"体温异常\\\"")
7         }
8     }()
9
10    if temp > 37.5 {
11        panic("体温异常") // 触发异常
12    }
13    // 函数正常结束
14 }
```

error

本身是一个预定义的接口，该接口方法下只有一个方法 `Error()`，该方法的返回值是字符串，用于输出类型信息

```
1 type error interface {
2     Error() string
3 }
```

创建error

- 使用 `errors` 包下的 `New` 函数

```
err := errors.New("这是一个错误")
```

- 是使用 `fmt` 包下的 `Errorf` 函数，可以得到一个格式化参数的 error

```
err := fmt.Errorf("这是%d个格式化参数的错误", 1)
```

错误传递

- 使用 `wrapError` 进行包装
 - 实现了 `error` 接口，用于返回当前当前层的错误
 - 拥有 `Unwrap` 方法，用于返回被包装的下层错误

```
1 type wrapError struct{
2     msg string
3     err error
4 }
5
6 func (e *wrapError) Error() string {
7     return e.msg
8 }
9
10 func (e *wrapError) Unwrap() error {
11     return e.err
12 }
```

- `wrapError` 不对外暴露，需要使用 `fmt.Errorf()` 进行创建
 - 创建时必须使用 `%w` 对错误进行格式化，且参数只能是一个有效的 `err`

```
1 err := errors.New("错误")
2 wrapErr := fmt.Errorf("错误,%w",err)
```

处理

- 使用 `errors.Is` 方法判断错误链中是否包含指定的错误

```
func Is(err, target error) bool
```

- `errors.Unwrap()` 函数用于解包一个错误链

panic

注意：当程序中存在多个协程时，只要任一协程发生 `panic`，如果不将其捕获的话，整个程序都会崩溃

创建

- 使用内置函数 `panic`，签名如下
 - 当输出错误的堆栈信息时，`v` 也会被输出

```
func panic(v any)
```

善后

- 使用 `defer` 进行出现 `panic` 后的善后工作，这里不细说

恢复

- 使用 `recover` 函数进行及时的处理并且保证程序继续运行，且必须要在 `defer` 里运行

```

1 func main() {
2     dangerOp()
3     fmt.Println("程序正常退出")
4 }
5
6 func dangerOp() {
7     defer func() {
8         if err := recover(); err != nil {
9             fmt.Println(err)
10            fmt.Println("panic恢复")
11        }
12    }()
13    panic("发生panic")
14 }

```

条件判断

- `if` 后面必须要有大括号，且不能把 `if` 语句写到同一行
- `if-else` 判断语句没有小括号
- 允许在判断条件之前执行一个简单的语句，用 `;` 隔开，一般用于声明临时变量
- `else` 一定要跟在 `if` 大括号后面
- 使用 `else if` 而不是 `elif`

```

1 // 不合法
2 if v > 10 work()
3 if v > 10{ work() }

4
5 // 合法
6 if v > 10{
7     work()
8 }

9
10 if st:=0 ;v > 10{
11     st = 1
12     work()
13 }else{
14     // ...
15 }

```

switch语句

- `switch` 语句后面不需要括号
- `switch` 的 `case` 可以判断多个值
- `switch` 里面的每个分支结尾自带 `break`

- 可以用 `fallthrough` 关键字强制进入下一个 `case`
- 若 `switch` 变量，则 `case` 后面必须写与变量匹配的值

```

1  switch{
2    case t < 12:
3      fmt.Println(" ")
4    default:
5      fmt.Println(" ")
6  }

```

循环

- `go` 只有 `for` 循环
- `continue` 和 `break` 和其它语言的功能一样

```

1  for{
2    这是一个死循环
3  }
4
5  for j:=7;j < 9;j++{
6    continue
7    break
8  }
9
10 i:=1
11 for i<=3{
12   ++i
13 }

```

defer语句

- `defer`后面必须是函数调用语句
- `defer`后面跟的函数会在外层函数返回之前触发
- 有多个`defer`的时候会按顺序入栈，外层函数返回之后会依次出栈
- `defer`是在`return`之前执行的

```

1  import "fmt"
2  func main(){
3    defer fmt.Println("world")
4
5    fmt.Println("hello")
6  } // 输出 hello world

```

调用defer函数体

- 在最后加`()`，是为了立刻调用这个`defer`函数

```
1  defer func(){
2      if r := recover(); r != nil{
3          msg = "体温异常"
4      }
5  }()
```

Slice

切片，也就是**动态数组**（内存空间动态开辟）

静态数组

- `var 数组名 数组长度 数据类型 {数据}`：定义数组，可以把数据的声明省略
 - 也可以用`:=`定义

```
1  var myArray1 [10]int
2  myArray2 := [10]int
```

- `len(数组名)`：获取数组长度

```
1  var myArray1 [10]int
2
3  for i:=1; i < len(myArray1); i++{
4      fmt.Println(i)
5  }
```

- `range`：可以使用这个关键字迭代数组，获取`index`（索引）和`value`（值）
 - `_`：如果不需要索引或者值，可以使用匿名变量`_`进行忽略

```

1 // 表示固定长度数组
2 var myArray1 [10]int
3
4 // 使用range迭代数组
5 for index,value := range myArray1{
6     fmt.Println("index = ",index,"value=",value)
7 }
8
9 // 使用匿名变量
10 for _,value := range myArray1{
11     fmt.Println("value=",value)
12 }
```

- 对数组进行传参的时候，需要注意：

- 数组是值传递，在函数内部修改数组的时候只修改副本，原数组不变，且声明的形参的数组长度要和传入的数组长度一致

```

1 // 正确
2 func method(arr [5]int){
3
4 }
5
6 // 错误
7 func method(arr [4]int){
8
9 }
10
11 func main(){
12     arr := [5] int
13     method(arr)
14 }
```

- 可以用 `fmt.Println()` 打印数组

动态数组

- 声明切片：**定义数组时不指定元素长度
 - 声明切片并初始化
 - 声明 `nil` 切片，使用 `make` 关键字进行空间分配
 - 第一个参数为数组类型，第二个为元素个数
 - 直接使用 `make` 关键字声明
 - 使用 `:=` 和 `make` 声明

```

1 // 声明切片并初始化
2 slice1 := []int{1,2,3}
3
4 // 声明slice是一个切片，但是并没有给slice分配空间
5 var slice1 []int
6 slice1 = make([]int,3)
7
8 // 直接使用`make`关键字声明
9 var slice1 []int = make([]int,3)
10
11 // 使用:=和make声明
12 var slice1 := make([]int,3)

```

- 传参时传递切片可以**避免拷贝**，因为切片是**引用类型**

```

1 // 避免拷贝
2 func modifySlice(s []int) {
3     s[0] = 100 // 修改会影响原数组
4 }
5
6 func main() {
7     a := []int{1, 2, 3, 4, 5} // 切片（非数组）
8     modifySlice(a)
9     fmt.Println(a[0]) // 输出 100
10 }

```

- `nil` 切片：一个声明但未初始化的切片变量会自动设置为 `nil`，**长度和容量都为0**

```

1 func main() {
2     var phone []int // nil类型切片
3 }

```

• 切片的追加

- 使用 `make` 关键字传参，定义**合法元素数量和切片总空间**
- 可以使用 `append` 关键字进行切片扩容，增加**合法元素数量**，`a = append(a,value)`
 - 也可以使用 `append` 进行**切片对切片的追加**
 - 当切片总空间不足，底层会进行扩容，**扩容一倍**
 - `append` 如果跟了多个独立切片，需要用 ... 解包运算符

```
1 // 声明切片
2 var numbers = make([]int, 3, 5)
3
4 // 扩容
5 numbers = append(numbers, 1)
```

- 从头部插入元素

```
1 nums = append([]int{-1, 0}, nums...)
2 fmt.Println(nums) // [-1 0 1 2 3 4 5 6 7 8 9 10]
```

- 从中间下标i插入元素

```
1 nums = append(nums[:i+1], append([]int{999, 999}, nums[i+1:]...)...)
2 fmt.Println(nums) // i=3, [1 2 3 4 999 999 5 6 7 8 9 10]
```

- 从尾部插入元素

```
1 nums = append(nums, 99, 100)
2 fmt.Println(nums) // [1 2 3 4 5 6 7 8 9 10 99 100]
```

- 切片的截取

- `s[i:]` : 从i切到末尾
- `s[:j]` : 从开头切到j(不含)
- 子切片的底层是**定义了一个新指针指向父切片的某个位置作为子切片的起点，而不是拷贝**
- 可以使用 `copy()` 函数进行**切片的拷贝**
 - `copy(s1, s2)` : 把 `s2` 中的值拷贝给 `s1`

```
1 s := []int{1, 2, 3}
2
3 // s1的值为1, 2
4 s1 = s[0:2]
```

Map

声明Map类型

- `[]` 里面存的是 `key` 的类型，外卖放 `value` 的类型
 - 空间不够会自动扩容
 - 使用 `make` 方法开辟内存空间
 - 使用 `:=` 直接声明

- 声明的时候进行初始化
 - 使用中括号插入键值对
- 可以使用 `key` 和 `value` 直接赋值

```

1 // 声明map
2 var myMap1 map[string]string
3 // 开辟内存空间
4 myMap1 = make(map[string]string,10)
5 // 直接赋值
6 myMap1["one"] = "php"
7 myMap2["two"] = 'js'
8 myMap3["three"] = "go"
9
10 // 直接声明
11 var myMap2 := make(map[int]string,10)
12
13 // 声明的时候初始化
14 myMap3 := map[string]string{
15     "one": "php",
16     "two": "js",
17     "three": "go"
18 }
```

Map的操作

- 遍历：使用 `range` 关键字进行遍历

```

1 myMap3 := map[string]string{
2     "one": "php",
3     "two": "js",
4     "three": "go"
5 }
6
7 for key,value := range myMap3{
8     fmt.Println("key = ",key)
9     fmt.Println("value = ",value)
10 }
```

- 删除：使用 `delete` 关键字进行删除
 - 第一个参数为map的变量名
 - 第二个参数为要删除的键值对的 `key`

```
1 myMap3 := map[string]string{
2     "one": "php",
3     "two": "js",
4     "three": "go"
5 }
6
7 delete(myMap3, "one")
```

- **修改：**直接根据 key 进行修改

```
1 myMap3 := map[string][string]{
2     "one": "php",
3     "two": "js",
4     "three": "go"
5 }
6
7 myMap3["one"] = "python"
```

- 直接进行传参的话， map 类型是**引用传递**

Struct

结构体声明

```
1 type Person struct{
2     Name string
3     Age int
4 }
```

结构体初始化

- 使用 var 关键字，不立刻进行初始化

```
1 var p person
2 p.name = "jhwang"
3 p.age = 20
```

函数传参

- 结构体作为函数参数默认是**值传递**
- 引用传递需要传递结构体地址

```

1 // 值传递
2 func changeStruct(person Person){
3     // ...
4 }
5
6 func main(){
7     var p person
8     p.name = "jhwang"
9     p.age = 20
10    changeStruct(person)
11 }
12
13
14 // 引用传递
15 func changeStruct(person *Person){
16     // ...
17 }
18
19 func main(){
20     var p person
21     p.name = "jhwang"
22     p.age = 20
23     changeStruct(&person)
24 }
```

结构体标签

- 定义结构体时还可以为字段指定一个标记信息
- 一个字段可以有多个**标记信息**，多个标记信息之间用**空格**隔开，标记信息为**键值对**形式，使用````包裹

```

1 type resume struct{
2     Name string `info:name` `doc:我的名字`
3     Sex string `info:sex`
4 }
```

封装

使用结构体来表示类

- 类名称首字母**大小写**都可以，**大写**则表示当前类公有
- 类的属性、方法**大小写**都可以，**大写**则表示当前类的属性、方法**公有**

直接初始化

- 使用`{}`对变量名进行赋值并进行初始化

```
1 type Person struct{
2     Name string
3     Age int
4 }
5 person := Person{name: "Alice", age: 25}
```

实现类方法

- 在方法名前使用 `this` 作为接收者名称
 - `this` 可以看作是调用者别名
 - 默认是值传递

```
1 // 值传递
2 func (this Person) SayHello() {
3     fmt.Printf("Hello, my name is %s\n", this.name)
4 }
5 person.SayHello()
6
7 // 引用传递
8 func (this *Person) SayHello() {
9     fmt.Printf("Hello, my name is %s\n", this.name)
10 }
11 person.SayHello()
```

继承

类的继承

- 在子类的结构体属性中加入父类名
 - 可以直接对父类已有方法进行重写

```
1 type Human struct{
2     name string
3     sex string
4 }
5
6 func (this *Human) Eat(){
7     // ...
8 }
9
10 // 继承
11 type SuperMan struct{
12     Human // SuperMan继承了Human类的方法、属性
13     level int
14 }
15
16 // 重写父类方法
17 func (this *SuperMan) Eat(){
18     // ...
19 }
```

多态

接口

- 使用 `interface` 关键字声明
 - 本质上是一个指针
 - 只要一个类实现了接口定义的**所有方法**, 就自动实现了该接口
 - **类**实现了接口的方法和**类的指针**实现接口的方法是不同的
 - 类实现了接口的方法, 那么**值类型**和**指针类型**都可以赋值给接口
 - 类的指针实现了接口的方法, 那么只有**指针类型**可以赋值给接口

```

1 type Animal interface {
2     Speak()
3 }
4
5 type Cat struct{}
6
7 type Dog struct{}
8
9 // 指针接收者实现方法
10 func (d *Dog) Speak() {
11     fmt.Println("Woof")
12 }
13 // 值接收者实现方法
14 func (c Cat) Speak() {
15     fmt.Println("Meow")
16 }
17
18 func main() {
19     var a Animal
20
21     // 值类型和指针类型均可赋值
22     a = Cat{}      // 合法
23     a = &Cat{}    // 也合法 (Go 自动解引用)
24     a = &Dog{}    // 合法
25     a = Dog{}     // 编译错误: Dog 未实现 Animal (缺少 *Dog 的方法)
26

```

- `interface{}` (空接口) 可以存储任意类型的值, 是万能容器

```

1 type iface struct {
2     tab *itab          // 类型信息
3     data unsafe.Pointer // 实际数据的指针
4 }

```

多态

- 使用接口声明, 实现接口的类定义
- 可以定义一个方法, 使用接口声明形参, 实现了接口的类都可以调用这个方法

```
1 type AnimalIF interface{
2     Sleep()
3     GetColor() string
4     GetType() string
5 }
6
7 func ShowAnimal(animal AnimalIF){
8     // ...
9 }
10
11 // 实现接口的类
12 type Cat struct{
13     color string
14 }
15
16 // 实现接口
17
18 func (this *Cat) Sleep(){
19     // ...
20 }
21
22 func (this *Cat) GetColor() string{
23     // ...
24 }
25
26 func (this *Cat) GetType() string{
27     // ...
28 }
29
30 type Dog struct{
31     color string
32 }
33
34
35 func (this *Dog) Sleep(){
36     // ...
37 }
38
39 func (this *Dog) GetColor() string{
40     // ...
41 }
42
43 func (this *Dog) GetType() string{
44     // ...
45 }
```

```

47 func main(){
48     // 声明接口
49     var animal AnimalIF
50
51     // 实现多态
52     animal = &Cat{"green"}
53     animal.Sleep()
54     fmt.Println(animal.GetColor())
55     fmt.Println(animal.GetType())
56
57     // 实现多态
58     animal = &Dog{"blue"}
59     animal.Sleep()
60     fmt.Println(animal.GetColor())
61     fmt.Println(animal.GetType())
62 }
```

通用万能类型

- 使用空接口来表示通用万能类型
- **类型断言**: 使用 `x.(T)` 判断 `x` 是不是和 `T` 的类型一样
 - 检查接口变量的**动态类型**是否满足目标接口, 即如果 `T` 是接口类型, 断言检查 `x` 的动态类型是否满足 `T` 接口 (`x` 是否实现接口 `T`)
 - 变量名一定要是**空接口类型**
 - 返回 `value` 和 `ok`
 - 类型相同 `ok` 为 `true`, `value` 为变量名的值

```

1 // 使用空接口来表示通用万能类型
2 func MyFunc(arg interface{}){
3     // ...
4     // 使用类型断言
5     value,ok = arg.(string)
6 }
7
8 type book struct{
9     // ...
10 }
11
12 func main(){
13     book := Book{}
14     // 函数能够正确识别book类型
15     MyFunc(book)
16 }
```

反射

变量构造 (pair)

Go中的每个变量，在底层都是一个 `(type,value)` 对

- 变量类型: `type`
 - 静态类型: `static type`，**声明**时就能确定的类型
 - 具体类型: `concrete type`，**运行**时才能确定的类型
- 变量值: `value`
- `pair` 会连续不断地传递，且**不会变化**

```
1 var a string
2 a = "aceld"
3
4 var allType interface{}
5 // allType里面的value和type和a的一样
6 allType = a
```

反射

用于接口 `interface`

- 需要导入 `reflect` 库
 - 使用 `ValueOf()` 返回传入的数据的值
 - 使用 `TypeOf()` 返回传入的数据的类型
- 对于简单和复杂数据类型都可以使用
- 对于复杂数据类型
 - 先获得**输入类型**
 - 使用 `.NumField()` 方法获得参数个数
 - 使用 `.Field()` 方法获得参数类型
 - 使用 `.Field().Interface()` 方法获得参数值
 - 使用 `.NumMethod()` 方法获得方法个数
 - 使用 `.Method()` 方法获得方法信息

```

1 func reflectNum(arg interface{}){
2     fmt.Println("Type=",reflect.Typeof(arg))
3     fmt.Println("Value=",reflect.Valueof(arg))
4 }
5
6 func main(){
7     var num float64 = 3.14
8     reflectNum(num)
9 }
10
11 // 反射
12 func DoFiledAndMethod(input interface{}) {
13     // 获取类型信息
14     inputType := reflect.TypeOf(input)
15     fmt.Println("inputType is :", inputType.Name())
16     // 获取值信息
17     inputValue := reflect.ValueOf(input)
18     fmt.Println("inputValue is:", inputValue)
19
20     // 遍历字段
21     for i := 0; i < inputType.NumField(); i++ {
22         field := inputType.Field(i)           // 获取字段定义信息
23         value := inputValue.Field(i).Interface() // 获取字段实际值
24
25         fmt.Printf("%s: %v=%v\n", field.Name, field.Type, value)
26     }
27
28     // 遍历方法
29     for i := 0; i < inputType.NumMethod(); i++ {
30         m := inputType.Method(i)
31         fmt.Printf("%s: %v\n", m.Name, m.Type)
32     }
33 }
```

反射获取结构体标签

- 使用 `.Field().Tag.Get("标签的key")` 获得字段标签
- `.Elem()` 方法用于获取指针、数组、切片、映射、通道或接口所指向的元素的类型

```

1 type resume struct{
2     Name string `info:name` `doc:我的名字`
3     Sex string `info:sex`
4 }
5
6 func findTag(str interface{}){
7     t := reflect.TypeOf(str).Elem()
8
9     for i:=0 ;i < t.NumField();i++{
10         taginfo = t.Field(i).Tag.Get("info")
11         tagdoc = t.Field(i).Tag.Get("doc")
12     }
13 }
```

结构体标签

将结构体标签转换为json格式

- 导入包: `encoding/json`
- 定义结构体标签
 - `key` 固定为 `json`
 - `value` 为 `json` 格式的 `key`
- 使用 `json.Marshal()` 方法传入结构体转换成 `json` 字符串
 - 返回 `json` 字符串和错误码
 - 发生错误时错误码不为空
- 使用 `json.Unmarshal()` 方法把 `json` 字符串转换为结构体
 - 需要传入结构体地址和 `json` 字符串
 - 返回错误码

```

1 import "encoding/json"
2
3 type Movie struct{
4     Title string `json:"title"`
5     Year int `json:"year"`
6 }
7
8 func main(){
9     movie := Movie{"喜剧之王",2000}
10    jsonStr,err = json.Marshal(movie)
11
12    movie := Movie{}
13    err = json.Unmarshal(jsonStr,&movie)
14
15 }
```

goroutine (协程)

多线程多进程操作系统

- 解决了阻塞问题，线程A阻塞，CPU可以切换到线程B
- 但是CPU利用率不高
 - CPU需要在每个线程之间进行切换，**切换成本高**

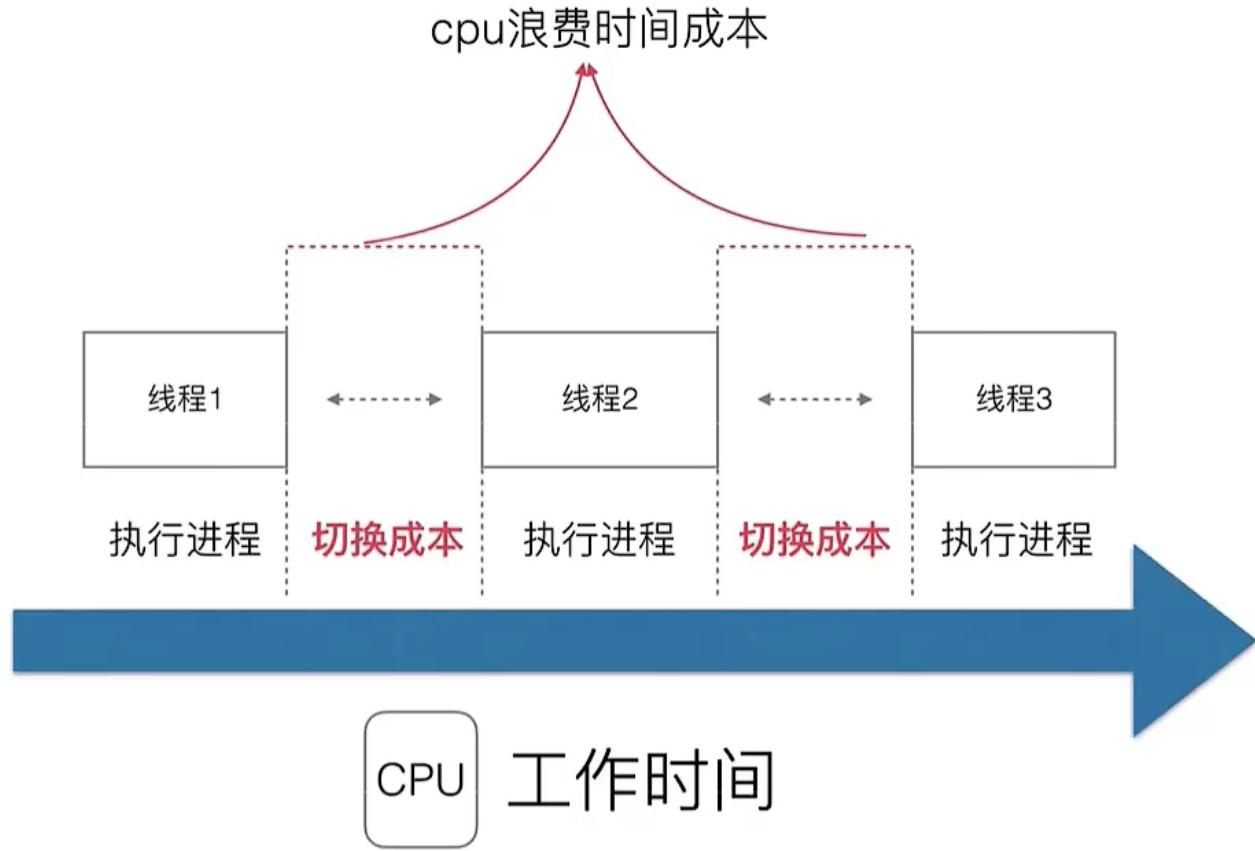


image-20251015154123809

因此协程应运而生

goroutine与系统线程的区别

- goroutine 创建与销毁的开销较小
- goroutine 的调度发生在**用户态**（轻量级的线程），**切换成本低**；系统线程的调度发生在**内核态**，**切换成本高**
- goroutine 的通信可通过 `channel` 完成，系统线程通信依赖**共享内存和锁机制**

协程的调度模型

- N:M 模型
- N个操作系统的线程通过协程调度器和M个协程进行通信
 - **N个线程**是操作系统调度的实体，**M个协程**是用户态任务
 - 协程调度器负责在**M个协程**之间切换，但它们运行在**N个线程**上

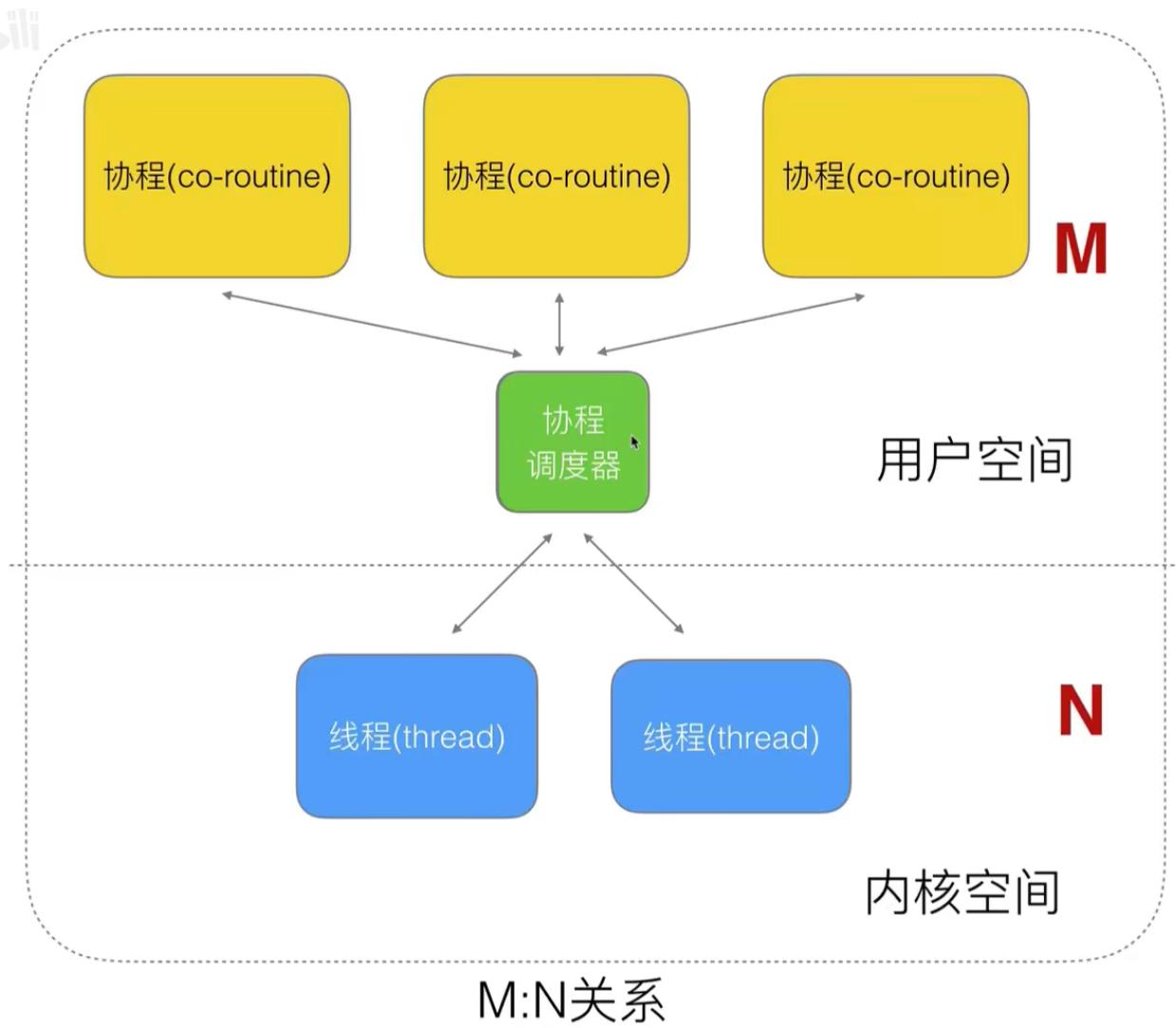


image-20251015122659139

Go的GMP模型

- G: goroutine 协程
- M: 操作系统内核的 thread 线程
- P: 协程处理器

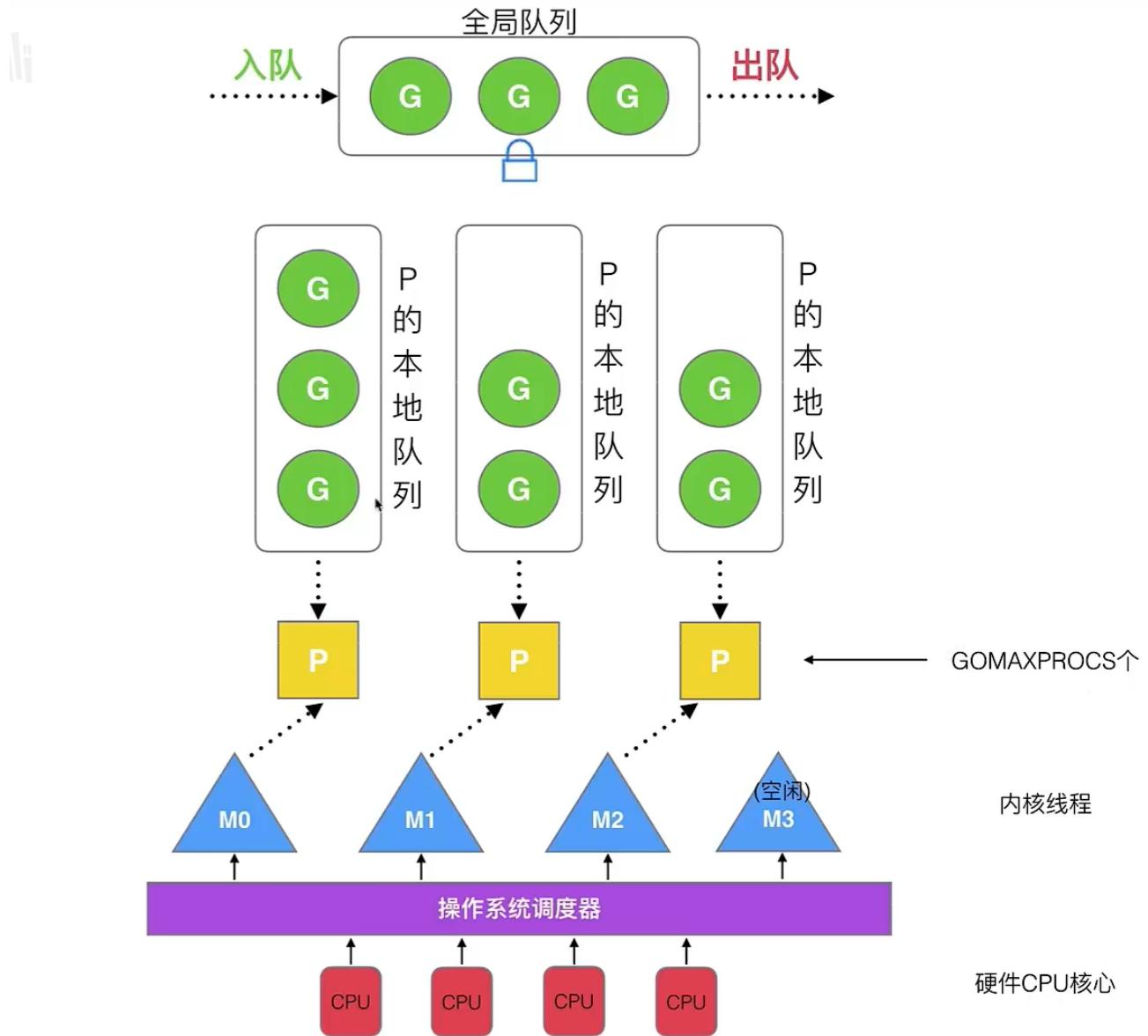


image-20251015160120180

解读

- 可以通过设置 `GOMAXPROCS` 来调整协程处理器的个数
- 每个 `P` 拥有一个 **本地队列 (LRQ)**
- 所有 `P` 共享一个 **全局队列 (GRQ)**
 - 当 `P` 的本地队列满了之后，新创建的协程会放进全局队列中

设计策略

- 复用线程： `work stealing` 机制， `hand off` 机制
 - `work stealing` 机制：当某一个 `thread` 空闲时，会去别的 `Processor` 的本地队列偷取一批协程执行
 - `hand off` 机制：当某一个 `thread` 执行协程遇到阻塞时，会唤醒一个 `thread`，将被阻塞的 `thread` 的 `Processor` 的本地队列交给被唤醒的 `thread`

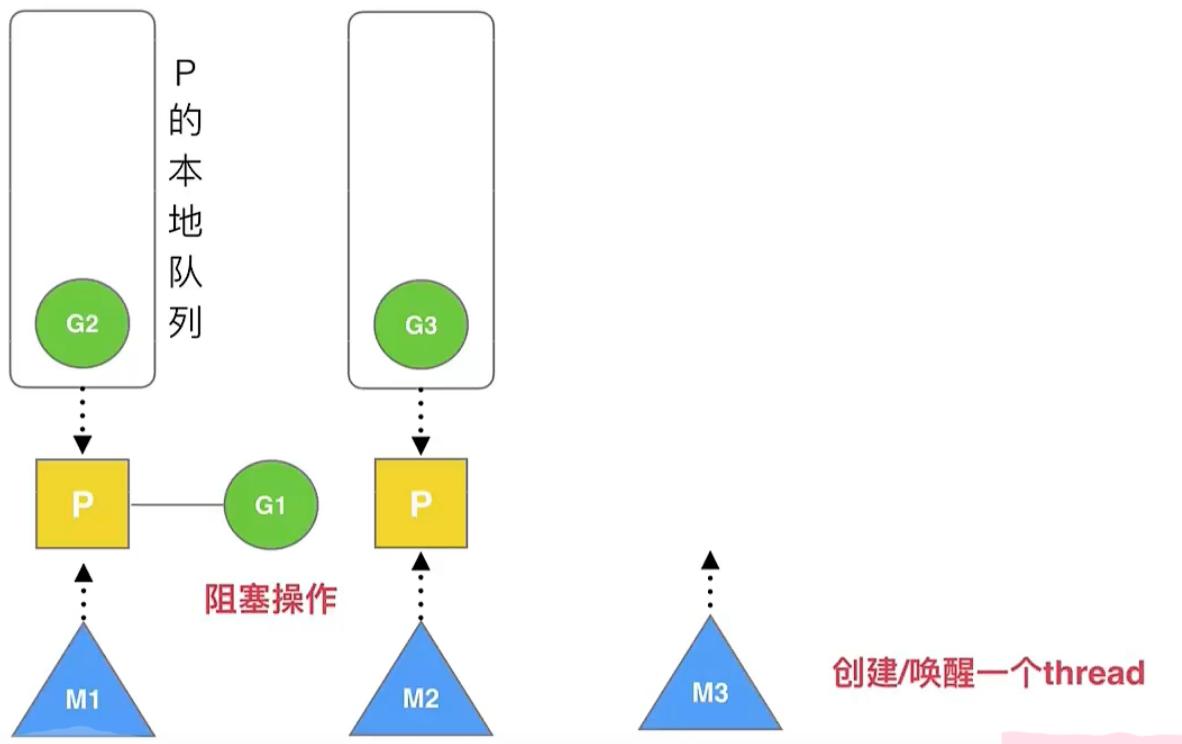


image-20251015161152465

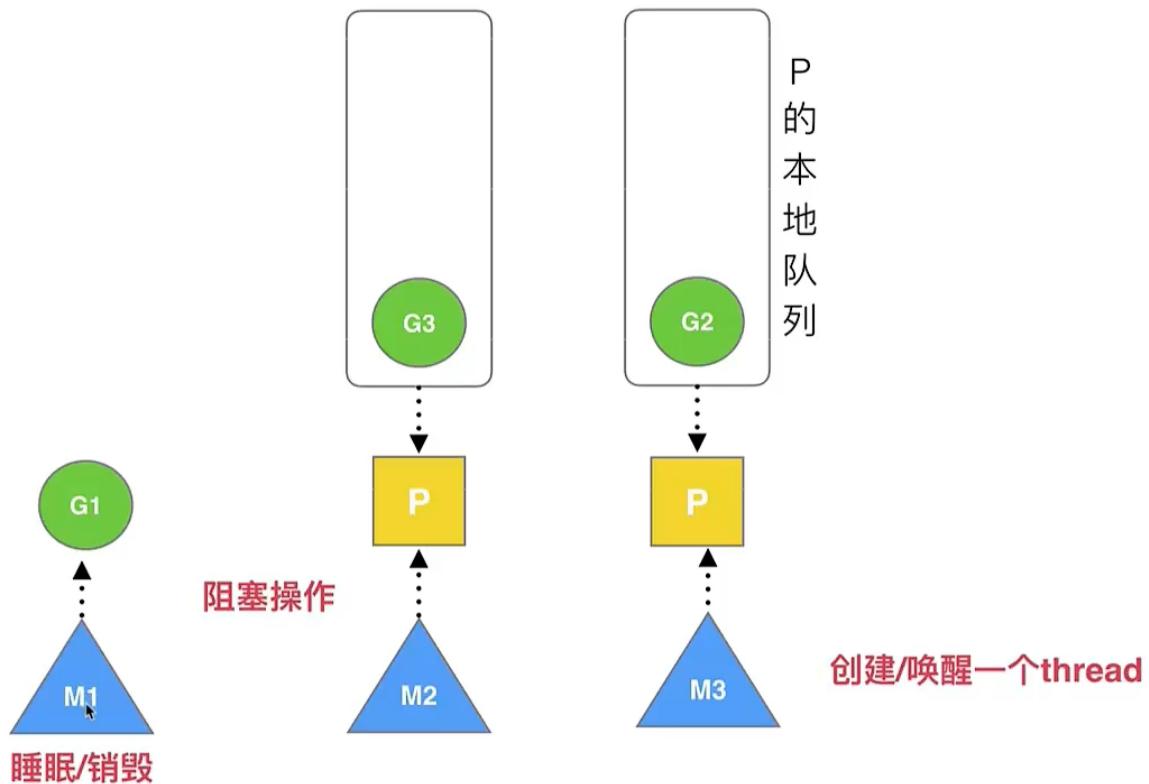


image-20251015161111010

- 利用并行：通过 `GOMAXPROCS` 限定P的个数，一般约定为CPU核数/2
- 抢占：当 `thread` 和某个 `goroutine` 绑定，且当前 `thread` 被阻塞，此时只允许 `thread` 等待一定时间，超过这个时间 `thread` 就会分配给其它在等待的 `goroutine`

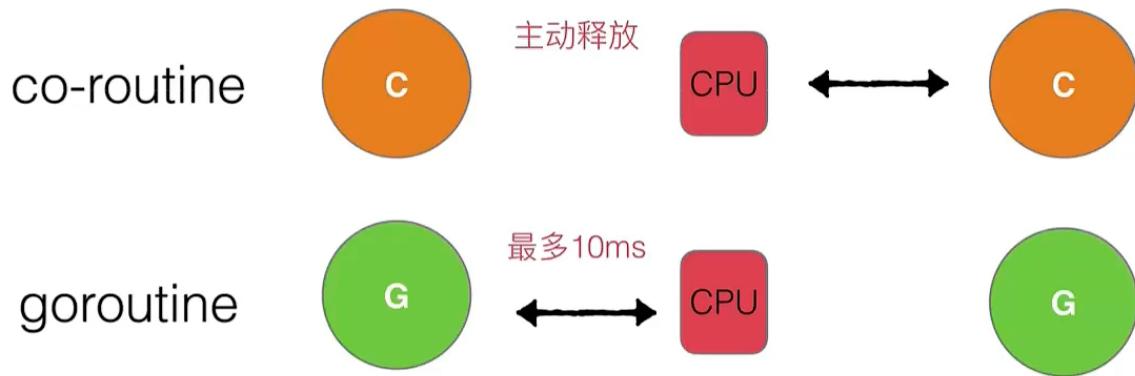


image-20251015165748812



image-20251015165732289

- **全局G队列**: 拥有锁的机制
 - 当 `thread` 空闲且其它 `thread` 也没有待处理的协程时，`thread` 就会去全局队列获取协程
 - **全局队列 (GRQ)** 需要加锁访问，频繁竞争会影响性能。因此Go优先通过P的**本地队列 (LRQ)** 和 `Work Stealing` 实现无锁调度，仅在 `LRQ` 不足时使用 `GRQ`

创建goroutine

在方法前加 `go` 关键字

- `main` 方法是主 `goroutine`，自定义方法是从 `goroutine`
 - `main` 方法退出时其它从 `goroutine` 会死亡

```
1 func newTask(){
2     i := 0
3     for{
4         i++
5             fmt.Printf("Hello")
6     }
7 }
8
9 func main(){
10    go newTask()
11 }
```

- 直接创建 `go` 协程并执行

- 创建形参为空，返回值为空的匿名函数

- 匿名函数需要在代码后面加上 `()`，告诉编译器立即执行
 - 在代码后加上括号，不填形参
 - 在 `go` 协程里面再创建匿名函数，可以使用 `runtime.Goexit()` 方法退出当前 goroutine

- 创建形参不为空，返回值不为空的匿名函数

- 在代码后加上括号，填入形参
 - 返回值需要通过 `channel` 拿到

```

1 func main(){
2     // 1.
3     go func() {
4         defer fmt.Println("A.defer")
5
6         func() {
7             defer fmt.Println("B.defer")
8             runtime.Goexit() // 退出当前goroutine
9             fmt.Println("B") // 这行不会执行
10        }()
11
12        fmt.Println("A") // 这行不会执行
13    }()
14
15    // 2.
16    go func(a int, b int) bool {
17        fmt.Println("a =", a, ", b =", b)
18        return true
19    }(10, 20)
20
21
22    for{
23        // ...
24    }
25}

```

Channel

常见方法

- `c:=make(chan int)` : 创建channel, 传递的数据类型是 `int`
- `channel <- value` : 发送value到channel, 默认传递引用
- `<- channel` : 接收并丢弃
- `x,ok := <-channel` : 从channel读取数据并赋值给 `x` , `ok` 检查管道是否为已经关闭

```

1 func main(){
2     c := make(chan int)
3
4     go func(){
5         c <- 666
6     }()
7
8     num := <- c
9
10 }

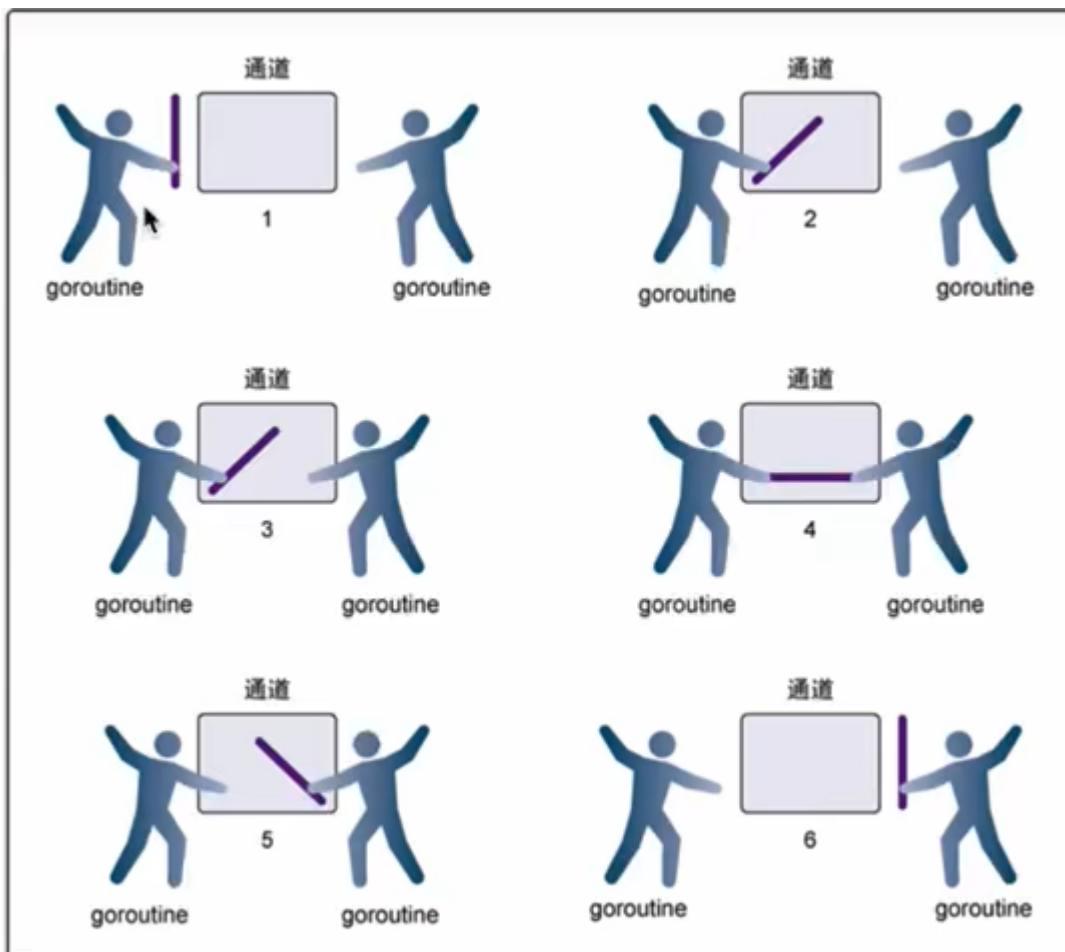
```

- `num := <- c` 和 `c <- 666` 是同步执行的，因此不能确定谁先谁后
 - 当 `num := <- c` 先执行时，对应的 `thread` 会进行阻塞，等待666的传入
 - 当 `c <- 666` 先执行时，要把666写入到channel，但是channel无缓冲，因此对应的 `thread` 也会进行阻塞，直到执行 `num := <- c`

无缓冲的channel和有缓冲的channel

无缓冲

- 传数据的 `goroutine` 必须等待拿数据的 `goroutine` 把手伸进来，否则阻塞

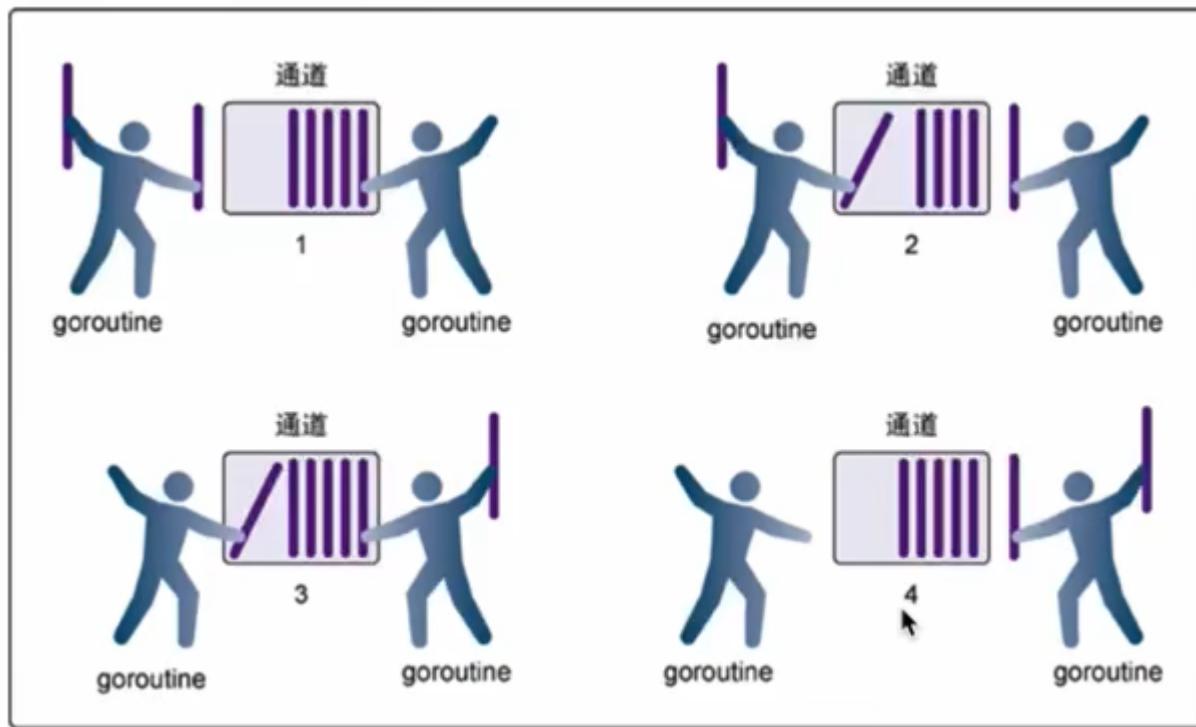


使用无缓冲的通道在 `goroutine` 之间同步

image-20251015173754231

有缓冲

- 传数据的 goroutine 只需要把数据放到通道，读数据的 goroutine 只需要从通道拿数据
- 当通道空了或者通道满了，协程才会阻塞



使用有缓冲的通道在 goroutine 之间同步数据

image-20251015173810451

创建有缓冲的channel

- 使用 `make(chan,int,3)` 方法创建通道，3表示通道容量
 - 使用 `len(c)` 获取通道元素数量
 - 使用 `cap(c)` 获取通道容量

```
1 func main(){
2     c := make(chan int,3)
3 }
```

channel的关闭特点

- 使用 `close(chan)` 可以关闭一个协程
- `x,ok := <-channel`：从channel读取数据并赋值给 `x`， `ok` 检查管道是否为已经关闭
- 确认已经没有数据发送之后，要把channel进行关闭，否则读取数据的协程会发生死锁
- 注意：对于有缓冲channel，关闭channel之后仍然可以从channel中接收数据

channel和range

- 使用 `range` 关键字从管道获取数据

```
1 c := make(chan int, 3)
2
3 for data := range c{
4     fmt.Println(data)
5 }
```

channel和select

- 在同一协程下监控多个 channel
- 使用 select 定义多个 case，哪个 case 先触发就会用哪个 case 的处理语句

```
1 select{
2     case <- chan1:
3         // 如果channel1读取到数据，就执行此case处理语句
4     case chan2 <- 1:
5         // 如果成功向channel2写入数据，就执行此case处理语句
6     default:
7         // 如果以上都没有成功，进入default处理流程
```

GoModules

是Go语言的依赖解决方案，解决了**依赖管理问题**

GoPath的弊端

- 没有版本控制概念
- 无法同步一致第三方版本号
- 无法指定当前项目引用的第三方版本号

go mod命令

- go mod init : 生成 go.mod 文件
 - 后面跟上**模块名称**
- go mod download : 下载 go.mod 文件中的所有依赖
- go mod tidy : 整理现有的依赖
- go mod graph : 查看所有的依赖结构
- go mod edit : 编辑 go.mod 文件
 - go : 修改go版本
 - require : 添加依赖
 - droprequire : 移除依赖
 - replace : 替换依赖
 - exclude : 排除版本
- go mod vendor : 导出项目所有的以爱到 vendor 目录
- go mod verify : 检查一个模块是否被篡改过

go mod环境变量

- `G0111MODULE` : 用来控制 Go modules 的开关
 - `auto` : 只要项目包含了 `go.mod` 文件的话就启用 Go modules
 - `on` : 启用 Go modules
 - `off` : 禁用 Go modules

可使用环境变量设置

```
go env -w G0111MODULE=on
```

- `GOPROXY` : 设置Go模块的代理，在后续拉取模块版本时直接通过镜像站点拉取
 - 默认值为 `https://proxy.golang.org,direct`

如：

```
go env -w GOPROXY=https://goproxy.cn.direct
```

- `GOSUMDB` : 拉取模块版本时检验代码是否经过篡改
 - 默认值为 `sum.golang.org`
 - 设置了 `GOPROXY` 可以不用管这个
- `GONOPROXY/GONOSUMDB/GOPRIVATE` : 用于管理私有模块行为的关键配置，即**不走代理、不进行校验和检查**
 - 直接使用 `GOPRIVATE`，它的值会作为 `GONOSUMDB` 和 `GONOPROXY` 的默认值
 - 可以设置多个模块，多个模块以英文逗号分隔

go.mod文件

```
1 module github.com/yourname/project // 模块路径 (必填)
2
3 go 1.21 // 最低要求的 Go 版本 (必填)
4
5 require ( // 直接依赖列表
6     github.com/gin-gonic/gin v1.9.1
7     golang.org/x/sync v0.3.0
8 )
9
10 replace ( // 替换依赖源 (可选)
11     golang.org/x/sync => ./local/sync // 本地替换
12 )
13
14 exclude ( // 排除特定版本 (可选)
15     github.com/old/lib v1.2.3
16 )
17
18 retract ( // 撤回发布的版本 (可选)
19     v1.0.0 // 严重漏洞
20 )
```

go.sum文件

- 罗列当前项目直接或间接的依赖所有模块的版本，保证今后项目依赖的版本不会被篡改
- 会生成一个哈希值用来进行校验