Golang 笔记

Release 0.1.0

hiyongz

Oct 26, 2021

GOLANG 基础语法

CHAPTER

ONE

GO 语言开发环境安装

Go 语言（Golang）由 Google 的 Robert Griesemer，Rob Pike 和 Ken Thompson 推出，Go 语言具有并发性，可以快速编译机器码，自动垃圾回收，是一种静态强类型、编译型语言。由于 Go 语言的并发特性，非常适用于 Web 服务器、分布式集群计算、云计算、游戏服务端等的开发。在区块链（如以太坊，Ethereum）、容器（如 Kubernetes）领域都有广泛的应用。本文介绍 Go 语言开发环境安装方法。

1.1 安装

Go 支持支持 windows、linux、mac 操作系统，下面介绍 Windows 系统安装方法：下载地址：https://golang.google.cn/doc/install

go 国内镜像下载地址：https://gomirrors.org/

双击 msi 文件安装，安装成功后会自动加入环境变量，测试是否安装成功：`go version` windows 打开 cmd

```
C:\Users\10287>go version
go version go1.15.6 windows/amd64
```

帮助命令：

```
$ go help
```

Linux 系统安装：

下载安装包，使用 root 用户执行：

```
$ rm -rf /usr/local/go && tar -C /usr/local -xzf go1.16.6.linux-amd64.tar.gz
```

将/usr/local/go/bin 添加到环境变量：`vim /etc/profile`

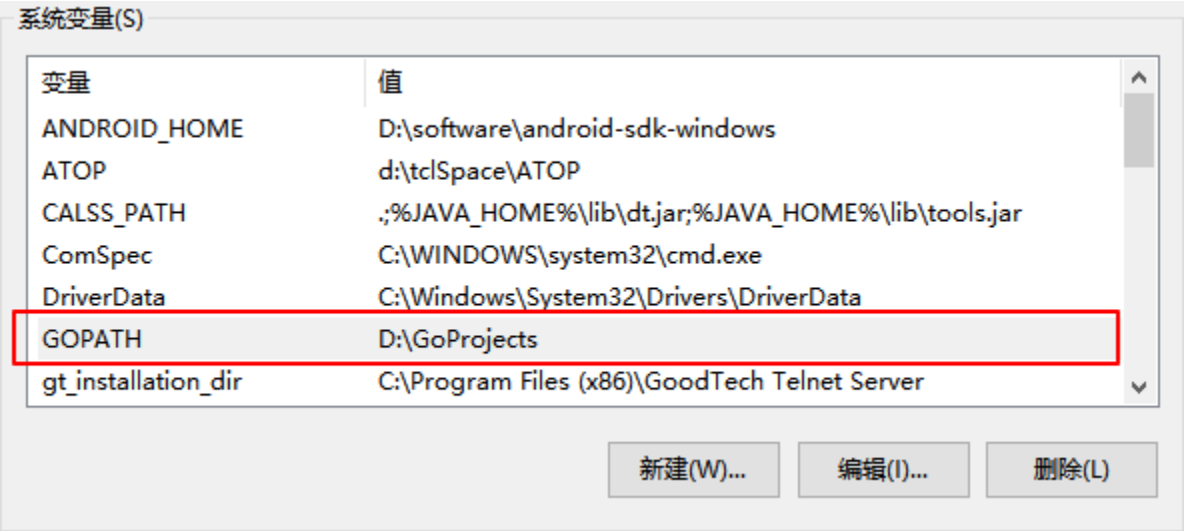
```
PATH=$PATH:/usr/local/go/bin
GOPATH=/var/GoProjects
```

```
source /etc/profile
go env -w GOPROXY=https://goproxy.cn
```

1.2 Go 代码编写运行

1.2.1 项目目录结构

配置一个变量名为 GOPATH 的系统变量，值为 Go 项目的工作目录，可以是多个路径。



在工作目录 D:\GoProjects 下创建 bin、pkg、src 三个目录

- bin：存放编译后可执行的文件。
- pkg：存放编译后的应用包。
- src：存放应用源代码。

还有一个环境变量叫 GOROOT，是 Go 的安装路径，这个可以不用配置，默认添加到了环境变量

在 src 目录下新建一个 HelloWorld 的项目，新建 hello.go，项目目录树如下：

```
D:\GOPROJECTS
|
├─bin
├─pkg
└─src
    └─HelloWorld
            hello.go
```

编写代码如下：

```
package main
import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

HelloWorld 目录下运行：

```
$ go run hello.go
Hello, World!
```

1.2.2 go modules 依赖管理

Go mod 是 Go 语言依赖库管理器，官方推荐使用这种方法来管理依赖，相比 GOPATH 方法更加灵活，记录和解析对其他模块的依赖性。

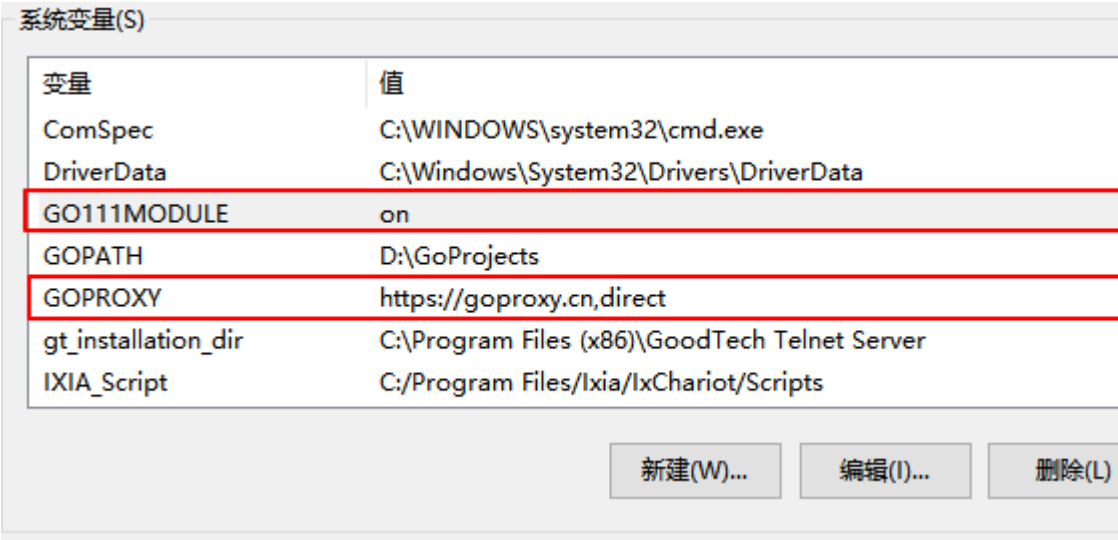
Go mod 是 go1.11 之后新增的功能，版本至少需要 1.11。

打开 Go mod，在 windows cmd 窗口输入如下命令设置环境变量：

```
$ setx GO111MODULE on
```

GO111MODULE=on 表示开启模块支持，忽略 GOPATH 文件夹，根据 go.mod 下载依赖。由于国内网络问题可能无法下载 go 依赖包，需配置一下国内代理（七牛云提供）：

```
$ setx GOPROXY https://goproxy.cn,direct
```



或者直接手动添加环境变量

下面使用 go mod 来管理一个项目：

初始化项目

项目 src 目录下创建的项目目录，执行如下命令：

```
$ go mod init
```

目录下会生成一个 go.mod 文件：

```
module HelloWorld

go 1.16
```

还有一个比较常用的命令是 `go mod tidy`，用于安装需要的依赖包，删除多余的包。

1.2.3 运行方式

Go 是一种编译型语言，代码编写完成后，需要先进行编译后再执行。

1、go run

编译 + 执行，不生成其它任何文件

```
$ go run hello.go
Hello, World!
```

2、go build

先编译，再手动执行

- `-a`：强制编译，目标代码包和依赖的代码包（包括标准库中的代码包）都会被编译
- `-x`：会打印执行日志
- `-n`：只查看具体操作而不执行它们
- `-v`：可以看到 go build 命令编译的代码包的名称

```
$ go build hello.go
$ hello.exe
```

执行 `go build hello.go` 后，在 windows 系统上会自动生成一个后缀为 exe 的可执行文件，可使用`-o` 参数指定编译文件名：

```
$ go build -o hello hello.go
$ hello
Hello, World!
```

3、go install

先编译，将编译好的可执行文件移动到 $GOPATH/bin 目录下，将包文件放到 pkg 目录下。

```
$ go install hello.go
```

4、go clean

清除执行 go 命令而遗留下来的临时目录和文件

- -i 参数：清除通过 go install 命令生成的文件，也就是会把 bin 和 pkg 目录下的相关文件清除
- -cache 参数：清除 go build 命令生成的文件
- -n 参数：打印要执行的清除命令，不执行清除
- -x 参数：打印要执行的清除命令，执行清除

```
$ go clean -i hello.go
$ go clean -n hello.go
cd D:\GoProjects\src\HelloWorld
rm -f HelloWorld HelloWorld.exe hello hello.exe HelloWorld.test HelloWorld.test.exe␣
↪hello.test hello.test.exe hello hello.exe
```

1.2.4 Go 源码安装：go get

go get 会自动从代码仓库（比如 GitHub.com、golang.org 等）下载目标代码包，安装的路径为设置的环境变量 GOPATH 中。下面介绍几个常用参数：

- -u：下载并安装代码包，不论工作区中是否已存在它们。
- -d：只下载代码包，不安装代码包。
- -fix：在下载代码包后先运行一个用于根据当前 Go 语言版本修正代码的工具，然后再安装代码包。
- -t：同时下载测试所需的代码包。
- -insecure：允许通过非安全的网络协议下载和安装代码包。比如 HTTP。

更多 go get 命令使用方法可参考：https://github.com/hyper0x/go_command_tutorial/blob/master/0.3.md

其它 go 命令详细文档可参考 Go 语言官方文档：https://golang.google.cn/cmd/go/

1.3 开发环境

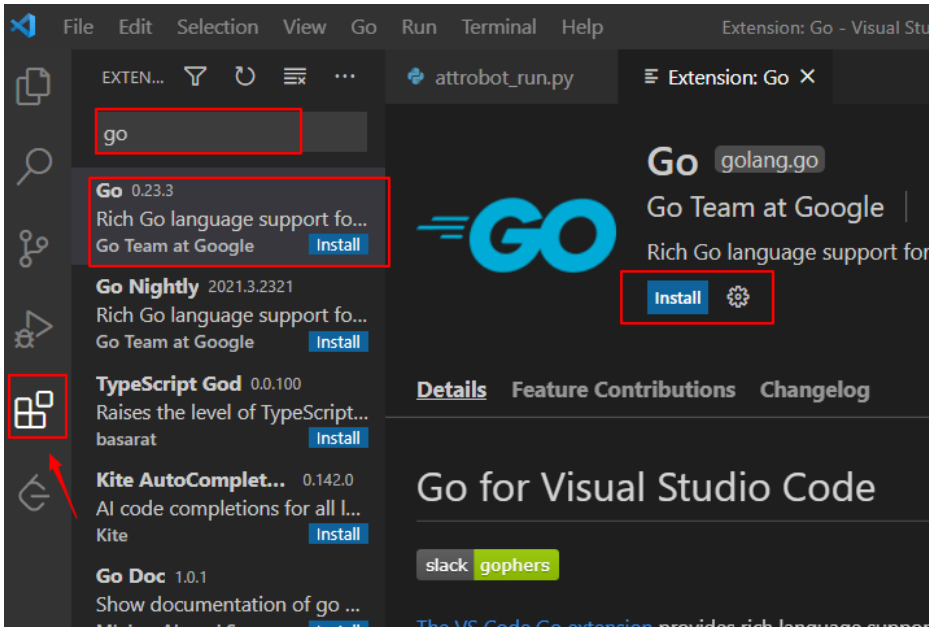
1.3.1 VSCode

VSCode 比较轻量级，是我比较常用的代码开发工具。

1、配置代理

按照前面的方法配置

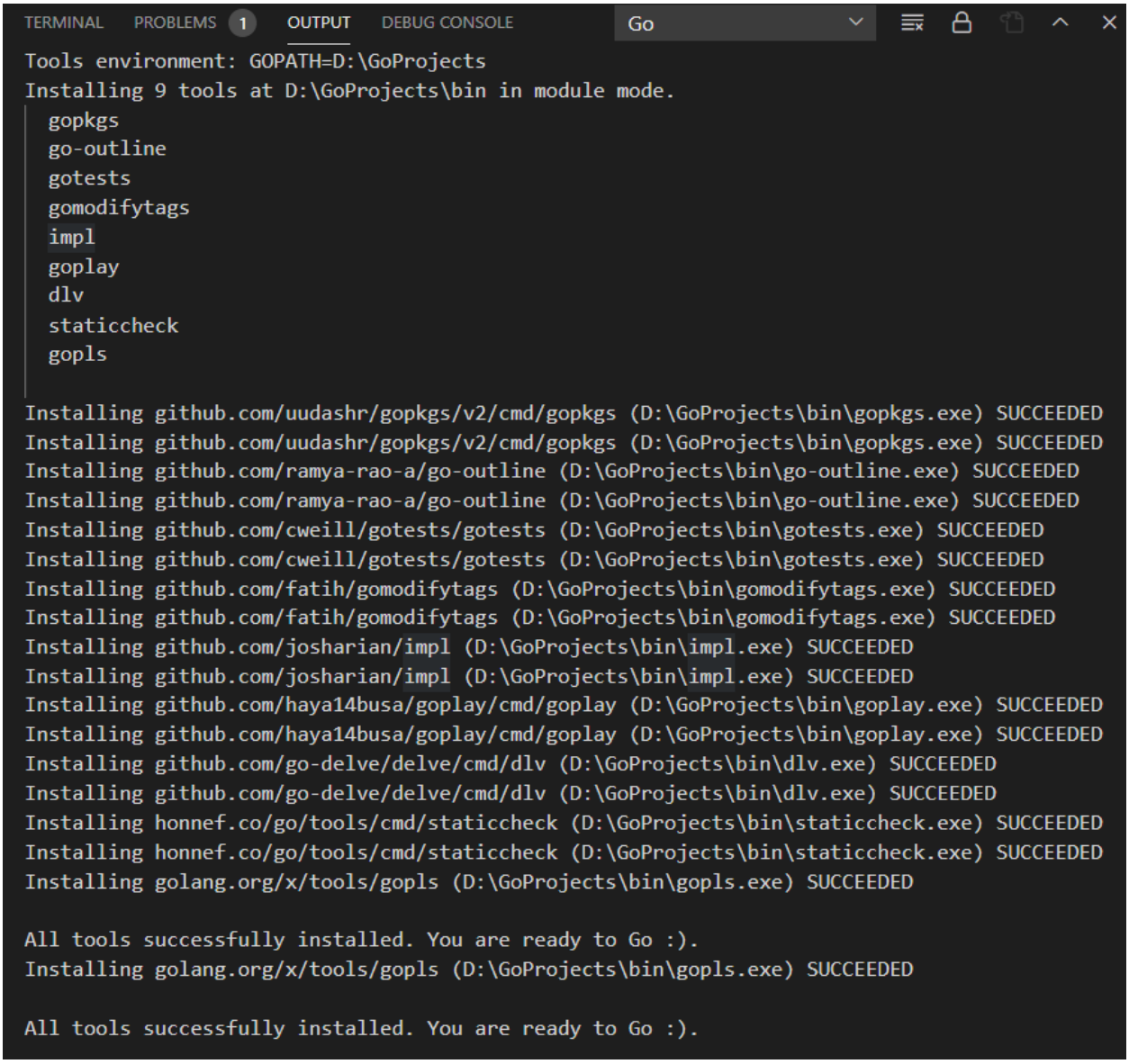
2、安装 go 插件



以管理员身份启动 VSCode，然后安装 go 插件

3、安装环境依赖

VSCode 打开前面创建的 HelloWorld 项目，打开 hello.go，右下角会提示安装环境依赖，点击安装就行



4、配置调试功能

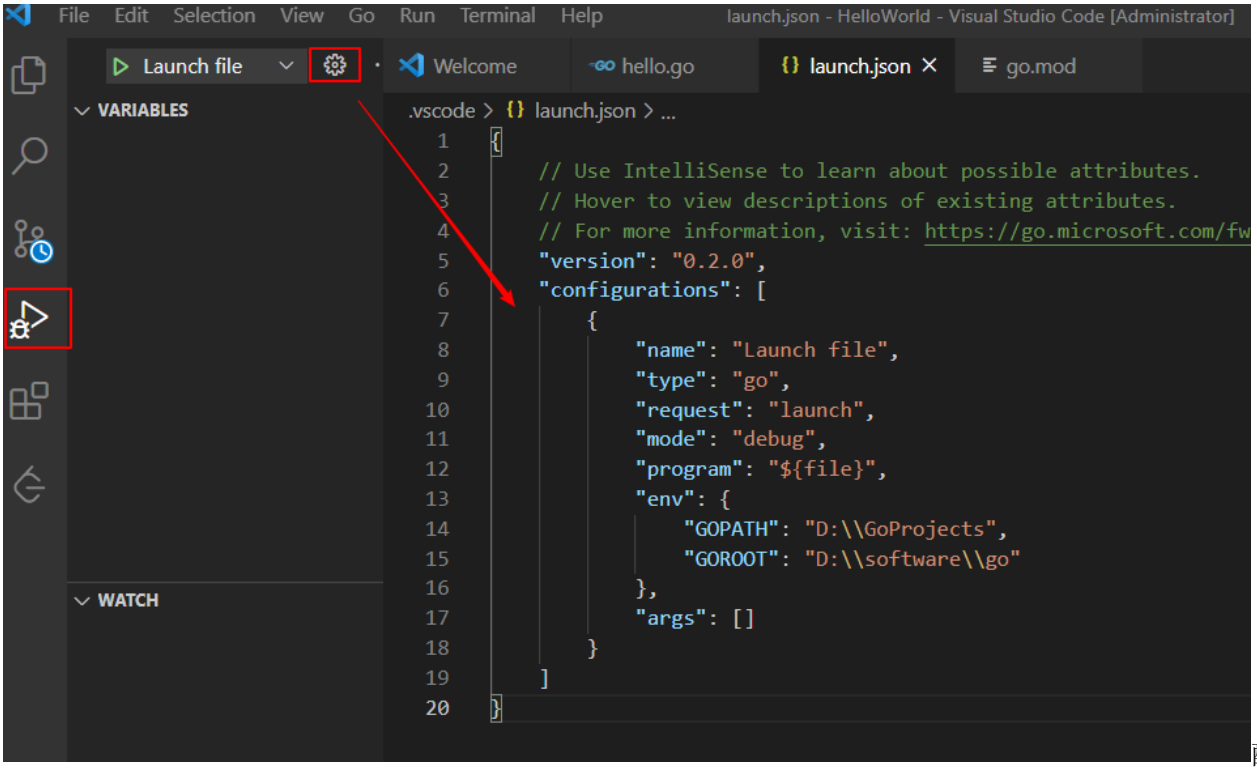
配置 launch.json 文件：

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Launch file",
            "type": "go",
            "request": "launch",
            "mode": "debug",
            "program": "${file}",
            "env": {
```

(continues on next page)

(continued from previous page)

```
                "GOPATH": "D:\\GoProjects",
                "GOROOT": "D:\\software\\go"
            },
            "args": []
        }
    ]
}
```



配置完成后在 go 代码文件内，按 F5 即可进入调试状态。

1.3.2 GoLand

也可以使用 JetBrains 公司推出的 GoLand 开发，功能更全面，下载地址：https://www.jetbrains.com/go/

1.4 学习资源

golang 国内站点：https://golang.google.cn/ 文档：https://golang.google.cn/doc/ Go 语言规范文档：https://golang.google.cn/ref/spec go 语言中文网：https://studygolang.com/

CHAPTER

TWO

GO 语言基础语法（一）

本文介绍一些 Go 语言的基础语法。

2.1 go 简单小例子

先来看一个简单的 go 语言代码：

```
package main
import "fmt"

// 加法运算
func add(x, y int) int {
        return x + y
}

func init() {
        fmt.Println("main  init....")
}

func main() {
        var value1 int = 2
        var value2 = 3
        sum := add(value1,value2)
        fmt.Printf("%d + %d = %d",value1,value2,sum)
}
```

- `package main`：定义 package 包名称为 main，表示当前文件所属的包
- `import "fmt"`：导入 Go 标准库中的 fmt 模块，主要用于打印输出。go 提供了很多标准库，具体可参考Golang 标准库文档。
- `init()`： init() 函数在 main() 函数之前执行。
- `main()`： main 函数，是当前程序的入口，init() 以及 main() 函数都无法被显式的调用。

go 语言的注释方法：

- 单行注释：`//`
- 多行注释：`/* */`

代码执行结果：

```
$ go run demo.go
main  init....
2 + 3 = 5
```

下面来进一步介绍 go 的基础语法。

2.2 格式化输出

go 语言中格式化输出可以使用 fmt 和 log 这两个标准库，

2.2.1 fmt

常用方法：

- `fmt.Printf`：格式化输出
- `fmt.Println`：仅打印，不能转义，会在输出结束后添加换行符。
- `fmt.Print`：和 Println 类似，但不添加换行符。
- `fmt.Sprintf`：格式化字符串并赋值给新的字符串

示例代码：

```
package main

import (
        "fmt"
)

func main() {
        var age = 22
        fmt.Printf("I'm %d years old\n", age)

        str1 := "Hello world !"
        fmt.Printf("%s\n", str1)
        fmt.Printf(str1)
        fmt.Print("\n")
        str_hex := fmt.Sprintf("% 02x", str1)
```

(continues on next page)

(continued from previous page)

```
        fmt.Printf("type of str_hex: %T\n", str_hex)
        fmt.Println(str_hex)
}
```

执行结果：

```
I'm 22 years old
Hello world !
Hello world !
type of str_hex: string
48 65 6c 6c 6f 20 77 6f 72 6c 64 20 21
```

更多格式化方法可以访问https://studygolang.com/pkgdoc中的 fmt 包。

2.2.2 log

log 包实现了简单的日志服务，也提供了一些格式化输出的方法。

- `log.Printf`：格式化输出，和 fmt.Printf 类似
- `log.Println`：和 fmt.Println 类似
- `log.Print`：和 fmt.Print 类似

```
package main

import (
        "log"
)

func main() {
        var age = 22
        log.Printf("I'm %d years old", age)

        str1 := "Hello world !"
        log.Println(str1)
        log.Print(str1)
        log.Printf("%s", str1)
}
```

执行结果：

```
2021/08/12 16:52:12 I'm 22 years old
2021/08/12 16:52:12 Hello world !
```

(continues on next page)

(continued from previous page)

```
2021/08/12 16:52:12 Hello world !
2021/08/12 16:52:12 Hello world !
```

下面来介绍一下 go 的数据类型

2.3 数据类型

下表列出了 go 语言的数据类型：

int、float、bool、string、数组和 struct 属于值类型，这些类型的变量直接指向存在内存中的值；slice、map、chan、pointer 等是引用类型，存储的是一个地址，这个地址存储最终的值。

2.4 常量声明

常量是在程序编译时就确定下来的值，程序运行时无法改变。

```
package main

import (
        "fmt"
)

func main() {
        const name string = "zhangsan"
        fmt.Println(name)

        const course1, course2 = "math", "english"
        fmt.Println(course1, course2)

        const age = 20
        age = age + 1 // 不能改变 age
        fmt.Println(age)
}
```

执行结果：

```
# command-line-arguments
.\test_const.go:15:6: cannot assign to age (declared const)
```

2.5 变量声明

go 的变量声明主要包括三种方法：

- 变量声明可指定变量类型，如果没有初始化，则变量默认为零值。
- 也可以不指定数据类型，由 go 自己判断。
- var 可以省略，使用:= 进行声明。注意： := 左边的变量必须是没有声明新的变量，否则会编译错误。

```
package main

import (
        "fmt"
)

func main() {
        var name string = "zhangsan"
        fmt.Println(name)

        var hight float32
        fmt.Println(hight)

        var course1, course2 = "math", "english"
        fmt.Println(course1, course2)

        age := 20
        age = age + 1
        fmt.Println(age)

        var (
                name1 string = "zhangsan"
                name2 string = "lishi"
        )
        fmt.Println(name1, name2)
}
```

执行结果：

```
zhangsan
0
math english
21
zhangsan lishi
```

2.6 运算符

Go 语言的运算符主要包括算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符以及指针相关运算符。

算术运算符：

关系运算符：

逻辑运算符：

位运算符：

赋值运算符：

指针相关运算符：

2.7 条件语句

下面介绍一下 go 语言中的 if 语句和 switch 语句。另外还有一种控制语句叫 select 语句，通常与通道联用，这里不做介绍。

2.7.1 if 语句

if 语法格式如下：

```
if [布尔表达式] {
   // do something
}
```

if …else：

```
if [布尔表达式] {
   // do something
} else {
   // do something
}
```

else if：

```
if [布尔表达式] {
   // do something
} else if [布尔表达式] {
   // do something
} else {
```

(continues on next page)

(continued from previous page)

```
    // do something
}
```

示例代码：

```
package main

import "fmt"

func main() {
        var grade = 70
        if grade >= 90 {
                fmt.Println("A" )
        } else if grade < 90 && grade >= 80 {
                fmt.Println("B" )
        } else if grade < 80 && grade > 60 {
                fmt.Println("C" )
        } else {
                fmt.Println("D" )
        }
}
```

2.7.2 switch 语句

语法格式：

```
switch var1 {
    case cond1:
        // do something
    case cond2:
        // do something
    default:
        // do something: 条件都不满足时执行
}
```

另外，添加 fallthrough 会强制执行后面的 case 语句，不管下一条 case 语句是否为 true。

示例代码：

```
package main

import "fmt"

```

(continues on next page)

(continued from previous page)

```
func main() {
        var grade = "B"
        switch grade {
        case "A":
                fmt.Println(" 优秀")
        case "B":
                fmt.Println(" 良好")
                fallthrough
        case "C":
                fmt.Println(" 中等")
        default:
                fmt.Println(" 不及格")
        }
}
```

执行结果：

```
良好
中等
```

2.8 循环语句

下面介绍几种循环语句：

2.8.1 for 循环：使用分号

```
package main

import "fmt"

func main() {
        sum := 0

    for i := 1; i < 5; i++ {
                sum += i
        }
        fmt.Println(sum) // 10 (1+2+3+4)
}
```

2.8.2 实现 while 效果

```
package main

import "fmt"

func main() {
        sum := 0
        n := 0

        for n < 5 {
                sum += n
                n += 1
        }
        fmt.Println(sum) // 10 (1+2+3+4)
}
```

2.8.3 死循环

```
package main

import "fmt"

func main() {
        sum := 0
        for {
                sum++
        }
        fmt.Println(sum)
}
```

2.8.4 for range 遍历

```
package main

import "fmt"

func main() {
        strings := []string{"hello", "world"}
        for index, str := range strings {
                fmt.Println(index, str)
```

(continues on next page)

(continued from previous page)

```
        }
}
```

执行结果：

```
0 hello
1 world
```

2.8.5 退出循环

- continue：结束当前迭代，进入下一次迭代
- break：结束当前 for 循环

```
package main

import "fmt"

func main() {
        sum := 0
        for {
                sum++
                if sum%2 != 0 {
                        fmt.Println(sum)
                        continue
                }
                if sum >= 10 {
                        break
                }
        }
        fmt.Println(sum)
}
```

执行结果：

```
1
3
5
7
9
10
```

也可以通过标记退出循环：

```
package main

import "fmt"

func main() {
        sum := 0
        n := 0
        LOOP: for n <= 10 {
                if n == 8 {
                        break LOOP
                }
                sum += n
                n++
        }
        fmt.Println(sum) // 28 (0+1+2+3+4+5+6+7)
}
```

2.8.6 goto 语句

```
package main

import "fmt"

func main() {
        sum := 0
        sum2 := 0
        n := 0

        LOOP: for n <= 10 {
                if n%2 == 0 {
                        sum += n
                        n++
                        goto LOOP
                }
                sum2 += n
                n++
        }
        fmt.Println(sum) // 30 (0+2+4+6+8+10)
        fmt.Println(sum2) // 25 (1+3+5+7+9)
}
```

CHAPTER

THREE

GO 语言基础语法（二）：函数

函数是一等（first-class）公民，可用来封装代码。在Go 语言基础语法（一）中介绍了函数也是一种数据类型，函数的值也可以在其他函数间传递、赋予变量、做类型判断和转换等。下面来介绍 Go 语言中的函数定义和使用方法。

3.1 普通函数声明与使用

下面先来介绍函数的简单使用方法。

函数定义语法：

```
func function_name( parameter-list ) ( return-types ) {
   // 函数体
}
```

Go 函数使用 `func` 关键字进行声明，输入参数和返回值都是可选的，可以没有参数，也可以没有返回值，函数体实现函数的功能逻辑。

除法运算例子：

```
package main

import (
        "fmt"
        "errors"
)

func add(x int, y int) (float64, error) {
        if y == 0 {
                return 0, errors.New("can't divide by zero!!")
        }
        res := float64(x) / float64(y)
        return res, nil
```

(continues on next page)

(continued from previous page)

```
}

func main() {
        value1 := 3
        value2 := 2
    value3 := 0
        res, err := add(value1, value2)
        fmt.Printf("%d / %d = %f (error: %v)\n", value1, value2, res, err)
        res, err = add(value1, value3)
        fmt.Printf("%d / %d = %f (error: %v)\n", value1, value3, res, err)
}
```

执行结果：

```
3 / 2 = 1.500000 (error: <nil>)
3 / 0 = 0.000000 (error: can't divide by zero!!)
```

3.2 函数类型

前面说了函数也是一种数据类型，函数类型的声明语法如下：

```
type function_name func(parameter-list) (return-types)
```

函数类型的函数签名（参数列表和结果列表）方法与函数声明一致，只要两个函数的函数签名一致（元素顺序和类型相同），它们就是相同的函数类型。

在前面除法运算例子中声明一个名为 calculate 的函数类型：

```
type calculate func(x int, y int) (float64, error)
```

函数签名和 add 函数一样，所以 add 和 calculate 是相同的函数类型。

```
var cal calculate
cal = add
res, err = cal(3,2)
fmt.Printf("The result: %f (error: %v)\n", res, err)
```

执行结果：

```
The result: 1.500000 (error: <nil>)
```

3.3 高阶函数

高阶函数和普通函数的区别在于高阶函数的形参或者返回参数列表中存在函数类型，也就是接收函数作为参数输入或者返回一个函数。

下面使用高阶函数实现加减乘除运算。

```
package main

import (
        "errors"
        "fmt"
)

type operate func(x, y int) int

func calculate(x int, y int, op operate) (int, error) {
        if op == nil {
                return 0, errors.New("invalid operation")
        }
        return op(x, y), nil
}

func add(x, y int) int {
        return x + y
}

func sub(x, y int) int {
        return x - y
}

func multiply(x, y int) int {
        return x * y
}

func divide(x, y int) int {
        return x / y
}

func main() {
        x, y := 36, 6

        result, _ := calculate(x, y, add)
        fmt.Println("The result: ",result)
```

(continues on next page)

(continued from previous page)

```
        result, _ = calculate(x, y, sub)
        fmt.Println("The result: ",result)

        result, _ = calculate(x, y, multiply)
        fmt.Println("The result: ",result)

        result, _ = calculate(x, y, divide)
        fmt.Println("The result: ",result)

        result, _ = calculate(x, y, nil)
        fmt.Println("The result: ",result)
}
```

执行结果：

```
The result:  42
The result:  30
The result:  216
The result:  6
The result:  0
```

3.4 闭包函数

闭包函数是引用了自由变量的代码块，闭包可以作为函数对象或者匿名函数。下面用闭包实现计算一个数的 n 次幂：

```
package main

import (
        "fmt"
)

type exponent func(uint64) uint64

func nth_power(exp uint64) exponent {
        return func(base uint64) uint64 {
                result := uint64(1)
                for i := exp ; i > 0; i >>= 1 {
                        if i&1 != 0 {
                                result *= base
                        }
```

(continues on next page)

(continued from previous page)

```
                        base *= base
                }
                return result
        }
}

func main() {
        square := nth_power(2) // 平方
        cube := nth_power(3) // 立方
        fmt.Println(square(5))
        fmt.Println(cube(5))
}
```

执行结果：

```
25
125
```

从代码中可以看出闭包返回的是一个函数，不是具体的值，使用闭包可以根据需要生成功能不同的函数。

3.5 参数传递

我在Python 函数的参数类型中介绍过 Python 函数中的参数传递，Python 中的参数传递属于对象的引用传递，而 Go 语言中均为**值传递**。

```
package main

import "fmt"


func modifyArray(a [3]int) [3]int {
        a[1] = 0
        return a
}

func modifySlice(a []int) []int {
        a[1] = 0
        return a
}

func main() {
        l1 := [3]int{1, 2, 3}
```

(continues on next page)

(continued from previous page)

```
        fmt.Println("value of l1:  ",l1)
        fmt.Printf("address of l1: %p\n",&l1)

        l2 := modifyArray(l1)
        fmt.Printf("address of l2: %p\n",&l2)
        fmt.Println("value of l1:  ",l1)
        fmt.Println("value of l2:  ",l2)

        slice1 := []int{1, 2, 3}
        fmt.Println("value of slice1:  ",slice1)
        fmt.Printf("address of slice1: %p\n",&slice1)

        slice2 := modifySlice(slice1)
        fmt.Printf("address of slice2: %p\n",&slice2)
        fmt.Println("value of slice1:  ",slice1)
        fmt.Println("value of slice2:  ",slice2)

        slice2[2] = 6
        fmt.Println("value of slice1:  ",slice1)
        fmt.Println("value of slice2:  ",slice2)

}
```

执行结果：

```
value of l1:   [1 2 3]
address of l1: 0xc000016198
address of l2: 0xc0000161c8
value of l1:   [1 2 3]
value of l2:   [1 0 3]
value of slice1:   [1 2 3]
address of slice1: 0xc000004078
address of slice2: 0xc0000040a8
value of slice1:   [1 0 3]
value of slice2:   [1 0 3]
value of slice1:   [1 0 6]
value of slice2:   [1 0 6]
```

由于数组是值类型，传给函数的参数值都会被复制，所以使用 modifyArray 对原数组进行修改时原数组不会改变，只是修改了它的副本而已，这和 Python 中的 list 不一样。

而对于引用类型，比如：切片、字典、通道，使用上面代码中的方式修改时，不会拷贝它们引用的底层数据，只是进行了浅表复制。所以上面例子中的原切片 slice1 也会跟着改变。

对于引用类型可以使用 copy 函数进行拷贝：

```
package main

import "fmt"

func main() {
        slice1 = []int{1, 2, 3}
        slice3 := make([]int, len(slice1))
        copy(slice3, slice1)
        slice3[1] = 6
        fmt.Printf("address of slice1: %p\n",&slice1)
        fmt.Printf("address of slice3: %p\n",&slice3)
        fmt.Println("value of slice1:  ",slice1)
        fmt.Println("value of slice3:  ",slice3)
}
```

执行结果：

```
address of slice1: 0xc000098060
address of slice3: 0xc000098108
value of slice1:   [1 2 3]
value of slice3:   [1 6 3]
```

CHAPTER

FOUR

GO 语言基础语法（三）：结构体及方法

结构体类型可以用来保存不同类型的数据，也可以通过方法的形式来声明它的行为。本文将介绍 go 语言中的结构体和方法，以及“继承”的实现方法。

4.1 结构体类型

结构体类型（struct）在 go 语言中具有重要地位，它是实现 go 语言面向对象编程的重要工具。go 语言中没有类的概念，可以使用结构体实现类似的功能，传统的 OOP（Object-Oriented Programming）思想中的继承在 go 中可以通过嵌入字段的方式实现。

结构体的声明与定义：

```
// 使用关键字 type 和 struct 定义名字为 Person 结构体
type Robot struct {
        name string
        height int
}
```

初始化及赋值：

```
// 通过 var 声明
var r1 Robot
r1.name = "Optimus Prime"

// 字面量直接赋值
r2 := Robot{name: "Optimus Prime"}
r3 := Robot{"Optimus Prime", 100} //如果不加字段名，值必须按定义顺序给出

// new 函数
r4 := new(Robot)
r4.name = "Optimus Prime"
//或者
```

(continues on next page)

(continued from previous page)

```
r5 := &Robot{}
r5.name = r1.name
```

4.2 方法

go 语言中的函数和方法是有区别的，方法必须有名字，必须隶属于某一个类型，这个类型通过方法声明中的接收者（receiver）声明定义。

接收者声明位于关键字 func 和方法名称之间的圆括号中，必须包含确切的名称和类型字面量。

- 类型就是当前方法所属的类型
- 名称用于当前方法中引用它所属类型的值

```
package main

import "fmt"

type Robot struct {
        name string
        height int
}

func (r Robot) String() string{
        return fmt.Sprintf("name: %s, height: %d",r.name, r.height)
}

func main() {
    r1 := Robot{name: "Optimus Prime", height: 100}
        fmt.Println(r1)  // 结果:  name: Optimus Prime, height: 100
}
```

从 String() 方法的接收者声明可以看出它隶属于 Robot 类型，接收者名称为 r。

4.3 结构体内嵌："继承" 与 "重写"

Go 语言中没有继承的概念，具体原因和理念可参考官网：Why is there no type inheritance?

go 语言可以通过嵌入字段来实现类似继承的效果，来看下面的代码：

```
package main
```

(continues on next page)

(continued from previous page)

```
import "fmt"

type Skills struct {
        speak string
}

func (s Skills) Speak()  {
        fmt.Println(s.speak)
}

type Robot struct {
        name string // 姓名
        height int // 身高
        Skills
}

func main() {
        skill := Skills{speak: "hello !"}
        skill.Speak()

        robot := Robot{
                name: "Optimus Prime",
                Skills: skill,
        }
        robot.Speak()
}
```

嵌入字段的方法集合会被合并到被嵌入类型的方法集合中。上面代码中，`robot.Speak()` 会调用嵌入字段 Skills 的 Speak() 方法，类似于继承了 Skills 的 Speak() 方法。执行结果如下：

```
hello !
hello !
```

下面添加一个 Robot 类型的 Speak() 方法：

```
func (r Robot) Speak() {
        fmt.Printf("My name is %s, ",r.name)
        r.Skills.Speak()
}
```

那么再次执行，会执行哪个 Speak() 方法呢？答案是 Robot 类型的 Speak() 方法，嵌入字段 Skills 的 Speak() 方法被“屏蔽”了，也就是说，被嵌入类型的方法覆盖了嵌入字段的同名方法，这与方法重写类似。

执行结果：

```
hello !
My name is Optimus Prime, hello !
```

可以通过链式的选择表达式，选择到嵌入字段的字段或方法，`r.Skills.Speak()` 就调用了嵌入字段 Skills 的 Speak() 方法。

4.4 小结

需要注意的是 Go 语言虽然支持面向对象编程，但是它没有继承的概念，可以通过嵌入字段的方式来实现类似继承的功能，这种组合方法相比多重继承更加简洁。

CHAPTER

FIVE

GO 语言中的字符串拼接方法介绍

本文介绍 Go 语言中的 string 类型、strings 包和 bytes.Buffer 类型，介绍几种字符串拼接方法。

5.1 string 类型

string 类型的值可以拆分为一个包含多个字符（rune 类型）的序列，也可以被拆分为一个包含多个字节 (byte 类型) 的序列。其中一个 rune 类型值代表一个 Unicode 字符，一个 rune 类型值占用四个字节，底层就是一个 UTF-8 编码值，它其实是 int32 类型的一个别名类型。

```
package main

import (
        "fmt"
)

func main() {
        str := " 你好 world"
        fmt.Printf("The string: %q\n", str)
        fmt.Printf("runes(char): %q\n", []rune(str))
        fmt.Printf("runes(hex): %x\n", []rune(str))
        fmt.Printf("bytes(hex): [% x]\n", []byte(str))
}
```

执行结果：

```
The string: " 你好 world"
runes(char): [' 你' ' 好' 'w' 'o' 'r' 'l' 'd']
runes(hex): [4f60 597d 77 6f 72 6c 64]
bytes(hex): e4 bd a0 e5 a5 bd 77 6f 72 6c 64
```

可以看到，英文字符使用一个字节，而中文字符需要三个字节。下面使用 `for range` 语句对上面的字符串进行遍历：

```
for index, value := range str {
    fmt.Printf("%d: %q [% x]\n", index, value, []byte(string(value)))
}
```

执行结果如下：

```
0: ' 你' [e4 bd a0]
3: ' 好' [e5 a5 bd]
6: 'w' [77]
7: 'o' [6f]
8: 'r' [72]
9: 'l' [6c]
10: 'd' [64]
```

index 索引值不是 0-6，相邻 Unicode 字符的索引值不一定是连续的，因为中文字符占用了 3 个字节，宽度为 3。

5.2 strings 包

5.2.1 strings.Builder 类型

strings.Builder 的优势主要体现在字符串拼接上，相比使用 + 拼接，效率更高。

- strings.Builder 已存在的值不可改变，只能重置（Reset() 方法）或者拼接更多的内容。
- 一旦调用了 Builder 值，就不能再以任何方式对其进行复制，比如函数间值传递、通道传递值、把值赋予变量等。
- 在进行拼接时，Builder 值会自动地对自身的内容容器进行扩容，也可以使用 Grow 方法进行手动扩容。

```
package main

import (
        "fmt"
        "strings"
)
func main() {
        var builder1 strings.Builder
        builder1.WriteString("hello")
        builder1.WriteByte(' ')
        builder1.WriteString("world")
        builder1.Write([]byte{' ', '!'})

        fmt.Println(builder1.String())
```

(continues on next page)

(continued from previous page)

```
        f1 := func(b strings.Builder) {
                // b.WriteString("world !")  //会报错
        }
        f1(builder1)

        builder1.Reset()
        fmt.Printf("The length 0f builder1: %d\n", builder1.Len())

}
```

执行结果：

```
hello world !
The length 0f builder1: 0
```

5.2.2 strings.Reader 类型

strings.Reader 类型可以用于高效地读取字符串，它通过使用**已读计数**机制来实现了高效读取，已读计数保存了已读取的字节数，也代表了下一次读取的起始索引位置。

```
package main

import (
        "fmt"
        "strings"
)
func main() {
        reader1 := strings.NewReader("hello world!")
        buf1 := make([]byte, 6)
    fmt.Printf("reading index: %d\n", reader1.Size()-int64(reader1.Len()))

    reader1.Read(buf1)
        fmt.Println(string(buf1))
    fmt.Printf("reading index: %d\n", reader1.Size()-int64(reader1.Len()))

        reader1.Read(buf1)
        fmt.Println(string(buf1))
    fmt.Printf("reading index: %d\n", reader1.Size()-int64(reader1.Len()))
}
```

执行结果：

```
reading index: 0
hello
reading index: 6
world!
reading index: 12
```

可以看到，每读取一次之后，已读计数就会增加。

strings 包的 **ReadAt 方法**不会依据已读计数进行读取，也不会更新已读计数。它可以根据偏移量来自由地读取 Reader 值中的内容。

```
package main

import (
        "fmt"
        "strings"
)
func main() {
    reader1 := strings.NewReader("hello world!")
    buf1 := make([]byte, 6)
        offset1 := int64(6)
        n, _ := reader1.ReadAt(buf1, offset1)
        fmt.Println(string(buf2))
}
```

执行结果：

```
world!
```

也可以使用 **Seek 方法**来指定下一次读取的起始索引位置。

```
package main

import (
        "fmt"
        "strings"
    "io"
)
func main() {
    reader1 := strings.NewReader("hello world!")
    buf1 := make([]byte, 6)
        offset1 := int64(6)
        readingIndex, _ := reader2.Seek(offset1, io.SeekCurrent)
        fmt.Printf("reading index: %d\n", readingIndex)
```

(continues on next page)

(continued from previous page)

```
        reader1.Read(buf1)
        fmt.Printf("reading index: %d\n", reader1.Size()-int64(reader1.Len()))
        fmt.Println(string(buf1))
}
```

执行结果：

```
reading index: 6
reading index: 12
world!
```

5.3 bytes.Buffer

bytes 包和 strings 包类似，strings 包主要面向的是 Unicode 字符和经过 UTF-8 编码的字符串，而 bytes 包面对的则主要是字节和字节切片，主要作为字节序列的缓冲区。bytes.Buffer 数据的读写都使用到了已读计数。

bytes.Buffer 具有读和写功能，下面分别介绍他们的简单使用方法。

5.3.1 bytes.Buffer：写数据

和 strings.Builder 一样，bytes.Buffer 可以用于拼接字符串，strings.Builder 也会自动对内容容器进行扩容。请看下面的代码：

```
package main

import (
        "bytes"
        "fmt"
)

func DemoBytes() {
        var buffer bytes.Buffer
        buffer.WriteString("hello ")
        buffer.WriteString("world !")
        fmt.Println(buffer.String())
}
```

执行结果：

```
hello world !
```

5.3.2 bytes.Buffer：读数据

bytes.Buffer 读数据也使用了已读计数，需要注意的是，进行读取操作后，Len 方法返回的是未读内容的长度。下面直接来看代码：

```
package main

import (
        "bytes"
        "fmt"
)

func DemoBytes() {
        var buffer bytes.Buffer
        buffer.WriteString("hello ")
        buffer.WriteString("world !")

    p1 := make([]byte, 5)
        n, _ := buffer.Read(p1)

        fmt.Println(string(p1))
        fmt.Println(buffer.String())
    fmt.Printf("The length of buffer: %d\n", buffer.Len())
}
```

执行结果：

```
hello
 world !
The length of buffer: 8
```

5.4 字符串拼接

简单了解了 string 类型、strings 包和 bytes.Buffer 类型后，下面来介绍 golang 中的字符串拼接方法。

https://zhuanlan.zhihu.com/p/349672248

go test -bench=. -run=^BenchmarkDemoBytes$

5.4.1 直接相加

最简单的方法是直接相加，由于 string 类型的值是不可变的，进行字符串拼接时会生成新的字符串，将拼接的字符串依次拷贝到一个新的连续内存空间中。如果存在大量字符串拼接操作，使用这种方法非常消耗内存。

```
package main

import (
        "bytes"
        "fmt"
        "time"
)

func main() {
        str1 := "hello "
        str2 := "world !"
    str3 := str1 + str2
    fmt.Println(str3)
}
```

5.4.2 strings.Builder

前面介绍了 strings.Builder 可以用于拼接字符串：

```
var builder1 strings.Builder
builder1.WriteString("hello ")
builder1.WriteString("world !")
```

5.4.3 strings.Join()

也可以使用 strings.Join 方法，其实 Join() 调用了 WriteString 方法；

```
str1 := "hello "
str2 := "world !"
str3 := ""

str3 = strings.Join([]string{str3,str1},"")
str3 = strings.Join([]string{str3,str2},"")
```

5.4.4 bytes.Buffer

bytes.Buffer 也可以用于拼接：

```
var buffer bytes.Buffer

buffer.WriteString("hello ")
buffer.WriteString("world !")
```

5.4.5 append 方法

也可以使用 Go 内置函数 append 方法，用于拼接切片：

```
package main

import (
        "fmt"
)

func DemoAppend(n int) {
        str1 := "hello "
        str2 := "world !"
        var str3 []byte

    str3 = append(str3, []byte(str1)...)
    str3 = append(str3, []byte(str2)...)
        fmt.Println(string(str3))
}
```

执行结果：

```
hello world !
```

5.4.6 fmt.Sprintf

fmt 包中的 Sprintf 方法也可以用来拼接字符串：

```
str1 := "hello "
str2 := "world !"
str3 := fmt.Sprintf("%s%s", str1, str2)
```

5.5 字符串拼接性能测试

下面来测试一下这 6 种方法的性能，编写测试源码文件 strcat_test.go：

```
package benchmark

import (
        "bytes"
        "fmt"
        "strings"
        "testing"
)

func DemoBytesBuffer(n int) {
        var buffer bytes.Buffer

        for i := 0; i < n; i++ {
                buffer.WriteString("hello ")
                buffer.WriteString("world !")
        }
}

func DemoWriteString(n int) {
        var builder1 strings.Builder
        for i := 0; i < n; i++ {
                builder1.WriteString("hello ")
                builder1.WriteString("world !")
        }
}

func DemoStringsJoin(n int) {
        str1 := "hello "
        str2 := "world !"
        str3 := ""
        for i := 0; i < n; i++ {
                str3 = strings.Join([]string{str3, str1}, "")
                str3 = strings.Join([]string{str3, str2}, "")
        }

}

func DemoPlus(n int) {

        str1 := "hello "
```

(continues on next page)

(continued from previous page)

```
        str2 := "world !"
        str3 := ""
        for i := 0; i < n; i++ {
                str3 += str1
                str3 += str2
        }
}

func DemoAppend(n int) {

        str1 := "hello "
        str2 := "world !"
        var str3 []byte
        for i := 0; i < n; i++ {
                str3 = append(str3, []byte(str1)...)
                str3 = append(str3, []byte(str2)...)
        }
}

func DemoSprintf(n int) {
        str1 := "hello "
        str2 := "world !"
        str3 := ""
        for i := 0; i < n; i++ {
                str3 = fmt.Sprintf("%s%s", str3, str1)
                str3 = fmt.Sprintf("%s%s", str3, str2)
        }
}

func BenchmarkBytesBuffer(b *testing.B) {
        for i := 0; i < b.N; i++ {
                DemoBytesBuffer(10000)
        }
}

func BenchmarkWriteString(b *testing.B) {
        for i := 0; i < b.N; i++ {
                DemoWriteString(10000)
        }
}

func BenchmarkStringsJoin(b *testing.B) {
        for i := 0; i < b.N; i++ {
```

(continues on next page)

(continued from previous page)

```
                DemoStringsJoin(10000)
        }
}

func BenchmarkAppend(b *testing.B) {
        for i := 0; i < b.N; i++ {
                DemoAppend(10000)
        }
}

func BenchmarkPlus(b *testing.B) {
        for i := 0; i < b.N; i++ {
                DemoPlus(10000)
        }
}

func BenchmarkSprintf(b *testing.B) {
        for i := 0; i < b.N; i++ {
                DemoSprintf(10000)
        }
}
```

执行性能测试:

```
$ go test -bench=. -run=^$
goos: windows
goarch: amd64
pkg: testGo/benchmark
cpu: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
BenchmarkBytesBuffer-8              3436            326846 ns/op
BenchmarkWriteString-8              4148            271453 ns/op
BenchmarkStringsJoin-8                 3         402266267 ns/op
BenchmarkAppend-8                   1923            618489 ns/op
BenchmarkPlus-8                        3         345087467 ns/op
BenchmarkSprintf-8                     2         628330850 ns/op
PASS
ok      testGo/benchmark        9.279s
```

通过平均耗时可以看到 WriteString 方法执行效率最高。Sprintf 方法效率最低。

1. 我们看到 Strings.Join 方法效率也比较低，在上面的场景下它的效率比较低，它在合并已有字符串数组的场合效率是很高的。
2. 如果要连续拼接大量字符串推荐使用 WriteString 方法，如果是少量字符串拼接，也可以直接使用 +。
3. append 方法的效率也是很高的，它主要用于切片的拼接。

4. fmt.Sprintf 方法虽然效率低，但在少量数据拼接中，如果你想拼接其它数据类型，使用它可以完美的解决：

```
name := "zhangsan"
age := 20
str4 := fmt.Sprintf("%s is %d years old", name, age)
fmt.Println(str4)  // zhangsan is 20 years old
```

CHAPTER

SIX

GO 语言中的通道

通道（channel）是 Go 语言中一种特殊的数据类型，通道本身就是并发安全的，可以通过它在多个 goroutine 之间传递数据。通道是 Go 语言编程理念：“*Do not communicate by sharing memory; instead, share memory by communicating*”（不要通过共享数据来通信，而应该通过通信来共享数据。）的完美实现，在并发编程中经常会遇到它。下面来介绍一下通道的使用方法。

6.1 通道的发送和接收

通道包括双向通道和单向通道，这里双向通道只的是支持发送和接收的通道，而单向通道是只能发送或者只能接收的通道。

6.1.1 双向通道

使用 make 函数声明并初始化一个通道：

```
ch1 := make(chan string, 3)
```

- `chan` 是表示通道类型的关键字
- `string` 表示该通道类型的元素类型
- `3` 表示该通道的容量为 3，最多可以缓存 3 个元素值。

一个通道相当于一个先进先出（FIFO）的队列，使用操作符 <- 进行元素值的发送和接收：

```
ch1 <- "1"  //向通道 ch1 发送数据 "1"
```

接收元素值：

```
elem1 := <- ch1 // 接收通道中的元素值
```

首先接收到的元素为先存入通道中的元素值，也就是先进先出：

```
package main

import "fmt"

func main() {
        str1 := []string{"hello","world", "!"}
        ch1 := make(chan string, len(str1))

        for _, str := range str1 {
                ch1 <- str
        }

        for i := 0; i < len(str1); i++ {
                elem := <- ch1
                fmt.Println(elem)
        }
}
```

执行结果：

```
hello
world
!
```

6.1.2 单向通道

单向通道包括只能发送的通道和只能接收的通道：

```
var WriteChan = make(chan<- interface{}, 1) // 只能发送不能接收的通道
var ReadChan = make(<-chan interface{}, 1) // 只能接收不能发送的通道
```

单向通道的这种特性可以用来约束函数的输入类型或者输出类型，比如下面的例子约束了只能从通道中接收元素值：

```
package main

import (
        "fmt"
)

func OnlyReadChan(num int) <-chan int {
        ch := make(chan int, 1)
        ch <- num
```

(continues on next page)

(continued from previous page)

```
        close(ch)
        return ch
}

func main() {

        Chan1 := OnlyReadChan(6)
        num := <- Chan1
        fmt.Println(num)
}
```

执行结果：

```
6
```

6.2 通道阻塞

通道操作是**并发安全**的，在同一时刻，只会执行对同一个通道的任意个发送操作中的某一个，直到这个元素值被完全复制进该通道之后，其他针对该通道的发送操作才可能被执行。接收操作也一样。另外，对于通道中的同一个元素值来说，发送操作和接收操作之间也是互斥的。

发送操作和接收操作是原子操作，也就是说，发送操作绝不会出现只复制了一部分的情况，要么还没有复制，要么已经复制完毕。接收操作在准备好元素值的副本之后，一定会删除掉通道中的原值，绝不会出现通道中仍有残留的情况。在进行发送操作和接收操作时，代码会一直阻塞在那里，完成操作后才会继续执行后面的代码。通道的发送操作和接收操作是很快的，那么什么情况下会出现长时间的阻塞呢？下面介绍几种情况。

6.2.1 缓冲通道的阻塞

缓冲通道是容量大于 0 的通道，也就是可以缓存数据的通道。

1、发送阻塞

如果缓冲通道已经填满，如果有 goroutine 继续向该通道发送数据就会阻塞。请看下面的例子：

```
package main

func main() {
        ch1 := make(chan int, 1)
        ch1 <- 1
        ch1 <- 2
}
```

执行结果：

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
..........
```

如果通道可以接收数据（有元素被接收），通道会通知最先等待发送操作的 goroutine 再次执行发送操作。

2、接收阻塞

类似的，如果通道已空，如果继续进行接收操作就会被阻塞。

```
package main

func main() {
        ch1 := make(chan int, 1)
        <- ch1
}
```

执行结果：

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan receive]:
..........
```

6.2.2 非缓冲通道

非缓冲通道是容量为 0 的通道，不能缓存数据。

非缓冲通道的数据传递是同步的，发送操作或者接收操作在执行后就会阻塞，需要对应的接收操作或者发送操作执行才会继续传递。由此可以看出缓冲通道使用的是异步方式进行数据传递。

```
package main

import (
    "fmt"
)

func main() {
    str1 := []string{"hello","world", "!"}
        ch1 := make(chan string, 0)

    go func() {
        for _, str := range str1 {
            ch1 <- str
```

(continues on next page)

(continued from previous page)

```
        }
    }()

        for i := 0; i < len(str1); i++ {
                elem := <- ch1
                fmt.Println(elem)
        }

}
```

执行结果：

```
hello
world
!
```

上面的代码中 3 个 goroutine 向通道写了三次数据，必须有三次接收，不然会阻塞。

对值为 nil 的通道进行发送操作和接收操作也会发生阻塞：

```
var ch1 chan int
ch1 <- 1 // 阻塞
<-ch1 // 阻塞
```

6.3 通道关闭

可以使用 close() 方法来关闭通道，通道关闭后，不能再对通道进行发送操作，可以进行接收操作。

```
package main

import "fmt"

func main() {
        ch1 := make(chan int, 1)
        ch1 <- 1
        close(ch1)

    ele := <-ch1
        fmt.Println(ele)

        ch1 <- 2
}
```

执行结果：

```
1
panic: send on closed channel

goroutine 1 [running]:
.....
```

如果通道关闭时，里面还有元素，进行接收操作时，返回的通道关闭标志仍然为 true：

```
package main

import "fmt"

func main() {
        ch1 := make(chan int, 1)
        ch1 <- 1
        close(ch1)

        ele1, statu1 := <-ch1
        fmt.Println(ele1, statu1)
        ele2, statu2 := <-ch1
        fmt.Println(ele2, statu2)
}
```

执行结果：

```
1 true
0 false
```

由于通道的这种特性，可以让发送方来关闭通道。前面的例子可以这样写：

```
package main

import (
    "fmt"
)

func main() {
    str1 := []string{"hello","world", "!"}
        ch1 := make(chan string, 0)

    go func() {
        for _, str := range str1 {
            ch1 <- str
```

(continues on next page)

(continued from previous page)

```
        }
        close(ch1)
    }()

        for i := 0; i < len(str1); i++ {
                elem := <- ch1
                fmt.Println(elem)
        }

}
```

另外，不能对关闭的通道再次关闭：

```
package main

// import "fmt"

func main() {
        ch1 := make(chan int, 1)
        ch1 <- 1
        close(ch1)
        close(ch1)

}
```

执行结果：

```
panic: close of closed channel
```

6.4 select 语句与通道

select 语句通常与通道联用，它是专为通道而设计的。select 语句执行时，一般只有一个 case 表达式或者 default 语句会被运行。

```
package main

import "fmt"

func main() {
    ch1 := make(chan int, 1)
        num := 2
```

(continues on next page)

(continued from previous page)

```
        select {
                case data := <-ch1:
                        fmt.Println("Read data: ", data)
                case ch1 <- num:
                        fmt.Println("Write data: ", num)
                default:
                        fmt.Println("No candidate case is selected!")
        }
}
```

执行结果：

```
Write data:  2
```

需要注意的是，如果没有 default 默认分支，case 表达式都没有满足条件，那么 select 语句就会被阻塞，直到至少有一个 case 表达式满足条件为止。

如果同时有多个分支满足条件，会随机选择一个分支执行

for 语句与 select 语句联用时，分支中的 break 语句只能结束当前 select 语句的执行，而不会退出 for 循环。下面的代码永远不会退出循环：

```
package main

import "fmt"

func main() {
    ch1 := make(chan int, 1)
        for {
                select {
                case ch1 <- 6:
                        fmt.Println("Write data: 6")
                case data := <-ch1:
                        fmt.Println(data)
                        break
                }
        }
}
```

解决方案是使用 goto 语句和标签。

方法 1：

```
package main
```

(continues on next page)

(continued from previous page)

```
import "fmt"

func main() {
    ch1 := make(chan int, 1)
        num := 6
        for {
                select {
                case ch1 <- num:
                        fmt.Println("Write data: ", num)
                case data := <-ch1:
                        fmt.Println("Read data: ", data)
                        goto loop
                }
        }
        loop:
        fmt.Println(ch1)
}
```

执行结果:

```
Write data:  6
Read data:  6
0xc00000e0e0
```

方法 2:

```
package main

import "fmt"

func main() {
        ch1 := make(chan int, 1)
        num := 6
        loop:
        for {
                select {
                case ch1 <- num:
                        fmt.Println("Write data: ", num)
                case data := <-ch1:
                        fmt.Println("Read data: ", data)
                        break loop
                }
        }
        fmt.Println(ch1)
```

(continues on next page)

(continued from previous page)

```
}
```

执行结果：

```
Write data:  6
Read data:  6
0xc0000e4000
```

6.5 小结

本文主要介绍了通道的基本操作：初始化、发送、接收和关闭，要注意在什么情况下会引起通道阻塞。select 语句通常与通道联用，介绍了分支的选择规则以及 for 语句与 select 语句联用时如何退出循环。

通道是 Go 语言并发编程的重要实现基础，还是有必要掌握的。

CHAPTER

SEVEN

GO 语言并发编程：原子操作

在程序执行过程中，操作系统会进行线程调度，同一时刻能同时执行的程序数量跟 CPU 的内核线程数有关，比如 4 核 CPU，同时最多只能有 4 个线程。Go 语言中的运行时系统也会对 goroutine 进行调度，调度器会频繁地让 goroutine 处于中断或者运行状态，这就不能保证代码执行的原子性（atomicity），即使使用互斥锁也不能保证原子性操作。Go 语言中的 atomic 包提供了原子操作方法，下面来介绍它的使用方法。

原子操作过程中是不允许中断的，是绝对并发安全的。由于原子操作不允许中断，所以它非常影响系统执行效率，因此，Go 语言的 sync/atomic 包只针对少数数据类型提供了原子操作函数。

7.1 atomic 原子操作类型和方法

支持的数据类型主要有 7 个：int32、int64、uint32、uint64、uintptr，Pointer（unsafe 包）以及 Value 类型，Value 类型可以用来存储任意类型的值。

对这些类型的操作函数包括：

- 增加 (Add)：`atomic.AddInt32(addr *int32, delta int32)`
- 加载（Load）：`atomic.LoadInt32(addr *int32)`
- 存储（Store）：`atomic.LoadInt32(addr *int32)`
- 交换（Swap）：`atomic.SwapInt32(addr *int32, new int32)`
- 比较并交换（CompareAndSwap）：`atomic.CompareAndSwapInt32(addr *int32, old int32, new int32)`

其中，unsafe.Pointer 类型没有 add 操作，Value 类型只要 Load 和 Store 两个方法。

注意，第一个参数值为被操作值的指针，原子操作根据指针定位到该值的内存地址，操作这个内存地址上的数据。

7.2 Add 增加

Add 可以用于增加操作：

```
package main

import (
        "fmt"
        "sync/atomic"
)

func main() {
        num := int32(20)
        atomic.AddInt32(&num, 3)
        fmt.Println(num)   // 23
}
```

Add 也可以做减法操作，其中 AddInt32 的第二个参数 int32 是有符号整型，所以 delta 值设置为负整数就是减法操作了。

```
num := int32(18)
atomic.AddInt32(&num, -3)
fmt.Println(num)
```

而 uint32 和 uint64 是无符号的，如果想对这两种类型做减法操作需要做一下转换，比如先把 delta 值转换为有符号类型，然后再转换为无符号类型：

```
num := uint32(18)
delta := int32(-3)
atomic.AddUint32(&num, uint32(delta))
fmt.Println(num)
```

也可以使用如下方式：

```
atomic.AddUint32(&num, ^uint32(-(-3)-1))
```

7.3 Load 加载

Load 可以实现对值的原子读取：

```
num := int32(20)
atomic.LoadInt32(&num)
fmt.Println(atomic.LoadInt32(&num))
```

7.4 Store 存储

原子的存储某个值：

```
num := int32(20)
atomic.StoreInt32(&num, 30)
fmt.Println(num) // 30
```

7.5 Swap 交换

将新的值赋给被操作的旧值，并返回旧值

```
num := int32(20)
old := atomic.SwapInt32(&num, 60)
fmt.Println(num) // 60
fmt.Println(old) // 20
```

7.6 CompareAndSwap 比较并交换

比较并交换（Compare And Swap，CAS 操作）和交换（Swap）不同，会先进行比较，满足条件后再进行交换操作，将新值赋给变量。返回值为 true 或者 false，true 表示执行了交换操作。

```
num:= int32(18)
atomic.CompareAndSwapInt32(&num, 20, 0)
fmt.Printf("The number: %d\n", num)
atomic.CompareAndSwapInt32(&num, 18, 0)
fmt.Printf("The number: %d\n", num)
```

执行结果：

```
The number: 18
The number: 0
```

CAS 操作可以用来实现自旋锁（spinlock），下面先来介绍一下什么是自旋锁，自旋锁和互斥锁都可以用来保护共享资源，它们的区别在于，资源被互斥锁锁定时，其它要操作资源的线程会进入睡眠状态；如果是自旋锁，线程将循环等待，不会释放 cpu，直到获取到锁才会退出循环。由于自旋锁的这种特性，一般会对等待时间或者尝试次数进行一定的限制。

由于自旋锁不需要进行上下文切换，它的效率比互斥锁高，适用于保持锁的时间比较短，并且不会频繁操作共享资源的场景。

下面的代码实现一个简单的自旋锁，存满 10000 后全部取出：

```
package main

import (
        "fmt"
        "sync/atomic"
        "time"
        "sync"
)

var (
    balance int32
        wg sync.WaitGroup
)

// 存钱
func deposit(value int32) {
        for {

                fmt.Printf(" 余额: %d\n", balance)
                atomic.AddInt32(&balance, value)
                fmt.Printf(" 存 %d 后的余额: %d\n", value, balance)
                fmt.Println()
                if balance == 10000 {
                        break
                }
                time.Sleep(time.Millisecond * 500)
        }
    wg.Done()
}

// 取钱
func withdrawAll(value int32) {
    defer wg.Done()

        for {
                if atomic.CompareAndSwapInt32(&balance, value, 0) {
                        break
                }
                time.Sleep(time.Millisecond * 500)
        }
        fmt.Printf(" 余额: %d\n", value)
    fmt.Printf(" 取 %d 后的余额: %d\n", value, balance)
    fmt.Println()
}
```

(continues on next page)

(continued from previous page)

```
func main() {
        wg.Add(2)
        go deposit(1000)  // 每次存 1000
        go withdrawAll(10000)
    wg.Wait()

    fmt.Printf(" 当前余额: %d\n", balance)
}

func init() {
    balance = 1000 // 初始账户余额为 1000
}
```

7.7 atomic.Value

Value 类型可以被用来“原子地”存储（Store）和加载（Load）任意的值。

```
var valu atomic.Value
valu := [...]int{1, 2, 3}
box.Store(valu)
fmt.Println(valu.Load())
```

使用 Value 类型时需要注意以下事项：

1、Value 不能用来存储 nil 值。

2、一个 Value 变量不能存储不同类型的值，存储的类型只能是第一个存储值的类型。

```
var box atomic.Value
v1 := "123"
box.Store(v1)
v2 := 123
box.Store(v2)
```

上面的写法会引发一个 panic：`panic: sync/atomic: store of inconsistently typed value into Value`

3、尽量不要使用 Value 存储引用类型的值。

先来看下面的例子：

```
package main
```

(continues on next page)

(continued from previous page)

```
import (
        "fmt"
        "sync/atomic"
)

func main() {
    var valu atomic.Value
    v1 := []int{1, 2, 3}
    valu.Store(v1)
    fmt.Println(valu.Load())
    v1[1] = 6
    fmt.Println(valu.Load())
}
```

执行结果：

```
[1 2 3]
[1 6 3]
```

修改引用类型的值相当于修改了 valu 中存储的值，可以使用深拷贝 copy 方法来解决这个漏洞：

```
package main

import (
        "fmt"
        "sync/atomic"
)

func main() {
    var valu atomic.Value
    v1 := []int{1, 2, 3}
        store := func(v []int) {
                replica := make([]int, len(v))
                copy(replica, v)
                valu.Store(replica)
        }
        fmt.Printf("Store %v to box6.\n", v6)
        store(v1)
        fmt.Println(valu.Load())
    v1[1] = 6
    fmt.Println(valu.Load())
}
```

执行结果：

```
[1 2 3]
[1 2 3]
```

7.8 小结

原子操作函数支持的数据类型有限，互斥锁可能使用的场景更多一些，在可以使用原子操作的情况下还是建议使用它，因为相对来说原子操作函数的执行速度比互斥锁快，且使用简单。另外在使用 CAS 操作时，要防止进入死循环，导致“阻塞”流程。

在使用 Value 类型时要注意尽量不要存储引用类型的值，是非并发安全的。

CHAPTER

EIGHT

GO 语言并发编程：互斥锁

在并发编程中，多个 Goroutine 访问同一块内存资源时可能会出现竞态条件，我们需要在临界区中使用适当的同步操作来以避免竞态条件。Go 语言中提供了很多同步工具，本文将介绍互斥锁 Mutex 和读写锁 RWMutex 的使用方法。

8.1 互斥锁 Mutex

8.1.1 Mutex 介绍

Go 语言的同步工具主要由 sync 包提供，互斥锁 (Mutex) 与读写锁 (RWMutex) 就是 sync 包中的方法。

互斥锁可以用来保护一个临界区，保证同一时刻只有一个 goroutine 处于该临界区内。主要包括锁定（Lock 方法）和解锁（Unlock 方法）两个操作，首先对进入临界区的 goroutine 进行锁定，离开时进行解锁。

使用互斥锁 (Mutex) 时要注意以下几点：

1. 不要重复锁定互斥锁，否则会阻塞，也可能会导致死锁（deadlock）；
2. 要对互斥锁进行解锁，这也是为了避免重复锁定；
3. 不要对未锁定或者已解锁的互斥锁解锁；
4. 不要在多个函数之间直接传递互斥锁，sync.Mutex 类型属于值类型，将它传给一个函数时，会产生一个副本，在函数中对锁的操作不会影响原锁

总之，一个互斥锁只用来保护一个临界区，加锁后记得解锁，对于每一个锁定操作，都要有且只有一个对应的解锁操作，也就是加锁和解锁要成对出现，最保险的做法时使用 **defer 语句**解锁。

8.1.2 Mutex 使用实例

下面的代码模拟取钱和存钱操作：

```
package main

import (
        "flag"
        "fmt"
        "sync"
)

var (
    mutex   sync.Mutex
    balance int
    protecting uint  // 是否加锁
    sign = make(chan struct{}, 10) //通道，用于等待所有 goroutine
)

// 存钱
func deposit(value int) {
    defer func() {
        sign <- struct{}{}
    }()

    if protecting == 1 {
        mutex.Lock()
        defer mutex.Unlock()
    }

    fmt.Printf(" 余额: %d\n", balance)
    balance += value
    fmt.Printf(" 存 %d 后的余额: %d\n", value, balance)
    fmt.Println()

}

// 取钱
func withdraw(value int) {
    defer func() {
        sign <- struct{}{}
    }()

    if protecting == 1 {
        mutex.Lock()
```

(continues on next page)

(continued from previous page)

```
        defer mutex.Unlock()
    }

    fmt.Printf(" 余额: %d\n", balance)
    balance -= value
    fmt.Printf(" 取 %d 后的余额: %d\n", value, balance)
    fmt.Println()

}

func main() {

    for i:=0; i < 5; i++ {
        go withdraw(500) // 取 500
        go deposit(500)  // 存 500
    }

    for i := 0; i < 10; i++ {
                <-sign
        }
    fmt.Printf(" 当前余额: %d\n", balance)
}

func init() {
    balance = 1000 // 初始账户余额为 1000
    flag.UintVar(&protecting, "protecting", 0, " 是否加锁, 0 表示不加锁, 1 表示加锁")
}
```

上面的代码中，使用了通道来让主 goroutine 等待其他 goroutine 运行结束，每个子 goroutine 在运行结束之前向通道发送一个元素，主 goroutine 在最后从这个通道接收元素，接收次数与子 goroutine 个数相同。接收完后就会退出主 goroutine。

代码使用协程实现多次（5 次）对一个账户进行存钱和取钱的操作，先来看不加锁的情况：

```
余额: 1000
存 500 后的余额: 1500

余额: 1000
取 500 后的余额: 1000

余额: 1000
存 500 后的余额: 1500

余额: 1000
```

(continues on next page)

(continued from previous page)

```
取 500 后的余额: 1000

余额: 1000
存 500 后的余额: 1500

余额: 1000
取 500 后的余额: 1000

余额: 1000
取 500 后的余额: 500

余额: 1000
存 500 后的余额: 1000

余额: 1000
取 500 后的余额: 500

余额: 1000
存 500 后的余额: 1000

当前余额: 1000
```

可以看到出现了混乱，比如第二次 1000 的余额取 500 后还是 1000，这种对同一资源的竞争出现了竞态条件 (Race Condition)。

下面来看加锁的执行结果：

```
余额: 1000
取 500 后的余额: 500

余额: 500
存 500 后的余额: 1000

余额: 1000
取 500 后的余额: 500

余额: 500
存 500 后的余额: 1000

余额: 1000
取 500 后的余额: 500

余额: 500
存 500 后的余额: 1000
```

(continues on next page)

(continued from previous page)

```
余额： 1000
存 500 后的余额： 1500

余额： 1500
取 500 后的余额： 1000

余额： 1000
取 500 后的余额： 500

余额： 500
存 500 后的余额： 1000

当前余额： 1000
```

加锁后就正常了。

下面介绍更细化的互斥锁：读/写互斥锁 RWMutex。

8.2 读写锁 RWMutex

8.2.1 RWMutex 介绍

读/写互斥锁 RWMutex 包含了读锁和写锁，分别对共享资源的“读操作”和“写操作”进行保护。sync.RWMutex 类型中的 Lock 方法和 Unlock 方法分别用于对写锁进行锁定和解锁，而它的 RLock 方法和 RUnlock 方法则分别用于对读锁进行锁定和解锁。

有了互斥锁 Mutex，为什么还需要读写锁呢？因为在很多并发操作中，并发读取占比很大，写操作相对较少，读写锁可以并发读取，这样可以提供服务性能。读写锁具有以下特征：

也就是说，

- 如果某个共享资源受到读锁和写锁保护时，其它 goroutine 不能进行写操作。换句话说就是读写操作和写写操作不能并行执行，也就是读写互斥；
- 受读锁保护时，可以同时进行多个读操作。

在使用读写锁时，还需要注意：

1. 不要对未锁定的读写锁解锁；
2. 对读锁不能使用写锁解锁
3. 对写锁不能使用读锁解锁

8.2.2 RWMutex 使用实例

改写前面的取钱和存钱操作，添加查询余额的方法：

```
package main

import (
        "fmt"
        "sync"
)

// account 代表计数器。
type account struct {
        num uint          // 操作次数
        balance int                  // 余额
        rwMu  *sync.RWMutex // 读写锁
}

var sign = make(chan struct{}, 15) //通道，用于等待所有 goroutine

// 查看余额：使用读锁
func (c *account) check() {
        defer func() {
        sign <- struct{}{}
    }()
        c.rwMu.RLock()
        defer c.rwMu.RUnlock()
        fmt.Printf("%d 次操作后的余额：%d\n", c.num, c.balance)
}

// 存钱：写锁
func (c *account) deposit(value int) {
        defer func() {
        sign <- struct{}{}
    }()
    c.rwMu.Lock()
        defer c.rwMu.Unlock()

        fmt.Printf(" 余额：%d\n", c.balance)
        c.num += 1
    c.balance += value
    fmt.Printf(" 存 %d 后的余额：%d\n", value, c.balance)
    fmt.Println()
}
```

(continues on next page)

(continued from previous page)

```
// 取钱: 写锁
func (c *account) withdraw(value int) {
    defer func() {
        sign <- struct{}{}
    }()
        c.rwMu.Lock()
        defer c.rwMu.Unlock()
        fmt.Printf(" 余额: %d\n", c.balance)
        c.num += 1
    c.balance -= value
        fmt.Printf(" 取 %d 后的余额: %d\n", value, c.balance)
    fmt.Println()
}


func main() {
        c := account{0, 1000, new(sync.RWMutex)}

        for i:=0; i < 5; i++ {
        go c.withdraw(500) // 取 500
        go c.deposit(500)  // 存 500
                go c.check()
    }

    for i := 0; i < 15; i++ {
                <-sign
        }
        fmt.Printf("%d 次操作后的余额: %d\n", c.num, c.balance)

}
```

执行结果:

```
余额: 1000
取 500 后的余额: 500

1 次操作后的余额: 500
1 次操作后的余额: 500
1 次操作后的余额: 500
1 次操作后的余额: 500
1 次操作后的余额: 500
余额: 500
存 500 后的余额: 1000
```

(continues on next page)

(continued from previous page)

```
余额： 1000
取 500 后的余额： 500

余额： 500
存 500 后的余额： 1000

余额： 1000
存 500 后的余额： 1500

余额： 1500
取 500 后的余额： 1000

余额： 1000
取 500 后的余额： 500

余额： 500
存 500 后的余额： 1000

余额： 1000
取 500 后的余额： 500

余额： 500
存 500 后的余额： 1000

10 次操作后的余额： 1000
```

读写锁和互斥锁的不同之处在于读写锁把对共享资源的读操作和写操作分开了，可以实现更复杂的访问控制。

8.3 小结

读写锁也是一种互斥锁，它是互斥锁的扩展。在使用时需要注意：

1. 加锁后一定要解锁
2. 不要重复加锁或者解锁
3. 不解锁未锁定的锁
4. 不要传递互斥锁

CHAPTER

NINE

GO 语言并发编程：SYNC.ONCE

sync.Once 用于保证某个动作只被执行一次，可用于单例模式中，比如初始化配置。我们知道 init() 函数也只会执行一次，不过它是在 main() 函数之前执行，如果想要在代码执行过程中只运行某个动作一次，可以使用 sync.Once，下面来介绍一下它的使用方法。

先来看下面的代码：

```
package main

import (
        "fmt"
        "sync"
)


func main() {
        var num = 6
        var once sync.Once

        add_one := func() {
                num = num + 1
        }

        minus_one := func() {
                num = num - 1
        }

        once.Do(add_one)
        fmt.Printf("The num: %d\n", num)
        once.Do(minus_one)
        fmt.Printf("The num: %d\n", num)
}
```

执行结果：

```
The num: 7
The num: 7
```

sync.Once 类型提供了一个 Do 方法，Do 方法只接受一个参数，且参数类型必须是 func() ，也就是没有参数声明和结果声明的函数。

Do 方法只会执行首次被调用时传入的那个函数，只执行一次，也不会执行其它函数。上面的例子中，即使传入的函数不同，也只会执行第一次传入的那个函数。如果有多个只执行一次的函数，需要为每一个函数分配一个 sync.Once 类型的值：

```
func main() {
        var num = 6
        var once1 sync.Once
        var once2 sync.Once

        add_one := func() {
                num = num + 1
        }

        minus_one := func() {
                num = num - 1
        }

        once1.Do(add_one)
        fmt.Printf("The num: %d\n", num)
        once2.Do(minus_one)
        fmt.Printf("The num: %d\n", num)
}
```

sync.Once 类型是一个结构体类型，一个是名为 done 的 uint32 类型字段，还有一个互斥锁 m。

```
type Once struct {
        done uint32
        m    Mutex
}
```

done 字段的值只可能是 0 或者 1，Do 方法首次调用完成后，done 的值就变为了 1。done 的值使用四个字节的 uint32 类型的原因是为了保证对它的操作是“原子操作”，通过调用 atomic.LoadUint32 函数获取它的值，如果为 1，直接返回，不会执行函数。

如果为 0，Do 方法会立即锁定字段 m，如果这里不加锁，多个 goroutine 同时执行到 Do 方法时判断都为 0，则都会执行函数，所以 Once 是并发安全的。

加锁之后，会再次检查 done 字段的值，如果满足条件，执行传入的函数，并用原子操作函数 atomic.StoreUint32 将 done 的值设置为 1。

下面是 Once 的源码：

```
func (o *Once) Do(f func()) {

        if atomic.LoadUint32(&o.done) == 0 {
                o.doSlow(f)
        }
}

func (o *Once) doSlow(f func()) {
        o.m.Lock()
        defer o.m.Unlock()
        if o.done == 0 {
                defer atomic.StoreUint32(&o.done, 1)
                f()
        }
}
```

源码非常简洁，和 GoF 设计模式中的单例模式非常相似。

CHAPTER

TEN

GO 语言并发编程：WAITGROUP

我们知道，在并发编程中，主要线程需要等待子线程运行结束后才能退出，go 语言中，主 goroutine 等待其他 goroutine 运行结束可以使用通道来解决，具体实现可以参考文章Go 语言并发编程：互斥锁中的例子。使用通道可能不是很简洁，本文介绍另一种方法，也就是 sync 包中的 WaitGroup 类型来等待 goroutine 执行完成。

sync.WaitGroup 类型主要包括 3 个方法：

- Add：用于需要等待的 goroutine 的数量
- Done：对计数器的值进行减一操作，一般在需要等待的 goroutine 运行完成之前执行这一操作，可以通过 defer 语句调用它
- Wait：用于阻塞当前的 goroutine，直到其所属值中的计数器归零

下面直接修改Go 语言并发编程：互斥锁中的例子，使用 WaitGroup 来等待 goroutine：

```
package main

import (
        "flag"
        "fmt"
        "sync"
)

var (
    mutex   sync.Mutex
    balance int
    protecting uint  // 是否加锁
    sign = make(chan struct{}, 10) //通道，用于等待所有 goroutine
)

var wg sync.WaitGroup

// 存钱
func deposit(value int) {
    if protecting == 1 {
```

(continues on next page)

(continued from previous page)

```
        mutex.Lock()
        defer mutex.Unlock()
    }

    fmt.Printf(" 余额: %d\n", balance)
    balance += value
    fmt.Printf(" 存 %d 后的余额: %d\n", value, balance)
    fmt.Println()

    wg.Done()
}

// 取钱
func withdraw(value int) {
    defer wg.Done()
    if protecting == 1 {
        mutex.Lock()
        defer mutex.Unlock()
    }
    fmt.Printf(" 余额: %d\n", balance)
    balance -= value
    fmt.Printf(" 取 %d 后的余额: %d\n", value, balance)
    fmt.Println()
}

func main() {
        wg.Add(10)
    for i:=0; i < 5; i++ {
        go withdraw(500) // 取 500
        go deposit(500)  // 存 500
    }
    wg.Wait()

    fmt.Printf(" 当前余额: %d\n", balance)
}

func init() {
    balance = 1000 // 初始账户余额为 1000
    flag.UintVar(&protecting, "protecting", 1, " 是否加锁, 0 表示不加锁, 1 表示加锁")
}
```

先声明了一个 WaitGroup 类型的全局变量 wg。main 方法中的 wg.Add(10) 表示有 10 个 goroutine 需要等待，wg.Wait() 表示等待那 10 个 goroutine 执行结束。

另外，WaitGroup 值是可以被复用的，wg 归 0 后，可以继续使用：

```
func main() {
        wg.Add(5)
    for i:=0; i < 5; i++ {
        go deposit(500)  // 存 500
    }
    wg.Wait()

    time.Sleep(time.Duration(3) * time.Second)

    wg.Add(5)
    for i:=0; i < 5; i++ {
        go withdraw(500) // 取 500
    }
    wg.Wait()

    fmt.Printf(" 当前余额: %d\n", balance)
}
```

如果你有多组任务，而这些任务需要串行执行，可以使用上面这种写法。

比如实现按顺序存钱：

```
func main() {
    for i:=0; i < 5; i++ {
        wg.Add(1)
        go deposit(500+i)  // 存 500
        wg.Wait()
    }

    fmt.Printf(" 当前余额: %d\n", balance)
}
```

执行结果：

```
余额: 1000
存 500 后的余额: 1500

余额: 1500
存 501 后的余额: 2001

余额: 2001
存 502 后的余额: 2503

余额: 2503
存 503 后的余额: 3006
```

(continues on next page)

(continued from previous page)

```
余额： 3006
存 504 后的余额： 3510

当前余额： 3510
```

CHAPTER

ELEVEN

GO 语言并发编程：上下文 CONTEXT

context.Context 类型是在 Go 1.7 版本引入到标准库的，上下文 Context 主要用来在 goroutine 之间传递截止日期、停止信号等上下文信息，并且它是并发安全的，可以控制多个 goroutine，因此它可以很方便的用于并发控制和超时控制，标准库中的一些代码包也引入了 Context 参数，比如 os/exec 包、net 包、database/sql 包，等等。下面来介绍 Context 类型的使用方法。

11.1 Context 介绍

Context 类型的应用还是比较广的，比如 http 后台服务，多个客户端或者请求会导致启动多个 goroutine 来提供服务，通过 Context，我们可以很方便的实现请求数据的共享，比如 token 值，超时时间等，可以让系统避免额外的资源消耗。

11.1.1 Context 类型

Context 类型是一个接口类型，定义了 4 个方法：

```
type Context interface {
    Deadline() (deadline time.Time, ok bool)
    Done() <-chan struct{}
    Err() error
    Value(key interface{}) interface{}
}
```

- `Deadline()`：获取设置的截止日期，到截止日期时，Context 会自动发起取消请求，ok 为 false 表示没有设置截止日期；
- `Done()`：返回一个只读通道 chan，在当前工作完成、超时或者 context 被取消后关闭；
- `Err()`：返回 Context 结束原因，取消时返回 `Canceled` 错误，超时返回 `DeadlineExceeded` 错误。
- `Value`：获取 key 对应的 value 值，可以用它来传递额外的信息和信号。

11.2 Context 衍生

Context 值是可以繁衍的，也就是可以通过一个 Context 值产生任意个子值，这些子值携带了父值的属性和数据，也可以响应通过其父值传达的信号。

Context 根节点是一个已经在 context 包中预定义好的 Context 值，是全局唯一的，它既不可以被撤销，也不能携带任何数据，可以通过调用 context.Background 函数获取到它。

context 包提供了 4 个用于繁衍 Context 值的函数：

- `WithCancel`：基于 parent context 产生一个可撤销（cancel）的子 context
- `WithDeadline`：产生可以定时撤销的子 context，达到截止日期后，context 会收到 cancel 通知。
- `WithTimeout`：与 `WithDeadline` 类似，产生可以定时撤销的子 context
- `WithValue`：产生携带额外数据的子 context

下面介绍这 4 个函数的使用示例。

11.2.1 WithCancel

WithCancel 返回两个结果值，第一个是可撤销的 Context 值，第二个则是用于触发撤销信号的函数。在撤销函数被执行后，先关闭内部的接收通道，然后向所有子 Context 发送 cancel 信号，最终断开与父 Context 之间的关联。其中 cancel 信号的传递采用的是深度优先搜索算法。

仍然是取钱的例子，要求是账户的钱大于 10000 后停止存钱：

```
package main

import (
        "context"
        "fmt"
        "math/rand"
        "sync/atomic"
        "time"
)

var (
        balance int32
)

// 存钱
func deposit(value int32, id int, deferFunc func()) {
        defer func() {
                deferFunc()
        }()
```

(continues on next page)

(continued from previous page)

```
        for {
                currBalance := atomic.LoadInt32(&balance)
                newBalance := currBalance + value
                time.Sleep(time.Millisecond * 500)

                if atomic.CompareAndSwapInt32(&balance, currBalance, newBalance) {
                        fmt.Printf("ID: %d, 存 %d 后的余额: %d\n", id, value, balance)
                        break
                } else {
                        // fmt.Printf(" 操作失败\n")
                }
        }
}

// 取钱
func withdraw(value int32) {
        for {
                currBalance := atomic.LoadInt32(&balance)
                newBalance := currBalance - value
                if atomic.CompareAndSwapInt32(&balance, currBalance, newBalance) {
                        fmt.Printf(" 取 %d 后的余额: %d\n", value, balance)
                        break
                }
        }
}

func WithCancelDemo() {
        total := 10000
        ctx, cancelFunc := context.WithCancel(context.Background())
        for i := 1; i <= 100; i++ {
                num := rand.Intn(2000) // 随机数
                go deposit(int32(num), i, func() {
                        if atomic.LoadInt32(&balance) >= int32(total) {
                                cancelFunc()
                        }
                })
        }
        <-ctx.Done()
        withdraw(10000)
        fmt.Println(" 退出")

}
```

(continues on next page)

(continued from previous page)

```
func main() {
        WithCancelDemo()
}

func init() {
        balance = 1000 // 初始账户余额为 1000
}
```

执行结果：

```
ID: 95, 存 1940 后的余额: 2940
ID: 19, 存 1237 后的余额: 4177
ID: 78, 存 1463 后的余额: 5640
ID: 17, 存 1211 后的余额: 6851
ID: 80, 存 420 后的余额: 7271
ID: 28, 存 888 后的余额: 8159
ID: 32, 存 408 后的余额: 8567
ID: 50, 存 1353 后的余额: 9920
ID: 38, 存 631 后的余额: 10551
取 10000 后的余额: 551
退出
```

11.2.2 WithDeadline

设置截止日期，达到截止日期后停止存钱：

```
func DeadlineDemo() {
        total := 10000
        deadline := time.Now().Add(2 * time.Second)
        ctx, cancelFunc := context.WithDeadline(context.Background(), deadline)
        for i := 1; i <= 100; i++ {
                num := rand.Intn(2000) // 随机数
                go deposit(int32(num), i, func() {
                        if atomic.LoadInt32(&balance) >= int32(total) {
                                cancelFunc()
                        }
                })
        }
        select {
        case <-ctx.Done():
            fmt.Println(ctx.Err())
        }
```

(continues on next page)

(continued from previous page)

```
        fmt.Println(" 超时退出")
}
```

截止日期参数 deadline 是一个时间对象：time.Time

执行结果：

```
ID: 7, 存 1410 后的余额: 1961
ID: 5, 存 81 后的余额: 2042
ID: 69, 存 783 后的余额: 2825
context deadline exceeded
超时退出
```

WithDeadline 和 WithTimeout 函数生成的 Context 值也是可撤销的，可以实现自动定时撤销，也可以在截止时间达到之前进行手动撤销（代码中的 cancelFunc() 操作）。

11.2.3 WithTimeout

和 WithDeadline 不同之处在于，时间参数为持续时间：time.Duration：

```
func WithTimeoutDemo() {
        total := 10000
        ctx, cancelFunc := context.WithTimeout(context.Background(), 2*time.Second)
        for i := 1; i <= 100; i++ {
                num := rand.Intn(2000) // 随机数
                go deposit(int32(num), i, func() {
                        if atomic.LoadInt32(&balance) >= int32(total) {
                                cancelFunc()
                        }
                })
        }
        select {
        case <-ctx.Done():
            fmt.Println(ctx.Err())
        }
        fmt.Println(" 超时退出")
}
```

执行结果：

```
ID: 36, 存 1356 后的余额: 4181
ID: 100, 存 1598 后的余额: 5779
ID: 25, 存 47 后的余额: 5826
ID: 10, 存 292 后的余额: 6118
```

(continues on next page)

(continued from previous page)

```
context deadline exceeded
超时退出
```

11.2.4 WithValue

WithValue 函数产生的 Context 可以携带数据，和另外 3 种函数不同，它是不可撤销的。Value 方法用来获取数据，没有提供改变数据的方法。

WithValue 函数产生的 Context 携带的值可以在子 Context 中传递。

```
func WithValueDemo() {

        rootNode := context.Background()
        ctx1, cancelFunc := context.WithCancel(rootNode)
        defer cancelFunc()

        ctx2 := context.WithValue(ctx1, "key2", "value2")
        ctx3 := context.WithValue(ctx2, "key3", "value3")
        fmt.Printf("ctx3: key2 %v\n", ctx3.Value("key2"))
        fmt.Printf("ctx3: key3 %v\n", ctx3.Value("key3"))

        fmt.Println()

        ctx4, _ := context.WithTimeout(ctx3, time.Hour)
        fmt.Printf("ctx4: key2 %v\n", ctx4.Value("key2"))
        fmt.Printf("ctx4: key3 %v\n", ctx4.Value("key3"))

}
```

执行结果：

```
ctx3: key2 value2
ctx3: key3 value3

ctx4: key2 value2
ctx4: key3 value3
```

11.3 小结

Context 类型是一个可以实现多 goroutine 并发控制的同步工具。Context 类型主要分为三种，即：根 Context、可撤销的 Context 和携带数据的 Context。根 Context 和衍生的 Context 构成一颗 Context 树。需要注意的是，携带数据的 Context 不能被撤销，可撤销的 Context 无法携带数据。

Context 比 sync.WaitGroup 更加灵活，在使用 WaitGroup 时，我们需要确定执行子任务的 goroutine 数量，如果不知道这个数量，使用 WaitGroup 就有风险了，采用 Context 就很容易解决了。

关注公众号【测试开发小记】及时接收最新技术文章！