

Go

xbZhong

2025-10-11

Contents

Golang 的优势	1
Golang 基本语法	2
goroutine (协程)	20
GoModules	29

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

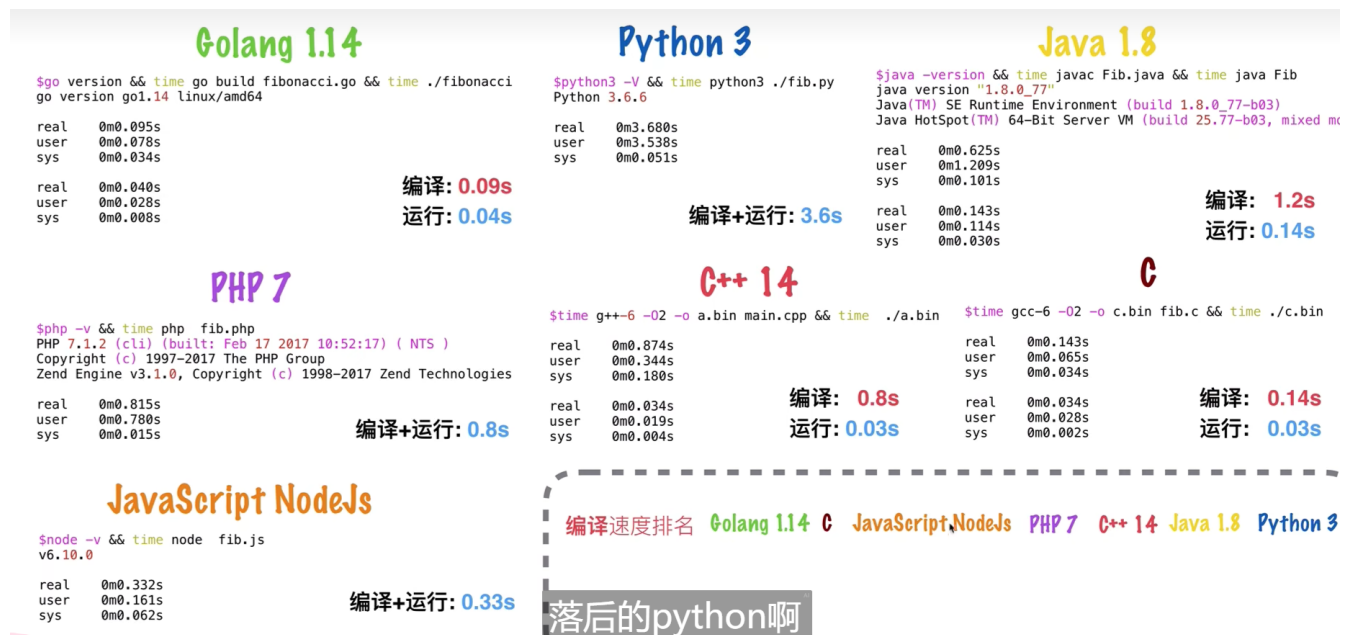


Figure 1: image-20251011122440219

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

```
package main // 声明包

import "fmt" // 导入包

// 导多个包
import(
```

```

    "fmt"
    "time"
)

// 主函数
func main(){
    fmt.Println("Hello World")
}

```

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

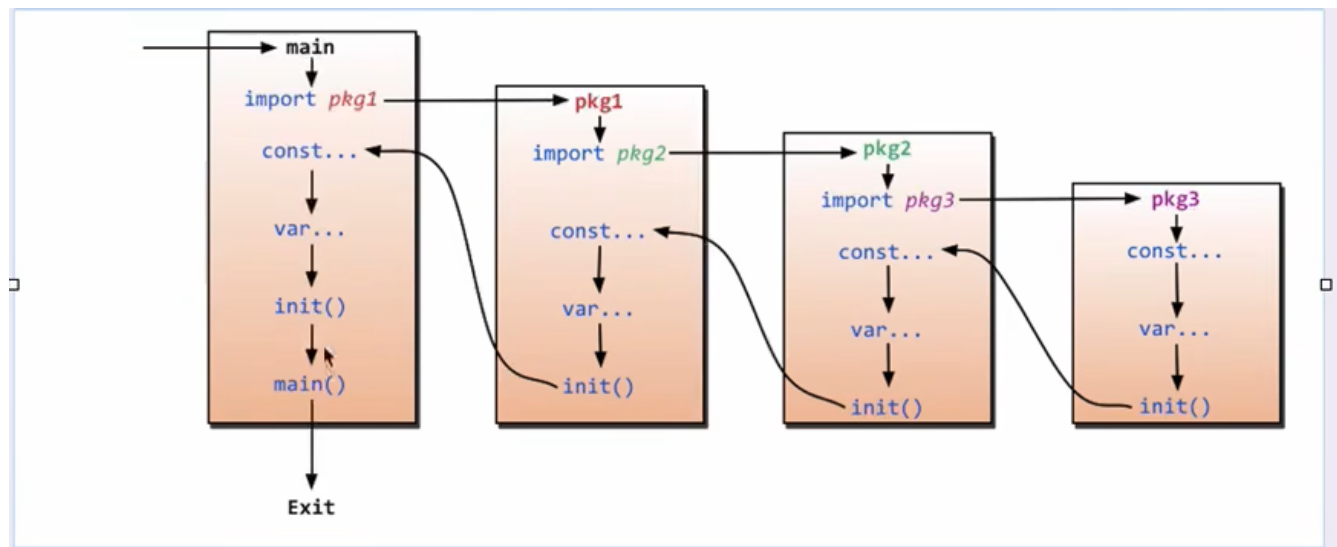


Figure 2: image-20251011171819781

- 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

```

package main

import(
    _ "./lib1"
    mylib2 "./lib2"
    . "./lib3"
)

func main(){
    // 用别名启用方法
    mylib2.Lib2Test()

    // 直接调用方法
    Lib3Test()
}

```

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

基本数据类型

```

// 布尔类型
bool

// 字符串类型
string

// 整数类型
int int8 int16 int32 int64
uint uint8 uint16 uint32 uint64 uintptr

// uint8 的别名，即一个字节
byte

// int32 的别名，表示一个 Unicode 字符，常用来表示单个字符
rune

// 浮点类型
float32 float64

// 复数类型
complex64 complex128

// 字符串类型，使用"" 或者``表示
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

```
func main(){  
    // 方式一  
    var a int  
    // 方式二  
    var b int = 100  
    // 声明多个变量  
    var b,c int = 1,2  
    var bb,cc = 100,"zxb"  
    var(  
        bbb int = 100  
        ccc string = "zxb"  
    )  
    // 方式三  
    var c = 100  
    // 方法四：省略 var 关键字  
    e := 100  
}
```

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

```
const(
    BEIJING = 10*iota // iota=0
    SHANGHAI      // iota=1
    SHENZHEN      // iota=2
)
func main(){
    const length int = 10
}
```

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

```
import "fmt"

// 示例
func fool1(a string,b int) int{
    fmt.Println("a = ",a)
    fmt.Println("b = ",b)

    c := 100
    return c
}

// 返回多个返回值
func fool1(a string,b int) (int,int){
    return 666,777
}
```

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

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

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

```
package main

import "fmt"

func add(n int) {
    n += 2
}

func addptr(n *int){
    // 此时 n 里面存的是 p 的地址
    *n += 2
}

func main(){
    p := 5
    add(p)
    fmt.Println(p) // p = 5
    addptr(&p)
    fmt.Println(p) // p = 7
}
```

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

```
// 不合法
if v > 10 work()
if v > 10{ work() }

// 合法
if v > 10{
    work()
}

if st:=0 ;v > 10{
    st = 1
    work()
}
```

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

```
switch{
case t < 12:
    fmt.Println("")
default:
    fmt.Println("")
}
```

循环

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

```
for{
    这是一个死循环
}

for j:=7;j < 9;j++){
    continue
    break
}

i:=1
for i<=3{
    ++i
}
```


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 之后执行的**

```
import "fmt"
func main(){
    defer fmt.Println("world")

    fmt.Println("hello")
} // 输出 hello world
```

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

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

- len(数组名)：获取数组长度

```
var myArray1 [10]int

for i:=1; i < len(myArray1); i++){
    fmt.Println(i)
}
```

- 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

```
// 表示固定长度数组
var myArray1 [10]int

// 使用 range 迭代数组
for index,value := range myArray1{
    fmt.Println("index = ",index,"value=",value)
}

// 使用匿名变量
for _,value := range myArray1{
    fmt.Println("value=",value)
}
```

- 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**

```
// 正确
func method(arr [5]int){

}

// 错误
func method(arr [4]int){

}

func main(){
    arr := [5] int
    method(arr)
}
```

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

```
// 声明切片并初始化
slice1 := [int]{1,2,3}

// 声明 slice 是一个切片，但是并没有给 slice 分配空间
var slice1 []int
slice1 = make([]int,3)

// 直接使用`make`关键字声明
var slice1 []int = make([]int,3)

// 使用:= 和 make 声明
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**

```
// 避免拷贝
func modifySlice(s []int) {
    s[0] = 100 // 修改会影响原数组
}

func main() {
    a := []int{1, 2, 3, 4, 5} // 切片（非数组）
    modifySlice(a)
    fmt.Println(a[0]) // 输出 100
}
```

- 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**

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

• 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**

```
// 声明切片
var numbers = make([]int,3,5)

// 扩容
numbers = append(numbers,1)
```

• 切片的截取

- s[i:]: 从 i 切到末尾
- s[:j]: 从开头切到 j(不含 j)
- 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

```
s := []int{1,2,3}

// s1 的值为 1, 2
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
 - 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

```
// 声明 map
var myMap1 map[string]string
// 开辟内存空间
myMap1 = make(map[string][string],10)
// 直接赋值
myMap1["one"] = "php"
myMap2["tow"] = 'js'
myMap3["three"] = "go"

// 直接声明
var myMap2 := make(map[int]string,10)
```

```
// 声明的时候初始化
myMap3 := map[string]string{
    "one": "php",
    "two": "js",
    "three": "go"
}
```

Map 的操作

- **遍历**：使用 range 关键字进行遍历

```
myMap3 := map[string]string{
    "one": "php",
    "two": "js",
    "three": "go"
}

for key, value := range myMap3{
    fmt.Println("key = ", key)
    fmt.Println("value = ", value)
}
```

- **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

```
myMap3 := map[string]string{
    "one": "php",
    "two": "js",
    "three": "go"
}

delete(myMap3, "one")
```

- **修改**：直接根据 key 进行修改

```
myMap3 := map[string]string{
    "one": "php",
    "two": "js",
    "three": "go"
}

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

```
type Person struct{
    Name string
    Age int
}
```

结构体初始化

- 使用 var 关键字，不立刻进行初始化

```
var p person
p.name = "jhwang"
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

```
// 值传递
func changeStruct(person Person){
    // ...
}
```

```
func main(){
    var p person
    p.name = "jhwang"
    p.age = 20
    changeStruct(person)
}
```

```
// 引用传递
func changeStruct(person *Person){
    // ...
}
```

```
func main(){
    var p person
    p.name = "jhwang"
    p.age = 20
    changeStruct(&person)
}
```

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

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

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

```
type Person struct{
    Name string
    Age  int
}
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

```
// 值传递
func (this Person) SayHello() {
    fmt.Printf("Hello, my name is %s\n", this.name)
}
person.SayHello()

// 引用传递
func (this *Person) SayHello() {
    fmt.Printf("Hello, my name is %s\n", this.name)
}
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

```
type Human struct{
    name string
    sex  string
}

func (this *Human) Eat(){
    // ...
}

// 继承
type SuperMan struct{
    Human // SuperMan 继承了 Human 类的方法、属性
```

```

    level int
}

// 重写父类方法
func (this *SuperMan) Eat(){
    // ...
}

```

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

```

type Animal interface {
    Speak()
}

type Cat struct{}

type Dog struct{}

// 指针接收者实现方法
func (d *Dog) Speak() {
    fmt.Println("Woof")
}

// 值接收者实现方法
func (c Cat) Speak() {
    fmt.Println("Meow")
}

func main() {
    var a Animal

    // 值类型和指针类型均可赋值
    a = Cat{}           // 合法
    a = &Cat{}          // 也合法 (Go 自动解引用)
    a = &Dog{}           // 合法
    a = Dog{}           // 编译错误: Dog 未实现 Animal (缺少 *Dog 的方法)
}

```

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

```
type AnimalIF interface{
    Sleep()
    GetColor() string
    GetType() string
}

func ShowAnimal(animal AnimalIF){
    // ...
}

// 实现接口的类
type Cat struct{
    color string
}

// 实现接口

func (this *Cat) Sleep(){
    // ...
}

func (this *Cat) GetColor() string{
    // ...
}

func (this *Cat) GetType() string{
    // ...
}

type Dog struct{
    color string
}

func (this *Dog) Sleep(){
    // ...
}

func (this *Dog) GetColor() string{
    // ...
}

func (this *Dog) GetType() string{
    // ...
}

func main(){
```



```
// 声明接口
var animal AnimalIF

// 实现多态
animal = &Cat{"green"}
animal.Sleep()
fmt.Println(animal.GetColor())
fmt.Println(animal.GetType())

// 实现多态
animal = &Dog{"blue"}
animal.Sleep()
fmt.Println(animal.GetColor())
fmt.Println(animal.GetType())
}
```

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

```
// 使用空接口来表示通用万能类型
func MyFunc(arg interface{}){
    // ...
    // 使用类型断言
    value,ok = arg.(string)
}

type book struct{
    // ...
}

func main(){
    book := Book{}
    // 函数能够正确识别 book 类型
    MyFunc(book)
}
```

反射

变量构造 (pair)

- 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

```
var a string
a = "aceld"

var allType interface{}
// allType 里面的 value 和 type 和 a 的一样
allType = a
```

反射

- 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

```
func reflectNum(arg interface{}){
    fmt.Println("Type=",reflect.TypeOf(arg))
    fmt.Println("Value=",reflect.Valueof(arg))
}

func main(){
    var num float64 = 3.14
    reflectNum(num)
}

// 反射
func DoFiledAndMethod(input interface{}) {
    // 获取类型信息
    inputType := reflect.TypeOf(input)
    fmt.Println("inputType is :", inputType.Name())
    // 获取值信息
    inputValue := reflect.ValueOf(input)
    fmt.Println("inputValue is:", inputValue)

    // 遍历字段
    for i := 0; i < inputType.NumField(); i++ {
        field := inputType.Field(i)           // 获取字段定义信息
        value := inputValue.Field(i).Interface() // 获取字段实际值

        fmt.Printf("%s: %v=%v\n", field.Name, field.Type, value)
```

```

}

// 遍历方法
for i := 0; i < inputType.NumMethod(); i++ {
    m := inputType.Method(i)
    fmt.Printf("%s: %v\n", m.Name, m.Type)
}
}

```

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

```

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

func findTag(str interface{}){
    t := reflect.TypeOf(str).Elem()

    for i:=0 ;i < t.NumField();i++){
        taginfo = t.Field(i).Tag.Get("info")
        tagdoc = t.Field(i).Tag.Get("doc")
    }
}

```

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

```

import "encoding/json"

type Movie struct{
    Title string `json:"title"`
    Year int `json:"year"`
}

```

```
func main(){
    movie := Movie{" 喜剧之王",2000}
    jsonStr,err = json.Marshal(movie)

    movie := Movie{}
    err = json.Unmarshal(jsonStr,&movie)
}
```

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**

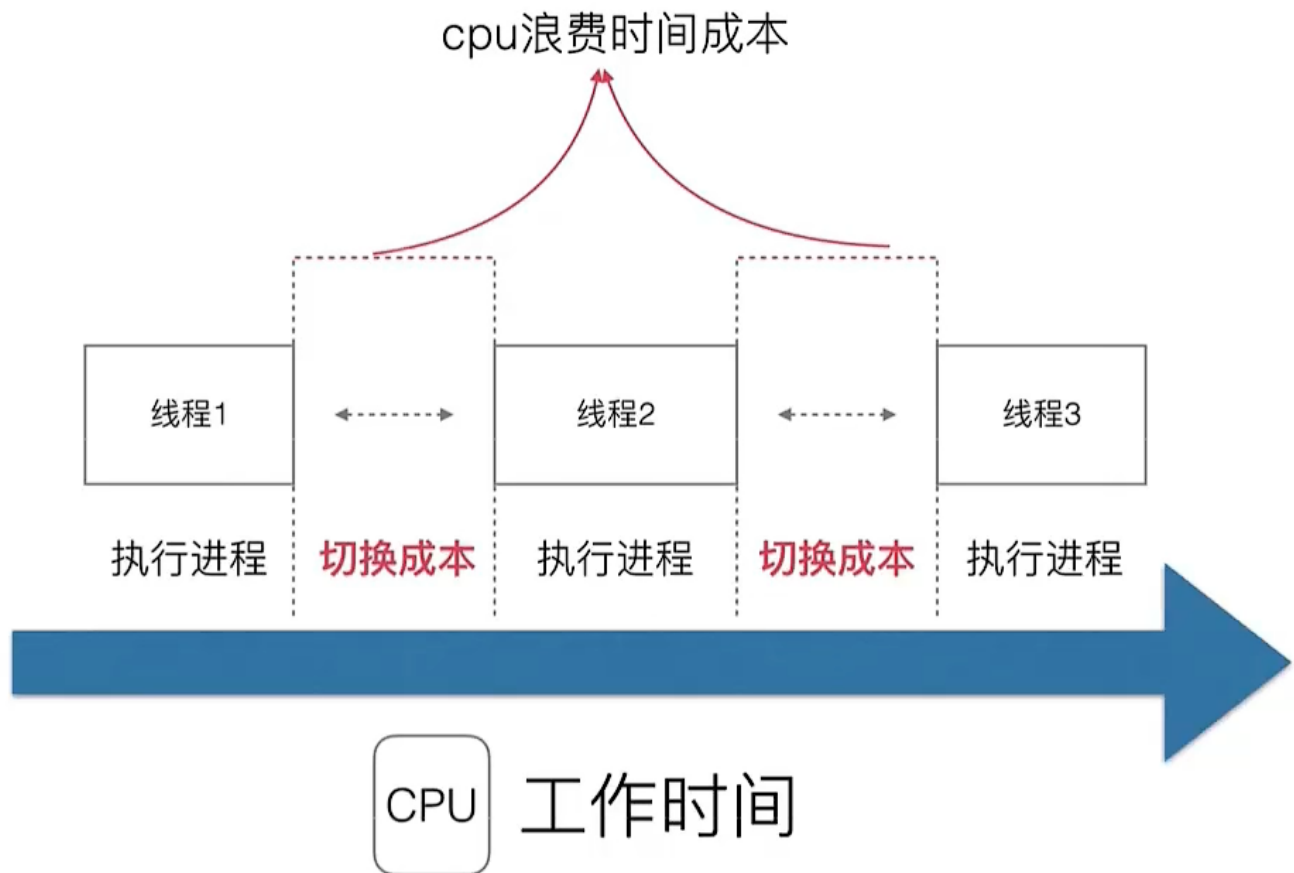


Figure 3: image-20251015154123809

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

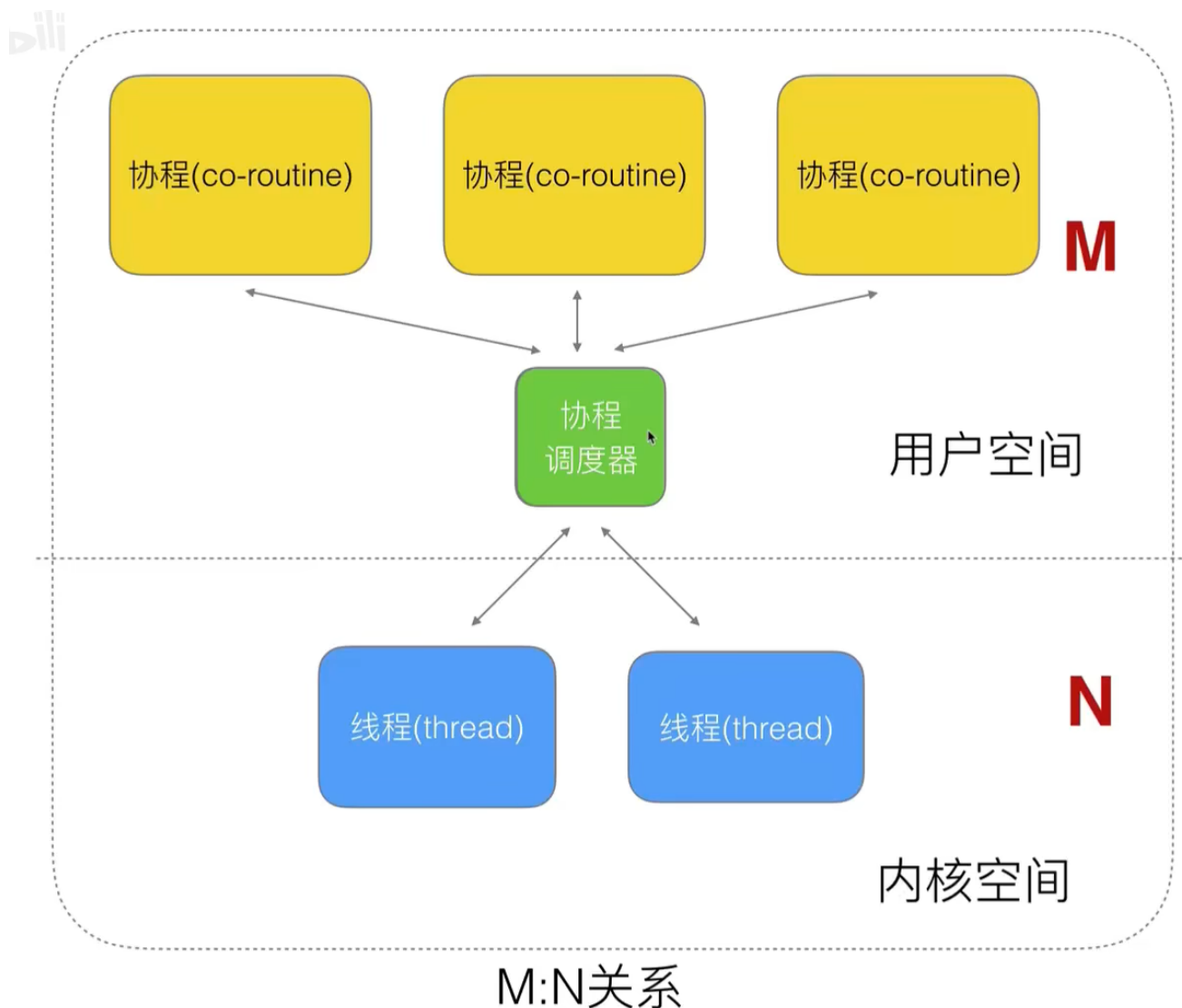


Figure 4: image-20251015122659139

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**

解读

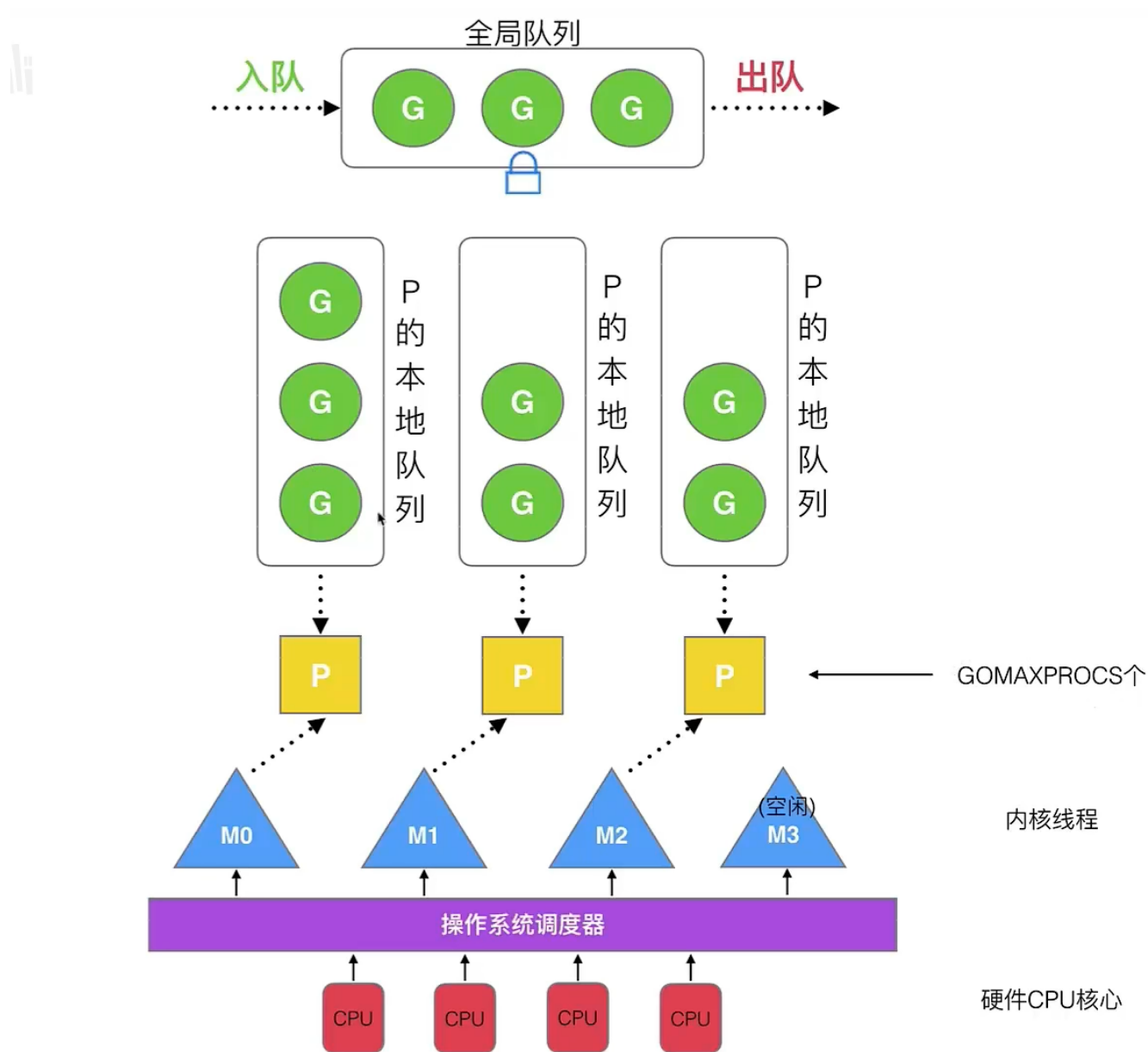


Figure 5: image-20251015160120180

- 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

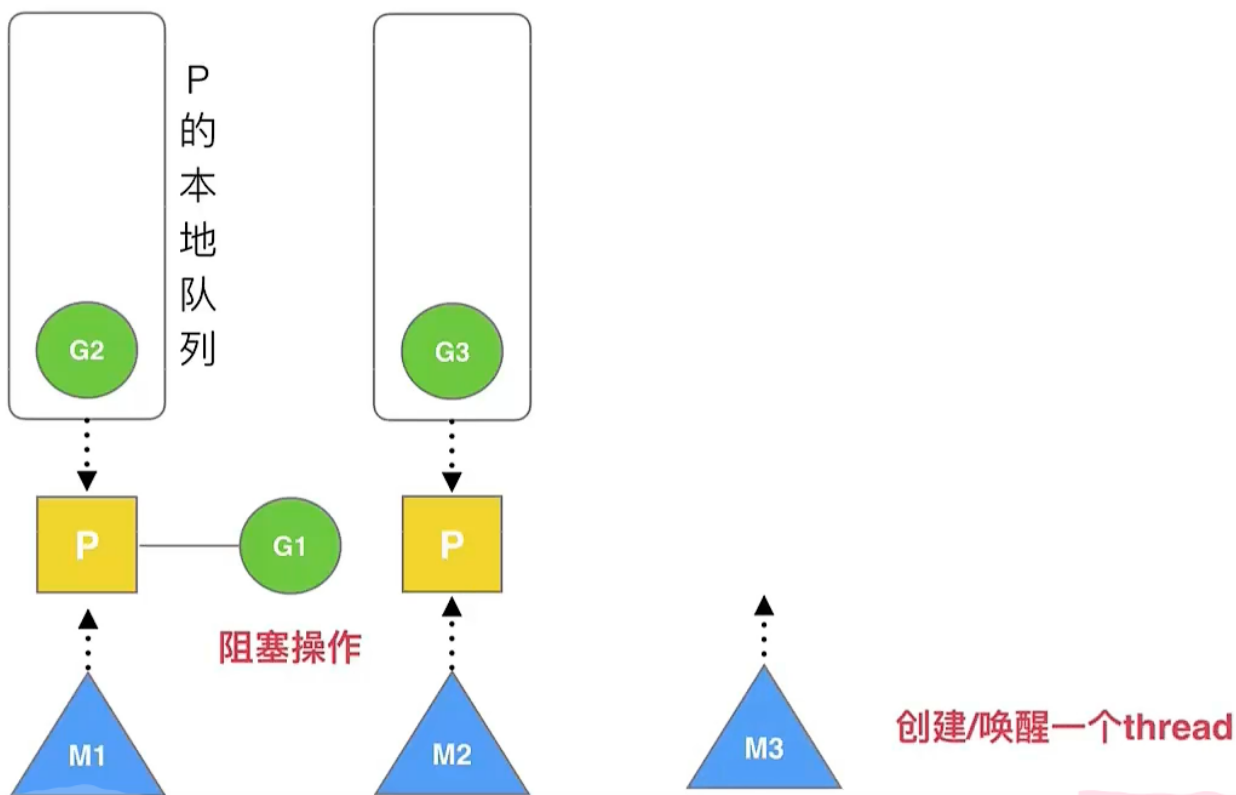


Figure 6: image-20251015161152465

- **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
- **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

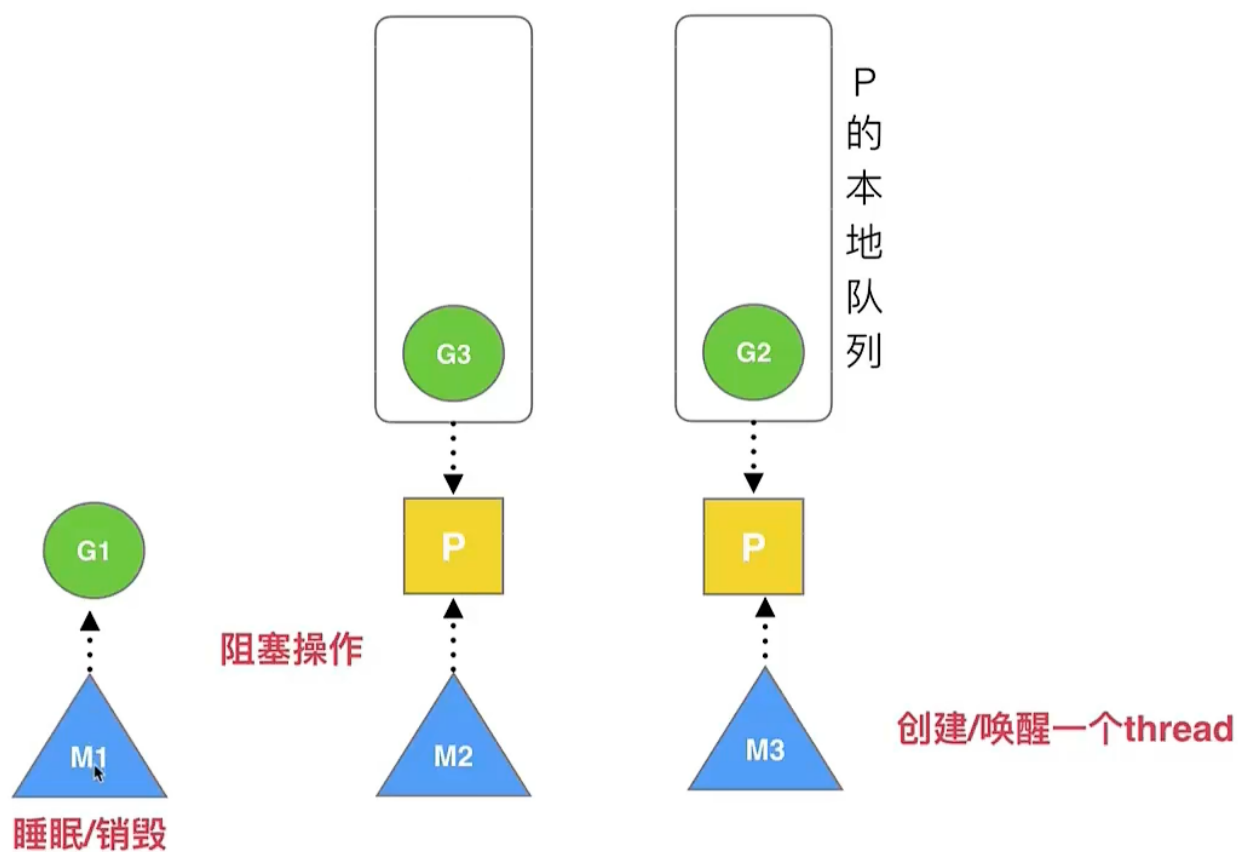


Figure 7: image-202510151611111010

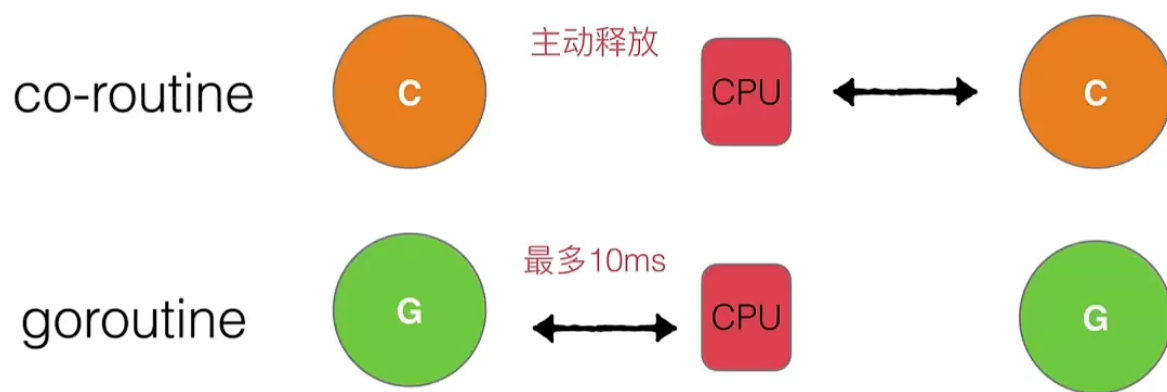


Figure 8: image-20251015165748812



Figure 9: image-20251015165732289

- 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

```
func newTask(){
    i := 0
    for{
        i++
        fmt.Printf("Hello")
    }
}

func main(){
    go newTask()
}
```

- 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

```
func main(){
    // 1.
    go func() {
        defer fmt.Println("A.defer")

        func() {
            defer fmt.Println("B.defer")
            runtime.Goexit() // 退出当前 goroutine
            fmt.Println("B") // 这行不会执行
        }()
    }()
}
```

```

    fmt.Println("A")    // 这行不会执行
}()

// 2.
go func(a int, b int) bool {
    fmt.Println("a =", a, ", b = ", b)
    return true
}(10, 20)

for{
    // ...
}
}

```

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

```

func main(){
    c := make(chan, int)

    go func(){
        c <- 666
    }()

    num := <- c
}

```

- `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`

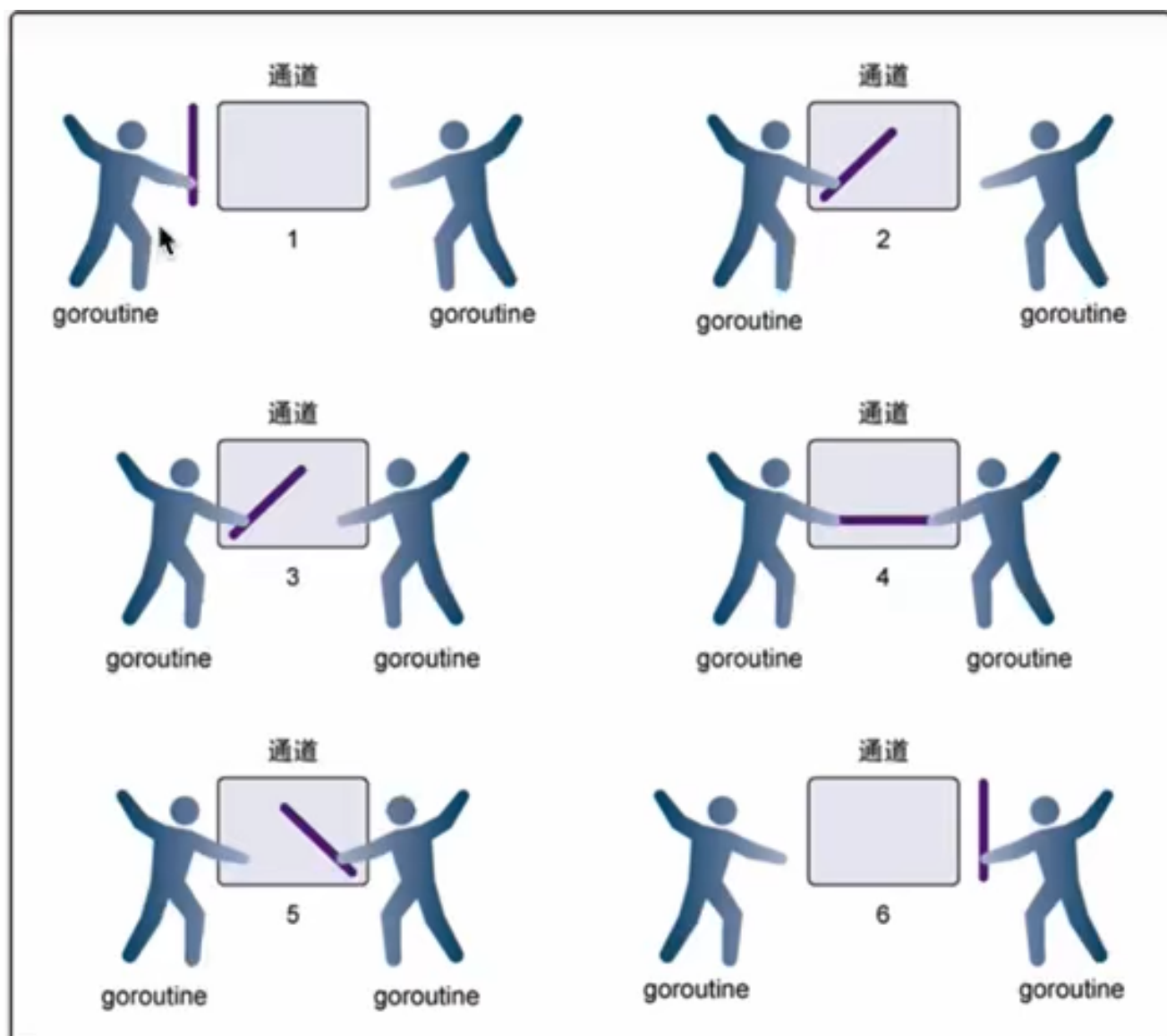
无缓冲的 channel 和有缓冲的 channel

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

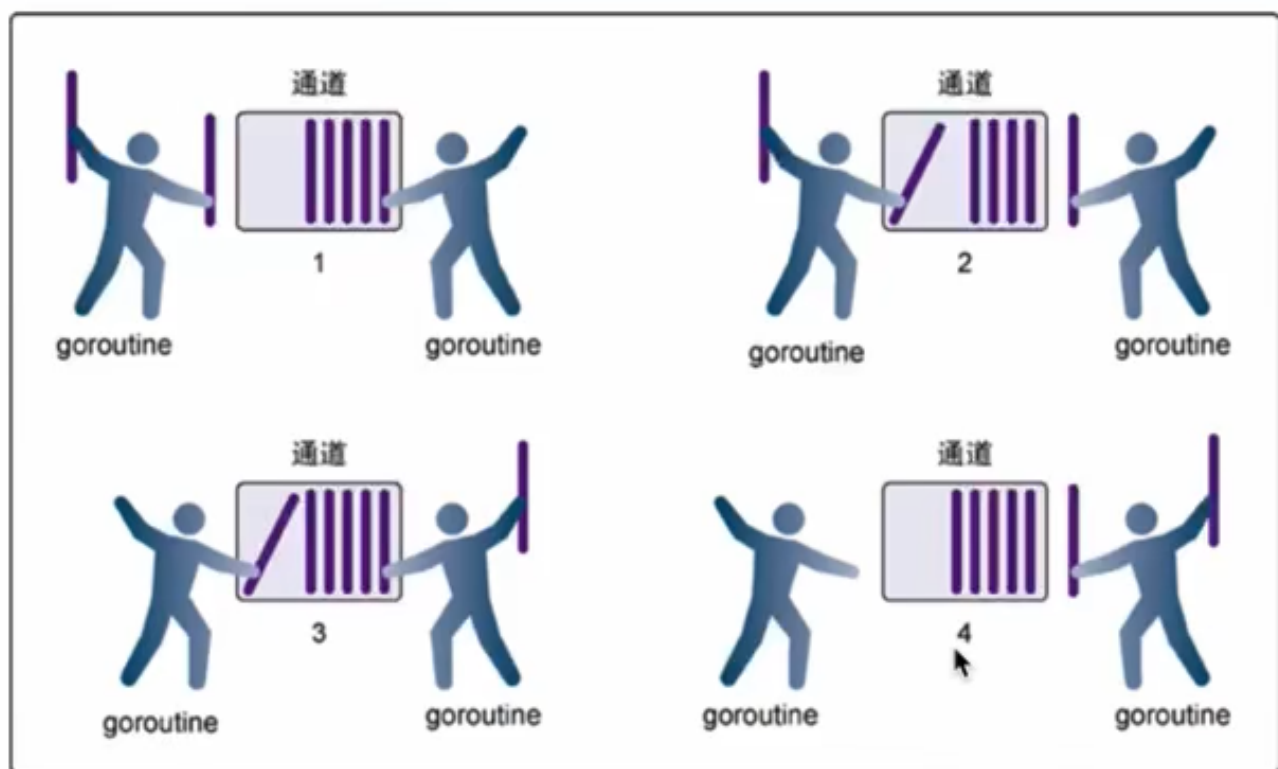
有缓冲

- 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



使用无缓冲的通道在 goroutine 之间同步

Figure 10: image-20251015173754231



使用有缓冲的通道在 goroutine 之间同步数据

Figure 11: image-20251015173810451

- 当通道空了或者通道满了，**协程才会阻塞**

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**

```
func main(){
    c := make(chan, int, 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

```
c := make(chan, int, 3)

for data := range c{
    fmt.Println(data)
}
```

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

```
select{
case <- chan1:
    // 如果 channel1 读取到数据，就执行此 case 处理语句
case chan2 <- 1:
    // 如果成功向 channel2 写入数据，就执行此 case 处理语句
default:
    // 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

- GOM11MODULE: 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 GOM11MODULE=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

```
module github.com/yourname/project // 模块路径 (必填)

go 1.21 // 最低要求的 Go 版本 (必填)

require (
    github.com/gin-gonic/gin v1.9.1
    golang.org/x/sync v0.3.0
)

replace (
    golang.org/x/sync => ./local/sync // 本地替换
)

exclude (
    github.com/old/lib v1.2.3 // 排除特定版本 (可选)
```

```
)  
  
retract (                                // 撤回发布的版本（可选）  
    v1.0.0 // 严重漏洞  
)
```

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