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

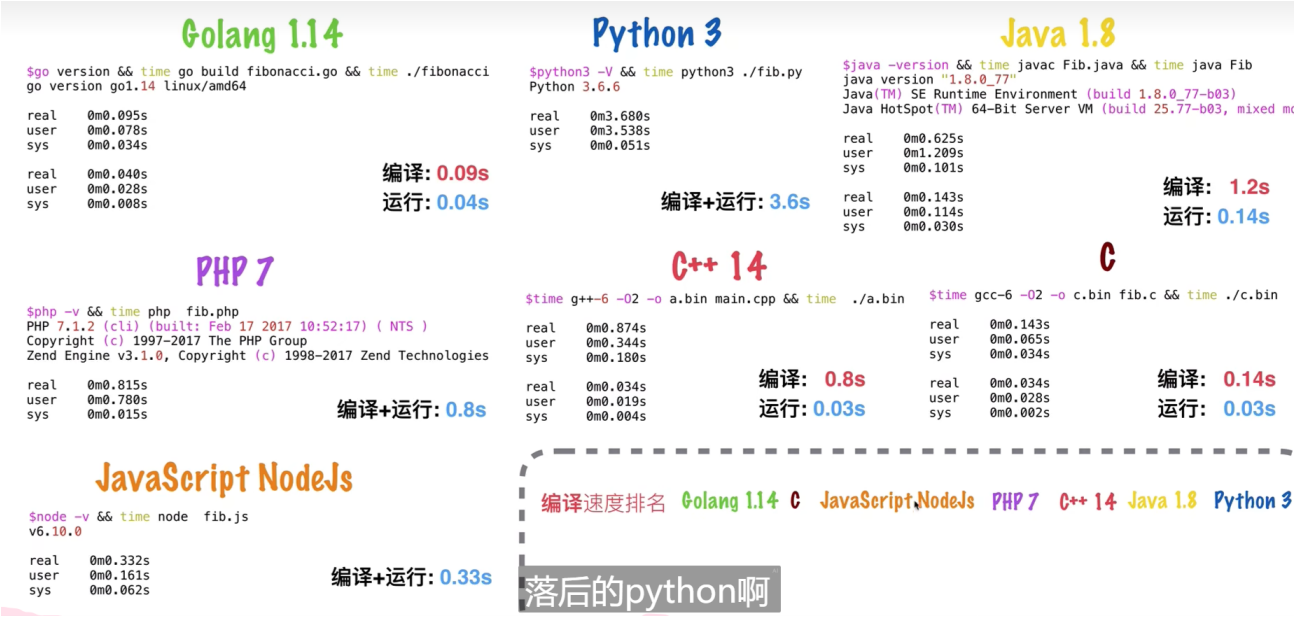


image-20251011122440219

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
```

```go
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

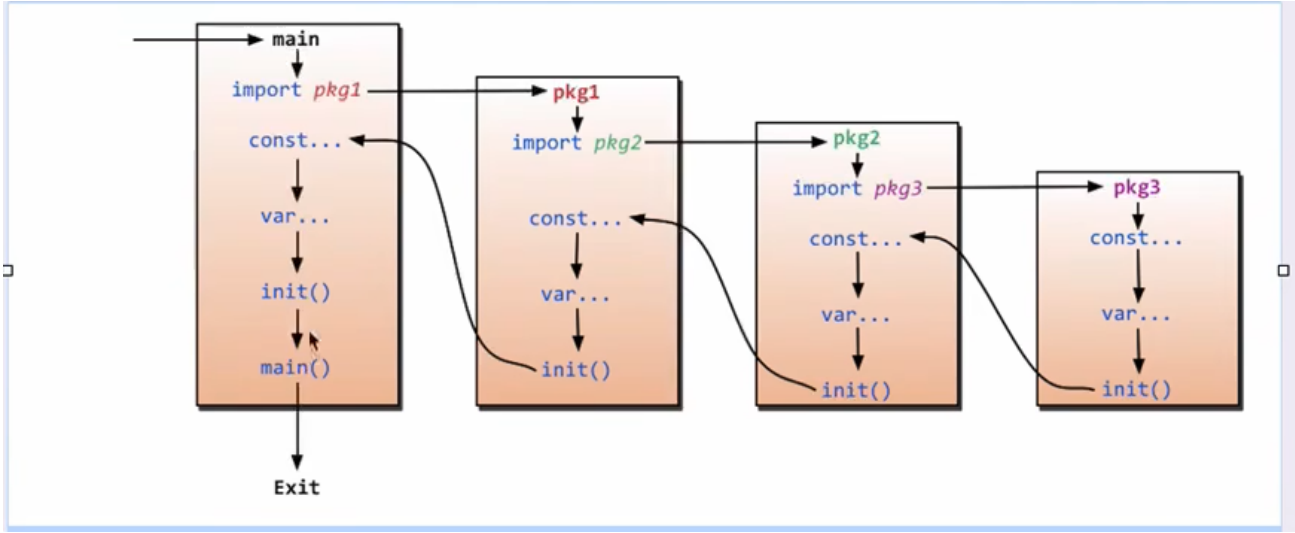


image-20251011171819781

- 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

```go
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

常见API

- `fmt.Print()`：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.Sprintln()`：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
- `strconv.Atoi`：直接转成 `int`
- `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
- `time.Now()`：获取当前时间，返回类型为 `time.Time`
  - `.AddDate(year,month,day)`：在当前日期进行日期变换

    - `.Formate()`：对日期进行格式化
    - `.Unix()`：转换成秒级时间戳
- 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

```go
// 整数切片排序
sort.Ints(nums []int)                    // 升序
sort.IntsAreSorted(nums []int) bool      // 检查是否已排序
sort.SearchInts(a []int, x int) int      // 二分查找

// 浮点数切片排序
sort.Float64s(f []float64)               // 升序
sort.Float64sAreSorted(f []float64) bool

// 字符串切片排序
sort.Strings(s []string)                  // 字典序
sort.StringsAreSorted(s []string) bool
```

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，四个字节，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

```go
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

```go
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
}
```

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

```go
func checkTemperature(temp float64) {
    // defer 函数会在 checkTemperature 返回前执行,无论是因为正常 return 还是因为 panic
    defer func() {
        // 在函数结束前检查是否有 panic
        if r := recover(); r != nil {
            fmt.Println("\"体温异常\"")
        }
    }()

    if temp > 37.5 {
        panic("体温异常")  // 触发异常
    }
    // 函数正常结束
}
```

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

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

创建error

- 使用 `errors` 包下的 `New` 函数

```go
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

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

```go
type wrapError struct{
    msg string
    err error
}

func (e *wrapError) Error() string {
   return e.msg
}

func (e *wrapError) Unwrap() error {
   return e.err
}
```

- `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`

```go
err := errors.New("错误")
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

```go
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

```go
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

```go
func main() {
   dangerOp()
   fmt.Println("程序正常退出")
}

func dangerOp() {
   defer func() {
      if err := recover(); err != nil {
         fmt.Println(err)
         fmt.Println("panic恢复")
      }
   }()
   panic("发生panic")
}
```

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`

```go
// 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()
}else{
    // ...
}
```

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

```
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之前执行的**

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

    fmt.Println("hello")
}// 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

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

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

```go
// 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**

```go
// 正确
func method(arr [5]int){

}

// 错误
func method(arr [4]int){

}

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

- 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

```go
// 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**

```go
// 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**

```go
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**
    - `append` 如果跟了多个独立切片，需要用 `...` 解包运算符

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

// 扩容
numbers = append(numbers,1)
```

- 从头部插入元素

```go
nums = append([]int{-1, 0}, nums...)
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

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

- 从尾部插入元素

```go
nums = append(nums, 99, 100)
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`

```go
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
  - 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

```
// 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

```go
// 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

```go
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

```go
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**

```go
// 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

```go
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

```go
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）

```

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

```go
type iface struct {
    tab  *itab          // 类型信息
    data unsafe.Pointer // 实际数据的指针
}
```

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

```go
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{
    // ...
}

```

```go
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

```go
// 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`）

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**

```
var a string
a = "aceld"

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

```go
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**
- `.Kind()` 方法**用于获取一个值的底层类型种类**
  - 可以统一处理所有类型的数据，如 `reflect.Int8` 、 `reflect.Bool` 等

```go
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

```go
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**

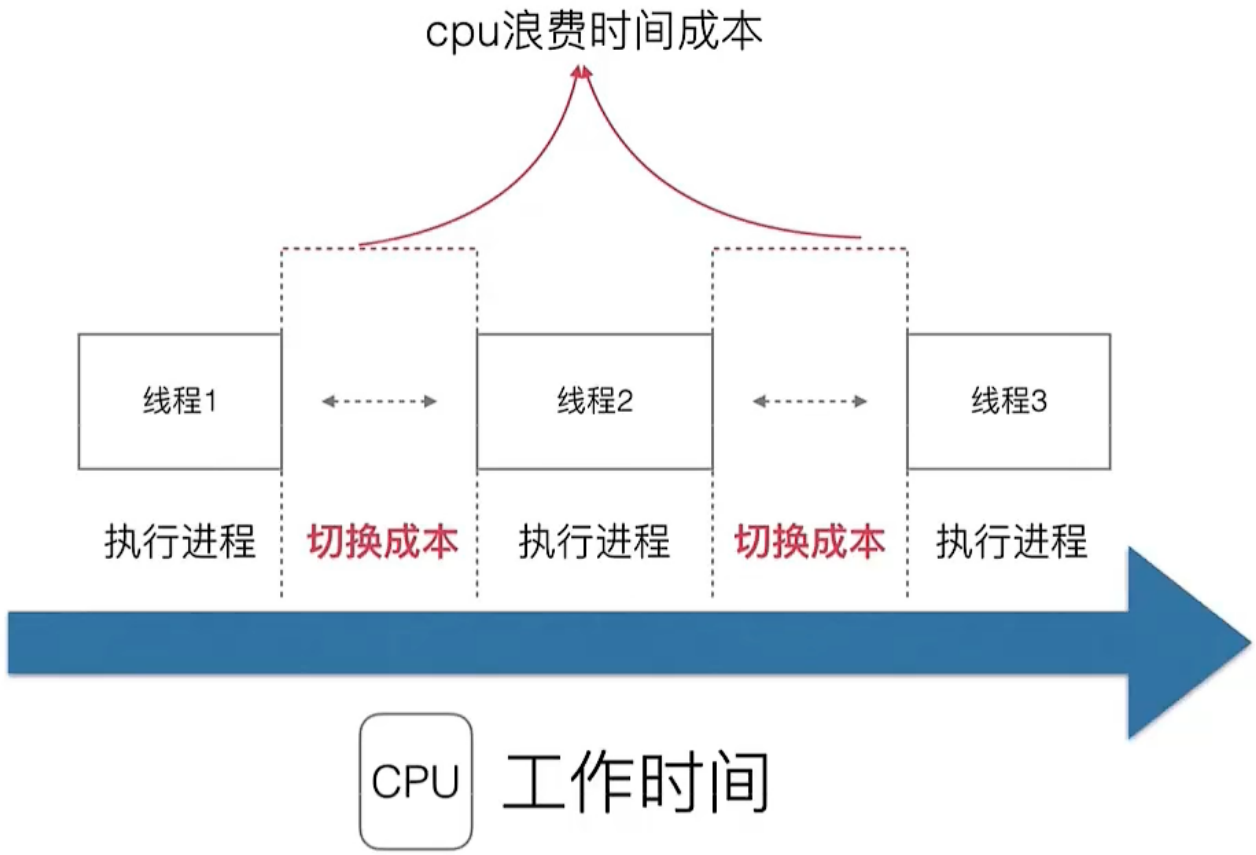


image-20251015154123809

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

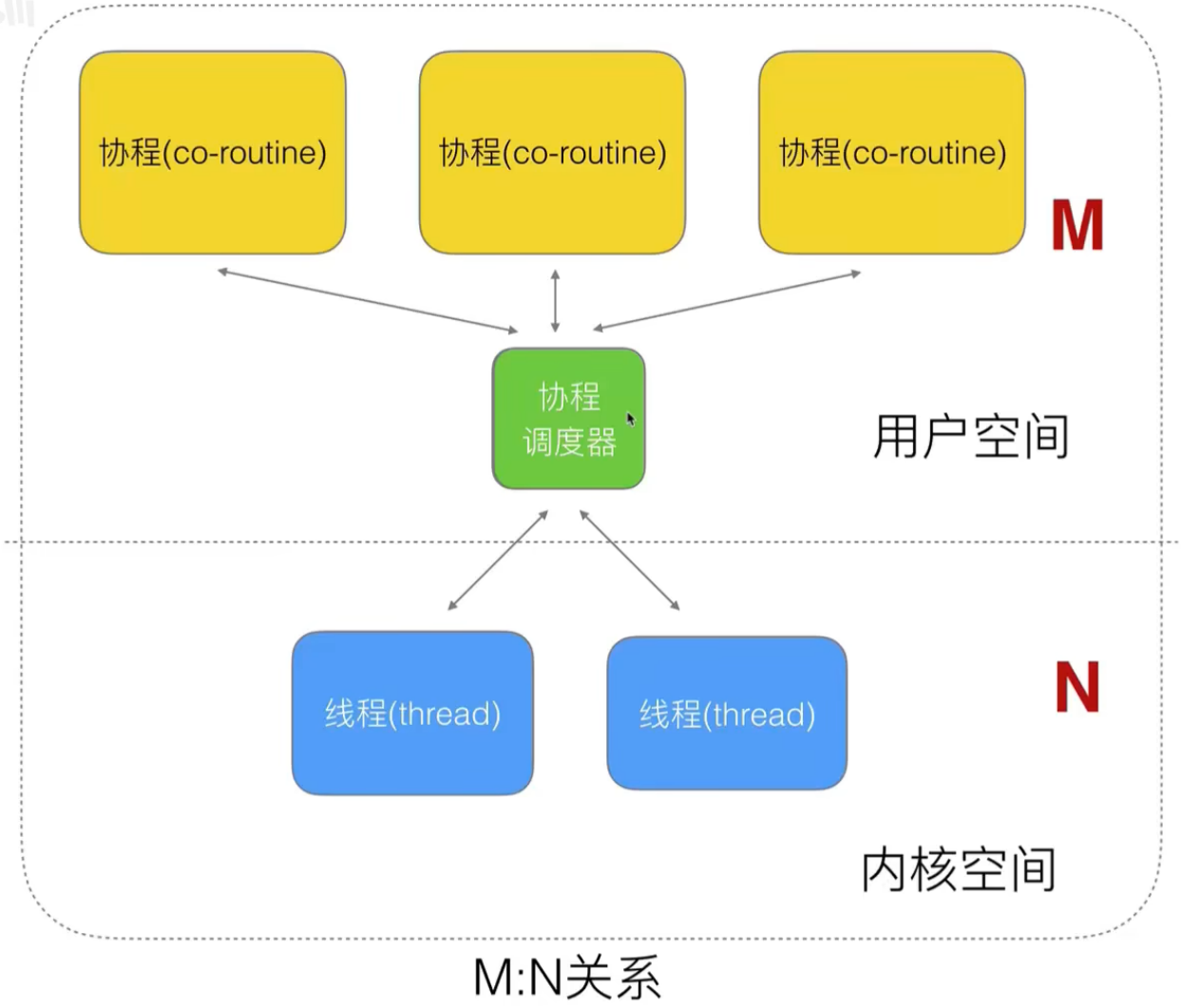


image-20251015122659139

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

```
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

```go
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

常见类型

- 双向 `channel`

```go
ch := make(chan int)
```

- 只发送 `channel`

```
var out chan<- int
```

- 只接收 `channel`

```
var in <- chan int
```

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

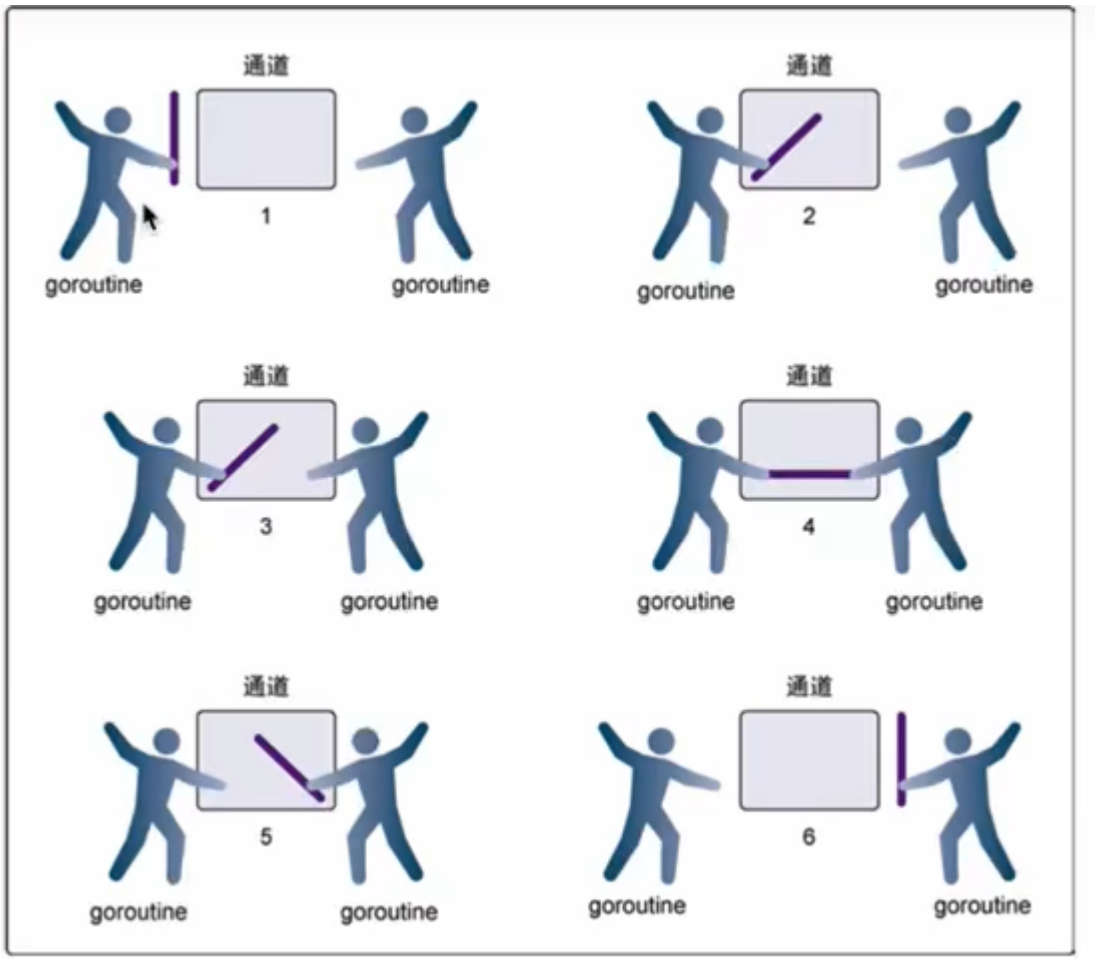
```
1  func main(){
2      c := make(chan,int)
3
4      go func(){
5          c <- 666
6      }()
7
8      num := <- c
9
10 }
```

- `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

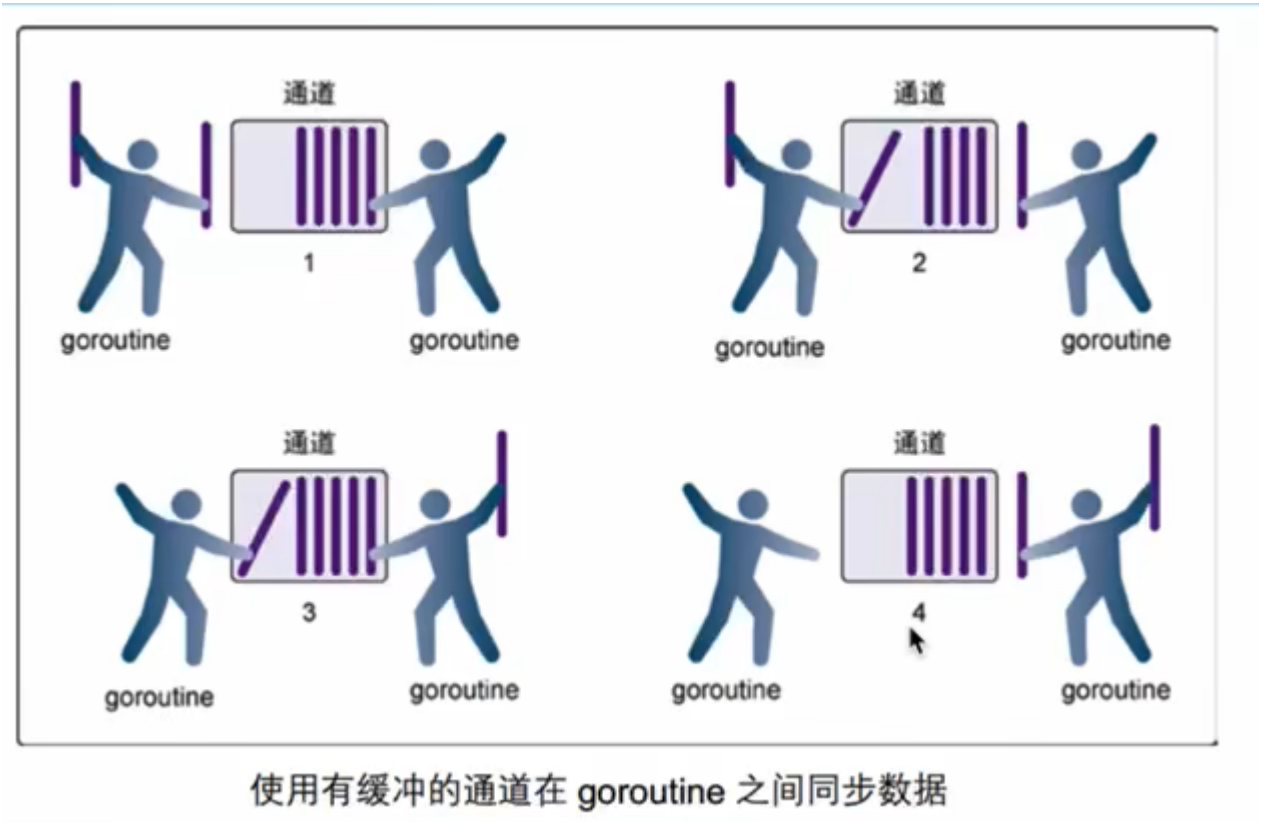


使用无缓冲的通道在 goroutine 之间同步

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**

```
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

```go
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

```go
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

- `GO111MODULE`：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 GO111MODULE=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 (                            // 直接依赖列表
    github.com/gin-gonic/gin v1.9.1
    golang.org/x/sync v0.3.0
)

replace (                            // 替换依赖源（可选）
    golang.org/x/sync => ./local/sync  // 本地替换
)

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

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

其它补充知识点

time

三件套

- `time.Time`：某个具体时刻
- `time.Duration`：时间长度，纳秒为单位的 `int64`
- `time.Location`：时区

固定模板

```
// 秒级
2006-01-02 15:04:05

// 毫秒级
2006-01-02 15:04:05.000

// 天
2006-01-02

// 小时
2006-01-02 15

// 分钟
2006-01-02 15:04

// ISO8601，带时区
time.RFC3339
```

常用方法

```go
// 获取当前时间
now := time.Now()

// 进行格式化
now := time.Now().Format("2006-01-02")

// 加减时间间隔
now := time.Now().Add(time.Duration)

// 按年月日添加
now := time.Now().AddDate(year,month,day)

// 加载时区并对时间t进行转换
loc, _ := time.LoadLocation("Asia/Shanghai")
tLocal := t.In(loc)

// 获取时间戳
tsSec := time.Now().Unix()      // 秒
tsMs  := time.Now().UnixMilli() // 毫秒
tsNs  := time.Now().UnixNano()  // 纳秒

// 时间戳转回Time
t := time.Unix(tsSec, 0)             // 秒
t := time.UnixMilli(tsMs)            // 毫秒
t := time.Unix(0, tsNs)              // 纳秒

// 计算时间差
start := time.Now()
cost := time.Since(start)  // 相当于time.Now().Sub(start)

// 分桶对齐到整点整分
t := time.Now().UTC()
hourStart := t.Truncate(time.Hour)   // 对齐到整小时
minStart  := t.Truncate(time.Minute) // 对齐到整分钟

// 时间先后判断
t.After(u)      // t 在 u 之后
t.Before(u)     // t 在 u 之前

// 取年月日
y, m, d := t.Date()

// 取时分秒
h, min, sec := t.Clock()

// 取星期
```

```go
wd := t.Weekday()

// 解析字符串
t1, err := time.Parse("2006-01-02 15:04:05", s)  // 默认按UTC解析
loc, _ := time.LoadLocation("Asia/Shanghai")
t2, err := time.ParseInLocation("2006-01-02 15:04:05", s, loc)  // 按时区进行解析
```

error

几个常见概念

- `error`：GO内置的接口类型
- `Error()`： `error` 接口必须有的方法，返回的是字符串
- `errors`：标注库中的 `errors` 包
  - `errors.New()`：创建简单错误
  - `errors.Is(err,target)`：错误比较
    - 是进行**指针比较**，会沿着错误链去寻找 `target` ，查看是否是同一个对象
    - 支持自定义方法
  - `errors.As(err,&target)`：错误的类型断言
    - 会沿着错误链去寻找 `target` ，判断是不是同个类型
    - 然后将 `err` 的值赋给 `target`
  - `errors.Unwrap()`：解包错误
- `fmt.Errorf()`：用格式化字符串创建一个 `error`
  - `%v`：把错误进行字符串拼接，不形成错误链
  - `%w`：对错误进行包裹，形成错误链

```go
// 声明普通错误
err := fmt.Errorf("invalid id: %d", id)

// 对错误进行包裹
err := fmt.Errorf("read file failed: %w", err0)

// 把err0.Error()拼进字符串
err := fmt.Errorf("read file failed: %v", err0)
```

- `.Err()`： `gorm` 框架中使用，可以调用这个方法拿到一个 `error`

结构体标签

JSON

基本用法

```go
type User struct {
    ID   int    `json:"id"`
    Name string `json:"name"`
}
```

常用Tag

- `omitempty`：空值就不输出，以下几种情况会被判定成空值
  - 数字0
  - 空字符串
  - `False`
  - `nil`
- `-`：完全忽略字段，不参与JSON
- `string`：整数、布尔转字符串

```go
ID   int    `json:"id,string"`
```

Binding

调用了`ShouldBind`方法校验就会生效

常用校验

- `required`：必须存在且非零值
- `omitempty`：字段为空则跳过校验
- 数值类
  - `gt/lt`：**大于/小于**某个值
  - `gte/lte`：**大于等于/小于等于**某个值
  - `eq/ne`：**等于/不等于**某个值
  - `oneof`：枚举验证，值之间用空格隔开
- `email`：满足邮箱格式
- `uuid`：满足`uuid`格式
- `url`：满足`url`格式
- 字符串类
  - `min`：最小长度
  - `max`：最大长度
  - `len`：必须要为这个长度

Form

Query/表单参数绑定，主要用于

- `GET`：`?page=1&pageSize=10`
- `POST`：表单提交

常常和`binding`配合使用，`-`表示忽略

uri

路径参数绑定，用于RESTful API

使用方法：`uri:"param_name"`

header

绑定**HTTP请求头**到结构体

常用的header绑定

- `Authorization`：标准token入口，不会自动去掉`Bearer`
- `Cookie`：session/jwt
- `X-API-Key`：`API-Key`鉴权
- `X-Request-Id`：请求唯一ID
- `User-Agent`：浏览器/客户端信息
- `Referer`：来源页面
- `Origin`：跨域来源
- `X-Forwarded-For`：反向代理转发的客户端IP，可能是列表
- `X-Real-IP`：客户端真实IP
- `Content-Type`：请求体类型

类型断言与类型转换

类型断言

从接口值把里面真实的具体类型拿出来

接口值（`interface`/`any`）包括了

- **动态类型**，会随着装进去的值的变化而变化
- **动态值**

语法：`x.(T)`

- `x`必须是接口类型（`interface`或`any`或其它接口），判断它里面的**动态类型**是不是`T`
- 两种方法
  - `v := x.(T)`：失败会panic
  - `v,ok := x.(T)`：安全写法，转换失败`ok`为`false`

类型转换

把一个值变成另一种具体类型

语法：`T(x)`

- `x`本身是具体类型，要转换成另一个具体类型

strings

一个处理字符串的包

常用方法

- `strings.Contains(s,substr)`：判断 `s` 是否包含子串 `substr`
- `strings.HasPrefix(s,prefix)/strings.HasSuffix(s,suffix)`：判断前缀/后缀
- `strings.Split(s,seq)`：用分隔符把字符串切成切片
- `strings.Join(elems,seq)`：将切片用分隔符 `seq` 拼接成字符串
- `strings.TrimSpace(s)`：去除首尾的空白字符
- `strings.ToUpeer(s)/ToLower(s)`：大小写转换

注意

- `len(str)` 返回的是**字符串的字节数，不是字符数**

strings.Builder

特点：底层直接操作 `[]byte` 缓冲区，避免频繁内存分配和 `GC`

比正常的 `+` 快

- 正常的 `+`，每次都会创建一个新的字符串对象，重新分配内存并拷贝数据，效率很低
- `strings.Builder` 底层维护一个 `[]byte`，只有在需要扩容时才分配内存
- 将 `[]byte` 转换成 `string` 时，使用了 `unsafe` **指针转换进行零拷贝**
  - **零拷贝**：直接让 `String` 指向 `Slice`（`[]byte`）现有的 `Data` 内存

常用方法

- `Builder.Grow(n)`：预先申请 `n` 个字节内存
- `Builder.WriteString(s string)`：把字符串拼接到末尾
- `Builder.WriteByte(c byte)/WriteRune(r rune)`：拼接单个字符
- `Builder.String()`：将 `[]byte` 转换成 `string`
- `Builder.Reset()`：清空内容，长度置零，但保留已分配的内存

匿名字段

空结构体

`struct{}`

- 零内存占用
- 所有空结构体的值都是相等的，且共享**同一内存地址**
- 无字段，不能存储数据

make

创建切片

`make([]T,length,capacity)`

- `length` 为当前切片使用的大小，`capacity` 为底层数组总大小

创建map

`make(map[K]V, capacity)`

- `capacity` 为容量，空间不够会扩容

创建通道

`make(chan T, capacity)`

- 无 `capacity` 为无缓冲通道，否则为有缓冲通道
- **不会自动扩容**

八股

Channel

CSP模型：Go中不通过共享内存通信，而是通过通信实现内存共享

注意

- 给一个 `nil` 的通道发送数据，会造成**永久阻塞**
- 从一个 `nil` 的通道获取数据，会造成**永久阻塞**
- 关闭一个为 `nil` 的通道，会**引起** `panic`
- 重复关闭通道，会**引起** `panic`
- 给一个关闭的通道发送数据，会**引起** `panic`
- 从一个关闭的通道接收数据，**如果缓冲区为空，返回零值**

如何优雅关闭chan

- 统一由发送者关闭
- 如果有多个发送者，创建一个 `stopCh` ，关闭时使用 `stopCh` 通知所有发送者

hchan结构体

`chan` 的底层实现，本质上是**带锁的环形队列**

- 核心结构
  - `qcount` ：当前队列中剩余的元素个数
  - `dataqsiz` ：环形队列的长度
  - `closed` ：标识通道是否关闭
  - `sendq` ：**发送等待队列**，因缓冲区满而阻塞的 `Goroutine` ，双向链表
  - `recvq` ：**接收等待队列**，因缓冲区空而阻塞的 `Goroutine` ，双向链表
  - `lock` ：**互斥锁**，保证 `Channel` 操作的原子性，实现 `Channel` 线程安全

Ring Buffer（环形队列）

- 环形缓冲区，缓存写入的数据
- 维护两个指针， `recvx` 与 `sendx`
  - `recvx` ：指向缓冲区中未被读取的第一个数据
  - `sendx` ：指向缓冲区中新数据插入的位置

sendq/recvq

- 双向链表

- 存储的不是原始协程G，是封装后的结构体 `sudog`

gopark/goready

- `gopark`：让协程休眠
  - 将当前协程的状态从 `_Grunning` 改为 `_Gwaiting`
  - 将当前协程从M上剥离下来
  - 让M去执行其它协程
- `goready`：让协程复活
  - 让被阻塞的协程的状态从 `_Gwaiting` 改为 `_Grunnable`
  - 让这个协程重新回到P的本地队列

调度逻辑

- 发送数据
  - **阻塞挂起**：缓冲区已满且无接收者
    - 协程G1被包装成 `sudog` 结构体，存入 `sendq` 双向链表
    - 执行 `gopark` 方法，G1进入休眠，告诉M去执行其它G
  - **直接发送**：有接收者等待
    - 此时 `recvq` 不为空，G2在等待
    - 协程G1直接绕过缓冲区，将数据直接拷贝到G2的栈空间
    - 对G2调用 `goready` 方法，唤醒G2
  - **存入缓冲区**：缓冲区未满
    - 协程G1直接把数据拷贝到缓冲区的 `sendx` 位置
- 接收数据
  - **阻塞挂起**：缓冲区为空且无发送者
    - 协程G2被包装成 `sudog` 结构体，存入 `recvq` 双向链表
    - 调用 `gopark` 方法，G2 进入等待状态，通知M去执行P里的其它协程
  - **直接接收**：缓冲区为空且有发送者（**无缓冲通道**）
    - 接收者G2发现 `sendq` 有节点，直接把数据拷贝到自己的栈空间中
    - 对等待者G1调用 `goready` 方法
  - **缓冲满时接收**：缓冲区已满且无发送者
    - 接收者从 `buf` 取走 `recvx` 位置上的数据
    - 接收者发现 `sendq` 有G1阻塞，顺手把G1的数据放到 `buf` 里面
    - 对G1调用 `goready` 方法
  - **从缓冲区接收**：缓冲区未满且无发送者
    - 接收者从 `buf` 取走 `recvx` 位置上的数据
    - 接收者继续执行代码

make与new

new

- 创建一个**类型**的实例，并返回新分配地址的**指针**
- 使用 `new` 来分配空间，传入的参数是**类型**

make

- 给 `slice` 、 `map` 、 `chan` 进行初始化，然后返回**初始化后的值**

Slice

底层实现

- 指向底层数据的指针
- `len`：切片长度
- `cap`：切片容量

扩容机制

- 如果新申请的容量大于旧容量的两倍，**新容量直接等于新申请的容量**
- 当切片元素小于256个时，**新切片容量直接翻倍**
- 当切片元素大于等于256个时，每次增加$(\text{oldcap}+3\times 256)/4$，**直到满足新申请的容量**

内存对齐

- 计算出最终要扩容的容量后，Go并不会直接按这个数字分配内存
- Go会根据预设的内存块大小**进行向上取整**

注意

- 由于扩容会指向**新的底层数组**，扩容后对新切片的修改不会影响旧切片

数组

编译时确定类型，**数组的长度是类型的一部分**

- 函数传递时拷贝整个数组，开销大
- 数组分配在连续内存上

Map

`map`并不是线程安全的，如果两个协程同时写一个`map`，程序会直接`panic`

底层实现

`map`本质上是一个**哈希表**，使用**链地址法（数组+链表）**解决哈希冲突

- `hmap`：是`map`的大脑，包括
  - `count`：成员数量
  - `B`：桶的数量（2^B），**不包括溢出桶**
  - `buckets`：指向桶数组的指针
  - `extra`：溢出桶信息
- `bmap`：桶，每个桶固定存储**8个键值对**
  - `tophash`数组：存储每个key哈希值的高8位，用于快速定位key是否在桶内
  - `Key&Values`：**内存对齐，防止内存浪费**，紧凑排列，key放在一起，value放在一起
  - `Overflow Pointer`：指向下一个溢出桶的指针
- 溢出桶：桶内位置不够用，就会申请溢出桶
  - 和`bmap`结构一样，最后会形成**溢出桶单向链表**
  - **预分配机制**
    - 动态分配：使用`mallocgc`向系统申请内存
    - 预留池：初始化大`map`时向系统**额外申请一些内存**作为备用溢出桶

定位机制

- **定位桶**：使用哈希值的**最后B位（低位）**定位到具体的桶
- **定位桶内位置**：使用哈希值的**高8位（高位）**在桶内进行快速线性匹配
- **插入**时找到桶内的空插槽，存入Key和Value

扩容机制

根据**装载因子**决定如何进行扩容

$$\text{LocalFactor} = \frac{\text{count}}{2^B}$$

- **翻倍扩容**：装载因子超过6.5，桶快占满了，哈希碰撞严重，**溢出桶多**
  - `B` 变为 `B+1` ，旧桶的数据被分流到两个新桶中，通过 `Key` **哈希值的第B位**决定去向
    - 该位置是0：留在**原序号**的新桶中
    - 该位是1：留在**原序号+**2^B的新桶中
- **等量扩容**：装载因子低于6.5，使用**过多的溢出桶**，空间利用率低，导致**桶内数据稀疏**
  - 通常发生在频繁删除哈希表中的数据的场景下，因为删除操作只把数据删除，**不会回收溢出桶占用的内存**
  - 新开辟一块和旧桶一样大的空间
  - 将旧桶里的数据重新排列，紧凑放入新桶中，**消除内存碎片**

渐进式扩容

核心：数据搬迁被拆分到了每一次的**写和删**中

- 新开辟内存空间
- 标记旧桶：将原有的 `buckets` 挂载到 `hmap.oldbuckets` 指针上
- 初始化进度条： `hmap.nevacuate` 置为0
- 当尝试修改或删除某个Key时，先定位到这个Key所在的旧桶，然后把这个旧桶及其挂载的所有数据迁移到新桶

扩容期间的读写逻辑

- **读操作**
  - 先去旧桶找，如果发现旧桶已经被搬迁，去新桶找
  - **读操作本身不触发搬迁**
- **写操作**
  - 先触发搬迁
  - 搬迁完成后，在新桶写数据

值传递

Go里面只有值传递，没有引用传递（小心面试官钓鱼）

即使是 `map` 、 `chan` 、 `slice` ，在进行传递时也只是拷贝了他们的底层结构体**（包含指向底层数据的指针）**

- 如果在函数内部修改了 `slice` 元素，**外部可见**
- 如果使用 `append` ，**外部不可见**，因为外部的 `len` 和 `cap` 变量的值没有改变

三色标记

白色：不可达对象，需要进行GC

灰色：中间状态对象，它内部引用的其他对象还没被检查完

黑色：存活对象，已经被检查过，并且它引用的对象已经被标记成黑色或灰色

原理

- 首先将所有的对象放到白色集合中
- 从**根节点**开始遍历对象，遍历到的白色对象放到灰色集合中
- 遍历**灰色集合**中的对象，将它引用的对象标记成灰色，同时把**它自己标记成黑色**
- 循环上述步骤，直到**无灰色对象为止**

STW

STW是为了捕获即时的 `GC Roots` 指针，从而捕获到最后的指针状态，防止回收正在使用的内存

- 在并发标记期间，栈的状态在第一次扫栈后会一直变化，如果最后不进行STW，就会导致GC回收掉正在使用的内存

本质： `sysmon` 让所有的处理器（P）都停止运行用户代码，进入一个**安全点**（Safe Point），确保内存状态不再发生变化

写屏障

> 若在执行GC时，程序把白色对象挂到了黑色对象下面，怎么办？

写屏障就是专门解决这个问题的：当程序试图修改对象的引用关系时，写屏障会强行把被引用的对象涂成**灰色**

GC回收流程

- 扫描栈和堆，获取初始的 `GC Roots` 集合
- 进行并发三色标记
- GC结束前进行**栈重扫**
  - 在混合写屏障诞生前会产生较长的STW

插入写屏障

概念：当一个**黑色对象**引用一个白色对象时，立刻把这个白色对象涂成**灰色**

- 在栈上不开启，因此GC结束前必须**暂停程序（STW）**重新扫描一遍栈，会造成卡顿

删除写屏障

概念：一个**灰色对象**引用了一个白色对象，程序执行时白色对象**被取消引用**，此时立刻把白色对象涂成**灰色**

- 当**黑色对象持有白色对象，且灰色对象取消这个白色对象的引用时**，就需要删除写屏障
- 因为如果不把白色对象涂灰，由于**黑色对象不会再扫描**，这个被取消引用的白色对象就会被回收，造成程序崩溃

混合写屏障

融合了**插入写屏障和删除写屏障**的优点，但是是保守的策略，会导致内存占用较大

核心：

- 初始进行一次短暂的STW，**将栈上的对象全部标黑**
- 在堆上把旧引用和新引用的对象**涂灰**
- GC期间把**栈上新创建的对象涂黑**

内存逃逸

基本概念：Go编译器在**编译阶段**通过静态分析，决定一个变量应该分配在**栈**上还是**堆**上的过程

- **栈**：随函数调用创建，结束销毁，**无需GC介入**，持有空间小
- **堆**：持有的内存空间大，但需要GC介入进行内存回收

常见逃逸场景

- **指针逃逸**：函数返回变量地址，需要进行逃逸
- **接口类型逃逸**：编译阶段类型不确定
- **变量内存太大**：栈无法承载，逃逸给内存更大的堆
- **动态长度的切片**：切片长度在编译期间不确定

注意

- 在实际开发时，传递指针会**增大GC压力**，因为对象逃逸到堆，增加扫描时长
- `new` 出来的对象不一定在堆上，如果对象没被外部引用，**编译器会对其进行优化分配到栈上**

Go Runtime

负责四大核心工作

- **协程调度**：GMP模型
- **垃圾回收**：三色标记、混合写屏障
- **内存管理与分配**
- **运行时检查与支撑**

sysmon

`System Monitor`

- 独立于GMP模型之外
- 运行在物理线程之上
- 可以看作是一个**后台监控线程**

四大基本任务

- **抢占长任务**：强制中断长时间占用CPU的协程
- **Hand Off**：当G遇到 `syscall` ，M会阻塞，**此时** `sysmon` **强行把与这个M绑定的P进行解绑**，交给其他空闲的M
- **强制GC**：系统长时间未进行GC， `sysmon` 会进行强制GC
- **释放物理内存**

抢占式调度

出现的场景

- STW
- 计算密集型无限循环任务

协作式的抢占式调度

Go1.14版本之前使用的调度方法

具体流程如下

- 后台监控线程sysmon巡检发现某个协程运行超过了10ms，把这个G的 `stackguard0` 标志位设为一个特殊值

    - 这是因为Go编译器在编译阶段会在每个函数的入口处插入一段指令，**用于检查当前协程的栈空间是否足够**
    - 检查过程需要检查标志位 `stackguard0`
- G发现自己的标志位 `stackguard0` 被设为特殊值，主动保存自己的寄存器状态，让出CPU给其它协程

弊端

- 若协程执行的是死循环，由于无函数调用，协程不会检查标志位也就无法退出，此时协程占用CPU资源无法释放

基于信号的抢占式调度

Go1.14版本及之后使用的调度方法

具体流程如下

- 后台监控线程sysmon巡检发现某个协程运行超过了10ms，sysmon调用系统调用，向运行该协程的线程M发送一个 `SIGURG` 信号
- 内核接受到信号后，暂停M的指令流，强制将 CPU 的控制权交给预先注册好的**信号处理函数（sighandler）**
- `sighandler` 会把当前协程的寄存器状态进行保存，并修改协程的程序计数器，让他指向一个特殊的 `Runtime` 函数
- 这个 `Runtime` 函数把原本被中断的代码地址压栈，然后调用调度逻辑，将协程放回全局队列，让M运行其它的协程

Go的GM模型

- **G**： `goroutine` **协程**
- **M**：操作系统内核的 `thread` **线程**

运行流程

- 维护一个全局队列，全局队列存储G
- 系统中所有的M空闲时，就去全局队列取出G执行

系统调用（System Call）

- 简称 `syscall` ，**应用程序向操作系统内核请求服务**的唯一入口
- 一次系统调用的代价非常昂贵，往往需要**几百纳秒**

弊端：

- 任何关于全局队列的操作都需要**加锁**，锁竞争导致严重的性能瓶颈
- 系统调用引起**线程爆炸**，易导致OOM
    - 当协程G1需要进行 `syscall` 时，CPU会把G1踢出CPU，并且把M1挂起（此时M1无法进行任何操作），如果此时全局队列还有很多G未被执行时，系统会创建更多的线程执行G，导致线程爆炸
- **局部性灾难**，缓存失效
    - 当G1执行 `syscall` 时，M1会被系统挂起进入休眠，当系统调用结束后，G1会被送入全局队列，此时可能由M2拿到G1，但是M2并没有G1中的数据，需要从内存重新加载，造成缓存失效

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**

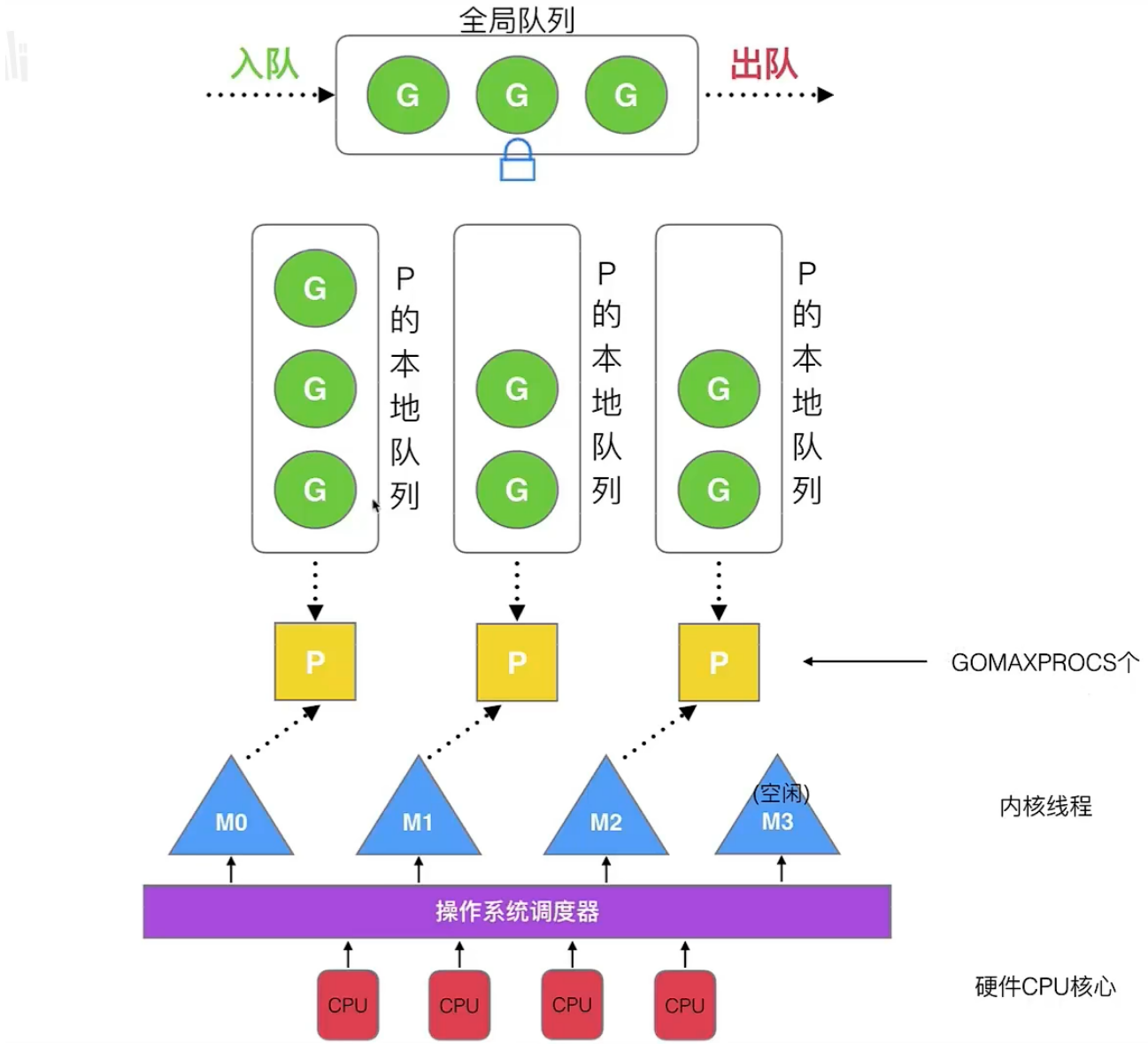


image-20251015160120180

GMP调度流程

- 每个P和一个M绑定，M是真正执行 `goroutine` 的实体，M从**绑定的P的局部队列**获取 `goroutine` 来执行
- 每个P都有一个**局部队列**，局部队列保存待执行的 `goroutine` ，当局部队列满了之后就会把 `goroutine` 放到**全局队列**
- 当M绑定的P的**局部队列为空**时，M会从**全局队列**获取到本地队列来执行G；当全局队列为空时，M会从**其它P的局部队列**偷取G来执行（ `Work Stealing` ）

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` 进行 `syscall` **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`
  - 发生 `hand off` 时，需要对线程上下文的状态进行存储，以便下次调度时进行恢复。**M的栈保存在G对象上，将M所需要的寄存器保存到G对象上即可实现现场保护**

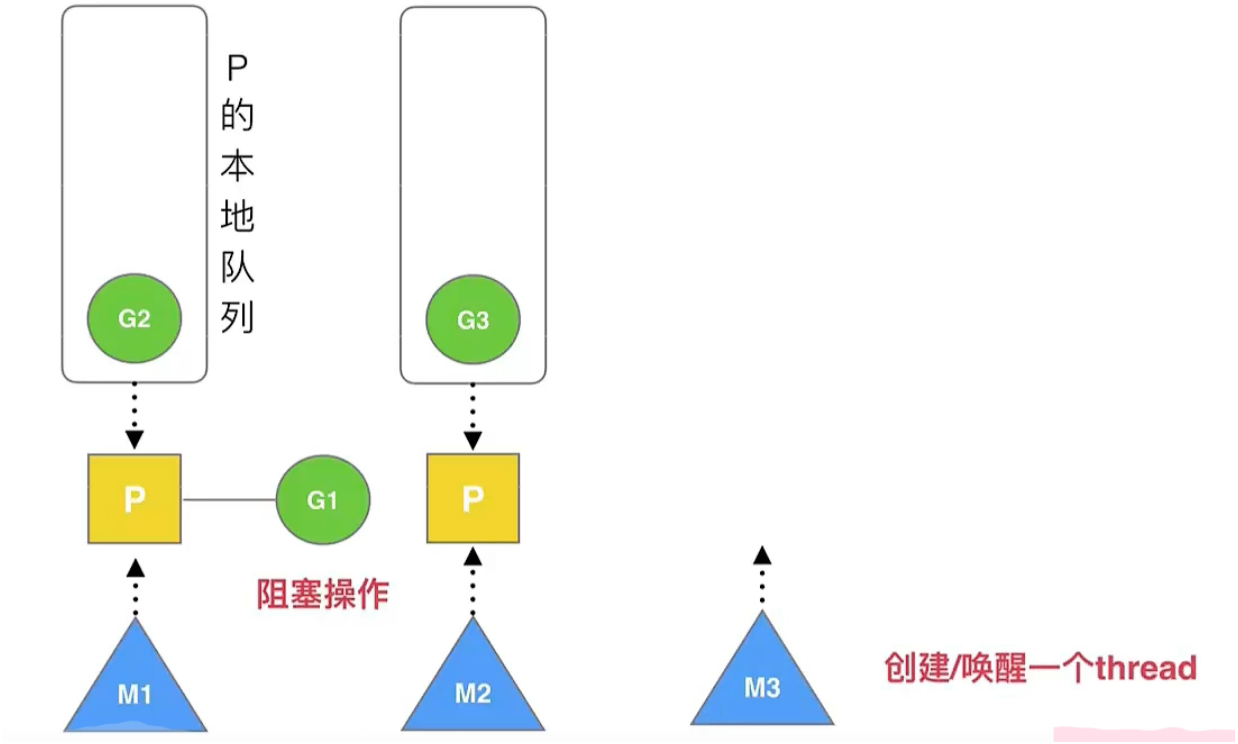


image-20251015161152465

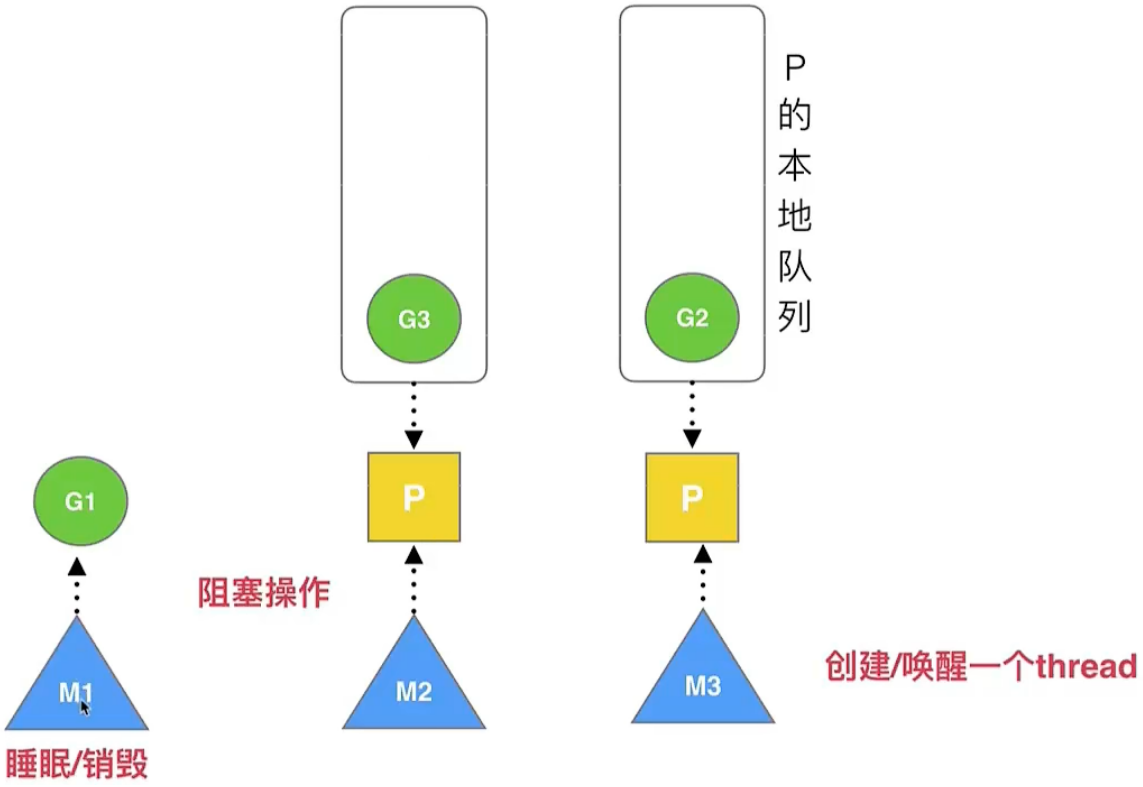


image-20251015161111010

- **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`

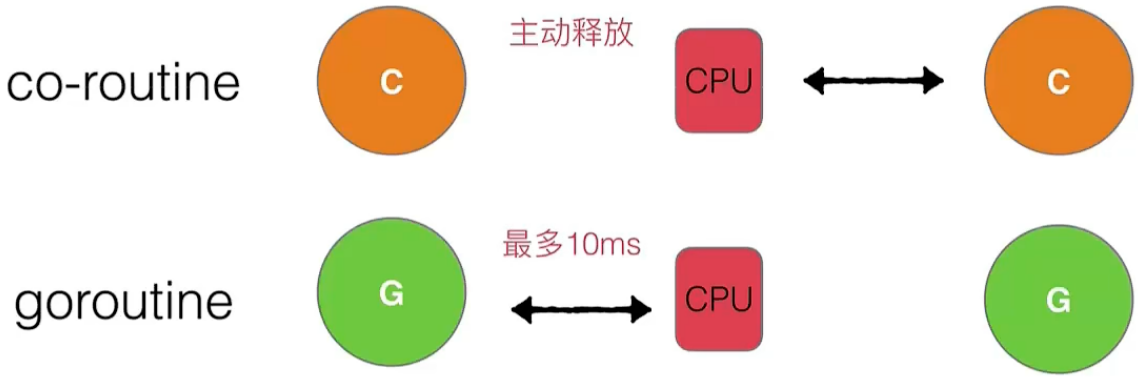


image-20251015165748812

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`

> 为什么每个P要绑定一个局部队列，而不是直接使用全局队列?

减小锁竞争，实现无锁化调度，因为每个M从局部队列获取G的过程是无锁的，极其迅速的，如果使用全局队列，就会加剧锁竞争，降低工作效率