Go 语言格式化动词

这期算是 《Go 语言实战 》的番外，内容以翻译整理为主。

但随着内容的深入，程序变得越来越复杂，我们将不可避免地会遇到 bug，需要调试，需要（往 console 或 日志）输出调试信息。这时数据的格式化输出变得尤为重要。

实际上，前面已经多次用到了格式化。与其每次用到零碎地介绍，不如集中一期整理好。

介绍、翻译、注释、举例，内容有点多，不必全篇记忆。记住常用部分，剩下的留个印象，需要时回来翻阅就好。

fmt 包

格式化的功能，主要在 `fmt` 包内，`fmt` 是 **format** 的略写。

当然，除了临时的简单调试，直接用 `fmt` 输出到终端（terminal）来调试不太规范。标准输出的内容非常容易丢失，还是写入日志文件方便事后对比分析。

更多的时候，我们会用各种日志库来输出。但这些日志库，要么底层还是调用了 `fmt` ，要么自己实现的格式化也会尽量和 `fmt` 兼容。所以学习格式化仍然是必要的。下面主要的内容均来自 `fmt` 包。

输出 **Printing**

> 注：print 对应的中文翻译应为 印刷、打印。
>
> 但在当前上下文中，print 并非指将内容打印到纸张等介质。而是指的是将各种数据，按照某种格式，转换为字符序列（并输出到抽象文件）的过程。
>
> 所以为了方便理解，我将其替换成了『输出』，请读者知悉。

`fmt` 包中名字里带 `Print` 的函数很多，但无非是两个选项的排列组合。理解了每个部分的含义，一眼就能明白函数的用途。

前缀代表输出目标：

1. `Fprint` 中前缀 `F` 代表 **file**，表示内容 **输出到文件**。

   当然这里的文件是抽象的概念，实际对应的是 `io.Writer` 接口。`Fprint` 开头的函数，第一个参数总是 `io.Writer`。通过传递不同的文件给函数，可以把内容输出到不同的地方。

   常见的用法，是打开一个文件，将文件对象作为第一个参数，将内容输出到该文件。当然，不要被 **文件** 这个词误导了，抽象的文件可以是任意的字节流（stream）。具体到这里，只要是可写入的对象（带 `Write([]byte)(int, error)` 方法），都满足 `io.Writer` 接口。

2. `Print` （没有前缀）表示内容 **输出到标准输出**，也就是 控制台（console）或者叫终端（terminal）。

   实际上调用的是 `Fprint(os.Stdout, a...)`，换言之背后指定输出的文件为标准输出。

3. `Sprint` 中前缀 `S` 表示 **string**，表示内容 **输出到字符串**，然后将字符串返回。

后缀表示格式：

1. `Print` （没有后缀），表示输出时格式不进行额外的处理。

   也就是按参数的 **默认格式**，顺序输出。

2. `Println` 的后缀 `ln` 代表 **line**，表示按行输出。

   实际上它只是比 `Print` 的多做两件事：所有参数之间增加一个空格；输出的最后会追加一个换行符。

3. `Printf` 的后缀 `f` 代表 **format**，表示格式化输出。

   第一个参数是 **格式化字符串**，通过里面的 **格式化动词（verb）** 来控制后续参数值的输出格式。

直接看代码：

```
fmt.Print("Print:", 1, "two", 3.0, "over.")
fmt.Println("Println:", 1, "two", 3.0, "over.")
fmt.Printf("Printf: %d, %s, %f, %s", 1, "two", 3.0, "over.")
fmt.Print("======") // 增加一行观察 Printf 的换行行为
```

输出：

```
Print:1two3over.Println: 1 two 3 over.
Printf: 1, two, 3.000000, over.======
```

给三个函数都输入 5 个参数

- `Print` 将 5 个参数的值以默认格式依次输出，每个值中间没有加分隔符，末尾也没有换行。（因为没有换行，这里特意加了一个句点 `.` 方便区分不同函数的输出）
- `Println` 同样以默认格式输出，只是增加了空格分隔不同的值，并且末尾增加了换行。
- `Printf` 的第一个参数跟其它参数有所区别，必须是格式化字符串（format specifier）。后续参数跟字符串里的格式化动词一一对应，按照动词指定的方式，格式化后填入对应的位置，再一起输出。

接下来，重点就是这些结尾带 `f` 的函数里面，格式化动词的使用。为了跟格式化字符串里一般的内容区分开来，格式化动词以百分号 `%` 开头，后面接一个字母表示。有时候为了更精确地控制格式，在百分号和字母之间还会可能会有标志选项（如整型数填充前导零，浮点数控制小数点的位数）。

在不是特别严谨的语境，**动词** 可以是指由 百分号（%）、标志选项（可选）、字母 这三者组合的整体。但更严谨地说，动词特指后面的字母。理解这一点有助于读懂下面的文档。

下面直接 选译/注释 文档中关于格式化动词的部分：

（部分格式与 Go 的版本有关，这里选译的是当下最新的 1.16 版本）

`fmt` 包实现了格式化输入输出（I/O），其功能类似于 C 语言的 `printf` 和 `scanf` 。格式化 **动词（verbs）** 是从 C 语言的动词中衍生出来的，但更简单。

动词：

一般动词

```
%v      以默认格式输出值 （v 代表 Value，不同类型的默认格式参见下方内容）
        当打印结构体 （struct） 时，加号 (%+v) 会添加字段名
%#v     输出值的 Go 语法表示
%T      输出值类型的 Go 语法表示 （T 代表 Type）
%%      输出一个百分号 (%)；不消耗任何值 （因为 % 用作了动词开头，为了区分，输出 % 需要转义）
```

> 注：只看介绍，所谓输出 “ Go 的语法表示” 并不直观。实际上这是指一个值在代码里的字面量形式。
>
> 对于输出值和字面量一样的类型（布尔类型、数字类型），没有差别；对于字符串，“语法表示意味着带上引号；对于剩下的派生类型，意味着语法表示需要包含类型信息。
>
> 看几个例子：

```go
i := 1
fmt.Printf("%v\n", i)
fmt.Printf("%#v\n", i)
fmt.Printf("%T\n", i)

fmt.Println()

a := struct {
    name string
    age  int
}{name: "alice", age: 24}
fmt.Printf("%v\n", a)
fmt.Printf("%#v\n", a)
fmt.Printf("%T\n", a)

fmt.Println()

type Person struct {
```

```
    name string
    age  int
}
p := Person{name: "bob", age: 26}
fmt.Printf("%v\n", p)
fmt.Printf("%#v\n", p)
fmt.Printf("%T\n", p)

fmt.Println()

s := []int{1, 2, 3}
fmt.Printf("%v\n", s)
fmt.Printf("%#v\n", s)
fmt.Printf("%T\n", s)
```

```
1
1
int

{alice 24}
struct { name string; age int }{name:"alice", age:24}
struct { name string; age int }

{bob 26}
main.Person{name:"bob", age:26}
main.Person

[1 2 3]
[]int{1, 2, 3}
[]int
```

布尔类型（**Boolean**）

```
%t      单词 true 或者 false （t 代表 True value，真值）
```

整型数（**Integer**）

```
%b      以 2 为基数输出 （b 代表 Binary，二进制）
%c      输出对应 Unicode 码点所代表的字符 （c 代表 Character，字符）
%d      以 10 为基数输出 （d 代表 Decimal，十进制）
%o      以 8 为基数输出 （o 代表 Octal，八进制）
%O      以 8 为基数输出，以 0o 为前缀 （同上，大写表示增加前缀）
%q      一个单引号字符，按 Go 语法安全转义。 （q 代表 quote，引号）
%x      以 16 为基数输出，a-f 为小写字母 （x 代表 heXadecimal）
%X      以 16 为基数输出，A-F 为大写字母 （同上，大写表示字母大写）
%U      Unicode 格式：如 U+1234；与 "U+%04X" 效果一样 （U 代表 Unicode）
```

> 注：特别说明一下 `%c` 和 `%q`。
>
> 首先需要注意到，自 1.9 以后，`byte` 类型实际上是 `uint8` 的别名（alias），`rune` 则是 `int32` 的别名。
>
> 这意味着如果以 `%v` 输出，这两个类型都会被当做数字输出。
>
> 想要输出对应的字符，就要考虑使用 `%c` 。
>
> `%q` 也是输出字符，只是有两点区别：
>
> 1. 带单引号
> 2. 对于不可打印字符（non-printable characters，不过叫『不可见字符』更容易理解），会按 Go 语法进行转义。
>
> 举例说，对于字母 A，`%c` 输出 `A`，`%q` 输出 `'A'` ；中文也是类似效果。而对于换行符，对应一个换行的动作，而不是一个可以看得见的字符，用 `%c` 输出会得到一个换行，用 `%q` 输出则得到 `'\n'` （得到一个转义）。
>
> 两者的区别跟 `%v` 与 `%#v` 的区别比较类似。

浮点数和复数的（浮点数）成分

(Floating-point and complex constituents)

```
%b      无小数科学记数法，指数以 2 为底数（但整数部分和指数部分均为十进制数），
        相当于以 strconv.FormatFloat 函数的 'b' 格式，
        如：-123456p-78 （分隔的 p 代表 power (of 2)， 2 的幂）
%e      科学记数法，如：-1.234456e+78 （e 代表 Exponent，指数）
%E      科学记数法，如：-1.234456E+78
%f      无指数的小数，如：123.456 （f 代表 floating-point，浮点数）
%F      %f 的同义词
%g      指数较大时等同于 %e ，否则为 %f 。精度在下面讨论。
        （换言之，%g 取 %e 和 %f 中较短的格式表示）
%G      指数较大时等同于 %E ，否则为 %F
%x      十六进制记数法(指数为十进制，底数为 2 )，如：-0x1.23abcp+20
        （与 %b 的区别是，左边的实数为十六进制，而且可以有小数）
%X      十六进制符号大写，如：-0X1.23ABCP+20
```

> 注：这部分的个别动词，在输出时可能同时混用 二进制、十进制和十六进制，记忆起来会比较混乱。如 `%x`，实数（又叫尾数）为十六进制，底数为 2，指数却又是十进制。建议大家自己在代码里实际尝试，加深印象。
>
> 还好如果不是涉及特殊数值的运算和表示，特殊的动词一般用得不多。日常表示浮点数，掌握 `%f`, `%e` 和 `%g` 就够了。关于浮点数的多种字面量表示方法，可以参考往期的内容 Go 语言实战（2）： 常量与变量 中，浮点数字面量部分。

字符串与字节切片

(对以下动词而言两者等价)

```
%s      字符串或字节切片未经解释的字节（uninterpreted bytes） （s 代表 String，字符串）
%q      一个双引号字符串，按 Go 语法安全转义
%x      以十六进制数输出，小写，每个字节对应两个字符
%X      以十六进制数输出，大写，每个字节对应两个字符
```

> 注：想理解何为 uninterpreted，先要理解何为 interpreted。
>
> 对于脚本语言，解释器就叫 interpreter；分析或执行读入的内容，得到结果的过程，就是解释 interpret。如解释 `1 + 2`，得到 `3`。
>
> 在这里，对于字符串（字符序列）而言，解释主要是指字符转义。`%s` 动词不会对字符序列的内容进行转义。
>
> 但这里有一个非常容易让人迷惑的点，看下面例子：

```
str1 := "1\t2\n3"
fmt.Printf("%s\n-----\n", str1)

str2 := `1\t2\n3`
fmt.Printf("%s\n-----\n", str2)

str3 := []byte{'1', '\\', 't', '2', '\\', 'n', '3'}
fmt.Printf("%s\n-----\n", str3)
```

输出

```
1       2
3
-----
1\t2\n3
-----
1\t2\n3
-----
```

> 第一个例子很容易让人以为 `%s` 还是发生了转义。
>
> 实际上转义发生在源码编译阶段，而不是输出阶段。也就是对于双引号字符串，编译器已经对其完成了转义。`str1` 储存在内存里的内容，是 [‘1’, 9, ‘2’, 10, ‘3’]，其中 9 就是制表符的 ascii 码，10 是 换行符的 ascii 码。这里已经找不到 反斜杠、字母 t 和 n 了。

再看接下来的两个例子就很好理解了。反引号字符串告诉编译器不要转义，字节切片则直接逐个指定每个字节的内容，所以 `str2` 和 `str3` 的字节序列里，储存的就是字面意义的 “\t” 和 “\n” 。

当然还有更直观的方式，可以看出字节序列的不同：

```
str1 := "1\t2\n3"
fmt.Printf("% x\n", str1)

str2 := `1\t2\n3`
fmt.Printf("% x\n", str2)

str3 := []byte{'1', '\\', 't', '2', '\\', 'n', '3'}
fmt.Printf("% x\n", str3)
```

输出：

（具体每个十六进制数对应的字符，这里就不再解释了，反正不同是非常直观的）

```
31 09 32 0a 33
31 5c 74 32 5c 6e 33
31 5c 74 32 5c 6e 33
```

切片

```
%p      以十六进制数表示的第 1 个元素（下标 0）的地址，以 0x 开头
        （p 代表 Pointer，指针，也就是以指针形式输出地址）
```

指针

```
%p      十六进制数地址，以 0x 开头
        %b, %d, %o, %x 和 %X 动词也可以用于指针，
```

```
3           实际上就是把指针的值当作整型数一样格式化。
```

%v 的默认格式

```
bool:                                   %t
int, int8 等有符号整数:          %d
uint, uint8 等无符号整数:        %d, 如果以 %#v 输出则是 %#x
float32, complex64 等:          %g
string:                                 %s
chan:                                   %p
指针:                                   %p
```

复合对象

对于复合对象，将根据这些规则，递归地打印出元素，像下面这样展开：

```
struct:                         {field0 field1 ...}
array, slice:           [elem0 elem1 ...]
maps:                           map[key1:value1 key2:value2 ...]
上述类型的指针:          &{}, &[], &map[]
```

宽度与精度

宽度由紧接在动词前的一个可选的十进制数指定。如果没有指定，则宽度为表示数值所需的任何值。

精度是在（可选的）宽度之后，由一个句点（`.`，也就是小数点）和一个十进制数指定。如果没有句点，则表示使用默认精度。如果有句点，句点后面却没有数字，则表示精度为零。例如：

```
%f              默认宽度，默认精度
%9f             宽度 9，默认精度
%.2f    默认宽度，精度 2
%9.2f   宽度 9，精度 2
```

```
%9.f    宽度 9，精度 0
```

宽度和精度以 Unicode 码点为单位，也就是 runes。（这与 C 语言的 `printf` 不同，后者总是以字节为单位。）标志中的任意一个或两个都可以用字符 `*` 代替，从而使它们的值从下一个操作数获得（在要格式化的操作数之前），这个操作数的类型必须是 `int` 。

> 注：`*` 的用法并不直观，举个例子就很好理解。
>
> `fmt.Printf("%*.*f", 6, 3, 4.5)`
>
> 输出 `4.500` （注意 4 前面有一个并不明显的空格，加上数字和小数点，宽度正好为 6 ）

对于大多数的值来说，宽度是要输出的最小符号（rune）数，必要时用空格填充。

然而，对于 字符串、字节切片 和 字节数组 来说，精度限制了要格式化的输入长度（而不是输出的大小），必要时会进行截断。通常它是以符号（rune） 为单位的，但当这些类型以 `%x` 或 `%X` 格式进行格式化时，以字节（byte）为单位。

对于浮点值，宽度设置字段的最小宽度，精度设置小数点后的位数；但对于 `%g` / `%G`，精度设置最大的有意义数字（去掉尾部的零）。例如，给定 `12.345`，格式 `%6.3f` 打印 `12.345`，而 `%.3g` 打印 `12.3`。`%e`、`%f` 和 `%#g` 的默认精度是 6；对于 `%g`，默认精度是唯一识别数值所需的最少数字个数。

> 注：关于如何精确控制浮点值的宽度和精度，这段说明看似说清楚了，实际执行中却常常让人迷惑。看网上的讨论，已经有很多人在诟病这一点。跟更早的文档相比，现在的版本好像已经调整过表述，但是帮助有限。
>
> 如果你需要精确控制以达到排版对齐一类的目的，可以参考这个讨论 https://stackoverflow.com/questions/36464068/fmt-printf-with-width-and-precision-fields-in-g-behaves-unexpectedly
>
> 讨论篇幅过长且拗口，不再翻译。总的来说，精度控制有效数字，但因为有效数字不包括小数点和前导零，带前导零和小数点的数会更长；宽度控制最小宽度，在长度不足时会填充到指定宽度，但超出时并不会截断，总位数仍然可能超出。最后你可能需要制表符 `\t` 来帮助对齐。

对于复数，宽度和精度分别应用于两个分量（均为浮点数），结果用小括号包围。所以 `%f` 应用于 `1.2+3.4i` 输出 `(1.200000+3.400000i)` 。

其它标志

```
+       始终输出数字值的符号（正负号）；
```

```
        对 %q(%+q) 保证只输出 ASCII 码 （ASCII 码以外的内容转义）
-       空格填充在右边，而不是左边
#       备选格式：二进制(%#b)加前导 0b ，八进制(%#o)加前导 0 ；
        十六进制(%#x 或 %#X)加前导 0x 或 0X ；%p (%#p) 取消前导 0x ；
        对于 %q ，如果 strconv.CanBackquote 返回true，则输出一个原始（反引号）字符串；
        总是输出 %e, %E, %f, %F, %g 和 %G 的小数点；
        不删除 %g 和 %G 的尾部的零；
        对于 %U (%#U)，如果该字符可打印（printable，即可见字符），则在 Unicode 码后面输出字符，
        例如 U+0078 'x'。
' '     (空格) 为数字中的省略的正号留一个空格 (%d)；
        以十六进制输出字符串或切片时，在字节之间插入空格 (%x, %X)
0       用前导零而不是空格来填充；
        对于数字来说，这会将填充位置移到符号后面
```

动词会忽略它不需要的标志。例如十进制没有备选格式，所以 `%#d` 和 `%d` 的行为是一样的。

对于每个类似 `Printf` 的函数，都有一个对应的 `Print` 函数，它不接受格式，相当于对每个操作数都应用 `%v` 。另一个变体 `Println` 在操作数之间插入空格，并在结尾追加一个换行。（注：这个我们在开头就已经讨论过）

无论用什么动词，如果操作数是一个接口值，则使用内部的具体值，而不是接口本身。因此：

```
var i interface{} = 23
fmt.Printf("%v\n", i)
```

会输出 `23` 。

除了使用动词 `%T` 和 `%p` 输出时，对于实现特定接口的操作数，需要考虑特殊格式化。以下规则按应用顺序排列：

1. 如果操作数是 `reflect.Value`，则操作数被它所持有的具体值所代替，然后继续按下一条规则输出。

2. 如果操作数实现了 `Formatter` 接口，则会被调用。在这种情况下，动词和标志的解释由该实现控制。

3. 如果 `%v` 动词与 `#` 标志 (`%#v`) 一起使用，并且操作数实现了 `GoStringer` 接口，则该接口将被调用。

如果格式 （注意 `Println` 等函数隐含 `%v` 动词）对字符串有效 (`%s` , `%q` , `%v` , `%x` , `%X`)，则适用以下两条规则：

1. 如果操作数实现了 `error` 接口，将调用 `Error` 方法将对象转换为字符串，然后按照动词（如果有的话）的要求进行格式化。
2. 如果操作数实现了 `String() string` 方法，则调用该方法将对象转换为字符串，然后按照动词（如果有的话）的要求进行格式化。

对于复合操作数，如 切片 和 结构体，格式递归地应用于每个操作数的元素，而不是把操作数当作一个整体。因此， `%q` 将引用字符串切片中的每个元素，而 `%6.2f` 将控制浮点数组中每个元素的格式。

然而，当以适用于字符串的动词(`%s` , `%q` , `%x` , `%X`)，输出一个字节切片时，它将被视为一个字符串，作为一个单独的个体。

为了避免在以下情况出现递归死循环：

```
type X string
func (x X) String() string { return Sprintf("<%s>", x) }
```

在触发递归之前先转换类型：

```
func (x X) String() string { return Sprintf("<%s>", string(x)) }
```

无限递归也可以由自引用的数据结构触发，例如一个包含自己作为元素的切片，然后该类型还要有一个 `String` 方法。然而，这种异常的情况是非常罕见的，所以 `fmt` 包并没有对这种情况进行保护。

在输出一个结构体时， `fmt` 不能，也不会，对未导出字段调用 `Error` 或 `String` 等格式化方法。

显式参数索引

在 `Printf` , `Sprintf` 和 `Fprintf` 中，默认的行为是，每个格式化动词对调用中传递的连续参数进行格式化。然而，紧接在动词前的符号 `[n]` 表示第 n 个单一索引参数将被格式化。在宽度或精度的 `*` 前同样的记号，表示选择对应参数索引的值。在处理完括号内的表达式 `[n]` 后，除非另有指示，否则后续的动词将依次使用 n+1、n+2等参数。

举例：

```
fmt.Sprintf("%[2]d %[1]d\n", 11, 22)
```

将输出 `22 11` 。 而

```
fmt.Sprintf("%[3]*.[2]*[1]f", 12.0, 2, 6)
```

等价于

```
fmt.Sprintf("%6.2f", 12.0)
```

将输出 `12.00` （注意 12 前有一个空格）。

因为显式索引会影响后续的动词，所以这个记号可以通过重置索引为第一个参数，达到重复的目的，来多次打印相同的数值：

```
fmt.Sprintf("%d %d %#[1]x %#x", 16, 17)
```

将输出 `16 17 0x10 0x11` 。

格式错误

如果给一个动词提供了无效的参数，比如给 `%d` 提供了一个字符串，生成的字符串将包含对问题的描述，像以下这些例子：

```
Wrong type or unknown verb: %!verb(type=value)
        Printf("%d", "hi"):        %!d(string=hi)
Too many arguments: %!(EXTRA type=value)
        Printf("hi", "guys"):      hi%!(EXTRA string=guys)
Too few arguments: %!verb(MISSING)
        Printf("hi%d"):            hi%!d(MISSING)
Non-int for width or precision: %!(BADWIDTH) or %!(BADPREC)
```

```
        Printf("%*s", 4.5, "hi"):  %!(BADWIDTH)hi
        Printf("%.*s", 4.5, "hi"): %!(BADPREC)hi
Invalid or invalid use of argument index: %!(BADINDEX)
        Printf("%*[2]d", 7):       %!d(BADINDEX)
        Printf("%.[2]d", 7):       %!d(BADINDEX)
```

所有的错误都以字符串 `%!` 开头，有时后面跟着一个字符（动词），最后以括号内的描述结尾。

如果一个 `Error` 或 `String` 方法在被输出例程调用时触发了 panic，那么 `fmt` 包会重新格式化来自 panic 的错误消息，并在其上注明它是通过 `fmt` 包发出的。例如，如果一个 `String` 方法调用 `panic("bad")`，则产生的格式化消息看起来会是这样的

```
%!s(PANIC=bad)
```

`%!s` 只是显示失败发生时使用的打印动词。然而，如果 panic 是由 `Error` 或 `String` 方法的 nil 接收者（receiver）引起的，则输出的是未修饰的字符串 `<nil>`。

实际上，这一套函数的命名规则和格式化动词，基本继承自 C 语言，只是做了少量的调整和改进。有 C/C++ 经验的朋友应该非常熟悉。没有写过 C 的朋友，经过整理，也会有助于记忆和理解。

上述内容涉及到类型方面的知识，如果有朋友还不熟悉，可以参考往期的内容：Go 语言实战（3）： 类型

Errorf()

Go 在 1.13 中专门为 `fmt.Errorf()` 新增了一个动词 `%w` 。文档是这样介绍的：

> 如果格式化字符串包含一个 `%w` 动词，并且该动词对应一个 `error` 操作数，`Errorf` 返回的 `error` 将实现一个 `Unwrap` 方法，会返回前面传入的 `error` 。包含一个以上的 `%w` 动词 或 提供一个没有实现 `error` 接口的操作数是无效的。无效的 `%w` 动词是 `%v` 的同义词。

文档的说明严谨但拗口。好在这部分代码不长，直接贴出来看看：

```
// go/src/fmt/errors.go

```

```go
func Errorf(format string, a ...interface{}) error {
    p := newPrinter()
    p.wrapErrs = true
    p.doPrintf(format, a)
    s := string(p.buf)
    var err error
    if p.wrappedErr == nil {
        err = errors.New(s)
    } else {
        err = &wrapError{s, p.wrappedErr}
    }
    p.free()
    return err
}

type wrapError struct {
    msg string
    err error
}

func (e *wrapError) Error() string {
    return e.msg
}

func (e *wrapError) Unwrap() error {
    return e.err
}
```

传入的参数，实际上通过 `p.doPrintf` （一系列 `Printf` 函数的内部实现） 变成了字符串 `s`。此时 `%w` 是 `%v` 的同义词，参数里即使有 `error` ，也是取 `Error()` 方法返回的字符串。

然后再看是否有需要包裹（wrap）的 `error` 。这需要一个 `%w` 动词并对应的操作数满足 `error` 接口，仅有其中之一，或者参数顺序不对应，都不算。如无，则通过 `errors.New(s)` 返回一个只有字符串的最基本的 `error` ； 否则返回一个同时包含 格式化字符串 和 内部错误的 `wrapError` 。跟基本的 `error` 相比，它多了一个获取内部错误的 `Unwrap` 方法。

输入 Scanning

除了输出（Printing），`fmt` 包还提供了一系列类似的函数负责输入，将特定格式的文本（formated text）解析为对应的值。

与 Printing 类似，通过前后缀的组合来区分读取的来源和格式化方式：

- 前缀：`Fscan` 表示从文件（`io.Reader`）读取；`Scan` （无前缀）表示从标准输入 `os.Stdin` 读取；`Sscan` 表示从字符串读取；
- 后缀：`Scan` （无后缀）表示把换行当成普通空白字符，遇到换行不停止；`Scanln` 表示遇到换行或者 `EOF` 停止；`Scanf` 表示根据格式化字符串里的动词控制读取。

Scanning 使用几乎一样的一系列动词（除了没有 `%p`，`%T` 动词，没有 `#` 和 `+` 标志），这里不再重复介绍这些动词。动词的含义也基本一致，只是在非常细微的地方，为方便输入做了变通：

- 对于浮点数和复数，所有有效动词都是等价的；进制以文本内容、而不是动词为准。（因为尾数和指数可能是不同的进制，无法单靠动词指定）
- 对于整型数，则以动词指定的进制为准；仅在 `%v` 时依靠前缀判断进制。
- 宽度仍然有效，用来限制读取的最大符号数（去掉前导空格）；如 `123456` ，如果用 `%3d%d` 来解析，会被理解为 `123` 和 `456` 两个数；精度不再有意义。
- 对于数字类型，数字之间可以添加下划线提高可读性，读取时会忽略下划线，不影响解析。

其它更细致的差别（包括与 C 语言的差别），像符号的消耗，空白字符串的匹配，就不再展开。建议大家自己尝试，遇到问题直接去看文档。

参考资料

- https://pkg.go.dev/fmt ： `fmt` 官方文档，翻译整理难免有理解偏差，以文档为准
