Golang

sanfusu <sanfusu@foxmail.com>

2018 年 7 月 10 日

第一章　Introduction

这是一份关于 Go 编程语言的参考指南。更多信息以及其余文档，请查看 golang.org

Go 是以系统语言为宗旨而设计的通用目的语言。Go 具有强类型和垃圾回收并且带有对并发编程的显式支持。程序通过 *packages* 构造，其属性可以有效的管理依赖。

语法方面，紧凑且有规律，这允许便捷的使用类似集成开发环境的自动化工具进行分析。

第二章　Notation

语法格式通过扩展的巴克斯范式[1]表示。

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

Productions 是由 terms 以及下列操作符构成的表达式，按递增优先级如下：

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

小写的产生式名被用来标识词法记号。非终止符使用峰驼命名法。词法记号被包含在双引号”” 或者反撇号 `` 中。

a … b 形式标识 a 到 b 的可替代集合。水平省略号 …在本规范中用来表示各种枚举或代码片段。字符 …（与三个字符 ...相反）并不是 Go 语言中的记号。

[1] Extended Backus-Naur Form(EBNF)

第三章　Source code representation

原代码被编码成 UTF-8 格式的 Unicode 文本。这些文本并不是正规化的，所以单个重音编码点与那些通过重音符合字母构成的相同字符并不一样，组合出来的字符被视为两个编码点。简单的来说，本文档将使用未加任何限制的术语 *character*（字符） 来指示源文本中的 Unicode 编码点。

区分每一个编码点，比如，大写字母和小写字母是不同的字符。

实现上的限制：为了和其他工具兼容，编译器可能不允许 NUL 字符（U+0000）出现在原文本中。

同样是实现上的限制：为了兼容其他工具，如果 UTF-8 的字节序记号（U+FEFF）位于源文本的第一个 Unicode 编码点，则编译器可能会忽视该记号。字节序记号可能会被禁止出现在源文本的其他任何地方。

† I　Characters

下列术语被用来标识特定的 Unicode 字符类：

```
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 "Number, decimal digit" */ .
```

在 Unicode 8.0 标准中，第 4.5 节 “General Category” 中，定义了一组字符类别。Go 将字母分类 Lu，LI，Lt，Lm 或者 Lo 中的所有字符均作为 Unicode 字母，并将 Number 类别中的 Nd 作为 Unicode 数字。

† II　Letters and digits

下划线字符 _(U+005F) 被视为一个字母。

```
letter        = unicode_letter | "_" .
decimal_digit = "0" … "9" .
```

```
octal_digit   = "0" … "7" .
hex_digit     = "0" … "9" | "A" … "F" | "a" … "f" .
```

第四章　Lexical elements

† I　注释

注释充当程序文档。有两种形式：

1. 行注释以字符序列 `//` 开头，并且结束于行尾。
2. 通用注释起始于 `/*` 并且结束于随后所遇到的第一个字符序列 `*/`。

注释不能起始于一个 rune 或者字符串文字，或者一个注释内部。不包含换行的通用注释和空格的作用一样。任何其他注释都和换行的作用一致。

† II　记号

记号形成 Go 语言的词汇。有四种类别的记号：标识符，关键词，操作符和标点符，以及字面量。*White space* 由空格 (U+0020)，水平制表 (U+0009)，回车 (U+000D) 和换行 (U+000A) 组成，如果空白字符没起到分割记号的作用，则将会被忽视。另外，换行和文件结尾会触发分号的插入。当输入分解为记号时，下一个记号将会是形成有效记号的最长字符序列。

† III　分号

形式化语法使用分号 ";" 作为产生式的终止符。Go 程序可以通过以下两个规则省略大部分分号：

1. 当输入分解为语言符号时，如果一行的最后一个语言符号为以下几种情况，则会自动插入到行末。
   - 标志符
   - 整型, 浮点，虚数，rune，或者字符串文字
   - 关键词 `break`, `continue`, `fallthrough`, 或者 `return` 中的一个。
   - `++`，`--`，`)`，`]` 或 `}` 运算符和标点符号中的一个。
2. 为了能让复合语句占据单行，`)` 和 `}` 前的分号可以省略。

为了反映惯用的用法，本文档中的代码会使用以上规则来省略掉分号。

† IV 标识符

标识符命名程序条目，比如变量和类型。一个标识符是一个或者多个字母和数字的序列。标识符中的第一个字符必须是一个字母。

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

```
a
_x9
ThisVariableIsExported
αβ
```

部分标识符是预先声明的。

† V 关键词

以下保留的关键词，并且不能用作标识符。

```
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
```

† VI 运算符和标点符号

以下字符序列标识运算符（包括赋值操作符）以及标点符号：

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

† VII 整型字面量

一个整型字面量是标识整型常量的一个数字序列。可选前缀可以设置一个非十进制基：0 为八进制，0x 或者 0X 为十六进制。在十六进制字面量中，字母 a-f 和 A-F 代表 10 到 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 } .
```

```
42
0600
0xBadFace
170141183460469231731687303715884105727
```

† VIII 浮点字面量

浮点字面量是浮点常量的十进制表示。其具有一个整数部分，一个十进制小数点，一个小数部分，和一个指数部分。整数和小数部分有十进制数组成；指数部分为 e 或者 E 后面紧跟着一个可选的有符号十进制指数。可以省略整数部分或者小数部分中的一个；类似小数点或者指数中的一个可以被省略。

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

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

† IX 虚数字面量

虚数字面量是复数常量虚部的十进制表示。他由后面跟着字母 i 的浮点字面量或十进制整数组成。

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

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

```
.12345E+5i
```

† X Rune literals

一个符文字面量表示一个符文常量，一个用来标识 Unicode 编码点的整数值。一个符文字面量使用单引号包含一个或多个字符来表达，如 'x' 或者 '\n'。除换行和未转义的单引号外，任何字符都可以出现在单引号内。使用单引号包含的字符代表了字符本身的 Unicode 值，但是以反斜杠开头的多字符序列会以各种格式对值进行编码。

符文字面量的最简单形式是在引号内表示单个字符；由于 Go 源文本是 UTF-8 编码的 Unicode 字符，所以多个 UTF-8 编码的字节可以使用单个整数值表示。比如字面值 'a' 拥有表示字面量 a 的单个字节，Unicode 编码点为 U+0061，值为 0x61，但是 'ä'则拥有两个字节，用来表示 a 的分音符号，U+00E4，值为 0xe4。

有几个反斜杠转义允许将任意值编码为 ASCII 文本。有四种方式将整型值表示为数字常量：\x 后面跟上两个十六进制数；\u 后面跟上四个十六进制数；\U 后面跟上八个十六进制数；\ 后面跟上三个八进制数。每一种表示方法中，字面量的值为数字使用相应的进制表示的值。

尽管这些表示的结果均为整型，但是他们拥有不同的有效范围。八进制转义表示的值必须在 0 到 255 范围内（包含 0 和 255）。十六进制转义通过构造来满足该条件。转义 \u 和 \U 表示 Unicode 编码点，因此他们所表示的部分值可能是非法的，特别是大于 0x10FFFF 的值以及 surrogate havles。

反斜杠后面跟上特定的单字符表示特殊的值：

```
\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)
```

在符文字面量里的其余以反斜杠开始的序列均是非法的。

```
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'
'\''         // rune literal containing single quote character
'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
```

† XI 字符串字面量

字符串字面量表示从相连的字符序列中所获取的字符串常量。字符串字面量有两种形式：原始字符串字面量和已翻译的字符串字面量。

原始字符串字面量是两个反撇号之间的字符序列，比如 `foo`。除了反撇号自身，任何字符均可以出现在反撇号里。原始字符串字面量的值是由反撇号之间未翻译过（默认是 UTF-8 编码）的字符序列所组成的字符串；值得注意的是，反斜杠没有特殊意义，并且字符串可以包含换行。原始字符串字面量中的回车字符（'\r'）会被舍弃。

翻译过的字符串字面量为双引号之间的字符序列，比如 "bar"。除了双引号自身和换行之外任何字符都可以出现在双引号内。双引号之间的文本形成字面量的值，但其中的反斜杠转义会根据符文字面量中的限制来解释（除了 \' 非法，而 \" 结果字符串中的合法之外）。三个八进制数字（\nnn 和两个十六进制数转义表示结果字符串中的个体字节，其他所有的转义均表示个体字符的 UTF-8 编码（也可能是多字节）。因此字符串字面量里面的 \377 和 \xFF 表示值为 0xFF=255 的单字节，而 ÿ, \u00FF, \U000000FF 和 \xc3\xbf 表示 UTF-8 编码的字符 U+00FF 的两个字节 0xc3 0xbf。

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

下面示例均表示相同的字符串：

```
"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
```

如果源代码将一个字符表示为两个编码点，比如音调符号和字母的组合形式，若放在符文字面值中, 则结果将会出错，若放置在字符串字面量中则会出现两个编码点。

第五章　常量

有 布尔常量，符文常量，整型常量，浮点常量，复数常量和字符串常量。符文，整型，浮点，和复数常量通常为数值常量。

一个常量通过符文, 整型，浮点，虚数或字符串字面量，也可以通过一个表示常量的标识符，常量表达式，结果为常量的转换来表示，或者使用一些内置函数，比如适用于任何值的`unsafe.Sizeof` 或者适用于一些表达式的 `cap` 或 `len`，`real` 和 `imag` 适用于复数常量，`complex` 适用于数值常量。布尔真值由欲声明常量 `true` 和 `false` 来表示。预声明标识符 `iota` 则表示一个整型常量。

一般来说，复数常量是常量表达式的一种形式，并且会在相应的章节中讨论。

数值常量表示任意精度的确切值，并且不会导致溢出。因此没有常量可以表示 IEEE-754 的负零，无穷，非数值。常量既可以具有类型也可是无类型的。字面常量，`true`, `false`, `iota` 以及一些只包含无类型常量操作数的常量表达式属于无类型常量。

一个常量可以通过常量声明或者转换显式的给出其类型，也可以在使用变量声明或赋值亦或表达式中的一个操作数时隐式的给出类型。如果常量值无法表示相应类型的值，则视为错误。

一个无类型常量具有默认类型，该类型为常量在上下文环境中隐式转换的所需值的类型，比如，没有显式类型的短变量声明 `i := 0`。一个无类型常量的默认类型可以为 `bool`，`rune`，`int`，`float64`，`complex128` 或者 `string`，这取决于其是否是相应类型的常量。

实现上的约束：尽管数值常量在语言中具有任意的精度，但是编译器可能会使用具有受限精度的内部表示。也就是说，每一个实现必须：

- 至少使用 256 个 bit 来表示整型常量。
- 浮点常量，包括复数常量的各分部的尾数部分至少使用 256 bits 来表示，有符二进制指数则至少 16 bits
- 如果无法精确的表示整型常量，则视为错误。
- 如果由于溢出而无法表示一个浮点或复数常量，则视为出错。
- 如果受限于精度而无法表示一个浮点或复数常量，则取最接近的可表示的常量。

这些要求同时适用于字面常量和常量表达式的计算结果。

第六章　变量

一个变量时用来保存值的存储位置。变量所允许的值有其类型来决定。

变量声明或者用作函数参数和结果时，以及函数声明的签字或者函数字面量都会为命名变量保留存储空间。通过内置函数 new 或者获取符合字面量的地址，则会在运行时为变量申请变量。这种匿名变量（可以隐式的）通过指针间接寻址来访问。

数组，切片，和结构体类型的结构化变量拥有可以独立寻址的元素和字段。每一个这种元素都表现为一个变量。

变量声明时给出的类型，使用 new 调用或者复合字面值亦或结构化变量的元素类型为变量的静态类型（简称为类型）。接口类型变量具有不同的动态类型，其类型为运行时所赋值给变量的值的具体类型（除非值为预声明标识符 nil，这时候没有类型）。在执行期间动态类型可能会不同，但是存储在接口变量中的值，永远可以赋值给变量的静态类型。

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

一个变量的值可以通过参考表达式中的变量来获取，其值为赋值给变量的最新值。如果变量还未被赋予一个值，则值为其类型的零值。

第七章　类型

一个类型决定了一组值和特定于这些值的操作以及方法。一个类型可以通过类型名来表示，如果有的话也可以通过*类型字面量*来指定。类型字面量从现有类型中组成一个类型。

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

本语言预先声明了一些特定的类型名。其余由类型声明来引入。*复合类型*—数组，结构体，指针，函数，接口，切片，图，通道类型 — 可以通过类型字面量构造。

每一个类型 `T` 都有一个底层类型：如果 `T` 时预声明的布尔，数值，字符串类型或类型字面量中的一个，则相应的底层类型时其自身。否则，`T` 的底层类型是 `T` 在类型声明中引用到的类型的底层类型。

```
type (
    A1 = string
    A2 = A1
)

type (
    B1 string
    B2 B1
    B3 []B1
    B4 B3
)
```

`string`, `A1`, `A2`, `B1` 和 `B2` 的底层类型是 `string`。类型 `[]B1`, `B3`和 `B4` 的底层类型是 `[]B1`。

† I　方法集

一个类型可能会有与其相关联的方法集。一个接口类型的方法集是其接口。其他任何类型 `T` 的方法集由所有使用类型 `T` 作为接收器声明的方法组成。指针类型 `*T` 相应的方法集为所有使用接收器 `*T` 或 `T` 声明的方法的集合（也就是说，包含 `T` 的方法集）。此外，如结构体类型章节中所说，这些规则同样适用于包含嵌入字段的结构体。任何其他类型则具有空方法集。在一个方法集中，每一个方法必须有一个独一无二的非空方法名。

一个类型的方法集决定了该类型实现的接口和可通过该类型接收器所调用的方法。

† II　布尔类型

一个布尔类型代表通过预声明的常量 `true` 和 `false` 表示的布尔真值集合。预声明的布尔类型为 `bool`；这是一个定义的类型。

† III　数值类型

一个数值类型表示整数或浮点值的集合。预声明的依赖于体系结构的数值类型为：

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

n-bit 整数的值具有 n 个 bit 位，并使用二的补码算术表示。

一些预声明的数值类型具有特定于实现的大小：

uint	32 bit 或 64 bit
int	和 uint 具有相同大小
uintptr	足以存储一个未释义的指针值的所有 bit 的无符号整型。

为了避免移植性问题，所有的数值类型都是定义的类型，也就是各不相同，但除了 `byte` 是 `rune` 的别名，以及 `rune` 是 `int32` 的别名。如果一个表达式或赋值中混合了不同的数值类型，则需要进行转换。比如 `int32` 和 `int` 尽管在特定体系结构中具有相同大小，但并不是相同类型。

† IV　字符串类型

一个字符串类型代表字符串值的集合。一个字符串是一个字节序列（可能为空序列）。字符串是不可修改的：一旦创建，便无法改变字符串的内容。预声明的字符串类型为 `string`，为定义的类型。

字符串 s 的长度可以通过内置函数 len 来求得。如果字符串是常量，则长度会在编译期间计算得出。字符串中的字节可通过 0 到 len(s)-1 的整型索引访问。对字符串中的字节元素取地址是非法的；如果 s[i] 是字符串的第 i 个字节，则 &[si] 是无效的。

† V　数组类型

一个数组一个编号的单类型元素（被称为元素类型）序列。元素的数量被称为长度，并且永远不为负。

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

长度作为数组类型的一部分，值必须为类型 int 的非负常量。数组 a 的长度可以通过内置函数 len 计算得出。数组元素可以通过 0 到 len(a)-1 的整型下标寻址。数组类型永远是一维的，但可以组合形成多维类型。

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

† VI　切片类型

切片是用来提供编号的序列元素访问的底层元素中连续片段的描述符。一个切片类型表示其元素类型数组的所有切片的集合。未初始化切片的值是 nil。

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

和数组类似，切片是可索引的并且具有长度。切片 s 的长度可以通过内置函数 len 来计算得出；和数组不同的是，可以在执行期间改变。元素可以通过 0 到 len(s)-1 的整型下标寻址。一个给定元素的切片索引可能会小于底层数组中相同元素的索引。

一个切片，一旦初始化后，便永远的和保存其元素的底层数组相关联。一个切片因此会和其数组以及相同数组的切片共享存储；相反，不同的数组永远具有不同的存储。

底层数组可能会超过切片的结尾。切片的容量为该区间的度量：切片的长度加上数组超过切片的长度。长度为其容量的切片可以通过从原有切片中重新划分。切片 a 的容量可以通过内置函数 cap(a) 计算得出。

通过内置函数 make 创建一个给定的初始化过的切片值，该函数需切片类型和指定长度的参数以及可选的容量参数。通过 make 创建的切片永远会申请一个新的隐藏的数组，返回的切片将会指向这个数组。也就是说，执行

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

会和申请一个数组并分片提供一样的切片，因此下面两个表达式是等价的：

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

和数组一样，切片永远是一维的，但是可以组合成更高维的对象。使用数组的数组是，内部数组永远具有相同的长度；但是在使用切片的切片时（或者切片的数组），内部长度可以动态的变化。此外，内部切片必须独立的初始化。

† VII　结构体类型

一个结构体是一个被称为字段的命名元素序列，每一个元素都有一个名字和类型。字段名既可以显式的指明（标识符列表），也可以隐式的指明（嵌入字段）。在结构体中，非空白字段名必须独一无二的。

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

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

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

一个字段使用一个类型进行声明但没有显式的字段名则称为嵌入的字段。一个嵌入的字段必须指定为类型名 `T` 或者为一个指向非接口类型的指针 `*T`，并且 `T` 本身不能是一个指针类型。无限制的类型名可以作为字段名。

```
// A struct with four embedded fields of types T1, *T2, P.T3 and *P.T4
struct {
    T1        // field name is T1
    *T2       // field name is T2
    P.T3      // field name is T3
    *P.T4     // field name is T4
    x, y int  // field names are x and y
}
```

下面声明是非法的，应为字段名在结构体类型中必须是独一无二的。

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

如果 `x.f` 可以合法的表示一个字段或者方法 `f`，而字段或方法 `f` 是结构体 `x` 中的嵌入字段，则称之为被提升。

除了不能再结构体复合字面量中作为字段名外，提升的字段和普通字段一样。

给定一个结构体类型 `S` 和定义的类型 `T`，以下情况中，提升的方法会被包含在结构体的方法集中：

- 如果 `S` 包含嵌入的字段 `T`，则 `S` 和 `*S` d 的方法集都包含接收器为 `T` 的提升方法。`*S` 的方法集中也会包含带有接收器 `*T` 的提升方法。
- 如果 `S` 包含嵌入字段 `*T`，则 `S` 和 `*S` 同时包含带有接收器 `T` 或 `*T` 的提升方法。

一个字段声明后面可以跟上可选的字符串字面量标签，该标签将成为相应字段声明中所有字段的属性 (attribute)。一个空标签字符串等价于一个缺省标签。一个标签可以通过反射接口可见，并参与结构体的类型识别，其他情况下会被标签忽略掉。

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

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