官网：

https://golang.org/

环境：

$GOROOT：

GOROOT环境变量指定了Go的安装目录。

$GOPATH：

GOPATH 环境变量指定workspace的目录。

命令行：

go：

env:

描述：

Env prints Go environment information.

例子：

go env

get:

描述：

Get downloads the packages named by the import paths, along with their dependencies. It then installs the named packages, like 'go install'.

用法：

go get [-d] [-f] [-fix] [-insecure] [-t] [-u] [build flags] [packages]

list:

描述：

List lists the packages named by the import paths, one per line.

用法：

go list [-e] [-f format] [-json] [build flags] [packages]

例子：

go list all

go list std

clean：

描述:

Clean removes object files from package source directories. The go command builds most objects in a temporary directory, so go clean is mainly concerned with object files left by other tools or by manual invocations of go build.

build：

描述：

Build compiles the packages named by the import paths, along with their dependencies, but it does not install the results.

install:

描述：

Install compiles and installs the packages named by the import paths, along with their dependencies.

run：

描述：

Run compiles and runs the main package comprising the named Go source files.

用法：

go run [build flags] [-exec xprog] gofiles... [arguments...]

例子：

go run test.go

godoc：

godoc -http=:6060

golint:

install:

go get github.com/golang/lint/golint

usage:

golint <import_path>

delve:

描述：

a debugger for Go, written in Go

install:

go get github.com/derekparker/delve/cmd/dlv

usage:

dlv -h

Go版本管理：

install：

bash < <(curl -s -S -L https://raw.githubusercontent.com/moovweb/gvm/master/binscripts/gvm-installer)

usage：

gvm install go1.4

gvm use go1.4 [--default]

包管理：

方案一：gvp + gpm

方案二：gom

Go脚本：Go语言版ShaBang脚本

方案一：gonow

install:

go get github.com/kison/gonow

shabang:

#!/usr/bin/env gonow

方案二：gorun

install:

go get github.com/erning/gorun

shabang:

#!/usr/bin/env gorun

交叉编译：

支持：

在Linux和Mac 下，可以生成以下目标格式：x86 ELF、AMD64 ELF、ARM ELF；x86 PE、AMD64 PE。

在Windows下，可以生成以下目标格式：x86 PE、AMD64 PE。

方法：

通过设置$GOOS和$GOARCH两个环境变量来指定交叉编译的目标格式。

$GOOS：darwin、freebsd、linux，windows

$GOARCH：386(32bits x86)、amd64(64bits x86)，arm

语言特性：

使用UTF-8字符串和标识符。

大写开头的标识符是公共的。

小写开头的标识符是私有的。

使用聚合和嵌入而不是继承。

Go没有什么？

没有？表达式。

没有Linq，Stream。

没有泛型。

没有指针运算。

没有类，继承。

没有静态库，动态库。

没有动态加载。

没有函数重载，运算符重载。

语法特点：

后置式类型声明。

一般不需要在行末加分号。

导入却未使用的包编译器会报错。

声明却未使用的变量编译器会报错。

工程结构：

典型地，Go 将所有 Go 代码都存放到单一的 workspace 中。一个 workspace 包含多个版本控制仓库如 Git Repo，即一个 workspace 包含多个 go 工程。每个版本控制仓库也即工程包含一个或多个包。一个包包含一个或多个 Go 源文件。包的路径决定了它的 import 路径。

Workspace：

一个 workspace 就是一个目录的层次结构，并且在顶层目录中包含src、pkg以及bin三个子目录。src，包含多个版本控制仓库即工程；pkg，包含编译后的 package 对象；bin，包含编译后的可执行文件。

Import Path：

一个 import path 是一个唯一标识一个包的字符串。一个包的 import path 由这个包在 workspace 中的路径决定，或者与这个包所在的远程仓库有关。如果代码是使用代码仓库如Github管理，那么 Go 建议使用仓库名作为包的基路径。例如 github.com/xyz 是包的基路径，如果需要创建一个名为 abc 的工程，那么它的包路径就是 github.com/xyz/abc。

包名：

根据 Go 规定，包名是一个包的 import path 的最后一个元素，即如果 import path 是 "com/xyz

/abc"，则包名必须是 abc，而com/xyz则是abc包在workspace中的路径。

程序组织：

包：被用来组织代码。

规则：

Go语言中包导入的搜索路径是首先搜$GOROOT，然后搜$GOPATH。按照惯例，包名使用小写字母，包的源代码应该放在一个同名的文件夹下面，允许将同一个包的代码分散在多个单独的源代码文件中。

创建:

```
package <pkgName>

func init() {
}
```

导入:

正常导入：

```
import "<importPath>"
```

别名导入：

```
import <newName> "<importPath>"
```

匿名导入：

```
import _  "<importPath>"
```

本地导入：

```
import . "<importPath>"
```

分组导入：

```
import (
    "<importPath>"
    _ "<importPath>"
    . "<importPath>"
)
```

平台特定代码：

方式一：

```
if runtime.GOOS == "windows" {
}
```

方式二：

******_linux.go

******_darwin.go

******_freebsd.go

******_windows.go

使用：

<pkgName>.<VarName>

<pkgName>.<FuncName>(...)

入口包：可程序程序。

```
package main
func main(){
}
```

编码注释：

```
//*************************
/*                              */
```

类型系统：

说明：

Go语言使用内置的基础类型如bool、int和string等类型来表示数据，或者使用struct来对基本类型进行聚合。Go语言的自定义类型建立在基本类型，struct或者其他自定义类型之上。
Go语言同时支持命名和匿名的自定义类型，相同结构的匿名类型等价，可以相互替换，但是不能有任何方法。任何命名的自定义类型都可以有方法，并且这些方法一起构成该类型的接口，不同的命名自定义类型即使结构完全相同，也不能相互替换，除非特别声明之外。

接口也是一种类型，可以通过指定一组方法的方式定义，接口是抽象的，因此不可以实例化。如果某个具体类型实现了某个接口的所有方法，那么这个类型就被认为实现了该接口，也就是说这个具体类型的值既可以当作该接口类型的值来使用，也可以当作该具体类型的值来使用。
空接口，即没有定义方法的接口，用interface{}来表示，它可以用来表示任意值，无论这个值是一个内置类型的值还是自定义类型的值。
make函数返回的是引用，引用的行为和指针非常相似，当把它们传入函数的时候，函数内对该引用所做的任何改变都会作用到该引用所指向的原始数据上，但引用不需要被解引用，因此大部分情况下不需要将其与*一起使用。如果我们要在一个函数或方法内部使用append修改一个slice而不是修改该slice其中的一个元素内容，必须要么传入指向这个slice的指针，要么就返回该slice，因为有时候append返回的slice与调用所传入的不同。

零值：因为没有构造函数。

When storage is allocated for a variable, either through a declaration or a call of new, or when a new value is created, either through a composite literal or a call of make, and no explicit initialization is provided, the variable or value is given a default value. Each element of such a variable or value is set to the zero value for its type: false for booleans, 0 for integers, 0.0 for floats, "" for strings, and nil for pointers, functions, interfaces, slices, channels, and maps. This initialization is done recursively, so for instance each element of an array of structs will have its fields zeroed if no value is specified.

变量定义：

定义：

以零值定义：

var <varName> <varType>

var <varName1>, <varName2> <varType>

以初值定义：

var <varName> <varType> = <value>

var <varName1>, <varName2> <varType> = <value1> <value2>

类型推演：

var <varName> = <value>

var <varName1>, <varName2> = <value1>, <value2>

简短定义：只能用在函数内部。

<varName> := <value>

<varName1>, <varName2> := <value1>, <value2>

分组定义：

var （

<varName1> = <value>

<varName2> <varType>

<varName3> <varType> = <value>

）

匿名变量：_

_是个特殊的变量名，任何赋予它的值都会被丢弃。

常量定义：

说明：

常量在编译时被创建，只能是数字、字符串或者布尔值。

定义：

正常定义：

const <constName> = <value>

const <constName> <constType> = <value>

分组定义：

const （

<constName1> = <value1>

<constName2> <constType> = <value2>

}

预定义常量：

nil：

值，表示空指针以及空引用。

true，false

类型转换：

说明：

Go语言提供了一种在不同但相互兼容的类型之间相互转换的方式，非数值类型之间的转换不会丢失精度，数值类型之间的转换可能会发生丢失精度或其他问题。

对于数字，本质上可以将任意的整型或浮点型数据转换成别的整型或者浮点型，但转换可能会丢失精度。

一个字符串可以转换成一个[]byte（其底层为UTF-8的字节）或一个[]rune（它的Unicode码点），并且[]byte和[]rune都可以转换成一个字符串类型。单个字符是一个rune类型数据（即int32），可以转换成一个单字符的字符串。

语法：

<DstType>(<varOfSrcType>)

基本类型：

bool：

true，false

Numbers：

整数类型：

有符号：int，int8，int16，int32，int64

无符号：uint，unit8，uint16，uint32，uint64，uintptr

byte是uint8的别称，rune是int32的别称。

byte or rune：'*'

浮点类型：

float32，float64（默认）

复数类型：

complex64，complex128（默认）

形式为RE + IMi，RE为实数部分，IM为虚数部分

string：

单行形式：使用\转移。

"*******"

多行形式：Raw字符串，不含转义，原样输出。

`***********

**********`

操作：

<stringa> + <stringb>

error:

type error interface {

Error() string

}

itoa枚举：

// A Weekday specifies a day of the week (Sunday = 0, ...).

type Weekday int

const (

Sunday Weekday = iota
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday

)

array：值类型，静态数组，长度固定

表示：

[<num>]<varType>

创建：

a := [3]int{1, 2, 3}

a := [3]int{1, 2}

a := [...]int{1, 2, 3}

使用：

<varName>[index]

<varName>[index] = <value>

slice：引用类型，动态数组

表示：

[]<varType>

创建：

By literal：

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

From Array：

s := a[i:j]，

i为开始位置，默认为零

a[:j] == a[0:j]

j为结束位置，默认为数组长度

a[:]返回整个数组

By make：

s := make([]int, 3)

操作：

get：

s[0]

assign：

s[0] = 13

len：取长度

len(<varName>)

cap：取最大容量

cap(<varName>)

append：追加一个或多个元素

append(<varName>,<val1>,<val2>)

copy:

copy(<destVarName>, <srcVarName>)

map：引用类型

说明：

map的键必须是支持==和!=操作符的类型，大部分Go语言的基本类型都可以作为映射的键，例如，int，float，rune，string，可比较的数组和结构体，基于这些类型的自定义类型，以及指针。Go语言的切片和不能用于比较的数组和结构体（成员或字段不支持==和!=操作符）或者基于这些的自定义类型不能作为键。

表示：

map[<keyType>]<valueType>

创建：

By literal：

<varNameOfMap> = map[<keyType>]<valueType>{<key>:<value>}

By make

<varNameOfMap> = make(map[<keyType>]<valueType>)

操作：

get：

<valueOfKey> = <varNameOfMap>[<key>]

add or update：

<varNameOfMap>[<key>] = <valueOfKey>

check exists：

_, ok = <varNameOfMap>[<key>]

delete：

delete(<varNameOfMap>, <key>)

function：引用类型

说明：

函数实例也是值，可以作为变量、参数或返回值的值。

函数类型也是类型，可以做为变量、参数或返回值的类型。

定义：

基本定义：

func <funcName>(<argsDefine>) <returnDefine>{

.....

}

匿名定义：

func (<argsDefine>) <returnDefine>{

.....

}

参数定义：

方式一：

<arg1> <varType1>, <arg2> <argType2>

方式二：所有参数类型相同

<arg1>, <arg2> <argType>

方式三：可变参数，slice

定义：

<argn> ... <argType>

调用：

<func_name>(<fixed_args>, <elem>, <elem>)

<func_name>(<fixed_args>, <slice>...)

返回定义：

方式一：无返回值

.....{

}

方式二：单返回值

.....<returnType>{

.....

return <value>

}

方式三：多返回值

.....(<returnType1> <returnType2>){

.....

return <value1>, <value2>

}

方式四：多返回值之命名的返回变量

.....(<varReturn1> <returnType1>, <varReturn2> <returnType2>){

.....

<varReturn1> = <value1>

<varReturn1> = <value1>

return

}

.....(<varReturn1>, <varReturn2> <returnType>){

.....

<varReturn1> = <value1>

<varReturn1> = <value1>

return

}

预定义函数：

complex：

func complex(r, i FloatType) ComplexType

real：

func real(c ComplexType) FloatType

imag：

func imag(c ComplexType) FloatType

new:

func new(Type) *Type

说明：

new生成一个固定大小的内存块并返回一个指针指向它。

表达式 new(T) 分配了一个零初始化的 T 值，并返回指向它的指针。

var t *T = new(T) 或 t := new(T)

make：用来创建slice, map, channel

func make(Type, size IntegerType) Type

说明：

make是生成一个可变大小的内存块，并返回一个它的引用。

slice:

make(T, n) slice of type T with length n and capacity n

make(T, n, m) slice of type T with length n and capacity m

map:

make(T) map of type T

make(T, n) map of type T with initial space for n elements

channel:

make(T) unbuffered channel of type T

make(T, n) channel buffered channel of type T, buffer size n

len：len(s)

func len(v Type) int

说明：

string type string length in bytes

[n]T, *[n]T array length (== n)

[]T slice length

map[K]T map length (number of defined keys)

chan T number of elements queued in channel buffer

cap：cap(s)

func cap(v Type) int

说明：

[n]T, *[n]T array length (== n)

[]T slice capacity

chan T channel buffer capacity

append:

func append(slice []Type, elems ...Type) []Type

copy:

func copy(dst, src []Type) int

delete:

func delete(m map[Type]Type1, key Type)

close:

func close(c chan<- Type)

panic：

func panic(interface{})

recover：

func recover() interface{}

print：

func print(args ...Type)

println：

func println(args ...Type)

复杂类型：

type：创建一个自定义类型，指向原有类型或匿名复合类型或函数类型。

语法：

```
type typeName typeSpecification
type (
    typeName typeSpecification
    typeName typeSpecification
)
```

说明：

typeSpecification可以是任何内置的类型（如Int、String、Slice、Map或者Channel）、一个接口、一个结构体或者一个函数签名。

struct：结构体是字段的集合。

表示：

匿名类型：

```
struct {
    <field1> <fieldType>
    <field2> <fieldType> `<tags>`
    .....
}
```

命名类型：

```
type <typeNameOfStruct> struct {
    <field1> <fieldType>
    <field2> <fieldType> `<tags>`
    .....
}
```

创建：

方式一：按字段定义顺序

```
<varNameOfStruct> = <typeNameOfStruct>{<value1>, <value2>}
```

方式二：使用Name：语法

```
<varNameOfStruct> = <typeNameOfStruct>{<field2>: <value2>}
```

使用：

```
<varNameOfStruct>.<field1>
<varNameOfStruct>.<field1> = <value>
```

聚合和嵌入：

聚合和嵌入结构体：

聚合：（具名的）

定义：

```
type Person struct {
    Titile          string     //具名字段（聚合）
```

```
    Forenames    []string    //具名字段（聚合）
    Surname      string      //具名字段（聚合）
}
type Author struct {
    Names        Person      //具名字段（聚合）
    Title        []string    //具名字段（聚合）
    YearBorn     int         //具名字段（聚合）
}
```

使用：

```
author = Author(Person{"Mr", []string{"Rebort", "Louis"}, "Stevenson"},
    []string{"Kidnapped","Treasure Island"}, 1850)
```

嵌入：（匿名的）

定义：

```
type Person struct {
    Titile       string      //具名字段（聚合）
    Forenames    []string    //具名字段（聚合）
    Surname      string      //具名字段（聚合）
}
type Author struct {
    Person                   //匿名字段（嵌入）
    Title        []string    //具名字段（聚合）
    YearBorn     int         //具名字段（聚合）
}
```

使用：

```
author = Author(Person{"Mr", []string{"Rebort", "Louis"}, "Stevenson"},
    []string{"Kidnapped","Treasure Island"}, 1850)
author.Person.Title   //必须使用类型名一消除歧义
author.Surname
```

说明：

如果一个嵌入结构体类型带有方法，那么就可以在外部结构体中直接调用

它，并且只有嵌入的字段会作为接收者传递给这些方法。

聚合和嵌入接口：

当一个结构体聚合（具名的）或者嵌入（匿名的）接口类型的字段时，这就意味着该结构体可以将任意满足该接口规格的值存贮在该字段中。

method：方法是与类型关联起来的function。

规则：

事实上，可以对包中的任意命名类型定义任意方法，而不仅仅是结构体；不能对来自其他包的类型或基础类型定义方法。
方法可以与命名类型或命名类型的指针关联。有两个原因需要使用指针接收者。首先避免在每个方法调用中拷贝值（如果值类型是大的结构体的话会更有效率）。其次，方法可以修改接收者指向的值。

定义：

Common:

```
func (<argNameOfReceiver> <typeNameOfReceiver>) <nameOfMethod>(<args>) <returnType> {
    .....
}
```

for Struct:

```
func (<argNameOfStructe> *<typeNameOfStruct>) <nameOfMethod>(.....) <....> {
    .....
}
```

for Basic Type：

```
type MyFloat float64
func (f MyFloat) Abs() float64 {
    .....
}
```

方法=》函数：

Method Express:

```
<typeNameOfReceiver>.Method => func (<argNameOfReceiver>, <args>)
```

<returnType>

Method Value:

<instance_of_receiver>.Method => func(<args>) <returnType>

interface：接口类型是由一组方法定义的集合。

说明：

An interface type specifies a method set called its interface. A variable of interface type can store a value of any type with a method set that is any superset of the interface. Such a type is said to implement the interface. The value of an uninitialized variable of interface type is nil.

规则：

类型通过实现接口中定义的方法来实现接口，没有显式声明的必要。隐式接口实现解藕了实现接口的包和定义接口的包：互不依赖。单方法接口命名为方法名加上-er后缀：Reader，Writer，Closer等。根据Go语言的惯例，定义接口时接口的名字需以er结尾。

表示：

匿名类型：

interface {

.....Funcs(Without func keyword) in interface.....

}

命名类型：

type <typeNameOfInterface> interface {

.....Funcs(Without func keyword) in interface.....

}

嵌入：

type <typeNameOfInterface> interface {

.....Other Interfaces.....

}

任意类型：

interface{}

channel：引用类型

说明：

channel 是有类型的管道，可以用 channel 操作符 <- 对其发送或者接收值，“箭头”就是数据流的方向。

默认情况下，在另一端准备好之前，发送和接收都会阻塞。

发送者可以 close 一个 channel 来表示再没有值会被发送了。

接收者可以通过赋值语句的第二参数来测试 channel 是否被关闭：当没有值可以接收并且 channel 已经被关闭，那么第二个参数返回false。

循环 for i := range c 会不断从 channel 接收值，直到它被关闭。

只有发送者才能关闭 channel，而不是接收者。向一个已经关闭的 channel 发送数据会引起 panic。

如果容量大于0，通道就是异步的了，缓冲满载（发送）或变空（接收）之前通信不会阻塞，元素会按照发送的顺序被接收。如果容量是0或者未设置，通信仅在收发双发准备好的情况下才可以成功。

表示：

chan <typeOfContent>　　=>read/write

chan <- <typeOfContent> =>write only

<- chan <typeOfContent> =>read only

创建：

形式一：

<varNameOfChannel> = make(chan <typeOfContent>)

形式二：带缓冲的channel

<varNameOfChannel> = make(chan <typeOfContent>, <sizeOfCache>)

例子：

工厂模式：

不将通道作为参数传递给协程函数，用一个普通函数来生成一个通道并返回，在这个函数内部有一个匿名函数以协程的方式被创建，这个协程会用到创建的通道。

例子：

func pump() chan int {

```
    ints := make(chan int)
    go func() {
        for i := 0; i < 10000; i++ {
            ints <- i
        }
        close(ints)
    }
    return ints
}
```

goroutine：

说明：

goroutine 是由 Go 运行时环境管理的轻量级线程。

启动：开启一个新的 goroutine 执行

go f(x, y, z)

注：f，x，y 和 z 是当前 goroutine 中定义的，但是在新的 goroutine 中运行 f。

规则：

goroutine 在相同的地址空间中运行，因此访问共享内存必须进行同步。 sync 提供了这种可能。

指针：

表示：

*<typeNameOfContent>

创建：

By create：for struct, array, slice, map

<varNameOfPoint> = &<typeNameOfStructOrArrayOrSliceOrMap>{.....}

By convert:

<varNameOfPoint> = &<varNameOfContentType>

By new：表达式 new(T) 分配了一个零初始化的 T 值，并返回指向它的指针。

<varNameOfPoint> = new(<typeNameOfContent>)

使用：

<varNameOfPoint> or (<varNameOfPoint>)

特别：

Struct or array的指针在使用[]或.时，无需使用*对指针取值。

创建Interface的指针是无意义的，也是不被允许的。

逻辑系统：

运算符：

算术：

+ - * / %

+= -= *= /= %=

++ --

位：

& | ^ << >> &^

&= |= ^= <<= >>= &^=

逻辑：

> < >= <= == !=

&& || !

管道：

<-

指针：

取值：*

取地址：&

流程控制：

label && goto：

label:

<nameOfLabel>:

goto:

goto <nameOfLabel>

if / else if / else：

形式一：

```
if <condition> {
.....
}
```

形式二：

```
if <varDefine>; <condition> {
    //use the var defined in if
    .....
}
else {
    //use the var defined in if
    .....
}
```

for / break /continue：

形式一：

```
for <init>; <condition>; <post> {
    .....
}
```

形式二：

```
for <condition> {
    .....
```

}

形式三：无限循环

for{

.....

}

形式三：for array, slice, string, map or reading from a channel

for index, char := range aString {

.....

}

for index := range aString {

.....

}

for index, item := range anArrayOrSlice {

.....

}

for index := range anArrayOrSlice {

.....

}

for key, value := range aMap {

.....

}

for key := range aMap {

.....

}

for item := range aChannel {

.....

}

switch / case / fullthrought / default:

形式一：

switch [varDefine;] <varName>{

case <value1>:

```
        case <value2>:
        default:
    }
```

形式二：

```
    switch {
        case <condiftion1>:
        case <condiftion2>:
        default:
    }
```

形式三: type switch

```
    switch i := x.(type){
        case nil:
        case <type>:
        default:
    }
```

go:

A "go" statement starts the execution of a function call as an independent concurrent thread of control, or goroutine, within the same address space.
GoStmt = "go" Expression .
The expression must be a function or method call; it cannot be parenthesized. Calls of built-in functions are restricted as for expression statements.

The function value and parameters are evaluated as usual in the calling goroutine, but unlike with a regular call, program execution does not wait for the invoked function to complete. Instead, the function begins executing independently in a new goroutine. When the function terminates, its goroutine also terminates. If the function has any return values, they are discarded when the function completes.

select / case /default：

说明：

A "select" statement chooses which of a set of possible send or receive operations will proceed. It looks similar to a "switch" statement but with the cases all referring to communication operations.

select 语句使得一个 goroutine 在多个通讯操作上等待；select 会阻塞，直到条件分支中的某个可以继续执行，这时就会执行那个条件分支。当多个都准备好的时候，会随机选择一个。

例子：

Nomal：

```
select {
    case <var> := <-chan_rev:
    .........
    case chan_sd <- <var>:
    .........
    default:
    .........
}
```

Block forever:

```
select {}
```

Timeout:

```
select {
    case <- time.After(<duration>):
    ..........
}
```

defer：

A "defer" statement invokes a function whose execution is deferred to the moment the surrounding function returns, either because the surrounding function executed a return statement, reached the end of its function body, or because the corresponding goroutine is panicking.

DeferStmt = "defer" Expression .
The expression must be a function or method call; it cannot be parenthesized. Calls of built-in functions are restricted as for expression statements.

Each time a "defer" statement executes, the function value and parameters to the call are evaluated as usual and saved anew but the actual function is not invoked. Instead, deferred functions are invoked immediately before the surrounding function returns, in the reverse order they were deferred. If a deferred function value evaluates to nil, execution panics when the function is invoked, not when the "defer" statement is executed.

For instance, if the deferred function is a function literal and the surrounding function has named result parameters that are in scope within the literal, the deferred function may access and modify the result parameters before they are returned. If the deferred function has any return values, they are discarded when the function completes.
type assertion:

For an expression x of interface type and a type T, the primary expression, x.(T) asserts that x is not nil and that the value stored in x is of type T. The notation x.(T) is called a type assertion.

More precisely, if T is not an interface type, x.(T) asserts that the dynamic type of x is identical to the type T. In this case, T must implement the (interface) type of x; otherwise the type assertion is invalid since it is not possible for x to store a value of type T. If T is an interface type, x.(T) asserts that the dynamic type of x implements the interface T.

If the type assertion holds, the value of the expression is the value stored in x and its type is T. If the type assertion is false, a run-time panic occurs.

type assertion:

For an expression x of interface type and a type T, the primary expression, x.(T) asserts that x is not nil and that the value stored in x is of type T. The notation x.(T) is called a type assertion.

More precisely, if T is not an interface type, x.(T) asserts that the dynamic type of x is identical to the type T. In this case, T must implement the (interface) type of x; otherwise the type assertion is invalid since it is not possible for x to store a value of type T. If T is an interface type, x.(T) asserts that the dynamic type of x implements the interface T.

If the type assertion holds, the value of the expression is the value stored in x and its type is T. If the type assertion is false, a run-time panic occurs.

错误处理：

错误：指可能出错的出错的东西，程序需以优雅的方式处理。惯用法：Return。

error

异常：指不可能发生的事情。惯用法：Throw。

throw=》panic
catch=》recover

并发编程：

并发编程模型：

共享数据型：

Lock，原子操作

消息传递型：

Queue

Go：（Go优先使用消息传递型的并发模型）

go + channel