Go-18

2021年8月27日 11:00

https://golang.google.cn/ref/spec
Version of Jul 26, 2021

强类型，垃圾回收，多线程，Package。

本文的语法描述使用Extended Backus-Naur Form (EBNF)，扩展巴科斯范式。

```
Production  = production_name "=" [ Expression ] "." .
Expression  = Alternative { "|" Alternative } .
Alternative = Term { Term } .
Term        = production_name | token [ "…" token ] | Group | Option |
Repetition .
Group       = "(" Expression ")" .
Option      = "[" Expression "]" .
Repetition  = "{" Expression "}" .
```

。。上面是 EBNF的定义。

注释：
//
/* */

Token是go的词汇表，有4类：标识符，关键字，运算符和标点符号，字面量。
空格，tab，回车，换行符会被忽略(除非它们分隔了TOken)。
换行符 或文件结果 会插入一个 分号。

; 分号 代码语句的结束。

; 分号，go中可以在某些地方省略;。
当输入被分为Token时，如果这行的最后一个Token是下面之一，那就在行末尾自动添加一个分号
一个标识符
一个整数，浮点数，虚数，rune，字符串字面量
关键字break,continue,fallthrough,return
++ --)] }
为了把复杂语句放在一行中，) 或 } 前的分号可以省略。（。。估计是说，coder可以省略，不过go会自动帮你添加，不过 省略仅仅一个;，就可以让 复杂语句在一行？）

标识符
letter开头，后续可以是letter或digit。（百度 unicode letter， 是有这种定义的）

关键字：

break	default	func	interface	select
case	defer	go	map	struct
chan	else	goto	package	switch
const	fallthrough	if	range	type
continue	for	import	return	var

操作符和标点符号

+	&	+=	&=	&&	==	!=	(	)
-	|	-=	|=	||	<	<=	[	]
*	^	*=	^=	<-	>	>=	{	}
/	<<	/=	<<=	++	=	:=	,	;
%	>>	%=	>>=	--	!	...	.	:
	&^		&^=					

整数字面量

0b or 0B for binary, 0, 0o, or 00 for octal, and 0x or 0X for hexadecimal

基数描述 和 值之间可以 放一个下划线。
值中也可以放下划线。。 值中，不能是开头，不能是结尾。放开头就是 标识符，结尾是非法的。
不能连续2个_。 基数描述中不能_。

```
int_lit        = decimal_lit | binary_lit | octal_lit | hex_lit .
decimal_lit    = "0" | ( "1" … "9" ) [ [ "_" ] decimal_digits ] .
binary_lit     = "0" ( "b" | "B" ) [ "_" ] binary_digits .
octal_lit      = "0" [ "o" | "0" ] [ "_" ] octal_digits .
hex_lit        = "0" ( "x" | "X" ) [ "_" ] hex_digits .

decimal_digits = decimal_digit { [ "_" ] decimal_digit } .
binary_digits  = binary_digit { [ "_" ] binary_digit } .
octal_digits   = octal_digit { [ "_" ] octal_digit } .
hex_digits     = hex_digit { [ "_" ] hex_digit } .
```

浮点数字面量
十进制或 十六进制。

十进制浮点数由以下组成： 整数部分，小数点(decimal point)，小数部分，和一个E/e与整数（幂部分）

整数部分和小数部分的 其中一个可以省略(不能2个都省略。。)
小数点和幂部分 其中一个可以省略。
幂部分 是 10的 多少次方。

十六进制浮点数，包含0x或0X前缀，16进制的整数，小数点(radix point)，16进制的小数部分，幂部分(p/P)

整数部分 和 小数部分 的其中一个可以省略。

小数点可以省略。
幂部分不能省略。
幂部分是 2的多少次方。

```
float_lit         = decimal_float_lit | hex_float_lit .

decimal_float_lit = decimal_digits "." [ decimal_digits ] [ decimal_exponent ] |
                    decimal_digits decimal_exponent |
                    "." decimal_digits [ decimal_exponent ] .
decimal_exponent  = ( "e" | "E" ) [ "+" | "-" ] decimal_digits .

hex_float_lit     = "0" ( "x" | "X" ) hex_mantissa hex_exponent .
hex_mantissa      = [ "_" ] hex_digits "." [ hex_digits ] |
                    [ "_" ] hex_digits |
                    "." hex_digits .
hex_exponent      = ( "p" | "P" ) [ "+" | "-" ] decimal_digits .


072.40       // == 72.40
1_5.         // == 15.0
0.15e+0_2    // == 15.0

0x1p-2       // == 0.25
0x2.p10      // == 2048.0
0x1.Fp+0     // == 1.9375
0X.8p-0      // == 0.5
```

虚数字面量
整数或浮点数+i

```
0123i         // == 123i for backward-compatibility
0o123i        // == 0o123 * 1i == 83i
0xabci        // == 0xabc * 1i == 2748i
0.i
2.71828i
.12345E+5i
0x1p-2i       // == 0x1p-2 * 1i == 0.25i
```

Rune 字面量
是一个或多个char，被包含在单引号内。如'x' , '\n'

一个char，被''包围，则代表这个char自己。
多个char，以\开头，被''包围，则 按照不同的标准，可以表达不同的值。

有几种情况下，会将值作为ASCII解码。4种方式，来表达整数值是一个常量字符：
\x 后跟 2个16进制字符

\u 后跟 4个16进制字符
\U 后跟 8个16进制字符
\ 后跟 3个8进制字符

\后跟一个 字符：
\a   U+0007 alert or bell
\b   U+0008 backspace
\f   U+000C form feed
\n   U+000A line feed or newline
\r   U+000D carriage return
\t   U+0009 horizontal tab
\v   U+000B vertical tab
\\   U+005C backslash
\'   U+0027 single quote (valid escape only within rune literals)
\"   U+0022 double quote (valid escape only within string literals)

String 字面量
是char的连接。
2种类型： 原生(raw)string字面量， 解释后(interpreted)string字面量。

原生字面量是`` 包围的。
\没有特殊含义。。并且可以包含多行。

解释后字面量是 "" 包围的
会把 \及后面的字符 interpreted 成 rune字面量。 \'是非法的； \"是合法的。
\和3个八进制，\x和2个16进制，会在最终string中 表达成 单个的byte。
\377, \xFF， 都代表一个byte， 是ÿ 。

```
`abc`                // same as "abc"
`\n
\n`                  // same as "\\n\n\\n"
"\""                 // same as `"`
```

These examples all represent the same string:

```
"日本語"                                 // UTF-8 input text
`日本語`                                 // UTF-8 input text as a raw literal
"\u65e5\u672c\u8a9e"                    // the explicit Unicode code points
"\U000065e5\U0000672c\U00008a9e"        // the explicit Unicode code points
"\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e"  // the explicit UTF-8 bytes
```

。。都是 nihongo

常量(Constants)
boolean constants, rune constants, integer constants, floating-point constants, complex constants, and string constants

Rune, integer, floating-point, and complex constants are collectively called numeric constants.

这几个集合起来叫 数字常量。。。 Rune也是数字常量。。

unsafe.Sizeof 可以对任何值使用。
cap,len 对某些 表达式。
real，img 对 复数。
预定义的 true 和false 是boolean
iota 代表 整数常量。

常量可能需要手工声明类型，也可能不需要， 字面常量，true，false，iota，只包含不需要声明类型的常量的常量表达式， 不需要声明类型。

常量的类型可能被 常量声明(constant declaration) 或 转换(conversion) 来确定， 或 隐式地被声明 当被用在一个 变量声明 或 分配 或 表达式的一个操作数。如果常量无法转换成类型，就会报错。

无类型常量，有一个默认类型，这个默认类型是，这个常量在上下文中被使用的时候 被赋给的变量的类型(如果这个变量有类型)。
i := 0， 这里没有隐式类型。

默认类型是 bool,rune,int,float64,complex128,string。

变量(Variables)
变量是保存值的位置。
可以存储的值的范围，是由 变量的类型 决定的。

变量声明，方法参数，方法返回，方法前面或方法字面量，保存到一个 命名的变量中。
A variable declaration or, for function parameters and results, the signature of a function declaration or function literal reserves storage for a named variable.
。。。这是什么语言。。。

运行时，(可以)调用内置的new方法 或 获得一个合成字面量的地址 来 申请 变量的空间。
一个匿名的变量被关联到指针。 。。。 (不能命名的？)

结构变量(array,slice,struct) 有元素 和 field， 这些可能被单独地 定位(addressed)

变量的静态类型(or just type)在 声明它时 给出，或者 在new 或 复杂字面量时 给出， 或在结构变量的 每个元素的类型。

对于接口类型的变量，它也有一个 distinct 动态类型，这个类型是 在运行时 会是值的类型(除非值是nil，这个没有类型)。
执行期间，动态类型可能不同， 但 保存在接口变量中的值 总可以 转换成/可赋值到 变量的静态类型。

```
var x interface{}  // x is nil and has static type interface{}
var v *T           // v has value nil, static type *T
x = 42             // x has value 42 and dynamic type int
x = v              // x has value (*T)(nil) and dynamic type *T
```

。。静态类型就是 写好的类型，，， 动态类型就是 根据值 推导出来的类型。。
。。maybe。。 动态类型 是 实际的值， 静态类型是 声明的值。

如果一个变量没有被赋予一个值，那么它的值是 它的类型的 zero value.

类型(Types)
类型决定了 值的集合 及 这些值的 operations，methods。

类型可以通过以下来表示： 一个类型名称， 如果需要的话，可以通过指定一个类型字面量，这个字面量可以 通过 已有的类型 组合。

```
Type      = TypeName | TypeLit | "(" Type ")" .
TypeName  = identifier | QualifiedIdent .
TypeLit   = ArrayType | StructType | PointerType | FunctionType | InterfaceType |
SliceType | MapType | ChannelType .
```

go预定义了一些类型名字。 其他的需要通过类型声明来定义。 复杂类型：
array,struct,pointer,function,interface,slice,map,channel 这些类型可以通过 类型字面量 来创建。

每个类型T 有一个潜在的类型(underlying type/基础类型)： 如果T是 预定义的boolean，numeric,string类型，或者是 一个类型字面量， 那么响应的 潜在类型就是T 自己。 否则，T的潜在类型 就是 T在类型声明中 refer 的 潜在类型。。

```
type (
      A1 = string
      A2 = A1
)
type (
      B1 string
      B2 B1
      B3 []B1
      B4 B3
)
```
The underlying type of string, A1, A2, B1, and B2 is string. The underlying type of []B1, B3, and B4 is []B1.

。。type只是说 里面会定义 多个类型 ， 所以 B1 是类型名称，B1实际是string类型。以后可能会出现 组合类型。

方法集合 method sets

类型有一个 方法集合(可能是空的)。
一个接口类型的方法集合 就是它的接口。
任何其他类型(估计是指非接口类型) T的 方法集合 包含 所有声明在T中的方法。

指针类型*T 的 方法集合 是 所有声明在 *T 或 T 中的方法。
struct类型的方法集合，是在struct 类型中 描述的 方法。
其他类型 有一个 空 的方法集合。

类型的方法集合 决定了 类型实现的接口 和 方法可以被 这个类型的接收者(recevier) 调用

布尔类型。
布尔类型表示 真/假，预定义的类型是 bool， 包含true false。

数值类型
代表 整型 或 浮点数 值。 预定义的数值类型：

```
uint8       the set of all unsigned  8-bit integers (0 to 255)
uint16      the set of all unsigned 16-bit integers (0 to 65535)
uint32      the set of all unsigned 32-bit integers (0 to 4294967295)
uint64      the set of all unsigned 64-bit integers (0 to 18446744073709551615)
int8        the set of all signed  8-bit integers (-128 to 127)
int16       the set of all signed 16-bit integers (-32768 to 32767)
int32       the set of all signed 32-bit integers (-2147483648 to 2147483647)
int64       the set of all signed 64-bit integers (-9223372036854775808 to
9223372036854775807)
float32     the set of all IEEE-754 32-bit floating-point numbers
float64     the set of all IEEE-754 64-bit floating-point numbers
complex64   the set of all complex numbers with float32 real and imaginary parts
complex128  the set of all complex numbers with float64 real and imaginary parts
byte        alias for uint8
rune        alias for int32

uint     either 32 or 64 bits
int      same size as uint
uintptr  an unsigned integer large enough to store the uninterpreted bits of a
pointer value
```

所有的类型都是 定义的类型，除了 byte，rune， 它们是别名。

当不同的数字类型 混合使用时，需要显式转换。 int int32是不同类型。

String类型
string值是 byte的序列。 byte的数量就是 string的长度。。。(？一个char可以多个byte吧。究竟是字符的数量还是 byte的数量？)
string是不可变的。
预定义的类型是 string， 是一个定义类型。

使用内置的方法 len， 可以获得 string 的长度。
如果string是常量，那么长度是 编译时常量。
string的byte 能通过下标 [0, len(s)-1] 访问。尝试获得这样一个元素的地址 是非法的，如果s[i]是 string的第i个byte， &s[i]是非法的。

数组类型
数组是 某个类型的元素 的 可以计数的 序列。 元素的个数就是 数组的长度。

```
ArrayType   = "[" ArrayLength "]" ElementType .
ArrayLength = Expression .
ElementType = Type .
```

长度是数组类型的一部分。必须能evaluate出 一个 非负 constant int。

内置的方法 len 能获得 数组的长度。
[0, len(arr) - 1]。
数组类型总是一维的，但是可以组成多维类型。

```
[32]byte
[2*N] struct { x, y int32 }
[1000]*float64
[3][5]int
[2][2][2]float64  // same as [2]([2]([2]float64))
```

。。前置。。

Slice 类型 切片类型
切片是 数组的一个连续段 的描述，并提供 一种访问 到 那个数组的 编号序列。
切片类型 标志着 它的元素类型的数组的 所有切片的集合。
元素的数量 就是 切片的长度。
未初始化的切片的值 是 nil。

```
SliceType = "[" "]" ElementType .
```

切片的长度可以通过内建的方法 len 获得。
切片的长度，在执行中可能发生变化。(数组的长度不会)
切片的元素可以通过[0,len(s)-1]的下标来定位。
对于一个给定元素，它在切片中的下标 可能 小于 它在数组中的下标。

切片，一旦初始化，就总是关联着 一个数组，这个数组保存着 切片的元素。
切片和它的数组 及它的数组的其他切片 共享存储空间。不同的数组对应着不同的存储空间。
。。just a pointer..

切片可能比 它的数组短。
capacity(容量)是以下的度量：它是一个sum of 切片的长度和 数组的长度。
切片的长度不变 直到 通过slicing从原切片 生成一个新切片。
切片的capacity可以通过 内建的方法 cap(a) 来获得

使用内建方法 make来生成一个 新的，已初始化的。类型是T的 切片。make方法需要以下形参：切片类型，长度，容量(可选)。
make生成的切片，总是 分配 一个新的，隐式的数组，这个数组被 切片refer。

执行 make([]T, length, capacity) 获得的切片，等同于 分配(allocate)一个数组 并且 slicing 数组。
所以下面的2个 表达式是 相同的：
make([]int, 50, 100)
new([100]int)[0:50]

和数组一样，切片总是 一维的(one-dimensional)。 但是可以组合 来创建 多维对象 (higher-dimensional objects).

数组的数组，inner array 总是相同长度的。
但是，切片的切片(或切片的数组)，内部(切片)的长度可以 动态修改(vary dynamically)。
此外，内部切片必须被独立地初始化。
。。没有例子啊，看起来是，内部切片初始化的时候可以不同长度，并且运行时可以修改？

结构类型(struct type)
结构是 一串 有名字的元素(称为 field)， 每个field都有 名字和类型。
field名字 可能是 明确指定的(IdentifierList)，也可能是 隐式的(EmbeddedField)。
在一个结构中，非空白的field的名字 必须 唯一。

```
StructType    = "struct" "{" { FieldDecl ";" } "}" .
FieldDecl     = (IdentifierList Type | EmbeddedField) [ Tag ] .
EmbeddedField = [ "*" ] TypeName .
Tag           = string_lit .
```

```
// An empty struct.
struct {}

// A struct with 6 fields.
struct {
    x, y int
    u float32
    _ float32  // padding
    A *[]int
    F func()
}
```

一个field，声明了类型，但是没有 名字， 这种被称为 embadded field。
一个隐式属性 必须被以下说明(specified)：一个类型名T 或 一个非接口类型的指针类型 *T。。 类型T 不能是接口类型。
类型全名(unqualified) 就是 属性名。

```
// A struct with four embedded fields of types T1, *T2, P.T3 and *P.T4
struct {
    T1        // field name is T1
    *T2       // field name is T2
    P.T3      // field name is T3
```

```
    *P.T4     // field name is T4
    x, y int  // field names are x and y
}
```
。。这就不能 2个相同类型的 隐式属性。

下面是非法的，因为 属性名称 必须唯一。
```
struct {
    T     // conflicts with embedded field *T and *P.T
    *T    // conflicts with embedded field T and *P.T
    *P.T  // conflicts with embedded field T and *T
}
```

struct x中的 一个属性/非法 f， 能被调用(通过x.f)， 如果x.f 是一个合法的 selector。

Promoted fields(就是指上面的x.f的f) 行为就像 结构的普通field。 除了它们不能被 用作 属性名字 在结构的 复合字面量 中。
。。。

现有 一个结构类型S，一个已经定义的类型 T。 promoted methods 被包含在 结构的 方法集中， 通过如下的规则：
    如果S 包含一个 嵌入属性T，S和 *S 的方法集 都 包含 promoted methods with receiver T。 *S的方法集还包含 promoted methods with receiver *T.
    如果S包含一个嵌入属性 *T。S和*S的方法集都包含 promoted methods with receiver T 或*T。
。。。不知道 promoted methods with receiver T 是什么意思。

一个属性的声明后面 可能跟随着 一个可选的string字面量tag，这个tag变成了 在相关的属性声明中的 所有属性 的 一个attribute。
空的tag字符串("") 等于 没有。
tag可以通过 反射来变得可见。 对于struct来说，tag还可以成为 类型标签 一部分。

```
struct {
    x, y float64 ""  // an empty tag string is like an absent tag
    name string  "any string is permitted as a tag"
    _    [4]byte "ceci n'est pas un champ de structure"
}

// A struct corresponding to a TimeStamp protocol buffer.
// The tag strings define the protocol buffer field numbers;
// they follow the convention outlined by the reflect package.
struct {
    microsec  uint64 `protobuf:"1"`
    serverIP6 uint64 `protobuf:"2"`
}
```

指针 pointer type

指针类型 表示了 所有 指向确定类型的变量的 指针。 确定类型被称为 指针的基础类型。
未初始化的指针的值是nil

```
PointerType = "*" BaseType .
BaseType    = Type .

*Point
*[4]int
```

方法类型 Function types
方法类型表示 所有 有相同形参和返回值类型 的方法的集合。
没有初始化的变量的值是 nil

```
FunctionType   = "func" Signature .
Signature      = Parameters [ Result ] .
Result         = Parameters | Type .
Parameters     = "(" [ ParameterList [ "," ] ] ")" .
ParameterList  = ParameterDecl { "," ParameterDecl } .
ParameterDecl  = [ IdentifierList ] [ "..." ] Type .
```

在形参或返回结果 列表中，names(IdentifierList) 必须 全部存在 或全部不存在。如果存在，每个名字代表 一个 item(参数/返回)，所有 签名中的非空白的名字 都必须唯一。
。。go可以多返回！

如果不存在，每个类型 代表 一个 item(参数/返回)
参数和结果列表 总是 被 圆括号包围，除非 只有一个匿名的结果(返回只有一个，并且没有命名)，此时，这个匿名的结果可以 不用 ()包围。

方法签名的入参，可能存在 类型+前面3个点。 这种被称为 可变长参数，这个可以通过 0个或 多个实参 来匹配。

```
func()
func(x int) int
func(a, _ int, z float32) bool
func(a, b int, z float32) (bool)
func(prefix string, values ...int)
func(a, b int, z float64, opt ...interface{}) (success bool)
func(int, int, float64) (float64, *[]int)
func(n int) func(p *T)
```

接口类型 interface types
接口类型 明确了 一个方法集合，这个方法集合称为 它的接口。
一个接口类型的变量 可以保存 任何类型，只要这个类型的方法集 是 这个接口 的 超集。
没有初始化的接口类型的 变量 是 nil。
。。duck or extends?

```
InterfaceType      = "interface" "{" { ( MethodSpec | InterfaceTypeName ) ";" }
"}" .
MethodSpec         = MethodName Signature .
```

```
MethodName         = identifier .
InterfaceTypeName  = TypeName .
```

接口类型可能 明确定义方法， 也可能 通过接口类型名字 嵌入 其他接口的方法

```
// A simple File interface.
interface {
    Read([]byte) (int, error)
    Write([]byte) (int, error)
    Close() error
}
```

每个明确声明的方法的名字 必须是 唯一 且 非空白的。

```
interface {
    String() string
    String() string  // illegal: String not unique
    _(x int)         // illegal: method must have non-blank name
}
```

。。不能是 _ 作为方法名吗？ 下划线起步是可以的吧。

可能有多个类型 实现了 同一个接口。
比如：如果有 2个类型 S1，S2 有 方法集：

```
    func (p T) Read(p []byte) (n int, err error)
    func (p T) Write(p []byte) (n int, err error)
    func (p T) Close() error
```

(当T代表S1 或 S2)，那么 File接口 被 S1 S2 实现了， 不管S1，S2 中 其他的方法。
。。。duck？ 还是多继承。。。

A type implements any interface comprising any subset of its methods and may therefore implement several distinct interfaces.
类型实现包含其方法的任何子集的任何接口，因此可以实现几个不同的接口。 。。百度翻译的。。

所有类型 都实现了 空接口

```
interface{}
```

下面的接口定义，使用 类型声明 来定义 一个接口。

```
type Locker interface {
    Lock();
    Unlock()
}
```

如果S1，S2 都实现了：

```
func (p T) Lock() {…}
func (p T) Unlock() {…}
```

他们 实现了 Locker接口 和 File 接口。

一个接口T 可能使用一个 接口类型(名字E) 代替 一个方法定义(就是 方法定义的位置 写一

个接口类型)。 这种被称为 T 中嵌入接口E。

T的方法集 是T中明确声明的方法 加上 T的嵌入式的接口中的 方法。

```
type Reader interface {
    Read(p []byte) (n int, err error)
    Close() error
}

type Writer interface {
    Write(p []byte) (n int, err error)
    Close() error
}

// ReadWriter's methods are Read, Write, and Close.
type ReadWriter interface {
    Reader  // includes methods of Reader in ReadWriter's method set
    Writer  // includes methods of Writer in ReadWriter's method set
}
```

。。。接口的 继承，不是通过 接口 extends 接口。 而不是 接口 包含接口。
。。type 是 命名需要的，interface{} 是无名的接口。 type Name interface{}。。 就是命名，
。。而且 这里Reader 出现在 方法/属性 的位置。。

方法集的联合 包含 每个方法集中的方法一次，同名方法必须有相同的方法签名。

```
type ReadCloser interface {
    Reader   // includes methods of Reader in ReadCloser's method set
    Close()  // illegal: signatures of Reader.Close and Close are different
}
```

一个接口类型T 不能嵌套它自己，或者 任何嵌套了T的 接口。
。。。不能递归定义。。

Map types
map是 一个无序的组，元素是一种类型，通过一个唯一的key集合(key可以是另外一种类型)来索引(index)，
未初始化的map 的值 是nil

```
MapType     = "map" "[" KeyType "]" ElementType .
KeyType     = Type .
```

Key的类型必须有 == 和 != 比较操作 的 定义。
Key的类型不能是 function，map，slice。
如果Key的类型是 一个接口类型，比较操作 必须在 动态确定的key的值 有定义。

```
map[string]int
map[*T]struct{ x, y float64 }
```

map[string]interface{}

map元素的个数就是 长度，长度可以使用内置函数len 来获得。
运行时，使用 assignments(分配) 来增加一个元素，通过 索引表达式 来获得。通过内置函数 delete 来删除

使用 内置函数make 来创建一个 新的空的map。make需要map的类型，和一个可选的容量 hint。
make(map[string]int)
make(map[string]int, 100)

初始的容量 不是 它的size。 map会动态增长。
A nil map 等于 一个空map， 除了不能 add元素。

Channel types
一个channel 提供了 用于并发执行通过sending和receiving来交换 明确的类型的值 的方法的结构。
未初始化的channe 是nil

ChannelType = ("chan" | "chan" "<-" | "<-" "chan") ElementType .

<-操作符 明确了 channel的方向，send或receive。 如果没有说 方向，那么就是 双向的 channel可以被声明为 只send，或者 只receive。

```
chan T          // can be used to send and receive values of type T
chan<- float64  // can only be used to send float64s
<-chan int      // can only be used to receive ints
```

<- 操作符匹配最左的可能chan

```
chan<- chan int    // same as chan<- (chan int)
chan<- <-chan int  // same as chan<- (<-chan int)
<-chan <-chan int  // same as <-chan (<-chan int) chan (<-chan int)
```

通过 内置函数make 来创建 一个新的初始化过的channel值， 需要传递 channel类型和 一个可选的容量
make(chan int, 100)

容量 代表元素的个数， 设置了 channel的 buffer的size。
如果容量是0或者不写， channel是 没有buffer的，只有当sender和receiver都准备好的时候，才能传递数据。
其他情况下，channel是buffer的，如果buffer没有满(sends)，或者 buffer非空 (receives)，那么就可以 交流数据 而不会阻塞。
。。。buffer是自己的，所以buffer没有满，那就说明还能send，会send到buffer，然后等buffer真正发送出去。 非空，那就还能读取。

A nil channel 不能用于通信。

内置函数close，能把channel关闭。

receive operator 的多值的表单 报告了： 在channel关闭前，是否 一个已收到的值会被发送。

单独的channel 可能被用于 发送，接受，调用内置函数cap和len by任意数量的goroutine(协程)(并不需要进一步的同步(sync))

channel 就像一个FIFO 队列。 如果一个协程发送数据，另一个协程接受数据，那么 值会 按发送的顺序 被接受。

Properties of types and values

Type identity

Two types are either identical or different.
。。这句话有点经典。。还有第三种吗。。可能是说 嵌套类型？

一个定义的类型 总是 不同于 任意其他类型。
2个类型是一致的，如果它们underlying的类型字面量 是 结构上相等(structurally equaivalent)， 就是说，它们有相同的 字面量结构 和 相应的组件有相同的类型。
具体来说：

- 2个数组类型是相同的，如果它们有相同的元素类型 和相同的长度。
- 2个切片是相同的，如果它们有相同的 元素类型。
- 2个结构类型是相同的，如果它们有相同的 属性序列，并且属性的名字相同，类型相同，tag相同。来自不同包的 Non-exported 属性名字 总是不同的。
- 2个指针类型相同，如果它们有相同的 基础类型
- 2个方法类型相同，如果它们有相同的 参数个数，相同的返回个数，相应的参数和返回类型 都相同， 要么全都有可变长参数，要么全都没有， 参数和返回结果的 名字不需要相同。
- 2个接口类型是相同，如果它们有相同的方法集(相同的名字和相同的方法类型)，来自不同包的Non-exported方法名字 总是不同的。方法的顺序是无所谓的。
- 2个map类型相同，如果有相同的 key类型，相同的value类型
- 2个channel类型相同，如果有 相同元素类型 和相同的方向。

如果有如下声明：

```
type (
    A0 = []string
    A1 = A0
    A2 = struct{ a, b int }
    A3 = int
    A4 = func(A3, float64) *A0
    A5 = func(x int, _ float64) *[]string
)

type (
    B0 A0
    B1 []string
```

```
    B2 struct{ a, b int }
    B3 struct{ a, c int }
    B4 func(int, float64) *B0
    B5 func(x int, y float64) *A1
)

type  C0 = B0
```

=======
下面的类型是相等的：
A0, A1, and []string
A2 and struct{ a, b int }
A3 and int
A4, func(int, float64) *[]string, and A5

B0 and C0
[]int and []int
struct{ a, b *T5 } and struct{ a, b *T5 }
func(x int, y float64) *[]string, func(int, float64) (result *[]string), and A5
======

B0 and B1 are different because they are new types created by distinct type definitions; func(int, float64) *B0 and func(x int, y float64) *[]string are different because B0 is different from []string.

。。type里 == 和 空格 有什么区别。。。似乎是 ==是完全相等， 空格是指 前面的变量是后面的类型。。 所以 A0 A1 保存的是类型，而不是 string数组的值。

Assignability， 可转让的。

一个值x 能转为/赋值给 一个类型T的变量，如果 下面的任一条件成立：
    x的类型 等于T
    x的类型V 和T 有相同的 underlying 类型，并且 V T中至少有一个 不是 defined类型
。。。难道underlying/defined 是指 上面的 A0 这种 == 获得 的类型？
    T是接口，x实现了T
    x是一个双向channel值，T是channel类型，x的类型V 和 T 有相同的元素类型，并且V，T中至少有一个 不是 defined类型。
    x是一个预定义的标识符 nil，T是一个指针，方法，切片，map，channel，接口类型。
    x是一个 无类型的常量，可以作为 类型T的 一个值。

Representability 可被代表的
一个常量x 能作为类型T 的值， 如果 满足下面的某一个条件：
    x 在 T决定的值集合中。
    T是浮点类型， x 能被四舍五入到 T的精度，并且不会溢出。
    T是实数类型， x的组成部分 real(x)和imag(x) 能 作为 T的组件的 type(float32/float64)的值。

x T x is representable by a value of T because

'a'	byte	97 is in the set of byte values
97	rune	rune is an alias for int32, and 97 is in the set of 32-bit integers
"foo"	string	"foo" is in the set of string values
1024	int16	1024 is in the set of 16-bit integers
42.0	byte	42 is in the set of unsigned 8-bit integers
1e10	uint64	10000000000 is in the set of unsigned 64-bit integers
2.718281828459045	float32	2.718281828459045 rounds to 2.7182817 which is in the set of float32 values
-1e-1000	float64	-1e-1000 rounds to IEEE -0.0 which is further simplified to 0.0
0i	int	0 is an integer value
(42 + 0i)	float32	42.0 (with zero imaginary part) is in the set of float32 values

。。。0i也是0 。。。42+0i == 42。。。

x	T	x is not representable by a value of T because
0	bool	0 is not in the set of boolean values
'a'	string	'a' is a rune, it is not in the set of string values
1024	byte	1024 is not in the set of unsigned 8-bit integers
-1	uint16	-1 is not in the set of unsigned 16-bit integers
1.1	int	1.1 is not an integer value
42i	float32	(0 + 42i) is not in the set of float32 values
1e1000	float64	1e1000 overflows to IEEE +Inf after rounding

Blocks

block 是一个可能为空 序列 的 定义和代码 块，并且包含在 {} 中。

```
Block = "{" StatementList "}" .
StatementList = { Statement ";" } .
```

除了 代码中 明确的block外，还有一些隐式block：

- universe block 包括了所有的 Go source text。
- 每个package有一个 package block 包含那个包中所有的 go source text
- 每个文件有个 file block 包含 那个文件中 所有的 Go source text
- 每个 if for switch 代码块 被认为在它们自己的 隐式block中。
- switch，select中的每个 clause 代码块 act as 隐式block

Declarations and scope 声明和范围

declaration 绑定 一个非空白的 标识符 到 一个 constant，type，variable,function,label,package。每个标识符必须被声明(declared)。
在同一个block中，不能声明一个标识符2次。 一个标识符不能同时声明在 文件和package

block中。

在声明中，空白标识符 能被用作 就像 其他标识符，但是 它 不能代表/引入一个绑定 也不会声明。
在package block，标识符 init 只用来作为 初始化方法的 声明,它也不会引入新的绑定。

。。空白标识符，blank identifier, 是指 下划线 _ 。。。。

```
Declaration   = ConstDecl | TypeDecl | VarDecl .
TopLevelDecl  = Declaration | FunctionDecl | MethodDecl .
```

一个声明的标识符 的作用域是 这个标识符denote到具体的constant，type，variable,function,label,package 的 source text的 范围。

Go使用block 作为语法上的范围

- 预定义的标识符的 范围是 universe block
- 一个标识符denote了constant,type,variable,function(not method),并且是在 顶层中(不在任何方法中) 的 作用域 是 package block
- 一个导入进来的包的包名 的作用域 是 包含这个import语句的 文件的 文件block。
- 一个表示(denote) method receiver,function parameter,或result variable 的 标识符 的作用域 是 function body
- function中的 常量或变量 标识符 的作用域 是 从 ConstSpec或VarSpec 的结尾 开始，到 最小包含标识符 的block结尾 结束。
- function中的 type标识符的 作用域是 从 TypeSpec中的标识符开始， 最小包含标识符的block 结尾 结束。

block中的 标识符可能被重新声明(redeclared) 在一个内部block中。
当 内部声明的标识符 在作用域时， 它就代表了 在内部声明的 实体。
。。。话说 {int a = 1; { cout<<a; int a = 2}} 是什么 cpp中是可以编译执行的，并且是1.

package clause(就是类似java的package) 不是一个声明， 包名不会出现在 任何scope中。
只是用来identify(识别) 同一个包里的文件 , 以及 用于 import声明时 导入 包。

Label scopes
标签 通过 labeled statements 来声明， 在break continue goto中使用。
定义一个 不被使用的 标签 是非法的。
和其他标识符对比，标签 没有block作用域，不和其他 非标签 的标识符 冲突。
label的作用域是 它声明所在的 function体， 不包含任何 嵌套的方法体。
。。方法还能嵌套定义？ 就像py的 def helper 一样？

Blank identifier
是下划线。。 就像一个 匿名的 占位符(placeholder) 替换掉 规则(非空白)的标识符。
在 declarations，operand，assignments 中有特殊意义。

Predeclared identifiers
下面的 标识符 定义在 universe block中。

```
Types:
    bool byte complex64 complex128 error float32 float64
    int int8 int16 int32 int64 rune string
    uint uint8 uint16 uint32 uint64 uintptr

Constants:
    true false iota

Zero value:
    nil

Functions:
    append cap close complex copy delete imag len
    make new panic print println real recover
```

Exported identifiers

一个标识符 可能被export 来允许从另一个包访问它，一个标识符是exported，如果下列全部满足：

- 标识符首字符是 unicode大写字母
- 标识符在 package block中被声明，或 它是一个 属性名 或 方法名。

其他的标识符都是 非 exported

Uniqueness of identifiers

给定一个 标识符集合，一个标识符是unique的 如果 它和集合中其他标识符都不同。不同是指：拼写不同，或 出现在不同的包中且都是非exported的。

Constant declarations

一个常量声明，绑定 一系列的标识符(常量的名字) 到 一系列的常量表达式的值。
标识符的个数 必须等于 表达式的个数。 按照顺序赋值。

```
ConstDecl      = "const" ( ConstSpec | "(" { ConstSpec ";" } ")" ) .
ConstSpec      = IdentifierList [ [ Type ] "=" ExpressionList ] .

IdentifierList = identifier { "," identifier } .
ExpressionList = Expression { "," Expression } .
```

如果Type存在，所有的常量都是这个Type，表达式必须能转成这个Type。
如果Type不存在，常量的类型就是 相应的 表达式的值的类型，多个常量可以不同类型。
如果表达式的值是 无类型的常量，声明的常量保持无类型，常量标识符表示常量的值。

```
const Pi float64 = 3.14159265358979323846
const zero = 0.0         // untyped floating-point constant
const (
    size int64 = 1024
    eof        = -1  // untyped integer constant
)
```

```
const a, b, c = 3, 4, "foo"  // a = 3, b = 4, c = "foo", untyped integer and string constants
const u, v float32 = 0, 3    // u = 0.0, v = 3.0
```

通过一个 括号包围的声明，表达式可以省略(除了第一个)
空的list 等价于 前一个非空表达式的文字替换，并且它的类型是任何。
省略表达式列表 等价于 重复上一个list。标识符的个数必须等于 上一个list中表达式的个数。
Within a parenthesized const declaration list the expression list may be omitted from any but the first ConstSpec. Such an empty list is equivalent to the textual substitution of the first preceding non-empty expression list and its type if any. Omitting the list of expressions is therefore equivalent to repeating the previous list. The number of identifiers must be equal to the number of expressions in the previous list.

使用iota可以生成连续的值。

```
const (
    Sunday = iota
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Partyday
    numberOfDays  // this constant is not exported
)
```

。。是全部都是 noe exported 还是 只有最后一个是not exported

iota
在常量声明中，预定义的标识符iota 代表了 连续的 无类型的 integer 常量。
它的值 是 在常量声明中，各自ConstSpec 的下标的值，，从0开始。
。。ConstSpec估计是指 () 这个。 例子的最后的 是无括号的，都是0. 不，spec是指一行。。
。。并且 无论用不用 都会自增。
。。不知道 第一个iota 出现在 第3个元素，此时itoa是2还是 0？ 感觉是2. 是整个spec的下标。

```
const (
    c0 = iota  // c0 == 0
    c1 = iota  // c1 == 1
    c2 = iota  // c2 == 2
)

const (
    a = 1 << iota  // a == 1  (iota == 0)
    b = 1 << iota  // b == 2  (iota == 1)
    c = 3          // c == 3  (iota == 2, unused)
    d = 1 << iota  // d == 8  (iota == 3)
)
```

```
const (
    u         = iota * 42  // u == 0     (untyped integer constant)
    v float64 = iota * 42  // v == 42.0  (float64 constant)
    w         = iota * 42  // w == 84    (untyped integer constant)
)

const x = iota  // x == 0
const y = iota  // y == 0
```

在一个ConstSpec中 多次使用iota，获得的是相同的值

```
const (
    bit0, mask0 = 1 << iota, 1<<iota - 1  // bit0 == 1, mask0 == 0  (iota == 0)
    bit1, mask1                           // bit1 == 2, mask1 == 1  (iota == 1)
    _, _                                  //                        (iota == 2,
    unused)
    bit3, mask3                           // bit3 == 8, mask3 == 7  (iota == 3)
)
```

。。。这里go怎么知道 是 。。。好像是重复计算 之前的 表达式。 并不是+1 这种， 所以。。

上面例子利用了 隐式地重复上一个非空表达式list
。。。 这里的list是指 = 后面的2个表达式。。 不是 竖向的。。。 多值赋予。。。

Type declarations
type声明 绑定 一个标识符 和类型名字 到 类型。
类型声明有2种格式， 别名声明 和 类型声明

```
TypeDecl = "type" ( TypeSpec | "(" { TypeSpec ";" } ")" ) .
TypeSpec = AliasDecl | TypeDef .
```

别名声明 alias declarations
一个别名声明 绑定 标识符到 给定的类型

```
AliasDecl = identifier "=" Type .
```

在标识符的作用域内，它就是一个 类型的别名

```
type (
    nodeList = []*Node  // nodeList and []*Node are identical types
    Polar    = polar    // Polar and polar denote identical types
)
```

Type definitions 类型定义
类型定义 创建一个新的 distinct的类 with 相同的underlying类型 和 该类型的操作，然后绑定到 一个标识符。

```
TypeDef = identifier Type .
```

新类型被称为 一个defined类型， 它和其他任何类型都不同，包括 type it is created from

```
type (
```

```
    Point struct{ x, y float64 }  // Point and struct{ x, y float64 } are
    different types
    polar Point                   // polar and Point denote different types
)

type TreeNode struct {
    left, right *TreeNode
    value *Comparable
}

type Block interface {
    BlockSize() int
    Encrypt(src, dst []byte)
    Decrypt(src, dst []byte)
}
```

一个defined类型可能需要方法，它不会继承任何方法 从给定的类型，但 接口类型的方法集 或者 组合类型的元素类型的方法集 仍然没有修改。

```
// A Mutex is a data type with two methods, Lock and Unlock.
type Mutex struct         { /* Mutex fields */ }
func (m *Mutex) Lock()    { /* Lock implementation */ }
func (m *Mutex) Unlock()  { /* Unlock implementation */ }

// NewMutex has the same composition as Mutex but its method set is empty.
type NewMutex Mutex

// The method set of PtrMutex's underlying type *Mutex remains unchanged,
// but the method set of PtrMutex is empty.
type PtrMutex *Mutex

// The method set of *PrintableMutex contains the methods
// Lock and Unlock bound to its embedded field Mutex.
type PrintableMutex struct {
    Mutex
}

// MyBlock is an interface type that has the same method set as Block.
type MyBlock Block
```

类型定义可以用来 定义 不同的 boolean，numeric string 类型，并且 associate 它们的方法。

```
type TimeZone int

const (
    EST TimeZone = -(5 + iota)
    CST
    MST
```

```
    PST
)

func (tz TimeZone) String() string {
    return fmt.Sprintf("GMT%+dh", tz)
}
```

Variable declarations

一个变量声明 创建一个或多个变量，绑定 相应的标识符到它们，给每个标识符一个类型和一个初始值。

```
VarDecl     = "var" ( VarSpec | "(" { VarSpec ";" } ")" ) .
VarSpec     = IdentifierList ( Type [ "=" ExpressionList ] | "="
ExpressionList ) .

var i int
var U, V, W float64
var k = 0
var x, y float32 = -1, -2
var (
    i       int
    u, v, s = 2.0, 3.0, "bar"
)
var re, im = complexSqrt(-1)
var _, found = entries[name]  // map lookup; only interested in "found"
```

如果给了表达式列表，那么变量们会按照下面的规则 使用表达式来初始化 来 赋值。
否则，每个变量被初始化成 它的 零值。

如果提供了类型，每个变量都是这个类。否则，变量的类型由 初始值的类型决定。
如果值是无类型常量，它先会被隐式转为它的默认类型
如果是一个无类型boolean值，它先被隐式转为bool类型。
不能使用nil 来 初始化一个 没有确切类型的 变量。

```
var d = math.Sin(0.5)  // d is float64
var i = 42             // i is int
var t, ok = x.(T)      // t is T, ok is bool
var n = nil            // illegal
```

实现限制，一个编译器may认为以下是非法行为：在方法体内部声明一个 永不被使用 的变量

Short variable declarations

使用下面的语法

```
ShortVarDecl = IdentifierList ":=" ExpressionList .
```

它是 规则的有初始表达式但不带类型的变量声明 的 缩写

```
"var" IdentifierList = ExpressionList .
```

```
i, j := 0, 10
f := func() int { return 7 }
ch := make(chan int)
r, w, _ := os.Pipe()  // os.Pipe() returns a connected pair of Files and an error,
if any
_, y, _ := coord(p)   // coord() returns three values; only interested in y
coordinate
```

和规则的变量声明不同，简短变量声明may重定义 在同一个block中(或形参列表中) 已经定义过的 变量 为 相同类型， 但声明列表中至少有一个 是新声明的变量。作为结果，重定义只能出现在一个 多值的 简短变量声明中。重定义不会引入一个新变量，它只是将新值赋给原来的变量。

```
field1, offset := nextField(str, 0)
field2, offset := nextField(str, offset)  // redeclares offset
a, a := 1, 2                              // illegal: double declaration of a or
no new variable if a was declared elsewhere
```

短变量声明可能 只出现在 方法内部。在一些情况下，如if，for，switch代码块中，可能用来声明本地临时变量。

Function declarations

方法声明，把 标识符和方法名 绑定到一个方法上。

```
FunctionDecl = "func" FunctionName Signature [ FunctionBody ] .
FunctionName = identifier .
FunctionBody = Block .
```

如果方法的签名 声明了返回参数，方法体的代码块必须以 terminating statement结束。

```
func IndexRune(s string, r rune) int {
    for i, c := range s {
        if c == r {
            return i
        }
    }
    // invalid: missing return statement
}
```

方法声明可能不带方法体。这种声明提供了一个 非go 实现的方法的 签名，如一段 assembly routine(汇编)

```
func min(x int, y int) int {
    if x < y {
        return x
    }
    return y
}
```

```
func flushICache(begin, end uintptr)  // implemented externally
```

Method declarations
。。。method是方法， function是函数。。。
方法是一个有receiver的函数。
一个方法绑定 一个标识符和方法名 到 方法，并且把 这个方法关联到 receiver的基础类型。

```
MethodDecl = "func" Receiver MethodName Signature [ FunctionBody ] .
Receiver   = Parameters .
```
。。friend?

receiver 通过 在方法名 签名加一个额外的 参数项 来指明。
参数项必须声明一个 单独的 不可变的参数 --- receiver， 它的类型必须是一个 定义的类型 T 或者一个 指向定义的类型T的 指针。 T被称为receiver的base type。 receiver的基础类型不能是 指针 或 接口类型， 它必须 和method定义在 同一个包里。 method被认为绑定到 receiver的基础类型，方法名 只有在 类型T或*T 的selector中 可见。

非空白的 receiver 标识符 在方法签名中必须唯一。 如果receiver的值 没有在方法体中被引用，那么它的标识符可以从声明中省略。
方法和函数的 参数 是相同的规则的。

对于基础类型，方法绑定的 非空白名字 必须唯一。如果基础类型是 struct类型，非空白的方法和属性名 必须唯一。

现有一个定义的类型Point， 下面的声明

```
func (p *Point) Length() float64 {
    return math.Sqrt(p.x * p.x + p.y * p.y)
}

func (p *Point) Scale(factor float64) {
    p.x *= factor
    p.y *= factor
}
```

通过receiver类型 *Point 将 Length和Scale方法 绑定到 基础类型Point。

方法的类型 等于 将receiver作为第一个形参 的函数的类型。比如Scale有类型：

```
func(p *Point, factor float64)
```

然而，当一个 函数用 上面的格式声明时， 它不是方法。

Expressions
一个表达式 说明了 通过应用操作符和函数 到操作数 的计算。

Operands
操作数 代表 表达式中的 基础值。

操作数可以是 字面量，非空白标识符(表示 常量，变量， 方法，括号包围的表达式)

空白标识符只会在赋值语句的 左侧 作为一个操作数 出现。

```
Operand     = Literal | OperandName | "(" Expression ")" .
Literal     = BasicLit | CompositeLit | FunctionLit .
BasicLit    = int_lit | float_lit | imaginary_lit | rune_lit | string_lit .
OperandName = identifier | QualifiedIdent .
```

Qualified identifiers
一个合格的标识符，是一个 带包名前缀的 标识符。包名和标识符都不能为空。

```
QualifiedIdent = PackageName "." identifier .
```

一个合格的标识符access另一个包中的标识符，which必须被导入。标识符必须被导出，且在那个包的包block中声明。
。。。。
A qualified identifier accesses an identifier in a different package, which must be imported. The identifier must be exported and declared in the package block of that package.

```
math.Sin    // denotes the Sin function in package math
```

Composite literals
组合字面量 为 struct，array,slice,map 创建值；每次被evaluate时，都生成一个新的值。它们包含 字面量的类型 及后续的 括号包围的 元素列表。每个元素的前面可能有key。

```
CompositeLit  = LiteralType LiteralValue .
LiteralType   = StructType | ArrayType | "[" "..." "]" ElementType |
                SliceType | MapType | TypeName .
LiteralValue  = "{" [ ElementList [ "," ] ] "}" .
ElementList   = KeyedElement { "," KeyedElement } .
KeyedElement  = [ Key ":" ] Element .
Key           = FieldName | Expression | LiteralValue .
FieldName     = identifier .
Element       = Expression | LiteralValue .
```

LiteralType的underlying类型必须是一个struct，array,slice,map类型(语法强制要求，除了当type被当做TypeName被给与时)。
element和key的类型 必须能赋予到 literal type中 各自的属性，元素和key类型。
这里不会有附加的转换。
key被理解为 struct的属性，array和slice的下标，map的key。
对于map字面量，所有元素都是必须有key。
多个元素有相同的 变量名称或常量key，将会是一个 错误。
对于非常量的map的key，看 evaluation order中部分。

对于struct字面量，还有下面的限制：
key必须是struct类型中声明的属性的名字
不包含任何键的元素列表必须按照声明字段的顺序列出每个结构字段的元素。。。An

element list that does not contain any keys must list an element for each struct field in the order in which the fields are declared. 。。baidu翻译的。
如果任意一个元素有key，那么所有的元素都要有key。
An element list that contains keys does not need to have an element for each struct field. Omitted fields get the zero value for that field. 。。。。。。。。。感觉是需要一个元素列表来表明哪些key代表的属性不需要元素。。但是这个元素列表 的名字是什么？怎么知道这个列表是这个功能？而且上面也有一个元素列表的。。
一个字面量 可以忽视 元素列表，这样的一个字面量evaluate出 这个类型的0值。
对其他包中的struct的 未导出的 属性 指定元素 是错误的。

给出如下定义：

```
type Point3D struct { x, y, z float64 }
type Line struct { p, q Point3D }
```

可以写出：

```
origin := Point3D{}                            // zero value for Point3D
line := Line{origin, Point3D{y: -4, z: 12.3}}  // zero value for line.q.x
```

对于数组和切片 字面量 有下面的规则：
每个元素有一个关联的整型下标来指出它在数组中的下标。
拥有key的元素把key作为它的下标。key必须是一个非负常量代表了一个int值，并且如果它(应该是指key) 是typed，那么必须是 integer类型。
一个元素没有key，那么就使用 前一个元素的下标+1 (作为key？),如果第一个元素没有key，那么就是0.

对组合字面量 取地址 会生成一个 指针 指向 一个唯一的 用字面量的值初始化 的变量。

```
var pointer *Point3D = &Point3D{y: 1000}
```

零值 对于切片或 map 类型来说， 并不等于 对这个类型进行初始化并且设置空值。所以，对空slice或map 组合字面量 的取地址 和 对于用new申请的slice或map值 取地址 的效果不一样。

```
p1 := &[]int{}    // p1 points to an initialized, empty slice with value []int{} and length 0
p2 := new([]int)  // p2 points to an uninitialized slice with value nil and length 0
```

。。new的是 nil， 字面量是 空数组。。。 我的感觉是反过来的。毕竟new才会申请。。结果是nil。。。

数组字面量的长度是 字面量类型中说明的长度。如果字面量中列举的元素少，那么剩下的元素会是数组元素类型的 0值。如果多，那么是越界错误。
符号 ... 代表 数组的长度是 最大下标+1。

```
buffer := [10]string{}             // len(buffer) == 10
intSet := [6]int{1, 2, 3, 5}       // len(intSet) == 6
days := [...]string{"Sat", "Sun"}  // len(days) == 2
```

。。空格里不写 是0？

切片字面量 描述了 整个underlying 数组字面量。切片字面量的长度和容量是最大元素下标+1。
切片字面量就像下面：
[]T{x1, x2, … xn}

下面是 对数组进行slice操作的shorthand：
tmp := [n]T{x1, x2, … xn}
tmp[0 : n]

在一个 数组，slice，map 组成的 类型T 的 复合字面量 中，元素或map的key 如果类型还是
Within a composite literal of array, slice, or map type T, elements or map keys that are themselves composite literals may elide the respective literal type if it is identical to the element or key type of T.
。。感觉是 如果元素的类型相同，那么可能省略点什么。。根据下面的，好像是省略类型T。。

类似的，那些 表示组合字面量的地址的 元素或者key 可以省略 &T，当元素或key的类型是*T时。
Similarly, elements or keys that are addresses of composite literals may elide the &T when the element or key type is *T.

```
[...]Point{{1.5, -3.5}, {0, 0}}     // same as [...]Point{Point{1.5, -3.5}, Point{0, 0}}
[][]int{{1, 2, 3}, {4, 5}}          // same as [][]int{[]int{1, 2, 3}, []int{4, 5}}
[][]Point{{{0, 1}, {1, 2}}}         // same as [][]Point{[]Point{Point{0, 1}, Point{1, 2}}}
map[string]Point{"orig": {0, 0}}    // same as map[string]Point{"orig": Point{0, 0}}
map[Point]string{{0, 0}: "orig"}    // same as map[Point]string{Point{0, 0}: "orig"}

type PPoint *Point
[2]*Point{{1.5, -3.5}, {}}          // same as [2]*Point{&Point{1.5, -3.5}, &Point{}}
[2]PPoint{{1.5, -3.5}, {}}          // same as [2]PPoint{PPoint(&Point{1.5, -3.5}), PPoint(&Point{})}
```

。。就是类型推到，如果内层不带type，那么类型就是 外层的少一维 。 指针有点不太一样(外层*Point, 内层&Point)。

产生一个 模棱两可的转换， 当 一个组合字面量 使用 ListeralType的 TypeName格式，并且作为一个操作数 出现在 关键字 和 if/for/switch代码块的左括号 之间，并且 组合字面量没有用 括号(圆/方/大)包围。 这种罕见的情况下，字面中的开始的括号被错误的转换，被认为引入了一个代码块。
为了解决这种模棱两可，组合字面量必须出现在括号中。

```
if x == (T{a,b,c}[i]) { … }
if (x == T{a,b,c}[i]) { … }
```

。。那么第一个 就转换成什么了？ 是 {…}作为 T的初始值？ 还是说 T方法的方法体？

有效的array,slice,map字面量的例子:

```
// list of prime numbers
primes := []int{2, 3, 5, 7, 9, 2147483647}
```

```
// vowels[ch] is true if ch is a vowel
vowels := [128]bool{'a': true, 'e': true, 'i': true, 'o': true, 'u': true, 'y': true}

// the array [10]float32{-1, 0, 0, 0, -0.1, -0.1, 0, 0, 0, -1}
filter := [10]float32{-1, 4: -0.1, -0.1, 9: -1}

// frequencies in Hz for equal-tempered scale (A4 = 440Hz)
noteFrequency := map[string]float32{
    "C0": 16.35, "D0": 18.35, "E0": 20.60, "F0": 21.83,
    "G0": 24.50, "A0": 27.50, "B0": 30.87,
}
```

Function literals

一个函数字面量 代表了一个 匿名的函数。

```
FunctionLit = "func" Signature FunctionBody .

func(a, b int, z float64) bool { return a*b < int(z) }
```

一个函数字面量 能被赋给 一个变量或 直接调用。

。。一等成员。

```
f := func(x, y int) int { return x + y }
func(ch chan int) { ch <- ACK }(replyChan)
```

函数字面量是闭包，它们可能更喜欢 那些定义在surrounding函数中的 变量。这些变量在surrounding函数 和 函数字面量直接 进行分享。 只要它们是能访问的，那么它们就活着。。

Primary expressions

基本表达式 是 一元和二元 表达式 的操作数

```
PrimaryExpr =
    Operand |
    Conversion |
    MethodExpr |
    PrimaryExpr Selector |
    PrimaryExpr Index |
    PrimaryExpr Slice |
    PrimaryExpr TypeAssertion |
    PrimaryExpr Arguments .

Selector       = "." identifier .
Index          = "[" Expression "]" .
Slice          = "[" [ Expression ] ":" [ Expression ] "]" |
                 "[" [ Expression ] ":" Expression ":" Expression "]" .
TypeAssertion  = "." "(" Type ")" .
Arguments      = "(" [ ( ExpressionList | Type [ "," ExpressionList ] ) [ "..." ]
[ "," ] ] ")" .



x
2
(s + ".txt")
```

```
f(3.1415, true)
Point{1, 2}
m["foo"]
s[i : j + 1]
obj.color
f.p[i].x()
```

Selectors

对于一个 基本表达式x (不是一个包名)，selector表达式：

```
x.f
```

表示 值x(有时可能是*x) 的 f元素或方法。
标识符f 被称为 (属性或方法)选择器，不能是空白标识符。
selector表达式的类型是 f的类型。
如果x 是包名，那么请看 qualified identifiers 部分

选择器f 象征 类型T的 一个属性或方法f，或者 它可能refer到 T的一个嵌入属性的 属性或方法f。
访问到f时 通过的 嵌入属性的数量 被称为 T的深度。直接定义在T中的深度是0.
。。。这个 x.f 能自动找到 x.a.b.c.f ？ 那么就不能有重名 啊。

下列规则被应用到 selector上：

对于一个T类型或*T类型的 值x(T不是指针或接口类型)， x.f 表示 T中最浅层(估计是0层)的属性或方法 f，如果最浅层没有f，表达式非法。

。。t.y // t.T1.y 。。。 不是0层，就是 bfs碰到的第一个，，但是这一层有多个怎么弄？

对于 接口类型I 的对象x， x.f 表示 x所代表的动态值的 f方法，如果I的方法集中没有f，表达式非法。

有一个例外，如果 x的类型是一个 定义的指针类型， 且 (*x).f 是一个合法的 选择器表达式 表达了 一个属性(不是方法)， x.f 就是 (*x).f 的简写。

在其他任何地方，x.f 是非法的。

如果x是 指针类型，并且值是nil。x.f象征一个结构体的属性，对x.f赋值或计算x.f 会造成一个 运行时错误

如果x是 接口类型，且值是nil。 call 或 evaluate x.f (f是方法) 导致 运行时恐慌。

给出如下定义：

```
type T0 struct {
    x int
}

func (*T0) M0()

type T1 struct {
    y int
}

func (T1) M1()
```

```
type T2 struct {
    z int
    T1
    *T0
}

func (*T2) M2()

type Q *T2

var t T2     // with t.T0 != nil
var p *T2    // with p != nil and (*p).T0 != nil
var q Q = p
```

可以写：

```
t.z          // t.z
t.y          // t.T1.y
t.x          // (*t.T0).x

p.z          // (*p).z
p.y          // (*p).T1.y
p.x          // (*(*p).T0).x

q.x          // (*(*q).T0).x        (*q).x is a valid field selector

p.M0()       // ((*p).T0).M0()      M0 expects *T0 receiver
p.M1()       // ((*p).T1).M1()      M1 expects T1 receiver
p.M2()       // p.M2()              M2 expects *T2 receiver
t.M2()       // (&t).M2()           M2 expects *T2 receiver, see section on Calls
```

下面是非法的

```
q.M0()       // (*q).M0 is valid but not a field selector
```

Method expressions

如果M 是类型T 的 方法集中一个元素。T.M 是一个函数，能被调用就像 普通调用方法一样，只是前面加一个方法的 receiver(估计指T)，

```
MethodExpr    = ReceiverType "." MethodName .
ReceiverType  = Type .
```

考虑一个结构类型T，有2个方法，Mv的receiver是类型T，Mp的receiver是类型*T。

```
type T struct {
    a int
}
func (tv  T) Mv(a int) int         { return 0 }  // value receiver
func (tp *T) Mp(f float32) float32 { return 1 }  // pointer receiver

var t T
```

。。拆分 类/结构的定义 和 类/结构的方法 好恶心。。friend满天飞。哪里找得到啊。

表达式 T.Mv 提供了 一个函数等同于 Mv， 但是有一个明确的receiver作为它的首个参数，它有签名：
func(tv T, a int) int

函数可以被正常的调用with一个确切的receiver，所以 这里5个调用是等价的：
t.Mv(7) 。。 这个是 method value，只不过 结果是相同的。其他的都是function value
T.Mv(t, 7)
(T).Mv(t, 7)
f1 := T.Mv; f1(t, 7)
f2 := (T).Mv; f2(t, 7)

类似的，表达式 (*T).Mp 产生一个 函数值 代表 Mp with 签名：
func(tp *T, f float32) float32

对于一个有值选择器的 方法， 可以获得 一个函数with 一个确切的指针接受者
所以 (*T),Mv 产生了一个 函数value 代表 Mp with 签名：
func(tv *T, a int) int

这样一个函数 间接 通过 receiver 来创建一个 值 pass as(被当做？) receiver 到 underlying method，方法不会重写 值，这个值的地址 在 方法调用时被传递。
Such a function indirects through the receiver to create a value to pass as the receiver to the underlying method; the method does not overwrite the value whose address is passed in the function call.

a value-receiver function for a pointer-receiver method, is illegal because pointer-receiver methods are not in the method set of the value type.
指针接收器方法的值接收器函数是非法的，因为指针接收器方法不在值类型的方法集中。 。。翻译的。

function values来自 使用调用语法调用方法。 receiver作为第一个参数 来调用，就是说，f := T.Mv f被调用时就像 f(t, 7)，而不是 t.f(7)。
为了构建一个函数，这个函数绑定到receiver， 可以使用 function literal 或 method value
。。。？？？。。。。

合法的： 获得一个 function value 从 一个接口类型的方法。 获得的function take 一个那个接口类型的确切的 receiver。

Method values
。。。函数值，方法值。。。应该是为了传递方法，所以 有值。

如果表达式 x 有静态类型T，并且M在T的方法集中， x.M 被称为 method value.
x.M是一个 函数值，能被调用 就像调用 x.M一样(参数一致)。
表达式x 被evaluate 和 save，在 method value的 evaluate期间。被保存的副本被用作 receiver 在后续的任何调用中。
。。。lazy-init?

类型T可以是接口，或非接口类型。。。。

在之前的 method expression 中，考虑 一个结构类型T with 2个方法 Mv，它的receiver是类型T， Mp，它的receiver是类型*T。

```
type T struct {
    a int
}
func (tv  T) Mv(a int) int         { return 0 }  // value receiver
func (tp *T) Mp(f float32) float32 { return 1 }  // pointer receiver

var t T
var pt *T
func makeT() T
```

表达式 t.Mv 提供了 一个 function value of type: func(int) int
下面的2种调用是等价的:

```
t.Mv(7)
f := t.Mv; f(7)
```

。。。so crazy。。。类.方法 是 function value， 对象.方法是 method value。。。。。有点问题，这里都说 yield a function value of type。。但是 这里是 method value 块 啊。。

类似的 表达式 pt.Mp 产生了一个 function value of type: func(float32) float32

就select而言，一个 通过 使用指针指向receiver 的 指向 非接口方法 的引用 会自动 对指针 解引用。 pt.Mv 等同于 (*pt).Mv。

就method call而言，一个通过 使用addressable 值 作为 指针接受者 指向 非接口方法的引用 会被自动 take address， t.Mp 等于 (&t).Mp

As with selectors, a reference to a non-interface method with a value receiver using a pointer will automatically dereference that pointer: pt.Mv is equivalent to (*pt).Mv.

As with method calls, a reference to a non-interface method with a pointer receiver using an addressable value will automatically take the address of that value: t.Mp is equivalent to (&t).Mp.

```
f := t.Mv; f(7)   // like t.Mv(7)
f := pt.Mp; f(7)  // like pt.Mp(7)
f := pt.Mv; f(7)  // like (*pt).Mv(7)
f := t.Mp; f(7)   // like (&t).Mp(7)
f := makeT().Mp   // invalid: result of makeT() is not addressable
```

。。感觉就是，T 和 *T 在调用的时候并没有什么区别， go 可以自动 转换成 指针类型 或

把指针类型 反引用成对象。 。。。。 对于非接口方法。

上面是 非接口类型。 从 接口类型的值 创建一个 method value 也是可以的。
```
var i interface { M(int) } = myVal
f := i.M; f(7)  // like i.M(7)
```

。。。= myVal 又是什么。。。

Index expressions
基础的表达式是 a[x]。 表示 数组的元素， 指向数组，切片，map的下标x。 x被称为下标或 map key。

有以下规则：
如果a不是map：
    x必须是 integer类型或 无类型的 常量
    常量下标 必须是 非负，且 能代表一个 int类型的值
    无类型的常量 被当做 int类型
    x 在 [0,len(a)) 中。 否则就越界。

对于 a 是 数组类型A
    一个常量下标 必须在范围内
    如果x 越界， run-time panic 发生
    a[x] 是 数组中下标x的 元素，a[x]的类型是 A的element tpye(应该是指声明A时声明的[]类型)。

对于 a是 数组类型的指针
    a[x] 是 (*a)[x] 的简写

对于 a是 切片类型S
    如果x越界，runtime panic
    a[x] 是 slice中下标为x的元素， a[x]的类型是 S的element type

对于 a是 string 类型
    如果 string a 也是常量，那么 常量下标必须在范围内。
    如果 x越界， runtime panic
    a[x] 是 非常量byte值 在下标x， a[x]的类型是 byte
    a[x] may not be assigned to。。。 may…
。。a[x] 是byte。。 不是char(rune)么。。 那么rune哪里拿。。

对于 a是 map类型M
    x的类型 必须 能转为 M的key的类型
    如果map包含一个entry with x， a[x]就是map中 key为x 的元素， a[x]的类型 是 M的element type
    如果map 是nil ，或不包含entry， a[x]是 M的elementType 的0值

其他情况下 a[x] 是非法的。

一个类型map[K]V 的map 的 下标表达式，使用在 一个赋值或初始化中， 通过一个特殊的形式：

```
v, ok = a[x]
v, ok := a[x]
var v, ok = a[x]
```

产生了一个附加的 无类型boolean 值， ok是true，如果x存在。 false，如果x不存在。

对nil map的 某个元素赋值 产生一个 runtime panic

Slice expressions
切片表达式构造一个 substring 或 slice 从 一个 string，array，array指针，slice。
2种变体： 简单的(指明low high bound)， 丰富(full)的(简单的+ 指定capacity)

Simple slice expressions
对于 string， array ,pointer to array, slice a ,基本的表达式：
a[low: high]
创建了一个 substring 或 slice。 通过 下标 low 和high 从 a中 获得元素 作为结果。
表达式的结果的下标从0开始，长度是 high-low.

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

the slice s has type []int, length 3, capacity 4, and elements
```
s[0] == 2
s[1] == 3
s[2] == 4
```

。。capacity 是 len + 1. 这是什么 默认规则？ 多的那个放了什么？ nil？ 0值？ s[3]是否会 runtime panic?
。。high 是取不到的。下面默认是 len(a)。

更方便的是， low 和high 可以省略。 low省略就是0， high省略就是 操作数的长度
```
a[2:]  // same as a[2 : len(a)]
a[:3]  // same as a[0 : 3]
a[:]   // same as a[0 : len(a)]
```

如果 a 是 数组的pointer， a[low:high] 是一个 (*a)[low:high] 的简写。

对于数组 或 string， 0<=low<=high<=len(a)。 否则是 越界。
对于slices，下标上界是切片的容量 cap(a) 而不是 长度。
常量下标必须非负 且代表了一个 int类型的值。 对于数组或常量string。常量下标必须在范围内。 如果2个下标都是常量，则必须满足low <= high. 如果 越界，runtime panic.

除了 无类型的string， 如果 slice的操作数 是string或slice，slice操作的结果是 一个和被操作数类型相同的 非 常量的值
对于无类型string 操作数，结果是一个 非 常量的值 of string类型。
如果slice操作数是一个数组，它必须是 可寻址的，并且 slice操作 的结果是 a slice with the samce element type as the array.。。。到底是slice 还是 array。。感觉是slice，但是 as the array 到底是什么意思。

如果slice表达式的 操作数 是 一个nil slice， 结果是 nil slice。否则，结果是 slice，和 操作数的 underlying array 共享数组。
。。。如果是一个 nil string， nil array呢。。。

```
var a [10]int
s1 := a[3:7]   // underlying array of s1 is array a; &s1[2] == &a[5]
s2 := s1[1:4]  // underlying array of s2 is underlying array of s1 which is array a; &s2[1] == &a[5]
s2[1] = 42     // s2[1] == s1[2] == a[5] == 42; they all refer to the same underlying array element
```

Full slice expressions

对于数组，pointer to 数组，slice (这里没有string) 基本的表达式是：
a[low : high : max]

构建相同类型的slice，和简单的slice表达式a[low:high] 有相同的长度 和元素。
额外的，它设置了 返回的slice的 容量 为 max-low.
只有第一个下标能被忽略，默认为0.

```
a := [5]int{1, 2, 3, 4, 5}
t := a[1:3:5]
the slice t has type []int, length 2, capacity 4, and elements
t[0] == 2
t[1] == 3
```

如果a 是指向数组的指针， a[low:high:max] 是 (*a)[low:high:max] 的简写。
如果 切片操作的 操作数 是数组，那么它必须是 addressable

如果 0 <= low <= high <= max <= cap(a)，则是in range， 否则是out of range。
一个常量下标必须非负 且能代表 int类型的一个值。
对于数组，常量下标 必须是 in range。如果多个下标是常量，它们都需要满足上面的关系。
如果下标 越界， runtime panic

Type assertions
对于 接口类型的 x表达式 和 类型T， 基本表达式：
x.(T)

断言：x不是nil，且 x的值是类型T。
The notation(符号) x.(T) is called a type assertion.

更准确的说，如果T 不是接口类型，x.(T) 断言 x的动态值的类型 是 identical(完全一样) 和 T类型。 此时，T必须实现 x的(接口)类型，否则 类型断言是无效的 由于 x无法保存一个类型T的值。
如果T是接口类型， x.(T) 断言： x的动态类型 实现了接口T。

如果 类型断言 hold，表达式的值 是存储在x中的值，并且值的类型是T。 如果类型断言 失败，runtime-panic。

换言之，尽管x 的动态值 只在运行时确定， 但是 x.(T)的类型 在正确的程序中 可以认为是T类型。

```
var x interface{} = 7          // x has dynamic type int and value 7
。。。so crazy…
i := x.(int)                   // i has type int and value 7

type I interface { m() }

func f(y I) {
    s := y.(string)        // illegal: string does not implement I (missing
    method m)
    r := y.(io.Reader)     // r has type io.Reader and the dynamic type of y must
    implement both I and io.Reader
    …
}
```

类型断言 被用在 赋值或 初始化 中的 形式：

```
v, ok = x.(T)
v, ok := x.(T)
var v, ok = x.(T)
var v, ok interface{} = x.(T) // dynamic types of v and ok are T and bool
```

生成了一个 额外的 无类型boolean值， ok的值是true，如果assertion hold。 否则是false， 此时v的值 是 Type T的 0值。 不会有 runtime panic

Calls

一个表达式f of function type F: f(a1, a2, ... an)

使用参数a1,a2,..an 调用f。 除了一个特殊的例子，参数必须 是 单值表达式，这个表达式的值可以被转为 F的参数类型， 这个表达式 被evaluate 在 函数被调用前。

表达式的类型是 F的result type。

A method invocation is similar but the method itself is specified as a selector upon a value of the receiver type for the method.

```
math.Atan2(x, y)  // function call
var pt *Point
pt.Scale(3.5)     // method call with receiver pt
```

在函数调用中，方法值 和 参数 被evaluate 通过 the usual order 的顺序(，，这个是指向了Order of evaluation，里面挺烦了，感觉是 默认从左到右 (包括=左侧一起 从左到右)。)

evaluate后，参数通过value 传递到 函数，然后 被调用的函数开始执行。

函数的返回参数 通过value 传递返回给caller 当函数return时。

call 一个 nil 函数值 会导致 runtime panic

一个特殊情况，如果 一个函数或方法g的 返回值 的个数和类型 完全等于 另一个 函数或方法f 的形参， 那么调用 f(g(params_of_g)) 会调用f，在按顺序 绑定g的返回值到f 的参数后。

f的调用 只能包含 g的调用，不能包含其他参数，且g必须至少有一个返回。 如果f有一

个 ... 参数，它会被赋予 g的返回值中 多余的部分。
The call of f must contain no parameters other than the call of g, and g must have at least one return value.
。。我怎么觉得不对呢，下面doc的例子，Split中除了len 还有其他的参数啊。

```
func Split(s string, pos int) (string, string) {
    return s[0:pos], s[pos:]
}

func Join(s, t string) string {
    return s + t
}

if Join(Split(value, len(value)/2)) != value {
    log.Panic("test fails")
}
```

一个方法调用 x.m() 是有效的， 如果 x的类型的方法集 包含m，并且 实参列表能赋给 m的形参列表。
如果x是 可寻址的， 且 &x 的方法集 包含 m， 那么 x.m() 是 (&x).m() 的简写。

```
var p Point
p.Scale(3.5)
```

There is no distinct method type and there are no method literals.
。。。没有方法字面量。。 估计有 函数字面量。。
。。没有独一无二的方法类型。

Passing arguments to ... parameters
如果f是可变长的，有着 一个final参数p，p的类型是 ...T。 在f中， p的类型就是 []T。如果f 被调用的时候 p没有实际参数，那么 p是nil。其他情况下， 被传递是值 是一个 新的切片，切片是[]T 类型，并且 underlying 数组是 新的。这个数组的连续的元素就是 实际的参数，这些参数必须全都 能转为T类型。 切片的长度和容量 是 绑定到p的实参的长度，容量，每次call的时候可能不同。

给出如下的方法和调用：

```
func Greeting(prefix string, who ...string)
Greeting("nobody")
Greeting("hello:", "Joe", "Anna", "Eileen")
```

第一次call中， who是nil。 第二次是 []string{"J","A","E"}

如果最终实参 能被转为 []T，并且最终实参后面有... ，它会被传递 不做任何修改 作为 ...T参数的值。 这种情况下，没有新的 slice 被创建。

```
s := []string{"James", "Jasmine"}
Greeting("goodbye:", s...)
```

在Greeting方法中， 有着 和 s 相同的 underlying数组。

Operators
在表达式中，操作符 结合 操作数。

```
Expression = UnaryExpr | Expression binary_op Expression .
UnaryExpr  = PrimaryExpr | unary_op UnaryExpr .

binary_op  = "||" | "&&" | rel_op | add_op | mul_op .
rel_op     = "==" | "!=" | "<" | "<=" | ">" | ">=" .
add_op     = "+" | "-" | "|" | "^" .
mul_op     = "*" | "/" | "%" | "<<" | ">>" | "&" | "&^" .

unary_op   = "+" | "-" | "!" | "^" | "*" | "&" | "<-" .
```

比较运算符在其他地方讨论。
对于其他二进制操作符，操作数的类型必须一致 除非 操作包含 shift 或 无类型的常量。
对于只包含常量的操作，查看 constant expression 部分。。。

除了shift 操作，如果一个 操作数 是无类型的常量，另一个不是，那么常量会被隐式转为另一个操作数的类型。

shift表达式的 右操作数 必须是 integer类型或 无类型常量但代表了一个 uint类型的值。
如果 非常量的shift表达式的 左操作符 是一个 无类型的常量，它首先被隐式转为 它被认为应该是(assume)的类型，这个类型是 整个shift表达式被 它的左值 替换时 会退到出来的类型。

```
var a [1024]byte
var s uint = 33

// The results of the following examples are given for 64-bit ints.
var i = 1<<s                  // 1 has type int
var j int32 = 1<<s            // 1 has type int32; j == 0
var k = uint64(1<<s)          // 1 has type uint64; k == 1<<33
var m int = 1.0<<s            // 1.0 has type int; m == 1<<33
var n = 1.0<<s == j           // 1.0 has type int; n == true
var o = 1<<s == 2<<s          // 1 and 2 have type int; o == false
var p = 1<<s == 1<<33         // 1 has type int; p == true
var u = 1.0<<s                // illegal: 1.0 has type float64, cannot shift
var u1 = 1.0<<s != 0          // illegal: 1.0 has type float64, cannot shift
var u2 = 1<<s != 1.0          // illegal: 1 has type float64, cannot shift
var v float32 = 1<<s          // illegal: 1 has type float32, cannot shift
var w int64 = 1.0<<33         // 1.0<<33 is a constant shift expression; w ==
1<<33
var x = a[1.0<<s]             // panics: 1.0 has type int, but 1<<33 overflows
array bounds
var b = make([]byte, 1.0<<s)  // 1.0 has type int; len(b) == 1<<33

// The results of the following examples are given for 32-bit ints,
// which means the shifts will overflow.
var mm int = 1.0<<s           // 1.0 has type int; mm == 0
```

```
var oo = 1<<s == 2<<s          // 1 and 2 have type int; oo == true
var pp = 1<<s == 1<<33         // illegal: 1 has type int, but 1<<33 overflows int
var xx = a[1.0<<s]             // 1.0 has type int; xx == a[0]
var bb = make([]byte, 1.0<<s)  // 1.0 has type int; len(bb) == 0
```

。。。go 怎么判断 int 应该是32/64 ？ 也是靠 os?

Operator precedence 运算符优先级
一元操作符优先级最高。 在 代码块(而不是表达式) 中的++/-- 不属于 运算符的体系，导致 statement *p++ 等同于 (*p)++。

。。statement(语句) expression(表达式) 。。。 区别 好小 啊。 不知道go里的定义 是什么。。。。看这个网址 最上面， 就有 2个 顶级模块，一个 是 expression (目前还在 expression中。。)，一个是 statement(还在后面。。)
。An expression specifies the computation of a value by applying operators and functions to operands.
。Statements control execution.
。。计算值， 和 控制流程 的区别。。 但是上面 *p++ 不应该算 控制流程吧。。。

二元操作符 有5个优先级， multiplication 操作符最高，addition操作符第二，comparison 第三， &&第四，||第五。

Precedence	Operator
5	* / % << >> & &^
4	+ - | ^
3	== != < <= > >=
2	&&
1	||

相同等级的二元操作符，从 左到右 结合

```
+x
23 + 3*x[i]
x <= f()
^a >> b
f() || g()
x == y+1 && <-chanInt > 0
```

Arithmetic operators

算术操作符 应用到 数值 值，生成一个 和 第一个操作数类型相同的 结果。
4个标准算术操作符(+-*/) 应用到 整型，浮点，复数。 + 也可以用到 string
位运算符 和 shift 只应用到 整型

```
+    sum                    integers, floats, complex values, strings
-    difference             integers, floats, complex values
```

```
*    product                integers, floats, complex values
/    quotient               integers, floats, complex values
%    remainder              integers

&    bitwise AND            integers
|    bitwise OR             integers
^    bitwise XOR            integers
&^   bit clear (AND NOT)    integers

<<   left shift             integer << integer >= 0
>>   right shift            integer >> integer >= 0
```

。。&^ 是什么操作。。

Integer operators
the integer quotient q = x / y and remainder r = x % y satisfy the following relationships:
x = q*y + r and |r| < |y| 。。这个好像没有用 -5 = -3*2 + 1 也符合这个的

```
 x     y     x / y     x % y
 5     3       1         2
-5     3      -1        -2
 5    -3      -1         2
-5    -3       1        -2
```

。。是 舍弃小数 。靠近0.

一个例外： 如果 dividend(被除数) x 是一个 x的类型的 最小负数， x/-1 会等于 x 并且余数是0. 由于 整型溢出。

```
                        x, q
int8                    -128
int16                 -32768
int32            -2147483648
int64    -9223372036854775808
```

如果除数 是一个 常量，它必须不能等于0。 如果 运行时 除数为0，那么 runtime-fabric。
如果 被除数 是 非负数 ，除数 是2的 幂。 除法会被替换为 右移， 计算余数会被替换为位与。

```
 x      x / 4      x % 4      x >> 2      x & 3
 11       2          3          2           3
-11      -2         -3         -3           1
```

。。之前的 x = q*y + r and |r| < |y| 规定了 乘法 和 除法，余数 的关系。

shift操作的 右操作数 必须 非负。 否则 runtime panic。
shift操作 实现了 算术shift， 如果 左操作数 是 一个有符号整型， 逻辑shift，如果左操作数是 无符号整型。

。。炸裂，算术左/右移，逻辑左/右移。 几个方面： 算术的最高位不动。 算术>>会补最高位。。 记住 <<>>是为了 倍增 半减 。 所以 负数>> 需要补1 。 正数>>不需要，所以就是

补符号位。　最好 还是试试。。。

shift次数没有上限(there is no upper limit on the shift count)。
Shifts behave as if the left operand is shifted n times by 1 for a shift count of n.
As a result, x << 1 is the same as x*2 and x >> 1 is the same as x/2 but truncated towards negative infinity.

对于 整型 操作数， 一元操作符 + - ^ 的定义：

```
+x                          is 0 + x
-x    negation              is 0 - x
^x    bitwise complement    is m ^ x  with m = "all bits set to 1" for unsigned x
                                      and  m = -1 for signed x
```

Integer overflow
对于无符号整型值，操作符+，-，*，<< are computed modulo 2^n （这个是把结果%，还是每个操作数。。。操作数不可能超过2^n。。。等于就是 溢出的直接移除了。。mod 2^n 好像就是 & 1111..111)， n是 无符号整数(操作数)的类型的bit width。
Loosely speaking(不精确地说)，这些 无符号整型 discard high bits upon overflow，并且 程序可能依靠 "wrap around"。

。。上面没有/ ，下面有
对于有符号整型值，+ - * / << 可能 合法 溢出，并且存在返回值 并且 是 有符号整型 定义的。
溢出不会导致 runtime panic。 编译器可能不会优化代码， 在 溢出不会发生 这种假设下。例如，它可能不会假定 x<x+1 永远为真。

Floating-point operators
对于浮点数 和 复数， +x 等于 x， -x是 x的负数。
浮点数或 复数 除以0 的结果是 未定义的，在IEEE-754 标准中。 在实现中，会runtime panic.

一个实现 可能 结合多个 浮点操作 到一个 单独的 fused 的 操作，可能 across(穿过) statements，且 提供一个结果，这个结果 可能和 分别执行和 舍入 的结果 不同。
。。。就是 精度可能不一样， 现在看起来 go提供的 更精确， 只不过 和 分步执行的 结果可能不同。

比如，一些架构 提供一个 "fused multiply and add" (FMA) 操作， 这种操作 在计算 x*y+z时 不会立刻 对 x*y 进行round(四舍五入)。

下面的例子展现了 什么时候 go的实现 能用 那种操作。

```
// FMA allowed for computing r, because x*y is not explicitly rounded:
r  = x*y + z
r  = z;   r += x*y
t  = x*y; r = t + z
*p = x*y; r = *p + z
r  = x*y + float64(z)
```

```
// FMA disallowed for computing r, because it would omit rounding of x*y:
r   = float64(x*y) + z
r   = z; r += float64(x*y)
t   = float64(x*y); r = t + z
```

String concatenation

能用+ ， += 操作符 来连接 string。

```
s := "hi" + string(c)
s += " and good bye"
```

string的相加，通过连接操作数，生成了一个新的string。

Comparison operators

比较2个操作数，生成一个 无类型的boolean 值。

```
==    equal
!=    not equal
<     less
<=    less or equal
>     greater
>=    greater or equal
```

对于任何比较， 第一个操作数 必须能 赋给 第二个操作数， 或者 反之。

等于比较 == ，!= 应用到的操作数 是 comparable。
顺序比较 > >= < <= 应用到的操作数 是 ordered。
这些条件和 比较的结果 如下定义：

- Boolean值是 comparable。 2个Boolean值是相等，如果 全是true 或全是 false。
- Integer是 comparable 和ordered，
- 浮点值 comparable 和 ordered。
- 复数 comparable， 相等 当 实数部分== 且 虚数部分==
- string值 comparable，ordered，字典顺序
- 指针 comparable， == 当 指向同一个变量 或都是nil。 指向distinct 0值变量的指针可能相等 也可能不相等
- channel comparable，== 当 它们被同一个call to make 而生成， 或者都是nil
- 接口 compareable，== 当 它们有相同的 动态类型 且动态值相等或都为nil。
- 非接口类型X的一个值x 和 接口类型T的值t 是comparable 当类型X的值是可比较的且X实现了T。 == 如果 t的动态类型等于X 且 t的动态值等于x。
- 结构体值 comparable，如果它们全部的属性都是comparable，2个结构体值相等 如果它们对应的 非空白的属性 都相等。
- 数组 comparable 如果数组元素的值 是comparable。 2个数组== 如果它们对应的元素都相等。

2个具有相同 动态类型的 接口值 比较 产生一个 runtime panic，如果 type的值 是不可比较的。 这个行为不仅应用在 接口值直接比较，也应用在 接口值组成的数组的比较 或 具有接口值属性的结构的比较。

slice,map,function 值 不可比较。当然，作为一个特例，slie,map,function 值 可以和

nil比较。
指针，channel，interface 值和nil 的比较也是允许的。

```
const c = 3 < 4            // c is the untyped boolean constant true

type MyBool bool
var x, y int
var (
    // The result of a comparison is an untyped boolean.
    // The usual assignment rules apply.
    b3        = x == y // b3 has type bool
    b4 bool   = x == y // b4 has type bool
    b5 MyBool = x == y // b5 has type MyBool
)
```

Logical operators

逻辑运算符应用到 boolean值，产生一个 和 操作数相同类型的 结果。
右操作数 在某些条件下 会eval。 (短路)

```
&&    conditional AND    p && q  is  "if p then q else false"
||    conditional OR     p || q  is  "if p then true else q"
!     NOT                !p      is  "not p"
```

Address operators

对于类型T的一个 操作数x， 取地址操作&x 生成一个 *T类型的指针，指向了x。
操作数必须是addressable，that is, either a variable, pointer indirection, or slice indexing operation; or a field selector of an addressable struct operand; or an array indexing operation of an addressable array.
。。变量，指针反引用，切片取下标，结构体的属性，数组的取下标。

作为可寻址性要求的一个 exception(例外)，x可以是一个(可能被括号包围的) composite literal。
如果 对x 进行 evaluate 会导致 runtime panic，那么 evalutate &x 也会导致 rt-p

对于类型*T 的一个操作数x， pointer indirection (指针间接寻址) *x 表示 x指向的 类型为T的 变量的 值。
如果x 是nil， evaluate *x 产生 rt-p

```
&x
&a[f(2)]
&Point{2, 3}
*p
*pf(x)

var x *int = nil
*x   // causes a run-time panic
&*x  // causes a run-time panic
```

Receive operator
对于一个 channel 类型的 操作数 ch， receive操作 <-ch 的值是 从 channel ch中 接收到的值。
channel direction 必须允许 receive操作。 receive操作的类型是 channel的元素(element)类型。
表达式会 阻塞 直到 有值可用，从一个nil 的channel中 接收，永远阻塞。从一个已关闭的channel中 receive，会立刻执行，产生 元素类型的 0值 after 任何之前 送到的值 已经被receive(。。估计有个buffer，对 好像可以 有buffer，也可以没有buffer。估计是 closed后， buffer中的 依然可以读取，buffer读完 就是 0值了)。
。。元素类型，说明 声明channel的时候 可以确定 类型， 只是可以， 后面的例子 很多有不带 元素类型的。。

```
v1 := <-ch
v2 = <-ch
f(<-ch)
<-strobe  // wait until clock pulse and discard received value
```

特殊用法：

```
x, ok = <-ch
x, ok := <-ch
var x, ok = <-ch
var x, ok T = <-ch
```

这种会产生一个 额外的 无类型 boolean结果 表示 communication(通信) 是否成功。ok为true，当 收到的值 是 成功的send操作 传递到channel中的。ok是false，如果因为 channel是 已关闭或closed 而导致 生成一个 0值。

Conversions
一个转换 改变表达式的 类型 到 转换定义的类型。
一个转换可能 在源码中 直接以字面量的形式出现， 也可能 是隐式的 在 表达式所在的 上下文中。
一个明确的转换 是一个表达式 以 T(x)的格式， T是类型， x是表达式，x能被转为T。

```
Conversion = Type "(" Expression [ "," ] ")" .
```

如果类型以操作符 * 或 <- 开始，或 类型以 关键字 func 开始 且没有result列表， 它必须 被 ()包围 来 避免 ambiguity

```
*Point(p)        // same as *(Point(p))
(*Point)(p)      // p is converted to *Point
<-chan int(c)    // same as <-(chan int(c))
(<-chan int)(c)  // c is converted to <-chan int
func()(x)        // function signature func() x
(func())(x)      // x is converted to func()
(func() int)(x)  // x is converted to func() int
func() int(x)    // x is converted to func() int (unambiguous)
```

。。最后一个 func 开始， 有result列表。 所以 不是二义的

一个常量值 x 能被转为 类型T， 如果x 可以 用T的一个值 代表。
一个特例，常量值 x 能被 显式转为 string类型， 通过 使用 对非静态x 使用的 规则。

转换一个常量 产生一个 有类型的常量 结果：

```
uint(iota)               // iota value of type uint
float32(2.718281828)     // 2.718281828 of type float32
complex128(1)            // 1.0 + 0.0i of type complex128
float32(0.49999999)      // 0.5 of type float32
float64(-1e-1000)        // 0.0 of type float64
string('x')              // "x" of type string
string(0x266c)           // "♬" of type string
MyString("foo" + "bar")  // "foobar" of type MyString
string([]byte{'a'})      // not a constant: []byte{'a'} is not a constant
(*int)(nil)              // not a constant: nil is not a constant, *int is not a
boolean, numeric, or string type
int(1.2)                 // illegal: 1.2 cannot be represented as an int
string(65.0)             // illegal: 65.0 is not an integer constant
```

非常量值x 能被转为类型T 在以下的case：

- x 可以转为 T
- 忽视结构体的tag，x的类型 和 T 有 相同的 underlying types
- 忽视结构体的tag，x 的类型 和 T 都是 非定义的指针类型，且 它们的指针基础类型有相同的 underlying types
- x的类型 和T 都是 整型或浮点类型
- x的类型 和 T 都是 复数类型
- x是整型 或 bytes的切片 或 rune， T是 string类型
- x是string，T是 bytes的切片或rune
- x是slice，T是 指向 array的指针，且 切片和数组类型 有相同的元素类型。

结构体tag 被忽视 当 为了转换 而进行比较以判断是否一致：

```
type Person struct {
    Name    string
    Address *struct {
        Street string
        City   string
    }
}

var data *struct {
    Name    string `json:"name"`
    Address *struct {
        Street string `json:"street"`
        City   string `json:"city"`
    } `json:"address"`
}

var person = (*Person)(data)  // ignoring tags, the underlying types are identical
```

具体的规则 应用到 (非常量) 转换 between 数值类型 和 string类型 之间 互转。
这些 转换可能修改 x的 representation(代表，表现)，导致 runtime cost。
所有其他转换 只修改 x的类型，不修改 x的representation。
。。representation 感觉可以认为 是 x的 内容 (内存上的)。

没有语言的架构来转换 pointer和整型。 包"unsafe" 实现了这个功能 under restricted(受限的) circumstances(情况) 。。

Conversions between numeric types
对于非常量的数字值 的转换，以下规则被应用：

当 在 整数类型之间转换时，如果值是 有符号整型，it is sign extended to implicit infinite precision (它被有符号推广到 隐式无限精度)；否则 it is zero extended (0扩展)。。。。估计是指 int32转为 int64时，高位补什么，如果有符号，就补符号，如果无符号，就补0。。。 It is then truncated to fit in the result type's size.
。。。就是 有符号，就无限补符号， 没有符号就无限补 0， 都补到无穷长， 然后直接截短到 目标类型的长度。
例如：if v := uint16(0x10F0), then uint32(int8(v)) == 0xFFFFFFF0。转换从产生一个 有效值， no indication(表明，象征) of overflow.

当 把浮点数转为整型时， 小数部分直接丢弃 (向0截断 truncation towards zero)。。。岂不是 -1.8。。好像没有问题，符合 逻辑的。 -1.8 到 -1. 直接丢弃 小数部分。

当转换 整型或浮点数 到 浮点数类型时，或 转换 复数到 另一个复数类型时， 结果值是 rounded(四舍五入or银行家圆整) 到 目标类型的精度的。
比如，float32类型的变量x 的值 可以 使用超过IEEE 754-32bit数 的进度 进行 存储， 但是 float32(x) 表示 x的值的round后的结果 到 32bit精度。 类似的，x+0.1可能使用超过32位的精度来保存， 但是 float32(x+0.1)不会。

在所有 非常量的 涉及浮点数或复数值 的 转换中， 如果结果类型 不能代表 转换后的值，这个转换依然成功，但是 结果值依赖于 具体实现。

Conversions to and from a string type
转换一个有符号或无符号的整型值到 string类型 会生成一个 string，包含 整型的 utf-8格式的表示。 整型 是一个无效的 unicode码，则转为 "\uFFFD"。
。。是用 整型 去 unicode表中 搜 对应下标的 string。

```
string('a')       // "a"
string(-1)        // "\ufffd" == "\xef\xbf\xbd"
string(0xf8)      // "\u00f8" == "ø" == "\xc3\xb8"
type MyString string
MyString(0x65e5)  // "\u65e5" == "日" == "\xe6\x97\xa5"
```

转换bytes 的切片 到 string类型，产生 一个string，这个string的连续byte就是 切片的元素。

```
string([]byte{'h', 'e', 'l', 'l', '\xc3', '\xb8'})   // "hellø"
```

```
string([]byte{})                                    // ""
string([]byte(nil))                                 // ""

type MyBytes []byte
string(MyBytes{'h', 'e', 'l', 'l', '\xc3', '\xb8'})  // "hellø"
```

转换rune的切片 到 string， 产生一个string，由 每个独立的rune值转为string，然后组合。

```
string([]rune{0x767d, 0x9d6c, 0x7fd4})   // "\u767d\u9d6c\u7fd4" == "白鹏翔"
string([]rune{})                         // ""
string([]rune(nil))                      // ""

type MyRunes []rune
string(MyRunes{0x767d, 0x9d6c, 0x7fd4})  // "\u767d\u9d6c\u7fd4" == "白鹏翔"
```

转换string类型的值到 bytes类型的切片 会生成一个 切片，这个切片的 连续元素 就是string中的 bytes

```
[]byte("hellø")   // []byte{'h', 'e', 'l', 'l', '\xc3', '\xb8'}
[]byte("")        // []byte{}

MyBytes("hellø")  // []byte{'h', 'e', 'l', 'l', '\xc3', '\xb8'}
```

。。。。？这种是切片 还是数组？ 也是切片？

。。[]中有长度 就是数组， 没有就是 切片。

转换string类型的值 到 rune类型的切片， 生成一个 切片，包含 string的每个点的 独立的unicode码。

```
[]rune(MyString("白鹏翔"))  // []rune{0x767d, 0x9d6c, 0x7fd4}
[]rune("")                 // []rune{}

MyRunes("白鹏翔")           // []rune{0x767d, 0x9d6c, 0x7fd4}
```

Conversions from slice to array pointer

转换切片到数组指针 会生成 一个指针，指向切片的underlying 数组。 如果切片的长度 小于 数组的长度，会生成一个 rt-p

```
s := make([]byte, 2, 4)
s0 := (*[0]byte)(s)      // s0 != nil
s1 := (*[1]byte)(s[1:])  // &s1[0] == &s[1]
s2 := (*[2]byte)(s)      // &s2[0] == &s[0]
s4 := (*[4]byte)(s)      // panics: len([4]byte) > len(s)

var t []string
t0 := (*[0]string)(t)    // t0 == nil
t1 := (*[1]string)(t)    // panics: len([1]string) > len(t)
```

```
u := make([]byte, 0)
u0 = (*[0]byte)(u)        // u0 != nil
```

Constant expressions
常量表达式 可能 值包含 常量操作数， 会在编译时 evaluate
无类型boolean，数值，string 常量 可能被 用作 操作数，不管它是不是 合法的：被用作一个 布尔，数值，string 类型的操作数。

一个常量比较 总会生成 无类型的布尔。 如果 常量shift表达式 的 左操作数(左值) 是一个无类型的 常量，结果会是一个 整型常量， 否则，结果是一个 和左值相同类型的 常量，左值的类型必然是 一个 整型类型。

其他 对无类型常量的操作 生成一个 same kind 的无类型常量；即，一个布尔，整型，浮点，复数，string 常量。
如果 二元运算符(除了shift) 的 无类型操作数 是不同类型，结果的类型是 表达式中 最后一个 是整型，rune，浮点，复数 类型的 操作数的类型。
比如，一个无类型整型常量 被除以一个 无类型复数常量， 产生一个 无类型复数常量。

```
const a = 2 + 3.0          // a == 5.0   (untyped floating-point constant)
const b = 15 / 4           // b == 3     (untyped integer constant)
const c = 15 / 4.0         // c == 3.75  (untyped floating-point constant)
const Θ float64 = 3/2      // Θ == 1.0   (type float64, 3/2 is integer division)
const Π float64 = 3/2.     // Π == 1.5   (type float64, 3/2. is float division)
const d = 1 << 3.0         // d == 8     (untyped integer constant)
const e = 1.0 << 3         // e == 8     (untyped integer constant)
const f = int32(1) << 33   // illegal    (constant 8589934592 overflows int32)
const g = float64(2) >> 1  // illegal    (float64(2) is a typed floating-point
constant)
const h = "foo" > "bar"    // h == true  (untyped boolean constant)
const j = true             // j == true  (untyped boolean constant)
const k = 'w' + 1          // k == 'x'   (untyped rune constant)
const l = "hi"             // l == "hi"  (untyped string constant)
const m = string(k)        // m == "x"   (type string)
const Σ = 1 - 0.707i       //            (untyped complex constant)
const Δ = Σ + 2.0e-4       //            (untyped complex constant)
const Φ = iota*1i - 1/1i   //            (untyped complex constant)
```

。。。untyped 和 typed 的区别是什么？ 就是 只知道 是一个 整型，但是是 int32, 64 unsigned 不清楚 ？ 那每个变量 总有一个 类型啊。

应用内置方法 complex 到 无类型int，rune，浮点 整型，生成一个 无类型复数常量
```
const ic = complex(0, c)   // ic == 3.75i  (untyped complex constant)
const iΘ = complex(0, Θ)   // iΘ == 1i     (type complex128)
```

常量表达式 总是 被精确地 evaluate。 中间值 和 常量它们 可能需要 需要 比任何预定义的类型的 更大的精度。
下面是合法的声明：
```
const Huge = 1 << 100         // Huge == 1267650600228229401496703205376  (untyped
```

```
integer constant)
const Four int8 = Huge >> 98  // Four == 4            (type int8)
```

常量的 除法和 取余 都不能是 0.

```
3.14 / 0.0   // illegal: division by zero
```

typed的常量的 值 必须是 该类型 可以表达的

下面是非法的：

```
uint(-1)     // -1 cannot be represented as a uint
int(3.14)    // 3.14 cannot be represented as an int
int64(Huge)  // 12676506002282294014967032053760 cannot be represented as an int64
Four * 300   // operand 300 cannot be represented as an int8 (type of Four)
Four * 100   // product 400 cannot be represented as an int8 (type of Four)
```

一元 bitwise complement operator ^ (位补运算符) 的 mask 匹配规则：mask是全1 for 无符号常量， -1for 有符号 无类型常量。

```
^1         // untyped integer constant, equal to -2
uint8(^1)  // illegal: same as uint8(-2), -2 cannot be represented as a uint8
^uint8(1)  // typed uint8 constant, same as 0xFF ^ uint8(1) = uint8(0xFE)
int8(^1)   // same as int8(-2)
^int8(1)   // same as -1 ^ int8(1) = -2
```

实现限制：一个编译器可能使用 rounding 在计算 无类型浮点 或 实数常量 表达式时。这个 rounding 可能导致 在一个整型上下文中 一个浮点常量表达式 是无效的。

Order of evaluation

在包层级，初始化依赖 决定了 变量声明中 每个初始化表达式 的 evaluate 顺序。
否则 evaluate 表达式的操作符，赋值，return 语句，所有函数调用，方法调用，通信 操作， 是 evaluate 在 词汇的从左到右 顺序。

如，在 方法内部 的赋值语句：

```
y[f()], ok = g(h(), i()+x[j()], <-c), k()
```

方法调用 和 通信的 顺序是： f() h() i() j() <-c k()。 当然，其他事件比较顺序 用来 eval 和 x的取下标 和y的eval 是未定义的。

```
a := 1
f := func() int { a++; return a }
x := []int{a, f()}            // x may be [1, 2] or [2, 2]: evaluation order
between a and f() is not specified
m := map[int]int{a: 1, a: 2}  // m may be {2: 1} or {2: 2}: evaluation order
between the two map assignments is not specified
n := map[int]int{a: f()}      // n may be {2: 3} or {3: 3}: evaluation order
between the key and the value is not specified
```

在包级别，初始化依赖 覆盖了 单独初始化表达式的 left-to-right 规则，但不会覆盖 每个

表达式中的 操作数 的。

```
var a, b, c = f() + v(), g(), sqr(u()) + v()

func f() int        { return c }
func g() int        { return a }
func sqr(x int) int { return x*x }

// functions u and v are independent of all other variables and functions
```

函数调用顺序 u() sqr() v() f() v() g()

。。这是个什么顺序。。

一个单独表达式中的 浮点数运算，根据操作符优先级进行 eval。 明确的括号可以影响eval的顺序。

Statements

statements control execution.

```
Statement =
    Declaration | LabeledStmt | SimpleStmt |
    GoStmt | ReturnStmt | BreakStmt | ContinueStmt | GotoStmt |
    FallthroughStmt | Block | IfStmt | SwitchStmt | SelectStmt | ForStmt |
    DeferStmt .

SimpleStmt = EmptyStmt | ExpressionStmt | SendStmt | IncDecStmt | Assignment |
ShortVarDecl .
```

Terminating statements

一个 终止语句，阻止同一个block中 词汇上 出现在 后面的 所有的语句的 执行。
以下语句的后续语句被终止：

- return 或 go 语句
- 调用 内置方法 panic
- A block in which the statement list ends in a terminating statement. 。。感觉好像是 后续的 内嵌的 block 不执行。
- if 语句且满足下列条件：
  - 存在else分支，且 2个分支(if,else) 都是 终止语句。
- for语句且满足下列条件：
  - 没有指向for语句的 break语句， 且 不存在loop条件。。。。 死循环啊。。也算终止？
- switch语句 且满足条件：
  - 没有break语句 指向 switch语句， 有一个default分支， 所有分支(包括default)都终止于一个 终止语句，或 一个可能被标记为"fallthrough"的语句。
- select语句 且满足：
  - 没有 指向select 的break， 每个分支的语句(包括default) 终止于一个 终止语

句。
一个 labeled statement 标记了一个 终止语句。

所有其他语句 都是非 终止的。

一个语句list 是一个 终止语句，如果 list非空，且最终非空语句是终止的。

Empty statements
空语句不做任何事情。
EmptyStmt = .
。。最后一个. 不是空语句的一部分，， 实际上就一个 空白。

Labeled statements
一个标记语句 可能 是 goto，break，continue 语句的 目标。

```
LabeledStmt = Label ":" Statement .
Label       = identifier .
```

```
Error: log.Panic("error encountered")
```

Expression statements
With the exception of specific built-in functions, function and method calls and receive operations can appear in statement context.
除了特定的内置函数外， 函数，方法调用，接受操作 都可以出现在 语句中。

```
ExpressionStmt = Expression .
```

下列内置函数，不能出现在 语句上下文中：
append cap complex imag len make new real
unsafe.Add unsafe.Alignof unsafe.Offsetof unsafe.Sizeof unsafe.Slice

```
h(x+y)
f.Close()
<-ch
(<-ch)
len("foo")  // illegal if len is the built-in function
```

Send statements

一个send语句 发送一个值到 channel。channel 表达式 必须是 channel类型，channel direction必须允许 send操作， 发送的值的类型必须能转为 channel的元素类型。

```
SendStmt = Channel "<-" Expression .
Channel  = Expression .
```

channel 和 值表达式 在 通信开始前 被eval。通信阻塞知道 send可以执行。
一个没有buffer的channel的send操作可以执行 如果 receiver准备好。

一个有buffer的channel的send可以执行，如果buffer有空闲空间
在一个已关闭的channel上 执行send 会 rt-p。
在nil的channel上 send，block forever
。。nil / closeed channel 上 读取呢？

```
ch <- 3  // send value 3 to channel ch
```

IncDec statements
++, -- 语句 增加/减少 它们的操作数 一个 无类型的常量1。
在一个 赋值语句中，操作数必须是 可寻址的，或 一个 map index expression (映射索引表达式)

```
IncDecStmt = Expression ( "++" | "--" ) .
```

下面的 左右2列 语义上相等。

```
x++                    x += 1
x--                    x -= 1
```

。。没有前置++，--， 那么 x+x++ 后一个 是先计算+。。不，我记得前面有说， 一元运算符的优先级 最高。

Assignments

```
Assignment = ExpressionList assign_op ExpressionList .

assign_op = [ add_op | mul_op ] "=" .
```

。。这个能 a=b=c=2 吗？不。 上面的 =后面 是定义，前面是 定义的名字，不是定义。
。。可以多赋值。 这个是 list = list 赋值的。

每个左值都必须是 可寻址的，一个map index expression 或 空白标识符(空白只在 = 赋值中可以使用)。 操作数 可以用 括号包围。

```
x = 1
*p = f()
a[i] = 23
(k) = <-ch  // same as: k = <-ch
```

一个赋值操作 x op= y 是一个二元算术操作符， 等同于 x = x op (y) 但只eval x 一次。op= 结构 是 single token (单独象征？。。结合后面，估计是说 2侧都是单个的。)。在赋值操作中， 左值和右值 表达值列表 必须包含 确切的一个 单值表达式，左值表达式必须不能是 空白标识符

```
a[i] <<= 2
i &^= 1<<n
```

一个 tuple(元组) 赋值，分配 多值操作的每个元素 到 变量列表中。
有2种方式，

第一个，右值是一个 单独的 多值表达式，如 函数调用，channel，map操作，或 type断言。
左值的操作数数量 必须 匹配 值的数量。例如，如果f 是一个函数，返回2个值：
x, y = f()
分派 第一个值给x，第二个值给y。

第二种， 左值的数量 必须等于 右边 表达式的数量，每个表达式 必须是单值的， 左右两侧按序匹配。
one, two, three = '一', '二', '三'

。。。我在想，老外第一次看到 一 二 三 会认为是一种符号吧。。等一个 万。

空白标识符 提供一种 忽略 右值 的方法

```
_ = x       // evaluate x but ignore it
x, _ = f()  // evaluate f() but ignore second result value
```

赋值的执行有2个阶段。
第一，左侧的 索引表达式 和 指针反引用(包括在selector中隐式的指针反引用) 的操作数 和 右侧的表达式 都eval 在 普通的顺序。
第二，分配按照 从左到右 的 顺序执行。

。。就是 左侧是目标，先把 左侧 反引用出来，获得 变量。 然后 计算右侧的 结果。 然后赋值 给 左侧， 所以能 交换ab， 还有就是 a,b = b+1, a+1 . 这种 不会有二义。

```
a, b = b, a  // exchange a and b

x := []int{1, 2, 3}
i := 0
i, x[i] = 1, 2  // set i = 1, x[0] = 2

i = 0
x[i], i = 2, 1  // set x[0] = 2, i = 1

x[0], x[0] = 1, 2  // set x[0] = 1, then x[0] = 2 (so x[0] == 2 at end)

x[1], x[3] = 4, 5  // set x[1] = 4, then panic setting x[3] = 5.

type Point struct { x, y int }
var p *Point
x[2], p.x = 6, 7  // set x[2] = 6, then panic setting p.x = 7
```
。。这个 p.x 为什么 panic？难道说 p是空的？ 就是 var p *Point是 声明指针，并没有初始化？

```
i = 2
x = []int{3, 5, 7}
for i, x[i] = range x {  // set i, x[2] = 0, x[0]
    break
}
// after this loop, i == 0 and x == []int{3, 5, 3}
```

。。这个是什么操作。。

在赋值中，每个值都必须 能转换到 操作数的类型，下面是特例：

任何类型的值都可以赋给 空白标识符

如果一个无类型常量 被赋给 一个 接口类型的变量 或 空白标识符，常量首先被隐式转为 它的 默认类型。

如果一个无类型布尔值 被赋给 一个接口类型的变量或空白标识符，它首先被隐式转为bool类型。

If statements

if语句指定了 2个分支的有条件的执行 通过 一个布尔表达式的值。
如果表达式是true， if分支被执行，否则，如果存在else分支，则执行else分支。

```
IfStmt = "if" [ SimpleStmt ";" ] Expression Block [ "else" ( IfStmt | Block ) ] .
```

```
if x > max {
    x = max
}
```

表达式 前面可以放一个 简单语句，简单语句在 表达式之前被执行。

```
if x := f(); x < y {
    return x
} else if x > z {
    return z
} else {
    return y
}
```

Switch statements

提供了 多路 执行。 一个表达式或类型 用于比较， 来决定 走哪个case分支。

```
SwitchStmt = ExprSwitchStmt | TypeSwitchStmt .
```

有2种形式，expression switch 和 type switch。
表达式switch中 case包含表达式，case中的表达式 用于和 switch的表达式进行比较。
类型switch中，case 包含type， 这些type 用于和 特殊注解的switch表达式的类型 比较。
在一个switch语句中， switch的表达式 只计算一次。

Expression switches

在一个表达式 switch中，switch表达式被 eval，且 case表达式(不是必须常量的) 被计算按照 从左到右，从上到下的 顺序。 第一个等于 switch表达式的 case 执行代码，其他case跳过。 如果没有case匹配，且有一个default case，那么执行default。 最多一个default，它可以出现在 switch语句的任何地方。 不写switch表达式那么 就是bool的true。

```
ExprSwitchStmt = "switch" [ SimpleStmt ";" ] [ Expression ] "{" { ExprCaseClause } "}" .
ExprCaseClause = ExprSwitchCase ":" StatementList .
ExprSwitchCase = "case" ExpressionList | "default" .
```

如果 switch表达式 eval 出一个 无类型常量， 它首先被隐式转为 它默认的类型。
nil不能用作一个 switch 表达式。
switch 表达式 的 类型 必须是 可比较的。
。。。默认类型是什么。。

如果case 表达式是无类型的，它 首先被隐式转为 switch表达式的类型。 对于每个case 表达式x 和 switch表达式的值t， x==t 必须是一个有效的比较。

换句话说，switch表达式 被对待，就像 它不使用 明确的类型 来声明和初始化一个 临时变量t。 t然后和 case表达式x 测试 是否相等。

在一个 case 或default 分支， 最后一个语句可能是 (可能是labeled) "fallthrough"语句来 indicate(表明) 控制 应该流动，从这个分支的末尾 到下个分支的开始。否则控制会直接转到 switch的结尾。 fallthrough 语句可以 出现在 任何一个分支的 最后，除了最后一个分支。
。。就是 默认是 break的， 想继续执行下一个case就 fallthrough

在switch表达式前面可能有一个 简单语句， 这个简单语句在 表达式计算前 执行。

```
switch tag {
default: s3()
case 0, 1, 2, 3: s1()
case 4, 5, 6, 7: s2()
}

switch x := f(); {  // missing switch expression means "true"
case x < 0: return -x
default: return x
}

switch {
case x < y: f1()
case x < z: f2()
case x == 4: f3()
}
```

实现限制：编译器可能 不允许 多个case表达式 eval 出相同的常量。
例如，现在的编译器 不允许 重复的 int，浮点，string 在case表达式中。

Type switches
type switch 比较type 而不是 value，其他和 表达式switch 类似。
它被mark 通过 一个特别的 switch表达式，这个表达式有一个 type断言(assertion..断言，声明)，而不是一个真实的type。

```
switch x.(type) {
// cases
}
```

case 会匹配 真实的类型 T， 而不是 表达式x的动态类型。
就type assertion而言， x必须是 接口类型，case中的列举的 每个非接口类型T 必须是 x的实现。 每个case中列举的 类型 必须都不同。

```
TypeSwitchStmt  = "switch" [ SimpleStmt ";" ] TypeSwitchGuard "{"
{ TypeCaseClause } "}" .
TypeSwitchGuard = [ identifier ":=" ] PrimaryExpr "." "(" "type" ")" .
TypeCaseClause  = TypeSwitchCase ":" StatementList .
TypeSwitchCase  = "case" TypeList | "default" .
TypeList        = Type { "," Type } .
```

TypeSwitchGuard 可能包含一个 短变量声明，当这种格式使用时，每个分支的implicit block的 TypeSwitchCase 的最后 声明一个 变量。 分支的case 只列举一个类型，变量是那个类型，否则，变量是 TypeSwitchGuard 的表达式的类型。

case可能使用nil 代替类型， 当TypeSwitchGuard的 表达式 是 nil接口值时，选择这个case。 最多只有一个nil case。

给出一个表达式x， 类型是 interface{}：

```
switch i := x.(type) {
case nil:
    printString("x is nil")         // type of i is type of x (interface{})
case int:
    printInt(i)                    // type of i is int
case float64:
    printFloat64(i)                // type of i is float64
case func(int) float64:
    printFunction(i)            // type of i is func(int) float64
case bool, string:
    printString("type is bool or string")// type of i is type of x (interface{})
default:
    printString("don't know the type") // type of i is type of x (interface{})
}
```

能被重写为：

```
v := x  // x is evaluated exactly once
if v == nil {
    i := v                // type of i is type of x (interface{})
    printString("x is nil")
} else if i, isInt := v.(int); isInt {
    printInt(i)                     // type of i is int
} else if i, isFloat64 := v.(float64); isFloat64 {
    printFloat64(i)                    // type of i is float64
} else if i, isFunc := v.(func(int) float64); isFunc {
    printFunction(i)               // type of i is func(int) float64
} else {
    _, isBool := v.(bool)
    _, isString := v.(string)
    if isBool || isString {
```

```
        i := v                  // type of i is type of x (interface{})
        printString("type is bool or string")
    } else {
        i := v                  // type of i is type of x (interface{})
        printString("don't know the type")
    }
}
```

在 type switch guard 的前面可能有一个 简单语句，这个会在 guard eval前 执行。

type switch 中 不允许 fallthrough

For statements

for 语句定义了 一个block 的重复执行，只要 一个bool条件 eval 出true。条件在每次iteration前 eval。 如果条件不存在，那么就是 true。

```
for a < b {
    a *= 2
}
```

。。有点简陋。。

For statements with for clause

for语句 with 一个 ForClause 也被 它的条件 控制，但是 它可以定义 一个init，一个post语句，比如一个赋值，一个递增/递减 语句。init语句可以是一个短变量声明，但是 post语句不可以。 init语句中声明的变量在 每次iteration中 能被重用。

```
ForClause = [ InitStmt ] ";" [ Condition ] ";" [ PostStmt ] .
InitStmt = SimpleStmt .
PostStmt = SimpleStmt .

for i := 0; i < 10; i++ {
    f(i)
}
```

init语句在第一次迭代的 eval 条件前 执行一次。
post语句在每次 block执行完 后 执行。
ForClause 的任何元素都可以被省略，但是 ； 不能省略，除非只有一个条件。 如果condition省略，那么就是true。

```
for cond { S() }    is the same as    for ; cond ; { S() }
for      { S() }    is the same as    for true     { S() }
```

For statements with range clause

for语句with range clause 遍历 数组，切片，string，map或 从channel收到的values 的所有 实体。 对每个实体 它 赋值 遍历值 到对应的 存在的遍历变量，然后执行block。

```
RangeClause = [ ExpressionList "=" | IdentifierList ":=" ] "range" Expression .
```

"range" 右侧的表达式 被称为 range expression，这个可能是一个 数组，数组的指针，切片，string，map，允许receive操作的channel。
作为一个赋值，左侧存在的操作数 必须是可寻址的 或者 映射下标表达式， 它们代表了 iteration variable (遍历变量，迭代变量)。
如果 range表达式 是channel，那么允许 最多一个 遍历变量。 其他情况，可以最多有2个。
如果最后一个 迭代变量 是 空白标识符， range clause 等同于 没有那个变量的 range clause.

range expression x 被eval 一次，在loop开始前、除了一个例外：如果最多只有一个 迭代变量，且 len(x) 是 常量， range expression 不会被 eval。

左侧的方法调用，在每次迭代时都 eval 一次。对于每次迭代，迭代值 按下面的规则 产生，如果 各自的迭代变量 是存在的：

Range expression			1st value			2nd value	
array or slice	a	[n]E, *[n]E, or []E	index	i	int	a[i]	E
string	s	string type	index	i	int	see below	rune
map	m	map[K]V	key	k	K	m[k]	V
channel	c	chan E, <-chan E	element	e	E		

对于数组，指向数组的指针，切片值 a，下标迭代值 是递增的，从0开始。如果最多只有一个迭代变量，range loop 提供了 迭代值 从0带 len(a)-1， 不会 索引到 数组或切片自身。对于nil切片，迭代的数量是0。

对于string值，range clause 从第0个 byte 开始遍历，遍历 所有 Unicode。在 successive(连续的，相继的) 迭代中，下标值 是 string中 连续的 utf-8编码的 编码点 的第一个byte， 第二个值的类型是rune，是相应的 code point。
如果 迭代遇到 一个非法的 utf-8 序列，第二个值 会是 0xFFFD, unicode replacement character，下次迭代会 前进 一个single bit。

迭代map的 顺序是未知的，2次迭代不能保证一样。 如果 迭代期间，map的entry 在还没有reach的时候 被删除，相应的迭代值 不会生成(If a map entry that has not yet been reached is removed during iteration, the corresponding iteration value will not be produced.)。 如果迭代期间，生成了一个map entry，那么这个entry 可能被迭代，也可能不会。 如果map是nil，迭代数量是0。

channel，迭代值 是 successive 被发送到channel的值，直到channel close。如果 channel 是nil，range expression 永远阻塞。

range clause 中的 迭代变量的 声明 可能使用 短变量声明的形式。这种情况下，它们的类型被 设置为 各自的迭代值 的类型，它们的作用域 是for 语句块。 在每次迭代中重用。
如果迭代变量 是在 for语句外部 声明的， 那么执行后，它们的值会是 最后一次迭代的值。

```
var testdata *struct {
    a *[7]int
}
```

。。。。太。。 一个含有7个int的数组的指针， 7个int数值的指针，7个int指针，int指针

的数组。。。 应该是 长度为7的数组 的指针。

```
for i, _ := range testdata.a {
    // testdata.a is never evaluated; len(testdata.a) is constant
    // i ranges from 0 to 6
    f(i)
}

var a [10]string
for i, s := range a {
    // type of i is int
    // type of s is string
    // s == a[i]
    g(i, s)
}

var key string
var val interface{}  // element type of m is assignable to val
m := map[string]int{"mon":0, "tue":1, "wed":2, "thu":3, "fri":4, "sat":5, "sun":6}
for key, val = range m {
    h(key, val)
}
// key == last map key encountered in iteration
// val == map[key]

var ch chan Work = producer()
for w := range ch {
    doWork(w)
}

// empty a channel
for range ch {}
```

Go statements

go 语句 开始执行 一个 函数调用，作为一个 独立控制的线程 或 goroutine， 使用相同的地址空间。

GoStmt = "go" Expression .

表达式必须是 函数或方法调用，不能用括号包围， 调用内置函数 是受限制的，限制见 expression statements。(

下面是 不能调用的 内置方法。

```
append cap complex imag len make new real
unsafe.Add unsafe.Alignof unsafe.Offsetof unsafe.Sizeof unsafe.Slice
)
```

在goroutine调用中，函数值 和 参数 的eval 和平常一样， 但是和普通call不同的是，(主/调用者)程序执行 不会等待 被invoke的函数 执行完。反而，函数在一个新的 goroutine中 独立地执行。当函数终止，它的goroutine也终止。如果函数有任何的返回值，当函数结束时，它们被丢弃。

```
go Server()
go func(ch chan<- bool) { for { sleep(10); ch <- true }} (c)
```

Select statements

select语句选择 send或receive 操作 集合中的 哪个操作 被执行。 类似switch，但这里的case 都指向了 通信操作。

```
SelectStmt = "select" "{" { CommClause } "}" .
CommClause = CommCase ":" StatementList .
CommCase   = "case" ( SendStmt | RecvStmt ) | "default" .
RecvStmt   = [ ExpressionList "=" | IdentifierList ":=" ] RecvExpr .
RecvExpr   = Expression .
```

一个有 RecvStmt 的分支 分配 RecvExpr的结果 到一个或2个 变量，这些变量会通过 短变量声明 来 声明。RecvExpr必须是一个(可能被括号包围的) receive 操作。 最多有一个default分支，并且可以出现在 case列表的任何地方。

select语句的执行分为以下数步：

对于语句中的所有分支， receive操作的channel操作数 和 channel 和 send语句的右值表达式 都被 eval exactly 一次，按照source order，在进入select时。结果是 用于 接收和 发送 的 channel的集合，和对应的会发送的值。 无论是哪个通讯操作被执行，所有 在 eval中的 副作用(side effect)都会发生。

如果有1个或多个通信可以被执行， 只会有一个被执行， 执行的通信 通过 伪随机 来选择。否则，如果有default分支，那么就执行这个。 如果没有default，select会阻塞，直到 至少有一个 通信可以执行。

除非选择的case是default，否则 各自(相应)的通信操作被执行。(Unless the selected case is the default case, the respective communication operation is executed.)

如果选择的case是 有一个 短变量声明或赋值 的 RecvStmt，左值表达式被eval，且 接收到的 值 被赋值(到变量)。

被选择的case 的 语句列表 被执行。

由于 在nil channel上 通信 不会被执行，所以 一个只有 nil channel 且 没有default 分支 的 select 会被永远block。

```
var a []int
var c, c1, c2, c3, c4 chan int
var i1, i2 int
select {
case i1 = <-c1:
    print("received ", i1, " from c1\n")
case c2 <- i2:
    print("sent ", i2, " to c2\n")
case i3, ok := (<-c3):  // same as: i3, ok := <-c3
    if ok {
        print("received ", i3, " from c3\n")
    } else {
```

```
            print("c3 is closed\n")
        }
    case a[f()] = <-c4:
        // same as:
        // case t := <-c4
        //    a[f()] = t
    default:
        print("no communication\n")
    }

    for {  // send random sequence of bits to c
        select {
        case c <- 0:  // note: no statement, no fallthrough, no folding of cases
        case c <- 1:
        }
    }

    select {}  // block forever
```

Return statements

函数F中的 return语句，终止了F的执行，return可以提供1个或多个返回值。
F中 defer(一个关键字) 的函数 在F返回之前 执行。

```
ReturnStmt = "return" [ ExpressionList ] .
```

如果函数没有申明 result type， return 不能有任何返回值。

3种方法 从 一个有return type 的函数 返回 值：

返回 值 必须显式 列举在return 语句中，每个表达式必须是单值的 并且可以赋值给 对应的 函数返回类型。

```
func simpleF() int {
    return 2
}

func complexF1() (re float64, im float64) {
    return -7.0, -4.0
}
```

return语句中的 表达式列表 可能是 一个 单独的 调用 ，调用了一个多值函数。这种就像 把多值函数的结果 赋值给 临时变量，然后 return 这些临时变量。

```
func complexF2() (re float64, im float64) {
    return complexF1()
}
```

如果函数的 返回类型 指明了 返回参数名字，那么 return的表达式列表 可以是空的。返回参数名 就像普通本地变量，函数 会赋值给它们 as necessory， return会返回这些变量的值。

```
func complexF3() (re float64, im float64) {
    re = 7.0
    im = 4.0
    return
}

func (devnull) Write(p []byte) (n int, _ error) {
    n = len(p)
    return
}
```

不管它们如何被声明，所有的返回值 在程序进入函数时 都被初始化为 它们类型的 0值。
return语句指定 返回结果集， 在任何 defer 函数执行前。
。。。上面说 F会在 defer 执行完后 返回调用者， 这里说 return指定所有结果在 defer之前。。。 那么 defer 就是在 方法已经有返回结果 之后， 真正返回之前 执行？

实现限制：编译器可能不允许 return语句的 空的表达式列表， 如果 在return所处的 作用域范围内 有一个 同名的 不同实体(常量，类型，变量)。

```
func f(n int) (res int, err error) {
    if _, err := f(n-1); err != nil {
        return  // invalid return statement: err is shadowed
    }
    return
}
```

。。之前说 所有的返回值 在进入方法时 会被设置为 对应类型的0值， 并且也说过 返回参数 可以看做 普通local变量， 所以 返回参数列表中的 err 和 if的 err 是2个 同名 同类型的 不同对象。

Break statements

break语句 终止 最内层 的 for switch select 语句的执行 在同一个函数中。

```
BreakStmt = "break" [ Label ] .
```

如果有 Label， 它必须是 一个封闭的 for switch select 语句，

```
OuterLoop:
    for i = 0; i < n; i++ {
        for j = 0; j < m; j++ {
            switch a[i][j] {
            case nil:
                state = Error
                break OuterLoop
            case item:
                state = Found
                break OuterLoop
            }
        }
    }
```

Continue statements

continue语句 开始最内层的 for 循环的 下一次迭代 从for的post语句开始。 for 必须是在同一个函数内。

```
ContinueStmt = "continue" [ Label ] .
```

如果有 Label， 它必须是一个 封闭的 for循环。

```
RowLoop:
    for y, row := range rows {
        for x, data := range row {
            if data == endOfRow {
                continue RowLoop
            }
            row[x] = data + bias(x, y)
        }
    }
```

Goto statements

goto语句转移 控制 到 同一个函数中 相应的 label的 代码。

```
GotoStmt = "goto" Label .
```

```
goto Error
```

执行goto，不能造成 任何变量 进入 作用域，这些变量 在goto点时 没有进入作用域。
Executing the "goto" statement must not cause any variables to come into scope that were not already in scope at the point of the goto.
。。应该就是说 goto不能跳过 创建。

```
        goto L  // BAD
        v := 3
L:
```

是错误的，因为 跳到L 会导致 跳过 v的create。

block外的goto不能跳到 block内。

```
if n%2 == 1 {
    goto L1
}
for n > 0 {
    f()
    n--
L1:
    f()
    n--
}
```

是错的，因为 L1 在 for的block内，但是 goto不在 这个block内。

Fallthrough statements

fallthrought 在 switch语句中 转移控制 到 下一个case分支的 首个代码块。 它只能用做这个 分支的 最后一个语句。

FallthroughStmt = "fallthrough" .

Defer statements

defer语句 调用 一个方法，这个方法的执行 会推迟到 它的外层方法 返回 的时候，返回可能是因为 外层方法执行了一个return语句，到达函数体最后，或者因为相应的goroutine 是panicking(估计是指发生异常，但是不清楚 goroutine是什么。这里是指 外层方法发生异常而返回？)。

DeferStmt = "defer" Expression .

表达式必须是 一个 函数或方法 调用。 不能被括号包围， 调用部分内置方法会受到限制。

每次当一个defer语句执行时，用于call的 函数值和参数 都被 就像平常一样的eval 并且重新保存(saved anew)，但是 真正的函数 没有被调用。 反而，defer的函数 在 surrounding 的函数 返回前 被 立刻调用，按照它们 defer的 反序。
即，如果外层的函数通过一个 显式的return语句 返回， defer的函数 在 所有返回参数被return语句设置后 ，在函数返回 调用者之前 执行。 如果函数值 eval出 nil(函数也是一等成员，可能是nil)，那么 在调用时 发生panic，而不是 defer语句执行时。

例如，如果 defer的函数 是一个 函数字面量，外层函数存在 有名字的返回参数，这些参数在作用范围内，以字面量的形式， defer的函数可能 访问和修改 返回参数，在它们返回前。如果 defer的函数有 任何的返回值，在函数(不清楚是外层函数还是defer的函数)执行完时，都会被丢弃。

```
lock(l)
defer unlock(l)  // unlocking happens before surrounding function returns

// prints 3 2 1 0 before surrounding function returns
for i := 0; i <= 3; i++ {
    defer fmt.Print(i)
}

// f returns 42
func f() (result int) {
    defer func() {
        // result is accessed after it was set to 6 by the return statement
        result *= 7
    }()
    return 6
}
```

。。。有点。。。

Built-in functions

内置函数是预定义的， 它们可以像其他函数一样被调用，但是 有一些内置函数 接受 类型 而不是 表达式 作为 第一个参数。

内置函数 没有标准的 go类型， 所以它们只能出现在 调用语句中， 不能作为 function value。
。。只能被调用，不能传递。

Close

对于channel c， 内置方法 close(c) 记录：不会再有值 通过channel 发送。 如果c是receive-only的channel，就发生错误。 send或关闭 一个 已经关闭的channel 导致一个 rt-p，关闭 nil channel 也是rt-p。
在调用close，且 先前发送的value 已经被receive 后， receive操作会返回 channel的元素类型的 0值，而不是 block。
多值receive 操作(就是指 返回值+bool) 返回一个 收到的值 和 一个channel是否已经关闭的 指示。

Length and capacity

内置函数 len 和 cap 接收 各种类型的实参，返回一个 int。 它的实现 确保 返回值总是 fit into(纳入，归属) 一个int。

Call	Argument type	Result
len(s)	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(s)	[n]T, *[n]T	array length (== n)
	[]T	slice capacity
	chan T	channel buffer capacity

切片的 容量 是 underlying 数组中的 元素个数。任何时候，下列关系成立：
0 <= len(s) <= cap(s)

nil的切片，map，channel 的 长度是 0。
nil的切片，channel 的容量是0.

如果s 是string常量，表达式 len(s) 是常量。如果s的类型是 数组 或 指向数组的指针，并且s不包含 channel receive 或 (非常量的)function call， 那么 len(s),cap(s) 是常量；这个(些)例子中，s没有被eval。

否则，len 和cap 的 调用 不是 常量，且 s会被 eval。

```
const (
    c1 = imag(2i)                    // imag(2i) = 2.0 is a constant
    c2 = len([10]float64{2})         // [10]float64{2} contains no function calls
    c3 = len([10]float64{c1})        // [10]float64{c1} contains no function
    calls
    c4 = len([10]float64{imag(2i)})  // imag(2i) is a constant and no function
    call is issued
    c5 = len([10]float64{imag(z)})   // invalid: imag(z) is a (non-constant)
    function call
)
var z complex128
```

Allocation 分配

内置函数 new ，接受一个 类型T，分配 存储空间 给 类型T的变量，返回一个 *T类型的指针 指向空间。 变量初始化 就像 initial values 中描述的那样。

```
new(T)

type S struct { a int; b float64 }
new(S)
```

分配空间给类型S的一个变量，初始化它(a=0, b=0.0) 返回一个*S 类型的值，这个值保存了那块空间的地址。

Making slices, maps and channels

内置函数 make 接受类型T，T可以是 slice，map，channel类型， 可选的 后面跟随一个 类型确定的 表达式列表。 它会返回 一个 T类型的 值 (不是 *T)， 内存就像 initial values 中描述的那样 初始化。

Call	Type T	Result
make(T, n)	slice	slice of type T with length n and capacity n
make(T, n, m)	slice	slice of type T with length n and capacity m
make(T)	map	map of type T
make(T, n)	map	map of type T with initial space for approximately n elements
make(T)	channel	unbuffered channel of type T
make(T, n)	channel	buffered channel of type T, buffer size n

size参数，m和n，都必须是 整型类型 或一个 无类型常量。一个常量size参数 必须是 非负和 代表了一个 int类型的值。如果是无符合常量，那么它的类型是int。
如果n和m 都被提供了 且都是常量，那么 n必须 <= m。如果运行时，n是负数 或 n大于m，则

rt-p。
。。n在前面 m在后面。

```
s := make([]int, 10, 100)       // slice with len(s) == 10, cap(s) == 100
s := make([]int, 1e3)           // slice with len(s) == cap(s) == 1000
s := make([]int, 1<<63)         // illegal: len(s) is not representable by a value
of type int
s := make([]int, 10, 0)         // illegal: len(s) > cap(s)
c := make(chan int, 10)         // channel with a buffer size of 10
m := make(map[string]int, 100)  // map with initial space for approximately 100
elements
```

使用 map类型 和 size hint n 来调用 make ， 会创建 一个 map with 初始空间能hold n个 map elements。 精确的行为 取决于 实现。

Appending to and copying slices

内置的 append 和 copy 协助(assist in) 普通slice操作。 对于这2个函数，结果和 实参引用的内存 是否 重叠 无关。(For both functions, the result is independent of whether the memory referenced by the arguments overlaps.)

可变长参数 函数 "append" 追加 0或多个值 x 到类型S的 s，S必须是一个 切片类型， 返回 处理后的切片， 也是S类型。 值x 传递到 一个 类型是 ...T 的参数， T是 S的元素类型，应用各自的参数传递规则。 有一个特例，"append" 也接受 一个[]byte 做为第一个参数，...string 作为第二个参数， 这种会 追加 string的byte 到 []byte。

```
append(s S, x ...T) S  // T is the element type of S
```

如果s的capacity 不够大 以放下所有的 需要追加的值，"append" 会 分配一个 新的，足够大的 unrderlying 数组，来保存 现有的切片元素 和 追加的值。 否则，重用underlying 数组。

```
s0 := []int{0, 0}
s1 := append(s0, 2)               // append a single element     s1 == []int{0, 0, 2}
s2 := append(s1, 3, 5, 7)         // append multiple elements    s2 == []int{0, 0, 2, 3,
5, 7}
s3 := append(s2, s0...)           // append a slice              s3 == []int{0, 0, 2, 3,
5, 7, 0, 0}
s4 := append(s3[3:6], s3[2:]...)  // append overlapping slice    s4 == []int{3, 5, 7, 2,
3, 5, 7, 0, 0}

var t []interface{}
t = append(t, 42, 3.1415, "foo")  //                            t == []interface{}{42,
3.1415, "foo"}

var b []byte
b = append(b, "bar"...)           // append string contents      b == []byte{'b', 'a',
'r' }
```

函数 "copy" 从 源 复制 切片元素 到目标，返回 复制的 元素个数。 2个参数 必须有相同的元素

类型T，必须都能 赋值到 一个类型为[]T 的 切片。
复制的元素的个数 是 min(len(src), len(dst))。
特例，"copy" 也可以接受一个 可以赋值给[]byte类型的 目标， source是一个 string。 这种形式会 从string 拷贝它的byte 到 byte切片。

```
copy(dst, src []T) int
copy(dst []byte, src string) int
```
。。目标在前面。

```
var a = [...]int{0, 1, 2, 3, 4, 5, 6, 7}
var s = make([]int, 6)
var b = make([]byte, 5)
n1 := copy(s, a[0:])            // n1 == 6, s == []int{0, 1, 2, 3, 4, 5}
n2 := copy(s, s[2:])            // n2 == 4, s == []int{2, 3, 4, 5, 4, 5}
n3 := copy(b, "Hello, World!")  // n3 == 5, b == []byte("Hello")
```

Deletion of map elements

内置函数delete 从map m中移除 关键字为 k 的元素。k的类型必须能赋给 m的 关键字的类型

```
delete(m, k)  // remove element m[k] from map m
```

如果m 是nil， 或 m[k] 不存在， delete 是no-op

Manipulating complex numbers

3个函数 来 assemble(装配) 和 disassemble(拆卸) 复数。
内置函数 complex 构造一个复数值 从 浮点数 的 real 和 imaginary 部分。

```
complex(realPart, imaginaryPart floatT) complexT
real(complexT) floatT
imag(complexT) floatT
```
。。T代表了空，32+64 64+128

实参和 返回值的类型 是对应的， 对于 complex，2个实参 必须是 浮点数类型，返回类型是complex类型。 complex64对应float32， complex128对应float64.
如果一个实参 eval出 无类型常量，它首先被隐式转为 另一个参数的类型。
如果2个实参 都eval 出 无类型常量，它们必须是 非complex数字 或它们的 虚部必须是0，返回值 是一个 无类型的 复数常量。

对于 real 和imag， 实参必须是 复数类型， 返回值是对应的 浮点数类型， float32对应complex64，float64对应complex128。如果实参eval一个无类型常量，它必须是一个数字，返回值是 无类型浮点常量。

real 和 imag 方法 合并在一起，inverse 复数。
比如一个 复数类型Z 的值 z， z == Z(complex(real(z), imag(z))).

如果这些函数的 操作数 都是常量， 返回值也是常量。

```
var a = complex(2, -2)             // complex128
const b = complex(1.0, -1.4)       // untyped complex constant 1 - 1.4i
```

```
x := float32(math.Cos(math.Pi/2))  // float32
var c64 = complex(5, -x)           // complex64
var s int = complex(1, 0)          // untyped complex constant 1 + 0i can be
converted to int
_ = complex(1, 2<<s)               // illegal: 2 assumes floating-point type,
cannot shift
var rl = real(c64)                 // float32
var im = imag(a)                   // float64
const c = imag(b)                  // untyped constant -1.4
_ = imag(3 << s)                   // illegal: 3 assumes complex type, cannot
shift
```

Handling panics

2个内置函数 panic revocer， 协助 报告 和 处理 rt-p 和 程序定义的错误condition(状况)

```
func panic(interface{})
func recover() interface{}
```

当执行一个函数F，一个显示调用 panic 或 一个rt-p 中断了F的执行。 任何F defer的函数会 then执行 如同平常那样。 然后，任何defer函数 run by F的调用者 run，直到 任何 defer by最高层 函数 到 executing goroutine。那时，程序中断，然后错误状况 报告，包含panic的参数的值， 这个中断序列称为 panicking
。。就是 有panic 后， 会执行 本函数中defer的函数，然后 调用者中 defer的函数，调用者的调用者的 defer的函数。。。 然后程序中断，报告错误。
。。。这里有try 这种吗？ 不然 panic毫无意义啊。 估计是一路向外执行 defer 到 第一次碰到的 try。

```
panic(42)
panic("unreachable")
panic(Error("cannot parse"))
```

recover函数 允许 程序管理 panicking goroutine 的行为
假设(suppose) 函数G defer 函数D，D调用 recover， 当G执行时，在相同的goroutine中的一个函数中 发生了一个panic。 当执行defer函数 执行到 D时，D 调用revocer 的结果 会作为 值 传递到 panic的调用。如果D正常返回，没有开始一个新的panic，panicking sequence停止。 在那个case中，函数调用的 状态 between G 和 调用panic 之间 被丢弃， 正常执行resume(重新开始，继续)。任何G defer的 在D之前 的函数 then run and G的执行停止 by返回到它的调用者。
The recover function allows a program to manage behavior of a panicking goroutine. Suppose a function G defers a function D that calls recover and a panic occurs in a function on the same goroutine in which G is executing. When the running of deferred functions reaches D, the return value of D's call to recover will be the value passed to the call of panic. If D returns normally, without starting a new panic, the panicking sequence stops. In that case, the state of functions called between G and the call to panic is discarded, and normal execution resumes. Any functions deferred by G before D are then run and G's execution terminates by

returning to its caller.

。。记得前面说过 defer 的顺序， 执行的时候 是 反序？ 是的 in the reverse order they were deferred ， defer的顺序的反序。
。。revocer的结果到底是 传给 panic 还是 正常执行？ 这个是 我决定的？怎么搞。

如果有下列情况，那么 recover 返回 nil：
panic的参数 是nil
goroutine 不是 panicking
recover 不是被 defered函数 直接调用的。

下面例子中的 protect 函数 调用 函数参数 g， 保护 caller 免受 g触发的 rt-p。

```
func protect(g func()) {
    defer func() {
        log.Println("done")  // Println executes normally even if there is a
        panic
        if x := recover(); x != nil {
            log.Printf("run time panic: %v", x)
        }
    }()
    log.Println("start")
    g()
}
```

。。是不是 defer 的 函数 没有 重新 panic() 所以 就认为是普通结束， 如果 重新 panic 了 就 是 异常 抛到 caller。。。
。。可繁可简 啊。

Bootstrapping

当前实现 提供了 一些有用的内置函数 during bootstrapping。
These functions are documented for completeness but are not guaranteed to stay in the language. They do not return a result.
。。只是为了完整，所以展示下， 不保证在 go中 能调用。 它们不会返回结果。

Function	Behavior
print	prints all arguments; formatting of arguments is implementation-specific
println	like print but prints spaces between arguments and a newline at the end

实现限制：print,println 不能接受 随意的实参类型，除了 boolean，numeric，string，这3个必然被支持。

Packages

go程序 通过 把packages link起来 来构造。(Go programs are constructed by linking together packages.)
package 构造 from 一个或多个 源文件，这些文件一起定义了 属于这个package 的 常量，

类型，变量，函数， 这些可以在同一个包的 所有文件中 被访问。 这些元素可以 导出和使用 在另一个package里。

Source file organization

每个源文件 包含一个 package clause(从句，条款) 定义 它属于哪个包， 后续是 一个可能为空 的 导入声明，声明了 源文件希望使用 哪些包的内容， 后续是一个可能为空的 function，type，variable，constant的 声明。

```
SourceFile       = PackageClause ";" { ImportDecl ";" } { TopLevelDecl ";" } .
```

Package clause

package clause 开始 每个源文件， 定义了 它属于哪个包。

```
PackageClause  = "package" PackageName .
PackageName    = identifier .
```

包名不能是 空白标识符

```
package math
```

多个文件 共享相同的 包名 from 包的实现。 实现可能 要求 一个包的 所有的 源文件 inhabit(居住) 在同一个目录。

Import declarations

导入声明 说明 源文件 包含 对导入的包的 功能性依赖。(An import declaration states that the source file containing the declaration depends on functionality of the imported package) 且 允许访问 那个包 导出的 标识符。
导入的名字作为一个 标识符(PackageName) 用于访问， 一个 ImportPath 定义了导入的包。。。包 还能声明 哪些标识符 导出。。 没见过啊。

```
ImportDecl       = "import" ( ImportSpec | "(" { ImportSpec ";" } ")" ) .
ImportSpec       = [ "." | PackageName ] ImportPath .
ImportPath       = string_lit .
```

PackageName used in qualified identifiers 来 访问 导入的源文件中 包导出的标识符。。。ImportPath 是文件路径， 一个包 可能由多个 文件组成，这里 只导入了一个文件， 那么 不是这个包中所有的 都可以访问。
。。这个PackageName 是不是 必须 是 导入的源文件的 包名？ 还是说 只是一个 自定义的标识符。。 是导入的源文件的包名。。不是，，默认值 是 导入的源文件的包名。。

PackageName 在 file block 中被 声明。
如果 PackageName 被省略， 默认就是 导入的包的 package clause的 标识符。

。。ImportPath 是文件路径，还是 package 路径， 虽然 package 好像没有路径这种。。

如果有一个 显式的句号(.) 取代 名字， 所有的 在那个包中声明的 导出的 标识符 会在导入的源文件的 文件block 中被声明， 访问不能加 qualifier。

ImportPath的解释/翻译 是 依赖于 实现的， 但 它通常是 已编译的包 的 文件全名的一个 substr， 可能是 已安装的包的 相对的 repository。

实现限制： 编译器可能 限制 ImportPath 到 非空 只使用 Unicode的 L,M,N,P,S 类别(可视的字符 without space(空白，间隔)) 的 string， 也可能 排除 !"#$%&'()*,:;<=>?[\]^`{|} 和 Unicode 替换字符 U+FFFD 。

假定我们编译了一个包，这个包 包含 "package math"，这个 导出了函数 Sin，安装的 已编译的包 在 文件 "lib/math" 中。 下面说明了 在不同的导入声明下， 如何访问Sin

```
Import declaration          Local name of Sin
import   "lib/math"         math.Sin
import m "lib/math"         m.Sin
import . "lib/math"         Sin
```

导入声明 声明了一个 导入和导出包的 依赖关系
直接或间接导入 自己 或 直接导入一个包，但是没有引用到 它导出的任何标识符， 是非法的。

仅仅为了导入一个包，不触发任何 副作用， 使用空白标识符 作为显式的 包名。

```
import _ "lib/math"
```

An example package

下面是一个 完整的go包 ，实现了 并发 质数 筛选。

```
package main

import "fmt"

// Send the sequence 2, 3, 4, … to channel 'ch'.
func generate(ch chan<- int) {
    for i := 2; ; i++ {
        ch <- i  // Send 'i' to channel 'ch'.
    }
}

// Copy the values from channel 'src' to channel 'dst',
// removing those divisible by 'prime'.
func filter(src <-chan int, dst chan<- int, prime int) {
    for i := range src {  // Loop over values received from 'src'.
        if i%prime != 0 {
            dst <- i  // Send 'i' to channel 'dst'.
        }
    }
}
```

```
// The prime sieve: Daisy-chain filter processes together.
func sieve() {
    ch := make(chan int)  // Create a new channel.
    go generate(ch)       // Start generate() as a subprocess.
    for {
        prime := <-ch
        fmt.Print(prime, "\n")
        ch1 := make(chan int)
        go filter(ch, ch1, prime)
        ch = ch1
    }
}

func main() {
    sieve()
}
```

。。for里的go 是新建线程的话， 这个for 岂不是 无限建线程？ 还是说 go filter 是同步的？ 同步的话 没有任何意义啊。 应该是新建线程，毕竟同步的话，外部的go 就卡死了。 但是感觉 肯定有一些限制的， 主要是 ch=ch1， 这个 如果 go filter 不处理完的话， ch1 就是 不正确的啊。不，还有个 ch在， ch 就是为了线程间通信的。 不过会新建好多 ch。。 而且 ch的回收 是怎么样的？ 自动？但是我没有close啊。 不不不，for里的第一行，就是要读取 ch的， 所以 会限速的。
。。。这个 好绕啊。。不，还好，记得 generate方法。
。。和流有点像啊， generate 不停产生 2,3,4,… . 然后第一个for 读取了2，然后 用2filter， 然后filter后流 就把 2的倍数全部删除， 就变成了2,3,5,7,9... 然后下一次读取3，用3 filter， 就变成了 2,3,5,7,11...， 而且有 ch在，所以 就是流，不需要 集合。

Program initialization and execution

The zero value

为变量分配空间时，无论是通过 声明 或通过 new，或 一个新的值被创建，或通过 一个 合成字面量 或 调用make， 且 没有 提供 显示的初始化， 变量或值 会被给与 一个默认值。变量或值 的每个元素 都被设置为它的类型的 0值： boolean-false, 数值-0，string-""，指针，函数，接口，切片，channel，map 是 nil。 这个初始化会递归执行。

下面2个声明是等同的

```
var i int
var i int = 0
```

在有如下代码的情况下

```
type T struct { i int; f float64; next *T }
t := new(T)
```

下面的结果

```
t.i == 0
t.f == 0.0
t.next == nil
```

都是true，　下面的也会都是true
```
var t T
```

Package initialization
在包中，包层级 的变量 初始化 的执行 是逐步的， 每步选择 声明顺序中最早的，且不依赖未初始化值的 变量。
。。每次初始化一个？ 感觉是 重排序了，然后依次执行。
。。话说，go允许 int a = b; int b = 1; 吗？ 算了，这个问题有点。。。感觉肯定不允许。 反正也别写这种代码 就行了。。。下面就由这种 依赖后面的变量的 声明。。所以允许的。有点。。。 int a = b; int b = 1; b = 3.. 这种情况 a应该是1把？

更精确的说，包层级的 变量 被认为 可以初始化 如果 它还没有初始化 且 either 没有初始化表达式 or 它的初始化表达式没有依赖到未初始化的遍历。

初始化继续执行 by 重复地初始化 下一个package-level 中 声明最早且准备好初始化 的 变量，直到 没有 变量 ready for 初始化。

如果任何变量还没有初始化，当流程结束时，这些变量 是 一个或多个 初始化循环 中的一部分， 这代码 是 无效的。
。。循环依赖。

变量声明 中 左值的 多个变量，这些变量 被 右值 的 single (多值)表达式初始化， 这些变量 会同时被初始化： 如果左值中任意的遍历已经初始化了，那么所有其他的遍历 都需要在 同一步中 初始化。

```
var x = a
var a, b = f() // a and b are initialized together, before x is initialized
```
。。。。乱序。。

为了package初始化的目的，声明中 空白变量 和 其他变量 同等对待。

多个文件中变量的声明顺序 determined(被确定) by 对于编译器，文件presented(送达？)的顺序： 声明在第一个文件中的 变量声明 before 任何在第二个文件中声明的遍历。

依赖分析 不依靠 变量的真实值，只依靠 源码中 lexical reference (感觉是指 源码中的第x行的顺序，不应该是字典顺序)， 传递地分析(analyzed transitively)。 例如，如果一个变量x 的 初始化列表 指向了 一个函数，这个函数体中 指向了 变量 y， 那么 x 依赖于 y。Specifically(具体地)：
- 指向变量或 函数 的引用 是 一个标识符，表示了 那个 变量或函数。
- 引用指向 方法m 是 一个方法值 或 形如t.m的方法表达式，where(估计是指t) (静态) 类型 不是一个 接口类型，方法m 在 t的方法集合中。 resulting function value t.m 是否被invoke 是immaterial(无形的，不重要的)
- 变量，函数，方法x 依赖于 变量 y 如果 x的 初始化表达式 或 函数/方法的body 包含 指向y的 引用， 或 函数或方法 依赖于 y。

比如，给出声明：

```
var (
    a = c + b  // == 9
    b = f()    // == 4
    c = f()    // == 5
    d = 3      // == 5 after initialization has finished
)

func f() int {
    d++
    return d
}
```

the initialization order is d, b, c, a.
初始化表达式中的 sub表达式 的顺序 是无关的。a=c+b 和 a=b+c 导致相同的 初始化顺序。

依赖分析 执行 在每个包； 只有 在当前包中的 引用refer to 变量，函数，(非接口)方法定义 会被考虑(consider)。 如果 其他，隐藏的，数据依赖 存在于 变量之间，这些变量的初始化顺序 是未定义的。

例如，给出下面的声明：

```
var x = I(T{}).ab()   // x has an undetected, hidden dependency on a and b
var _ = sideEffect()  // unrelated to x, a, or b
var a = b
var b = 42

type I interface      { ab() []int }
type T struct{}
func (T) ab() []int   { return []int{a, b} }
```

变量a 会在 b之后 初始化，但是 其他的顺序(x是否在b之前初始化，还是在b和a之间，还是在a之后，还有sideEffect()被调用是在x初始化之前还是之后) 是 没有明确说明的。

变量也可能被初始化，通过 定义在package block 的 init 函数，没有实参和返回参数。

```
func init() { … }
```

每个包中可能会 定义多个 init函数， 甚至一个源文件中也有多个 init函数。在package block，init标识符 只能用来 声明 init函数。 init函数 不能 被程序的其他地方refer。

没有import的 包 被初始化 by 分配初始值 到 它的所有 package-level的 变量，然后 按被发送给编译器的 源文件(可能多个文件)中的出现顺序 调用 所有的 init函数。
如果包有 import，那么导入的包会被 初始化一次。 通过构造，包的导入 确保 不会出现 初始化的循环依赖。

包初始化 - 变量初始化 和 init函数的调用 - 发生在一个 single goroutine，依次的， 同一时间只有一个包。
一个init函数 可能 launch(发起，发动) 其他goroutine，这些可以同步执行 with 初始化代

码。当然，初始化总是 按序执行init函数： 它不会invoke 下一个init，直到 上一个init返回。

为了确保 reproducible(可再现的、可重复的) 初始化行为，构造系统(build system) 被鼓励 present(提交) 同一个包的 多个文件 按照 lexical 文件名顺序 到编译器。

Program execution
一个完整的程序 创建 by link 一个单独的，非导入的package 称为 main package， with 它导入的所有包，transitively。 main package 必须有 包的名字 "main" 并且 声明一个不带形参 和 返回值 的 main 函数

```
func main() { … }
```

程序的执行 以 初始化main package 为开始，然后 调用 main函数。 当 main函数返回，程序退出。 它不会等待 其他的 (non-main) goroutine 完成。

Errors
预定义的error 类型 如下：

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

它是一个传统的(conventional) 接口 for 代表一个错误condition(状况)，值为nil 则表示没有错误。比如，一个从文件读取数据的 函数可能如下定义：

```
func Read(f *File, b []byte) (n int, err error)
```

。。如果err 返回 nil ，就是没有错误。

Run-time panics
执行错误，如下标越界，会触发一个 rt-p，等价于 使用runtime.Error接口类型的 实现类型的值 调用 panic 内置函数。 那个类型满足 预定义的接口类型error。 精确的错误值，代表distinct(明确的，有区别的)的runtime 错误状况 是未说明的。

```
package runtime

type Error interface {
    error
    // and perhaps other methods
}
```

System considerations

Package unsafe

内置包 unsafe，编译器知道它，通过导入路径"unsafe" 可以访问。 提供了 低层 编程(包括违反type系统的操作)的 工具。 一个使用unsafe的包 必须被 手工审核 for 类型安全，可能

不是protable(跨平台？)。
这个包提供了下面的接口：
package unsafe

```
type ArbitraryType int  // shorthand for an arbitrary Go type; it is not a real type
type Pointer *ArbitraryType

func Alignof(variable ArbitraryType) uintptr
func Offsetof(selector ArbitraryType) uintptr
func Sizeof(variable ArbitraryType) uintptr

type IntegerType int  // shorthand for an integer type; it is not a real type
func Add(ptr Pointer, len IntegerType) Pointer
func Slice(ptr *ArbitraryType, len IntegerType) []ArbitraryType
```

Pointer 是一个 指针类型 但 Pointer的值 可能无法被 反引用。任何 指针 或 值 of underlying type uintptr 可以被转换为 一个 underlying type Pointer的 类型，反之亦然。 Pointer 和 uintptr 的转换 的 效果 是 依赖于 实现的。

```
var f float64
bits = *(*uint64)(unsafe.Pointer(&f))

type ptr unsafe.Pointer
bits = *(*uint64)(ptr(&f))

var p ptr = nil
```

Alignof 和 Sizeof 函数 接受一个任意类型的 表达式x，返回 alignment 或 size，respectively，一个假想的变量v 就像 v被声明 通过 var v = x.

Offsetof 函数 接受一个 (可能被括号包围的) selector s.f ， 代表了 s或*s 代表的 结构体 的属性 f， 返回 这个field 和 struct的地址 的offset。 如果f是一个 内嵌的属性，它必须是可到达的 不通过 指针 间接 访问。
对于一个 结构s 有属性f：

```
uintptr(unsafe.Pointer(&s)) + unsafe.Offsetof(s.f) ==
uintptr(unsafe.Pointer(&s.f))
```

电脑架构 可能需要 内存地址 来 align(排列，校准)，就是说，一个变量的地址 是一个因子(factor) 的倍数，变量的类型的alignment。
Alignof函数 接受一个 表达式 代表了一个 任意类型的变量， 返回 变量的类型的 alignment in bytes。
对于一个变量x：

```
uintptr(unsafe.Pointer(&x)) % unsafe.Alignof(x) == 0
```

调用 Alignof，Offsetof，Sizeof 是 编译时 uintptr 类型的 常量表达式。

Add函数 增加 len到ptr， 返回 更新后的 指针 unsafe.Pointer(uintptr(ptr) + uintptr(len))。 len参数 必须是 整数类型 或 无类型常量(必须代表一个int类型的值，并且 类型被认为是int)。 指针的校验 规则(https://golang.google.cn/pkg/unsafe/#Pointer) 依然 有效的。

Slice函数 返回一个 切片，它的 underlying 数组 开始于 ptr ，长度和capacity 是len。Slice(ptr,len) 等价于： (*[len]ArbitraryType)(unsafe.Pointer(ptr))[:]

除了 有个特殊case， 如果 ptr 是nil ，len是0， Slice 返回 nil 。。等价的那个表达式会返回什么，感觉是 rt-p？ 毕竟 本身nil的指针 不存在吧。

len 参数必须是 整型类型 或 无类型常量。 常量len参数 必须非负 且 代表了 int类型的一个值。如果是无类型常量，就认为是 int类型。 run time时，如果 len是负数，或者 ptr是nil且len非0， rt-p。

Size and alignment guarantees

对于数值类型，下面的size 确保

type	size in bytes
byte, uint8, int8	1
uint16, int16	2
uint32, int32, float32	4
uint64, int64, float64, complex64	8
complex128	16

下面的 minimal alignment properties (最小对齐特性) 保证：

对于 任何类型的x变量， unsage.Alignof(x) 至少是1.

对于 struct类型的x变量，unsage.Alignof(x) 是 x的每个属性f 的 unsage.Alignof(x.f)的最大值，至少1。

对于一个 数组类型的x变量， unsage.Alignof(x) 等于 数组的元素类型的一个变量的alignment。

一个结构 或数组 类型的 size 是 0， 如果 它不包含 size大于0的 属性(或元素) 。2个distinct 的 0size 变量 可能 在内存中 有相同的地址。