背景和初衷

关于写作，自认为，自己不是一个很会写作的人，但是我有一个习惯，在做一件事儿以后，总喜欢用文字总结记录下来， 虽然自己词汇不够美，但是，对于我而言这就是一件快乐的事儿，同时，这个过程也会提升自己思考能力和码字能力。

几年前，就接触Go语言，由于工作性质原因，中间也陆陆续续写过一些脚本和工具，总感觉不是很完美的样子； 最近在系统性复习和学习，所以想着把这个过程记录成册，也是这本电子书由来，希望手册能帮到你。

联系方式

dushu100#qq.com(用@替换#)

转载声明

Go语言介绍

Go语言即Golang，起始于2007年，于2009年11月份谷歌公司正式对外发布开源。

Golang是一种静态，强类型、编译型、并发型的编程语言。

Go的语法接近C语言，但其支持垃圾回收功能，开发效率远高于C语言。而其对海量并发的支持，以及在运行效率，低内存消耗方面的优异表现，也有人称其为互联网时代的C语言。与C++相比，Go语言并不包括如异常处理、继承、泛型、断言、虚函数等功能，但增加了 Slice 型、并发、管道、垃圾回收、接口（Interface）等特性的语言级支持。

Go语言对并发编程的支持是天生的、自然的和高效的。Go语言为此专门创造出了一个关键字“go”。使用这个关键字，我们就可以很容易的使一个函数被并发的执行。

2012年，Go语言的创造者们发布了它的1.0版本。随着后边几年，逐步发布了1.1，1.2，1.3，1.4，但是特别注意的是2015年8月19日，Go语言Go 1.5版发布，本次更新中移除了”最后残余的C代码”。go1.5的发布被认为是历史性的。完全移除C语言部分，使用GO编译GO（ps：少量代码使用汇编实现），GO编译GO称之为Go的自举，是一门编程语言走向成熟的表现。

另外，他们请来了内存管理方面的权威专家Rick Hudson，对GC进行了重新设计，支持并发GC，解决了一直以来广为诟病的GC时延（STW）问题。并且在此后的版本中，又对GC做了更进一步的优化。到go1.8时，相同业务场景下的GC时延已经可以从go1.1的数秒，控制在1ms以内。GC问题的解决，可以说GO语言在服务端开发方面，几乎抹平了所有的弱点。

目前，Go语言最新版为1.20，且Golang 2.0版本提案也于2019年提出来了，相信不久的将来也很快会发布。

Go语言特点：

- 简单、易学习、开源
- 高并发、高效执行
- 内存管理
- 语法简介、数组安全、编译迅速快、跨平台

Go语言创造者

- 1、Robert Griesemer罗伯特.格利茨默（Java HotSpot虚拟机、著名的Javascript引擎V8的创造者）
- 2、Rob Pike罗伯.派克（贝尔实验室UNIX团队成员，曾参与Plan9、Inferno和Limbo等项目）
- 3、Ken Thompson肯.汤普森（贝尔实验室UNIX团队成员，C语言、UNIX和Plan 9创始人之一，与Rob Pike共同开发了UTF-8字符集规范）

Go语言解决问题

- 多核硬件架构
- 超大规模分布式计算集群
- 开发效率和扩展

Go语言主要使用领域有哪些?

Go语言和大多数服务器端编程语言一样，可以进行web相关应用的开发，其应用场景也比较多，比如服务器编程、分布式集群、中间件、网络编程、数据库、云平台等，其实通过这些年的发展来看，其更多的被应用在于游戏、区块链、云计算、人工智能、爬虫等领域.

谁在使用Golang

- 国外：google这个是自然不用说了，Meta(facebook)、Uber、Netflix、Paypal、MicroSoft、Cloudflare、Twitter等等
- 国内：阿里、腾讯、京东、阿里云、蚂蚁金服、网易、小米、百度、360、七牛、B站、字节跳动、PingCAP等等等

代表著名项目

- 容器化领域： Docker、Kubernetes(k8s)、Consul、Traefik、Etcd、Argo等
- Web框架：Gin、Beego、Echo、Revel、Goframe、Go-zero、Gorm等
- 数据库存储：tidb、influxdb等
- 微服务：go-kit、go-micro、Kitex、kratos、grpc-go等
- 区块链：以太坊协议（go-ethereum）、endermint、cosmos-sdk等
- 游戏开发：g3n、leaf、gonet、engo、nano、pitaya等等
- 其他：Grafana、Raft等

IDE选择

目前市面很多IDE，对于golang日常开发的IDE，常用主要有liteIDE、Vscode、Goland等

LiteIDE

LiteIDE是一个国人开发专门针对Golang的IDE，功能很全面，具备语法高亮、自动补全、自动编译、调试、包浏览及管理。

官网地址：http://liteide.org/

VScode

VScode是一款由微软公司开发的，能运行在 Mac OS X、Windows 和 Linux 上的跨平台开源代码编辑器。

VSCode 使用 JSON 格式的配置文件进行所有功能和特性的配置，同时它还可以通过扩展程序为编辑器实现编程语言高亮、参数提示、编译、调试、文档生成等各种功能。

其实，文本编辑器里譬如Sublime Text也是很优秀，也支持Golang日常开发，而他们都是通过插件机制来实现整个功能，属于轻量化IDE。

官网地址：https://code.visualstudio.com/

Goland

Goland 是由JetBrains公司开发的一个新的商业IDE，旨在为Go开发者提供的一个符合人体工程学的新的商业IDE。Goland 整合了 IntelliJ 平台（一个用于 java 语言开发的集成环境，也可用于其他开发语言），提供了针对Go语言的编码辅助和工具集成，是一款功能非常强大的IDE。

其实，对于其他语言，他们家的IDE做的一样出色，比如Java的 Intellj IDEA、PHP的PHPStorm、Python的PyCharm、C++的CLion、前端的WebStorm等。使用JetBrains的IDE，我们可以享受到它优秀的开箱即用的体验和jetbrains积累十几年的插件体系。

官网地址：https://www.jetbrains.com/go/

最后，汇总三款IDE最大优缺点：

- liteIDE运行速度快，代码提示特别好用，但是调试功能不太好用
- VSCode调试功能好用，但是代码提示非常一般，写起来特别费劲
- GoLand各项功能非常完善，但是是收费的，并且占用资源较多

go命令

执行go help可以查看，go支持的所有命令以及功能介绍：

```
# go help
    bug         start a bug report(开始一个错误报告)
    build       compile packages and dependencies(编译包和依赖)
    clean       remove object files and cached files(删除目标文件和缓存文件)
    doc         show documentation for package or symbol(显示包或符号的文档)
    env         print Go environment information(打印 Go 环境信息)
    fix         update packages to use new APIs(更新包以使用新的API)
    fmt         gofmt (reformat) package sources(重新格式化,包源)
    generate    generate Go files by processing source(通过处理源生成生成 Go 文件)
    get         add dependencies to current module and install them(将依赖项添加
到当前模块并安装它们)
    install     compile and install packages and dependencies(编译安装包和依赖)
    list        list packages or modules(列出包或模块)
    mod         module maintenance(模块维护)
    run         compile and run Go program(编译并运行 Go 程序)
    test        test packages(测试包)
    tool        run specified go tool(运行指定的go tool)
    version     print Go version (打印版本)
    vet         report likely mistakes in packages (报告包裹中可能存在的错误)
```

go version

查看当前golang版本

```
# go version
go version go1.17.2 darwin/arm64
```

go env

打印go的开发环境信息，比如当前操作系统框架、支持的平台、安装路径以及goproxy的设置，

```
# go env
GO111MODULE="on"
GOARCH="arm64"
GOBIN=""
GOCACHE="/Users/wanzi/Library/Caches/go-build"
GOENV="/Users/wanzi/Library/Application Support/go/env"
GOEXE=""
GOEXPERIMENT=""
GOFLAGS=""
GOHOSTARCH="arm64"
GOHOSTOS="darwin"
GOINSECURE=""
GOMODCACHE="/Users/wanzi/go/pkg/mod"
GONOPROXY=""
GONOSUMDB=""
GOOS="darwin"
GOPATH="/Users/wanzi/go"
GOPRIVATE=""
GOPROXY="https://mirrors.aliyun.com/goproxy/"
GOROOT="/opt/homebrew/Cellar/go/1.17.2/libexec"
GOSUMDB="sum.golang.org"
GOTMPDIR=""
GOTOOLDIR="/opt/homebrew/Cellar/go/1.17.2/libexec/pkg/tool/darwin_arm64"
GOVCS=""
GOVERSION="go1.17.2"
GCCGO="gccgo"
AR="ar"
CC="clang"
CXX="clang++"
CGO_ENABLED="1"
GOMOD="/dev/null"
CGO_CFLAGS="-g -O2"
CGO_CPPFLAGS=""
CGO_CXXFLAGS="-g -O2"
CGO_FFLAGS="-g -O2"
CGO_LDFLAGS="-g -O2"
PKG_CONFIG="pkg-config"
GOGCCFLAGS="-fPIC -arch arm64 -pthread -fno-caret-diagnostics -Qunused-
arguments -fmessage-length=0 -fdebug-prefix-
map=/var/folders/7z/k8kkw4rx5md6wxfrz1_c9fv80000gn/T/go-build497036738=/tmp/go-
build -gno-record-gcc-switches -fno-common"
```

如果想调整具体参数可以通过go env -w进行设置，例如：

```
go env -w GOPROXY=https://mirrors.aliyun.com/goproxy/
```

如果想恢复初始设置，可以使用go env -u来解决。

go env结果中:

GOROOT是go的安装目录；

GOPATH是go工作目录，也是go包安装路径，GPATH可以是多个；

GOMODCACHE是1.15新加环境变量，用于加速编译构建;

另外，需要注意两点：

- 一个是在实际项目中会用到私有仓库化，就需要设置特殊变量GOPRIVATE和GOINSECURE：

```
go env -w GOPRIVATE=privacy.com
go env -w GOINSECURE=privacy.com
```

- 另外一个是在交叉编译的时候适配不同硬件框架和操作系统，就需要调整GOOS、GOARCH等变量的值。

go get

获取go最新模块，并缓存到本地GOMODCACHE="/Users/wanzi/go/pkg/mod"，如下获取grpc包：

参数：

```
-v      //显示get的日志详细信息，方便排错
-u      //下载丢失的包，但不会更新已经存在的包
-d      //只下载，不安装
-insecure     //允许使用不安全http方式下载
```

```
# go get google.golang.org/grpc
go: downloading google.golang.org/grpc v1.51.0
go: downloading golang.org/x/net v0.0.0-20220722155237-a158d28d115b
go: downloading golang.org/x/sys v0.0.0-20220722155257-8c9f86f7a55f
go: downloading golang.org/x/text v0.4.0
# tree -L 2 /Users/wanzi/go/pkg/mod
/Users/wanzi/go/pkg/mod
├── cache
│   ├── download
│   └── lock
├── github.com
│   ├── davecgh
│   ├── golang
│   ├── kr
│   ├── pkg
│   ├── pmezard
│   └── stretchr
├── golang.org
│   └── x
└── google.golang.org
    ├── genproto@v0.0.0-20200526211855-cb27e3aa2013
    ├── grpc@v1.50.1
    ├── grpc@v1.51.0
    └── protobuf@v1.27.1

16 directories, 1 file
```

go run

go run 编译并运行源码文件，由于包含编译步骤，所以go build参数都可用于go run，在go run中只接受go源码文件而不接受代码包。

```
# go run firstcode.go
Hello, World!
```

go build

主要用于编译代码，并生成可执行文件，默认与源文件相同，可以通过-o指定名称。

```
# go build main.go
# ls -l
total 3784
-rwxr-xr-x  1 wanzi  staff  1932642 Nov 29 22:21 main
-rw-r--r--  1 wanzi  staff       85 Nov 29 22:20 main.go
# ./main
Golang编译成二进制
# go build -o prt main.go
# ./prt
Golang编译成二进制
```

如果是普通文件,go build不会产出任何文件，如果是main包，则会在当前文件下生成名为main的可执行文件。

另外，go build还有一些其他参数，通过go help build获取，比如-p编译是指定并行可运行cpu数，-compiler name制定编译器是gccgo还是gc，这里就不过多介绍。

go install

主要用于编译代码，并生成可执行文件，于go build不同的是，go install在编译导入的包文件，所有导入包文件编译完成后，才编译主程序； 最后将编译后生成可执行文件放到$GOPATH/bin目录下，如果指定源码文件，则以源码文件名称命名可执行文件名称，如果不指定源码文看，则以源码文件所在目录名称命名。

```
# go install xiaoli.go
# ls -l ~/go/bin
total 3776
-rwxr-xr-x  1 wanzi  staff  1932642 Nov 29 23:25 xiaoli
# go install
# ls -l ~/go/bin
total 7552
-rwxr-xr-x  1 wanzi  staff  1932642 Nov 29 23:26 base
-rwxr-xr-x  1 wanzi  staff  1932642 Nov 29 23:25 xiaoli
```

go fmt

Go自带格式化代码工具，其实主要用gofmt命令来检查并格式化代码。

我们都知道，go语言里有严格代码规范，如果不按照规范来，就编译不通过

参数：

```
-l   显示那些需要格式化的文件
-w   把改写后的内容直接写入到文件中，而不是作为结果打印到标准输出。
-r   添加形如“a[b:len(a)] -> a[b:]”的重写规则，方便我们做批量替换
-s   简化文件中的代码
-d   显示格式化前后的diff而不是写入文件，默认是false
-e   打印所有的语法错误到标准输出。如果不使用此标记，则只会打印不同行的前10个错误。
-cpuprofile   支持调试模式，写入相应的cpufile到指定的文件
```

go mod

go modules是Go1.1新增加特性，目前最新版本已经1.20版本。

常用命令：

```
go mod init         //go modules方式初始化当前目录项目
go mod download     //下载依赖包并缓存到GOPATH/pkg/mod目录中，并且是只读文件
go mod vendor       //将依赖复制到项目下vendor目录中，目录存放go.mod文件描述的依赖包，文件夹下同时还有一个文件modules.txt
go mod verify       //验证依赖项是否达到预期的目的，如果所有的模块都没有被修改过，那么执行这条命令之后，会打印all modules verified
go mod tidy         //添加缺失的模块以及移除无用的模块 执行后会生成go.sum文件
go mod tidy -v      //可以将执行的信息，即删除和添加的包打印到命令行
go mod graph        //打印模块依赖图
go mod why          //解释为什么需要依赖
```

go.mod文件内容解释：

go.mod提供了module, require、replace和exclude四个命令

module语句指定包的名字（路径） require语句指定的依赖项模块 replace语句可以替换依赖项模块 exclude语句可以忽略依赖项模块

go clean

用户删除执行其他命令产生的文件或者目录，这些文件主要包括：

```
_obj/ 旧的object目录，由Makefiles遗留
_test/ 旧的test目录，由Makefiles遗留
_testmain.go 旧的gotest文件，由Makefiles遗留
test.out 旧的test记录，由Makefiles遗留
build.out 旧的test记录，由Makefiles遗留
*.[568ao] object文件，由Makefiles遗留
DIR(.exe) 由go build产生
DIR.test(.exe) 由go test -c产生
MAINFILE(.exe) 由go build MAINFILE.go产生
*.so 由 SWIG 产生
```

go clean参数：

```
-i   清除关联的安装的包和可运行文件，也就是通过go install安装的文件
-n   把需要执行的清除命令打印出来，但是不执行，这样就可以很容易的知道底层是如何运行的
-r   循环的清除在import中引入的包
-x   打印出来执行的详细命令，其实就是-n打印的执行版本
```

go list

列出当前项目安装的包

```
# go list
go: downloading github.com/caddyserver/caddy v1.0.5
go: downloading github.com/miekg/dns v1.1.29
go: downloading github.com/Azure/azure-sdk-for-go v40.6.0+incompatible
go: downloading github.com/Azure/go-autorest/autorest v0.10.0
go: downloading github.com/Azure/go-autorest/autorest/azure/auth v0.4.2
go: downloading github.com/infobloxopen/go-trees v0.0.0-20190313150506-
2af4e13f9062
go: downloading github.com/prometheus/client_golang v1.5.1
```

Golang基础语法

golang的基础语法栏目主要涵盖golang的变量、常量、关键字、命名规范、基本数据类型、内置函数、指针等知识点。

什么是变量?

在Golang中，变量可以理解为用于存储值的一种容器，这些值可以是数字、字符串、布尔值等。在声明变量时，需要指定变量的类型。

```
var i int      // 声明一个整型变量 i
i = 1          // 将值 1 赋给变量 i
```

此外，Golang的变量还具有作用域和生命周期。在同一个作用域中，不能使用相同名称的变量。变量的生命周期由变量在内存中的存储时间来决定。变量的生命周期可以是全局、局部或动态分配的。

例如，以下是一个在函数内定义并初始化的变量：

```
func main() {
    var s string = "hello"
    fmt.Println(s)
}
```

在这个例子中，变量 s 的作用域为 main 函数。在 main 函数外无法访问变量 s。变量 s 的生命周期取决于变量 s 的创建和销毁时间。

变量声明

1.var关键字定义变量

格式如下：

```
var name type = value
```

其中，name是变量名，type是变量的类型，value是变量的初始值。

例如：

```
var age int = 30
var name string = "Alice"
```

在定义变量时，可以省略type，由Go语言自动推断类型。例如

```
var age = 30
var name = "Alice"
```

也可以定义多个变量：

```
a, b, c := 1, 2, 3
```

2.类型推断定义变量

使用关键字var或者:=定义变量时，如果没有指定变量的类型，则可以通过变量的值推断出变量的类型。例如：

```
var a = 10 // 推断a为整型
b := "hello world" // 推断b为字符串
```

3.使用:=运算符定义变量

格式如下：

```
name := value
```

其中，name是变量名，value是变量的初始值。注意，:=只能在函数内部使用，且用于定义新变量。例如：

```
func main() {
  age := 30
  name := "Alice"
}
```

4.批量声明变量

在使用 var 批量声明变量时，可以使用逗号分隔的形式，如下所示：

```
var (
    x int
    y string
    z bool
)
```

需要注意的是，批量声明必须在函数体内进行，不能在全局作用域中使用。

批量声明变量时，也可以指定类型和初始值。例如：

```
var (
    name string = "Alice"
    age  int    = 30
)
```

或者使用简短变量声明方式：

```
name, age := "Alice", 30
```

在简短变量声明中，Golang 会根据值自动推断变量类型。例如：

```
x, y := 1, 2.0
fmt.Printf("%T %T", x, y) // Output: int float64
```

此时，x 会被自动推断为 int 类型，y 则被推断为 float64 类型。

Golang常量

在 Golang 中，使用 const 关键字来声明常量。与变量不同，常量的值是不可变的，且必须在声明时进行初始化，即必须给定一个值。

常量可以是任何基本数据类型，如整型、浮点型、布尔型、字符串型等，也可以是指向类型的指针。

使用const关键字来定义常量，语法如下：

```
const constantName [type] = constantValue
```

其中 [type] 可以省略，此时常量类型根据赋值的类型自动推导。

例如:

```
const Pi = 3.1415926
const (
    StatusOK       = 200
    StatusNotFound = 404
)
```

上述代码中，第一个常量 Pi 用于存储圆周率的值，第二个常量使用了批量声明，定义了两个 HTTP 状态码。

值得注意的是，在批量声明常量时，第一个常量的值默认被赋值为 0，后续常量的值会根据前面的常量值自动递增。

常量的命名规则与变量相同，不过常量一般使用大写字母来进行命名，以区分于变量。

常量在程序运行时不可修改，因此常量在程序中常常被用来表示不可改变的参数或者设定值，如圆周率或 HTTP 状态码等。

特殊常量iota

iota是Go语言的常量计数器，只能在常量的表达式中使用。在 const 关键字出现时将被重置为 0，之后每出现一次 iota，其所代表的数字会自动增1。 iota 常量生成器可用于在定义枚举时生成一系列相关值。

例如，下面使用iota定义枚举：

```
const (
    Monday = iota // 0
    Tuesday       // 1
    Wednesday     // 2
    Thursday      // 3
    Friday        // 4
    Saturday      // 5
    Sunday        // 6
)
```

在这个例子中，iota赋值给 Monday，然后在随后的常量定义中，iota 会自动递增。Tuesday 的值为 1，Wednesday 的值为 2，以此类推。

Golang关键字

Golang中的关键字是指被编程语言保留使用的单词，这些单词在代码中具有特殊的意义，不能作为标识符使用。

```
break：用于在循环中跳出循环或在 switch 语句中跳出 switch 语句。
case：用于在 switch 语句中分支选择。
chan：用于声明通道类型。
const：用于声明常量。
continue：用于跳过循环中剩余的语句并开始下一次循环。
default：在 switch 语句中所有 case 都不匹配时执行的语句块。
defer：用于函数结束前执行一个语句块，常用于资源释放。
else：在 if 语句中，如果条件不成立时执行的语句块。
fallthrough：在 switch 语句中，将控制权转移到下一个 case 语句。
for：用于循环语句。
func：用于定义函数和方法。
go：用于启动一个新的 goroutine。
goto：用于无条件跳转到代码中的某个标签。
if：用于条件语句。
import：用于导入其他包。
interface：用于声明接口类型。
map：用于声明映射类型。
package：用于定义包，每个 Go 文件必须在 package 声明的包中。
range：用于循环迭代数组、切片、字符串、映射和通道。
return：用于从函数返回一个值。
select：用于同时等待多个通道操作。
struct：用于声明结构体类型。
switch：用于根据不同的条件执行不同的分支语句。
type：用于声明自定义类型。
var：用于声明变量。
```

命名规范

golang命名规范主要包含变量、函数、类型、常量等命名规范。

变量命名规范

- 变量名必须由字母、数字、下划线组成，不能以数字开头。
- 变量名应该具有一定的描述性，可以让人理解其含义，尽量不要使用单个字母作为变量名。
- 变量名采用驼峰式命名，即第一个单词小写，后续单词首字母大写。

例如：

```
var userName string
var orderNum int
```

函数命名规范

- 函数名采用驼峰式命名，第一个单词小写，后续单词首字母大写。
- 函数名应该具有一定的描述性，可以让人理解其作用。
- 如果函数是私有函数，函数名应该以小写字母开头。

例如：

```
func getUserInfo(userID int) (*User, error) {
    // ...
}

func (u *User) updateUser() error {
    // ...
}
```

类型命名规范

- 类型名采用驼峰式命名，第一个单词大写，后续单词首字母大写。
- 类型名应该具有一定的描述性，可以让人理解其作用。
- 如果类型是私有类型，类型名应该以大写字母开头。

```
type User struct {
    // ...
}

type UserType int
```

常量命名规范

- 常量名全部大写，单词之间用下划线分隔。
- 常量名应该具有一定的描述性，可以让人理解其含义。

例如：

```
const MaxSize = 100
const HTTP_STATUS_OK = 200
```

其他命名规范

- 公有变量、常量或函数名应该使用 Pascal Case 命名法，并且要有注释说明。
- 私有变量、常量或函数名应该使用 Camel Case 命名法，并且第一个字母应该小写。
- 接口类型的名称以单个方法的名称加上 er 后缀命名，如 Reader、Writer 等。
- 结构体类型的名称应该是一个或多个单词的组合，每个单词的首字母大写，如：CustomerOrder、UserInfo。
- 结构体中的字段名称采用 Camel Case 命名法，首字母小写。
- 方法名称应该是一个或多个单词的组合，每个单词的首字母大写，如：GetUserInfo、SaveOrder。
- 包名应该是短小的小写单词组成的，如：math、encoding、io。
- 常量名称应该是全大写字母，单词之间用下划线分隔，如：MAX_COUNT、TOTAL_NUM

基本数据类型

- bool：布尔类型，表示真或假。默认长度为1。
- string：字符串类型，表示一组字符序列。默认长度为0。
- int、int8、int16、int32、int64：整型，表示带符号的整数类型。默认长度分别为32位和64位。
- uint、uint8、uint16、uint32、uint64：无符号整型，表示不带符号的整数类型。默认长度分别为32位和64位。
- uintptr：无符号整型，用于存储指针的地址。默认长度与机器字长相同。
- float32、float64：浮点型，表示单精度和双精度浮点数类型。默认长度分别为32位和64位。
- complex64、complex128：复数类型，表示具有实部和虚部的复数。默认长度分别为64位和128位。
- byte：相当于uint8类型，用于表示字符类型中的单个字节。默认长度为1。
- rune：相当于int32类型，用于表示Unicode字符。默认长度为4。

引用类型或非引用类型?

在Go语言中，有两种类型的数据类型：引用类型和非引用类型。

- 基本数据类型（如 int、float、bool、string 等）是非引用类型。这些类型的变量在内存中分配的是实际值的空间。当传递这些变量时，函数会复制实际值而不是变量本身。在处理基本数据类型时，我们使用值传递。

- 引用类型（如 slice、map、channel、interface 和函数类型）则是指向底层数据结构的指针的包装器。这些类型的变量在内存中分配的是指向底层数据结构的指针，而不是实际值的空间。在处理引用类型时，我们使用指针传递。

- 指针类型是一种特殊的引用类型，用于指向变量的内存地址。指针变量保存的是变量的地址，而不是实际值。使用指针类型可以通过指针间接访问变量，也可以在函数中传递指针以通过指针访问原始变量。

字符串

在golang中，字符串是一种基本数据类型，其值是Unicode码点（code point）序列，通常被解释为 UTF-8 编码的字节序列。在 Go 语言中，字符串的底层数据结构是一个只读的字节数组，也就是一组连续的字节。

字符串类型数据，通常包含两个字段，一个是指向底层字节数组的指针，一个是表示字符串长度的整数。因为字符串是只读的，所以可以在不同的字符串变量之间共享底层字节数组，这种机制可以减少内存分配的次数和内存占用的大小，提高了Go语言的内存利用率。

字符串底层实现

字符串的底层实现是一个只读的字节数组，也就是说字符串的值不能被改变，但是可以通过 []byte 类型的转换来修改字符串的值。

字符串是一个值类型，而不是一个引用类型。

因此，字符串是可以进行比较运算，两个相等的字符串的字节数组的内容也是相等的。

字符串转换

字符串可以通过转换函数将其他数据类型转换为字符串，例如：

- strconv.Itoa() 可以将整数转换为字符串。
- strconv.FormatFloat() 可以将浮点数转换为字符串。
- fmt.Sprintf() 可以使用格式化字符串将其他数据类型转换为字符串。

反过来，字符串也可以通过解析函数将字符串转换为其他数据类型，例如：

- strconv.Atoi() 可以将字符串转换为整数。
- strconv.ParseFloat() 可以将字符串转换为浮点数。
- strconv.ParseBool() 可以将字符串转换为布尔值。

字符串、byte、rune关系和转换

在 Go 语言中，字符串底层是一个只读的字节数组，也就是 []byte 类型，但是因为字符串是只读的，所以需要使用 rune 类型来表示 Unicode 字符，而不是 byte 类型。因此，我们可以简单地把 string 类型看成是一个包含了若干个 rune 类型的数组，其中每个 rune 表示一个 Unicode 字符。

[]byte 转 string：使用 string() 方法将字节数组转为字符串类型。

```
bytes := []byte{65, 66, 67, 68}
str := string(bytes)  // "ABCD"
```

string 转 []byte：使用 []byte() 方法将字符串类型转为字节数组。

```
str := "ABCD"
bytes := []byte(str)  // [65 66 67 68]
```

string 转 []rune：使用 []rune() 方法将字符串类型转为 rune 数组。

```
str := "ABCD"
runes := []rune(str)  // [65 66 67 68]
```

[]rune 转 string：使用 string() 方法将 rune 数组转为字符串类型。

```
runes := []rune{65, 66, 67, 68}
str := string(runes)  // "ABCD"
```

需要注意的是，虽然 Go 语言的字符串是 Unicode 编码的，但是它并不是固定长度的。因为一个 Unicode 字符可能由多个字节组成，所以在 Go 语言中，使用 UTF-8 编码来表示 Unicode 字符，一个字符可能由 1 到 4 个字节组成。因此，在处理字符串时，需要注意字符的长度和字节的长度并不是一一对应的关系。

Golang内置函数

内置函数就是可以直接使用而不需要进行导入的函数。

下面常用的Golang内置函数列表：

```
close：用于关闭通道
len：用于求字符串、数组、切片、map和通道等的长度
cap：用于求切片、数组、和通道等的容量
new：用于动态分配类型的内存空间，并返回该类型的指针
make：用于创建切片、map和通道等集合类型
append：用于向切片中添加元素
copy：用于复制切片
delete：用于从map中删除元素
panic：用于引发异常
recover：用于捕获并处理异常
```

内置函数例子：

len：返回字符串、数组、切片、字典和通道的长度，例如：

```
s := "hello"
l := len(s) // l == 5

a := []int{1, 2, 3, 4, 5}
l = len(a) // l == 5
```

cap：返回切片或数组的容量，例如：

```
a := make([]int, 5, 10)
c := cap(a) // c == 10
```

append：用于在切片尾部追加元素，例如：

```
a := []int{1, 2, 3}
a = append(a, 4, 5)
// a == [1 2 3 4 5]
```

make：用于创建切片、字典和通道，例如：

```
a := make([]int, 5) // 创建一个长度为 5 的切片

m := make(map[string]int) // 创建一个空字典

c := make(chan int) // 创建一个通道
```

copy：将一个切片的元素复制到另一个切片中，例如：

```
a := []int{1, 2, 3, 4, 5}
b := make([]int, len(a))
copy(b, a)
// b == [1 2 3 4 5]
```

panic 和 recover：用于处理异常，例如：

```
func test() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered from", r)
        }
    }()
    panic("test")
}
test() // 输出: Recovered from test
```

close：用于关闭通道，例如：

```
c := make(chan int)
close(c)
```

delete：用于删除字典中的元素，例如：

```
m := map[string]int{
    "a": 1,
    "b": 2,
    "c": 3,
}
delete(m, "b")
// m == map[a:1 c:3]
```

print 和 println：用于输出到标准输出，例如：

```
s := "hello"
print(s) // 输出: hello

a := []int{1, 2, 3}
println(a) // 输出: [1 2 3]
```

new：用于创建指针类型的变量，例如：

```
var p *int = new(int)
```

Golang编码

在计算机中，编码是将一种形式的数据转换为另一种形式的过程。

在Golang中，编码是指将一组字符或字符串转换为一组字节或二进制数据的过程，通常用于将数据在网络或存储介质中进行传输或存储。

Golang 中支持的编码方式有多种，包括 ASCII、UTF-8、UTF-16、UTF-32 等。下面简要介绍几种常见的编码方式：

- ASCII 编码：ASCII 编码使用一个字节来表示一个字符，即采用 8 位二进制数表示一个字符，共可以表示 2^8 = 256 种不同的字符。
- UTF-8 编码：UTF-8 是一种可变长度的编码方式，使用 1~4 个字节来表示一个字符。对于 ASCII 字符，使用一个字节表示；对于中文、韩文等字符，使用 3 个字节表示；对于某些特殊字符，如表情符号等，需要使用 4 个字节表示。因此，一个字符串的长度并不等于它所占用的字节数，它由其中的字符数量决定。
- UTF-16 编码：UTF-16 使用 2 个字节来表示一个字符，对于 ASCII 字符，使用一个字节表示；对于大多数中文、日文、韩文等字符，使用 2 个字节表示；对于某些特殊字符，需要使用 4 个字节表示。
- UTF-32 编码：UTF-32 使用 4 个字节来表示一个字符，无需考虑可变长度的问题，但对于大部分字符而言，会造成空间浪费。

在Golang中，字符串默认采用 UTF-8 编码，每个字符占用 1~4 个字节。

```
对于 ASCII 字符，使用一个字节表示；
对于中文、韩文等字符，使用 3 个字节表示；
对于某些特殊字符，如表情符号等，需要使用 4 个字节表示。
```

Golang 中还提供了很多用于处理不同编码方式的库和函数，如 encoding/json、encoding/base64、unicode/utf8 等。这些库和函数可以帮助开发者在不同编码方式之间进行转换和处理。

指针

在Go语言中，指针是一个变量，它存储了另一个变量的内存地址。因此，指针变量指向的是另一个变量的内存地址，而不是变量本身。

声明指针变量时，需要在变量名前加上 *，表示这是一个指针变量，例如：

```
var p *int
```

这表示声明了一个名为 p 的指向整型变量的指针。使用 & 运算符可以取得一个变量的地址，例如：

```
x := 10
p := &x
```

这表示取得变量 x 的地址，并将地址赋给指针变量 p。

使用指针访问变量时，需要使用 * 运算符，它表示解引用操作符，例如：

```
x := 10
p := &x
fmt.Println(*p)
```

这表示打印出指针变量 p 指向的变量的值，即变量 x 的值。

除了声明指针变量和获取变量的地址之外，还可以使用 new 函数来创建指针变量。例如，创建一个指向整型变量的指针：

```
p := new(int)
```

这将创建一个新的整型变量，并返回它的地址，然后将地址赋给指针变量 p。

指针还可以用于函数参数和返回值，以便在函数调用之间共享数据。

在使用指针时需要小心，因为如果指针指向一个无效的内存地址，程序可能会崩溃或产生不可预测的行为。因此，使用指针时需要确保指针指向的内存地址是有效的。

指针使用场景

- 通过指针函数传参

指针可以用于函数参数的传递，当一个函数需要修改实参的值时，可以将实参的地址作为形参传递给函数，通过操作指针来达到修改实参的值的目的。例如：

```go
func modify(s *string) {
    *s = "hello, world"
}

func main() {
    s := "hello"
    modify(&s)
    fmt.Println(s) // 输出: hello, world
}
```

- 通过指针访问结构体字段

通过指针可以更方便地访问结构体中的字段，特别是当结构体很大时，传递指针比较传递整个结构体更高效。例如：

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

func main() {
    p := &Person{
        Name: "Alice",
        Age:  18,
    }
    p.Name = "Bob"
    fmt.Println(p) // 输出: &{Bob 18}
}
```

- 通过指针动态分配内存

通过指针可以在运行时动态地分配内存，比如使用 new() 函数创建一个新的变量并返回它的地址。例如：

```go
func main() {
    p := new(int)
    *p = 42
    fmt.Println(*p) // 输出: 42
}
```

- 传递变长参数

在Golang中，可以使用指针传递变长参数。这是因为在使用变长参数时，传递的是一个切片（slice），而切片本身就是指向一个数组的指针。这样的操作，可以避免拷贝大量的数据。

例如，考虑以下函数，它接受一个变长参数并将其打印出来：

```
func printArgs(args ...int) {
    for _, arg := range args {
        fmt.Println(arg)
    }
}
```

现在，如果要将切片作为参数传递给另一个函数，可以使用指针：

```
func printArgsPtr(args *[]int) {
    for _, arg := range *args {
        fmt.Println(arg)
    }
}

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

在这个示例中，我们定义了一个新函数printArgsPtr，它使用指针作为参数来接收切片。函数体内，我们使用*args来解引用指针，从而得到实际的切片，并遍历它以打印每个元素。在主函数中，我们创建了一个切片，并将其地址传递给printArgsPtr函数。

需要注意的是，在使用指针传递变长参数时，如果切片为空，则不能传递nil指针，而应该传递一个空的切片。这是因为在Golang中，使用空的切片和nil指针的含义不同,后面我会介绍。

除了上述几个方面，指针在一些特定的场景下也非常有用，比如在处理数据结构时，通过指针可以更高效地操作链表、树等数据结构。但需要注意，过度使用指针可能会导致代码难以维护，需要谨慎使用。

常见空数据类型

在 Go 语言中，有一些空数据类型，包括：

- 空指针（nil pointer）：指向一个不存在地址的指针，一般用 nil 表示。
- 空切片（nil slice）：长度为 0 的切片，一般用 nil 表示。
- 空字典（nil map）：长度为 0 的字典，一般用 nil 表示。
- 空接口（nil interface）：类型和值均为 nil 的接口，一般用 nil 表示。
- 空通道（nil channel）：没有分配存储空间的通道，一般用 nil 表示。

nil表示一个空指针，即指针变量未指向任何地址，可以表示指针、接口、map、通道和函数类型的零值，而不能用来表示其他类型的零值，如int、float、bool等

这些空数据类型的定义均为对应类型的零值，其中指针、切片、字典、接口和通道的零值都是 nil，表示这些类型的默认值为 nil，即没有指向任何实际数据的指针、切片、字典、接口和通道。因此在实际使用这些数据类型时，可以将其初始化为零值，也可以显式地赋值为 nil。

例如:

```go
var s []int = nil // 定义一个空切片
var p *int = nil  // 定义一个空指针
var m map[string]int = nil //定义一个空字典
var i interface{} = nil //定义一个空接口
var c chan int = nil //定一个空通道
```

需要注意，如下几点:

- nil不是关键字，而是预定义的标识符。在 Go 中，可以将 nil 赋给任何类型的指针、切片、字典、函数和通道等变量。
- 空切片和空字典在使用前需要初始化，否则会出现 panic 异常。
- 空通道需要使用 make 函数初始化后才能使用。
- 空指针可以直接使用，因为它的值为 nil。
- 空接口不需要初始化即可直接使用。因为空接口本质上就是一个类型为 interface{} 的变量，它可以接受任何类型的值，包括 nil。所以在使用空接口时，如果没有显式地赋值，它就是一个空接口，可以直接使用。

避免空指针异常

在使用指针类型的变量时，需要进行 nil 检查以避免出现空指针异常。

例如:

```go
var p *int
fmt.Println(p == nil) // 输出 true
```

代码中，p 是一个整型指针类型的变量，因为没有被初始化，它的值为 nil。可以通过比较 p 是否为 nil 来检查它是否为空指针。

```go
var s []int
fmt.Println(s == nil) // 输出 true
```

代码中，s 是一个整型切片类型的变量，因为没有被初始化，它的值为 nil。可以通过比较 s 是否为 nil 来检查它是否为空切片。

```go
var m map[string]int
fmt.Println(m == nil) // 输出 true
```

代码中，m 是一个字典类型的变量，因为没有被初始化，它的值为 nil。可以通过比较 m 是否为 nil 来检查它是否为空字典。

```go
var f func(int) int
fmt.Println(f == nil) // 输出 true
```

代码中，f 是一个函数类型的变量，因为没有被初始化，它的值为 nil。可以通过比较 f 是否为 nil 来检查它是否为空函数。

```
var ch chan int
fmt.Println(ch == nil) // 输出 true
```

代码中，ch 是一个整型通道类型的变量，因为没有被初始化，它的值为 nil。可以通过比较 ch 是否为 nil 来检查它是否为空通道。

最后，特别注意，空数组、空切片和空 map是可以使用 len() 函数求长度，结果都为 0。但是，空指针、空接口、空通道是不能使用 len() 函数求长度，否则会引发运行时错误。

这是因为，在 Go 中，len()函数返回一个值的长度，这个长度是预先定义好的，通常是由编译器计算得出的。空切片、空数组和空字典等类型变量的长度为0，因为它们在内存中已经有了一块空间，所以可以使用len()函数获取它们的长度。

而对于空接口和空通道，它们是不存储数据的，所以它们的长度是没有意义的，因此在使用len()函数时会引发运行时错误。如果要判断它们是否为空，应该使用nil值进行判断。

复合数据类型

golang中复合数据类型，它们可以用来组织和处理更复杂的数据。以下是一些主要的复合数据类型：

- 数组（Array）：数组是具有固定长度并包含特定类型元素的数据结构。数组的长度是其类型的一部分，这意味着数组的长度在定义时就已经固定，不能在运行时更改。例如：var a [5]int
- 切片（Slice）：切片是对数组的一个连续片段的引用。这个片段可以是整个数组，或者是由起始和终止索引标识的一部分。切片是可变长的，可以在运行时添加和删除元素。例如：var s []int = a[1:3]
- 映射（Map）：映射是一个无序的键值对集合。它的键和值可以是任何类型的数据，但所有的键必须是唯一的。映射是动态的，可以在运行时添加和删除键值对。例如：m := make(map[string]int)
- 结构体（Struct）：结构体是一种聚合的数据类型，它是由零个或多个任意类型的值聚合成的实体。每个值被称为结构体的成员。例如：

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

- 通道（Channel）:通道是一种特殊的类型，它是Go的并发编程特性中的一个关键组成部分。可以让你在不同的Goroutine之间发送类型化的数据。它是连接并发Goroutine的管道。你可以在一个Goroutine中向通道发送值，然后在另一个Goroutine中接收这个值。
- 接口（Interface）：接口是一种类型，它定义了一组方法（方法集），但这些方法不实现任何功能。接口提供了一种方式来指定对象应该具有哪些方法，而不需要指定这些方法如何实现。例如：

```
type Writer interface {
    Write([]byte) (int, error)
}
```

数组介绍

数组是一种特殊的数据类型，它可以用于存储固定大小的相同类型的元素。数组的大小在定义时就已经确定，并且不能在运行时更改。这是它与切片（slices）最大的区别，切片的大小可以在运行时动态更改。

数组有如下特点：

- 数组长度不可变，且它不是引用类型
- 数组里数据是相同类型数据
- 数组中元素可以是任意的原始类型，比如int、string等
- 一个数组中元素的个数被称为数组长度
- 数组的长度属于类型一部分，也就是[5]int和[10]int属于不同类型
- 数组占用内存连续性，也就是数组中的元素被分配到连续内存地址中，因而索引数组元素速度非常快。

数组底层原理

在Go语言中，数组是一种值类型。当你创建一个数组时，Go会在内存中为该数组分配一个连续的内存块，每个元素都占据该内存块中的一部分。数组中的每个元素都可以通过其索引直接访问。因为这种内存布局，数组的访问速度非常快。

下面是一个简单的图解，展示了一个长度为3，元素类型为int的数组在内存中的布局

```
内存地址: |  0x00  |  0x04  |  0x08  |
索引:     |   0    |   1    |   2    |
值:       | arr[0] | arr[1] | arr[2] |
```

这里，我们假设一个int类型的值占用4个字节（这取决于你的系统架构，可能会有所不同）。如你所见，每个数组元素都存储在连续的内存地址中。

当你将一个数组赋值给另一个数组时，Go会创建一个新的内存块，并将原数组的所有元素值复制到新的内存块。这就是为什么说Go中的数组是值类型，而不是引用类型。这也意味着如果你在函数中修改了一个数组，原数组不会被修改，除非你使用指针或者切片。

数组定义

下面是定义数组格式：

```
var array_name [n]T
```

在这里，n表示数组的长度；T 表示数组中元素的类型。

比如，我们定义一个长度为5的整数数组，你可以这样做：

```
var numbers [5]int
```

或者定义数组同时进行初始化：

```
var numbers [5]int = [5]int{1, 2, 3, 4, 5}
```

更简洁方法可以这样：

```
numbers := [5]int{1, 2, 3, 4, 5}
```

如果不确定数组长度，可以让Go编译器自动计算数组的长度，你可以使用 ...：

```
numbers := [...]int{1, 2, 3, 4, 5}
```

数组遍历

有两种主要的方法可以遍历 Go 中的数组。你可以使用传统的 for 循环，也可以使用 range 关键字。

通过for遍历数组：

```
for i := 0; i < len(numbers); i++ {
    fmt.Println(numbers[i])
}
```

通过range关键字遍历数组:

```
for index, value := range numbers {
    fmt.Printf("Index: %d, Value: %d\n", index, value)
}
```

如果不需要使用索引，可以这样：

```
for _, value := range numbers {
    fmt.Println(value)
}
```

切片基础

切片(Slice)是一种拥有相同类型元素得可变长序列，本质是一个数组的引用，但是与数组不同的是切片是动态的，长度可以在运行时改变。

切片的底层逻辑是通过一个指向数组的指针，以及长度和容量来实现的。切片的长度是切片中元素的数量，容量是从创建切片的数组元素开始数，到数组末尾元素的数量。

因此，切片是一种复合数据类型(引用类型)。

切片的数据结构可以理解为一个拥有三个元素的结构体：

- 指针(pointer)：指向数组的指针。
- 长度(len)：切片中元素的数量。
- 容量(cap)：从创建切片的数组元素开始数，到数组末尾元素的数量。

例如：

```
package main

import "fmt"

func main() {
    s1 := [8]int{1, 2, 3, 4, 5, 6, 7}
    s2 := s1[3:6]
    fmt.Printf("值=%d，长度=%d，容量=%d\n", s2, len(s2), cap(s2))
}
```

上面例子中，定义了一个切片s1，然后从s1中截取后重新定义切片s2,然后通过内置函数len和cap获取切片长度和容量。

需要注意如下：

- 切片之间是没法比较，不能直接使用 == 或 != 来比较两个切片。
- 切片唯一合法得比较操作是和nil比较，一个nil值的切片并没有底层数组，一个nil值的切片长度和容量都是0
- 我们不能说一个长度和容量都为0的切片，一定是nil

切片定义

1、使用make函数定义

make函数的第一个参数是切片的类型，第二个参数是切片的长度，第三个参数是切片的容量。如果省略第三个参数，那么切片的容量将和长度相同。

例如：

```
s1 := make([]int, 5) // 创建一个长度和容量都是5的切片
s2 := make([]int, 5, 10) // 创建一个长度为5，容量为10的切片
```

2、直接定义切片

可以直接使用[]定义切片，这种方式定义的切片的长度和容量都是元素的数量。 例如:

```
s := []int{1, 2, 3, 4, 5} // 创建一个包含1, 2, 3, 4, 5的切片
```

3、使用数组或者已有切片定义新切片

可以使用[:]从数组或已有切片创建新的切片。

例如:

```
arr := [5]int{1, 2, 3, 4, 5} // 创建一个数组
s := arr[1:4] // 创建一个新的切片，包含数组的第2个到第4个元素
```

切片循环

在Go语言中，切片（Slice）的遍历主要有两种方法：

1、使用for循环：可以通过索引遍历切片中的每一个元素。例如：

```
s := []int{1, 2, 3, 4, 5}
for i := 0; i < len(s); i++ {
    fmt.Println(s[i])
}
```

在这个例子中，我们使用len(s)获取切片的长度，然后使用for循环从0遍历到len(s)-1，并使用i作为索引来访问切片中的每一个元素。

2、使用range关键字：range关键字可以遍历切片，返回索引和元素。例如：

```
s := []int{1, 2, 3, 4, 5}
for i, v := range s {
    fmt.Println(i, v)
}
```

在这个例子中，range s会返回两个值，第一个值是元素的索引，第二个值是元素的值。我们可以使用i和v来获取这两个值。

如果你只需要元素的值，而不需要元素的索引，那么你可以使用range关键字，并忽略第一个返回值：

```
s := []int{1, 2, 3, 4, 5}
for _, v := range s {
    fmt.Println(v)
}
```

切片操作

切片的操作包括访问元素、修改元素、添加元素和删除元素。

访问元素：可以通过索引访问切片中的元素，例如s[i]。

修改元素：可以通过索引修改切片中的元素，例如s[i] = x。

添加元素：可以使用append函数在切片的末尾添加新的元素，例如s = append(s, x)。 如果切片的容量不足以容纳新的元素，append函数会创建一个新的数组，并将原始数组的内容复制到新数组中。

删除元素：切片没有内置的删除元素的函数，但可以通过append和copy函数结合使用来删除元素。例如，要删除索引为i的元素，可以使用s = append(s[:i], s[i+1:]...)。

举例：

```
s := []int{1, 2, 3, 4, 5} // 创建一个切片
i := 2 // 要删除元素的索引

// 删除索引为i的元素
s = append(s[:i], s[i+1:]...)

fmt.Println(s) // 输出: [1 2 4 5]
```

在这个例子中，我们首先创建了一个切片s。然后，我们使用append函数和切片操作来删除索引为i的元素。s[:i]获取切片中从第一个元素到第i-1个元素的子切片，s[i+1:]获取切片中从第i+1个元素到最后一个元素的子切片。append(s[:i], s[i+1:]...)将这两个子切片连接起来，形成一个新的切片，这个新的切片就是删除了索引为i的元素的切片。

需要注意的是，这种方法会修改原始切片，如果你不想修改原始切片，你可以先复制原始切片，然后再删除元素：

```
s := []int{1, 2, 3, 4, 5} // 创建一个切片
t := make([]int, len(s)) // 创建一个和s长度相同的切片
copy(t, s) // 复制s到t

i := 2 // 要删除元素的索引

// 删除索引为i的元素
t = append(t[:i], t[i+1:]...)

fmt.Println(s) // 输出: [1 2 3 4 5]
fmt.Println(t) // 输出: [1 2 4 5]
```

在这个例子中，我们首先创建了一个切片s，然后创建了一个和s长度相同的切片t，并将s的元素复制到t。然后，我们在t上删除了索引为i的元素。这样，s就不会被修改。

结构体介绍

结构体一种复合的数据类型，它允许将不同的数据段组合在一起，形成一个独立的实体，

- 类似于其他语言中的类，但是Go语言中的结构体更加灵活、简洁；
- 结构体中不支持继承，而是通过组合嵌套和接口实现多态的方式来完成代码复用和模块化设计。

结构体定义

go语言中是通过type关键字定义，后面跟着结构体名称和一对大括号，大括号内是结构体的字段列表，每个字段由字段名和字段类型组成，字段之间用分号分隔。

如下，一个基本的结构体定义：

```
package main

import "fmt"

// 定义一个名为Student的结构体类型
type Student struct {
    Name string // 姓名
    Age  int    // 年龄
}

func main() {
    // 创建Student结构体的一个实例
    stu := Student{Name: "Larry", Age: 35}
    fmt.Println(stu)

    stu.Age = 31 // 修改年龄字段
    fmt.Println(stu.Name) // 访问姓名字段
}
```

结构体实例化方法：

```
// 定义一个名为Student的结构体类型
type Student struct {
    Name string
    Age  int
}
```

1、直接赋值实例化:

> 指定结构体类型和对应的字段值来创建一个新的结构体实例

举例：

```
func main() {
    // 创建Student结构体的一个实例
    stu := Student{"xiaoming", 30}
}
```

2、字面量方式实例化:

举例:

```
func main() {
    // 通过结构体字面量创建实例
    stu := Student{Name: "Larry", Age: 35}

    // 初始化部分字段
    stuName := Student{Name: "Larry"}

    fmt.Printf("字面量实例化 Name: %s, Age: %d \n", stu.Name, stu.Age)
    fmt.Printf("部分字段实例化: %v %s", stuName, stuName.Name)
}
```

3、new函数实例化(创建一个指向结构体的指针)

> new函数为结构体分配内存并返回一个指向该内存地址的指针。

这种方式初始化的结构体需要通过指针访问其字段。

举例：

```
func main() {
    // new函数实例化
    larry := new(Student)
    (*larry).Name = "Larryliang" // 或者larry.Name = "Larry"
    larry.Age = 35
}
```

4、顺序初始化实例化

> 当你知道结构体字段的顺序时，可以使用这种方法。这种方法不需要指定字段名。

举例

```
// 定义结构体Graph
type Graph struct {
    x int
    y int
}

func main() {
    g := Graph{5, 8}
    fmt.Println(g) //输出{5 8}
}
```

5、&操作符实例化

其实，这里类似new函数，创建的也是一个指向结构体实例的指针

```
func main() {
    stu := &Student{
        Name: "Larry",
        Age:  35,
    }
    fmt.Println(*stu, stu.Name, stu.Age) //输出结果 {Larry 35} Larry 35
}
```

6、使用构造函数

虽然Go语言没有专门的构造函数关键字，但可以通过定义一个返回结构体指针的函数来实现构造函数的功能。

```
func NewStudent(name string, age int) *Student {
    return &Student{
        Name: name,
        Age:  age,
    }
}

func main() {
    stu := NewStudent("Larry", 35)
    fmt.Printf("Name: %s, Age: %d", stu.Name, stu.Age) //输出结果 Name: Larry,
Age: 35
}
```

结构体方法的接受者

1、结构体方法接收着是值类型

> 当方法使用值接收者时，它接收的是该类型的一个副本。对副本的修改不会影响原始值。

```go
package main

import "fmt"

type Counter struct {
    count int
}

// Increment 是一个值类型接收者的方法
func (c Counter) Increment() {
    c.count++
}

func main() {
    counter := Counter{count: 0}
    fmt.Println(counter.count) // 输出: 0
    counter.Increment()
    fmt.Println(counter.count) // 仍然输出: 0，因为Increment操作的是副本
}
```

2、结构体方法接收者是指针类型

> 当方法使用指针接收者时，它接收的是该类型的一个指针，这意味着方法可以修改接收者的值。

```go
package main

import "fmt"

type Counter struct {
    count int
}

// Increment 是一个指针类型接收者的方法
func (c *Counter) Increment() {
    c.count++
}

func main() {
    counter := Counter{count: 0}
    fmt.Println(counter.count) // 输出: 0
    counter.Increment()
    fmt.Println(counter.count) // 输出: 1，因为Increment操作的是原结构体
}
```

3、结构体方法选择值类型接收者还是指针类型接收者?

值接收着:

- 接收的是调用者的副本。
- 对副本的修改不会影响原始值。
- 如果结构体较大，使用值接收者可能会导致性能问题，因为需要复制整个结构体。

指针接收着

- 接收的是调用者的副本。
- 对副本的修改不会影响原始值。
- 如果结构体较大，使用值接收者可能会导致性能问题，因为需要复制整个结构体。

> 如果方法需要修改接收者，或者接收者是一个大型结构体，应该使用指针接收者。 如果方法不需要修改接收者，或者接收者是一个小型结构体（如int、string、bool等），可以使用值接收者。

需要注意：

指针接收者可以调用值类型和指针类型的方法。 值接收者只能调用值类型的方法，不能直接调用指针类型的方法，即使该值是可寻址的（即可以取地址）。

map介绍

map是一种无序基于key-value的内置数据结构，提供了高效访问数据方式，key是唯一，且可以通过key快速检索、更新、删除对应的值。

由于map是引用类型，所以必须用make初始化才能用。

map语法：

```
map[keyType]valueType
```

其中KeyType是键的类型，ValueType是键对应的值的类型。

键可以是任何类型的，只要它可以进行比较，比如整数、浮点数、字符串、复数等。值可以是任何类型，包括其他map。

map原理

- 内部结构：Go中的map底层实现是一个哈希表。当插入键值对时，会先计算键的哈希值，然后根据哈希值确定其在表中的位置。如果发生冲突（两个不同的键哈希到同一位置），则通过链地址法（链表或平衡树）解决。
- 动态扩容：随着元素数量的增长，map会在需要时自动扩容，以保持高效的查询性能。扩容涉及到重新哈希和数据迁移，这个过程对用户透明。
- 零值：未初始化的map的值是nil，尝试访问一个nil map会引发运行时错误。使用make函数可以避免这种情况。

map定义

Golang中声明一个map的基本语法：

```
var mapName map[keyType]valueType
```

或者

```
mapName := make(map[keyType]valueType)
// 或者直接初始化健值对
mapName := map[keyType]valueType{key1: value1, key2: value2, ...}
```

举例：

```go
package main

import "fmt"

func main() {
    // 声明但未初始化map
    var Grade map[string]int

    // 初始化map
    Grade = make(map[string]int)

    // 直接初始化健值对
    Grade = map[string]int{"zhangsan": 10001, "lisi": 10002, "xiaoming": 10003}
    fmt.Println(Grade) //输出map[lisi:10001 xiaoming:10002 zhangsan:10003]
}
```

map操作

1、map里判断一个健是否存在：

```go
package main

import "fmt"

func main() {
    // 声明但未初始化map
    var Grade map[string]int

    // 初始化map
    Grade = make(map[string]int)

    // 直接初始化健值对
    Grade = map[string]int{"zhangsan": 10001, "lisi": 10002, "xiaoming": 10003}
    fmt.Println(Grade) //输出map[lisi:10001 xiaoming:10002 zhangsan:10003]

    // 判断某一个健是否存在
    val, ok := Grade["zhangsan"]
    if ok {
        fmt.Printf("zhangsan is a student in the class. His ID is %d \n", val)
    } else {
        fmt.Println("zhangsan is not found in the class.\n")
    }

    // 或者更简洁写法
    if val, ok := Grade["lisi"]; ok {
        fmt.Printf("lisi is a student in the class. His ID is %d \n", val)
    } else {
        fmt.Println("lisi is not found in the class.\n")
    }

}
```

2、修改、增加、删除map键值对

```
    // 修改值
    Grade["zhangsan"] = 10010
    fmt.Println(Grade) //输出map[lisi:10002 xiaoming:10003 zhangsan:10010]

    // 添加新健值对
    Grade["lili"] = 10005
    fmt.Println(Grade) //输出map[lili:10005 lisi:10002 xiaoming:10003
zhangsan:10010]

    // 删除健值对，需要用到内置函数delete(map, "key")
    delete(Grade, "xiaoming")
    fmt.Println(Grade) //输出map[lili:10005 lisi:10002 zhangsan:10010]

    // 遍历map
    for k, v := range Grade {
        fmt.Println(k, v)
    }
/*
lisi 10002
lili 10005
zhangsan 10010
*/
```

注意事项:

- map是引用类型，在作为函数参数传递时，传递的是引用的副本，因此在函数内部修改map会影响到外部，但替换整个map则不会。
- map的迭代是随机的，因此，每次迭代得到的结果可能不同。
- map的键值对在添加、删除操作后可能会发生扩容或缩容，这可能会改变已有键值对的内存地址。
- map在并发环境下，对同一个map进行读写操作可能会导致数据竞争，应使用sync.Map或其他并发安全机制。

接口(Interface)介绍

go语言中接口(interface)是一种抽象类型，它定义了一组由某个类型实现的方法，但是不包含具体实现，也就是说只定义规范不实现，由具体对象实现规范细节。

任何类型对象，只要实现了接口所定义的所有方法，就可以说它实现了这个接口。

这种设计让接口成为了Go语言实现多态的关键机制，即不同的具体类型可以以相同的方式交互，因为它们都实现了相同的接口。

接口定义

接口定义非常简单，它是一个方法签名集合，不包含任何实现：

格式如下:

```
type MyInterface interface {
    MyMethod1(paramType1 ParamType1) ReturnType1
    MyMethod2(paramType2 ParamType2) ReturnType2
}
```

例如：

```go
package main

import "fmt"

// Animal 接口定义了两个方法：Speak 和 Move
type Animal interface {
    Speak() string
    Move() string
}

// Dog 结构体实现了 Animal 接口
type Dog struct{}

// Speak 方法返回狗的叫声
func (d Dog) Speak() string {
    return "Woof!"
}

// Move 方法返回狗的移动方式
func (d Dog) Move() string {
    return "Running"
}

// Cat 结构体也实现了 Animal 接口
type Cat struct{}

// Speak 方法返回猫的叫声
func (c Cat) Speak() string {
    return "Meow!"
}

// Move 方法返回猫的移动方式
func (c Cat) Move() string {
    return "Walking"
}

// MakeAnimalSound 函数接受一个 Animal 类型的参数并调用其 Speak 方法
func MakeAnimalSound(a Animal) {
    fmt.Println(a.Speak())
}

func main() {
    // 创建 Dog 和 Cat 的实例
    dog := Dog{}
    cat := Cat{}

    // 调用 MakeAnimalSound 函数，传入不同的动物实例
    MakeAnimalSound(dog) // 输出：Woof!
    MakeAnimalSound(cat) // 输出：Meow!
}
```

这个例子中，我们定义了一个名为 Animal 的接口，它要求实现它的类型必须提供 Speak 和 Move 两个方法。然后我们定义了 Dog 和 Cat 两个结构体，并为它们分别实现了这两个方法。最后，我们编写了一个 MakeAnimalSound 函数，它接受任何实现了 Animal 接口的对象作为参数，并调用其 Speak 方法来打印动物的叫声。

当我们在 main 函数中创建 Dog 和 Cat 的实例并传递给 MakeAnimalSound 函数时，由于它们都实现了 Animal 接口，所以可以正常工作，并且根据传入的对象类型打印出相应的叫声。

面向对象

函数基础

Go语言中函数定义使用关键字func开始，其后跟函数名、参数列表（在圆括号内）、返回值列表（也在圆括号内，可选）和函数体。

函数可以是具体名称，也可以是匿名，即可执行特定任务，还可以作为其他函数参数进行传递(回调)。

1、基本语法:

```
func name (parameter1 type, parameter2 type) returnType {
    // 函数体
}
```

其中:

- func 是函数声明的关键字。
- name 是函数名，可以由字母、数字或下划线组成，但第一个字符不能是字母。
- parameter1 和 parameter2 分别代表接受的第一个和第二个参数。
- returnType 是返回类型。
- 函数体内包含了返回值列表，如果没有返回值，则可以省略返回值列表

举例:

```
func sum(x int, y int) int {
    return x + y
}
```

这个函数是一个求和函数，函数传递两个整数参数x和y，并计算出两个参数的和。

2、返回值:

- go语言中通过return关键字，向外输出返回值。
- 函数支持多个返回值，如果有多个返回值，必须用()将所有返回值包裹起来

```
func vals() (int, int) {
    return 3, 5
}
```

3、参数传递:

> 值传递：当函数被调用时，实际的参数值会被拷贝到函数的形式参数中。这意味着在函数内部对形式参数所做的任何修改，都不会影响到原始数据。

```go
package main

import "fmt"

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

func main() {
    num1 := 10
    num2 := 20
    result := sum(num1, num2)
    fmt.Println("Result:", result)
}
```

上面的例子中，sum函数接收的是num1和num2值的副本，无论sum函数如何使用这些值，都不会影响到num1和num2的原始值。

引用传递：

```go
package main

import "fmt"

func modifyPointer(p *int) {
    *p = 10
}
func main() {
    studentNumber := 100
    modifyPointer(&studentNumber)
    fmt.Println(studentNumber) //输出:10
}
```

上面的例子中，modifyPointer函数接收的是一个指向整数的指针。在函数内部对该指针所指向的值进行修改，实际上改变了原始的studentNumber变量的值。

函数特性

1、初始化函数

Go语言中的初始化函数（Init）是一种特殊的函数，主要用于在程序启动前对包进行初始化。这种函数自动执行，不需要手动调用，它在包被导入时立即执行

int函数特性：

- 自动执行，通常在main函数执行之前完成必要的初始化工作，如设置环境变量、加载配置文件等
- 无参数和无返回值，因此只能执行一些简单任务。

- 多个init函数，每个包可以包含多个init函数，每个源文件也可以包含多个init函数，具体执行顺序不确定。
- 执行顺序，go语言中没有明确定义同一个包内初始化函数执行顺序，但是不同包文件可以根据依赖来确定。
- 只能隐式调用（调用函数时不使用括号，后边回讲解），init函数不能被其他函数显式调用，只能在被包导入时执行。

举例：

```
package main

import "fmt"

func init() {
    fmt.Println("Init函数 设置启动变量，第一次初始化")
}

func init() {
    fmt.Println("Init函数 初始化配置文件，第二次初始化")
}

func main() {
    fmt.Println("Main函数")
}

/* 输出结果
Init函数 设置启动变量，第一次初始化
Init函数 初始化配置文件，第二次初始化
Main函数
*/
```

2、匿名函数 匿名函数，也称为lambda函数，是没有名称的函数。它们可以被赋值给变量、作为参数传递给其他函数或作为其他函数的返回值。

```
add := func(x, y int) int {
    return x + y
}
result := add(3, 4) // result is 7
```

3、闭包

Go语言中的闭包是一种强大的编程特性，它允许函数内部包含对外部作用域变量的引用。闭包可以访问其定义时的上下文环境中的变量，并且可以在调用之间保持状态。此外，闭包还可以被赋值给变量或作为参数传递给其他函数。

- 函数外部可以访问函数内部变量，与其他语言不通，Go语言中的闭包不会复制外围变量，而是共享这些变量。
- 闭包函数的返回值是匿名函数
- 避免函数内部环境（变量等）被外部污染，如Gin中间件
- 支持函数式编程，允许函数既可以返回一个函数，也可以接受一个函数作为参数

通常，闭包常与Goroutine和channel一起使用，以实现并发编程中的任务管理和数据流控制。

例如，可以使用闭包来创建一个计数器，每次调用都会增加计数器的值，并通过闭包保持这个计数器的状态。

```
package main

import "fmt"

// 创建计数器函数
func createCounter() func() int {
    count := 0 // 这个变量将被闭包捕获并维持其状态

    // 返回一个匿名函数作为计数器，这个匿名函数就是闭包
    return func() int {
        count++ // 每次调用时，count的值加1
        return count // 返回当前的计数器值
    }
}

func main() {
    // 调用createCounter获得一个新的计数器函数
    counter := createCounter()

    // 演示计数器的使用
    fmt.Println(counter()) // 输出：1
    fmt.Println(counter()) // 输出：2
    fmt.Println(counter()) // 输出：3

    // 注意，每个新的createCounter调用都会产生一个独立的计数器
    anotherCounter := createCounter()
    fmt.Println(anotherCounter()) // 输出：1，因为这是另一个独立的计数器
}
```

这个例子中，createCounter函数内部定义了一个局部变量count，并且返回了一个匿名函数。这个匿名函数是一个闭包，因为它引用了createCounter函数作用域内的非局部变量count。每次调用这个匿名函数时，它都会使count的值增加1并返回当前的计数值。由于闭包维持了对外部变量count的引用，因此计数器的状态得以保留，即使在createCounter函数执行完毕后也是如此。

函数作用域

- 全局变量：在任何函数外定义，对整个包内的函数可见，可通过导出（首字母大写）使其对其他包可见。
- 局部变量：函数内部定义的变量，仅在该函数内部可见。
- 块作用域：在if、for、switch等语句块内部定义的变量，仅在该块内或嵌套的更小块内可见。
- 特例：闭包可以访问函数定义时所在作用域内的变量，即使在其外部函数已经返回。。

高级用法

- 函数作为参数和返回值：Go函数可以接受其他函数作为参数，也可以返回函数。
- defer：defer关键字用于延迟执行函数，通常用于资源清理，如关闭文件或解锁互斥锁，会在函数返回之前执行。
- panic与recover：用于处理异常情况。panic会导致当前函数停止执行并向上抛出错误，直至被recover捕获或程序终止。
- 类型转换与接口实现：函数可以作为特定类型或接口的一部分，促进多态行为。

函数工厂模式

Go语言是一种静态类型、编译型语言，并且原生不支持类和继承等面向对象的特性，因此它采用了一种不同的方式来实现类似面向对象的编程范式。

Go语言通过结构体（struct）和方法（method）来模拟对象和对象的行为，而函数的工厂模式是实现对象创建的一种常用模式。

工厂模式是一种在软件工程中用来创建对象的设计模式，它不会显式地要求使用new运算符实例化一个类。在Go中，你可以通过定义一个工厂函数来创建并返回结构体实例，这样的函数通常以构造器的形式存在，并返回一个指针。

举例：

```
package main

import "fmt"

type Person struct {
    Name string
    Age  int
}

// 定义工厂函数，接收姓名和年龄，并返回Person结构体实例
func NewPerson(name string, age int) *Person {
    return &Person{
        Name: name,
        Age:  age,
    }
}

// 为Person定义一个方法
func (p *Person) Greet() {
    fmt.Printf("Hello, my name is %s and I am %d years old. \n", p.Name, p.Age)
}
func main() {
    // 使用工厂函数创建Person对象
    zhangsan := NewPerson("zhangsan", 25)
    lisi := NewPerson("lisi", 30)

    // 调用对象方法
    zhangsan.Greet()
    lisi.Greet()
}
```

使用工厂模式的好处是它提供了一个统一的接口来创建对象，使得创建逻辑更加集中，便于维护和扩展。

同时，它也允许你在创建对象时执行一些额外的初始化操作，或者根据不同的参数返回不同的结构体实现，从而实现多态。

结构体嵌套继承

Go语言不支持传统意义上的类继承概念，但是可以通过结构体嵌套来实现代码的复用和组合。

举例：

```
package main

import "fmt"

// Animal 基础结构体，包含共享属性和方法
type Animal struct {
    Name string
}

// Speak 方法
func (a Animal) Speak() string {
    return fmt.Sprintf("%s makes a noise.", a.Name)
}

// Dog 类型，通过包含Animal实例实现"继承"
type Dog struct {
    Animal // 匿名字段，相当于Dog继承了Animal的所有字段和方法
    Breed  string
}

func main() {
    dog := Dog{
        Animal: Animal{Name: "Fei fei"},
        Breed:  "Shepherd",
    }
    fmt.Println(dog.Speak()) // 输出: Fei fei makes a noise.
}
```

其实，在go语言实现继承的方式，处了结构体嵌套，还可以通过接口多态的形式实现，这个后边会讲到。

接口多态

Go语言中，接口多态体现在用一个接口类型变量来引用实现了该接口的任何类型的实例，并且通过这个接口调用他们的共有方法， 而无需了解底层类型。

举例：

```go
package main

import "fmt"

// SoundMaker 接口，定义了MakeSound方法
type SoundMaker interface {
    MakeSound()
}

// Dog 狗的结构体
type Dog struct{}

// MakeSound 实现了SoundMaker接口
func (d Dog) MakeSound() {
    fmt.Println("Woof!")
}

// Cat 猫的结构体
type Cat struct{}

// MakeSound 实现了SoundMaker接口
func (c Cat) MakeSound() {
    fmt.Println("Meow!")
}

// Cow 牛的结构体
type Cow struct{}

// MakeSound 实现了SoundMaker接口
func (co Cow) MakeSound() {
    fmt.Println("Moo!")
}

// makeSounds 函数，接受任何实现了SoundMaker接口的类型
func makeSounds(soundMakers ...SoundMaker) {
    for _, maker := range soundMakers {
        maker.MakeSound()
    }
}

func main() {
    dog := Dog{}
    cat := Cat{}
    cow := Cow{}

    // 不同类型的动物对象通过接口多态性调用MakeSound方法
    makeSounds(dog, cat, cow)
}
```

在这个例子中，我们定义了一个SoundMaker接口，它声明了一个MakeSound方法。接着，我们定义了Dog、Cat和Cow三种动物结构体，并各自实现了MakeSound方法，因此它们都隐式地实现了SoundMaker接口。

makeSounds函数接受任意数量的SoundMaker接口类型的参数，这意味着它可以接受任何实现了MakeSound方法的类型的实例。当我们调用makeSounds(dog, cat, cow)时，即使传入的具体类

型不同，但因为它们都实现了相同的接口，函数能够正确地调用每个动物的MakeSound方法，分别输出它们的叫声。这就是Go语言中接口多态性的体现。

空接口

Go语言中，空接口是指没有定义任何方法的接口。因此，空接口可以表示任何类型(泛型概念)。

空接口没有表示任何约束，因此任何类型变量都可以实现空接口。

空接口在Go中有多种用途，以下是常用的场景和例子：

空接口作为函数的参数：

举例:

```
package main

import "fmt"

func PrintAnything(a interface{}) {
    fmt.Printf("值:%v 类型:%T \n", a, a)
}
func main() {
    PrintAnything("Hello, world!")                          // 打印字符串
    PrintAnything(25)                                       // 打印整数
    PrintAnything(true)                                     // 打印bool
    PrintAnything([]int{1, 2, 3, 4, 5, 6, 7})               // 打印一个整形切片
    PrintAnything([5]string{"apple", "banana", "orange"}) // 打印map
    PrintAnything(struct {
        Name string
    }{Name: "zhangsan"}) //打印结构体
}
```

空接口作为容器

空接口可以创建用于存储任何类型值的容器，这里我们举例使用切片或者容器来实现空接口。

切片实现空接口

切片本身是一种复合数据类型，它并不直接实现空接口，但任何类型的切片（包括切片本身）都可以被赋值给空接口变量，因为所有类型都隐式实现了空接口。

所以，你可以使用空接口来存储、传递或操作任何类型的切片。

```go
package main

import "fmt"

func main() {
    // 创建一个空接口切片，用于存储不同类型的切片
    var slices []interface{}

    // 添加整数切片
    intSlice := []int{1, 2, 3}
    slices = append(slices, intSlice)

    // 添加字符串切片
    strSlice := []string{"a", "b", "c"}
    slices = append(slices, strSlice)

    // 遍历并打印每个切片的内容
    for _, v := range slices {
        switch v := v.(type) {
        case []int:
            fmt.Println("Integer Slice:", v)
        case []string:
            fmt.Println("String Slice:", v)
        default:
            fmt.Println("Unknown Slice Type")
        }
    }
}
```

map实现空接口

map也不直接实现空接口，但是任何类型的map都可以赋值给空接口变量，因为所有类型包括map都隐式实现了空接口。

因此，你可以使用空接口来存储、传递或操作任何类型的map

```go
package main

import "fmt"

func main() {
    // 创建一个空接口类型的map，用于存储不同类型的map
    var maps map[string]interface{}

    // 初始化maps
    maps = make(map[string]interface{})

    // 添加整数键值对的map
    intMap := map[int]string{1: "one", 2: "two"}
    maps["intMap"] = intMap

    // 添加字符串键值对的map
    strMap := map[string]int{"a": 1, "b": 2}
    maps["strMap"] = strMap

    // 访问并打印每个map的内容
    for key, value := range maps {
        switch value := value.(type) {
        case map[int]string:
            fmt.Printf("%s: Integer String Map - %v\n", key, value)
        case map[string]int:
            fmt.Printf("%s: String Integer Map - %v\n", key, value)
        default:
            fmt.Printf("%s: Unknown Map Type\n", key)
        }
    }
}
```

断言

Go语言中，空接口（interface{}）可以存储任何类型的值，但是在使用这些值之前，通常需要将它们转换回具体的类型。这个过程称为类型断言。

类型断言主要用于提取存储在空接口中的具体类型的值

其语法：x.(T)

x表示类型为interface{}的变量，T表示断言x的可能类型

在进行类型断言时，最好使用安全类型断言进行判断处理，不然断言失败容易触发panic，格式如下：

```go
value, ok := container.(int)
if ok {
    // 在这里安全地使用value作为int类型
} else {
    // 类型断言失败的处理
}
```

举例：

```
package main

import "fmt"

func main() {
    var x interface{} = "Hello, World!"
    if value, ok := x.(string); ok {
        fmt.Println("Type assertion successded:", value)
    } else {
        fmt.Println("Type assertion failed.")
    }
}
```

请注意，使用空接口会牺牲类型安全，因为在使用空接口的值之前，通常需要进行类型断言来恢复具体的类型信息。此外，空接口的值会涉及额外的间接访问和类型断言的开销，这可能会影响性能。因此，在设计程序时，应该权衡使用空接口的便利性和潜在的性能成本。

反射

Go语言中的反射机制允许程序在运行时检查和修改其数据类型和值。

这种能力对于编写通用代码、序列化/反序列化、框架开发以及需要动态类型操作的场景尤为重要。

Go的反射API主要由标准库中的reflect包提供，主要包括reflect.Type和reflect.Value两个核心类型。

基本步骤：

- 获取反射对象：使用reflect.TypeOf()或reflect.ValueOf()函数从接口值获取类型信息或值信息。
- 类型检查与转换：通过反射对象，可以检查类型信息，如类型名称、类型种类等。
- 操作值：若反射对象表示的值是可写的，可以修改该值或调用其方法。

假设我们想编写一个函数，能够打印任何类型变量的值及其类型，无论该变量事先是否已知。

```go
package main

import (
    "fmt"
    "reflect"
)

// printInfo 打印变量的类型和值
func printInfo(value interface{}) {
    // 获取反射类型对象
    typ := reflect.TypeOf(value)
    // 获取反射值对象
    val := reflect.ValueOf(value)

    fmt.Printf("Type: %v\n", typ)
    fmt.Printf("Value: %v\n", val)

    // 检查并展示值是否可写
    fmt.Printf("Value is settable: %v\n", val.CanSet())

    // 如果值是可写的，并且是int类型，尝试修改其值
    if val.Kind() == reflect.Int && val.CanSet() {
        val.SetInt(val.Int() + 1)
        fmt.Printf("Modified Value: %v\n", val)
    }
}

func main() {
    num := 10
    printInfo(num)

    str := "hello"
    printInfo(str)

    // 注意：反射不能修改不可变类型的值，比如字符串
}
```

在这个例子中，我们使用 reflect.TypeOf 和 reflect.ValueOf 函数来获取变量的类型和值。然后，我们检查这个值是否可以被设置（CanSet），如果可以，我们使用 SetInt 方法来修改它。最后，我们使用反射来调用值的方法。

反射是一种强大的工具，但应该谨慎使用。因为它会破坏类型安全和静态检查，容易导致错误和性能问题。通常，只有在无法预先知道数据类型的情况下，或者需要实现某些高级特性（如泛型编程）时，才考虑使用反射。

并发

并发编程

在学习go并发编程之前，我们需要理解几个概念，比如并发、并行、串行以及进程、线程、协程

1、串行、并发和并行：

串行（Serial）

- 定义：串行执行是指程序或任务按照一定的顺序，一个接一个地执行。在没有特别安排的情况下，大多数传统程序都是串行执行的。
- 特点：执行过程简单，不需要复杂的任务协调和资源管理。但当任务较多或某个任务耗时较长时，整体执行效率会降低，因为后续任务必须等待前一个任务完成才能开始。
- 应用场景：对于资源受限或任务间依赖性强的场景较为适用。

并发（Concurrent）

- 定义：并发是指在一段时间内，多个任务都开始了执行，但不保证所有任务同时执行。它可以发生在单核或多核系统上，重点在于任务的启动时机而非实际执行的重叠。
- 特点：提高了系统的响应速度和处理能力，可以更好地利用CPU等待时间（如I/O操作时）。通过任务的交错执行，给人一种“同时”处理多个任务的感觉。
- 应用场景：适合于I/O密集型应用（如网络服务）、用户界面交互等，能有效提升用户体验。

并行（Parallel）

- 定义：并行则是指在多核或多处理器系统中，多个任务能够同时执行，即物理上的同时处理。这是并发的一种特殊情况，要求硬件支持同时处理多个计算单元。
- 特点：充分利用了多核处理器的能力，可以显著提高计算密集型任务的执行效率。但并行编程相对复杂，需要考虑数据一致性、同步等问题。
- 应用场景：适用于计算密集型任务，如大数据处理、科学计算、图像渲染等，这些场景下，任务之间相对独立，可以被拆分到不同处理器上并行处理。

对比：

- 并发和并行是两个相关但不同的概念。并发强调的是任务在时间上的重叠，而并行强调的是任务在空间上的重叠。
- 并发可以在单核或多核处理器上执行，而并行通常指在多核或多处理器上同时执行多个任务。
- 并发可能不需要大量的资源，但需要有效地管理任务；并行需要有足够的资源来支持多个任务的同时执行。

2、进程：

进程是操作系统进行资源分配和调度的基本单位，每个进程都有独立的内存空间和系统资源（如打开的文件）。一个程序运行起来后，就成为一个进程。

进程之间相对独立，通信和数据共享通常需要通过进程间通信（IPC）机制，如管道、套接字等。

3、线程：

线程是进程内的执行单元，是CPU调度的基本单位。多个线程可以共享同一进程的内存空间和资源，实现并发执行。 相比于进程，线程更轻量级，线程间的通信和切换开销较小。但线程依然受到操作系统管理，创建和销毁成本较高，且存在资源竞争和同步问题。

4、协程(goroutine)

Go语言中的协程是一种用户态的轻量级线程，由Go运行时（runtime）管理。协程的创建和切换成本远低于线程，可以在一个进程中轻松创建成千上万个协程。

协程在Go语言中实现了高效率的并发执行，它们在一个或多个操作系统线程上调度执行，由Go runtime自动管理线程和协程之间的关系。相对于线程，协程的控制权更多地交给了程序员，简化了并发编程的复杂度。

goroutine

Goroutine是Go中的并发执行单元，它是轻量级的线程，由Go运行时管理。

启动一个goroutine非常简单，只需要在函数调用前加上关键字go即可。与操作系统线程相比，goroutine的创建和切换开销极小，使得Go程序可以轻松创建成千上万个并发任务。

定义：

```
go funcName(params)
```

这将创建一个Goroutine并执行funcName函数，并发地运行。

举例：

```go
package main

import (
    "fmt"
    "time"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
    go say("World")                      // 启一个新的goroutine执行say函数
    say("Hello")                         // 在main的goroutine中执行say函数
    time.Sleep(600 * time.Millisecond) //等待足够时间让goroutine执行完成。
}

/* 执行结果:
Hello
World
World
Hello
World
Hello
Hello
World
World
Hello
*/
```

Go协程

1、go协程和主程序退出顺序:

举例

```
package main

import (
    "fmt"
    "time"
)

func test() {
    for i := 0; i < 10; i++ {
        fmt.Println("test() 你好")
        time.Sleep(100 * time.Millisecond)
    }
}

func main() {
    go test()
    for i := 0; i < 2; i++ {
        fmt.Println("main()  你好")
        time.Sleep(100 * time.Millisecond)
    }
}

/*打印结果:
main()  你好
test() 你好
test() 你好
main()  你好
*/
```

这个例我们看到:

- 主进程每隔1秒打印一次，总共打印2次；协程每隔1秒打印一次，总共打印10次；
- 主进程和协程的打印是交替的，说明是并行执行的；
- 协程只打印了两次就退出了（主程序已经退出了），其实这种情况就很尴尬了。

2、WaitGroup

对于第一个例子里协程还没执行完的情况，主线程已经退出了，这种情况下主进程可以使用sync.WaitGroup来等待协程执行完成。

WaitGroup是通过内部的一个计数器来跟踪goroutine的完成情况。

原理:

- sync.WaitGroup内部维护了一个原子性的计数器，这个计数器可以增加和减少，初始值为0。计数器的增加通常在启动goroutine之前通过Add方法完成，减少则在goroutine执行完毕后通过Done方法完成。此外，还有一个Wait方法用于阻塞调用它的goroutine，直到计数器归零，表示所有等待的goroutine都已经完成。

定义：

```
var wg sync.WaitGroup  // 第一步：声明变量，并初始化一个实例(定义一个计数器)
wg.Add(1) // 第二步：启动每个goroutine之前，开启一个协程计数器就+1
wg.Done() // 第三步：协程执行完成，计数器-1（这里每个goroutine内部应当包含一个defer wg.Done()语句来调用Done）
wg.Wait() // 第四步：计数器为0时退出
```

修改第一个例子如下：

```go
package main

import (
    "fmt"
    "sync"
    "time"
)

var wg sync.WaitGroup

func test1() {
    for i := 0; i < 5; i++ {
        fmt.Println("test1() 你好", i)
        time.Sleep(100 * time.Millisecond)
    }
    defer wg.Done()
}

func test2() {
    for i := 0; i < 2; i++ {
        fmt.Println("test2()  你好", i)
        time.Sleep(100 * time.Millisecond)
    }
    defer wg.Done()
}

func main() {
    wg.Add(1)
    go test1()
    wg.Add(1)
    go test2()
    wg.Wait()
    fmt.Println("主线程退出。。。")
}

/*打印结果:
test2()  你好 0
test1() 你好 0
test1() 你好 1
test2()  你好 1
test1() 你好 2
test1() 你好 3
test1() 你好 4
主线程退出。。。
*/
```

sync.WaitGroup通过一个计数器和阻塞等待机制，提供了一种简单且高效的方式来同步并发任务的完成。它使得开发者能够方便地控制并发流程，确保关键操作的顺序执行，从而避免了复杂的锁和条件变量的直接使用，简化了并发编程。

3、开启多个协程

举例：

```go
package main

import (
    "fmt"
    "sync"
    "time"
)

var wg sync.WaitGroup

func worker(id int) {
    wg.Done() // goroutine结束前调用
    fmt.Printf("Worker %d is running \n", id)
    time.Sleep(2 * time.Second) // 模拟goroutine正在执行任务。
    fmt.Printf("Worker %d is done \n", id)
}

func main() {
    for i := 0; i < 5; i++ {
        wg.Add(1) // 初始化等待goroutine数量
        go worker(i)
    }
    wg.Wait() // 等待所有goroutine执行完成
    fmt.Println("所有Goroutine执行完成，主程序退出！！")
}

/*
Worker 0 is running
Worker 1 is running
Worker 4 is running
Worker 3 is running
Worker 2 is running
所有Goroutine执行完成，主程序退出！！
*/
```

这里5个goroutine是并发执行，而函数任务调度是随机的。

通道

Channel是Go语言提供的用于goroutines之间安全通信的管道，可以发送和接收数据。Channel有方向性，可以是发送通道（<-chan）或接收通道(chan<-)，也可以是双向通道(chan)。通过channel进行通信，可以有效地解决并发编程中的同步问题，避免了复杂的锁机制。

基本语法:

```
ch := make(chan type)
```

或者

```
ch := make(chan type, 10) // 定义缓冲通道，不想一下子接受太多，可以定义缓冲区域大小，这里通道最多存储10个值。
```

如果channel的缓冲区满了，发送操作将会阻塞，直到有空间可用。接收操作则不会阻塞，如果缓冲区中有值，它会立即接收

1. 通道基本发送和接受：

```
package main

import "fmt"

func main() {
    var s1 = []int{1, 2, 3, 4} // 声明一个变量s1，并赋值初始化变量
    ch := make(chan []int) // 定义一个只能发送切片类型的channel
    go func() {
        ch <- s1 // 向通道传递数据
    }()
    num := <-ch // 从通道中接收数据
    fmt.Println(num)
}
```

2. 定义有缓冲通道：

```go
package main

import "fmt"

func main() {
    message := make(chan int, 2) // 创建一个带有缓冲区域的channel，缓冲大小为2

    // 发送消息到通道，即使没有接收方也不会阻塞(直到缓冲区满)
    message <- 1
    message <- 2

    // 接收消息
    fmt.Println(<-message)
    fmt.Println(<-message)
}
```

3. range和close

```go
package main

import "fmt"

func generateNumber(ch chan<- int) {
    for i := 0; i < 5; i++ {
        ch <- i
    }
    close(ch) // 关闭通道，表示不再接收数据
}

func main() {
    numbers := make(chan int) // 定一个只接收整形的通道
    go generateNumber(numbers)

    // for-range循环接收通道数据，直到通道关闭
    for num := range numbers {
        fmt.Println("Received:", num)
    }
}
```

select

在Go语言中，select语句是一种用于在多个通道（channel）操作上进行同步和通信的控制结构,并且是阻塞的，直到有一个case可以继续执行。

它类似于其他语言中的switch语句，但专门用于处理通道操作，并且可以看作是一个等待多个事件的复用器。

1. 基本用法:

```
select {
case <-ch1:
    // 处理ch1的接收操作
case ch2 <- someData:
    // 处理ch2的发送操作
default:
    // 当没有任何case准备好时执行的代码
}
```

select特性：

- 公平性：如果多个case都准备好了，select会随机但公平地选择一个执行。这意味着如果有一个case被频繁选中，它不会一直被选中，从而保证了各通道间的均衡处理。
- 非阻塞：当至少有一个case可以执行时，select会立即执行一个操作。如果没有case准备好，且存在default分支，则执行default分支。
- 多路复用：可以在一个select语句中监听多个通道，有效管理多个并发操作。
- 关闭通道检测：可以通过select判断一个通道是否被关闭，这对于优雅地终止goroutines和清理资源非常有用。

2. select监听多个channel 举例：

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    c1 := make(chan string)
    c2 := make(chan string)

    go func() {
        time.Sleep(1 * time.Second)
        c1 <- "one"
    }()
    go func() {
        time.Sleep(2 * time.Second)
        c2 <- "two"
    }()

    // 使用select监听多个channel
    for i := 0; i < 2; i++ {
        select {
        case msg1 := <-c1:
            fmt.Println("Received from c1:", msg1)
        case msg2 := <-c2:
            fmt.Println("Received from c2:", msg2)
        }
    }
}
```

3. select语句中实现超时功能：

默认的case并不支持超时。如果你想在select语句中实现超时功能，你可以使用time.Sleep函数或者time.After函数来模拟超时。

举例：

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    c1 := make(chan string, 1)
    c2 := make(chan string, 1)

    go func() {
        time.Sleep(2 * time.Second) // 模拟耗时操作
        c1 <- "message from c1"
    }()

    go func() {
        time.Sleep(1 * time.Second) // 模拟耗时操作
        c2 <- "message from c2"
    }()

    select {
    case msg := <-c1:
        fmt.Println("Received:", msg)
    case msg := <-c2:
        fmt.Println("Received:", msg)
    case <-time.After(3 * time.Second): // 超时时间为3秒
        fmt.Println("Timeout occurred")
    }
}
```

在这个例子中，我们有两个通道c1和c2，以及一个超时时间为3秒的case。由于c1通道上的goroutine会睡眠2秒，而c2通道上的goroutine会睡眠1秒，所以在这个例子中，select将会接收到c2通道上的消息，而不是执行超时case。

互斥锁介绍

在Go语言中，sync.Mutex（互斥锁）是sync包提供的一个基础同步原语，用于保护共享资源免受并发访问的影响，防止数据竞争和并发冲突。

互斥锁通过以下方式确保同一时间只有一个goroutine可以访问被保护的资源：

举例：

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
    var mu sync.Mutex
    counter := 0

    var wg sync.WaitGroup
    for i := 0; i < 10; i++ {
        wg.Add(1) // 启动goroutine之前，计数器+1
        go func() {
            counter++
            defer wg.Done()                // goroutine执行完成,计数器-1
            mu.Lock()                      // 访问共享资源前，获取锁
            defer mu.Unlock()              // 完成访问后，释放锁
            time.Sleep(time.Microsecond) // 模拟耗时操作
        }()
    }
    wg.Wait()
    fmt.Println("最后统计：", counter) // 输出最后统计：  10
}
```

原理:

- Lock: 当一个goroutine尝试获取锁时，如果锁未被其他goroutine持有，则成功获取并将其状态标记为已锁定；如果锁已被持有，则该goroutine会被阻塞，直到锁被释放。
- Unlock: 当goroutine完成对共享资源的操作后，调用Unlock方法释放锁，此时如果有其他goroutine正在等待该锁，其中一个（通常是等待时间最长的那个）会被唤醒并尝试获取锁。
- 饥饿避免: Go在某些版本中引入了“饥饿避免”机制，确保等待中的goroutine不会永远等待下去，特别是在高竞争的情况下，尽量让等待时间公平。

互斥锁结构

```
type Mutex struct {
    state int32
    sema  uint32
}
```

- state字段记录锁的状态，包括是否被锁定以及是否有goroutine正在等待。
- sema字段是一个信号量，用于控制goroutine的阻塞和唤醒。

举例：

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
    var mu sync.Mutex
    var sharedResource int

    go func() {
        mu.Lock()
        sharedResource++
        fmt.Println("goroutine 1")
        time.Sleep(1 * time.Second) // 模拟耗时
        mu.Unlock()                 // 访问完成，释放锁
    }()

    go func() {
        mu.Lock()
        sharedResource++
        fmt.Println("goroutine 2")
        mu.Unlock()
    }()

    time.Sleep(2 * time.Second) // 等待所有goroutine执行完成
    fmt.Println("最后共享资源的值为：", sharedResource)
}

/* 输出结果为:
goroutine 1
goroutine 2
最后共享资源的值为：  2
*/
```

在这个示例中，两个goroutine尝试递增同一个共享变量，但由于互斥锁的存在，每次只有一个goroutine能够访问该变量，从而避免了数据竞争。

Go标准库

fmt

fmt是Go语言中的一个核心标准库，全称格式化输入输出（Formatting I/O），它提供了丰富的函数来处理文本的格式化输出和输入。fmt包的设计灵感来源于C语言的printf和scanf家族函数，但在功能和安全性上有所增强。

简介

fmt包主要用于处理字符串的格式化操作，包括格式化输出到标准输出（通常是终端）、格式化为字符串、从标准输入读取数据等。它支持多种数据类型的格式化，并且通过格式化字符串中的占位符来控制输出格式。

基本特性

- 占位符：fmt支持丰富的格式化占位符，如 %d（十进制整数）、%s（字符串）、%f（浮点数）等，以及宽度、精度等控制输出格式的选项。
- 颜色输出：虽然fmt本身不直接支持颜色输出，但可以通过转义序列结合字符串格式化实现。
- 类型检查：Printf等函数会在编译时检查格式化字符串与参数类型是否匹配，避免运行时错误。

基本使用

格式化输出：

```
fmt.Println("Hello, World!")                        // 自动换行
fmt.Printf("Value: %d, Name: %s\n", 42, "Alice") // 格式化输出，需手动换行
fmt.Print("No newline at the end")                 // 不自动换行
```

格式化输入：

fmt.Scan 直接读取输入并填充变量，遇到空格、换行等分隔符停止。

```
var name string
var age int
fmt.Print("Please enter your name: ")
fmt.Scan(&name) // 读取字符串

fmt.Print("Please enter your age: ")
fmt.Scan(&age) //读取整数
fmt.Printf("Hello %s ! Your are %d years old.\n", name, age) // Hello larry !
Your are 25 years old.
```

fmt.Scanf 类似于 scanf，可以指定输入格式。

```
fmt.Print("Enter name and age (e.g., Alice 30): ")
fmt.Scanf("%s %d", &name, &age) // 格式化读取

fmt.Printf("Welcome, %s! You are %d years old.\n", name, age) // Welcome,
Larry! You are 25 years old.
```

- 使用fmt进行输入输出时，要注意错误处理，例如，Scan和Scanf可能会遇到无法解析的输入，此时可以通过检查返回的错误来处理这种情况。
- 当从标准输入读取数据时，如果用户输入的数据不符合预期的格式，程序可能会出现未定义行为或错误，因此在实际应用中应做好充分的错误处理和验证。

高级用法

fmt.Sprint

Sprintf将格式化后的字符串返回而不是打印，适用于构建字符串。

fmt.Fprint

Fprintf，可以将格式化后的数据输出到任何实现了io.Writer接口的地方，比如文件或网络连接。

动态精度与宽度

可以使用*来指定宽度或精度为动态值，这些值作为额外的参数传递。

```
width := 10
fmt.Printf("%*d", width, 30) # 输出宽度为10的42
```

格式化布尔值、指针等

%t用于布尔值，%p用于指针地址。

格式化对齐

格式化字符串对齐由正常文本和以%开始的转换说明符组成。

转换说明符的一般形式为

```
%[flags][width][.precision]specifier，其中：
```

```
[flags]是可选的标志，如-(左对齐)，+(总是显示符号)，#(特殊格式)，0(用零填充)等。
[width]指定输出的最小宽度。
[.precision]用于指定精度，如浮点数的小数位数。
specifier指定了要格式化的数据类型。
```

- 对于字符串：%-（左对齐）或%+（右对齐）控制字符串的对齐方式。
- 对于整形：%Nd和%-Nd 分别控制右对齐和左对齐；而对于%+Nd或%-Nd 控制正负号的显示。

```go
fmt.Printf("|%6s|%6s|\n", "Name", "Age") //使用宽度指定列，右对齐
fmt.Printf("|%-6s|%3d|\n", "Larry", 30)  // 使用-左对齐，并指定宽度

fmt.Printf("%5d\n", 42)  // 输出:   42 （前方有两位空格）
fmt.Printf("%-5d\n", 42) // 输出: 42    （后方有三位空格）

fmt.Printf("%+5d\n", 42)  // 输出: +42 （前方有一位空格）
fmt.Printf("%-+5d\n", 42) // 输出: +42   （后方有两位空格，同时显示正号）
```

```
%6s 表示输出至少占6个字符宽度的字符串，右对齐。
%-6s 表示输出至少占6个字符宽度的字符串，左对齐。
%3d 表示输出至少占3个字符宽度的十进制数，右对齐。
%5d 表示输出至少占5个字符宽度的十进制数，右对齐
%-5d 表示输出至少占5个字符宽度的十进制数，左对齐

%+5d 表示输出至少占5个字符宽度的十进制数，+作为前缀标志来控制正负号的显示
%-+5d 表示输出至少占5个字符宽度的十进制数，-作为前缀标志来控制正负号的显示
```

fmt包的强大之处在于其灵活性和易用性，几乎可以满足所有基本的文本格式化需求。通过合理利用fmt，可以轻松地提升Go程序的输出质量和可读性。

time

Go语言中的time包是用于处理时间和日期的标准库之一，提供了丰富而强大的功能，适用于各种计时、定时任务和日期格式化等应用场景。

基本特性

- Time类型：表示一个时间点，包含年、月、日、时、分、秒、纳秒等信息，比如time.Time类型。
- 时间运算：可以计算两个时间点的差值，得到Duration类型表示的时间间隔。
- 时间格式化：使用Format方法按照指定格式输出时间字符串，格式化字符串遵循Go特定的规则。
- 定时器：通过time.NewTimer或time.AfterFunc创建定时任务。
- Tickers：周期性触发事件，通过time.NewTicker创建。

获取当前时间

获取本地时间：

```
package main

import (
    "fmt"
    "time"
)

func main() {
    now := time.Now()
    fmt.Println("The time is:", now)

    // 获取年、月、日、时、分、秒等信息
    year := now.Year()
    month := now.Month()
    day := now.Day()
    hour := now.Hour()
    minute := now.Minute()
    second := now.Second()
    nanosecond := now.Nanosecond()
    microsecond := nanosecond / 1000

    // 输出具体时间信息
    fmt.Printf("year: %d, month: %d, day: %d\n", year, month, day)
    fmt.Printf("hour: %d, minute: %d, second: %d\n", hour, minute, second)
    fmt.Printf("second: %d microsecond: %d , nanosecond: %d \n", second,
microsecond, nanosecond)
}
```

基本使用

```
now := time.Now()                                  // 获取当前时间
strTime := now.Format("2006-01-02 11:04:05") // 格式化时间，这里的2006-01-02 11:04:05是固定格式不变，任何时间格式化都用这个模版
fmt.Println(strTime)                               // 输出2024-05-22 55:00:37
fmt.Println(now.Unix())                            // 转换为时间戳，输出1716372037

duration := time.Minute + 30*time.Second // Duration类型，表示两个时间点之间的时间间隔，支持加减运算和转换
fmt.Println(duration.String())

parsedTime, err := time.Parse("2006-01-02 15:04:05", "2023-04-01 12:34:56") // 解析时间字符串，如果解析失败则返回nil
if err != nil {
    fmt.Println(err)
} else {
    fmt.Println(parsedTime)
}
```

经典案例：

定时任务：

实现一个每10秒打印当前时间的定时器。

```
package main

import (
    "fmt"
    "time"
)

func main() {
    // 创建一个Ticker，每隔5秒(5*time.Second)触发一次
    ticker := time.NewTicker(5 * time.Second)
    defer ticker.Stop()
    for {
        select {
        case <-ticker.C:
            fmt.Println(time.Now().Format("2006-01-02 15:04:05"))
        }
    }
}
```

这个例子中，声明Ticker，使用time.NewTicker创建了一个新的Ticker实例。Ticker是time包中的一个类型，用于按固定时间间隔发送时间信号。这里设置的时间间隔是5秒（5 * time.Second），意味着每隔10秒，Ticker会发出一个信号。

defer确保在当前函数（或者当前代码块）执行结束前，Ticker会被停止。调用ticker.Stop()会停止Ticker不再发送时间信号，这是一个很好的实践，用来防止在不再需要时还继续运行定时任务，从而浪费资源。

ticker.C是一个channel，Ticker会通过这个channel发送时间信号。

时间解析：

当你有一个字符串形式的时间，需要将其转换成time.Time类型时，可以使用time.Parse函数

```
package main

import (
    "fmt"
    "time"
)

func main() {
    timeStr := "2024-04-01 10:30:00"
    layout := "2006-01-02 15:04:05"
    parsedTime, err := time.Parse(layout, timeStr)
    if err != nil {
        fmt.Println("解析错误:", err)
        return
    }
    fmt.Println("解析后的时间为:", parsedTime)
}
```

这里time.Parse("2006-01-02 15:04:05", timeStr)将字符串timeStr按照指定的格式解析成了time.Time类型。

延迟执行：

---

time.AfterFunc是Go语言标准库time包中的一个函数，它用于在指定的延迟时间之后执行一个函数。这个函数返回一个*time.Timer，该计时器会在到期时调度执行提供的函数。

举例：

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    // 延迟5秒后执行hello函数
    timer := time.AfterFunc(5*time.Second, hello)

    // 阻塞主goroutine，防止程序提前退出
    <-timer.C
}

func hello() {
    fmt.Println("Hello after 5 seconds!")
}
```

在这个例子中，time.AfterFunc(5*time.Second, hello)创建了一个新的计时器，该计时器在5秒后会调用hello函数。<-timer.C这一行是等待计时器的通道C发送信号，这可以防止主goroutine提前结束，确保计时器有机会执行。

os

os包是Go语言标准库中的一个重要组成部分，它提供了一系列接口来与操作系统进行交互，包括文件和目录操作、进程管理、执行外部命令、处理环境变量、获取系统信息等功能。

特性

- 文件操作：Open、Create、ReadFile、WriteFile等函数用于打开、创建、读取、写入文件。
- 目录操作：Mkdir、Chdir、Remove、RemoveAll等函数用于创建、切换、删除目录或文件。
- 环境变量：Getenv、Setenv、Environ等函数用于获取、设置、列出环境变量。
- 执行系统命令：虽然直接在os包中没有，但在os/exec包中可以使用Cmd结构体来执行外部命令。
- 信号处理：虽然信号处理主要在os/signal包中，但基础信号常量如SIGINT、SIGKILL等定义在os包中。

基本操作

```
package main

import (
    "fmt"
    "log"
    "os"
)

func main() {
    // 1. 获取当前目录
    fmt.Println(os.Getwd())

    // 2. 修改当前目录
    os.Chdir("/Users/wanzi/tmp")
    fmt.Println(os.Getwd())

    // 3. 创建文件夹
    err := os.Mkdir("go_test", 0777)
    if err != nil && !os.IsExist(err) {
        log.Fatal(err)
    }
    //files, err := os.ReadDir(".")
    //if err != nil {
    //  fmt.Println("无法读取目录, %v", err)
    //}
    //for _, file := range files {
    //  fmt.Println(file.Name())
    //}

    // 4. 修改文件夹或者文件名称
    os.Rename("go_test", "new_go_test")

    // 5. 新建文件，写入内容
    os.Create("./new_go_test/file.txt")
    err = os.WriteFile("./new_go_test/file.txt", []byte("Hello, Gophers!"),
0660)
    if err != nil {
        log.Fatal(err)
    }

    // 6. 读取文件内容
    content, err := os.ReadFile("./new_go_test/file.txt")
    if err != nil {
        log.Fatalf("读取文件错误: %v", err)
    }
    fmt.Printf("读取文件内容为: %s", string(content))

    // 7. 删除文件夹或者文件
    os.RemoveAll("new_go_test")
}
```

注意： 当我们读取或者写入文件的时候，根据不同场景选择不同函数。

os.WriteFile是一个便捷函数，用于直接将数据写入到一个文件中。它内部封装了打开、写入和关闭文件的操作，使得文件写入变得简单直接。

另外，如果文件已存在，WriteFile会覆盖原有内容，如果不存在就会新建文件，使用完毕后回自动关闭文件，不需要手动调用close.

```
func WriteFile(filename string, data []byte, perm os.FileMode) error
```

- filename: 要写入的文件名。
- data: 要写入文件的数据，类型为字节切片。
- perm: 文件的权限模式，如0644表示文件所有者可读写，同组用户和其他用户可读。

os.OpenFile是一个更为通用的函数，它提供了更多的灵活性，可以用来以多种模式（读、写、追加等）打开文件。

基本用法:

```
func OpenFile(name string, flag int, perm os.FileMode) (*os.File, error)
```

- name: 要打开的文件名。
- flag: 文件打开模式，如os.O_RDONLY（只读）、os.O_WRONLY（只写）、os.O_RDWR（读写）、os.O_CREATE（如果不存在则创建）、os.O_APPEND（追加模式）等，可以通过按位或|组合多个标志。
- perm: 同WriteFile中的权限模式。

os/exec

os/exec 是Go语言的标准库之一，用于在程序中启动外部命令或程序，并与它们进行交互。这个包提供了一个高级接口来执行系统命令，相比于直接使用 os.StartProcess，它更加方便，因为它允许你更容易地重定向标准输入、输出和错误流，并且能够处理命令行参数和环境变量。

os/exec包的核心是Cmd结构体，它代表一个准备执行的命令。以下是 Cmd 结构体的一些重要字段：

- Path string: 将要执行的命令的路径。必须非空，相对路径会相对于 Dir 字段指定的目录解析。
- Args []string: 命令的参数列表，包括命令名作为第一个元素。如果为空，则表示没有参数。
- Env []string: 指定进程的环境变量，如果为 nil，则使用当前进程的环境变量。
- Dir string: 指定命令的工作目录，如果为空字符串，则在调用者进程的当前工作目录下执行。

使用os/exec执行外部命令ls，并打印其输出：

```go
package main

import (
    "fmt"
    "log"
    "os/exec"
)

func main() {
    cmd := exec.Command("ls", "-l", "libs") // 创建一个cmd的实例来执行ls -l命令

    //  这里CombinedOutput获取命令的标准输出和标准错误输出，都以字符串返回，如果只是标准输出可以使用Output
    output, err := cmd.CombinedOutput()
    if err != nil {
        log.Fatalf("cmd.Run() failed with %s \n", err)
    }
    fmt.Printf("Output: %s \n", output)
}
```

使用os/exe重定向输入输出

你可以通过管道 (io.Pipe) 来重定向命令的标准输入、输出和错误流，实现更复杂的交互。

例如，将一个命令的输出作为另一个命令的输入：

```go
package main

import (
    "bytes"
    "fmt"
    "io"
    "log"
    "os/exec"
)

func main() {
    cmd1 := exec.Command("echo", "Hello, Golang!")
    cmd2 := exec.Command("wc")

    // 创建管道用于两个命令见传递数据
    reader, writer := io.Pipe()

    // 将cmd1的标准输出，连接到pipe的写端
    cmd1.Stdout = writer
    // 将cmd2的标准输入，连接到pipe的读端
    cmd2.Stdin = reader

    // 执行第一个命令
    go func() {
        defer writer.Close()
        if err := cmd1.Run(); err != nil {
            log.Fatal(err)
        }
    }()

    // 读取cmd2的输出
    var buf bytes.Buffer
    cmd2.Stdout = &buf
    if err := cmd2.Run(); err != nil {
        log.Fatal(err)
    }
    fmt.Printf("output of wc: %s", buf.String()) // output of wc:       1
2       15
}
```

在这个例子中，echo "Hello World" 的输出通过一个管道传递给了 wc 命令作为输入，最后打印出 wc 的输出结果，即统计了字符串的行数、单词数和字节数。

strings

strings是Go语言的标准库之一，提供了对字符串操作的一系列功能，包括搜索、替换、比较、分割等。这个包非常实用，几乎在任何涉及文本处理的Go程序中都会用到。

主要函数和方法：

- 字符串比较：如 Compare 和 EqualFold，用于比较字符串是否相等，后者支持大小写不敏感比较。
- 查找：如 Contains, ContainsAny, ContainsRune, Index, IndexAny, IndexRune 等，用于查找子串或字符在字符串中的位置。
- 替换：如 Replace，用于替换字符串中的子串。
- 分割与连接：如 Split, Join，分别用于分割字符串和连接字符串数组为一个字符串。
- 前缀与后缀：如 HasPrefix, HasSuffix, TrimPrefix, TrimSuffix，用于检测和移除字符串的前缀或后缀。
- 大小写转换：如 ToLower, ToUpper，用于转换字符串的大小写。
- 其他：还有很多其他有用的功能，比如 Count 计算某个字符或子串出现的次数，Repeat 重复字符串等。

案例分析

查找并替换字符串：

```
Text := "Hello, Gophers! Welcome to the world of Go."
replaceText := strings.Replace(Text, "Gophers", "Developers", -1)
fmt.Println(replaceText)
```

分隔字符串：

```
fruits := "Apple, Banana, Orange, Pear, Watermelon, Cherry, Grape, Strawberry"
splitList := strings.Split(fruits, ",")
fmt.Println(splitList)
```

检测前缀：

```
url := "https://wnote.com/notes/golang/"
if strings.HasPrefix(url, "https://") { // 检测前缀，检测后缀使用strings.HasSuffix
    fmt.Println("This URL uses HTTPS")
} else {
    fmt.Println("This URL does not use HTTPS.")
}
```

大小写转换:

```
text := "GO IS AWESOME!"
text1 := "go language learning manual "
lowerText := strings.ToLower(text)
upperText := strings.ToUpper(text1)
fmt.Println("Lower:", lowerText)
fmt.Println("Upper:", upperText)
```

剔除字符串里两端空白字符：

这里的空白字符包括空格、制表符（\t）、换行符（\n）、垂直制表符（\v）、跳格符（\f）和换行符（\r）。

```
s := "  \t\n\v\f\rHello, world!  "
s = strings.TrimSpace(s)
fmt.Println(s) // 输出 "Hello, world!"
```

log介绍

log是Go语言的标准库之一，提供了简单的日志记录功能。它被设计用于在程序运行过程中输出日志信息，帮助开发者进行调试、错误追踪和状态记录。

log包的核心是一个预定义的标准日志器（std），以及一些便捷的函数，如Print, Printf, Println, Fatal, Fatalf, 和 Panic等，这些函数可以直接使用而无需创建log.Logger实例。

主要功能：

- 标准日志器: log 包提供了一个默认的全局日志器 log.Logger 实例 std，它默认将日志输出到标准错误流（os.Stderr）。
- 格式化: 日志条目可以是无格式的（使用 Print 系列函数）或格式化的（使用 Printf 系列函数），类似于 fmt.Print 和 fmt.Printf。
- 日志级别: 虽然 log 包本身不直接支持日志级别，但可以通过条件语句手动控制不同级别的日志输出。
- 输出重定向: 可以通过 log.SetOutput 函数改变日志的输出目的地，例如将其导向文件。
- 文件名和行号: log 支持在日志条目中包含调用日志函数的文件名和行号，通过 SetFlags 函数设置。

log 函数介绍：

```
func Println(v ...interface{}): 输出日志，自动添加换行符。
func Printf(format string, v ...interface{}): 按照指定的格式输出日志。
func Print(v ...interface{}): 输出日志，不添加换行符。
func Fatal(v ...interface{}): 输出日志后，调用os.Exit(1)终止程序。
func Fatalf(format string, v ...interface{}): 格式化输出日志后，终止程序。
func Fatalln(v ...interface{}): 输出日志并换行，然后终止程序。
func Panic(v ...interface{}): 输出日志后，触发一个panic。
func Panicf(format string, v ...interface{}): 格式化输出日志后，触发一个panic。
func Panicln(v ...interface{}): 输出日志并换行，然后触发一个panic。
```

log基本设置

1、设置日志前缀

```
log.SetPrefix("前缀: ")
log.Println("这是一条带前缀的日志。") // 输出结果: 前缀: 2024/05/28 10:55:53 这是一条带前缀的日志。
```

2、设置日志标志

log包提供了Flags和SetFlags函数来设置日志的标志，可以通过组合不同的标志来控制日志输出中是否包含时间、日期、文件名、行号等信息

log.Flags()是用来获取当前的日志标志设置，而log.SetFlags()是用来设置或改变日志的输出标志。

- Ldate: 输出本地时间的日期，如 2009/01/23。
- Ltime: 输出本地时间的时间，如 01:23:23。
- Lmicroseconds: 微秒级时间精度，需配合 Ltime 使用。
- Llongfile 或 Lshortfile: 输出调用日志函数的文件名和行号，长格式或短格式。
- Lshortfile：显示文件名和行号，但不包含完整路径。

```
// 设置日志标志，包括日期和时间
log.SetFlags(log.Ldate | log.Ltime | log.Lshortfile)
log.Println("这是一条带时间和文件信息的日志。") // 输出结果：前缀：2024/05/28 10:55:53
log1.go:12: 这是一条带时间和文件信息的日志。
```

3、设置日志输出路径

```
package main

import (
    "log"
    "os"
)

func main() {
    // 创建或打开一个日志文件
    file, err := os.OpenFile("app.log", os.O_CREATE|os.O_WRONLY|os.O_APPEND,
0666)
    if err != nil {
        log.Fatalf("Failed to open log file: %v", err)
    }
    defer file.Close()

    // 设置日志输出到文件
    log.SetOutput(file)

    log.Println("This message will be logged into app.log.")
}
```

注意事项

1、使用log.Fatal或log.Panic 会导致程序终止执行，前者直接退出，后者引发 panic。

2、谨慎处理日志输出的性能影响，尤其是在高并发或大量日志产出的场景下。

3、log默认不支持配置一些日志特性，比如日志级别、结构化日志等，但是Go社区有许多成熟的第三方日志库可供我们选择，如Zap、Logrus、zerolog等，它们内置了丰富的日志级别控制功能，以及格式化、性能优化等特性。

sort介绍

在Go语言中，sort包提供了对slices进行排序的功能，支持升序和降序排列，以及自定义排序规则。

- 升序排序：使用sort.Slice() 或sort.Ints()（针对整型切片）等函数直接对切片进行排序。
- 降序排序：可以通过提供自定义的比较函数来实现降序排序。
- 自定义排序：当切片元素类型不是基础类型或者需要基于非自然顺序排序时，可以提供自定义的比较函数。

整型切片升序排序

```
numbers := []int{1, 100, 300, 5, 3}
sort.Ints(numbers) // 升序排序整形切片
fmt.Println("Sorted numbers: ", numbers) // 输出结果: Sorted numbers:  [1 3 5 100 300]
```

字符串切片按字母顺序排序

```
fruits := []string{"apple", "orange", "cherry", "watermelon", "strawberry", "pear", "cherry"}
sort.Strings(fruits) // 按字母升序排序字符串切片
fmt.Println("Sorted fruits: ", fruits) // 输出结果: Sorted fruits:  [apple cherry cherry orange pear strawberry watermelon]
```

自定义排序

```go
package main

import (
    "fmt"
    "sort"
)

// 定义一个结构体
type Person struct {
    Name string
    Age  int
}

// 定义一个切片类型
type PersonSlice []Person

// 实现了interface的Len方法，返回切片长度
func (p PersonSlice) Len() int {
    return len(p)
}

// 实现interface的Less 方法，定义了排序规则，按照Age字段升序排序。
func (p PersonSlice) Less(i, j int) bool {
    return p[i].Age < p[j].Age
}

// 实现Interface的Swap 方法，交换切片中的两个元素位置
func (p PersonSlice) Swap(i, j int) {
    p[i], p[j] = p[j], p[i]
}

func main() {
    people := PersonSlice{
        {"Zhangsan", 30},
        {"Lisi", 20},
        {"Wangwu", 35},
    }

    // 进行排序，其实这里PersonSlice就是一个排序函数
    sort.Sort(people) // 或者sort.Sort(PersonSlice(people))

    for _, p := range people {
        fmt.Printf("Name: %s, Age: %d\n", p.Name, p.Age)
    }
}
```

通过定义类型（如PersonSlice）并实现这些方法，你实际上就是在为你的类型定义了一个sort.Interface的实现。这样，sort.Sort函数就能识别并使用这些方法来对切片进行排序。

bytes介绍

Go语言中的bytes包提供了一系列操作字节切片([]byte)的函数和类型，这对于处理二进制数据、文本编码转换、I/O操作等场景非常有用。

bytes.Buffer

用途: 类似于一个可读写的字节流，支持动态扩容。常用于构建或累积字节序列，然后转换为字符串或写入到IO中。

方法:

- Write: 将数据写入缓冲区，自动扩容。
- Read: 从缓冲区读取数据。
- String(): 将缓冲区内容转换为字符串。
- Bytes(): 返回缓冲区的字节切片副本。
- Reset(): 清空缓冲区，重置状态。

案例1: 创建并写入数据

```
package main

import (
    "bytes"
    "fmt"
)

func main() {
    var buf bytes.Buffer
    buf.WriteString("Hello, ")
    buf.WriteString("World")
    buf.WriteByte('!')
    fmt.Println(buf.String()) // 输出 Hello, World!
}
```

案例2：读取数据

```go
package main

import (
    "bytes"
    "fmt"
)

func main() {
    var buf = bytes.NewBufferString("Golang bytes buffer")
    data := make([]byte, 5)
    n, _ := buf.Read(data)
    fmt.Println(string(data[:n])) // 输出: Golan
}
```

bytes.Reader

用途: 提供了对固定字节切片的只读访问，支持Seek等操作，模拟文件读取行为。 方法: Read: 读取数据到给定的缓冲区。 Seek: 改变Reader的偏移量，进行随机访问。

案例1: 从字节切片创建Reader

```go
package main

import (
    "bytes"
    "fmt"
)

func main() {
    data := []byte("Using bytes Reader")
    reader := bytes.NewReader(data)

    p := make([]byte, 5)
    reader.Read(p)
    fmt.Println(string(p)) /// 输出Using
}
```

案例2: 读取指定长度数据

```go
package main

import (
    "bytes"
    "fmt"
)

func main() {
    reader := bytes.NewReader([]byte("Read Example"))

    buf := make([]byte, 3)
    n, _ := reader.Read(buf)
    fmt.Println(string(buf[:n])) // 输出Rea
    fmt.Println(reader.Len())    // 打印剩余可读字节数
}
```

案例3: 调过部分数据然后读取

```go
package main

import (
    "bytes"
    "fmt"
)

func main() {
    reader := bytes.NewReader([]byte("Skip this part"))

    // 跳过前5个字节
    reader.Seek(5, io.SeekStart)

    data := make([]byte, 4)
    reader.Read(data)
    fmt.Println(string(data)) // 输出: this
}
```

请注意，最后一个bytes.Reader的例子中使用了Seek方法来改变读取位置，这是Reader接口不具备的，而是io.Seeker接口提供的功能，bytes.Reader恰好实现了这个接口。

其他函数

bytes.Compare: 比较两个字节切片是否相等。

```go
package main

import (
    "bytes"
    "fmt"
)

func main() {
    slice1 := []byte("GoLang")
    slice2 := []byte("GoLang")
    slice3 := []byte("golang")

    if bytes.Equal(slice1, slice2) {
        fmt.Println("Slices are equal") // 输出: Slices are equal
    }

    if !bytes.Equal(slice1, slice3) {
        fmt.Println("Slices are not equal") // 输出: Slices are not equal
    }
}
```

Contains: 检查一个字节切片是否包含另一个字节切片。

```go
package main

import (
    "bytes"
    "fmt"
)

func main() {
    mainSlice := []byte("Hello, world!")
    subSlice := []byte("world")

    if bytes.Contains(mainSlice, subSlice) {
        fmt.Println("Substring found") // 输出: Substring found
    } else {
        fmt.Println("Substring not found")
    }
}
```

Equal: 判断两个字节切片是否完全相等。

```go
package main

import (
    "bytes"
    "fmt"
)

func main() {
    slice1 := []byte("GoLang")
    slice2 := []byte("GoLang")
    slice3 := []byte("golang")

    if bytes.Equal(slice1, slice2) {
        fmt.Println("Slices are equal") // 输出: Slices are equal
    }

    if !bytes.Equal(slice1, slice3) {
        fmt.Println("Slices are not equal") // 输出: Slices are not equal
    }
}
```

Trim, TrimLeft, TrimRight: 去除字节切片前后的特定字节。

```go
package main

import (
    "bytes"
    "fmt"
)

func main() {
    slice := []byte(" **Hello, World!** ")

    trimmed := bytes.Trim(slice, " *")
    fmt.Println(trimmed) // 输出: Hello, World!

    trimmedLeft := bytes.TrimLeft(slice, " *")
    fmt.Println(trimmedLeft) // 输出: Hello, World!**

    trimmedRight := bytes.TrimRight(slice, " *")
    fmt.Println(trimmedRight) // 输出:  **Hello, World!
}
```

Split, SplitAfter, SplitN: 切分字节切片为子切片。

```go
package main

import (
    "bytes"
    "fmt"
)

func main() {
    sentence := []byte("Go,Lang,Is,Fun")

    // Split
    parts := bytes.Split(sentence, []byte(","))
    fmt.Println(parts) // 输出: [[71 111] [44] [76 97 110 103] [44] [73 115] [44] [70 117 110]]

    // SplitAfter
    partsAfter := bytes.SplitAfter(sentence, []byte(","))
    fmt.Println(partsAfter) // 输出包含分隔符: [[71 111 44] [76 97 110 103 44] [73 115 44] [70 117 110]]

    // SplitN
    partsN := bytes.SplitN(sentence, []byte(","), 2)
    fmt.Println(partsN) // 输出限制数量的切片: [[71 111] [44 76 97 110 103 44 73 115 44 70 117 110]]
}
```

flag

strconv

regexp

net/http

sync

math

context

archive

compress

crypto

Go第三方库

viper

cobra

cast

gin介绍

介绍安装

流程原理解析

路由与传参数

响应返回

路由分发

中间件

Gorm实践

ORM是什么?

什么是GORM?

Gorm基本使用

模型定义

一对多关

多对多关

项目开发实践

图书管理系统

IP在线查询网站实现

k8s在线管理平台