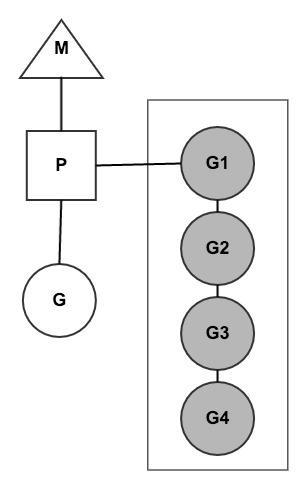
# go并发



### M：是内核线程

### P : 是调度协调，用于协调M和G的执行，内核线程只有拿到了 P才能对goroutine继续调度执行，一般都是通过限定P的个数来控制golang的并发度

### G : 是待执行的goroutine，包含这个goroutine的栈空间

* **Gn : 灰色背景的Gn 是已经挂起的goroutine，它们被添加到了执行队列中，然后需要等待网络IO的goroutine，当P通过 epoll查询到特定的fd的时候，会重新调度起对应的，正在挂起的goroutine。**

Golang为了调度的公平性，在调度器加入了steal working 算法 ，在一个P自己的执行队列，处理完之后，它会先到全局的执行队列中偷G进行处理，如果没有的话，再会到其他P的执行队列中抢G来进行处理。

## 总结

Golang实现了 CSP 并发模型做为并发基础，底层使用goroutine做为并发实体，goroutine非常轻量级可以创建几十万个实体。实体间通过 channel 继续匿名消息传递使之解耦，在语言层面实现了自动调度，这样屏蔽了很多内部细节，对外提供简单的语法关键字，大大简化了并发编程的思维转换和管理线程的复杂性。

作者：falm  
链接：https://www.jianshu.com/p/36e246c6153d  
來源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

# 短声明变量

在函数中， := 简洁赋值语句在明确类型的地方，可以用于替代 var 定义。

函数外的每个语句都必须以关键字开始（ var 、 func 、等等）， := 结构不能使用在函数外。

### 总结:

Go加速：

### http://blog.csdn.net/erlib/article/details/51219512

### new 的作用是初始化一个指向类型的指针(\*T)，make 的作用是为 slice，map 或 chan 初始化并返回引用(T)。

内存分配

Go 同样也垃圾收集，也就是说无须担心内存分配和回收。

Go 有两个内存分配原语，new 和 make。它们应用于不同的类型，做不同的工作，可能有些迷惑人，但是规则很简单。下面的章节展示了在 Go 中如何处理内存分配，并且希望能够让 new 和 make 之间的区别更加清晰。

用 new 分配内存

内建函数 new 本质上说跟其他语言中的同名函数功能一样：new(T) 分配了零值填充的 T 类型的内存空间，并且返回其地址，一个 \*T 类型的值。用 Go 的术语说，它返回了一个指针，指向新分配的类型 T 的零值。记住这点非常重要。

这意味着使用者可以用 new 创建一个数据结构的实例并且可以直接工作。如 bytes.Buffer 的文档所述 “Buffer 的零值是一个准备好了的空缓冲。” 类似的，sync.Mutex 也没有明确的构造函数或 Init 方法。取而代之， sync.Mutex 的零值被定义为非锁定的互斥量。

零值是非常有用的。例如这样的类型定义，57 页的”定义自己的类型” 内容。

SyncedBuffer 的值在分配内存或定义之后立刻就可以使用。在这个片段中，p 和 v 都可以在没有任何更进一步处理的情况下工作。

用 make 分配内存

回到内存分配。内建函数 make(T, args) 与 new(T) 有着不同的功能。它只能创建 slice，map 和 channel，并且返回一个有初始值（非零）的 T 类型，而不是 \*T。本质来讲，导致这三个类型有所不同的原因是指向数据结构的引用在使用前必须被初始化。

例如，一个 slice，是一个包含指向数据（内部 array）的指针，长度和容量的三项描述符；在这些项目被初始化之前，slice 为 nil。对于 slice，map 和 channel，make 初始化了内部的数据结构，填充适当的值。

例如，make([]int, 10, 100) 分配了 100 个整数的数组，然后用长度 10 和容量 100创建了 slice 结构指向数组的前 10 个元素。区别是，new([]int) 返回指向新分配的内存的指针，而零值填充的 slice 结构是指向 nil 的 slice 值。

这个例子展示了 new 和 make 的不同。

务必记得 make 仅适用于 map，slice 和 channel，并且返回的不是指针。应当用 new 获得特定的指针。

new 分配；make 初始化

上面的两段可以简单总结为：

•new(T) 返回 \*T 指向一个零值 T

•make(T) 返回初始化后的 T

当然 make 仅适用于slice，map 和channel。

### 二．避免堆分配可以成为优化的主要方向。通过在栈中分配空间（即多使用A{}的方式创建对象，三.而不是使用new(A)的方式），我们避免了昂贵的malloc调用。

**四.如果可能的话避免使用JSON。如果确实需要用JSON，生成序列化和反序列化代码。一般来说，最好避免依靠反射和interface，而是编写使用的具体类型。对结构体进行排序比对interface进行排序快92％。**

**五．在Channel的底层实现中，使用的还是锁。在没有锁竞争的单线程应用中，它能工作的很好，但是在多线程场景下，性能会急剧下降。我们可以很容易的使用**无锁队列ring buffer**来替代channel的功能。**

**六．方法接收参数是通过值传递的，所以通过接收到的值是一个原值的copy，所以不能通过值获取原指针,如果接收的参数是一个指针，就可以通过指针来获取原值，并且可以改变原值**

**七．Go语言给接口赋值结构体时，如果结构体没有显式实现非指针的方法（struct STruct）但是实现了指针方法（struct \*STruct）,系统会自动生成通过指针获取值，也就是非指针的方法，就可以给接口赋值结构体指针。反过来则不行，比如只显式实现了非指针方法，没有实现指针方法，不会自动生成指针方法。也就不能给接口赋值结构体变量。**

### new返回指针。

内建函数new本质上说跟其它语言中的同名函数功能一样：new(T)分配了零值填充的T类型的内存空间，并且返回其地址，即一个\*T类型的值。用Go的术语说，它返回了一个指针，指向新分配的类型T的零值。

### make返回初始化后的（非零）值。

make只能创建slice、map和channel，并且返回一个有初始值(非零)的T类型，而不是\*T。

导致这三个类型有所不同的原因是指向数据结构的引用在使用前必须被初始化。例如，一个slice，是一个包含指向数据（内部array）的指针、长度和容量的三项描述符；在这些项目被初始化之前，slice为nil。对于slice、map和channel来说，make初始化了内部的数据结构，填充适当的值。

# 常量

常量的定义与变量类似，只不过使用 const 关键字。

常量可以是字符、字符串、布尔或数字类型的值。

常量不能使用 := 语法定义。

不能用&取常量地址

# 指针

Go 具有指针。 指针保存了变量的内存地址。

类型 \*T 是指向类型 T 的值的指针。其零值是 nil 。

var p \*int

& 符号会生成一个指向其作用对象的指针。

i := 42

p = &i

\* 符号表示指针指向的底层的值。

fmt.Println(\*p) // 通过指针 p 读取 i

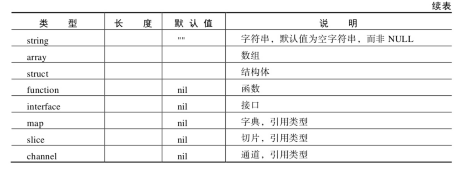
\*p = 21 // 通过指针 p 设置 i

这也就是通常所说的“间接引用”或“非直接引用”。

与 C 不同，Go 没有指针运算。

# 基本类型





### 注意：var a float32= 1. 1234567899 // 注意： 默认 浮点 类型 是 float64

# 引用类型--所谓 引用 类型（ reference type） 特指 slice、 map、 channel 这 三种 预定 义 类型。

## 切片：

package main

import "fmt"

func main() {

a:=[5]string{"a","b","c","d","e"}

b:=a[:1]

fmt.Printf("a--type:%T,%v \n",a,a)

fmt.Printf("b--%v\n",b)

c:=append(b,"f")

fmt.Printf("a--type:%T,%v \n",a,a)

fmt.Printf("b--%v\n",b)

fmt.Printf("c--%v\n",c)

}

输出结果：

a--type:[5]string,[a b c d e]

b--[a]

a--type:[5]string,[a f c d e]

b--[a]

c--[a f]

# 内置 函数 new 按指 定 类型 长度 分配 零值 内存， 返回 指针， 并不 关心 类型 内部 构造 和 初始化 方式。

new 设置空值，返回指针

make 设置不空，返回引用

# 类型转换-表达式类型必须一致

a:= 10

b:= byte( a)

c:= a+ int( b) // 混合 类型 表达式 必须 确保 类型 一致

转换成指针，通道与没有返回值的函数要加括号

(\*int)(p)

(<-chan int)(c)

(func())(x)

有返回值的函数可以省略，但还是建议加上

省略方式： func() int(x)

加上： (func() int)(x)

# 自定义类型--使用关键字type

使用 关键字 type 定义 用户 自定义 类型， 包括 基于 现有 基础 类型 创建， 或者是 结构 体、 函数 类型 等。

# 表达式-类型必须一致（除位移）

除 位移 操作 外， 操作 数 类型 必须 相同。

如果 其中 一个 是 无 显 式 类型 声明 的 常量， 那么 该 常量 操 作数 会 自动 转型。

func main() {

const v= 20 // 无 显 式 类型 声明 的 常量

var a byte= 10

b:= v+ a //v 自动 转换 为 byte/ uint8 类型

fmt. Printf("% T, %v\ n", b, b)

const c float32= 1. 2

d:= c+ v //v 自动 转换 为 float32 类型

fmt. Printf("% T, %v\ n", d, d)

}

二进制 位 运算符 比较 特别 的 就是“ bit clear”， 在 其他 语言 里 很少 见到。

AND 按 位 与： 都为 1 a& b 0101& 0011= 0001

OR 按 位 或： 至少 一个 1 a| b 0101| 0011= 0111

XOR 按 位 亦或： 只有 一个 1 a^ b 0101^ 0011= 0110

NOT 按 位 取 反 (一元) ^ a ^ 0111= 1000

AND NOT 按 位 清除 (bit clear) a&^ b 0110&^ 1011= 0100

LEFT SHIFT 位 左移 a<< 2 0001<< 3= 1000

RIGHT SHIFT 位 右移 a>> 2 1010>> 2= 0010

清除（ AND NOT） 和 位 亦或（ XOR） 是不同 的。 它将 左右 操作 数 对应 二进制 位 都为 1 的 重置 为 0（ 有些 类似 位图）， 以 达到 一次 清除 多个 标记 位 的 目的。

# 指针-支持相等运算，不支持其它加减等

取 址 运算符“&” 用于 获取 对象 地址。

指针 运算符“\*” 用于 间接 引用 目标 对象。

二级 指针\*\* T， 如 包含 包 名 则 写成\* package. T。

指针 运算符 为 左 值 时， 我们 可更新 目标 对象 状态； 而为 右 值 时 则是 为了 获取 目标 状态。

指针 类型 支持 相等 运算符， 但不能 做 加减法 运算 和 类型 转换。 如果 两个 指针 指向 同一 地址， 或 都为 nil， 那么 它们 相等。

可通过 unsafe. Pointer 将 指针 转换 为 uintptr 后进 行 加减法 运算， 但 可能 会 造成 非法 访问。

# 流程控制

## If ... else ...

条件 表达式 值 必须 是 布尔 类型， 可省略 括号， 且 左 花 括号 不能 另起 一行。

表达式前加以加初始化语句，支持， 可定义 块 局部 变量 或 执行 初始化 函数。

if xinit(); x== 0{ // 优先 执行 xinit 函数

println(" a")

}

## Switch

switch x{ // 将 x 与 case 条件 匹配

case a, b: // 多个 匹配 条件 命中 其一 即可（ OR）， 变量

println(" a| b")

case c: // 单个 匹配 条件

println(" c")

if x>= 15{

break // 终止， 不再 执行 后续 语句

}

fallthrough

case 4: // 常量

println(" d")

default:

println(" z")

}

无须 显 式 执行 break 语句， case 执行 完毕 后 自动 中断。

如 须 贯通 后续 case（ 源 码 顺序）， 须 执行 fallthrough， 但不 再 匹配 后续 条件 表达式。

最后一个case不要加fallthrough会报错：// 导致 "cannot fallthrough final case in switch" 错误

不会执行default

fallthrough 必须 放在 case 块 结尾， 可使用 break 语句 阻止。

SWITCH 替换if用法

时候， switch 还 被用 来 替换 if 语句。 被 省略 的 switch 条件 表达式 默认值 为 true， 继而 与 case 比较 表达式 结果 匹配。

func main() {

switch x:= 5; { // 相当于 “switch x:= 5; true{ ... }”

case x> 5:

println(" a")

case x> 0&& x<= 5: // 不能 写成 “case x> 0, x<= 5”， 因为 多 条件 是 OR 关系

println(" b")

default:

println(" z") }

}

输出： b

switch 语句 也可 用于 接口 类型 匹配，

雨痕. Go语言学习笔记 (Kindle 位置 777-783). 电子工业出版社. Kindle 版本.

雨痕. Go语言学习笔记 (Kindle 位置 774-777). 电子工业出版社. Kindle 版本.

# For语句

跟其它类似主要说不一样的

条件不用括号不用说了

for i, c:= 0, count(); i< c; i++ { // 初始化 语句 的 count 函数 仅 执行 一次

println(" a", i)

}

c:= 0

for c< count() { // 条件 表达式 中的 count 重复 执行

println(" b", c)

c++

｝

可用 for... range 完成 数据 迭代， 支持 字符串、 数组、 数组 指针、 切片、 字典、 通道 类型， 返回 索引、 键值 数据。

func main() {

data:= [3] string{" a", "b", "c"}

for i, s:= range data{

println( i, s)

}

}

显示：

0 a

1 b

2 c

# Goto，continue,break语句

使用 goto 前， 须 先 定义 标签。 标签 区分 大 小写， 且 未使 用的 标签 会 引发 编译 错误。

不能 跳 转到 其他 函数， 或 内层 代码 块 内。

break： 用于 switch、 for、 select 语句， 终止 整个 语句 块 执行。

continue： 仅 用于 for 循环， 终止 后续 逻辑， 立即 进入 下一 轮 循环。

配合 标签， break 和 continue 可在 多层 嵌套 中指 定 目标 层级。

outer:

for x:= 0; x< 5; x++ {

for y:= 0; y< 10; y++ {

if y> 2{

println()

continue outer

}

if x> 2{

break outer

}

print( x, ":",y, " ")

}

}

# 函数

无须 前置 声明。

不支持 命名 嵌套 定义（ nested）。

不支持 同名 函数 重载（ overload）。

不支持 默认 参数。

支持 不定 长 变参。

支持 多 返回 值。

支持 命名 返回 值。

支持 匿名 函数 和 闭 包。

左 花 括号 不能 另起 一行。

函数本身可以成为另一个函数的参数。

函数 只能 判断 其 是否 为 nil， 不支持 其他 比较 操作。

func hello() {

println(" hello, world!")

}

func exec( f func()) {

f()

}

func main() {

f:= hello

exec( f)

}

## 建议 命名 规则

### 在 避免 冲突 的 情况下， 函数 命名 要 本着 精简 短小、 望 文 知 意 的 原则。

通常 是 动词 和 介词 加上 名词， 例如 scanWords。

避免 不必 要的 缩写， printError 要比 printErr 更好 一些。

避免 使用 类型 关键字， 比如 buildUserStruct 看上去 会很 别扭。

避免 歧义， 不能 有多 种 用途 的 解释 造成 误解。

避免 只能 通过 大小 写 区分 的 同名 函数。

避免 与 内置 函数 同名， 这 会 导致 误用。

避免 使用 数字， 除非 是 特定 专有 名词， 例如 UTF8。

避免 添加 作用域 提示 前缀。

统一 使用 camel/ pascal case 拼写 风格。

使用 相同 术语， 保持 一致性。

使用 习惯 用语， 比如 init 表示 初始化， is/ has 返回 布尔 值 结果。

使用 反义 词组 命名 行为 相反 的 函数， 比如 get/ set、 min/ max 等。

函数 和 方法 的 命名 规则 稍有 些 不同。 方法 通过 选择 符 调用， 且 具备 状态 上下文， 可使用 更 简短 的 动词 命名。

# 函数的参数

Go 对 参数 的 处理 偏向 保守， 不支持 有 默认值 的 可选 参数， 不支持 命 名实 参。 调用 时， 必须 按 签名 顺序 传递 指定 类型 和 数量 的 实 参， 就算 以“\_” 命名 的 参数 也不能 忽略。

参数 可 视作 函数 局部 变量， 因此 不能 在 相同 层次 定义 同名 变量。

不管 是 指针、 引用 类型， 还是 其他 类型 参数， 都是 值 拷贝拷贝 传递（ pass- by- value）。 区别 无非 是 拷贝 目标 对象， 还是 拷贝 指针 而已。 在 函数 调用 前， 会为 形 参 和 返回 值 分配 内存 空间， 并将 实 参 拷贝 到 形 参 内存。

另外， 并发 编程 也 提倡 尽可能 使用 不可变 对象（ 只读 或 复制）， 这可 消除 数据 同步 等 麻烦。 当然， 如果 复制 成本 很高， 或 需要 修改 原 对象 状态，自然 使用 指针 更好。

# 函数传出参数-也可以说是返回值

方法一，return a

方法二，使用二级指针

例如：

Func test(p\*\*int){

X:=100

\*p=&x

}

Func main(){

Var p\*int

Test(&p)

Println(\*p)

}

输出：100

# 如果 函数 参数 过多， 建议 将其 重 构 为 一个 复合 结构 类型， 也 算是 变相 实现 可选 参数 和 命 名实 参 功能。（类似于封装成一个结构类）

# 可变参数（本质上就是一个切片，只能接收同类型数组，必须放在参数列表末尾）

func test( s string, a... int) {

fmt. Printf("% T, %v\ n", a, a) // 显示 类型 和 值

}

func main() {

test(" abc", 1, 2, 3, 4)

}

输出： []int, [1 2 3 4]

### 将 切片 作为 变参 时， 须 进行 展开 操作。 如果 是 数组， 先 将其 转换 为 切片。 func test( a... int) {

### fmt. Println( a)

### }

### func main() {

### a:= [3] int{ 10, 20, 30}

### test( a[:]...) // 转换 为 slice 后 展开

### }

### 既然 变参 是 切片， 那么 参数 复制 的 仅是 切片 自身， 并不 包括 底层 数组， 也 因此 可 修改 原 数据。 如果 需要， 可用 内置 函数 copy 复制 底层 数据。

### func test( a... int) {

### for i:= range a{

### a[ i] += 100

### }

### }

### func main() {

### a:= []int{ 10, 20, 30}

### test( a...)

### fmt. Println( a)

### }

### 输出： [110 120 130]

# 命名返回值

### 命名 返回 值 和 参数 一样， 可当 作 函数 局部 变量 使用， 最后 由 return 隐式 返回。

### func div( x, y int) (z int, err error) {

### if y== 0{

### err= errors. New(" division by zero")

### Return

### }

### z= x/ y

### return // 相当于 "return z, err"

### }

# 普通 函数 和 匿名 函数 都可 作为 结构 体 字段， 或 经 通道 传递。

## 不曾 使用 的 匿名 函数 会被 编译器 当作 错误。

### func testStruct() {

### type calc struct{ // 定义 结构 体 类型

### mul func( x, y int) int // 函数 类型 字段

### }

### x:= calc{

### mul: func( x, y int) int{

### return x\* y

### },

### }

### println( x. mul( 2, 3))

### }

### func testChannel() {

### c:= make( chan func( int, int) int, 2)

### c<- func( x, y int) int{

### return x+ y

### }

### println((<- c)( 1, 2))

### }

# 闭包

### 闭 包（ closure） 是在 其词 法上 下文 中 引用 了 自由 变量 的 函数， 或者 说是 函数 和 其 引用 的 环境 的 组合 体。

### func test( x int) func() {

### return func() {

### println( x)

### }

### }

### func main() {

### f:= test( 123)

### f()

### }

### 输出： 123

### 就这 段 代码 而言， test 返回 的 匿名 函数 会 引用 上下文 环境 变量 x。 当 该 函数 在 main 中 执行 时， 它 依然 可 正确 读取 x 的 值， 这种 现象 就 称作 闭 包。

## 闭 包 是 函数 和 引用 环境 的 组合 体 更加 确切。

### 正 因为 闭 包 通过 指针 引用 环境 变量， 那么 可能 会 导致 其生 命 周期 延长， 甚至 被 分配 到 堆 内存。 另外， 还有 所谓“ 延迟 求值” 的 特性。

# 延迟调用-defer

### 多个 延迟 注册 按 FILO 次序 执行。（先进后出，写在最上面的最后执行）

### 直到 当前 函数 执行 结束 前 才被 执行， 常用 于 资源 释放、 解除 锁定， 以及 错误 处理 等 操作。

### 注意， 延迟 调用 注册 的 是 调用， 必须 提供 执行 所需 参数（ 哪怕 为 空）。 参 数值 在 注册 时 被 复制 并 缓存 起来。 如对 状态 敏感， 可改 用 指针 或 闭 包。

### defer func( a int) {

### println(" defer x, y= ",a, y) //y 为 闭 包 引用

### }(x) // 注册 时 复制 调用 参数

### 编译器 通过 插入 额外 指令 来 实现 延迟 调用 执行， 而 return 和 panic 语句 都会 终止 当前 函数 流程， 引发 延迟 调用。 另外， return 语句 不是 ret 汇编 指令， 它 会 先 更新 返回 值。

### 相比 直接 用 CALL 汇编 指令 调用 函数， 延迟 调用 则 须 花费 更大 代价。 这 其中 包括 注册、 调用 等 操作， 还有 额外 的 缓存 开销。

# 异常处理

### error是一个接口:(只要实现一个接口的所有方法，就算是实现了该接口)

### type error interface{

### Error() string

### }

### 一般需要自己定义错误类型：

### 某些 时候， 我们 需要 自定义 错误 类型， 以 容纳 更多 上下文 状态 信息。 这样 的 话， 还可 基于 类型 做出 判断。

### type DivError struct{ // 自定义 错误 类型

### x, y int

### }

### func( DivError) Error() string{ // 实现 error 接口 方法

### return" division by zero"

### }

### func div( x, y int) (int, error) {

### if y== 0{

### return 0, DivError{ x, y} // 返回 自定义自定义 错误 类型

### }

### return x/ y, nil

### }

### func main() {

### z, err:= div( 5, 0)

### if err!= nil{

### switch e:= err.( type) { // 根据 类型 匹配

### case DivError:

### fmt. Println( e, e. x, e. y)

### default:

### fmt. Println( e)

### }

### log. Fatalln( err)

### ｝

### Println(z)

### }

### 自定义 错误 类型 通常 以 Error 为名 称 后缀。 在用 switch 按类 型 匹配 时， 注意 case 顺序。 应将 自定义 类型 放在 前面， 优先 匹配 更 具体 的 错误 类型。

## 思路：

## 使用 专门 的 检查 函数 处理 错误 逻辑（ 比如 记录 日志）， 简化 检查 代码。 在 不 影响 逻辑 的 情况下， 使用 defer 延后 处理 错误 状态（ err 退化 赋值）。 在 不中 断 逻辑 的 情况下， 将 错误 作为 内部 状态 保存， 等 最终“ 提交” 时 再处理。

# panic与recover与（try...catch结构更类似）

## 语句。 panic 会 立即 中断 当前 函数 流程， 执行 延迟 调用。 而在 延迟 调用 函数 中， recover 可 捕获 并 返回 panic 提交 的 错误 对象。

import(

"runtime/ debug"

)

func test() {

panic(" i am dead")

}

func main() {

defer func() {

if err:= recover(); err!= nil{

debug. PrintStack()

}

}()

test()

}

### func main() {

### defer func() {

### if err:= recover(); err!= nil{ // 捕获 错误

### log. Fatalln( err)

### }

### }()

### panic(" i am dead") // 引发 错误

### println(" exit.") // 永不 会 执行

### }

### 因为 panic 参数 是 空 接口 类型， 因此 可使用 任何 对象 错误 状态。 而 recover 返回 结果 同样 要做 转型 才能 获得 具体 信息。

### 无论 是否 执行 recover， 所有 延迟 调用 都会 被 执行。 但 中断 性 错 误会 沿 调用 堆栈 向外 传递， 要么 被 外层 捕获， 要么 导致 进程 崩溃。

### 在 延迟 函数 中 panic， 不会 影响 后续 延迟 调用 执行。 而recover 之后 panic， 可被 再次 捕获。 另外， recover 必须 在 延迟 调用 函数 中 执行 才能 正常 工作。

## 建议： 除非 是 不可 恢复 性、 导致 系统 无法 正常 工作 的 错误， 否则 不 建议 使用 panic。 例如： 文件 系统 没有 操作 权限， 服务 端口 被占 用， 数据库 未 启动 等 情况。