Go 编程语言

文档 参考 包 项目 帮助 搜索

# Go编程语言规范

## 版本：2013年1月21日 || 译者：Oling Cat <olingcat@gmail.com>

## Introduction

## 前言

This is a reference manual for the Go programming language. For more information and other documents, see http://golang.org.

这是一份关于Go语言的参考手册。欲获取更多信息与文档， 请访问http://golang.org。

Go is a general-purpose language designed with systems programming in mind. It is strongly typed and garbage-collected and has explicit support for concurrent programming. Programs are constructed from *packages*, whose properties allow efficient management of dependencies. The existing implementations use a traditional compile/link model to generate executable binaries.

Go是通用型编程语言，它为系统编程而设计。它是强类型化的语言，具有垃圾回收机制，并显式支持并发编程。 程序由*包*构造，以此来提供高效的依赖管理功能。当前实现使用传统的“编译-链接”模型来生成可执行的二进制文件。

The grammar is compact and regular, allowing for easy analysis by automatic tools such as integrated development environments.

其语法紧凑而规则，便于IDE等自动化工具分析。

## Notation

## 记法

The syntax is specified using Extended Backus-Naur Form (EBNF):

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

语法使用扩展巴克斯-诺尔范式（EBNF）定义：

```
生成式 = 生成式名 "=" [ 表达式 ] "." .
表达式 = 选择项 { "|" 选择项 } .
选择项 = 条目 { 条目 } .
条目   = 生成式名 | 标记 [ "…" 标记 ] | 组 | 可选项 | 重复项 .
组     = "(" 表达式 ")" .
可选项 = "[" 表达式 "]" .
重复项 = "{" 表达式 "}" .
```

Productions are expressions constructed from terms and the following operators, in increasing precedence:

```
|   alternation
()  grouping
[]  option (0 or 1 times)
{}  repetition (0 to n times)
```

生成式由表达式构造，表达式通过术语及以下操作符构造，自上而下优先级递增（低=>高）：

```
|   选择
()  分组
[]  可选 （0 或 1 次）
{}  重复 （0 到 n 次）
```

Lower-case production names are used to identify lexical tokens. Non-terminals are in CamelCase. Lexical tokens are enclosed in double quotes `""` or back quotes ` `` `.

小写生成式名用于确定词法标记。非最终（Non-terminals）词法标记使用驼峰记法（CamelCase）。 置于双引号 `""` 或反引号 ` `` ` 中的为词法标记。

The form `a … b` represents the set of characters from `a` through `b` as alternatives. The horizontal ellipsis `…` is also used elsewhere in the spec to informally denote various enumerations or code snippets that are not further specified. The character `…` (as opposed to the three characters `...`) is not a token of the Go language.

形式 `a … b` 表示把从 `a` 到 `b` 的字符集作为选择项。 横向省略号 `…` 也在本文档的其它地方非正式地表示各种列举或简略的代码片断。 单个字符 `…`（不同于三个字符 `...`）并非Go语言本身的标记。

## Source code representation

## 源码的表示

Source code is Unicode text encoded in UTF-8. The text is not canonicalized, so a single accented code point is distinct from the same character constructed from combining an accent and a letter; those are treated as two code points. For simplicity, this document will use the unqualified term *character* to refer to a Unicode code point in the source text.

源码是采用UTF-8编码的Unicode文本。 该文本是非商业化的，因此单一的着重号码点不同于结合了字母与着重号的字符结构， 那些应当视为两个码点。为简单起见，本文档将使用未限定的术语*字符*在源文本中代替Unicode码点。

Each code point is distinct; for instance, upper and lower case letters are different characters.

每个码点都是不同的，例如，大写与小写的字母就是不同的字符。

Implementation restriction: For compatibility with other tools, a compiler may disallow the NUL character (U+0000) in the source text.

实现限制：为兼容其它工具，编译器会阻止字符NUL（U+0000）出现在源码文本中。

Implementation restriction: For compatibility with other tools, a compiler may ignore a UTF-8-encoded byte order mark (U+FEFF) if it is the first Unicode code point in the source text.

实现限制：为兼容其它工具，若UTF-8编码的字节序标记（U+FEFF）为源文本中的第一个Unicode码点， 编译器就会忽略它。

## Characters

## 字符

The following terms are used to denote specific Unicode character classes:

```
newline        = /* the Unicode code point U+000A */ .
unicode_char   = /* an arbitrary Unicode code point except newline */ .
unicode_letter = /* a Unicode code point classified as "Letter" */ .
unicode_digit  = /* a Unicode code point classified as "Decimal Digit" */ .
```

具体的Unicode字符类别由以下术语表示：

```
换行符      = /* Unicode码点 U+000A */ .
unicode字符 = /* 除newline以外的任意Unicode码点 */ .
unicode字母 = /* 类型为“字母”的Unicode码点 */ .
unicode数字 = /* 类型为“十进制数字”的Unicode码点 */ .
```

In The Unicode Standard 6.2, Section 4.5 "General Category" defines a set of character categories. Go treats those characters in category Lu, Ll, Lt, Lm, or Lo as Unicode letters, and those in category Nd as Unicode digits.

在Unicode标准6.2， 章节4.5 “一般类别” 中定义了字符集类别。 其中类别Lu，Ll，Lt，Lm及Lo被视为Unicode字母，类别Nd被视为Unicode数字。

## Letters and digits

## 字母和数字

The underscore character _ (U+005F) is considered a letter.

```
letter        = unicode_letter | "_" .
decimal_digit = "0" … "9" .
octal_digit   = "0" … "7" .
hex_digit     = "0" … "9" | "A" … "F" | "a" … "f" .
```

下划线字符_（U+005F）被视为一个字母。

```
字母         = unicode字母 | "_" .
十进制数字   = "0" … "9" .
八进制数字   = "0" … "7" .
十六进制数字 = "0" … "9" | "A" … "F" | "a" … "f" .
```

## Lexical elements

## 词法元素

## Comments

## 注释

There are two forms of comments:

注释有两种形式：

1. *Line comments* start with the character sequence `//` and stop at the end of the line. A line comment acts like a newline.
2. *General comments* start with the character sequence `/*` and continue through the character sequence `*/`. A general comment containing one or more newlines acts like a newline, otherwise it acts like a space.

1. *行注释* 以`//` 开始，至行尾结束。一条行注释视为一个换行符。
2. *块注释* 以 `/*` 开始，至 `*/` 结束。 块注释在包含多行时视为一个换行符，否则视为一个空格。

Comments do not nest.

注释不可嵌套。

## Tokens

## 标记

Tokens form the vocabulary of the Go language. There are four classes: *identifiers*, *keywords*, *operators and delimiters*, and *literals*. *White space*, formed from spaces (U+0020), horizontal tabs (U+0009), carriage returns (U+000D), and newlines (U+000A), is ignored except as it separates tokens that would otherwise combine into a single token. Also, a newline or end of file may trigger the insertion of a semicolon. While breaking the input into tokens, the next token is the longest sequence of characters that form a valid token.

标记构成Go语言的词汇。它有4种类型：*标识符*，*关键字*， *运算符与分隔符*以及*字面*。*空白符*由空格（U+0020）， 横向制表符（U+0009），回车符（U+000D）和换行符（U+000A）组成，除非用它来分隔会结合成单个的标记， 否则它将被忽略。此外，换行符或EOF（文件结束符）会触发分号的插入。 当把输入分解为标记时，可形成有效标记的最长字符序列将作为下一个标记。

## Semicolons

## 分号

The formal grammar uses semicolons `";"` as terminators in a number of productions. Go programs may omit most of these semicolons using the following two rules:

1. When the input is broken into tokens, a semicolon is automatically inserted into the token stream at the end of a non-blank line if the line's final token is
   - an identifier
   - an integer, floating-point, imaginary, rune, or string literal
   - one of the keywords `break`, `continue`, `fallthrough`, or `return`
   - one of the operators and delimiters `++`, `--`, `)`, `]`, or `}`
2. To allow complex statements to occupy a single line, a semicolon may be omitted before a closing `")"` or `"}"`.

正式语法使用分号 `";"` 作为一些生成式的终止符。Go程序会使用以下两条规则来省略大多数分号：

1. 当输入被分解成标记时，若该行的行末标记为以下标记之一，分号就会被自动插入到该标记流中的非空行末处：
   - 标识符
   - 整数， 浮点数， 虚数， 符文或 字符串 字面
   - 关键字 `break`， `continue`， `fallthrough` 或 `return`
   - 运算符与分隔符 `++`， `--`， `)`， `]`或 `}`
2. 为允许复合语句占据单行，闭合的 `")"` 或 `"}"` 之前的分号可以省略。

To reflect idiomatic use, code examples in this document elide semicolons using these rules.

为符合习惯用法，本文档中的代码示例将使用这些规则省略分号。

## Identifiers

## 标识符

Identifiers name program entities such as variables and types. An identifier is a sequence of one or more letters and digits. The first character in an identifier must be a letter.

```
identifier = letter { letter | unicode_digit } .
```

标识符被用来命名程序实体，例如变量和类型。一个标识符由一个或多个字母和数字组成。 标识符的第一个字符必须是字母。

```
标识符 = 字母 { 字母 | unicode数字 } .
```

```
a
```

```
_x9
ThisVariableIsExported
αβ
```

Some identifiers are predeclared.

有些标识符是预声明的。

## Keywords

## 关键字

The following keywords are reserved and may not be used as identifiers.

以下为保留关键字，不能用作标识符。

```
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
```

## Operators and Delimiters

## 运算符与分隔符

The following character sequences represent operators, delimiters, and other special tokens:

以下字符序列表示运算符，分隔符和其它特殊标记：

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

## Integer literals

## 整数字面

An integer literal is a sequence of digits representing an integer constant. An optional prefix sets a non-decimal base: `0` for octal, `0x` or `0X` for hexadecimal. In hexadecimal literals, letters `a-f` and `A-F` represent values 10 through 15.

```
int_lit     = decimal_lit | octal_lit | hex_lit .
decimal_lit = ( "1" … "9" ) { decimal_digit } .
octal_lit   = "0" { octal_digit } .
hex_lit     = "0" ( "x" | "X" ) hex_digit { hex_digit } .
```

整数字面由数字序列组成，代表整数常量。非十进制数由这些前缀定义： `0` 为八进制数前缀，`0x` 或 `0X`为十六进制数前缀。 在十六进制数字面中，字母 `a-f` 或 `A-F` 表示值10到15。

```
整数字面       = 十进制数字面 | 八进制数字面 | 十六进制数字面 .
十进制数字面   = ( "1" … "9" ) { 十进制数字 } .
八进制数字面   = "0" { 八进制数字 } .
十六进制数字面 = "0" ( "x" | "X" ) 十六进制数字 { 十六进制数字 } .
```

```
42
0600
0xBadFace
170141183460469231731687303715884105727
```

## Floating-point literals

## 浮点数字面

A floating-point literal is a decimal representation of a floating-point constant. It has an integer part, a decimal point, a fractional part, and an exponent part. The integer and fractional part comprise decimal digits; the exponent part is an `e` or `E` followed by an optionally signed decimal exponent. One of the integer part or the fractional part may be elided; one of the decimal point or the exponent may be elided.

```
float_lit = decimals "." [ decimals ] [ exponent ] |
            decimals exponent |
            "." decimals [ exponent ] .
decimals  = decimal_digit { decimal_digit } .
exponent  = ( "e" | "E" ) [ "+" | "-" ] decimals .
```

浮点数字面由十进制浮点常量表示。 它由整数部分，小数点，小数部分和指数部分构成。整数部分与小数部分由十进制数字组成； 指数部分由一个 e 或 E 紧跟一个带可选正负号的十进制指数构成。 整数部分或小数部分可以省略；小数点或指数亦可省略。

```
浮点数字面 = 十进制数 "." [ 十进制数 ] [ 指数 ] |
             十进制指数 |
             "." 十进制数 [ 指数 ] .
十进制数   = 十进制数字 { 十进制数字 } .
指数       = ( "e" | "E" ) [ "+" | "-" ] 十进制数 .
```

```
0.
72.40
072.40  // == 72.40
2.71828
1.e+0
6.67428e-11
1E6
.25
.12345E+5
```

## Imaginary literals

## 虚数字面

An imaginary literal is a decimal representation of the imaginary part of a complex constant. It consists of a floating-point literal or decimal integer followed by the lower-case letter `i`.

```
imaginary_lit = (decimals | float_lit) "i" .
```

虚数字面由十进制复数常量的虚部表示。 它由浮点数字面或十进制整数紧跟小写字母 `i` 构成。

```
虚数字面 = (十进制数 | 浮点数字面) "i" .
```

```
0i
011i  // == 11i
0.i
2.71828i
1.e+0i
6.67428e-11i
1E6i
.25i
.12345E+5i
```

## Rune literals

## 符文字面

A rune literal represents a rune constant, an integer value identifying a Unicode code point. A rune literal is expressed as one or more characters enclosed in single quotes. Within the quotes, any character may appear except single quote and newline. A single quoted character represents the Unicode value of the character itself, while multi-character sequences beginning with a backslash encode values in various formats.

符文字面由一个符文常量表示，一个整数值确定一个Unicode码点。 一个符文字面由围绕在单引号中的一个或更多字符表示。通常一个Unicode码点作为一个或更多字符围绕在单引号中。 在引号内，除单引号和换行符外的任何字符都可出现。引号内的单个字符通常代表该字符的Unicode值自身， 而以反斜杠开头的多字符序列则会编码为不同的形式。

The simplest form represents the single character within the quotes; since Go source text is Unicode characters encoded in UTF-8, multiple UTF-8-encoded bytes may represent a single integer value. For instance, the literal `'a'` holds a single byte representing a literal `a`, Unicode U+0061, value `0x61`, while `'ä'` holds two bytes (`0xc3 0xa4`) representing a literal `a`-dieresis, U+00E4, value `0xe4`.

最简单的形式就是引号内只有一个字符；由于Go源码文本是UTF-8编码的Unicode字符， 多个UTF-8编码的字节就可以表示一个单一的整数值。例如，字面 `'a'` 包含一个字节， 表示一个字面 `a`，Unicode字符U+0061，值 `0x61`，而 `'ä'` 则包含两个字节（`0xc3 0xa4`），表示一

个字面 分音符-a， Unicode字符U+00E4，值 0xe4。

Several backslash escapes allow arbitrary values to be encoded as ASCII text. There are four ways to represent the integer value as a numeric constant: \x followed by exactly two hexadecimal digits; \u followed by exactly four hexadecimal digits; \U followed by exactly eight hexadecimal digits, and a plain backslash \ followed by exactly three octal digits. In each case the value of the literal is the value represented by the digits in the corresponding base.

反斜杠转义允许用ASCII文本来编码任意值。将整数值表示为数字常量有4种方法： \x 紧跟2个十六进制数字；\u 紧跟4个十六进制数字；\U 紧跟8个十六进制数字；单个\紧跟3个八进制数字。 在任何情况下，字面的值都是其相应进制的数字表示的值。

Although these representations all result in an integer, they have different valid ranges. Octal escapes must represent a value between 0 and 255 inclusive. Hexadecimal escapes satisfy this condition by construction. The escapes \u and \U represent Unicode code points so within them some values are illegal, in particular those above 0x10FFFF and surrogate halves.

尽管这些表示方式都会产生整数，其有效范围却不相同。八进制转义只能表示 0 到 255 之间的值。 十六进制转义则视其结构而定。转义符 \u 和 \U 表示Unicode码点， 因此其中的一些值是非法的，特别是那些大于 0x10FFFF 的值和半代理值。

After a backslash, certain single-character escapes represent special values:

```
\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)
```

在反斜杠后，某些单字符转义表示特殊值：

```
\a   U+0007 警报或铃声
\b   U+0008 退格
\f   U+000C 换页
\n   U+000A 换行
\r   U+000D 回车
\t   U+0009 横向制表
\v   U+000b 纵向制表
\\   U+005c 反斜杠
\'   U+0027 单引号(仅在符文字面中有效)
\"   U+0022 双引号(仅在字符串字面中有效)
```

All other sequences starting with a backslash are illegal inside rune literals.

```
rune_lit         = "'" ( unicode_value | byte_value ) "'" .
unicode_value    = unicode_char | little_u_value | big_u_value | escaped_char .
byte_value       = octal_byte_value | hex_byte_value .
octal_byte_value = `\` octal_digit octal_digit octal_digit .
hex_byte_value   = `\` "x" hex_digit hex_digit .
little_u_value   = `\` "u" hex_digit hex_digit hex_digit hex_digit .
big_u_value      = `\` "U" hex_digit hex_digit hex_digit hex_digit
                           hex_digit hex_digit hex_digit hex_digit .
escaped_char     = `\` ( "a" | "b" | "f" | "n" | "r" | "t" | "v" | `\` | "'" | `"` ) .
```

```
'a'
'ä'
'本'
'\t'
'\000'
'\007'
'\377'
'\x07'
'\xff'
'\u12e4'
'\U00101234'
'aa'         // illegal: too many characters
'\xa'        // illegal: too few hexadecimal digits
'\0'         // illegal: too few octal digits
'\uDFFF'     // illegal: surrogate half
'\U00110000' // illegal: invalid Unicode code point
```

在符文字面中，所有其它以反斜杠开始的序列都是非法的。

```
符文字面        = "'" ( unicode值 | 字节值 ) "'" .
unicode值       = unicode字符 | 小Unicode值 | 大Unicode值 | 转义字符 .
字节值          = 八进制字节值 | 十六进制字节值 .
八进制字节值    = `\` 八进制数字 八进制数字 八进制数字 .
十六进制字节值 = `\` "x" 十六进制数字 十六进制数字 .
小Unicode值     = `\` "u" 十六进制数字 十六进制数字 十六进制数字 十六进制数字 .
大Unicode值     = `\` "U" 十六进制数字 十六进制数字 十六进制数字 十六进制数字
                             十六进制数字 十六进制数字 十六进制数字 十六进制数字 .
转义字符        = `\` ( "a" | "b" | "f" | "n" | "r" | "t" | "v" | `\` | "'" | `"` ) .
```

```
'a'
'ä'
'本'
'\t'
'\000'
'\007'
'\377'
'\x07'
'\xff'
'\u12e4'
'\U00101234'
'aa'         // 非法: 太多字符
'\xa'        // 非法: 太少16进制数字
'\0'         // 非法: 太少8进制数字
'\uDFFF'     // 非法: 半代理值
'\U00110000' // 非法: 无效Unicode码点
```

## String literals

## 字符串字面

A string literal represents a string constant obtained from concatenating a sequence of characters. There are two forms: raw string literals and interpreted string literals.

字符串字面表示字符串常量，可通过连结字符序列获得。 它有两种形式：原始字符串字面和解译字符串字面。

Raw string literals are character sequences between back quotes ``. Within the quotes, any character is legal except back quote. The value of a raw string literal is the string composed of the uninterpreted (implicitly UTF-8-encoded) characters between the quotes; in particular, backslashes have no special meaning and the string may contain newlines. Carriage returns inside raw string literals are discarded from the raw string value.

原始字符串字面为反引号 `` 之间的字符序列。在该引号内， 除反引号外的任何字符都是合法的。原始字符串字面的值为此引号之间的无解译（隐式UTF-8编码的）字符组成的字符串； 另外，反斜杠没有特殊意义且字符串可包含换行符。原始字符串字面中的回车符将会从原始字符串的值中丢弃。

Interpreted string literals are character sequences between double quotes "". The text between the quotes, which may not contain newlines, forms the value of the literal, with backslash escapes interpreted as they are in rune literals (except that \' is illegal and \" is legal), with the same restrictions. The three-digit octal (*nnn*) and two-digit hexadecimal (\x*nn*) escapes represent individual *bytes* of the resulting string; all other escapes represent the (possibly multi-byte) UTF-8 encoding of individual *characters*. Thus inside a string literal \377 and \xFF represent a single byte of value 0xFF=255, while ÿ, \u00FF, \U000000FF and \xc3\xbf represent the two bytes 0xc3 0xbf of the UTF-8 encoding of character U+00FF.

```
string_lit             = raw_string_lit | interpreted_string_lit .
raw_string_lit         = "`" { unicode_char | newline } "`" .
interpreted_string_lit = `"` { unicode_value | byte_value } `"` .
```

```
`abc`  // same as "abc"
`\n
\n`    // same as "\\n\n\\n"
"\n"
""
"Hello, world!\n"
"日本語"
"\u65e5本\U00008a9e"
"\xff\u00FF"
"\uD800"       // illegal: surrogate half
"\U00110000"   // illegal: invalid Unicode code point
```

解译字符串字面为双引号 "" 之间的字符序列。 在该引号内，不包含换行符的文本形成字面的值，反斜杠转义序列如同在符文字面中一样以相同的限制被解译（其中\'是非法的，而 \" 是合法的）。 3位八进制（*nnn*）和2位十六进制（\x*nn*） 转义表示字符串值的独立 *字节*；其它转义符表示（可多字节）UTF-8编码的独立 *字符*。 因此在字符串字面中，\377 和 \xFF 表示值为 0xFF=255的单个字节， 而ÿ、\u00FF、\U000000FF 和 \xc3\xbf 则表示UTF-8编码的字符U+00FF的两个字节0xc3 0xbf。

```
字符串字面     = 原始字符串字面 | 解译字符串字面 .
原始字符串字面 = "`" { unicode字符 | 换行符 } "`" .
解译字符串字面 = `"` { unicode值 | 字节值 } `"` .
```

```
`abc`  // 等价于 "abc"
`\n
\n`    // 等价于 "\\n\n\\n"
"\n"
""
"Hello, world!\n"
"日本語"
"\u65e5本\U00008a9e"
"\xff\u00FF"
"\uD800"       // 非法: 半代理值
"\U00110000"   // 非法: 无效的Unicode码点
```

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
```

这些例子都表示相同的字符串：

```
"日本語"                                // UTF-8输入的文本
`日本語`                                // UTF-8输入的原始字面文本
"\u65e5\u672c\u8a9e"                    // 显式的Unicode码点
"\U000065e5\U0000672c\U00008a9e"        // 显式的Unicode码点
"\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e"  // 显式的UTF-8字节
```

If the source code represents a character as two code points, such as a combining form involving an accent and a letter, the result will be an error if placed in a rune literal (it is not a single code point), and will appear as two code points if placed in a string literal.

如果源码将两个码点表示为一个字符，例如包含着重号和字母的结合形式， 那么将它放置在符文字面中就会产生一个错误（它不是单一码点），而放置在字符串字面中则会显示为两个码点。

## Constants

## 常量

There are *boolean constants*, *rune constants*, *integer constants*, *floating-point constants*, *complex constants*, and *string constants*. Character, integer, floating-point, and complex constants are collectively called *numeric constants*.

常量包含*布尔常量*，*符文常量*， *整数常量*，*浮点数常量*， *复数常量*和*字符串常量*。 字符，整数，浮点数和复数常量统称为*数值常量*。

A constant value is represented by a rune, integer, floating-point, imaginary, or string literal, an identifier denoting a constant, a constant expression, a conversion with a result that is a constant, or the result value of some built-in functions such as `unsafe.Sizeof` applied to any value, `cap` or `len` applied to some expressions, `real` and `imag` applied to a complex constant and `complex` applied to numeric constants. The boolean truth values are represented by the predeclared constants `true` and `false`. The predeclared identifier iota denotes an integer constant.

常量的值可由 符文， 整数， 浮点数， 虚数 或 字符串字面表示， 一个标识符可代表一个常量，一个常量表达式， 一个结果为常量的类型转换，或一些内建函数的返回值。 例如 `unsafe.Sizeof` 作用于任何值时产生的值， `cap` 或 `len` 作用于一些表达式时产生的值， `real` 和 `imag` 作用于复数常量时产生的值，以及 `complex` 作用于数值常量所产生的值。 布尔值由预声明的常量 `true` 和 `false` 来表示。 预声明标识符 iota 表示一个整数常量。

In general, complex constants are a form of constant expression and are discussed in that section.

通常，复数常量的形式为 常量表达式，这一点将在该节中讨论。

Numeric constants represent values of arbitrary precision and do not overflow.

数值常量可表示任意精度的值而不会溢出。

Constants may be typed or untyped. Literal constants, `true`, `false`, `iota`, and certain constant expressions containing only untyped constant operands are untyped.

常量可以是类型化的或无类型化的。字面常量，`true`，`false`， `iota` 和某些只包含无类型化操作数的常量表达式是无类型化的。

A constant may be given a type explicitly by a constant declaration or conversion, or implicitly when used in a variable declaration or an assignment or as an operand in an expression. It is an error if the constant value cannot be represented as a value of the respective type. For instance, `3.0` can be given any integer or any floating-point type, while `2147483648.0` (equal to `1<<31`) can be given the types `float32`, `float64`, or `uint32` but not `int32` or `string`.

常量可由常量声明或类型转换显式地赋予其类型， 也可由变量声明或赋值以及作为 表达式中的操作数隐式地赋予其类型。若常量的值不能由其类型表示就会产生一个错误。 例如，`3.0` 可赋予任何整数或浮点数类型的常量，而 `2147483648.0` （等价于 `1<<31`）则只能赋予 `float32`, `float64` 或 `uint32` 类型的常量，而不能赋予 `int32` 或 `string`类型的常量。

There are no constants denoting the IEEE-754 infinity and not-a-number values, but the `math` package's Inf, NaN, IsInf, and IsNaN functions return and test for those values at run time.

Go语言中没有代表 IEEE-754 无穷大和 NaN 值（非数值）的常量，然而 `math` 包中的 Inf， NaN， IsInf 和 IsNaN 函数会在运行时返回并检验这些值。

Implementation restriction: Although numeric constants have arbitrary precision in the language, a compiler may implement them using an internal representation with limited precision. That said, every implementation must:

- Represent integer constants with at least 256 bits.
- Represent floating-point constants, including the parts of a complex constant, with a mantissa of at least 256 bits and a signed exponent of at least 32 bits.
- Give an error if unable to represent an integer constant precisely.
- Give an error if unable to represent a floating-point or complex constant due to overflow.
- Round to the nearest representable constant if unable to represent a floating-point or complex constant due to limits on precision.

These requirements apply both to literal constants and to the result of evaluating constant expressions.

实现限制：尽管数值常量在该语言中可拥有任意精度， 但编译器可能使用其有限精度的内部表示来实现它们。即，每个实现必须：

- 使用至少256位表示整数常量。
- 使用至少256位表示浮点常量，包括复数常量及尾数部分； 使用至少32位表示指数符号。
- 若无法精确表示一个整数常量，则给出一个错误。
- 若由于溢出而无法表示一个浮点或复数常量，则给出一个错误。
- 若由于精度限制而无法表示一个浮点或复数常量，则舍入为最近似的可表示常量。

这些要求适用于字面常量和常量表达式的求值结果。

## Types

## 类型

A type determines the set of values and operations specific to values of that type. A type may be specified by a (possibly qualified) *type name* (§Type declarations) or a *type literal*, which composes a new type from previously declared types.

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

类型决定值的集合与该类型值特定的操作。类型可通过（或许为限定的） *类型名*（§类型声明）或*类型字面*指定， 它将根据之前声明的类型组成新的类型。

```
类型     = 类型名 | 类型字面 | "(" 类型 ")" .
类型名   = 标识符 | 限定标识符 .
类型字面 = 数组类型 | 结构类型 | 指针类型 | 函数类型 | 接口类型 |
           切片类型 | 映射类型 | 信道类型 .
```

Named instances of the boolean, numeric, and string types are predeclared. *Composite types*—array, struct, pointer, function, interface, slice, map, and channel types—may be constructed using type literals.

布尔值，数值与字符串类型的实例的命名是预声明的。 数组，结构，指针，函数，接口，切片，映射和信道这些*复合类型*可由类型字面构造。

The *static type* (or just *type*) of a variable is the type defined by its declaration. Variables of interface type also have a distinct *dynamic type*, which is the actual type of the value stored in the variable at run time. The dynamic type may vary during execution but is always assignable to the static type of the interface variable. For non-interface types, the dynamic type is always the static type.

变量的*静态类型*（或*类型*）是通过其声明定义的类型。 接口类型的变量也有一个独特的*动态类型*，这是在运行时存储在变量中的值的实际类型。 动态类型在执行过程中可能会有所不同，但对于接口变量的静态类型，它总是可赋值的。 对于非接口类型，其动态类型始终为其静态类型。

Each type `T` has an *underlying type*: If `T` is a predeclared type or a type literal, the corresponding underlying type is `T` itself. Otherwise, `T`'s underlying type is the underlying type of the type to which `T` refers in its type declaration.

每个类型 T 都有一个 *基本类型*：若 T 为预声明类型或类型字面， 其相应的基本类型为 T 本身。否则，T的基本类型为其 类型声明中所依据类型的基本类型。

```
type T1 string
type T2 T1
type T3 []T1
type T4 T3
```

The underlying type of `string`, `T1`, and `T2` is `string`. The underlying type of `[]T1`, `T3`, and `T4` is `[]T1`.

以上 `string`，`T1` 和 `T2` 的基本类型为 `string`。 `[]T1`，`T3` 和 `T4` 的基本类型为 `[]T1`。

## Method sets

## 方法集

A type may have a *method set* associated with it (§Interface types, §Method declarations). The method set of an interface type is its interface. The method set of any other type `T` consists of all methods with receiver type `T`. The method set of the corresponding pointer type `*T` is the set of all methods with receiver `*T` or `T` (that is, it also contains the method set of `T`). Further rules apply to structs containing anonymous fields, as described in the section on struct types. Any other type has an empty method set. In a method set, each method must have a unique method name.

类型可拥有一个与其相关联的 *方法集*（§接口类型，§方法声明）。 接口类型的方法集为其接口。其它任意类型 `T` 的方法集由所有带接收者类型 `T` 的方法组成。与指针类型 `*T` 相应的方法集为所有带接收者 `*T` 或 `T` 的方法的集（就是说，它也包含 `T` 的方法集）。根据结构类型一节的描述， 更进一步的规则也适用于包含匿名字段的结构。任何其它类型都有一个空方法集。 在方法集中，每个方法都必须有唯一的方法名。

The method set of a type determines the interfaces that the type implements and the methods that can be called using a receiver of that type.

一个类型的方法集决定了其所实现的接口 与可使用该类型接收者调用的方法。

## Boolean types

## 布尔类型

A *boolean type* represents the set of Boolean truth values denoted by the predeclared constants `true` and `false`. The predeclared boolean type is `bool`.

*布尔类型* 表示由预声明常量 `true` 和 `false`所代表的布尔值的集。 预声明的布尔类型为 `bool`。

## Numeric types

## 数值类型

A *numeric type* represents sets of integer or floating-point values. The predeclared architecture-independent numeric types are:

```
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
```

*数值类型*表示整数值和浮点数值的集。 架构中立的预声明数值类型为：

```
uint8       所有无符号 8位整数集（0 到 255）
uint16      所有无符号16位整数集（0 到 65535）
uint32      所有无符号32位整数集（0 到 4294967295）
uint64      所有无符号64位整数集（0 到 18446744073709551615）
```

```
int8        所有带符号 8位整数集（-128 到 127）
int16       所有带符号16位整数集（-32768 到 32767）
int32       所有带符号32位整数集（-2147483648 到 2147483647）
int64       所有带符号64位整数集（-9223372036854775808 到 9223372036854775807）

float32     所有IEEE-754 32位浮点数集
float64     所有IEEE-754 64位浮点数集

complex64   所有带float32实部和虚部的复数集
complex128  所有带float64实部和虚部的复数集

byte        uint8的别名
rune        int32的别名
```

The value of an *n*-bit integer is *n* bits wide and represented using two's complement arithmetic.

*n* 位的整数值是 *n* 位宽的，它使用 二进制补码运算表示。

There is also a set of predeclared numeric types with implementation-specific sizes:

```
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
```

大小取决于具体实现的预声明数值类型：

```
uint     32或64位
int      大小与uint相同
uintptr  大到足以存储指针值无解释位的无符号整数
```

To avoid portability issues all numeric types are distinct except `byte`, which is an alias for `uint8`, and `rune`, which is an alias for `int32`. Conversions are required when different numeric types are mixed in an expression or assignment. For instance, `int32` and `int` are not the same type even though they may have the same size on a particular architecture.

为避免可移植性问题，除 `byte` 为 `uint8` 的别名以及 `rune` 为 `int32` 的别名外，所有数值类型都是不同的。 当不同的数值类型混合在一个表达式或赋值操作中时，必须进行类型转换。 例如，`int32` 与 `int` 是不同的类型， 尽管它们在特定架构上可能有相同的大小。

## String types

## 字符串类型

A *string type* represents the set of string values. A string value is a (possibly empty) sequence of bytes. Strings are immutable: once created, it is impossible to change the contents of a string. The predeclared string type is `string`.

*字符串类型* 表示字符串值的集。字符串的值为（可能为空的）字节序列。字符串是不可变的： 一旦被创建，字符串的内容就不能更改。 预声明的字符串类型为 `string`。

The length of a string `s` (its size in bytes) can be discovered using the built-in function `len`. The length is a compile-time constant if the string is a constant. A string's bytes can be accessed by integer indices 0 through `len(s)-1`. It is illegal to take the address of such an element; if `s[i]` is the `i`'th byte of a string, `&s[i]` is invalid.

字符串 `s` 的长度（即其字节大小）可使用内建函数 `len` 获取。若该字符串为常量，则其长度即为编译时常量。 字符串的字节可通过整数（§下标）0 至 `len(s)-1` 访问。 获取这样一个元素的地址是非法的；若 `s[i]` 为字符串的第 `i` 个字节，`&s[i]` 就是无效的。

## Array types

## 数组类型

An array is a numbered sequence of elements of a single type, called the element type. The number of elements is called the length and is never negative.

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

数组是单一类型元素的编号序列，该单一类型称为元素类型。元素的数量称为长度且为非负数。

```
数组类型 = "[" 数组长度 "]" 元素类型 .
数组长度 = 表达式 .
元素类型 = 类型 .
```

The length is part of the array's type; it must evaluate to a non- negative constant representable by a value of type `int`. The length of array `a` can be discovered using the built-in function `len`. The elements can be addressed by integer indices 0 through `len(a)-1`. Array types are always one-dimensional but may be composed to form multi-dimensional types.

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

长度是数组类型的一部分，其求值结果必须可表示为 `int` 类型的非负常量。 数组 `a` 的长度可使用内建函数 `len`获取， 其元素可通过整数下标 0 到 `len(a)-1` 寻址。 数组类型总是一维的，但可组合构成多维的类型。

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

## Slice types

## 切片类型

A slice is a reference to a contiguous segment of an array and contains a numbered sequence of elements from that array. A slice type denotes the set of all slices of arrays of its element type. The value of an uninitialized slice is `nil`.

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

切片是数组连续段的引用及包含此数组的元素的编号序列。 切片类型表示元素类型为数组的所有切片的集。未初始化切片的值为 `nil`。

```
切片类型 = "[" "]" 元素类型 .
```

Like arrays, slices are indexable and have a length. The length of a slice `s` can be discovered by the built-in function `len`; unlike with arrays it may change during execution. The elements can be addressed by integer indices 0 through `len(s)-1`. The slice index of a given element may be less than the index of the same element in the underlying array.

类似于数组，切片是可索引的且拥有一个长度。切片 `s` 的长度可通过内建函数 `len`获取；不同于数组的是，切片可在执行过程中被改变， 其元素可通过整数（§下标） 0 到 `len(s)-1` 寻址。 给定元素的切片下标可能小于它在其基本数组中的下标。

A slice, once initialized, is always associated with an underlying array that holds its elements. A slice therefore shares storage with its array and with other slices of the same array; by contrast, distinct arrays always represent distinct storage.

切片一旦初始化，就总是伴随着一个包含其元素的基本数组。 因此，切片与其数组及其它本数组的切片共享存储； 与此相反，不同的数组总是表示其不同的存储。

The array underlying a slice may extend past the end of the slice. The *capacity* is a measure of that extent: it is the sum of the length of the slice and the length of the array beyond the slice; a slice of length up to that capacity can be created by `slicing' a new one from the original slice (§Slices). The capacity of a slice `a` can be discovered using the built-in function `cap(a)`.

切片的基本数组可扩展其切片的结尾。*容量* 是该扩展的量度： 它是切片的长度和切片往后数组的长度之和；长度达到其容量的切片可通过从原切片 （§Slices）'切下'一个新的来创建。 切片 `a` 的容量可使用内建函数 `cap(a)` 获取。

A new, initialized slice value for a given element type `T` is made using the built-in function `make`, which takes a slice type and parameters specifying the length and optionally the capacity:

给定元素类型 `T` 的一个新的，已初始化的切片值使用内建函数 `make`创建， 它需要一个切片类型和指定其长度与可选容量的形参：

```
make([]T, length)
make([]T, length, capacity)
```

A call to `make` allocates a new, hidden array to which the returned slice value refers. That is, executing

调用 `make`将分配一个被返回的切片值所引用的，新的、隐藏的数组。即，执行

```
make([]T, length, capacity)
```

produces the same slice as allocating an array and slicing it, so these two examples result in the same slice:

产生切片与分配数组后再对其进行切片相同，因此这两个例子的结果为相同的切片：

```
make([]int, 50, 100)
new([100]int)[0:50]
```

Like arrays, slices are always one-dimensional but may be composed to construct higher-dimensional objects. With arrays of arrays, the inner arrays are, by construction, always the same length; however with slices of slices (or arrays of slices), the lengths may vary dynamically. Moreover, the inner slices must be allocated individually (with `make`).

类似于数组，切片总是一维的，但可组合构造更高维的对象。 元素为数组的数组，根据其构造，其内部数组的长度始终相同； 然而元素为切片的切片（或元素为数组的切片），其长度会动态地改变。 此外，其内部的切片必须单独地（通过 `make`）分配。

## Struct types

## 结构类型

A struct is a sequence of named elements, called fields, each of which has a name and a type. Field names may be specified explicitly (IdentifierList) or implicitly (AnonymousField). Within a struct, non-blank field names must be unique.

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

结构是已命名的元素序列，被称为字段，其中每一个元素都有一个名字和类型。 字段名可显示地指定（标识符列表）或隐式地指定（匿名字段）。 在结构中，非空白字段名必须是唯一的。

```
结构类型 = "struct" "{" { 字段声明 ";" } "}" .
字段声明 = (标识符列表 类型 | 匿名字段) [ 标注 ] .
匿名字段 = [ "*" ] 类型名 .
标注     = 字符串字面 .
```

```
// 空结构.
struct {}

// 带6个字段的结构
struct {
        x, y int
        u float32
        _ float32  // 填充
        A *[]int
        F func()
}
```

A field declared with a type but no explicit field name is an *anonymous field*, also called an *embedded* field or an embedding of the type in the struct. An embedded type must be specified as a type name `T` or as a pointer to a non-interface type name `*T`, and `T` itself may not be a pointer type. The unqualified type name acts as the field name.

```
// A struct with four anonymous fields of type 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
}
```

通过有类型而无显式字段名声明的字段为 *匿名字段*，亦称为 *嵌入式* 字段或该结构中此种类型的嵌入。 这种字段类型必须作为一个类型名 `T` 或一个非接口类型名的指针 `*T`来实现， 且 `T` 本身不能为指针类型。未限定类型名的行为类似于字段名。

```
// 带类型为T1，*T2，P.T3和*P.T4的4个匿名字段的结构
struct {
	T1        // 字段名为T1
	*T2       // 字段名为T2
	P.T3      // 字段名为T3
	*P.T4     // 字段名为T4
	x, y int  // 字段名为x和y
}
```

The following declaration is illegal because field names must be unique in a struct type:

```
struct {
	T     // conflicts with anonymous field *T and *P.T
	*T    // conflicts with anonymous field T and *P.T
	*P.T  // conflicts with anonymous field T and *T
}
```

以下为非法声明，因为字段名在结构类型中必须是唯一的：

```
struct {
	T     // 与匿名字段*T及*P.T相冲突
	*T    // 与匿名字段T及*P.T相冲突
	*P.T  // 与匿名字段T及*T相冲突
}
```

A field or method `f` of an anonymous field in a struct `x` is called *promoted* if `x.f` is a legal selector that denotes that field or method `f`.

在结构 `x` 中，若 `x.f` 为表示字段或方法 `f` 的合法选择者，则匿名字段的字段或方法 `f` 即为*已提升*的。

Promoted fields act like ordinary fields of a struct except that they cannot be used as field names in composite literals of the struct.

已提升字段除了不能用作该结构复合字面中的字段名外， 其行为如同结构的一般字段。

Given a struct type `S` and a type named `T`, promoted methods are included in the method set of the struct as follows:

- If `S` contains an anonymous field `T`, the method sets of `S` and `*S` both include promoted methods with receiver `T`. The method set of `*S` also includes promoted methods with receiver `*T`.
- If `S` contains an anonymous field `*T`, the method sets of `S` and `*S` both include promoted methods with receiver `T` or `*T`.

给定结构类型 `S` 与名为 `T` 的类型，包含在结构方法集中的已提升方法如下：

- 若 `S` 包含一个匿名字段 `T`，则 `S` 与 `*S` 的方法集均包含带接收者 `T` 的已提升方法。`*S` 的方法集也包含带接收者 `*T` 的已提升方法。
- 若 `S` 包含匿名字段 `*T`，则 `S` 与 `*S` 的方法集均包含带接收者 `T` 或 `*T` 的已提升方法。

A field declaration may be followed by an optional string literal *tag*, which becomes an attribute for all the fields in the corresponding field declaration. The tags are made visible through a reflection interface but are otherwise ignored.

```
// A struct corresponding to the TimeStamp protocol buffer.
// The tag strings define the protocol buffer field numbers.
struct {
	microsec  uint64 "field 1"
	serverIP6 uint64 "field 2"
	process   string "field 3"
}
```

字段声明可后跟一个可选的字符串字面 *标注*，成为所有相应字段声明中字段的属性。 标注可通过 反射接口 获得，否则就会被忽略。

```
// 一个对应于时间戳协议缓存的结构.
// 标注字符串定义了协议缓存的字段号.
struct {
	microsec  uint64 "field 1"
	serverIP6 uint64 "field 2"
	process   string "field 3"
}
```

## Pointer types

## 指针类型

A pointer type denotes the set of all pointers to variables of a given type, called the *base type* of the pointer. The value of an uninitialized pointer is `nil`.

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

指针类型表示一个所有给定类型变量的指针的集，称为指针的 *基础类型*。 未初始化的指针的值为 nil。

```
指针类型 = "*" 基础类型 .
基础类型 = 类型 .
```

```
*Point
*[4]int
```

## Function types

## 函数类型

A function type denotes the set of all functions with the same parameter and result types. The value of an uninitialized variable of function type is nil.

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

函数类型表示所有带相同形参和返回类型的集。未初始化的函数类型变量的的值为 nil。

```
函数类型 = "func" 签名 .
签名     = 形参 [ 结果 ] .
结果     = 形参 | 类型 .
形参     = "(" [ 形参列表 [ "," ] ] ")" .
形参列表 = 形参声明 { "," 形参声明 } .
形参声明 = [ 标识符列表 ] [ "..." ] 类型 .
```

Within a list of parameters or results, the names (IdentifierList) must either all be present or all be absent. If present, each name stands for one item (parameter or result) of the specified type and all non-blank names in the signature must be unique. If absent, each type stands for one item of that type. Parameter and result lists are always parenthesized except that if there is exactly one unnamed result it may be written as an unparenthesized type.

在形参或结果的列表中，其名称（标识符列表）必须都存在或都不存在。 若存在，则每个名称代表一个指定类型的项（形参或结果），所有在签名中的非 空白名称必须是唯一的。 若不存在，则每个类型代表一个此类型的项。若恰好有一个未命名的值，它可能写作一个不加括号的类型， 除此之外，形参和结果的列表总是在括号中。

The final parameter in a function signature may have a type prefixed with .... A function with such a parameter is called *variadic* and may be invoked with zero or more arguments for that parameter.

函数签名中的最后一个形参可能有一个带 ... 前缀的类型。 带这样形参的函数被称为 *变参函数* 它可接受零个或多个实参的函数。

```
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)
```

## Interface types

## 接口类型

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

```
InterfaceType      = "interface" "{" { MethodSpec ";" } "}" .
MethodSpec         = MethodName Signature | InterfaceTypeName .
MethodName         = identifier .
InterfaceTypeName  = TypeName .
```

接口类型指定一个称为 *接口* 的 方法集。 接口类型变量可存储任何带方法集类型的值，该方法集为此接口的超集。 这种类型表示 *实现此接口*。未初始化的接口类型变量的值为 `nil`。

```
接口类型   = "interface" "{" { 方法实现 ";" } "}" .
方法实现   = 方法名 签名 | 接口类型名 .
方法名     = 标识符 .
接口类型名 = 类型名 .
```

As with all method sets, in an interface type, each method must have a unique name.

```
// A simple File interface
interface {
	Read(b Buffer) bool
	Write(b Buffer) bool
	Close()
}
```

对于所有的方法集，在一个接口类型中，每个方法必须有唯一的名字。

```
// 一个简单的File接口
interface {
	Read(b Buffer) bool
	Write(b Buffer) bool
	Close()
}
```

More than one type may implement an interface. For instance, if two types `S1` and `S2` have the method set

不止一个类型可实现同一接口。例如，若两个类型 `S1` 和 `S2` 拥有方法集

```
func (p T) Read(b Buffer) bool { return … }
func (p T) Write(b Buffer) bool { return … }
func (p T) Close() { … }
```

(where `T` stands for either `S1` or `S2`) then the `File` interface is implemented by both `S1` and `S2`, regardless of what other methods `S1` and `S2` may have or share.

（其中 `T` 代表 `S1` 或 `S2`） 那么 `File` 接口都将被 `S1` 和 `S2`所实现， 不论如何，方法 `S1` 和 `S2` 都会拥有或共享它。

A type implements any interface comprising any subset of its methods and may therefore implement several distinct interfaces. For instance, all types implement the *empty interface*:

类型可实现任何接口，包括任何其方法的子集，因此可能实现几个不同的接口。 例如，所有类型都实现了 *空接口*：

```
interface{}
```

Similarly, consider this interface specification, which appears within a type declaration to define an interface called `Lock`:

同样，考虑此接口的实现，它出现在类型声明 中以定义一个名为 `Lock` 的接口：

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

If `S1` and `S2` also implement

若 `S1` 和 `S2` 也实现

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

they implement the `Lock` interface as well as the `File` interface.

它们不仅会实现 `Lock` 还会实现 `File` 接口

An interface may use an interface type name `T` in place of a method specification. The effect, called embedding an interface, is equivalent to enumerating the methods of T explicitly in the interface.

```
type ReadWrite interface {
        Read(b Buffer) bool
        Write(b Buffer) bool
}

type File interface {
        ReadWrite  // same as enumerating the methods in ReadWrite
        Lock       // same as enumerating the methods in Lock
        Close()
}
```

一个接口可通过包含一个名为 T 的接口类型来代替一个方法的实现。 这称之为嵌入接口，其效果等价于在接口中显式枚举出 T 中的方法。

```
type ReadWrite interface {
        Read(b Buffer) bool
        Write(b Buffer) bool
}

type File interface {
        ReadWrite  // 等价于枚举ReadWrite中的方法
        Lock       // 等价于枚举Lock中的方法
        Close()
}
```

An interface type T may not embed itself or any interface type that embeds T, recursively.

```
// illegal: Bad cannot embed itself
type Bad interface {
        Bad
}

// illegal: Bad1 cannot embed itself using Bad2
type Bad1 interface {
        Bad2
}
type Bad2 interface {
        Bad1
}
```

接口类型 T 不能嵌入其自身或任何递归地嵌入 T 的接口类型。

```
// 非法: Bad不能嵌入其自身
type Bad interface {
        Bad
}

// 非法: Bad1不能通过Bad2嵌入其自身
type Bad1 interface {
        Bad2
}
type Bad2 interface {
        Bad1
}
```

## Map types

## 映射类型

A map is an unordered group of elements of one type, called the element type, indexed by a set of unique *keys* of another type, called the key type. The value of an uninitialized map is `nil`.

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

映射是一个同种类型元素的无序组，该类型称为元素类型； 映射通过另一类型唯一的 *键* 集索引，该类型称为键类型。 未初始化的映射值为 `nil`。

```
映射类型     = "map" "[" 键类型 "]" 元素类型 .
键类型      = 类型 .
```

The comparison operators == and != (§Comparison operators) must be fully defined for operands of the key type; thus the key type must not be a function, map, or slice. If the key type is an interface type, these comparison operators must be defined for the dynamic key values; failure will cause a run-time panic.

比较操作符 == 和 != （§比较操作符）必须由键类型的操作数完全定义； 因此键类型不能是函数，映射或切片。若该键类型为接口类型，这些比较运算符必须由动态键值定义； 失败将导致一个 运行时恐慌.

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

The number of map elements is called its length. For a map `m`, it can be discovered using the built-in function `len` and may change during execution. Elements may be added during execution using assignments and retrieved with index expressions; they may be removed with the `delete` built-in function.

元素的数量称为长度。 对于映射 `m`，其长度可使用内建函数 `len` 获取并可在执行时更改。元素可在执行时使用赋值来添加并通过 下标表达式 来检索；它们也可通过内建函数 `delete` 删除。

A new, empty map value is made using the built-in function `make`, which takes the map type and an optional capacity hint as arguments:

一个新的，空的映射值使用内建函数 `make` 创建， 它使该映射类型和可选容量作为实参提示：

```
make(map[string]int)
make(map[string]int, 100)
```

The initial capacity does not bound its size: maps grow to accommodate the number of items stored in them, with the exception of `nil` maps. A `nil` map is equivalent to an empty map except that no elements may be added.

初始容量不能限定它的大小：映射通过增长来适应存储在其中的项数，除了 `nil` 映射以外。 一个 `nil` 映射等价于一个空映射，只是 `nil` 映射不能添加元素。

## Channel types

## 信道类型

A channel provides a mechanism for two concurrently executing functions to synchronize execution and communicate by passing a value of a specified element type. The value of an uninitialized channel is `nil`.

```
ChannelType = ( "chan" [ "<-" ] | "<-" "chan" ) ElementType .
```

信道提供一种机制使两个并发执行的函数同步执行，并通过传递具体元素类型的值来通信。 未初始化的信道值为 `nil`。

```
信道类型 = ( "chan" [ "<-" ] | "<-" "chan" ) 元素类型 .
```

The `<-` operator specifies the channel *direction*, *send* or *receive*. If no direction is given, the channel is *bi-directional*. A channel may be constrained only to send or only to receive by conversion or assignment.

```
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
```

`<-` 操作符指定信道的 *方向*，*发送* 或 *接收*。 若没有给定方向，那么该信道就是 *双向的*。 信道可通过类型转换 或 赋值被强制为只发送或只接收。

```
chan T          // 可以被用来发送和接收类型T的值
chan<- float64  // 只能被用来发送浮点数
<-chan int      // 只能被用来接收整数
```

The `<-` operator associates with the leftmost `chan` possible:

```
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)
```

`<-` 操作符结合最左边的 `chan` 可能的方式：

```
chan<- chan int    // 等价于 chan<- (chan int)
chan<- <-chan int  // 等价于 chan<- (<-chan int)
<-chan <-chan int  // 等价于 <-chan (<-chan int)
chan (<-chan int)
```

A new, initialized channel value can be made using the built-in function `make`, which takes the channel type and an optional capacity as arguments:

一个新的，已初始化的信道值可使用内建函数 `make` 创建， 它接受信道类型和一个可选的容量作为实参：

```
make(chan int, 100)
```

The capacity, in number of elements, sets the size of the buffer in the channel. If the capacity is greater than zero, the channel is asynchronous: communication operations succeed without blocking if the buffer is not full (sends) or not empty (receives), and elements are received in the order they are sent. If the capacity is zero or absent, the communication succeeds only when both a sender and receiver are ready. A `nil` channel is never ready for communication.

容量根据元素的数量设置信道中缓存的大小。若容量大于零，则信道是异步的： 若缓存未满（发送）或非空（接收），则通信操作无阻塞成功，且元素在发送序列中被接收。 若容量为零或无，则只有当发送者和接收者都做好准备时通信才会成功。 `nil` 信道永远不会准备好通信。

A channel may be closed with the built-in function `close`; the multi-valued assignment form of the receive operator tests whether a channel has been closed.

信道可通过内建函数`close`关闭； 接收操作符的多值赋值形式可测试信道是否关闭。

## Properties of types and values

## 类型与值的性质

### Type identity

### 类型标识

Two types are either *identical* or *different*.

两个类型若非 *相同* 即为 *不同*。

Two named types are identical if their type names originate in the same TypeSpec. A named and an unnamed type are always different. Two unnamed types are identical if the corresponding type literals are identical, that is, if they have the same literal structure and corresponding components have identical types. In detail:

- Two array types are identical if they have identical element types and the same array length.
- Two slice types are identical if they have identical element types.
- Two struct types are identical if they have the same sequence of fields, and if corresponding fields have the same names, and identical types, and identical tags. Two anonymous fields are considered to have the same name. Lower-case field names from different packages are always different.
- Two pointer types are identical if they have identical base types.
- Two function types are identical if they have the same number of parameters and result values, corresponding parameter and result types are identical, and either both functions are variadic or neither is. Parameter and result names are not required to match.
- Two interface types are identical if they have the same set of methods with the same names and identical function types. Lower-case method names from different packages are always different. The order of the methods is irrelevant.
- Two map types are identical if they have identical key and value types.
- Two channel types are identical if they have identical value types and the same direction.

若两个已命名类型的类型名源自相同的类型实现，它们就是相同的。 一个已命名类型和一个未命名类型总不相同。若两个未命名类型其相应的类型字面相同， 那么它们的类型相同，即，它们的字面结构是否相同且其相应的组件类型是否相同。细节详述：

- 若两个数组类型其元素类型相同且长度相同，那么它们的类型相同。
- 若两个切片类型其元素类型相同，那么它们的类型相同。
- 若两个结构类型其字段序列相同，相应字段名相同，类型相同，标注相同，那么它们的类型相同。 两个匿名字段其名字被认为相同。出自不同包的小写字段名总不相同。
- 若两个指针类型其基础类型相同，那么它们的类型相同。
- 若两个函数类型其形参个数相同，返回值相同，相应形参类型相同，返回值类型相同， 两函数都可变或都不可变，那么它们的类型相同。形参和返回值名无需匹配。
- 若两个接口类型其方法集相同，名字相同，函数类型相同，那么它们的类型相同。 出自不同包的小写方法名总不相同。两接口类型是否相同与方法的次序无关。
- 若两个映射类型其键值类型相同，那么它们的类型相同。
- 若两个信道类型其值类型相同，方向相同，那么它们的类型相同。

Given the declarations

给定声明

```
type (
        T0 []string
        T1 []string
        T2 struct{ a, b int }
        T3 struct{ a, c int }
        T4 func(int, float64) *T0
        T5 func(x int, y float64) *[]string
)
```

these types are identical:

```
T0 and T0
[]int and []int
struct{ a, b *T5 } and struct{ a, b *T5 }
func(x int, y float64) *[]string and func(int, float64) (result *[]string)
```

这些类型是相同的：

```
T0 和 T0
[]int 和 []int
struct{ a, b *T5 } 和 struct{ a, b *T5 }
func(x int, y float64) *[]string 和 func(int, float64) (result *[]string)
```

`T0` and `T1` are different because they are named types with distinct declarations; `func(int, float64) *T0` and `func(x int, y float64) *[]string` are different because `T0` is different from `[]string`.

`T0` 和 `T1` 是不同的，因为它们由不同声明的类型命名； `func(int, float64) *T0` 和 `func(x int, y float64) *[]string` 是不同的， 因为 `T0` 不同于 `[]string`。

## Assignability

## 可赋值性

A value `x` is *assignable* to a variable of type `T` ("`x` is assignable to `T`") in any of these cases:

- `x`'s type is identical to `T`.
- `x`'s type `V` and `T` have identical underlying types and at least one of `V` or `T` is not a named type.
- `T` is an interface type and `x` implements `T`.
- `x` is a bidirectional channel value, `T` is a channel type, `x`'s type `V` and `T` have identical element types, and at least one of `V` or `T` is not a named type.
- `x` is the predeclared identifier `nil` and `T` is a pointer, function, slice, map, channel, or interface type.
- `x` is an untyped constant representable by a value of type `T`.

在下列情况下，值 `x` *可赋予* 类型为 `T` 的变量 （"`x` 可赋予 `T`"）：

- 当 `x` 的类型和 `T` 相同时。
- 当 `x` 的类型 `V` 和 `T` 有相同的 基本类型 且在 `V` 或 `T` 中至少有一个不是已命名类型时。
- 当 `T` 为接口类型且 `x` 实现了 `T`时。
- 当 `x` 为双向信道值、`T` 为信道类型、 `x` 的类型 `V` 和 `T` 的元素类型相同且在 `V` 或 `T` 中至少有一个不是已命名类型时。
- 当 `x` 为预声明标识符 `nil` 且 `T` 为指针、函数、切片、映射、通道或接口类型时。
- 当 `x` 为无类型化，可通过类型 `T` 的值来表示的 常量时。

Any value may be assigned to the blank identifier.

任何类型都可赋予空白标识符.

## Blocks

## 块

A *block* is a sequence of declarations and statements within matching brace brackets.

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

*块* 为一对大括号括住的声明和语句。

```
块 = "{" { 语句 ";" } "}" .
```

In addition to explicit blocks in the source code, there are implicit blocks:

1. The *universe block* encompasses all Go source text.
2. Each package has a *package block* containing all Go source text for that package.
3. Each file has a *file block* containing all Go source text in that file.
4. Each `if`, `for`, and `switch` statement is considered to be in its own implicit block.
5. Each clause in a `switch` or `select` statement acts as an implicit block.

除显式源码块外，还有隐式块：

1. *全域块* 包含所有的Go源码文本。
2. 每个包都有包含其所有Go源码文本的 *包块*。
3. 每个文件都有包含其所有Go源码文本的 *文件块*。
4. 每个 `if`、`for` 和 `switch` 语句都被视为处于其自身的隐式块中。
5. 每个 `switch` 或 `select` 语句中的子句其行为如同隐式块。

Blocks nest and influence scoping.

块可嵌套并会影响作用域。

## Declarations and scope

## 声明与作用域

A declaration binds a non-blank identifier to a constant, type, variable, function, or package. Every identifier in a program must be declared. No identifier may be declared twice in the same block, and no identifier may be declared in both the file and package block.

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

声明可将非空白标识符绑定到一个常量、类型、变量、函数或包。 在程序中，每个标识符都必须被声明。同一标识符不能在同一块中声明两次，且在文件与包块中不能同时声明。

```
声明 = 常量声明 | 类型声明 | 变量声明 .
顶级声明 = 声明 | 函数声明 | 方法声明 .
```

The *scope* of a declared identifier is the extent of source text in which the identifier denotes the specified constant, type, variable, function, or package.

已声明标识符的 *作用域* 即为该标识符所表示的具体常量、类型、变量、函数或包在源文本中的作用范围。

Go is lexically scoped using blocks:

1. The scope of a predeclared identifier is the universe block.
2. The scope of an identifier denoting a constant, type, variable, or function (but not method) declared at top level (outside any function) is the package block.
3. The scope of the package name of an imported package is the file block of the file containing the import declaration.
4. The scope of an identifier denoting a method receiver, function parameter, or result variable is the function body.
5. The scope of a constant or variable identifier declared inside a function begins at the end of the ConstSpec or VarSpec (ShortVarDecl for short variable declarations) and ends at the end of the innermost containing block.
6. The scope of a type identifier declared inside a function begins at the identifier in the TypeSpec and ends at the end of the innermost containing block.

Go使用块表示词法作用域：

1. 预声明标识符的作用域为全域块。
2. 在顶级（即在任何函数之外）声明的表示常量、类型、变量或函数 （而非方法）的标识符其作用域为该包块。
3. 已导入包的包名作用域为包含该导入声明的文件块。
4. 表示方法接收器、函数形参或返回值变量的标识符，其作用域为该函数体。
5. 在函数中声明为常量或变量的标识符，其作用域始于该函数中具体常量实现或变量实现 （ShortVarDecl表示短变量声明）的结尾，止于最内部包含块的结尾。
6. 在函数中声明为类型的标识符，其作用域始于该函数中具体类型实现的标识符， 止于最内部包含块的结尾。

An identifier declared in a block may be redeclared in an inner block. While the identifier of the inner declaration is in scope, it denotes the entity declared by the inner declaration.

在块中声明的标识符可在其内部块中重新声明。 当其内部声明的标识符在作用域中时，即表示其实体在该内部声明中声明。

The package clause is not a declaration; the package name does not appear in any scope. Its purpose is to identify the files belonging to the same package and to specify the default package name for import declarations.

包子句并非声明；包名不会出现在任何作用域中。 其目的是为了识别该文件是否属于相同的包并为导入声明指定默认包名。

### Label scopes

### 标签作用域

Labels are declared by labeled statements and are used in the `break`, `continue`, and `goto` statements (§Break statements, §Continue statements, §Goto statements). It is illegal to define a label that is never used. In contrast to other identifiers, labels are not block scoped and do not conflict with identifiers that are not labels. The scope of a label is the body of the function in which it is declared and excludes the body of any nested function.

标签通过标签语句声明，并用于 `break`、`continue` 和 `goto` 语句（§Break语句, §Continue语句, §Goto语句）。 定义不会使用的标签是非法的。与其它标识符相反，标签并不限定作用域且与非标签标识符并不冲突。 标签的作用域为除任何嵌套函数体外其声明的函数体。

### Blank identifier

### 空白标识符

The *blank identifier*, represented by the underscore character _, may be used in a declaration like any other identifier but the declaration does not introduce a new binding.

*空白标识符* 通过下划线字符 _ 表示， 它可像其它标识符一样用于声明，但该标识符不能传入一个新的绑定。

### Predeclared identifiers

### 预声明标识符

The following identifiers are implicitly declared in the universe block:

```
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
```

在全域块中，以下标识符是隐式声明的：

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

常量:
	true false iota

零值:
	nil

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

### Exported identifiers

### 已导出标识符

An identifier may be *exported* to permit access to it from another package. An identifier is exported if both:

1. the first character of the identifier's name is a Unicode upper case letter (Unicode class "Lu"); and
2. the identifier is declared in the package block or it is a field name or method name.

All other identifiers are not exported.

标识符可被 *导出* 以允许从另一个包访问。同时符合以下条件即为已导出标识符：

1. 标识符名的第一个字符为Unicode大写字母（Unicode类别“Lu”）；且
2. 该标识符在包块中已声明或为字段名或 方法名。

其它所有标识符均为未导出的。

## Uniqueness of identifiers

## 标识符的唯一性

Given a set of identifiers, an identifier is called *unique* if it is *different* from every other in the set. Two identifiers are different if they are spelled differently, or if they appear in different packages and are not exported. Otherwise, they are the same.

给定一个标识符集，若其中一个标识符*不同于*该集中的任一标识符，那么它就是*唯一的*。 若两个标识符拼写不同，或它们出现在不同的包中且未 导出，那么它们就是不同的。否则，它们就是相同的。

## Constant declarations

## 常量声明

A constant declaration binds a list of identifiers (the names of the constants) to the values of a list of constant expressions. The number of identifiers must be equal to the number of expressions, and the *n*th identifier on the left is bound to the value of the *n*th expression on the right.

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

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

常量声明将一个标识符（即常量名）列表绑定至一个常量表达式列表的值。 标识符的数量必须与表达式的数量相等，且左边第 *n* 个标识符会绑定至右边的第 *n* 个表达式的值。

```
常量声明   = "const" ( 常量实现 | "(" { 常量实现 ";" } ")" ) .
常量实现   = 标识符列表 [ [ 类型 ] "=" 表达式列表 ] .

标识符列表 = 标识符 { "," 标识符 } .
表达式列表 = 表达式 { "," 表达式 } .
```

If the type is present, all constants take the type specified, and the expressions must be assignable to that type. If the type is omitted, the constants take the individual types of the corresponding expressions. If the expression values are untyped constants, the declared constants remain untyped and the constant identifiers denote the constant values. For instance, if the expression is a floating-point literal, the constant identifier denotes a floating-point constant, even if the literal's fractional part is zero.

```
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
```

若该类型存在，所有常量都将获得该类型实现，且该表达式对于该类型必须是 可赋值的。若该类型被省略，则该常量将获得其对应表达式的具体类型。 若该表达式值为无类型化常量，则其余已声明无类型化常量与该常量标识符表示其常量值。 例如，若该表达式为浮点数字面，则该常量标识符表示一个浮点数常量，即使该字面的小数部分为零。

```
const Pi float64 = 3.14159265358979323846
const zero = 0.0       // 无类型化浮点常量
const (
        size int64 = 1024
        eof        = -1    // 无类型化整数常量
)
const a, b, c = 3, 4, "foo"  // a = 3, b = 4, c = "foo", 无类型化整数和字符串常量
const u, v float32 = 0, 3    // u = 0.0, v = 3.0
```

Within a parenthesized `const` declaration list the expression list may be omitted from any but the first declaration. 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. Together with the `iota` constant generator this mechanism permits light-weight declaration of sequential values:

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

在 const 后括号中的声明列表，除第一句声明外，任何表达式列表都可省略。 若前面第一个非空表达式有类型，那么这样的空列表等价于该表达式原文和类型的代换。 因此，省略表达式的列表等价于重复前面的列表。其标识符的数量必须与上一个表达式的数量相等。 连同 iota 常量生成器，该机制允许轻量级连续值声明：

```
const (
        Sunday = iota
        Monday
        Tuesday
        Wednesday
        Thursday
        Friday
        Partyday
        numberOfDays  // 该常量未导出
)
```

## Iota

Within a constant declaration, the predeclared identifier iota represents successive untyped integer constants. It is reset to 0 whenever the reserved word const appears in the source and increments after each ConstSpec. It can be used to construct a set of related constants:

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

const (
        a = 1 << iota  // a == 1 (iota has been reset)
        b = 1 << iota  // b == 2
        c = 1 << iota  // c == 4
)

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 (iota has been reset)
const y = iota  // y == 0 (iota has been reset)
```

在常量声明中预声明标识符 iota 表示连续的无类型化整数 常量。每当保留字 const 出现在源码中和每个 常量实现增量后，它都会被重置为0。它可被用来构造相关常量的集：

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

const (
        a = 1 << iota  // a == 1 （iota已重置）
        b = 1 << iota  // b == 2
        c = 1 << iota  // c == 4
)

const (
        u         = iota * 42  // u == 0     （无类型化整数常量）
        v float64 = iota * 42  // v == 42.0  （float64常量）
        w         = iota * 42  // w == 84    （无类型化整数常量）
)

const x = iota  // x == 0 （iota已重置）
const y = iota  // y == 0 （iota已重置）
```

Within an ExpressionList, the value of each `iota` is the same because it is only incremented after each ConstSpec:

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

在表达式列表中，每个 `iota` 的值都相同，因为它只在每个常量实现后增量。

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

This last example exploits the implicit repetition of the last non-empty expression list.

最后一个例子采用上一个非空表达式列表的隐式副本。

## Type declarations

## 类型声明

A type declaration binds an identifier, the *type name*, to a new type that has the same underlying type as an existing type. The new type is different from the existing type.

```
TypeDecl     = "type" ( TypeSpec | "(" { TypeSpec ";" } ")" ) .
TypeSpec     = identifier Type .
```

类型声明将标识符、*类型名* 绑定至一个与现存类型有相同的 基本类型的新类型。新类型不同于现有类型。

```
类型声明     = "type" ( 类型实现 | "(" { 类型实现 ";" } ")" ) .
类型实现     = 标识符 类型 .
```

```
type IntArray [16]int

type (
        Point struct{ x, y float64 }
        Polar Point
)

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

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

The declared type does not inherit any methods bound to the existing type, but the method set of an interface type or of elements of a composite type remains unchanged:

```
// 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 the base type of PtrMutex 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 anonymous field Mutex.
```

```
type PrintableMutex struct {
        Mutex
}

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

声明类型不继承任何方法绑定到现存类型， 但接口类型或复合类型元素的方法集保持不变：

```
// Mutex为带有Lock和Unlock两个方法的数据类型.
type Mutex struct         { /* Mutex字段 */ }
func (m *Mutex) Lock()    { /* Lock实现*/ }
func (m *Mutex) Unlock()  { /* Unlock实现*/ }

// NewMutex和Mutex拥有相同的组成，但它的方法集为空.
type NewMutex Mutex

// PtrMutex的基础类型的方法集保持不变.
// 但PtrMutex的方法集为空.
type PtrMutex *Mutex

// *PrintableMutex的方法集包含方法
// Lock和Unlock绑定至其匿名字段Mutex.
type PrintableMutex struct {
        Mutex
}

// MyBlock为与Block拥有相同方法集的接口类型.
type MyBlock Block
```

A type declaration may be used to define a different boolean, numeric, or string type and attach methods to it:

类型声明可用来定义不同的布尔值、数字或字符串类型并对其附上方法：

```
type TimeZone int

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

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

## Variable declarations

## 变量声明

A variable declaration creates a variable, binds an identifier to it and gives it a type and optionally an initial value.

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

变量声明将一个标识符绑定至一个创建的变量并赋予其类型和可选的初始值。

```
变量声明     = "var" ( 变量实现 | "(" { 变量实现 ";" } ")" ) .
变量实现     = 标识符列表 ( 类型 [ "=" 表达式列表 ] | "=" 表达式列表 ) .
```

```
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"
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]  // 映射检查；只与“found”有关
```

If a list of expressions is given, the variables are initialized by assigning the expressions to the variables (§Assignments) in order; all expressions must be consumed and all variables initialized from them. Otherwise, each variable is initialized to its zero value.

若给定一个表达式列表，则变量通过按顺序将该表达式赋予该变量（§赋值）来初始化； 所有表达式必须用尽且所有变量根据它们初始化。否则，每个变量初始化为其 零值。

If the type is present, each variable is given that type. Otherwise, the types are deduced from the assignment of the expression list.

若该类型已存在，每个变量都赋予该类型。否则，该类型根据该表达式列表赋值。

If the type is absent and the corresponding expression evaluates to an untyped constant, the type of the declared variable is as described in §Assignments.

若该类型不存在且其对应表达式计算结果为无类型化常量， 则该声明变量的类型由其赋值描述。

Implementation restriction: A compiler may make it illegal to declare a variable inside a function body if the variable is never used.

实现限制：若在函数体内声明不会使用的变量，编译器可能将其判定为非法。

## Short variable declarations

## 短变量声明

A *short variable declaration* uses the syntax:

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

*短变量声明* 使用此语法：

```
短变量声明 = 标识符列表 ":=" 表达式列表 .
```

It is a shorthand for a regular variable declaration with initializer expressions but no types:

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

```
i, j := 0, 10
f := func() int { return 7 }
ch := make(chan int)
r, w := os.Pipe(fd)  // os.Pipe() returns two values
_, y, _ := coord(p)  // coord() returns three values; only interested in y coordinate
```

它是有初始化表达式无类型化的常规变量声明的缩写：

```
"var" 标识符列表 = 表达式列表 .
```

```
i, j := 0, 10
f := func() int { return 7 }
ch := make(chan int)
r, w := os.Pipe(fd)  // os.Pipe() 返回两个值
_, y, _ := coord(p)  // coord() 返回三个值；只与和y同位的值相关
```

Unlike regular variable declarations, a short variable declaration may redeclare variables provided they were originally declared earlier in the same block with the same type, and at least one of the non-blank variables is new. As a consequence, redeclaration can only appear in a multi-variable short declaration. Redeclaration does not introduce a new variable; it just assigns a new value to the original.

```
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
```

不同于常规变量声明，在至少有一个非空白变量时， 短变量声明可在相同块中，对原先声明的变量以相同的类型重声明。因此，重声明只能出现在多变量短声明中。 重声明不能生成新的变量；它只能赋予新的值给原来的变量。

```
field1, offset := nextField(str, 0)
field2, offset := nextField(str, offset)  // 重声明 offset
a, a := 1, 2                              // 非法：重复声明了 a，或者若 a 在别处声明，但此处没有新的变量
```

Short variable declarations may appear only inside functions. In some contexts such as the initializers for `if`, `for`, or `switch` statements, they can be used to declare local temporary variables (§Statements).

短变量声明只能出现在函数内部。在某些情况下，例如初始化 `if`、 `for`、或 `switch` 语句时，它们可用来声明局部临时变量（§语句）。

## Function declarations

## 函数声明

A function declaration binds an identifier, the *function name*, to a function.

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

函数声明将标识符，即 *函数名* 绑定至函数。

```
函数声明 = "func" 函数名 签名 [ 函数体 ] .
函数名   = 标识符 .
函数体   = 块 .
```

A function declaration may omit the body. Such a declaration provides the signature for a function implemented outside Go, such as an assembly routine.

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

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

函数声明可省略函数体。这样的标识符为Go外部实现的函数提供签名，例如汇编例程。

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

func flushICache(begin, end uintptr)  // 外部实现
```

## Method declarations

## 方法声明

A method is a function with a *receiver*. A method declaration binds an identifier, the *method name*, to a method. It also associates the method with the receiver's *base type*.

```
MethodDecl   = "func" Receiver MethodName Signature [ Body ] .
Receiver     = "(" [ identifier ] [ "*" ] BaseTypeName ")" .
BaseTypeName = identifier .
```

方法为带 *接收者* 的函数。方法声明将标识符，即 *方法名* 绑定至方法。 它也将该接收者的 *基础类型* 关联至该方法。

```
方法声明    = "func" 接收者 方法名 签名 [ 函数体 ] .
接收者      = "(" [ 标识符 ] [ "*" ] 基础类型名 ")" .
```

```
基础类型名 = 标识符 .
```

The receiver type must be of the form T or *T where T is a type name. The type denoted by T is called the receiver *base type*; it must not be a pointer or interface type and it must be declared in the same package as the method. The method is said to be *bound* to the base type and the method name is visible only within selectors for that type.

接收者类型必须为形式 T 或 *T，其中 T 为类型名。 由 T 表示的类型称为接收者的 *基础类型*； 它不能为指针或接口类型且必须在同一包中声明为方法。 也就是说，该方法被 *绑定* 至基础类型且该方法名只对其内部此类型选择者可见。

A non-blank receiver identifier must be unique in the method signature. If the receiver's value is not referenced inside the body of the method, its identifier may be omitted in the declaration. The same applies in general to parameters of functions and methods.

For a base type, the non-blank names of methods bound to it must be unique. If the base type is a struct type, the non-blank method and field names must be distinct.

非空白接收器的标识符在该方法签名中必须是唯一的。 若该接收器的值并未在该方法体中引用，其标识符可在声明中省略。这同样适用于一般函数或方法的形参。

对于基础类型，方法绑定至该类型的非空白名称必须唯一。 若其基础类型为结构类型，则非空白方法与字段名不能相同。

Given type Point, the declarations

给定 Point 类型，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
}
```

bind the methods Length and Scale, with receiver type *Point, to the base type Point.

将接收者类型为 *Point 的方法 Length 和 Scale 绑定至基础类型 Point。

The type of a method is the type of a function with the receiver as first argument. For instance, the method Scale has type

方法的类型就是将接收者作为第一个实参的函数类型。例如，方法 Scale 拥有类型

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

However, a function declared this way is not a method.

然而，通过这种方式声明的函数不是方法。

## Expressions

## 表达式

An expression specifies the computation of a value by applying operators and functions to operands.

表达式通过将运算符和函数应用至操作数来指定值的计算。

### Operands

### 操作数

Operands denote the elementary values in an expression. An operand may be a literal, a (possibly qualified) identifier denoting a constant, variable, or function, a method expression yielding a function, or a parenthesized expression.

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

操作数表示表达式中的基本值。操作数可为字面，（可能为限定的）标识符可表示一个常量、变量或函数， 方法表达式可产生函数或者括号表达式。

```
操作数    = 字面 | 操作数名 | 方法表达式 | "(" 表达式 ")" .
字面      = 基本字面 | 复合字面 | 函数字面 .
基本字面 = 整数字面 | 浮点数字面 | 虚数字面 | 符文字面 | 字符串字面 .
操作数名 = 标识符 | 限定标识符.
```

## Qualified identifiers

## 限定标识符

A qualified identifier is an identifier qualified with a package name prefix. Both the package name and the identifier must not be blank.

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

限定标识符为使用包名前缀限定的标识符。包名与标识符均不能为空白的。

```
限定标识符 = 包名 "." 标识符 .
```

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
```

限定标识符用于访问另一个包中的标识符，它必须被导入。 标识符必须是已导出且在该包的包块中声明。

```
math.Sin        // 表示math包中的Sin函数
```

## Composite literals

## 复合字面

Composite literals construct values for structs, arrays, slices, and maps and create a new value each time they are evaluated. They consist of the type of the value followed by a brace-bound list of composite elements. An element may be a single expression or a key-value pair.

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

复合字面每次为结构、数组、切片、映射构造值，或创建一个新值时，它们都会被求值。 它们由值的类型后跟一个大括号括住的列表组成。元素可为单个表达式或一个键-值对。

```
复合字面 = 字面类型 字面值 .
字面类型 = 结构类型 | 数组类型 | "[" "..." "]" 元素类型 |
           切片类型 | 映射类型 | 类型名 .
字面值    = "{" [ 元素列表 [ "," ] ] "}" .
元素列表 = 元素 { "," 元素 } .
元素      = [ 键 ":" ] 值 .
键         = 字段名 | 元素索引 .
字段名    = 标识符 .
元素索引 = 表达式 .
值         = 表达式 | 字面值 .
```

The LiteralType must be a struct, array, slice, or map type (the grammar enforces this constraint except when the type is given as a TypeName). The types of the expressions must be assignable to the respective field, element, and key types of the LiteralType; there is no additional conversion. The key is interpreted as a field name for struct literals, an index for array and slice literals, and a key for map literals. For map literals, all elements must have a key. It is an error to specify multiple elements with the same field name or constant key value.

字面类型必须为结构、数组、切片或映射类型（语法规则强制实施此约束，除非该类型作为类型名给定）。 表达式的类型对于其各自的字段、元素以及该字面类型的键类型必须为可赋值的， 即没有附加转换。作为结构字面的字段名，即数组和切片的下标以及映射字面的键，其键是可解译的。 对于映射字面，所有元素都必须有键。指定多个具有相同字段名或常量键值的元素会产生一个错误。

For struct literals the following rules apply:

- A key must be a field name declared in the LiteralType.
- 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.
- If any element has a key, every element must have a 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.
- A literal may omit the element list; such a literal evaluates to the zero value for its type.
- It is an error to specify an element for a non-exported field of a struct belonging to a different package.

以下规则适用于结构字面：

- 键必须为字面类型中声明的字段名。
- 不包含任何键的元素列表必须按字段的声明顺序列出每个结构字段的元素。
- 若其中任何一个元素有键，那么每个元素都必须有键。
- 包含键的元素列表无需每个结构字段都有元素。被忽略的字段会获得零值
- 字面可忽略元素列表；这样的字面对其类型求值为零值。
- 为属于不同包的结构的未导出字段指定一个元素会产生一个错误。

Given the declarations

给定声明

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

one may write

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

可写为

```
origin := Point3D{}                            // Point3D 为零值
line := Line{origin, Point3D{y: -4, z: 12.3}}  // line.q.x 为零值
```

For array and slice literals the following rules apply:

- Each element has an associated integer index marking its position in the array.
- An element with a key uses the key as its index; the key must be a constant integer expression.
- An element without a key uses the previous element's index plus one. If the first element has no key, its index is zero.

以下规则适用于数组和切片字面：

- 在数组中每个元素都有与之对应的整数下标来标明它的位置。
- 带键的元素使用该键作为它的下标；键必须为常量整数表达式。
- 无键的元素使用上一个元素的下标加一。若第一个元素无键，则它的下标为零。

Taking the address of a composite literal (§Address operators) generates a pointer to a unique instance of the literal's value.

获取复合字面的地址（§地址操作符）就是为字面值的唯一实例生成一个指针。

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

The length of an array literal is the length specified in the LiteralType. If fewer elements than the length are provided in the literal, the missing elements are set to the zero value for the array element type. It is an error to provide elements with index values outside the index range of the array. The notation `...` specifies an array length equal to the maximum element index plus one.

数组字面的长度为字面类型指定的长度。 若元素少于字面提供的长度，则缺失的元素会置为该数组元素类型的零值。 向超出数组下标范围的下标值提供元素会产生一个错误。 记法 `...` 指定一个数组，其长度等于最大元素下标加一。

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

A slice literal describes the entire underlying array literal. Thus, the length and capacity of a slice literal are the maximum element index plus one. A slice literal has the form

切片字面描述全部的基本数组字面。因此，切片字面的长度和容量为其最大元素下标加一。切片字面具有形式

```
[]T{x1, x2, … xn}
```

and is a shortcut for a slice operation applied to an array:

它是切片操作应用到数组的捷径。

```
tmp := [n]T{x1, x2, … xn}
tmp[0 : n]
```

Within a composite literal of array, slice, or map type T, elements that are themselves composite literals may elide the respective literal type if it is identical to the element type of T. Similarly, elements that are addresses of composite literals may elide the &T when the element 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{{1.5, -3.5}, {0, 0}}  // same as [...]*Point{&Point{1.5, -3.5}, &Point{0, 0}}
```

在数组、切片或映射类型 T 的复合字面中，若其元素本身亦为复合字面， 且该复合字面的元素类型与 T 的相同，则可省略其各自的元素类型。 类似地，当元素类型为 *T 时，若其元素为复合字面的地址，则可省略 &T。

```
[...]Point{{1.5, -3.5}, {0, 0}}   // 等价于 [...]Point{Point{1.5, -3.5}, Point{0, 0}}
[][]int{{1, 2, 3}, {4, 5}}        // 等价于 [][]int{[]int{1, 2, 3}, []int{4, 5}}

[...]*Point{{1.5, -3.5}, {0, 0}}  // 等价于 [...]*Point{&Point{1.5, -3.5}, &Point{0, 0}}
```

A parsing ambiguity arises when a composite literal using the TypeName form of the LiteralType appears between the keyword and the opening brace of the block of an "if", "for", or "switch" statement, because the braces surrounding the expressions in the literal are confused with those introducing the block of statements. To resolve the ambiguity in this rare case, the composite literal must appear within parentheses.

当复合字面使用字面类型的类型名形式时，若它出现在关键字 "if"、"for" 或 "switch" 语句及其开大括号之间，就会产生解析歧义。因为在该字面中， 表达式外围的大括号会和那些语句块前的混淆。为解决此罕见情况中的歧义，该复合字面必须出现在小括号中。

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

Examples of valid array, slice, and map literals:

```
// 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,
}
```

有效的数组、切片和映射字面的例子：

```
// 素数列表
primes := []int{2, 3, 5, 7, 9, 2147483647}

// 若 ch 为元音则 vowels[ch] 为 true
vowels := [128]bool{'a': true, 'e': true, 'i': true, 'o': true, 'u': true, 'y': true}

// 数组 [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}

// 平均律以Hz为单位的频率（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

## 函数字面

A function literal represents an anonymous function. It consists of a specification of the function type and a function body.

```
FunctionLit = FunctionType Body .
```

函数字面表示匿名函数。它由函数类型和函数体的规范组成。

```
函数字面 = 函数类型 函数体 .
```

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

A function literal can be assigned to a variable or invoked directly.

函数字面可赋予一个变量或直接调用。

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

Function literals are *closures*: they may refer to variables defined in a surrounding function. Those variables are then shared between the surrounding function and the function literal, and they survive as long as they are accessible.

*闭包* 的函数字面：它们可引用定义在外围函数中的变量。 那些变量共享于外围函数与函数字面之间，并且只要它们可访问就会继续存在。

## Primary expressions

## 主表达式

Primary expressions are the operands for unary and binary expressions.

```
PrimaryExpr =
	Operand |
	Conversion |
	BuiltinCall |
	PrimaryExpr Selector |
	PrimaryExpr Index |
	PrimaryExpr Slice |
	PrimaryExpr TypeAssertion |
	PrimaryExpr Call .

Selector       = "." identifier .
Index          = "[" Expression "]" .
Slice          = "[" [ Expression ] ":" [ Expression ] "]" .
TypeAssertion  = "." "(" Type ")" .
Call           = "(" [ ArgumentList [ "," ] ] ")" .
ArgumentList   = ExpressionList [ "..." ] .
```

主表达式为一元和二元表达式的操作数。

```
主表达式 =
	操作数 |
	类型转换 |
	内建调用 |
	主表达式 选择者 |
	主表达式 下标 |
	主表达式 切片 |
	主表达式 类型断言 |
	主表达式 调用 .

选择者   = "." 标识符 .
下标     = "[" 表达式 "]" .
切片     = "[" [ 表达式 ] ":" [ 表达式 ] "]" .
类型断言 = "." "(" 类型 ")" .
调用     = "(" [ 实参列表 [ "," ] ] ")" .
实参列表 = 表达式列表 [ "..." ] .
```

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

## Selectors

## 选择者

For a primary expression `x` that is not a package name, the *selector expression*

对于不为包名的主表达式 `x`， *选择其表达式*

```
x.f
```

denotes the field or method `f` of the value `x` (or sometimes `*x`; see below). The identifier `f` is called the (field or method) *selector*; it must not be the blank identifier. The type of the selector expression is the type of `f`. If `x` is a package name, see the section on qualified identifiers.

表示值 `x`（有时为 `*x`，见下）的字段或方法 `f`。 标识符 `f` 称为（字段或方法）*选择者*，它不能为空白标识符。 该选择者表达式的类型即为 `f` 的类型。若 `x` 为包名， 见限定标识符的相关章节。

A selector `f` may denote a field or method `f` of a type `T`, or it may refer to a field or method `f` of a nested anonymous field of `T`. The number of anonymous fields traversed to reach `f` is called its *depth* in `T`. The depth of a field or method `f` declared in `T` is zero. The depth of a field or method `f` declared in an anonymous field `A` in `T` is the depth of `f` in `A` plus one.

选择者 `f` 可代表类型为 `T` 的字段或方法 `f`， 或引用 `T` 中嵌套匿名字段的字段或方法 `f`。 在 `T` 中遍历区域 `f` 的匿名字段所得的数量称为它的*深度*。 以 `T` 声明的字段或方法 `f` 的深度为0。 在 `T` 中以匿名字段 `A` 声明的字段或方法 `f` 的深度 为 `f` 在 `A` 中的深度加1。

The following rules apply to selectors:

1. For a value `x` of type `T` or `*T` where `T` is not an interface type, `x.f` denotes the field or method at the shallowest depth in `T` where there is such an `f`. If there is not exactly one `f` with shallowest depth, the selector expression is illegal.
2. For a variable `x` of type `I` where `I` is an interface type, `x.f` denotes the actual method with name `f` of the value assigned to `x`. If there is no method with name `f` in the method set of `I`, the selector expression is illegal.
3. In all other cases, `x.f` is illegal.
4. If `x` is of pointer or interface type and has the value `nil`, assigning to, evaluating, or calling `x.f` causes a run-time panic.

以下规则适用于选择者：

1. 对于非接口类型 `T` 或 `*T` 的值 `x`， `x.f` 中的 `f` 表示在 `T` 中最浅深度的字段或方法。 若并非只有一个 `f`，该选择者表达式即为非法的。
2. 对于接口类型 `I` 的变量 `x`，`x.f` 表示赋予 `x` 的值的名为 `f` 的真实方法。若在 `I` 的方法集中没有名为 `f` 的方法，该选择者即为非法的。
3. 其它情况下，所有 `x.f` 均为非法的。
4. 若 `x` 为指针或接口类型且值为 `nil`，对 `x.f` 进行赋值、求值或调用会产生 运行时恐慌.

Selectors automatically dereference pointers to structs. If `x` is a pointer to a struct, `x.y` is shorthand for `(*x).y`; if the field `y` is also a pointer to a struct, `x.y.z` is shorthand for `(*(*x).y).z`, and so on. If `x` contains an anonymous field of type `*A`, where `A` is also a struct type, `x.f` is a shortcut for `(*x.A).f`.

选择者会自动解引用指向结构的指针。 若 `x` 为指向结构的指针，`x.y` 即为 `(*x).y` 的缩写； 若字段 `y` 亦为指向结构的指针，`x.y.z` 即为 `(*(*x).y).z` 的缩写, 以此类推。 若 `x` 包含类型为 `*A` 的匿名字段，且 `A` 亦为结构类型， `x.f` 即为 `(*x.A).f` 的缩写。

For example, given the declarations:

```
type T0 struct {
        x int
}

func (recv *T0) M0()

type T1 struct {
        y int
}

func (recv T1) M1()

type T2 struct {
        z int
        T1
        *T0
}

func (recv *T2) M2()

var p *T2  // with p != nil and p.T0 != nil
```

例如，给定声明：

```
type T0 struct {
        x int
}

func (recv *T0) M0()

type T1 struct {
        y int
}

func (recv T1) M1()

type T2 struct {
        z int
        T1
        *T0
}

func (recv *T2) M2()

var p *T2  // 其中 p != nil 且 p.T0 != nil
```

one may write:

可以写：

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

p.M2()  // (*p).M2()
p.M1()  // ((*p).T1).M1()
p.M0()  // ((*p).T0).M0()
```

## Index expressions

## 下标表达式

A primary expression of the form

形式为

```
a[x]
```

denotes the element of the array, slice, string or map `a` indexed by `x`. The value `x` is called the *index* or *map key*, respectively. The following rules apply:

的主表达式表示数组、切片、字符串或映射 `a` 的元素通过 `x` 检索。 值 `x` 称为 *下标* 或 *映射键*。以下规则适用于其对应的类型：

If `a` is not a map:

- the index `x` must be an integer value; it is *in range* if `0 <= x < len(a)`, otherwise it is *out of range*
- a constant index must be non-negative and representable by a value of type `int`

若 `a` 并非一个映射：

- 下标 `x` 必须为整数值；若 `0 <= x < len(a)` 则该下标在*界内*，否则即为*越界*
- a 常量下标必须为可表示成 `int` 类型的值

For `a` of type `A` or `*A` where `A` is an array type:

- a constant index must be in range
- if `a` is `nil` or if `x` is out of range at run time, a run-time panic occurs

- a[x] is the array element at index x and the type of a[x] is the element type of A

对于数组类型 A 或 *A 的 a：

- 常量下标必在界内
- 若 a 为 nil 或 x 在运行时越界， 就会引发一个运行时恐慌
- a[x] 是下标为 x 的数组元素，且 a[x] 的类型即为 A 的元素类型

For a of type S where S is a slice type:

- if the slice is nil or if x is out of range at run time, a run-time panic occurs
- a[x] is the slice element at index x and the type of a[x] is the element type of S

对于切片类型 S 的 a：

- 若该切片为 nil，或 x 在运行时越界， 就会引发一个运行时恐慌
- a[x] 是下标为 x 的切片元素且 a[x] 的类型为 S 的元素类型

For a of type T where T is a string type:

- a constant index must be in range if the string a is also constant
- if x is out of range at run time, a run-time panic occurs
- a[x] is the byte at index x and the type of a[x] is byte
- a[x] may not be assigned to

对于类型为字符串类型 T 的 a：

- 若字符串 a 也为常量，常量下标必在界内。
- 若 x 超出范围，就会出现一个运行时恐慌
- a[x] 为下标 x 的字节且 a[x] 的类型为 byte
- a[x] 不可赋值

For a of type M where M is a map type:

- x's type must be assignable to the key type of M
- if the map contains an entry with key x, a[x] is the map value with key x and the type of a[x] is the value type of M
- if the map is nil or does not contain such an entry, a[x] is the zero value for the value type of M

对于类型为映射类型 M 的 a：

- x 的类型必须可赋值至 M 的键类型
- 若映射包含键为 x 的项，则 a[x] 为键 x 的映射值， 且 a[x] 的类型为 M 的值类型
- 若映射为 nil 或不包含这样的项， a[x] 为 M 值类型的零值

Otherwise a[x] is illegal.

否则 a[x] 即为非法的。

An index expression on a map a of type map[K]V may be used in an assignment or initialization of the special form

在类型为 map[K]V 的映射 a 中，下标表达式可使用特殊形式

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

where the result of the index expression is a pair of values with types (V, bool). In this form, the value of ok is true if the key x is present in the map, and false otherwise. The value of v is the value a[x] as in the single-result form.

赋值或初始化，该下标表达式结果的类型为 (V, bool) 的值对。 在此形式中，若键 x 已在映射中，则 ok 的值为 true， 否则即为 false。 v 的值为 a[x] 的单值形式。

Assigning to an element of a nil map causes a run-time panic.

向 nil 映射的元素赋值会引发运行时恐慌

## Slices

## 切片

For a string, array, pointer to array, or slice a, the primary expression

对于字符串，数组，数组指针或切片 a，主表达式

```
a[low : high]
```

constructs a substring or slice. The indices `low` and `high` select which elements appear in the result. The result has indices starting at 0 and length equal to `high` - `low`. After slicing the array `a`

会构造一个字串或切片。下标 `low` 和 `high` 则选出哪些元素出现在结果中。 该结果的下标起始于 0 且长度等于 `high` - `low`。 在切下数组 `a`

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

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

之后，切片 `s` 的类型为 `[]int`，长度为 3，容量为 4，且元素

```
s[0] == 2
s[1] == 3
s[2] == 4
```

For convenience, any of the indices may be omitted. A missing `low` index defaults to zero; a missing `high` index defaults to the length of the sliced operand:

```
a[2:]  // same a[2 : len(a)]
a[:3]  // same as a[0 : 3]
a[:]   // same as a[0 : len(a)]
```

为方便起见，任何下标都可省略。略去的 `low` 下标默认为零； 略去的 `high` 下标默认为已切下的操作数的长度：

```
a[2:]  // 等价于 a[2 : len(a)]
a[:3]  // 等价于 a[0 : 3]
a[:]   // 等价于 a[0 : len(a)]
```

For arrays or strings, the indices `low` and `high` are *in range* if `0 <= low <= high <= len(a)`, otherwise they are *out of range*. For slices, the upper index bound is the slice capacity `cap(a)` rather than the length. A constant index must be non-negative and representable by a value of type `int`. If both indices are constant, they must satisfy `low <= high`. If `a` is `nil` or if the indices are out of range at run time, a run-time panic occurs.

对于数组或字符串，若 `0 <= low <= high <= len(a)` 下标 `low` 和 `high` 即在*界内*，否则即在界外。 对于切片，其上界为该切片的容量 `cap(a)` 而非长度。常量下标必为非负值， 且可表示为 `int` 类型的值。若其下标也为常量，它们必定满足 `low <= high`。 若 `a` 为 `nil` 或其下标在运行时越界，就会引发一个运行时恐慌。

If the sliced operand is a string or slice, the result of the slice operation is a string or slice of the same type. If the sliced operand is an array, it must be addressable and the result of the slice operation is a slice with the same element type as the array.

若已切下操作数为字符串或切片，该切片操作的结果即为相同类型的字符串或切片。 若已切下操作数为数组，它必须为可寻址的， 且该切片操作的结果为以相同元素类型作为数组的切片。

## Type assertions

## 类型断言

For an expression `x` of interface type and a type `T`, the primary expression

对于接口类型的表达式 `x` 与类型 `T`，主表达式

```
x.(T)
```

asserts that `x` is not `nil` and that the value stored in `x` is of type `T`. The notation `x.(T)` is called a *type assertion*.

断言 `x` 不为 `nil` 且存储于 `x` 中的值其类型为 `T`。 记法 `x.(T)` 称为 *类型断言*。

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

更确切地说，若 `T` 为非接口类型，`x.(T)` 断言 `x` 的动态类型 与 `T`相同。在此情况下，`T` 必须实现 `x` 的（接口）类型，除非其类型断言由于无法为 `x` 存储类型为 `T` 的值而无效。若 `T` 为接口类型， `x.(T)` 则断言 `x` 的动态类型实现了接口 `T`。

If the type assertion holds, the value of the expression is the value stored in `x` and its type is `T`. If the type assertion is false, a run-time panic occurs. In other words, even though the dynamic type of `x` is known only at run time, the type of `x.(T)` is known to be `T` in a correct program.

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

type I interface { m() }
var y I
s := y.(string)        // illegal: string does not implement I (missing method m)
r := y.(io.Reader)     // r has type io.Reader and y must implement both I and io.Reader
```

若该类型断言成立，该表达式的值即为存储于 `x` 中的值，且其类型为 `T`。若该类型断言不成立， 就会出现一个运行时恐慌。换句话说，即使 `x` 的动态类型只能在运行时可知，在正确的程序中，`x.(T)` 的类型也可知为 `T`。

```
var x interface{} = 7  // x 拥有动态类型 int 与值 7
i := x.(int)           // i 拥有类型 int 与值 7

type I interface { m() }
var y I
s := y.(string)        // 非法: string 没有实现 I（缺少方法 m）
r := y.(io.Reader)     // r 拥有 类型 io.Reader 且 y 必须同时实现了 I 和 io.Reader
```

If a type assertion is used in an assignment or initialization of the form

若类型断言以

```
v, ok = x.(T)
v, ok := x.(T)
var v, ok = x.(T)
```

the result of the assertion is a pair of values with types `(T, bool)`. If the assertion holds, the expression returns the pair `(x.(T), true)`; otherwise, the expression returns `(Z, false)` where `Z` is the zero value for type `T`. No run-time panic occurs in this case. The type assertion in this construct thus acts like a function call returning a value and a boolean indicating success.

的形式用于赋值或初始化，该断言的结果即为类型为 `(T, bool)` 的值对。 若该断言成立，该表达式返回值对 `(x.(T), true)`；否则，该表达式返回 `(Z, false)`， 其中 `Z` 为类型为 `T` 的零值。此种情况不会产生运行时恐慌。 类型断言在这种构造中，其行为类似于函数调用返回一个值与一个布尔值以表示成功。

## Calls

## 调用

Given an expression `f` of function type `F`,

给定函数类型为 `F` 的表达式 `f`，

```
f(a1, a2, … an)
```

calls `f` with arguments `a1, a2, … an`. Except for one special case, arguments must be single-valued expressions assignable to the parameter types of `F` and are evaluated before the function is called. The type of the expression is the result type of `F`. 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
```

以实参 `a1, a2, … an` 调用 `f`。 除一种特殊情况外，实参必须为 可赋予 `F` 的形参类型的单值表达式，且在该函数被调用前求值。 该表达式的类型为 `F` 的返回类型。 方法调用也类似，只不过使用接收者类型值的选择者操作来指定方法。

```
math.Atan2(x, y)  // 函数调用
var pt *Point
pt.Scale(3.5)     // 带接收者 pt 的方法调用
```

In a function call, the function value and arguments are evaluated in the usual order. After they are evaluated, the parameters of the call are passed by value to the function and the called function begins execution. The return parameters of the function are passed by value back to the calling function when the function returns.

在函数调用中，函数值与实参按一般顺序求值。 在它们求值后，该调用的形参传值至该函数，被调用函数开始执行。 当函数返回时，该函数的返回形参将值传回调用函数。

Calling a `nil` function value causes a run-time panic.

调用 nil 函数值会引发 运行时恐慌。

As a special case, if the return parameters of a function or method g are equal in number and individually assignable to the parameters of another function or method f, then the call f(g(*parameters_of_g*)) will invoke f after binding the return values of g to the parameters of f in order. The call of f must contain no parameters other than the call of g. If f has a final ... parameter, it is assigned the return values of g that remain after assignment of regular parameters.

作为一种特殊情况，若函数或方法 g 的返回形参在数量上等于函数或方法 f 的形参， 且分别可赋予它，那么调用 f(g(*g的形参*)) 将在依序绑定 g 的返回值至 f 的形参后引用 f。除 g 的调用外， f 的调用必须不包含任何形参。若 f 的最后有 ... 形参， 它在常规的形参赋值后，可被赋予 g 余下的返回值。

```
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")
}
```

A method call x.m() is valid if the method set of (the type of) x contains m and the argument list can be assigned to the parameter list of m. If x is addressable and &x's method set contains m, x.m() is shorthand for (&x).m():

若 x（的类型）的方法集包含 m， 且其实参列表可赋予 m 的形参列表，方法调用 x.m() 即为有效的。 若 x 为 可寻址的且 &x 的方法集包含 m， x.m() 即为 (&x).m() 的简写：

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

There is no distinct method type and there are no method literals.

其中即没有明显的方法类型，也没有方法字面。

## Passing arguments to ... parameters

## 传递实参至...形参

If f is variadic with final parameter type ...T, then within the function the argument is equivalent to a parameter of type []T. At each call of f, the argument passed to the final parameter is a new slice of type []T whose successive elements are the actual arguments, which all must be assignable to the type T. The length of the slice is therefore the number of arguments bound to the final parameter and may differ for each call site.

若 f 为最后带有形参类型 ...T 的可变参函数， 那么在该函数中，实参等价于类型为 []T 的形参。 对于每一个 f 的调用，传递至最后形参的实参为类型为 []T 的一个新切片， 其连续的元素即为实际的实参，它们必须都可赋予类型 T。 因此，该切片的长度为绑定至最后形参的实参的个数，且对于每一个调用位置可能都不同。

Given the function and call

给定函数和调用

```
func Greeting(prefix string, who ...string)
Greeting("hello:", "Joe", "Anna", "Eileen")
```

within Greeting, who will have the value []string{"Joe", "Anna", "Eileen"}

在 Greeting 中，who 将拥有值 []string{"Joe", "Anna", "Eileen"}

If the final argument is assignable to a slice type []T, it may be passed unchanged as the value for a ...T parameter if the argument is followed by .... In this case no new slice is created.

若最后的实参可赋予类型为 []T 的切片且后跟着 ...， 它可能作为 ...T 形参的值不变而被传入。

Given the slice s and call

给定切片 s 与调用

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

within `Greeting`, `who` will have the same value as `s` with the same underlying array.

在 `Greeting` 中，`who` 将作为与 `s` 一样的值拥有与其相同的基本数组。

## Operators

## 操作符

Operators combine operands into expressions.

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

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

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

操作符与操作数结合成为表达式。

```
表达式     = 一元表达式 | 表达式 二元操作符 一元表达式 .
一元表达式 = 主表达式 | 一元操作符 一元表达式 .

二元操作符 = "||" | "&&" | 关系操作符 | 加法操作符 | 乘法操作符 .
关系操作符 = "==" | "!=" | "<" | "<=" | ">" | ">=" .
加法操作符 = "+" | "-" | "|" | "^" .
乘法操作符 = "*" | "/" | "%" | "<<" | ">>" | "&" | "&^" .

一元操作符 = "+" | "-" | "!" | "^" | "*" | "&" | "<-" .
```

Comparisons are discussed elsewhere. For other binary operators, the operand types must be identical unless the operation involves shifts or untyped constants. For operations involving constants only, see the section on constant expressions.

比较操作将在别处讨论。 对于其它二元操作符，操作数的类型必须相同， 除非该操作包含移位或无类型化常量。

Except for shift operations, if one operand is an untyped constant and the other operand is not, the constant is converted to the type of the other operand.

除移位操作外，若其中一个操作数为无类型化常量而另一个不是， 该常量需类型转换为另一个操作数的类型。

The right operand in a shift expression must have unsigned integer type or be an untyped constant that can be converted to unsigned integer type. If the left operand of a non-constant shift expression is an untyped constant, the type of the constant is what it would be if the shift expression were replaced by its left operand alone; the type is `int` if it cannot be determined from the context (for instance, if the shift expression is an operand in a comparison against an untyped constant).

```
var s uint = 33
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
var n = 1.0<<s != 0    // 1.0 has type int; n == false if ints are 32bits in size
var o = 1<<s == 2<<s   // 1 and 2 have type int; o == true if ints are 32bits in size
var p = 1<<s == 1<<33  // illegal if ints are 32bits in size: 1 has type int, but 1<<33 overflows int
var u = 1.0<<s         // illegal: 1.0 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
```

移位表达式中的右操作数必须为无符号整数，或可转换为无符号整数的无类型化常量。 若非常量移位表达式的左操作数为无类型化常量，且该移位表达式已被其左操作数独自取代， 则该移位表达式的类型将变为该常量的类型；若其类型不能从上下文中判定（例如，若该移位表达式在针对无类型化常量的比较操作中）， 则为 `int`类型。

```
var s uint = 33
var i = 1<<s           // 1 的类型为 int
var j int32 = 1<<s     // 1 的类型为 int32; j == 0
var k = uint64(1<<s)   // 1 的类型为 uint64; k == 1<<33
var m int = 1.0<<s     // 1.0 的类型为 int
var n = 1.0<<s != 0    // 1.0 的类型为 int; 若int的大小为 32位, 则 n == false
var o = 1<<s == 2<<s   // 1 与 2 的类型为 int; 若 int 的大小为32位, 则 o == true
var p = 1<<s == 1<<33  // 若 int 的大小为32位即为非法: 虽然 1 的类型为 int, 但 1<<33 溢出了 int
var u = 1.0<<s         // 非法: 1.0 的类型为 float64, 不能移位
var v float32 = 1<<s   // 非法: 1 的类型为 float32, 不能移位
var w int64 = 1.0<<33  // 1.0<<33 为常量移位表达式
```

## Operator precedence

## 操作符优先级

Unary operators have the highest precedence. As the ++ and -- operators form statements, not expressions, they fall outside the operator hierarchy. As a consequence, statement *p++ is the same as (*p)++.

一元操作符拥有最高优先级。 ++ 和 -- 操作符是语句，而非表达式，它们不属于运算符一级。 因此，语句 *p++ 等价于 (*p)++。

There are five precedence levels for binary operators. Multiplication operators bind strongest, followed by addition operators, comparison operators, && (logical AND), and finally || (logical OR):

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

二元操作符有五种优先级。 乘法操作符结合性最强，其次为加法操作符、比较操作符、&&（逻辑与）， 最后为 ||（逻辑或）：

```
  优先级        操作符
    5             *  /  %  <<  >>  &  &^
    4             +  -  |  ^
    3             ==  !=  <  <=  >  >=
    2             &&
    1             ||
```

Binary operators of the same precedence associate from left to right. For instance, x / y * z is the same as (x / y) * z.

相同优先级的二元操作符从左到右结合。 例如，x / y * z 等价于 (x / y) * z。

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

## Arithmetic operators

## 算数操作符

Arithmetic operators apply to numeric values and yield a result of the same type as the first operand. The four standard arithmetic operators (+, -, *, /) apply to integer, floating-point, and complex types; + also applies to strings. All other arithmetic operators apply to integers only.

```
+    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 << unsigned integer
>>   right shift            integer >> unsigned integer
```

算数操作符适用于数值，并产生相同类型的结果作为第一个操作数。四个基本算数操作符 （+, -, *, /）适用于整数、浮点数和复数类型； + 也适用于字符串。其它所有算数操作符仅适用于整数。

```
+    和              integers, floats, complex values, strings
-    差              integers, floats, complex values
*    积              integers, floats, complex values
/    商              integers, floats, complex values
%    余              integers
```

```
&    按位与          integers
|    按位或          integers
^    按位异或        integers
&^   位清除（与非）  integers

<<   向左移位        integer << unsigned integer
>>   向右移位        integer >> unsigned integer
```

Strings can be concatenated using the + operator or the += assignment operator:

字符串可使用 + 操作符连结或 += 赋值操作符：

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

String addition creates a new string by concatenating the operands.

字符串加法通过连结操作数创建一个新的字符串。

For two integer values x and y, the integer quotient `q = x / y` and remainder `r = x % y` satisfy the following relationships:

```
x = q*y + r  and  |r| < |y|
```

with `x / y` truncated towards zero ("truncated division").

对于两个整数值 x 与 y，整数除法 `q = x / y` 和取余 `r = x % y` 满足以下关系：

```
x = q*y + r  且  |r| < |y|
```

将 `x / y` 向零截断（“除法截断”）。

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

As an exception to this rule, if the dividend x is the most negative value for the int type of x, the quotient `q = x / -1` is equal to x (and `r = 0`).

作为该规则的一个例外，若被除数 x 为 x 的int类型的最小负值，商 `q = x / -1` 等于 x（且 `r = 0`）。

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

If the divisor is a constant, it must not be zero. If the divisor is zero at run time, a run-time panic occurs. If the dividend is non-negative and the divisor is a constant power of 2, the division may be replaced by a right shift, and computing the remainder may be replaced by a bitwise AND operation:

若被除数为常量，则它必不为零。若被除数在运行时为零， 就会出现一个运行时恐慌。若被除数为非负数，且除数为2的常量次幂， 则该除法可被向右移位取代，且计算其余数可被按位“与”操作取代：

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

The shift operators shift the left operand by the shift count specified by the right operand. They implement arithmetic shifts if the left operand is a signed integer and logical shifts if it is an unsigned integer. 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.

移位操作符通过右操作数指定的移位计数来移位左操作数。若左操作数为带符号整数，它们就执行算术移位； 若左操作数为无符号整数，它们则执行逻辑移位。移位计数没有上界。 若左操作数移 n 位，其行为如同移 1 位 n 次。 按照其结果，`x << 1` 等价于 `x*2`，而 `x >> 1` 等价于 `x/2` 但向负无穷大截断。

For integer operands, the unary operators +, -, and ^ are defined as follows:

```
+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
```

对于整数操作数，一元操作符 +、- 和 ^ 的定义如下：

```
+x                 即为 0 + x
-x    相反数        即为 0 - x
^x    按位补码      即为 m ^ x  对于无符号的 x, m = "所有位置为1"
                                                对于带符号的 x, m = -1
```

For floating-point and complex numbers, +x is the same as x, while -x is the negation of x. The result of a floating-point or complex division by zero is not specified beyond the IEEE-754 standard; whether a run-time panic occurs is implementation-specific.

对于浮点数与复数来说，+x 等价于 x，而 -x 则为 x 的相反数。 浮点数或复数除以零的结果仅满足 IEEE-754 标准而无额外保证；是否会出现运行时恐慌取决于具体实现。

## Integer overflow

## 整数溢出

For unsigned integer values, the operations +, -, *, and << are computed modulo $2^n$, where *n* is the bit width of the unsigned integer's type (§Numeric types). Loosely speaking, these unsigned integer operations discard high bits upon overflow, and programs may rely on ``wrap around''.

对于无符号整数值，操作 +、-、* 和 << 均被计算为取模 $2^n$，其中 *n* 为该无符号整数类型的位宽（§数值类型）。不严格地说，这些无符号整数操作抛弃高位向上溢出，程序可依赖这种形式的“回卷”。

For signed integers, the operations +, -, *, and << may legally overflow and the resulting value exists and is deterministically defined by the signed integer representation, the operation, and its operands. No exception is raised as a result of overflow. A compiler may not optimize code under the assumption that overflow does not occur. For instance, it may not assume that x < x + 1 is always true.

对于带符号整数，操作 +、-、* 和 << 可合法溢出， 而由此产生的值会继续存在，并由该带符号整数表现、操作、与其操作数决定性地定义。 除溢出外没有例外会导致此情况。在溢出不会发生的假定情况下编译器可能不会优化代码。 例如，我们无法假定 x < x + 1 总为真。

## Comparison operators

## 比较操作符

Comparison operators compare two operands and yield a boolean value.

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

比较操作符比较两个操作数并产生一个布尔值。

```
==    等于
!=    不等于
<     小于
<=    小于等于
>     大于
>=    大于等于
```

In any comparison, the first operand must be assignable to the type of the second operand, or vice versa.

在任何比较中，第一个操作数必须为可赋予第二个操作数的类型，反之亦然。

The equality operators == and != apply to operands that are *comparable*. The ordering operators <, <=, >, and >= apply to operands that are *ordered*. These terms and the result of the comparisons are defined as follows:

- Boolean values are comparable. Two boolean values are equal if they are either both true or both false.
- Integer values are comparable and ordered, in the usual way.
- Floating point values are comparable and ordered, as defined by the IEEE-754 standard.
- Complex values are comparable. Two complex values u and v are equal if both real(u) == real(v) and imag(u) == imag(v).

- String values are comparable and ordered, lexically byte-wise.
- Pointer values are comparable. Two pointer values are equal if they point to the same variable or if both have value `nil`. Pointers to distinct zero-size variables may or may not be equal.
- Channel values are comparable. Two channel values are equal if they were created by the same call to `make` (§Making slices, maps, and channels) or if both have value `nil`.
- Interface values are comparable. Two interface values are equal if they have identical dynamic types and equal dynamic values or if both have value `nil`.
- A value `x` of non-interface type `X` and a value `t` of interface type `T` are comparable when values of type `X` are comparable and `X` implements `T`. They are equal if `t`'s dynamic type is identical to `X` and `t`'s dynamic value is equal to `x`.
- Struct values are comparable if all their fields are comparable. Two struct values are equal if their corresponding non-blank fields are equal.
- Array values are comparable if values of the array element type are comparable. Two array values are equal if their corresponding elements are equal.

相等性操作符 == 和 != 适用于*可比较*操作数。 顺序操作符 <、<=、> 和 >= 适用于*有序的*操作数。这些比较操作的关系和值定义如下：

- 布尔值之间可比较。若两个布尔值同为 `true` 或同为 `false`，它们即为相等。
- 通常情况下，整数值之间可比较或排序。
- 根据 IEEE-754 标准的定义，浮点数值之间可比较或排序。
- 复数值之间可比较。对于两个复数值 `u` 与 `v`， 若 `real(u) == real(v)` 且 `imag(u) == imag(v)`，它们即为相等。
- 根据按字节词法，字符串值之间可比较或排序。
- 指针值之间可比较。若两个指针指向相同的值或其值同为 `nil`，它们即为相等。 指向明显为零大小变量的指针可能相等也可能不相等。
- 信道值可比较。若两个信道值通过相同的 `make` 调用 （§创建切片、映射和信道）创建或同为 `nil` 值，它们即为相等。
- 接口值可比较。若两个接口值拥有相同的动态类型与相等的动态值，或同为 `nil` 值，它们即为相等。
- 当非接口类型 `X` 的值可比较且 `X` 实现了 `T` 时， 非接口类型 `X` 的值 `x` 与接口类型 `T` 的值 `t` 则可比较。 若 `t` 的动态类型与 `X` 相同且 `t` 动态值等于 `x`，它们即为相等。
- 若两个结构值的所有字段可比较，它们即可比较。若其相应的非空白字段相等，它们即为相等。
- 若两个数组元素类型的值可比较，则数组值可比较。若其相应的元素相等，它们即为相等。

A comparison of two interface values with identical dynamic types causes a run-time panic if values of that type are not comparable. This behavior applies not only to direct interface value comparisons but also when comparing arrays of interface values or structs with interface-valued fields.

两个动态类型相同的接口值进行比较，若该动态类型的值不可比较，将引发一个运行时恐慌。 此行为不仅适用于直接的接口值比较，当比较接口值的数组或接口值作为字段的结构时也适用。

Slice, map, and function values are not comparable. However, as a special case, a slice, map, or function value may be compared to the predeclared identifier `nil`. Comparison of pointer, channel, and interface values to `nil` is also allowed and follows from the general rules above.

切片、映射和函数值同类型之间不可比较。然而，作为一种特殊情况，切片、映射或函数值可与预声明标识符 `nil` 进行比较。指针、信道和接口值与 `nil` 之间的比较也允许并遵循上面的一般规则。

The result of a comparison can be assigned to any boolean type. If the context does not demand a specific boolean type, the result has type `bool`.

```
type MyBool bool

var x, y int
var (
	b1 MyBool = x == y // result of comparison has type MyBool
	b2 bool   = x == y // result of comparison has type bool
	b3        = x == y // result of comparison has type bool
)
```

比较的结果可赋予任何布尔类型。若上下文无需特殊的布尔类型，其结果的类型即为 `bool`。

```
type MyBool bool

var x, y int
var (
	b1 MyBool = x == y // 比较结果为类型 MyBool
	b2 bool   = x == y // 比较结果为类型 bool
	b3        = x == y // 比较结果为类型 bool
)
```

## Logical operators

## 逻辑操作符

Logical operators apply to boolean values and yield a result of the same type as the operands. The right operand is evaluated conditionally.

```
&&    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"
```

逻辑操作符适用于布尔值并根据操作数产生一个相同类型的结果。右操作数有条件地求值。

```
&&    条件与    p && q  即  “若 p 成立则判断 q 否则返回 false”
||    条件或    p || q  即  “若 p 成立则返回 true 否则判断 q”
!     非         !p      即  “非 p”
```

## Address operators

## 地址操作符

For an operand x of type T, the address operation &x generates a pointer of type *T to x. The operand must be *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. As an exception to the addressability requirement, x may also be a (possibly parenthesized) composite literal.

对于类型为 T 的操作数 x，地址操作符 &x 将生成一个类型为 *T 的指针指向 x。操作数必须*可寻址*，即，变量、间接指针、切片索引操作， 或可寻址结构操作数的字段选择者，或可寻址数组的数组索引操作均不可寻址。作为可寻址性需求的例外， x 也可为（可能带有括号的）复合字面.

For an operand x of pointer type *T, the pointer indirection *x denotes the value of type T pointed to by x. If x is nil, an attempt to evaluate *x will cause a run-time panic.

对于指针类型为 *T 的操作数 x，间接指针 *x 表示类型为 T 的值指向 x。若 x 为 nil， 尝试求值 *x 将会引发运行时恐慌。

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

## Receive operator

## 接收操作符

For an operand ch of channel type, the value of the receive operation <-ch is the value received from the channel ch. The channel direction must permit receive operations, and the type of the receive operation is the element type of the channel. The expression blocks until a value is available. Receiving from a nil channel blocks forever. Receiving from a closed channel always succeeds, immediately returning the element type's zero value.

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

对于信道类型的操作数 ch，接收操作符 <-ch 的值即为从信道 ch 接收的值。该信道的方向必须允许接收操作， 且该接收操作的类型即为该信道的元素类型。该值前的表达式块是有效的。 从 nil 信道接收将永远阻塞。从已关闭的信道接收将总是成功， 它会立刻返回其元素类型的零值

```
v1 := <-ch
v2 = <-ch
f(<-ch)
<-strobe  // 在时钟脉冲和丢弃接收值之前等待
```

A receive expression used in an assignment or initialization of the form

接收表达式以

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

yields an additional result of type bool reporting whether the communication succeeded. The value of ok is true if the value received was delivered by a successful send operation to the channel, or false if it is a zero value generated because the channel is closed and empty.

的形式用于赋值或初始化将产生一个类型为 bool 的附加结果，来报告通信是否成功。 若接收的值由一次成功向信道发送的操作发出的，则 ok 的值为 true； 若接收的值是由于信道被关闭或为空而产生的零值，则为 false。

## Method expressions

## 方法表达式

If M is in the method set of type T, T.M is a function that is callable as a regular function with the same arguments as M prefixed by an additional argument that is the receiver of the method.

```
MethodExpr    = ReceiverType "." MethodName .
ReceiverType  = TypeName | "(" "*" TypeName ")" | "(" ReceiverType ")" .
```

若 M 在类型为 T 的方法集中， T.M 即为可调用函数，如同常规函数 带相同实参和 M 以该方法接收者的附加实参作为前缀 。

```
方法表达式 = 接收者类型 "." 方法名 .
接收者类型 = 类型名 | "(" "*" 类型名 ")" | "(" 接收者类型 ")" .
```

Consider a struct type T with two methods, Mv, whose receiver is of type T, and Mp, whose receiver is of type *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
```

考虑一个类型为 T 的结构和两个方法， Mv，其接收者的类型为 T， Mp，其接收者的类型为 *T。

```
type T struct {
        a int
}
func (tv  T) Mv(a int) int         { return 0 }  // 值接收者
func (tp *T) Mp(f float32) float32 { return 1 }  // 指针接收者
var t T
```

The expression

表达式

```
T.Mv
```

yields a function equivalent to Mv but with an explicit receiver as its first argument; it has signature

将产生一个等价于 Mv 的方法，但它将一个显式的接收者作为其第一个实参；它拥有签名

```
func(tv T, a int) int
```

That function may be called normally with an explicit receiver, so these five invocations are equivalent:

该函数可通过显式的接收者正常调用，因此以下五个调用是等价的：

```
t.Mv(7)
T.Mv(t, 7)
(T).Mv(t, 7)
f1 := T.Mv; f1(t, 7)
f2 := (T).Mv; f2(t, 7)
```

Similarly, the expression

同样，表达式

```
(*T).Mp
```

yields a function value representing Mp with signature

将产生一个带签名

```
func(tp *T, f float32) float32
```

For a method with a value receiver, one can derive a function with an explicit pointer receiver, so

的函数值来表示 Mp。 对于带值接收者的方法，它可以派生出一个带显式指针接收者的函数，因此

```
(*T).Mv
```

yields a function value representing Mv with signature

将产生一个带签名

```
func(tv *T, a int) int
```

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.

的函数值来表示 Mv。这样的函数会通过接收者间接创建一个值作为该基本方法的接收者来传递； 该方法不会覆盖地址已经传入该函数调用的值。

The final case, 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.

最后一种情况，一个值接收者函数对于一个指针接收者方法是非法的， 因为指针接收者方法不在该值类型的方法集中。

Function values derived from methods are called with function call syntax; the receiver is provided as the first argument to the call. That is, given f := T.Mv, f is invoked as f(t, 7) not t.f(7). To construct a function that binds the receiver, use a closure.

从方法中派生出的函数值被函数调用语法调用。接收者作为第一个该调用的实参被提供。 也就是说，给定 f := T.Mv，f 将作为 f(t, 7) 而非 t.f(7) 被调用。要构造一个绑定了接收者的函数，需使用闭包。

It is legal to derive a function value from a method of an interface type. The resulting function takes an explicit receiver of that interface type.

从类型为接口的方法中派生出一个函数值是合法的。产生的函数将获得一个该接口类型的显式接收者。

## Conversions

## 类型转换

Conversions are expressions of the form T(x) where T is a type and x is an expression that can be converted to type T.

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

类型转换是形式为 T(x) 的表达式，其中 T 为类型，而 x 是可转换为类型 T 的表达式。

```
类型转换 = 类型 "(" 表达式 [ "," ] ")" .
```

If the type starts with an operator it must be parenthesized: If the type starts with the operator * or <-, or the keyword func, it must be parenthesized:

```
*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())
```

若类型以操作符 *、<- 或关键字 func 开始则必须加上括号：

```
*Point(p)        // 等价于 *(Point(p))
(*Point)(p)      // p 被转换为 (*Point)
<-chan int(c)    // 等价于 <-(chan int(c))
(<-chan int)(c)  // c 被转换为 (<-chan int)
func()(x)        // 函数签名 func() x
(func())(x)      // x 被转换为 (func())
```

A constant value x can be converted to type T in any of these cases:

- x is representable by a value of type T.
- x is an integer constant and T is a string type. The same rule as for non-constant x applies in this case (§Conversions to and from a string type).

常量值 x 在这些情况下可转换为类型 T：

- x 可表示为类型为 T 的值。
- x 为整数常量且 T 为 字符串类型。 有关非常量 x 的相同规则也适用于此种情况（§字符串类型的转换）。

Converting a constant yields a typed constant as result.

```
uint(iota)               // iota value of type uint
float32(2.718281828)     // 2.718281828 of type float32
complex128(1)            // 1.0 + 0.0i of type complex128
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
```

转换一个常量将产生一个类型化的常量作为结果。

```
uint(iota)               // 类型为 uint 的 iota 值
float32(2.718281828)     // 类型为 float32 的 2.718281828
complex128(1)            // 类型为 complex128 的 1.0 + 0.0i
string('x')              // 类型为 string 的 "x"
string(0x266c)           // 类型为 string 的 "♬"
MyString("foo" + "bar")  // 类型为 MyString 的 "foobar"
string([]byte{'a'})      // 非常量: []byte{'a'} 不为常量
(*int)(nil)              // 非常量: nil 不为常量, *int 不为布尔、 数值或字符串类型
int(1.2)                 // 非法: 1.2 不能表示为 int
string(65.0)             // 非法: 65.0 不为整数常量
```

A non-constant value x can be converted to type T in any of these cases:

- x is assignable to T.
- x's type and T have identical underlying types.
- x's type and T are unnamed pointer types and their pointer base types have identical underlying types.
- x's type and T are both integer or floating point types.
- x's type and T are both complex types.
- x is an integer or a slice of bytes or runes and T is a string type.
- x is a string and T is a slice of bytes or runes.

非常量值 x 在这些情况下可转换为类型 T：

- 当 x 可赋予 T时。
- 当 x 的类型与 T 拥有相同的基本类型时。
- 当 x 的类型与 T 为未命名指针类型，且它们的指针基础类型拥有相同的基本类型时。
- 当 x 的类型与 T 同为整数或浮点数类型时。
- 当 x 的类型与 T 同为复数类型时。
- 当 x 为整数、字节切片或符文切片且 T 为字符串类型时。
- 当 x 为字符串且 T 为字节切片或符文切片时。

Specific rules apply to (non-constant) conversions between numeric types or to and from a string type. These conversions may change the representation of x and incur a run-time cost. All other conversions only change the type but not the representation of x.

具体规则应用于（非常量）与数值类型之间，或在字符串类型之间转换。 这些类型转换会改变 x 的表示并引发运行时的代价。 其它转换只改变类型而不改变 x 的表示。

There is no linguistic mechanism to convert between pointers and integers. The package `unsafe` implements this functionality under restricted circumstances.

没有语言机制能在指针和整数之间转换。包 `unsafe` 可在受限情况下实现此功能。

Implementation restriction: For backward-compatibility with the Go 1 language specification, a compiler may accept non-parenthesized literal function types in conversions where the syntax is unambiguous.

实现限制：为了与Go1语言规范向后兼容，编译器可在语法明确的转换中， 接受无括号的字面函数类型。

### Conversions between numeric types

### 数值类型间的转换

For the conversion of non-constant numeric values, the following rules apply:

1. When converting between integer types, if the value is a signed integer, it is sign extended to implicit infinite precision; otherwise it is zero extended. It is then truncated to fit in the result type's size. For example, if `v := uint16(0x10F0)`, then `uint32(int8(v)) == 0xFFFFFFF0`. The conversion always yields a valid value; there is no indication of overflow.
2. When converting a floating-point number to an integer, the fraction is discarded (truncation towards zero).
3. When converting an integer or floating-point number to a floating-point type, or a complex number to another complex type, the result value is rounded to the precision specified by the destination type. For instance, the value of a variable `x` of type `float32` may be stored using additional precision beyond that of an IEEE-754 32-bit number, but float32(x) represents the result of rounding `x`'s value to 32-bit precision. Similarly, `x + 0.1` may use more than 32 bits of precision, but `float32(x + 0.1)` does not.

对于非常量数值类型的类型转换，以下规则适用：

1. 当在整数类型间转换时，若该值为无符号整数，其符号将扩展为隐式无限精度，反之为零扩展。 然后截断以符合该返回类型的大小。例如，若 `v := uint16(0x10F0)`，则 `uint32(int8(v)) == 0xFFFFFFF0`。类型转换总产生有效值，且无溢出指示。
2. 当转换浮点数为整数时，小数部分将被丢弃（向零截断）。
3. 当转换整数或浮点数为浮点类型，或转换复数类型为另一个复数类型时，其返回值将舍入至目标类型指定的精度。 例如，类型为 `float32` 的变量 `x` 的值可能使用超出 IEEE-754 标准32位数的额外精度来存储，但 float32(x) 表示将 `x` 的值舍入为32位精度的结果。同样，`x + 0.1` 会使用超过32位的精度，但 `float32(x + 0.1)` 却不会。

In all non-constant conversions involving floating-point or complex values, if the result type cannot represent the value the conversion succeeds but the result value is implementation-dependent.

在所有涉及非常量浮点数或复数值的类型转换中，若该返回类型不能表示该转换成功的值，则该返回值取决于具体实现。

### Conversions to and from a string type

### 字符串类型的转换

1. Converting a signed or unsigned integer value to a string type yields a string containing the UTF-8 representation of the integer. Values outside the range of valid Unicode code points are converted to `"\uFFFD"`.

```
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"
```

2. Converting a slice of bytes to a string type yields a string whose successive bytes are the elements of the slice. If the slice value is `nil`, the result is the empty string.

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

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

3. Converting a slice of runes to a string type yields a string that is the concatenation of the individual rune values converted to strings. If the slice value is `nil`, the result is the empty string.

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

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

4. Converting a value of a string type to a slice of bytes type yields a slice whose successive elements are the bytes of the string. If the string is empty, the result is `[]byte(nil)`.

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

5. Converting a value of a string type to a slice of runes type yields a slice containing the individual Unicode code points of the string. If the string is empty, the result is `[]rune(nil)`.

```
[]rune(MyString("白鵬翔"))  // []rune{0x767d, 0x9d6c, 0x7fd4}
MyRunes("白鵬翔")           // []rune{0x767d, 0x9d6c, 0x7fd4}
```

1. 将有符号或无符号整数值转换为字符串类型将产生一个包含UTF-8表示的该整数的字符串。 有效Unicode码点范围之外的值将转换为 `"\uFFFD"`。

```
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"
```

2. 将字节切片转换为字符串类型将产生一个连续字节为该切片元素的字符串。 若该切片值为 `nil`，则其结果为空字符串。

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

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

3. 将符文切片转换为字符串类型将产生一个已转换为字符串的单个符文值的串联字符串。 若该切片值为 `nil`，则其结果为空字符串。

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

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

4. 将字符串类型值转换为字节类型切片将产生一个连续元素为该字符串字节的切片。 若该字符串为空，其结果为 `[]byte(nil)`。

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

5. 将字符串类型的值转换为符文类型切片将产生一个包含该字符串单个Unicode码点的切片。 若该字符串为空，其结果为 `[]rune(nil)`。

```
[]rune(MyString("白鵬翔"))  // []rune{0x767d, 0x9d6c, 0x7fd4}
MyRunes("白鵬翔")           // []rune{0x767d, 0x9d6c, 0x7fd4}
```

## Constant expressions

## 常量表达式

Constant expressions may contain only constant operands and are evaluated at compile time.

常量表达式可只包含常量操作数并在编译时求值。

Untyped boolean, numeric, and string constants may be used as operands wherever it is legal to use an operand of boolean, numeric, or string type, respectively. Except for shift operations, if the operands of a binary operation are different kinds of untyped constants, the operation and, for non-boolean operations, the result use the kind that appears later in this list: integer, rune, floating-point, complex. For example, an untyped integer constant divided by an untyped complex constant yields an untyped complex constant.

无类型化布尔、数值和字符串常量可被用作操作数，无论使用布尔、数值或字符串类型的操作数是否合法。 除移位操作外，若二元操作的操作数是不同种类的无类型化常量，对于非布尔操作，该操作与其结果使用出现在此列表中较后的种类： 整数、符文、浮点数、复数。例如，由无类型化复数常量分离的无类型化整数常量将产生一个无类型化复数常量。

A constant comparison always yields an untyped boolean constant. If the left operand of a constant shift expression is an untyped constant, the result is an integer constant; otherwise it is a constant of the same type as the left operand, which must be of integer type (§Arithmetic operators). Applying all other operators to untyped constants results in an untyped constant of the same kind (that is, a boolean, integer, floating-point, complex, or string constant).

```
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   // f == 0     (type 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)
```

常量比较总是产生无类型化布尔常量。若常量移位表达式 的左操作数为无类型化常量，则该结果为整数常量；否则为与左操作数类型相

同的常量，左操作数必须为整数类型（§算术操作符）。将其它所有操作符应用于同种类（即，布尔、整数、浮点数、复数或字符串常量）无类型化常量的结果：

```
const a = 2 + 3.0          // a == 5.0   （无类型化浮点数常量）
const b = 15 / 4           // b == 3     （无类型化整数常量）
const c = 15 / 4.0         // c == 3.75  （无类型化浮点数常量）
const Θ float64 = 3/2      // Θ == 1.0   (类型为float64, 3/2 是整数除法)
const Π float64 = 3/2.     // Π == 1.5   (类型为float64, 3/2. 是浮点数除法)
const d = 1 << 3.0         // d == 8     （无类型化整数常量）
const e = 1.0 << 3         // e == 8     （无类型化整数常量）
const f = int32(1) << 33   // f == 0     （类型为int32）
const g = float64(2) >> 1  // 非法       （float64(2)为无类型化浮点数常量）
const h = "foo" > "bar"    // h == true  （无类型化布尔常量）
const j = true             // j == true  （无类型化布尔常量）
const k = 'w' + 1          // k == 'x'   （无类型化符文常量）
const l = "hi"             // l == "hi"  （无类型化字符串常量）
const m = string(k)        // m == "x"   （类型为string）
const Σ = 1 - 0.707i       //            （无类型化复数常量）
const Δ = Σ + 2.0e-4       //            （无类型化复数常量）
const Φ = iota*1i - 1/1i   //            （无类型化复数常量）
```

Applying the built-in function `complex` to untyped integer, rune, or floating-point constants yields an untyped complex constant.

```
const ic = complex(0, c)   // ic == 3.75i  (untyped complex constant)
const iΘ = complex(0, Θ)   // iΘ == 1.5i   (type complex128)
```

将内建函数 `complex` 应用于无类型化整数、符文或浮点数常量将产生一个无类型化复数常量。

```
const ic = complex(0, c)   // ic == 3.75i （无类型化复数常量）
const iΘ = complex(0, Θ)   // iΘ == 1.5i  （类型为complex128）
```

Constant expressions are always evaluated exactly; intermediate values and the constants themselves may require precision significantly larger than supported by any predeclared type in the language. The following are legal declarations:

常量表达式总是精确地求值；中间值与该常量本身可能需要明显大于该语言中任何预声明类型所支持的精度。以下为合法声明：

```
const Huge = 1 << 100         // Huge == 1267650600228229401496703205376  (untyped integer constant)
const Four int8 = Huge >> 98  // Four == 4                                (type int8)
```

The divisor of a constant division or remainder operation must not be zero:

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

常量除法或求余的除数必不能为零：

```
3.14 / 0.0   // 非法：除以零
```

The values of *typed* constants must always be accurately representable as values of the constant type. The following constant expressions are illegal:

```
uint(-1)     // -1 cannot be represented as a uint
int(3.14)    // 3.14 cannot be represented as an int
int64(Huge)  // 1267650600228229401496703205376 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)
```

*类型化*常量的值必须总是可作为该常量类型的值被准确地表示。以下为非法常量表达式：

```
uint(-1)     // -1 无法表示为 uint
int(3.14)    // 3.14 无法表示为 int
int64(Huge)  // 1267650600228229401496703205376 无法表示为 int64
Four * 300   // 操作数 300 无法表示为 int8（Four 的类型）
Four * 100   // 其结果 400 无法表示为 int8（Four 的类型）
```

The mask used by the unary bitwise complement operator ^ matches the rule for non-constants: the mask is all 1s for unsigned constants and -1 for signed and untyped constants.

```
^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
```

用于一元按位补码操作符 ^ 的屏蔽与非常量相匹配的规则：对于无符号常量屏蔽全为1， 而对于带符号或无类型化常量为-1。

```
^1         // 无类型化整数常量，等于-2
uint8(^1)  // 非法，等价于 uint8(-2)，-2无法表示为 uint8
^uint8(1)  // 类型化uint8常量，等价于0xFF ^ uint8(1) = uint8(0xFE)
int8(^1)   // 等价于int8(-2)
^int8(1)   // 等价于-1 ^ int8(1) = -2
```

Implementation restriction: A compiler may use rounding while computing untyped floating-point or complex constant expressions; see the implementation restriction in the section on constants. This rounding may cause a floating-point constant expression to be invalid in an integer context, even if it would be integral when calculated using infinite precision.

实现限制：在计算无类型化浮点数或复数常量表达式时，编译器可能使用舍入，参考章节常量 的实现限制。次舍入可能会导致浮点常量表达式在整数上下文中无效，即使在它使用无限精度计算时会成为整体。

## Order of evaluation

## 求值顺序

When evaluating the operands of an expression, assignment, or return statement, all function calls, method calls, and communication operations are evaluated in lexical left-to-right order.

当对一个表达式、赋值或Return语句进行求值时， 所有函数调用、方法调用以及通信操作均按从左到右的词法顺序求值。

For example, in the assignment

例如，在赋值

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

the function calls and communication happen in the order `f()`, `h()`, `i()`, `j()`, `<-c`, `g()`, and `k()`. However, the order of those events compared to the evaluation and indexing of `x` and the evaluation of `y` is not specified.

```
a := 1
f := func() int { a = 2; return 3 }
x := []int{a, f()}  // x may be [1, 3] or [2, 3]: evaluation order between a and f() is not specified
```

中，函数调用与通信按顺序 `f()`、`h()`、`i()`、`j()`、 `<-c`、`g()` 和 `k()` 发生。然而，相较于这些事件的顺序，`x` 的求值与索引及 `y` 的求值并未指定。

```
a := 1
f := func() int { a = 2; return 3 }
x := []int{a, f()}  // x可能为[1, 3]或[2, 3]: a与f()之间的求职顺序并未指定
```

Floating-point operations within a single expression are evaluated according to the associativity of the operators. Explicit parentheses affect the evaluation by overriding the default associativity. In the expression `x + (y + z)` the addition `y + z` is performed before adding `x`.

单表达式中的浮点数操作根据该操作符的结合性求值。显式的圆括号通过覆盖默认结合性来影响求值。 在表达式 `x + (y + z)` 中，加法 `y + z` 会在与 `x` 相加前执行。

## Statements

## 语句

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 .
```

语句控制执行。

```
语句 =
        声明 | 标签语句 | 简单语句 |
        Go语句 | Return语句 | Break语句 | Continue语句 | Goto语句 |
        Fallthrough语句 | 块 | If语句 | Switch语句 | Select语句 | For语句 |
        Defer语句 .

简单语句 = 空语句 | 表达式语句 | 发送语句 | 递增递减语句 | 赋值 | 短变量声明 .
```

## Empty statements

## 空语句

The empty statement does nothing.

```
EmptyStmt = .
```

空语句不执行任何操作。

```
空语句 = .
```

## Labeled statements

## 标签语句

A labeled statement may be the target of a `goto`, `break` or `continue` statement.

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

标签语句可作为 `goto`、`break` 或 `continue` 语句的目标

```
标签语句 = 标签 ":" 语句 .
标签      = 标识符 .
```

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

## Expression statements

## 表达式语句

With the exception of specific built-in functions, function and method calls and receive operations can appear in statement context. Such statements may be parenthesized.

```
ExpressionStmt = Expression .
```

除特殊的内建函数外，函数与方法的调用及 接收操作均可出现在语句上下文中。这样的语句可能需要加小括号。

```
表达式语句 = 表达式 .
```

The following built-in functions are not permitted in statement context:

以下内建函数不允许出现在语句上下文中：

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

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

```
h(x+y)
f.Close()
<-ch
(<-ch)
len("foo")  // 若len为内建函数即为非法
```

## Send statements

## 发送语句

A send statement sends a value on a channel. The channel expression must be of channel type, the channel direction must permit send operations, and the type of the value to be sent must be assignable to the channel's element type.

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

发送语句在信道上发送值。信道表达式必须为信道类型， 信道的方向必须允许发送操作，且被发送的值的类型必须可赋予该信道的元素类型。

```
发送语句 = 信道 "<-" 表达式 .
信道     = 表达式 .
```

Both the channel and the value expression are evaluated before communication begins. Communication blocks until the send can proceed. A send on an unbuffered channel can proceed if a receiver is ready. A send on a buffered channel can proceed if there is room in the buffer. A send on a closed channel proceeds by causing a run-time panic. A send on a `nil` channel blocks forever.

信道与值表达式均在通信开始前求值。通信会阻塞，直到发送可继续进行。 若接收者已就绪，在无缓存信道上发送可继续进行。 若缓存中有空间，在有缓存信道上发送可继续进行。 在已关闭信道上进行发送会引发一个运行时恐慌。 在 `nil` 信道上进行发送将永远阻塞。

```
ch <- 3
```

## IncDec statements

## 递增递减语句

The "++" and "--" statements increment or decrement their operands by the untyped constant `1`. As with an assignment, the operand must be addressable or a map index expression.

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

"++" 与 "--" 语句会以无类型化常量 `1` 来递增或递减它们的操作数。 就赋值来说，操作数必须为可寻址的，或为映射的下标表达式。

```
递增递减语句 = 表达式 ( "++" | "--" ) .
```

The following assignment statements are semantically equivalent:

```
IncDec statement    Assignment
x++                 x += 1
x--                 x -= 1
```

以下赋值语句在语义上等价：

```
递增递减语句         赋值
x++                 x += 1
x--                 x -= 1
```

## Assignments

## 赋值

```
Assignment = ExpressionList assign_op ExpressionList .

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

```
赋值 = 表达式列表 赋值操作符 表达式列表 .
```

```
赋值操作符 = [ 加法操作符 | 乘法操作符 ] "=" .
```

Each left-hand side operand must be addressable, a map index expression, or the blank identifier. Operands may be parenthesized.

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

每个左操作数必须为可寻址的、映射下标表达式或空白标识符。 操作数可加小括号。

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

An *assignment operation* x *op*= y where *op* is a binary arithmetic operation is equivalent to x = x *op* y but evaluates x only once. The *op*= construct is a single token. In assignment operations, both the left- and right-hand expression lists must contain exactly one single-valued expression.

*赋值操作* x *op*= y，其中 *op* 为一个二元算术操作符，它等价于 x = x *op* y，但只对 x 求值一次。 *op*= 为单个标记。在赋值操作中，左、右表达式列表必须均刚好包含一个单值表达式。

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

A tuple assignment assigns the individual elements of a multi-valued operation to a list of variables. There are two forms. In the first, the right hand operand is a single multi-valued expression such as a function evaluation or channel or map operation or a type assertion. The number of operands on the left hand side must match the number of values. For instance, if f is a function returning two values,

```
x, y = f()
```

assigns the first value to x and the second to y. The blank identifier provides a way to ignore values returned by a multi-valued expression:

```
x, _ = f()  // ignore second value returned by f()
```

元组赋值将多值操作的个体元素赋予变量列表。它有两种形式。首先，右操作数为单个多值表达式，比如一个函数求值、 信道、映射操作或一个类型断言。 左操作数的数量必须与值的数量相匹配。例如，若 f 为返回两个值的函数，则

```
x, y = f()
```

会将第一个值赋予 x，而第二个值则会赋予 y。 空白标识符提供一种方式来忽略由多值表达式返回的值:

```
x, _ = f()  // 忽略由f()返回的第二值
```

In the second form, the number of operands on the left must equal the number of expressions on the right, each of which must be single-valued, and the *n*th expression on the right is assigned to the *n*th operand on the left.

在第二种形式中，左操作数的数量必须与右操作数的数量相等，其中的每一个必须为单值， 且右边第 *n* 个表达式会被赋予左边第 *n* 个表达式。此种赋值分两个阶段进行。

The assignment proceeds in two phases. First, the operands of index expressions and pointer indirections (including implicit pointer indirections in selectors) on the left and the expressions on the right are all evaluated in the usual order. Second, the assignments are carried out in left-to-right order.

```
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

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}
```

首先，左边的下标表达式与指针间接寻址的操作数 （包括选择者中隐式的指针间接寻址）和右边的表达式都会按通常顺序求值。 其次，赋值会按照从左到右的顺序进行。

```
a, b = b, a  // 交换a和b

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

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

x[0], x[0] = 1, 2  // 置 x[0] = 1，然后置 x[0] = 2（因此最后x[0] == 2）

x[1], x[3] = 4, 5  // 置 x[1] = 4，然后恐慌置 x[3] = 5

type Point struct { x, y int }
var p *Point
x[2], p.x = 6, 7  // 置 x[2] = 6，然后恐慌置 p.x = 7

i = 2
x = []int{3, 5, 7}
for i, x[i] = range x {  // 置 i, x[2] = 0, x[0]
        break
}
// 该循环结束之后，i == 0 且 x == []int{3, 5, 3}
```

In assignments, each value must be assignable to the type of the operand to which it is assigned. If an untyped constant is assigned to a variable of interface type, the constant is converted to type `bool`, `rune`, `int`, `float64`, `complex128` or `string` respectively, depending on whether the value is a boolean, rune, integer, floating-point, complex, or string constant.

在赋值中，每个值必须可赋予已赋值操作数的类型。若无类型化常量 已被赋予接口类型的变量，则该常量即被转换为类型 `bool`、`rune`、`int`、`float64`、`complex128` 或 `string` 之一，这取决于该值是否为布尔、符文、整数、浮点数、复数或字符串常量。

## If statements

## If语句

"If" statements specify the conditional execution of two branches according to the value of a boolean expression. If the expression evaluates to true, the "if" branch is executed, otherwise, if present, the "else" branch is executed.

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

"If"语句根据一个布尔表达式的值指定两个分支的条件来执行。 若该表达式求值为true，则执行"if"分支，否则执行"else"分支。

```
If语句 = "if" [ 简单语句 ";" ] 表达式 块 [ "else" ( If语句 | 块 ) ] .
```

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

The expression may be preceded by a simple statement, which executes before the expression is evaluated.

简单语句可能先于表达式，它将在表达式求值前执行。

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

```
	} else {
		return y
	}
```

## Switch statements

## Switch语句

"Switch" statements provide multi-way execution. An expression or type specifier is compared to the "cases" inside the "switch" to determine which branch to execute.

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

"Switch"语句提供多路执行。表达式或类型说明符与"switch"中的"cases"相比较从而决定执行哪一分支。

```
Switch语句 = 表达式选择语句 | 类型选择语句 .
```

There are two forms: expression switches and type switches. In an expression switch, the cases contain expressions that are compared against the value of the switch expression. In a type switch, the cases contain types that are compared against the type of a specially annotated switch expression.

它有两种形式：表达式选择与类型选择。在表达式选择中，case包含的表达式针对switch表达式的值进行比较， 在类型选择中，case包含的类型针对特别注明的switch表达式的类型进行比较。

### Expression switches

### 表达式选择

In an expression switch, the switch expression is evaluated and the case expressions, which need not be constants, are evaluated left-to-right and top-to-bottom; the first one that equals the switch expression triggers execution of the statements of the associated case; the other cases are skipped. If no case matches and there is a "default" case, its statements are executed. There can be at most one default case and it may appear anywhere in the "switch" statement. A missing switch expression is equivalent to the expression `true`.

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

在表达式选择中，switch表达式会被求值，而case表达式无需为常量，它按从上到下，从左到右的顺序求值； 第一个等于switch表达式的case表达式将引发相应情况的语句的执行；其它的情况将被跳过。 若没有情况匹配且有"default"情况，则该语句将被执行。 最多只能有一个默认情况且它可以出现在"switch"语句的任何地方。 缺失的switch表达式等价于表达式 `true`。

```
表达时选择语句 = "switch" [ 简单语句 ";" ] [ 表达式 ] "{" { 表达式情况子句 } "}" .
表达式情况子句 = 表达式选择情况 ":" { 语句 ";" } .
表达式选择情况 = "case" 表达式列表 | "default" .
```

In a case or default clause, the last statement only may be a "fallthrough" statement (§Fallthrough statement) to indicate that control should flow from the end of this clause to the first statement of the next clause. Otherwise control flows to the end of the "switch" statement.

在case或default子句中，最后一个语句可能只为"fallthrough"语句（§Fallthrough语句）， 它表明该控制流应从该子句的结尾转至下一个子句的第一个语句。 否则，控制流转至该"switch"语句的结尾。

The expression may be preceded by a simple statement, which executes before the expression is evaluated.

```
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()
}
```

简单语句可能先于表达式，它将在表达式求值前执行。

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

switch x := f(); {  // 缺失的switch表达式意为"true"
case x < 0: return -x
default: return x
}

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

### Type switches

### 类型选择

A type switch compares types rather than values. It is otherwise similar to an expression switch. It is marked by a special switch expression that has the form of a type assertion using the reserved word `type` rather than an actual type:

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

Cases then match actual types `T` against the dynamic type of the expression `x`. As with type assertions, `x` must be of interface type, and each non-interface type `T` listed in a case must implement the type of `x`.

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

类型选择比较类型而非值。它与表达式选择并不相似。它被一个特殊的switch表达式标记， 该表达式为使用保留字 `type` 而非实际类型的类型断言的形式：

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

此时的case针对表达式 `x` 的动态类型匹配实际的类型 `T`。 就像类型断言一样，`x` 必须为接口类型， 而每一个在case中列出的非接口类型 `T` 必须实现了 `x` 的类型。

```
类型选择语句 = "switch" [ 简单语句 ";" ] 类型选择监视 "{" { 类型情况子句 } "}" .
类型选择监视 = [ 标识符 ":=" ] 主表达式 "." "(" "type" ")" .
类型情况子句 = 类型选择情况 ":" { 语句 ";" } .
类型选择情况 = "case" 类型列表 | "default" .
类型列表     = 类型 { "," 类型 } .
```

The TypeSwitchGuard may include a short variable declaration. When that form is used, the variable is declared at the beginning of the implicit block in each clause. In clauses with a case listing exactly one type, the variable has that type; otherwise, the variable has the type of the expression in the TypeSwitchGuard.

类型选择监视可包含一个短变量声明。 当使用此形式时，变量会在每个子句的隐式块的起始处声明。 在case列表刚好只有一个类型的子句中，该变量即拥有此类型；否则，该变量拥有在类型选择监视中表达式的类型。

The type in a case may be `nil` (§Predeclared identifiers); that case is used when the expression in the TypeSwitchGuard is a `nil` interface value.

case中的类型可为 `nil`（§预声明标识符）， 这种情况在类型选择监视中的表达式为 `nil` 接口值时使用。

Given an expression `x` of type `interface{}`, the following type switch:

```
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{})
}
```

could be rewritten:

```
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")
	}
}
```

给定类型为 interface{} 的表达式 x，以下类型选择：

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

可被重写为：

```
v := x  // x 只被求值一次
if v == nil {
	i := v                                 // i 的类型为 x 的类型（interface{}）
	printString("x is nil")
} else if i, isInt := v.(int); isInt {
	printInt(i)                            // i 的类型为 int
} else if i, isFloat64 := v.(float64); isFloat64 {
	printFloat64(i)                        // i 的类型为 float64
} else if i, isFunc := v.(func(int) float64); isFunc {
	printFunction(i)                       // i 的类型为 func(int) float64
} else {
	_, isBool := v.(bool)
	_, isString := v.(string)
	if isBool || isString {
		i := v                         // i 的类型为 x 的类型（interface{}）
		printString("type is bool or string")
	} else {
		i := v                         // i 的类型为 x 的类型（interface{}）
		printString("don't know the type")
	}
}
```

The type switch guard may be preceded by a simple statement, which executes before the guard is evaluated.

简单语句可能先于类型选择监视，它将在类型选择监视求值前执行。

The "fallthrough" statement is not permitted in a type switch.

"fallthrough"语句在类型选择中不被允许。

## For statements

## For语句

A "for" statement specifies repeated execution of a block. The iteration is controlled by a condition, a "for" clause, or a "range" clause.

```
ForStmt = "for" [ Condition | ForClause | RangeClause ] Block .
Condition = Expression .
```

"for"语句指定块的重复执行。迭代通过条件、"for"子句或"range"子句控制。

```
For语句 = "for" [ 条件 | For子句 | Range子句 ] 块 .
条件 = 表达式 .
```

In its simplest form, a "for" statement specifies the repeated execution of a block as long as a boolean condition evaluates to true. The condition is evaluated before each iteration. If the condition is absent, it is equivalent to `true`.

在最简单的形式中，只要布尔条件求值为真，"for"语句指定的块就重复执行。 条件会在每次迭代前求值。若缺少条件，则它等价于 `true`。

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

A "for" statement with a ForClause is also controlled by its condition, but additionally it may specify an *init* and a *post* statement, such as an assignment, an increment or decrement statement. The init statement may be a short variable declaration, but the post statement must not.

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

带For子句的"for"语句也通过其条件控制，此外，它也可指定一个*初始化*或*步进*语句， 例如一个赋值、一个递增或递减语句。初始化语句可为一个短变量声明， 而步进语句则不能。

```
For子句     = [ 初始化语句 ] ";" [ 条件 ] ";" [ 步进语句 ] .
初始化语句 = 简单语句 .
步进语句   = 简单语句 .
```

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

If non-empty, the init statement is executed once before evaluating the condition for the first iteration; the post statement is executed after each execution of the block (and only if the block was executed). Any element of the ForClause may be empty but the semicolons are required unless there is only a condition. If the condition is absent, it is equivalent to `true`.

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

若初始化语句非空，则只在第一次迭代的条件求值前执行一次。 步进语句会在块的每一次执行（且仅当块被执行）后执行。 任何For子句的元素都可为空，但除非只有一个条件，否则分号是必须的。 若缺少条件，则它等价于 `true`。

```
for cond { S() }    等价于    for ; cond ; { S() }
for      { S() }    等价于    for true     { S() }
```

A "for" statement with a "range" clause iterates through all entries of an array, slice, string or map, or values received on a channel. For each entry it assigns *iteration values* to corresponding *iteration variables* and then executes the block.

```
RangeClause = ( ExpressionList "=" | IdentifierList ":=" ) "range" Expression .
```

带"range"子句的"for"语句通过遍历数组、切片、字符串或映射的所有项，以及从信道上接收的值来迭代。 对于每一项，它将*迭代值*赋予其相应的*迭代变量*，然后执行该块。

```
Range子句 = ( 表达式列表 "=" | 标识符列表 ":=" ) "range" 表达式 .
```

The expression on the right in the "range" clause is called the *range expression*, which may be an array, pointer to an array, slice, string, map, or channel permitting receive operations. As with an assignment, the operands on the left must be addressable or map index expressions; they denote the iteration variables. If the range expression is a channel, only one iteration variable is permitted, otherwise there may be one or two. In the latter case, if the second iteration variable is the blank identifier, the range clause is equivalent to the same clause with only the first variable present.

"range"子句右边的表达式称为*range表达式*，它可以是一个允许接受操作 的数组、数组指针、切片、字符串、映射或信道。就赋值来说，左边的操作数必须为 可寻址的或映射下标表达式；它们表示迭代变量。若range表达式为信道， 则只允有一个迭代变量，否则可为一个或两个。在后一种情况下，若第二个迭代变量为 空白标识符，则range子句等价于只存在第一个变量的相同子句。

The range expression is evaluated once before beginning the loop except if the expression is an array, in which case, depending on the expression, it might not be evaluated (see below). Function calls on the left are evaluated once per iteration. For each iteration, iteration values are produced as follows:

```
Range expression                          1st value          2nd value (if 2nd variable is present)

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
```

1. For an array, pointer to array, or slice value `a`, the index iteration values are produced in increasing order, starting at element index 0. As a special case, if only the first iteration variable is present, the range loop produces iteration values from 0 up to `len(a)` and does not index into the array or slice itself. For a `nil` slice, the number of iterations is 0.
2. For a string value, the "range" clause iterates over the Unicode code points in the string starting at byte index 0. On successive iterations, the index value will be the index of the first byte of successive UTF-8-encoded code points in the string, and the second value, of type `rune`, will be the value of the corresponding code point. If the iteration encounters an invalid UTF-8 sequence, the second value will be `0xFFFD`, the Unicode replacement character, and the next iteration will advance a single byte in the string.
3. The iteration order over maps is not specified and is not guaranteed to be the same from one iteration to the next. If map entries that have not yet been reached are removed during iteration, the corresponding iteration values will not be produced. If map entries are created during iteration, that entry may be produced during the iteration or may be skipped. The choice may vary for each entry created and from one iteration to the next. If the map is `nil`, the number of iterations is 0.
4. For channels, the iteration values produced are the successive values sent on the channel until the channel is closed. If the channel is `nil`, the range expression blocks forever.

除非range表达式为数组，否则它只会在开始循环前求值一次。在此情况下，它是否会被求值取决于该表达式（见下）。 左边的函数调用会在每次迭代时求值一次。对于每一次迭代，迭代值按照以下方式产生：

```
Range表达式                      第一个值            第二个值（若第二个变量存在）

数组或切片  a  [n]E、*[n]E 或 []E    下标  i  int       a[i]   E
字符串      s  string type           下标  i  int       见下   rune
映射        m  map[K]V               键    k  K         m[k]   V
信道        c  chan E, <-chan E    元素  e  E
```

1. 对于数组、数组指针或切片值 `a`，下标迭代值按照递增顺序产生，从元素下标0开始。 作为一种特殊情况，若只存在第一个迭代变量，则range循环提供从0到 `len(a)` 的迭代变量而非索引该数组或切片自身。对于 `nil` 切片，迭代次数为0。
2. 对于字符串值，"range"子句从字节下标0开始，遍历该字符串中的Unicode码点。在连续迭代中， 其下标值为该字符串中连续UTF-8编码码点第一个字节的下标。而类型为 `rune` 的第二个值为则其相应码点的值。若该迭代遇到无效的UTF-8序列，则第二个值将为Unicode占位字符 `0xFFD`，且下一次迭代将推进至此字符串中的单个字节。
3. 映射的遍历顺序并不确定且从某一次迭代到下一次并不保证相同。若在迭代过程中移除的映射项尚未受到影响， 则相应的迭代值不会产生。若在迭代过程中创建映射项，则该项可能会在迭代中产生或被跳过。 这种选择可能会改变已经创建的每一个项，并从一次迭代进入到下一次迭代中。 若该映射为 `nil`，则迭代的次数为0.
4. 对于信道，其迭代值产生为在该信道上发送的连续值，直到该信道被关闭。若该信道为 `nil`，则range表达式将永远阻塞。

The iteration values are assigned to the respective iteration variables as in an assignment statement.

迭代值在赋值语句中将分别赋予其各自的迭代变量。

The iteration variables may be declared by the "range" clause using a form of short variable declaration (`:=`). In this case their types are set to the types of the respective iteration values and their scope ends at the end of the "for" statement; they are re-used in each iteration. If the iteration variables are declared outside the "for" statement, after execution their values will be those of the last iteration.

```
var testdata *struct {
        a *[7]int
}
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
m := map[string]int{"mon":0, "tue":1, "wed":2, "thu":3, "fri":4, "sat":5, "sun":6}
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 {}  // value type of m is assignable to val
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)
}
```

迭代变量可通过"range"子句使用短变量声明形式（:=）来声明。 在这种情况下，它们的类型将置为其各自的迭代值，且它们的作用域 终止于"for"语句的结尾，它们将在每次迭代时被重用。若迭代变量在"for"语句之外声明，则在每次执行后， 它们的值为最后一次迭代的值。

```
var testdata *struct {
        a *[7]int
}
for i, _ := range testdata.a {
        // testdata.a永不会被求值，len(testdata.a)为常量
        // i从0延伸到6
        f(i)
}

var a [10]string
m := map[string]int{"mon":0, "tue":1, "wed":2, "thu":3, "fri":4, "sat":5, "sun":6}
for i, s := range a {
        // i的类型为int
        // s的类型为string
        // s == a[i]
        g(i, s)
}

var key string
var val interface {}  // m值的类型可赋予val
for key, val = range m {
        h(key, val)
}
// key == 在迭代中遇到的最后一个映射键
// val == map[key]

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

## Go statements

## Go语句

A "go" statement starts the execution of a function call as an independent concurrent thread of control, or *goroutine*, within the same address space.

```
GoStmt = "go" Expression .
```

"go"语句将函数调用的执行作为控制独立的并发线程或相同地址空间中的*Go程*来启动。

```
Go语句 = "go" 表达式 .
```

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

表达式必须为一个函数或方法调用，它不能被括号括住。内建函数调用被限制为表达式语句。

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

在调用Go程中，函数值与形参按照惯例求值， 但不像一般的调用，程序的执行并不等待已被调用的函数完成。取而代之，该函数在一个新的Go程中独立执行。 当该函数终止后，其Go程也将终止。若该函数拥有任何返回值，它们将在该函数完成后被丢弃。

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

## Select statements

## Select语句

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

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

"select"语句选择可能发生通信的集。它看起来与"switch"语句类似，但其case为所有涉及到通信的操作。

```
Select语句 = "select" "{" { 通信子句 } "}" .
通信子句   = 通信情况 ":" { 语句 ";" } .
通信情况   = "case" ( 发送语句 | 接收语句 ) | "default" .
接收语句   = [ 表达式列表 [ "," 标识符列表 ] ( "=" | ":=" ) ] 接收表达式 .
接收表达式 = 表达式 .
```

RecvExpr must be a receive operation. For all the cases in the "select" statement, the channel expressions are evaluated in top-to-bottom order, along with any expressions that appear on the right hand side of send statements. A channel may be `nil`, which is equivalent to that case not being present in the select statement except, if a send, its expression is still evaluated. If any of the resulting operations can proceed, one of those is chosen and the corresponding communication and statements are evaluated. Otherwise, if there is a default case, that executes; if there is no default case, the statement blocks until one of the communications can complete. There can be at most one default case and it may appear anywhere in the "select" statement. If there are no cases with non-`nil` channels, the statement blocks forever. Even if the statement blocks, the channel and send expressions are evaluated only once, upon entering the select statement.

接收表达式必须为接收操作。对于"select"语句中的所有case， 信道表达式连同任何出现在发送语句右侧的表达式均按照从上到下的顺序求值。信道可能为 `nil`， 它等价于select语句中不存在该case，若是一个发送操作，其表达式仍将被求值。 若有任何由此产生的操作可以进行，则选择其中之一，且相应的通信和语句将被求值。否则，若有默认case， 则执行；若没有默认case，则该语句将阻塞直到其中一个通信完成。默认case最多只能出现一次， 且它可以出现在 "select" 语句中的任何地方。若没有非 `nil` 信道的case， 该语句将永远阻塞。即使该语句阻塞，当进入select语句时，信道与发送表达式也只会被求值一次。

Since all the channels and send expressions are evaluated, any side effects in that evaluation will occur for all the communications in the "select" statement.

在所有信道与发送表达式求值后，对于所有"select"语句中的通信，任何在此求值中的副作用都将发生。

If multiple cases can proceed, a uniform pseudo-random choice is made to decide which single communication will execute.

若有多个case均可进行，则会构造一个均匀的伪随机数选择，以此决定哪一个通信将会执行。

The receive case may declare one or two new variables using a short variable declaration.

```
var c, c1, c2, c3 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")
        }
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
```

接收case可使用短变量声明来声明一个或两个新的变量。

```
var c, c1, c2, c3 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):  // 等价于 i3, ok := <-c3
        if ok {
                print("received ", i3, " from c3\n")
        } else {
                print("c3 is closed\n")
        }
default:
        print("no communication\n")
}

for {  // 向c发送位的随机序列
        select {
        case c <- 0:  // 注意：没有语句，没有fallthrough，也没有case折叠
        case c <- 1:
        }
}

select {}  // 永远阻塞
```

## Return statements

## Return语句

A "return" statement in a function F terminates the execution of F, and optionally provides one or more result values. Any functions deferred by F are executed before F returns to its caller.

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

函数 F 中的“return”语句终止 F 的执行，并可选地提供一个或多个返回值。 任何被 F 推迟的函数会在 F 返回给其调用者前执行。

```
Return语句 = "return" [ 表达式列表 ] .
```

In a function without a result type, a "return" statement must not specify any result values.

在没有返回类型的函数中，"return"语句不能指定任何返回值。

```
func noResult() {
        return
}
```

There are three ways to return values from a function with a result type:

1. The return value or values may be explicitly listed in the "return" statement. Each expression must be single-valued and assignable to the corresponding element of the function's result type.

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

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

2. The expression list in the "return" statement may be a single call to a multi-valued function. The effect is as if each value returned from that function were assigned to a temporary variable with the type of the respective value, followed by a "return" statement listing these variables, at which point the rules of the previous case apply.

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

3. The expression list may be empty if the function's result type specifies names for its result parameters (§Function types). The result parameters act as ordinary local variables and the function may assign values to them as necessary. The "return" statement returns the values of these variables.

```
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
}
```

从具有返回类型的函数返回值的三种方式:

1. 返回值可在"return"语句中显式地列出。每个表达式必须为单值， 且可赋予相应的函数返回类型的元素。

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

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

2. "return"语句中的表达式列表可为多值函数的单个调用。 其效果相当于将该函数返回的每一个值赋予同类型的临时变量， 并在"return"语句后面列出这些变量，在这点上与前面的情况规则相同。

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

3. 若函数的返回类型为其返回形参指定了名字（§函数类型）， 则表达式列表可为空。返回形参的行为如同一般的局部变量，且必要时该函数可向他们赋值。 "return"语句返回这些变量的值。

```
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
}
```

Regardless of how they are declared, all the result values are initialized to the zero values for their type (§The zero value) upon entry to the function. A "return" statement that specifies results sets the result parameters before any deferred functions are executed.

无论它们如何声明，在进入该函数时，所有的返回值都会被初始化为该类型的零值（§零值）。 指定了结果的“return”语句会在任何被推迟的函数执行前设置结果形参。

## Break statements

### Break语句

A "break" statement terminates execution of the innermost "for", "switch" or "select" statement.

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

"break"语句终止最内层的"for"、"switch"或"select"语句的执行。

```
Break语句 = "break" [ 标签 ] .
```

If there is a label, it must be that of an enclosing "for", "switch" or "select" statement, and that is the one whose execution terminates (§For statements, §Switch statements, §Select statements).

若存在标签，则它必须为闭合的"for"、"switch"或"select"语句，而此执行就会终止。（§For语句, §Switch语句, §Select语句）。

```
L:
	for i < n {
		switch i {
		case 5:
			break L
		}
	}
```

### Continue statements

### Continue语句

A "continue" statement begins the next iteration of the innermost "for" loop at its post statement (§For statements).

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

"continue"语句在最内层"for"循环的步进语句处开始下一次迭代（§For语句）。

```
Continue语句 = "continue" [ 标签 ] .
```

If there is a label, it must be that of an enclosing "for" statement, and that is the one whose execution advances (§For statements).

若存在标签，则它必须为闭合的"for"语句，而此执行就会前进。（§For语句）。

### Goto statements

### Goto语句

A "goto" statement transfers control to the statement with the corresponding label.

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

"goto"语句用于将控制转移到与其标签相应的语句。

```
Goto语句 = "goto" 标签 .
```

```
goto Error
```

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. For instance, this example:

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

执行"goto"不能在跳转点处产生任何还未在作用域中的变量来使其进入作用域。 比如，例子：

```
	goto L  // 这样不好
	v := 3
```

```
L:
```

is erroneous because the jump to label L skips the creation of v.

是错误的，因为跳转至标签 L 处将跳过 v 的创建。

A "goto" statement outside a block cannot jump to a label inside that block. For instance, this example:

在块外的"goto"语句不能跳转至该块中的标签。 比如，例子：

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

is erroneous because the label L1 is inside the "for" statement's block but the goto is not.

是错误的，因为标签 L1 在"for"语句的块中而 goto 则不在。

### Fallthrough statements

### Fallthrough语句

A "fallthrough" statement transfers control to the first statement of the next case clause in a expression "switch" statement (§Expression switches). It may be used only as the final non-empty statement in a case or default clause in an expression "switch" statement.

```
FallthroughStmt = "fallthrough" .
```

"fallthrough"语句将控制转移到表达式"switch"语句（§表达式选择） 中下一个case子句的第一个语句。它可能只被用作表达式"switch"语句中case或default子句里最后的非空语句。

```
Fallthrough语句 = "fallthrough" .
```

### Defer statements

### Defer语句

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

```
DeferStmt = "defer" Expression .
```

"defer"语句调用的函数将被推迟到其外围函数返回时执行，不论是因为该外围函数执行了 return 语句，到达了其函数体的末尾， 还是因为其对应的Go程进入了恐慌过程。

```
Defer语句 = "defer" 表达式 .
```

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

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

For instance, if the deferred function is a function literal and the surrounding function has named result parameters that are in scope within the literal, the deferred function may access and modify the result parameters before they are returned. If the deferred function has any return values, they are discarded when the function completes. (See also the section on handling panics.)

```
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 1
func f() (result int) {
        defer func() {
                result++
        }()
        return 0
}
```

该表达必须为一个函数或方法调用，它不能被括号括住。内建函数调用被限制为表达式语句。

"defer"语句每执行一次，它所调用的函数值与形参就会 像平时一样求值并重新保存，但实际的函数提并不会被执行。 取而代之的是，在外围的函数返回前，被推迟的函数会按照它们被推迟的相反顺序立即执行。

例如，若被推迟的函数为函数字面，而其外围函数在其作用域中的函数字面内拥有 已命名结果形参，则被推迟的函数可在该结果形参被返回前访问并更改。 若被推迟函数拥有任何返回值，则它们会在该函数完成时丢弃。 （另请参阅恐慌处理）一节。

```
lock(l)
defer unlock(l)  // 解锁在外围函数返回前发生

// 在外围函数返回前打印 3 2 1 0
for i := 0; i <= 3; i++ {
        defer fmt.Print(i)
}

// f 返回 1
func f() (result int) {
        defer func() {
                result++
        }()
        return 0
}
```

## Built-in functions

## 内建函数

Built-in functions are predeclared. They are called like any other function but some of them accept a type instead of an expression as the first argument.

内建函数是预声明的。它们可以像其他任何函数一样被调用， 但其中某些函数则接受类型而非表达式来作为第一个实参。

The built-in functions do not have standard Go types, so they can only appear in call expressions; they cannot be used as function values.

```
BuiltinCall = identifier "(" [ BuiltinArgs [ "," ] ] ")" .
BuiltinArgs = Type [ "," ArgumentList ] | ArgumentList .
```

内建函数并没有标准的Go类型，因此它们只能出现在调用表达式中，而不能作为函数值被使用。

```
内建调用 = 标识符 "(" [ 内建实参 [ "," ] ] ")" .
内建实参 = 类型 [ "," 实参列表 ] | 实参列表 .
```

## Close

## 关闭

For a channel `c`, the built-in function `close(c)` records that no more values will be sent on the channel. It is an error if `c` is a receive-only channel. Sending to or closing a closed channel causes a run-time panic. Closing the nil channel also causes a run-time panic. After calling `close`, and after any previously sent values have been received, receive operations will return the zero value for the channel's type without blocking. The multi-valued receive operation returns a received value along with an indication of whether the channel is closed.

对于一个信道 `c`，内建函数 `close(c)` 标明不再有值会在该信道上发送。 若 `c` 为只接收信道，则会产生一个错误。向已关闭的信道发送信息或再次关闭它将引发 运行时恐慌。关闭 `nil` 信道也会引发运行时恐慌。 在调用 `close` 之后，任何以前发送的值都会被接收，接收操作将无阻塞地返回该信道类型的零值。 多值 接收操作 会随着一个该信道是否关闭的指示返回一个已接收的值。

## Length and capacity

### 长度与容量

The built-in functions len and cap take arguments of various types and return a result of type int. The implementation guarantees that the result always fits into an 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
```

内建函数 len 与 cap 接受各种类型的实参并返回 int 类型的结果。 该实现保证其结果总符合 int 类型。

```
调用      实参类型         结果

len(s)    string type      字符串的字节长度。
          [n]T, *[n]T      数组长度（== n）
          []T              切片长度
          map[K]T          映射长度（已定义键的数量）
          chan T           信道缓存中元素队列的长度

cap(s)    [n]T, *[n]T      数组长度（== n）
          []T              切片容量
          chan T           信道缓存容量
```

The capacity of a slice is the number of elements for which there is space allocated in the underlying array. At any time the following relationship holds:

切片的容量为其基本数组中已分配的空间元素的数量。以下关系在任何时候都成立：

```
0 <= len(s) <= cap(s)
```

The length of a nil slice, map or channel is 0. The capacity of a nil slice and channel is 0.

nil 切片、映射或信道的长度为0。nil 切片和信道的容量为0。

The expression len(s) is constant if s is a string constant. The expressions len(s) and cap(s) are constants if the type of s is an array or pointer to an array and the expression s does not contain channel receives or function calls; in this case s is not evaluated. Otherwise, invocations of len and cap are not constant and s is evaluated.

若 s 为字符串常量，则表达式 len(s) 即为 常量。 若 s 的类型为数组或数组指针，且表达式 s 不包含信道接收 或函数调用，则表达式 len(s) 与 cap(s) 即为常量，在这种用情况下， s 不会被求值。否则，len 与 cap 的调用不为常量，且 s 会被求值。

## Allocation

## 分配

The built-in function new takes a type T and returns a value of type *T. The memory is initialized as described in the section on initial values (§The zero value).

内建函数 new 接受类型 T 并返回类型为 *T 的值。 其内存根据初始值（§零值）片段的描述来初始化。

```
new(T)
```

For instance

例如

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

dynamically allocates memory for a variable of type S, initializes it (a=0, b=0.0), and returns a value of type *S containing the address of the memory.

将为类型为 S 的变量动态分配内存、初始化（a=0，b=0.0）并返回类型为 *S 的包含内存地址的值。

## Making slices, maps and channels

## 创建切片、映射与信道

Slices, maps and channels are reference types that do not require the extra indirection of an allocation with new. The built-in function make takes a type T, which must be a slice, map or channel type, optionally followed by a type-specific list of expressions. It returns a value of type T (not *T). The memory is initialized as described in the section on initial values (§The zero value).

```
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 n elements

make(T)          channel    synchronous channel of type T
make(T, n)       channel    asynchronous channel of type T, buffer size n
```

切片、映射与信道为无需使用 new 来间接额外分配的引用类型。内建函数 make 接受的类型 T 必须为切片、映射或信道类型，可选地跟着一个特殊类型的表达式列表。它返回类型为 T（而非 *T）的值。其内存根据初始值（§零值）片段的描述来初始化。

```
调用              类型 T     结果

make(T, n)       slice      类型为T，长度为n，容量为n的切片
make(T, n, m)    slice      类型为T，长度为n，容量为m的切片

make(T)          map        类型为T的映射
make(T, n)       map        类型为T，初始空间为n个元素的映射

make(T)          channel    类型为T的同步信道
make(T, n)       channel    类型为T，缓存大小为n的异步信道
```

The size arguments n and m must be integer values. A constant size argument must be non-negative and representable by a value of type int. If both n and m are provided and are constant, then n must be no larger than m. If n is negative or larger than m at run time, a run-time panic occurs.

```
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 100 elements
```

用于指定大小的实参 n 与 m 必须为整数值。 常量大小的实参必须为非负值，且可表示为 int 类型的值。 若 n 和 m 已给定且为常量，则 n 必不大于 m。若 n 在运行时为负值或大于 m， 就会引发运行时恐慌。

```
s := make([]int, 10, 100)       // len(s) == 10, cap(s) == 100 的切片
s := make([]int, 1e3)           // len(s) == cap(s) == 1000 的切片
s := make([]int, 1<<63)   // 非法: len(s) 不能表示为 int 类型的值
s := make([]int, 10, 0)         // 非法: len(s) > cap(s)
c := make(chan int, 10)         // 缓存大小为10的信道
m := make(map[string]int, 100)  // 初始空间为100个元素的映射
```

## Appending to and copying slices

## 追加与复制切片

The built-in functions append and copy assist in common slice operations. For both functions, the result is independent of whether the memory referenced by the arguments overlaps.

内建函数 append 与 copy 协助一般的切片操作。对于这两个函数， 无论其内存引用是否与其实参重复，其结果都是独立的。

The variadic function append appends zero or more values x to s of type S, which must be a slice type, and returns the resulting slice, also of type S. The values x are passed to a parameter of type ...T where T is the element type of S and the respective parameter passing rules apply. As a special case, append also accepts a first argument assignable to type []byte with a second argument of string type followed by .... This form appends the bytes of the string.

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

变参函数 append 追加零个或更多值 x 至 类型为 S 的s，它必须为切片类型，且返回类型为 S 的结果切片， 值 x 被传至类型为 ...T 的形参，其中 T 为 S 的元素类型，且其各自的形参传递规则均适用。 作为一个特例，append 也接受第一个实参可赋予类型 []byte， 且第二个字符串类型的实参后跟 ...。此形式追加字符串类型的字节。

```
append(s S, x ...T) S  // T是类型为S的元素
```

If the capacity of s is not large enough to fit the additional values, append allocates a new, sufficiently large slice that fits both the existing slice elements and the additional values. Thus, the returned slice may refer to a different underlying array.

```
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' }
```

若 s 的容量不足够大以适应附加的值，append 会分配一个新的， 足够大的切片以适应现有切片元素与附加的值。因此，返回的切片可能涉及到不同的基本数组。

```
s0 := []int{0, 0}
s1 := append(s0, 2)                // 追加单个元素    s1 == []int{0, 0, 2}
s2 := append(s1, 3, 5, 7)          // 追加多个元素    s2 == []int{0, 0, 2, 3, 5, 7}
s3 := append(s2, s0...)            // 追加一个切片    s3 == []int{0, 0, 2, 3, 5, 7, 0, 0}
s4 := append(s3[3:6], s3[2:]...)   // 追加重复切片    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"...)            // 追加字符串常量  b == []byte{'b', 'a', 'r' }
```

The function copy copies slice elements from a source src to a destination dst and returns the number of elements copied. Both arguments must have identical element type T and must be assignable to a slice of type []T. The number of elements copied is the minimum of len(src) and len(dst). As a special case, copy also accepts a destination argument assignable to type []byte with a source argument of a string type. This form copies the bytes from the string into the byte slice.

函数 copy 将切片元素从来源 src 复制到目标 dst 并返回复制的元素数量。两个实参必须都拥有相同的元素类型 T，且必须都可赋予类型为 []T 的切片。 被复制的元素数量为 len(src) 与 len(dst) 中最小的那个。 作为一个特例，copy 也接受一个可赋予类型 []byte 的目标实参以及一个字符串类型的来源实参。此形式从该字符串中复制字节到该字节切片。

```
copy(dst, src []T) int
copy(dst []byte, src string) int
```

Examples:

例如：

```
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

## 映射元素的删除

The built-in function delete removes the element with key k from a map m. The type of k must be assignable to the key type of m.

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

内建函数 delete 从映射 m中移除键为 k 的元素。k的类型必须可赋予 m类型的键。

```
delete(m, k)  // 从映射m中移除元素m[k]
```

If the map `m` is `nil` or the element `m[k]` does not exist, `delete` is a no-op.

若映射 `m` 为 `nil` 或元素 `m[k]`不存在，则 `delete` 就是一个空操作。

## Manipulating complex numbers

## 复数操作

Three functions assemble and disassemble complex numbers. The built-in function `complex` constructs a complex value from a floating-point real and imaginary part, while `real` and `imag` extract the real and imaginary parts of a complex value.

三个函数用于组合并分解复数。内建函数 `complex` 从一个浮点数实部和虚部构造一个复数值。 而 `real` 和 `imag` 则提取一个复数值的实部和虚部。

```
complex(realPart, imaginaryPart floatT) complexT
real(complexT) floatT
imag(complexT) floatT
```

The type of the arguments and return value correspond. For `complex`, the two arguments must be of the same floating-point type and the return type is the complex type with the corresponding floating-point constituents: `complex64` for `float32`, `complex128` for `float64`. The `real` and `imag` functions together form the inverse, so for a complex value `z`, `z` `==` `complex(real(z), imag(z))`.

其实参与返回值的类型一致。对于 `complex`，两实参必须为相同的浮点类型且其返回类型为 浮点组成部分一致的复数类型：`complex64` 对应于 `float32`， `complex128` 对应于 `float64`。 `real` 与 `imag` 函数的形式则相反，因此对于一个复数值 `z`， `z` `==` `complex(real(z), imag(z))`。

If the operands of these functions are all constants, the return value is a constant.

若这些函数的操作数均为常量，则返回值亦为常量。

```
var a = complex(2, -2)             // complex128
var b = complex(1.0, -1.4)         // complex128
x := float32(math.Cos(math.Pi/2))  // float32
var c64 = complex(5, -x)           // complex64
var im = imag(b)                   // float64
var rl = real(c64)                 // float32
```

## Handling panics

## 恐慌处理

Two built-in functions, `panic` and `recover`, assist in reporting and handling run-time panics and program-defined error conditions.

内建函数 `panic` 和 `recover` 用于协助报告并处理运行时恐慌以及由程序定义的错误情况。

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

A `panic` call in a function `F` terminates the execution of `F`. Any functions deferred by `F` are executed before `F` returns to its caller. To the caller, the call of `F` then behaves itself like a call to `panic`, terminating its own execution and running deferred functions in the same manner. This continues until all functions in the goroutine have ceased execution, in reverse order. At that point, the program is terminated and the error condition is reported, including the value of the argument to `panic`. This termination sequence is called *panicking*.

函数 `F` 中的 `panic` 调用会终止 `F` 的执行。 任何被 `F` 推迟函数都会在 `F` 返回给其调用者前执行，且不执行该函数剩余的部分。对于其调用者，`F` 的调用行为自身如同对 `panic` 的调用，即终止其自身的执行， 并以相同的方式运行被推迟的函数。这会持续到该Go程中所有函数都按相反顺序停止执行之后。 到那时，该程序将被终止，而该错误情况会被报告，包括引发 `panic` 的实参值。 此终止序列称为*恐慌过程*。

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

The `recover` function allows a program to manage behavior of a panicking goroutine. Executing a `recover` call *inside* a deferred function (but not any function called by it) stops the panicking sequence by restoring normal execution, and retrieves the error value passed to the call of `panic`. If `recover` is called outside the deferred function it will not stop a panicking sequence. In this case, or when the goroutine is not panicking, or if the argument supplied to `panic` was `nil`, `recover` returns `nil`.

`recover` 函数允许程序管理恐慌过程Go程的行为。在*已推迟函数*（而不是任何被它调用的函数）中，执行 `recover` 调用会通过恢复正常

执行， 并取回传递至 `panic` 调用的错误值来停止恐慌过程序列。若 `recover` 在已推迟函数之外被调用，它将不会停止恐慌过程序列。在此情况下，或当Go程不在恐慌过程中时， 或提供给 `panic` 的实参为 `nil` 时，`recover` 就会返回 `nil`。

The `protect` function in the example below invokes the function argument `g` and protects callers from run-time panics raised by `g`.

```
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()
}
```

```
func protect(g func()) {
	defer func() {
		log.Println("done")  // 即使有恐慌Println也会正常执行
		if x := recover(); x != nil {
			log.Printf("run time panic: %v", x)
		}
	}()
	log.Println("start")
	g()
}
```

下面例子中的 `protect` 函数调用函数实参 `g` 并保护由 `g` 提升的来自运行时恐慌的调用者。

## Bootstrapping

## 引导

Current implementations provide several built-in functions useful during bootstrapping. These functions are documented for completeness but are not guaranteed to stay in the language. They do not return a result.

```
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
```

当前实现提供了几个在引导过程中有用的内建函数。这些函数因完整性而被保留， 但不保证会继续留在该语言中。它们并不返回结果。

```
函数       行为

print      打印所有实参；实参的格式取决于具体实现
println    类似print，但会在实参之间打印空格并在末尾打印新行
```

## Packages

## 包

Go programs are constructed by linking together *packages*. A package in turn is constructed from one or more source files that together declare constants, types, variables and functions belonging to the package and which are accessible in all files of the same package. Those elements may be exported and used in another package.

Go程序由联系在一起的*包*构造。包由一个或更多源文件构造转化而来， 源文件与其常量、类型、变量和函数声明一同属于该包，且在相同包的所有文件中它们可互相访问。 这些元素可被导出并用于其它包。

## Source file organization

## 源文件的组织

Each source file consists of a package clause defining the package to which it belongs, followed by a possibly empty set of import declarations that declare packages whose contents it wishes to use, followed by a possibly empty set of declarations of functions, types, variables, and constants.

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

每个源文件都由这些部分构成：首先是一个定义该源文件所属包的包子句， 然后是一个可能为空的声明所需内容所在包的导入声明的集，最后是一个可能为空的函数、 类型、变量与常量声明的集。

```
源文件 = 包子句 ";" { 导入声明 ";" } { 顶级声明 ";" } .
```

## Package clause

## 包子句

A package clause begins each source file and defines the package to which the file belongs.

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

包子句起始于每个源文件并定义该文件所属的包。

```
包子句   = "package" 包名 .
包名     = 标识符 .
```

The PackageName must not be the blank identifier.

包名不能为空白标识符.

```
package math
```

A set of files sharing the same PackageName form the implementation of a package. An implementation may require that all source files for a package inhabit the same directory.

一个文件集通过共享相同的包名来构成包的实现。实现可能要求包的所有源文件放置在同一目录下。

## Import declarations

## 导入声明

An import declaration states that the source file containing the declaration depends on functionality of the *imported* package (§Program initialization and execution) and enables access to exported identifiers of that package. The import names an identifier (PackageName) to be used for access and an ImportPath that specifies the package to be imported.

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

导入声明陈述了源文件中所包含的声明，这取决于*已导入*包的功能（§程序初始化与执行）， 并使其能够访问该包的已导出标识符。 导入通过命名一个标识符（包名）用于访问，而导入路径则指定已导出的包。

```
导入声明       = "import" ( 导入指定 | "(" { 导入指定 ";" } ")" ) .
导入指定       = [ "." | 包名 ] 导入路径 .
导入路径       = 字符串字面 .
```

The PackageName is used in qualified identifiers to access exported identifiers of the package within the importing source file. It is declared in the file block. If the PackageName is omitted, it defaults to the identifier specified in the package clause of the imported package. If an explicit period (`.`) appears instead of a name, all the package's exported identifiers declared in that package's package block will be declared in the importing source file's file block and can be accessed without a qualifier.

包名作为限定标识符来访问导入的源文件中包的已导出标识符。 它在文件块中声明。若该包名已省略，则默认由已导入包的包子句中的标识符指定。 若出现一个显式的点号（.）来代替名字， 所有在该包的包块中声明的已导出标识符将在导入源文件的文件块中声明，且无需标识符即可访问。

The interpretation of the ImportPath is implementation-dependent but it is typically a substring of the full file name of the compiled package and may be relative to a repository of installed packages.

导入路径的解释取决于具体实现，但它是一个典型的已编译包的完整文件名的子串，且可能是相对于一个已安装包的仓库的。

Implementation restriction: A compiler may restrict ImportPaths to non-empty strings using only characters belonging to Unicode's L, M, N, P, and S general categories (the Graphic characters without spaces) and may also exclude the characters `!"#$%&'()*,:;<=>?[\]^`{|}` and the Unicode replacement character U+FFFD.

实现限制：编译器会限制导入路径使用只属于Unicode 中L、M、N、P及S一般类别（不带空格的图形字符）中的非空字符串，且不含字

符 !"#$%&'()*,:;<=>?[\]^`{|} 以及Unicode占位字符U+FFFD。

Assume we have compiled a package containing the package clause `package math`, which exports function `Sin`, and installed the compiled package in the file identified by `"lib/math"`. This table illustrates how `Sin` may be accessed in files that import the package after the various types of import declaration.

```
Import declaration          Local name of Sin

import   "lib/math"         math.Sin
import m "lib/math"         m.Sin
import . "lib/math"         Sin
```

假定我们拥有包含包子句 `package math` 的已编译包，它导出函数 `Sin`， 且该包已被安装在标识为`"lib/math"`的文件中。此表单阐明了在各种类型的导入声明之后， `Sin` 在导入该包的文件中如何访问。

```
导入声明                    Sin的本地名

import   "lib/math"         math.Sin
import m "lib/math"         m.Sin
import . "lib/math"         Sin
```

An import declaration declares a dependency relation between the importing and imported package. It is illegal for a package to import itself or to import a package without referring to any of its exported identifiers. To import a package solely for its side-effects (initialization), use the blank identifier as explicit package name:

导入声明用来声明导入包与被导入包之间的从属关系。包导入其自身或导入一个不涉及任何已导出标识符的包是非法的。 要为包的副作用（初始化）而单独导入它，需使用空白标识符作为明确的包名:

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

## An example package

## 一个例子包

Here is a complete Go package that implements a concurrent prime sieve.

```
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()
}
```

以下为实现了并发质数筛的完整Go包。

```
package main

import "fmt"

// 将序列 2, 3, 4, … 发送至信道'ch'。
func generate(ch chan<- int) {
        for i := 2; ; i++ {
                ch <- i  // 将 'i' 发送至信道'ch'。
        }
}

// 将值从信道'src'中复制至信道'dst',
// 移除可被'prime'整除的数。
func filter(src <-chan int, dst chan<- int, prime int) {
        for i := range src {  // 循环遍历从'src'接收的值。
                if i%prime != 0 {
                        dst <- i  // 将'i'发送至'dst'。
                }
        }
}

// 质数筛: 将过滤器串联在一起处理。
func sieve() {
        ch := make(chan int)  // 创建一个新信道。
        go generate(ch)       // 将generate()作为子进程开始。
        for {
                prime := <-ch
                fmt.Print(prime, "\n")
                ch1 := make(chan int)
                go filter(ch, ch1, prime)
                ch = ch1
        }
}

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

## Program initialization and execution

## 程序初始化与执行

### The zero value

### 零值

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

当内存为存储值而分配时，不是通过声明就是通过 `make` 或 `new` 调用来分配。 而在未提供显式的初始化时，内存将被赋予一个默认的初始化。每个这样的值的元素将置为该类型的*零值*： 布尔类型为 `false`，整数类型为 `0`，浮点数类型为 `0.0`， 字符串类型为 `""`，而指针、函数、接口、切片、信道及映射类型则为 `nil`。 该初始化递归地完成，例如，对于结构的数组的每一个元素，若没有值被指定，则将其拥有的字段归零。

These two simple declarations are equivalent:

以下两个简单声明是等价的：

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

After

在

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

the following holds:

之后，以下表达式成立：

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

The same would also be true after

同样，在

```
var t T
```

之后，上面的表达式仍然为真。

## Program execution

## 程序执行

A package with no imports is initialized by assigning initial values to all its package-level variables and then calling any package-level function with the name and signature of

没有导入声明的包通过向所有其包级变量赋予初始值，并调用任何在其源中定义的名字和签名为

```
func init()
```

defined in its source. A package may contain multiple `init` functions, even within a single source file; they execute in unspecified order.

的包级函数来初始化。即使在单个源文件中，一个包也可能包含多个 `init` 函数，它们会按照不确定的顺序执行。

Within a package, package-level variables are initialized, and constant values are determined, in data-dependent order: if the initializer of `A` depends on the value of `B`, `A` will be set after `B`. It is an error if such dependencies form a cycle. Dependency analysis is done lexically: `A` depends on `B` if the value of `A` contains a mention of `B`, contains a value whose initializer mentions `B`, or mentions a function that mentions `B`, recursively. If two items are not interdependent, they will be initialized in the order they appear in the source. Since the dependency analysis is done per package, it can produce unspecified results if `A`'s initializer calls a function defined in another package that refers to `B`.

在一个包中，包级变量或常量值会根据数据依赖的顺序初始化或决定：若 `A` 的初始化器依赖于 `B` 的值，则`A` 将在 `B` 之后设置。若这种依赖形成一个循环， 则会产生一个错误。依赖分析根据词法完成：若 `A` 的值涉及到 `B`， 或包含一个初始化器涉及到 `B` 的值，或递归地涉及一个涉及到 `B` 的函数，则 `A` 依赖于 `B`。若两项并不互相依赖，它们将按照其出现在源里的顺序初始化。 在按包的依赖分析结束后，若 `A` 的初始化器调用另一个引用了 `B` 的包中定义的函数，它会产生不确定的结果。

An `init` function cannot be referred to from anywhere in a program. In particular, `init` cannot be called explicitly, nor can a pointer to `init` be assigned to a function variable.

`init` 函数不能在程序中的任何地方被引用。确切地说，`init` 既不能被显式地调用， 也不能被指针指向 `init` 以赋予函数变量。

If a package has imports, the imported packages are initialized before initializing the package itself. If multiple packages import a package `P`, `P` will be initialized only once.

若一个包拥有导入声明，则被导入的包会在初始化该包自身前初始化。若有多个包导入包 `P`，则 `P` 只会初始化一次。

The importing of packages, by construction, guarantees that there can be no cyclic dependencies in initialization.

根据构造导入的包可保证在初始化中没有循环依赖。

A complete program is created by linking a single, unimported package called the *main package* with all the packages it imports, transitively. The main package must have package name `main` and declare a function `main` that takes no arguments and returns no value.

一个完整的程序通过链接一个单一的、不会被导入的、称为 *主包* 的，带所有其传递地导入包的包创建。 主包必须拥有包名 `main` 且声明一个无实参无返回值的函数 `main`。

```
func main() { … }
```

Program execution begins by initializing the main package and then invoking the function `main`. When the function `main` returns, the program exits. It does not wait for other (non-`main`) goroutines to complete.

程序通过初始化主包然后调用函数 `main` 开始执行。当函数 `main` 返回后，该程序退出。它不会等待其它（非 `main`）Go程完成。

Package initialization—variable initialization and the invocation of `init` functions—happens in a single goroutine, sequentially, one package at a time. An `init` function may launch other goroutines, which can run concurrently with the initialization code. However, initialization always sequences the `init` functions: it will not start the next `init` until the previous one has returned.

包初始化—变量初始化与 `init` 函数的调用—连续地发生在单一的Go程中，一次一包。 一个 `init` 函数可能在其它Go程中启动，它可以与初始化代码一同运行。然而，初始化总是按顺序执行 `init` 函数：直到上一个 `init` 返回后，它才会启动下一个。

## Errors

## 错误

The predeclared type `error` is defined as

预声明类型 `error` 定义为

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

It is the conventional interface for representing an error condition, with the nil value representing no error. For instance, a function to read data from a file might be defined:

它是表示一种错误状态的传统接口，用 `nil` 值表示没有错误。 例如，一个从文件中读取数据的函数可被定义为：

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

## Run-time panics

## 运行时恐慌

Execution errors such as attempting to index an array out of bounds trigger a *run-time panic* equivalent to a call of the built-in function `panic` with a value of the implementation-defined interface type `runtime.Error`. That type satisfies the predeclared interface type `error`. The exact error values that represent distinct run-time error conditions are unspecified.

```
package runtime

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

例如试图索引一个越界的数组这类的执行错误会引发*运行时恐慌*，它等价于 一个定义实现了接口类型 `runtime.Error` 的值的内建函数 `panic` 的调用。该类型满足预声明接口类型 `error`。表示明显的运行时错误状态的准确错误值是不确定的。

```
package runtime

type Error interface {
        error
        // 可能还有其它方法
}
```

## System considerations

## 系统考虑

### Package `unsafe`

### 包 `unsafe`

The built-in package `unsafe`, known to the compiler, provides facilities for low-level programming including operations that violate the type system. A package using `unsafe` must be vetted manually for type safety. The package provides the following interface:

```
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
```

编译器已知的内建包 `unsafe` 为包括违反类型系统操作在内的低级编程提供工具。使用 `unsafe` 的包为了类型安全必须手动进行审查。该包提供以下接口：

```
package unsafe

type ArbitraryType int  // 任意Go类型的简写，它并非真正的类型
type Pointer *ArbitraryType

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

Any pointer or value of underlying type `uintptr` can be converted into a `Pointer` and vice versa.

任何基本类型为 `uintptr` 的指针或值均可转换为 `Pointer`，反之亦然。

The functions `Alignof` and `Sizeof` take an expression `x` of any type and return the alignment or size, respectively, of a hypothetical variable `v` as if `v` was declared via `var v = x`.

函数 `Alignof` 与 `Sizeof` 接受一个任何类型的表达式 `x`， 就好像通过 `var v = x` 声明的变量 `v` 一样，分别返回其列数或大小。

The function `Offsetof` takes a (possibly parenthesized) selector denoting a struct field of any type and returns the field offset in bytes relative to the struct's address. For a struct `s` with field `f`:

函数 `Offsetof` 接受一个表示任何类型结构字段的（可能带括号的）选择者 并返回该字段相对于该结构地址偏移的字节。对于带字段 `f` 的结构 `s`：

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

Computer architectures may require memory addresses to be *aligned*; that is, for addresses of a variable to be a multiple of a factor, the variable's type's *alignment*. The function `Alignof` takes an expression denoting a variable of any type and returns the alignment of the (type of the) variable in bytes. For a variable `x`:

计算机架构可能需要内存地址 *对齐*，即，为使变量的地址为因数的倍数，该变量的类型需要对齐。函数 `Alignof` 接受一个表示任何类型变量的表达式并返回该（类型的）变量对齐的字节。对于变量 `x`：

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

Calls to `Alignof`, `Offsetof`, and `Sizeof` are compile-time constant expressions of type `uintptr`.

调用 `Alignof`、`Offsetof` 和 `Sizeof` 是类型为 `uintptr` 的编译时常量表达式。

## Size and alignment guarantees

## 大小与对齐保证

For the numeric types (§Numeric types), the following sizes are guaranteed:

```
type                                 size in bytes

byte, uint8, int8                     1
uint16, int16                         2
uint32, int32, float32                4
uint64, int64, float64, complex64     8
complex128                           16
```

对于数值类型（§数值类型），以下大小给予保证：

```
类型                                 字节大小

byte, uint8, int8                     1
uint16, int16                         2
uint32, int32, float32                4
uint64, int64, float64, complex64     8
complex128                           16
```

The following minimal alignment properties are guaranteed:

1. For a variable `x` of any type: `unsafe.Alignof(x)` is at least 1.

2. For a variable `x` of struct type: `unsafe.Alignof(x)` is the largest of all the values `unsafe.Alignof(x.f)` for each field `f` of `x`, but at least 1.
3. For a variable `x` of array type: `unsafe.Alignof(x)` is the same as `unsafe.Alignof(x[0])`, but at least 1.

以下最小对齐属性给予保证：

1. 对于任何类型的变量 `x`：`unsafe.Alignof(x)` 至少为1。
2. 对于结构类型的变量 `x`：对于 `x` 的每一个字段 `f`， `unsafe.Alignof(x)` 的值为所有 `unsafe.Alignof(x.f)` 值中最大的，但至少为1。
3. 对于数组类型的变量 `x`：`unsafe.Alignof(x)` 与 `unsafe.Alignof(x[0])` 相同，但至少为1。

A struct or array type has size zero if it contains no fields (or elements, respectively) that have a size greater than zero. Two distinct zero-size variables may have the same address in memory.

若结构或数组类型不包含大小大于零的字段或元素，它们的大小即为零。两个不同的零大小变量在内存中可能有相同的地址。