Go 速成指南

作者	首次发布	最新发布	修订
徐涛	2019 年 11 月	2020 年 11 月	3

前言

为 Go 语言编一本速成指南是很早就计划好的事情，但是主要因为 Go 的依赖包管理始终处在变革过程中而拖延至今。自 Go 1.13 开始，Go Module 完全投入使用，这本速成指南又重新获得了编写动力。本指南将基于 Go 1.13 版本特性编写，并且在后续的修订中将逐步加入 Go 的各个新版本的新特性。如果你发现许多特性与你使用的 Go 不太一样，那么请先查看一下 Go 的版本，也许你需要升级一下，也可能这本指南已经过时需要更新。

人们在谈论 Go 语言的时候，往往都是将高并发作为 Go 的代号与其他语言相比较，然而许多架构师朋友也是这么做的。但是在实际学习工作中，高并发只是 Go 的众多特性之一，在很多被我们选择性忽略掉的特性中，Go 往往也具备拿出来与其他技术方案一较高下的能力。

本指南将尽可能的对 Go 语言特性做深入描述，方便读者在日常学习工作中 的参考以及与其他语言进行比较。

一些基础知识

Go 是一门全新的静态类型语言，被称为“NextC”的 Go，既简洁又强大。Go 与 C99、C11 等相似之处很多，但是相比 C，Go 没有历史包袱，可以从零开始，这就让 Go 形成了一个规则严谨、简单强悍的语言环境。Go 是由 Google 开发的的开源语言，但又不完全交予开源社区管理，这就保证了 Go 语言前进的活力。

Go 的特性

由于与 C 有着深厚的历史渊源，Go 的代码编写起来与 C 很相似，甚至可以直接与 C 语言搭配混合编写 C 语言代码。而由于 Go 可以从零开始设计，所以 Go 除了借鉴了 C 语言之外，还借鉴了 Java、Python 等多种流行的语言，这也就造就了 Go 语言学习曲线平缓的特点。Go 语言拥有强大的标准库和工具链，仅依赖标准库提供的功能，只需寥寥几句就可以实现一个高性能的 Web Server。Go 的工具链则提供了包括编译、代码格式化、错误检查、查看和形成帮助文档、下载和更新第三方依赖库等大部分常用功能。Go 的第三方支持库应该是目前市面上最强悍的，这是因为 Go 直接将 Github 作为其依赖库存储位置，可以直接对其进行检索和拉取。第三方依赖库只要发布到 Github 上，就可以被其他的 Go 用户获取到。

许多漫画中都将 Go 比喻成一把开火以后必须检查发射状态的发令枪，这其实是 Go 语言的异常处理特色。在习惯之后就会发现，如果想写出一个能够因异常而中止执行的程序，还真的要故意一下才行。

Go 内置了垃圾回收功能，但是因为对指针、并发的支持，所以 Go 的垃圾回收压力更大，但好在每次新的版本发布都会对垃圾回收功能做出改进。所以在一般规模的应用中，大可不必重点考虑垃圾回收问题，但是如果出现服务停顿，那么就需要对代码进行优化一下了。

一般的 Go 语言项目在编译之后仅有一个可执行文件，也就是说 Go 是采用静态链接编译的，所有依赖库都会被打包进可执行文件。项目一旦编译完成，无需任何附加支持就可以直接部署。但随着 Go 增加了 buildmode 功能，使得 Go 也开始支持动态链接，这为扩大 Go 的用途增加了支持。

Go 语言制定了严格的语法标准和代码书写格式等，这些在其他语言中最多是警告的项目，在 Go 语言中都被归类为了致命错误并且不能编译通过。虽然需要花点时间去适应，但是这样做的好处是消除了多人合作开发时产生的习惯差异。所以在进行 Go 语言编程的时候要注意养成正确的代码书写习惯，减少因为代码书写习惯造成的编译失败。

指针是许多编程者被劝退的主要原因。每每提到指针，所有人都会想到 C/C++中自由而复杂的指针应用。但是在 Go 中，指针的使用不仅十分普遍，而且十分简单，只需要牢记指针的含义，即可自由熟练的使用。

通过引入 goroutine 的概念，Go 的并发编程变得十分简单。Go 使用 CSP（通信顺序进程）模型来作为 goroutine 之间通信的推荐方式，多个 goroutine 之间只能通过通信原语实现协作，从而摆脱了传统多线程所使用的同步手段。但这并不代表 Go 没有提供完备的读写锁功能。

读者在看其他的 Go 语言教程的时候，经常可以看到两个常见的环境变量：`GOPATH` 和 `GOROOT`，从 Go 1.11 版本开始还进入了一个新的环境变量 `GO111MODULE`。Go 开发环境的正常工作需要用到这几个变量，所以在使用 Go 进行开发的时候要小心设置。

由于一些众所周知的原因，Google 的网站在中国大陆是无法直接访问的，这也就导致 Go 语言的官网也不是那么容易就能访问到的。如果需要下载已经编译好的开发环境安装包或者下载 `golang.org/x` 下的功能库，都需要花上一些心思，但好在目前 Go 已经提供了代理设置，国内也出现了性能不错的 Go 功能库代理服务可供使用，其具体设置方法后文将会细述。

算法复杂度的衡量

我们在很多描述算法复杂度的场合经常可以看到类似于 $O(n)$ 、$O(log\ n)$ 的标记，这种标记是被用来在复杂性理论中对算法所消耗的 CPU 和内存资源（时间资源与空间资源）给出近似边界的，表示资源消耗与需要处理的项的数量 n 之间的关系。资源消耗越高就表示算法复杂性越高，处理时所需要的时间越长。

常见的标记及其含义主要有以下几种：

1. $O(1)$，算法消耗常量资源，与 n 无关。
2. $O(log\ n)$，算法消耗资源与 n 成对数关系，随着 n 的增长，算法消耗资源增长不多。
3. $O(n)$，算法消耗资源与 n 成线性正比关系，随着 n 的增长，算法消耗资源成线性增长。
4. $O(n^2)$，算法消耗资源与 n 的二次方成正比关系，随着 n 的增长，算法消耗资源会较快增加。
5. $O(n!)$，算法消耗资源与 n 的阶乘成正比关系，随着 n 的增长，算法消耗资源也会快速增长。
6. $O(n^m)$，算法消耗资源与 n 的 m 次幂成正比关系，随着 n 的增长，算法消耗资源会飞速增长。

这里提供一些常见的操作的复杂度衡量供参考。

复杂度	常见操作描述
$O(1)$	线性执行的代码，无循环结构
$O(n)$	存在一个循环 n 次的循环体
$O(log\ n)$	指定循环 n 次的循环体在循环 i 次后退出循环，复杂度上界为 $log_2\ n = x$
$O(n^2)$	两个循环 n 次的循环体嵌套在一起

下载与安装

对于 Windows 平台的 Go 开发环境，Google 提供了编译好的安装包，可以直接根据自己的系统环境下载相应的安装包即可。但是在本文推荐使用 Scoop 软件管理工具来完成安装。

Scoop 软件管理工具可以通过以下命令在 PowerShell 中安装。

```
Set-ExecutionPolicy RemoteSigned -scope CurrentUser
iex (new-object net.webclient).downloadstring("http://get.scoop.sh")
```

完成安装后，在 PowerShell 提示符下输入命令 `scoop`，如果能够成功执行，即表示 Scoop 已经安装成功。如果提示错误，可以将文件夹 `C:\Users\<用户名>\scoop` 删除，之后重新执行 Scoop 安装命令。

使用 Scoop 安装 Go 只需要执行以下命令即可。

```
scoop install go
```

Scoop 会自动安装 Go 语言所必需的全部依赖项，并将 Go 工具链加入到 `PATH` 环境变量中。

对于 macOS 平台，可以直接使用 Homebrew 软件管理软件完成安装，命令如下。

```
brew install go
```

Homebrew 同样会自动安装 Go 语言所必须的全部依赖。

在Linux或者Windows 10的Linux子系统中安装Go工具链，可以使用以下指令来完成安装，假设安装目录为 `/home/go`。

```
cd /home
wget -O golang.tar.gz https://studygolang.com/dl/golang/go1.13.4.linux-amd64.tar.gz
tar -xzvf golang.tar.gz
mkdir go-projects
vim /etc/profile
```

在 `/etc/profile` 中添加以下内容来配置Go的环境。

```
export GOROOT=/home/go
export GOBIN=$GOROOT/bin
export GOPKG=$GOROOT/pkg/tool/linux_amd64
export GOARCH=amd64
export GOOS=linux
export GOPATH=/home/go-projects
export GO111MODULE=auto
export GOPROXY=https://goproxy.io,direct
export PATH=$PATH:$GOBIN:$GOPKG:$GOPATH/bin
```

之后运行以下命令刷新环境并测试。

```
source /etc/profile
go version
```

环境变量设置

前面曾提到配置 Go 开发环境需要三个环境变量，分别是 `GOPATH`、 `GOROOT` 和 `GO111MODULE`，但除此之外，还有几个环境变量可以选择配置，其功能和如下。

- `GOROOT`，指示 Go 工具链的安装路径，使用 Scoop 安装的 Go 工具链一般位于 `C:\Users\<用户名>\scoop\apps\go` 下。
- `GOPATH`，编译后二进制程序存放的目的地以及使用 `import` 引入依赖库时的搜索路径，在引入 Go Module 之前，所有的 Go 项目必须都在 `$GOPATH/src` 目录下。需要**注意**的是，`GOPATH` 不要设置为 Go 工具链的安装路径。`GOPATH` 目录中一般还存在有 `bin` 和 `pkg` 两个目录，分别用于存放编译好的二进制程序和中间连接文件。如果已经开始使用 Go Module 来开发项目，那么 Go 项目就不要求必须放置在 `GOPATH` 下了。
- `GO111MODULE`，指示是否使用 Go Module 功能，从 Go 1.11 版本开始引入。可以设置为 `on`（开启）、`off`（关闭）和 `auto`（由 Go 控制）。当设置为 `auto` 时，位于 `$GOPATH` 中的项目不开启 Go Module 模式，`$GOPATH` 以外的项目开启 Go Module 模式。如果使用的 Go 的版本在 1.14 以上，则可以直接设置为 `auto`，因为目前大部分流行的 Go 第三方库都已经实现了模块化。
- `GOPROXY`，设置依赖库下载代理。在中国大陆通常有两个代理可以使用：`https://goproxy.io` 和 `https://goproxy.cn`，设置时可择一使用。设置格式一般为：`GOPROXY=https://goproxy.io,direct`，加入 `direct` 是为了避免所有依赖库的获取都走代理。
- `GOOS` 和 `GOARCH`，用于在命令行中设置以使用交叉编译功能。

基本项目结构

在不使用 Go Module 方式编写 Go 程序，只需要在 `$PATH/src` 目录下建立一个新目录就可以开始项目的编写了。

但是如果是使用 Go Module 方式编写 Go 程序，则不需要在 `$PATH/src` 目录中创建项目目录，项目目录可以存放在任意位置，并且项目需要使用以下命令创建。

```
> go mod init 项目名称
```

Go 工具链会自动创建一个以指定项目名称命名的目录，并且新建一个 `go.mod` 文件用于保存项目的依赖描述。

每个应用项目需要至少存在一个名为 `main` 的包，并且其中要存在一个唯一的名为 `main()` 的函数，这是整个程序的入口，Go 工具链会自动寻找这个 `main()` 函数。如果项目只是一个库，那么项目中就不必再存在 `main` 包，也不需要存在 `main()` 函数。

如果是在一个旧项目下进行工作并需要将其转换到新的 Go Module 模式下，可以在项目根目录下执行以下命令来生成一个 `go.mod` 文件。

```
> go mod init
```

`go.mod` 文件一般不用手动编辑，其中内容会随着 Go 工具链命令的使用自动更新。但个别时候依旧需要手动编辑，例如某些依赖库被墙而不能下载，需要添加替换项。`go.mod` 中的内容格式一般如下例所示。

```
module 模块名称

go 1.13

require (
  github.com/hajimehoshi/ebiten v1.8.1
  // 这里是项目依赖库说明，其中声明了依赖库位置以及指定的版本号
  // 如果某个依赖库没有做到兼容Go Module，那么依赖就如以下所示
  github.com/docopt/docopt-go v0.0.0-20180111231733-ee0de3bc6815
  // 不兼容Go Module的依赖，需要指定Git提交标识作为版本说明
  // 如果使用go get安装依赖，Git提交标识会自动获取
)

replace (
  golang.org/x/exp => github.com/golang/exp v0.0.0-20180710024300-14dda7b62fcd
  // 这里一般书写被墙掉资源的替代位置
)
```

基本项目管理

前面仅使用 `go mod init` 来完成了项目以及 `go.mod` 文件的创建。但实际上 Go 工具链中还提供了更多的子命令用于项目管理，其中常用的主要有以下这些。

- `go mod download`，下载 `go.mod` 中定义的依赖到本地。
- `go mod edit`，编辑 `go.mod` 文件。
- `go mod tidy`，整理依赖关系，添加丢失的依赖并同时删除不需要的依赖。
- `go mod vendor`，将依赖复制到 `vendor` 目录下。
- `go mod verify`，校验依赖。
- `go run`，执行项目或执行项目中指定文件。
- `go build`，编译项目。
- `go test`，执行单元测试。
- `go get`，向项目中添加依赖，使用参数 `-u` 可将依赖升级到最新的次要版本或修订版本。
- `go fmt`，格式化项目代码。
- `go doc`，提取代码注释形成文档。

包与引用

在开始接触 Go 的基本语法之前，先来对每个 Go 文件都必须使用的两个结构做一个说明。一个 Go 语言项目一般是通过模块（module）和包（package）来包装管理的，一般一个项目会被称为一个 module，而项目则是由若干 package 组成的。

Go 的每一个文件都必须在开头声明其所在的包，格式如下。

```
1 package 包名
```

Go 中的包有两种类型，一种是 `main` 包，其中可以包括一个唯一的 `main()` 函数作为程序入口；一种是非 `main` 包，在定义时一般习惯上使用文件夹的名字作为包名。

> 使用文件夹的名称作为包名不是必须的，只是一种方便源码管理的习惯。

包必须在一个文件夹内，并且一个文件夹内只允许存在一个包。Go 中的包可以是由多个文件组成，只要它们都声明所在的包相同并且全部位于相同的目录下（不包括子目录），Go 在编译时会自动将其拼合为一个整体。一个包下的所有文件拥有相同的定义域，可以直接访问同一包下其他文件中定义的内容（例如变量、常量或者函数等）。

> 如果项目根目录声明使用了 `main` 包，那么根目录下其他的文件也都必须声明位于 `main` 包中。

要使用其他包中定义的内容，需要使用 `import` 语句。`import` 语句一般紧跟在 `package` 定义之后。每个 Go 文件中只允许存在一个 `import` 语句，当引用多个包时，`import` 语句将自动使用 `()` 拼合，格式如下例所示。

```
import (
  "fmt"
  "github.com/kataras/iris/v12"
  "math/big"
  "time"
)
```

`import` 语句常用的使用方式主要有以下这些，提供了不同功能的引入功能。

```
import (
  // 普通引入，其中内容以 package.A 的方式访问，例如：fmt.Printf()
  // 若被引入的包名称为 github.com/author/lib，则对其中内容的访问只需要以路径最后一个单元的内容为名称，即 lib.A
  "fmt"
  // 匿名引入，只引入副作用功能，并不在代码中实质性使用
  // 匿名引入将只执行被引入包中的初始化函数
  _ "github.com/xxxx/xxxx"
  // 全局引入，将被引入包中的内容与当前包融合，被引入的内容可直接使用 A 的方式访问
  . "github.com/xxxx/xxxx"
  // 具名引入，对被引入的包进行命名，其中的内容以 name.A 的方式访问
  name "github.com/xxxx/xxxx"
)
```

Go 会按照以下顺序来搜索被引入包的位置。

1. 当前项目目录。
2. `go.mod` 中定义的包位置。
3. 各级别的 vendor 目录，具体可参考后文对于 vendor 目录的说明。
4. `GOPATH` 目录下。
5. `GOROOT` 目录下。

可访问性

与 Python 等语言不同，Go 对于包、结构体等可导出结构中的成员默认是私有的，并且其访问性是通过标识符首字母的大小写来确定的。

如果不想使一个包成员或者结构体成员被外界访问到，那么成员的标识符首字母需要是小写字母。当标识符为大写字母时，该成员可以被外界访问到。

例如有以下包的定义：

```
package expa

// 定义一个私有函数成员
func somePrivateFunction() {}

// 定义一个公有函数成员
func SomePublicFunction() {}
```

那么在使用这个包的代码中效果是这样的：

```
package main

import expa

func main() {
  // 调用公有方法是可以正常运行的
  expa.SomePublicFunction()
  // 调用私有方法会直接报错，无法编译
  expa.somePrivateFunction()
}
```

vendor 目录

vendor 目录是用来在 `GOPATH` 和 `GOROOT` 之外保存依赖的，自 Go 1.6 之后，Go 工具链会按照以下顺序来搜索依赖包。

1. 当前包下的 vendor 目录。
2. 上级目录下的 vendor 目录，直至 `$GOPATH/src` 下的 vendor 目录。
3. 在 `GOPATH` 下查找依赖包。
4. 在 `GOROOT` 下查找依赖包。

vendor 目录可以支持不同的目录对相同依赖库的不同版本依赖的支持。例如项目 A 依赖与库 X 的 1.0 版本，项目 B 依赖于库 X 的 1.2 版本，那么在 Go 1.6 之前的环境中，项目 A 和 B 不能实现这种区分版本的依赖需要；在有了 vendor 目录之后，项目可以将自身的依赖下载到自身项目的 vendor 目录中，从而实现对这种区分版本的依赖需要。

在项目开发中，不建议将 vendor 目录放入项目的版本控制中，也不建议在代码各级建立 vendor 目录，一个项目有而且只应该有一个 vendor 目录。

语言基础

Go 程序与其他语言程序一样都是由变量、表达式、语句、函数等基本元素组成的。在编写 Go 语言程序的时候，请牢记 Go 是一门类似于 C 的语言，所以它的特点是面向过程，但又融合了部分面向对象的理念，这就要求在编写程序时要尽可能使用简单的数据结构去实现。

与 C++一样，Go 支持两种格式的注释，行注释以 `//` 开始直到换行符出现，块注释使用 `/* ... */` 包裹。如果块注释只有一行，那么 Go 编译器会将其当做一个空格，但如果块注释占据多行，则会被当做一个换行符处理。在 Go 程序中，要十分重视换行符的存在，因为虽然原则上 Go 程序的语句是需要使用分号 `;` 结尾的，但是实际编写程序时并不使用分号，而是由编译器来决定语句的结尾。分号的问题同样会影响左大括号 `{` 的放置，由于编译器会自动决定语句的结尾，所以代码中的所有左大括号 `{` 都必须跟随 `for`、`if` 等语句在同一行放置。

以下放置是正确的。

```
for i := 0; i < 10; i++ {
  fmt.Println(i)
}
```

以下放置是不能通过编译的。

```
for i := 0; i < 10; i++
{
  fmt.Println(i)
}
```

标识符

Go 支持在变量、常量、函数、自定义类型等命名位置使用的标识符是一个非空的字母或数字串，并且由于 Go 代码文件一律都是 UTF-8 编码，所以也同样支持使用汉字作为标识符。

> 请注意，虽然支持在标识符中使用汉字，但是出于编程习惯考虑，这种使用方法并不推荐。

标识符的命名通常优先选择由实际含义易于阅读和理解的字母和单词组合。在为标识符起名时，一般建议遵循以下建议：

1. 以字母或者下划线开始，由字母、数字和下划线组成。
2. Go 中的标识符是区分大小写的，通常首字母的大小写决定其作用域，首字母大写的成员为导出成员（公有），可以为其他包引用；首字母小写的为私有成员，仅可以在包内使用。
3. 使用驼峰法拼写。
4. 局部变量优先使用短名称。
5. 不要使用保留关键字作为标识符。
6. 尽量避免使用与预定义常量、类型、内置函数重名的标识符。
7. 专有名词全部采用大写形式。
8. `_`（单下划线）是特殊标识符，通常作为占位标识符用于表达式左值，表示忽略或者丢弃指定表达式结果。
9. Go 中的包名需要全部为小写。

Go 中的保留关键字有以下 25 个，在为标识符起名时要注意。

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

变量与常量

变量和常量都是一段或者多段用来存储数据的内存，给变量和常量命名只是为这一段内存起了一个代号，这个代号在编译后的程序中不再起作用。

> 请留意这段叙述，在后面描述指针时，这段叙述将有助于对指针的理解。

Go 是静态类型语言，变量和常量的数据类型都是固定的，并且数据类型规定了变量和常量所使用的内存长度和数据存储格式。对于变量来说，只能修改其值而不能修改其类型，而对于常量，则在完成初始化后一切都不可再修改。

变量使用关键字 `var` 定义，变量类型放置于变量名后，变量可以在声明时初始化值，也可以不赋予初始化值，而由 Go 运行时自动为变量初始化二进制零值。当显式为变量提供初始化值时，可以省略声明变量类型而由运行时自动推断变量类型。变量除了可以使用 `var` 定义以外，还可以使用快捷声明语法，即 `:=` 操作符。快捷语法只能用于定义变量，并且同时显式初始化；不能指定变量的数据类型；只能定义函数级作用域的变量，不能在包级使用。

常量使用关键字 `const` 定义，常量类型的书写位置与变量类型相同，但由于常量一般在定义时就已经完成了值的初始化，所以其类型可以省略。常量没有快捷声明语法。

Go 中的变量和常量的作用域可以是函数级，也可以是包级。其定义和赋值语句格式通常有以下几种。

```
// 声明一个 int 类型变量，由运行时完成初始化
var i int
// 同时声明多个变量
var i int, y float64
// 同时声明多个相同类型的变量可以将其合并书写
var i, j int
// 声明一个 bool 类型变量，变量类型由其初始值确定
var flag = false
// 多变量赋值操作将先计算所有右值，再依次完成对左值的赋值
var x, y = 1, 2
x, y = 3, 5
a, b := 2, 4
// 快捷声明一个 int 类型变量
start := 1
// 快捷声明中可以书写已经定义过的变量，但要保证至少有一个新变量
// 快捷声明中对已经定义过的变量会退化为赋值操作
start, end := 2, 10
```

```go
// 使用 _ 可以丢弃一些值，但赋值表达式左值不能全是 _
_, value := 7, 10
// 多行变量定义可以使用括号合并，即成组声明
var (
  a float64
  x, y = 3, 4
)
// 定义常量时一般不需要书写常量类型
const Red = 0
// 通常采用成组声明的方式声明一组有关联的常量
const (
  Red = 0
  Blue = 1
  Yellow = 2
)
```

编译器会将定义了但未使用的局部变量作为错误，所以在编写程序时要有计划的定义变量。

在许多示例中经常会看到使用预定义标识符 `iota` 来定义一组常量。`iota` 标识连续的无类型整数常量，每次当关键字 `const` 出现时，`iota` 的值就被重置为 0，而其后每个常量声明的值都会在前一个常量基础上加 1。因此上面示例中最后一个常量声明示例可以改写为以下格式。

```go
const (
  Red = iota
  Blue
  Yellow
)
```

除此之外，使用 `iota` 来声明常量的值，还可以使用表达式的方式来定义这一组常量的值的初始化方式。例如以下示例。

```go
// const 和 iota 的组合可以隐式设置常量初始化规则
const (
  Active = 1 << iota    // 值为1，即 1 << 0
  Send                  // 值为2，即 1 << 1
  Receive               // 值为4，即 1 << 2
)
```

`iota` 的初始化规则可以被普通赋值表达式打断，一旦打断后，就需要重新使用 `iota` 进行赋值，否则未赋值常量将延续上一个常量的值。

```go
const (
  a = iota  // 值为0
  b         // 值为1
  c = 100   // 值为100，打断 iota 的初始化规则
  d         // 值为100，延续上一个常量的值
  e = iota  // 值为4，恢复 iota 自增，但是包括了 c 和 d
  f         // 值为5
)
```

由于 Go 中没有枚举类型，所以通常需要枚举的场景，都使用一组常量来定义。

内置类型

Go 定义了丰富的内置类型可供使用，在跨平台编程时这些类型的表现是相同的，无需过多考虑兼容性。

类型标识	类型名称	长度	初始化值	备注
`bool`	布尔	1	`false`	
`byte`	字节整型	1	`0`	即 `uint8`
`int`、`uint`	整型	4 或 8	`0`	根据目标平台架构确定
`int8`、`uint8`	8 位整型	1	`0`	
`int16`、`uint16`	16 位整型	2	`0`	
`int32`、`uint32`	32 位整型	4	`0`	
`int64`、`uint64`	64 位整型	8	`0`	
`float32`	32 位浮点型	4	`0.0`	
`float64`	64 位浮点型	8	`0.0`	
`complex64`	64 位复数型	8		实部与虚部都是 `float32`
`complex128`	128 位复数型	16		实部与虚部都是 `float64`
`rune`	字符型	4	`0`	用于表示 Unicode 字符，底层为 `int32`
`uintptr`	指针型	4 或 8	`0`	用于存储指针，底层为 `uint`
`string`	字符型		`""`	默认值为空字符串，而不是 NULL
`array`	数组型			
`struct`	结构型			
`func`	函数		`nil`	
`interface`	接口		`nil`	
`map`	字典型		`nil`	引用类型
slice	切片型		`nil`	引用类型
`chan`	通道型		`nil`	引用类型

运算符

运算符在任何编程语言中都用来形成表达式，对变量和常量进行各种运算操作。不同的数据类型往往支持使用不同的运算符来进行操作。

在 Go 中，数值类型可以使用以下算术运算符。

语法	功能
`+x`	即 x 本身
`-x`	x 的负数
`x++`	x 自增
`x--`	x 自减
`x += y`	将 x 加上 y
`x -= y`	将 x 减去 y
`x *= y`	将 x 乘以 y
`x /= y`	将 x 除以 y，如果 x 与 y 均是整型则丢弃余数
`x + y`	返回 x 与 y 的和
`x - y`	返回 x 与 y 的差
`x * y`	返回 x 与 y 的积
`x / y`	返回 x 与 y 的商，如果 x 与 y 均是整型则丢弃余数

> 虽然自增与自减在这个表格里出现，但自增与自减在 Go 中不再是运算符，只能作为独立语句使用，并且不能前置。
>
> 始终要牢记，表达式通常会返回一个值，而语句是完成一个行为动作。表达式可以作为语句使用，但语句不能被当做表达式。

整型数值类型还支持以下运算符来进行操作。

语法	功能
`^x`	按位取反，`^ 0111 = 1000`
`x %= y`	将 x 设置为 x 除以 y 的余数
`x &= y`	将 x 设置为 x 和 y 按位与（AND）的结果
`x |= y`	将 x 设置为 x 和 y 按位或（OR）的结果
`x ^= y`	将 x 设置为 x 和 y 按位异或（XOR）的结果
`x &^= y`	将 x 设置为 x 和 y 按位与非（ANDNOT，按位清除）的结果
`x >>= u`	将 x 设置为 x 右移 u 个位的结果
`x <<= u`	将 x 设置为 x 左移 u 个位的结果
`x % y`	返回 x 与 y 的余数
`x & y`	返回 x 与 y 按位与的结果，`0101 & 0011 = 0001`
`x | y`	返回 x 与 y 按位或的结果，`0101 | 0011 = 0111`
`x ^ y`	返回 x 与 y 按位异或的结果，`0101 ^ 0011 = 0110`
`x &^ y`	返回 x 与 y 按位与非的结果，`0110 &^ 1011 = 0100`
`x << u`	返回 x 左移 u 个位的结果，`0001 << 3 = 1000`
`x >> u`	返回 x 右移 u 个位的结果，`1010 >> 2 = 0010`

Go 支持标准的逻辑操作和比较操作，布尔表达式会返回一个布尔值，通常会用在分支结构中。

语法	真值条件	功能
`!b`	b 为假	逻辑非
`a || b`	a 与 b 有一项为真	逻辑或，当表达式 a 的值为真，则 b 不再判断
`a && b`	a 与 b 同时为真	逻辑与
`x < y`	x 小于 y	小于
`x <= y`	x 小于等于 y	小于等于
`x == y`	x 与 y 相等	等于
`x != y`	x 与 y 不相等	不等
`x >= y`	x 大于等于 y	大于等于
`x > y`	x 大于 y	大于

虽然 Go 为所有的运算符提供了优先级规则，但本指南不准备列出所有运算符的优先级列表，而是推荐使用括号 `()` 来保证运算优先级的清晰，以免出现难以发现的错误。

数值类型

数值类型是处理数据的基础，Go 提供了 11 种整型，2 种精度的浮点类型和 2 种精度的复数类型。11 种整型中包括 5 种有符号整型、5 种无符号整型和 1 种用于存储指针的整型。其中字符类型 `rune` 保存的是字符的 Unicode 码点，但实际上是 `int32` 类型的别名。另外一个别名是 `byte`，这个用于表示数据原始字节的类型是 `uint8` 的别名。 在一般的操作情况下，可以不必要使用指定长度的整型，直接使用 `int` 类型即可， `int` 的长度根据平台的架构确定。

Go 中的浮点数使用 IEEE-754 格式表示，根据 IEEE-754 标准，浮点数在内存中分为符号位、指数位和尾数位三部分。以 64 位的双精度浮点数为例，符号位为 1 位，指数位为 11 位，尾数位为 52 位。在 IEEE-754 标准下，部分浮点数可以精确表示，例如 0.5，而其他浮点数则只能近似表示，如 0.1。这种浮点数问题不是 Go 语言所特有的，而是所有主流语言的浮点数表示都面临的。但是 Go 针对浮点数的这个特点，采用了比较智能的算法来输出浮点数，以尽可能的确保准确性。

> 尽管 Go 已经做了很多的工作，但是在实际应用中，对浮点数进行相等或不等的比较时要十分小心。

对于一些需要精确计算或者超出 `int64` 类型可容纳长度的数值的计算，Go 在标准库 `math/big` 中提供了 `Int`、`Float`、`Rat` 三种类型来处理大整型、大浮点型和分数计算。其中尤其是 `Rat` 类型，在财务和金融计算中是非常有用的。

复数类型是 Go 内置支持的数据类型，通常可以使用 `complex()` 函数创建，对于一个复数可以使用 `real()` 函数来获取其实部的值，用 `imag()` 获取其虚部的值。复数可以使用绝大部分算数运算符以及相等和不等比较运算符。

Go 标准库 `math` 提供了大量数学运算用的函数，这些函数默认是使用 `float64` 来进行处理的，标准库 `math/cmplx` 提供了复数数学运算功能，默认都是使用 `complex128` 来处理的。

字符串

对字符串的处理是目前大部分应用程序的日常任务，Go 也不例外。Go 的字符串是使用 UTF-8 编码的一个任意字节的常量序列，这意味着 Go 的字符串可以包含世界上任意语言而不受代码页限制。与 C/C++和 Java 的定宽字符序列不同，Go 采用变宽字符序列，每一个字符都使用一个或者多个字节表示。采用变宽字节就表示 Go 的字符串无法像 C/C++中一样进行字节索引操作，但 Go 支持逐字符迭代，并提供了包含大量字符串搜索和操作的标准库，而且 Go 的字符串还可以转换成 Unicode 码点切片来对其中的字符进行索引。

> 字符集一般是使用一张或者多张二维表来表示，每张二维表中行与列的交点称为一个码点。每个码点唯一的对应一个字符，码点值一般就是其对应字符的编号。Unicode 码点最初使用两个字节的十六进制表示，随着 Unicode 字符集的扩大，码点也随之扩大到使用三个或以上的字节。

Go 字符串使用双引号 `""` 或者反引号 ``` `` ``` 创建，例如：`sampleStr := "Hello world"`。双引号主要用来创建可解析的字面量，支持在字符串中使用转义，但不支持定义多行文本；反引号则正好相反，支持定义多行文本，但不支持使用转义字符。单一的字符可以使用一个单一的 `rune` 或者 `int32` 来表示，字面量使用 `''` 来创建。

转义字符

Go 字符串中可以使用的转义字符可参见下表。

转义字符	含义
`\\`	反斜线
`\ooo`	使用 3 位八进制数表示 Unicode 字符
`\'`	单引号
`\"`	双引号
`\a`	ASCII 响铃符
`\b`	ASCII 退格符
`\f`	ASCII 换页符
`\n`	ASCII 换行符
`\r`	ASCII 回车符
`\t`	ASCII 制表符
`\uhhhh`	使用 4 位十六进制数表示 Unicode 字符
`\Uhhhhhhhh`	使用 8 位十六进制数表示 Unicode 字符
`\v`	ASCII 垂直制表符
`\xhh`	使用 2 位十六进制数表示 Unicode 字符

常用操作

虽然 Go 的字符串是不可变的，但是其支持一系列编辑操作，所有的编辑操作都将返回新的字符串。常用的字符串操作主要有以下这些。

操作语法	功能
`s += t`	将字符串 t 追加到字符串 s 的末尾
`s + t`	将字符串 s 与字符串 t 拼接，返回拼接后的字符串
`s[n]`	获取字符串 s 中索引位置为 n（n 为 `unit8` 类型）处的原始字节
`s[n:m]`	取得从位置 n 到位置 m-1 处的字符串
`s[n:]`	取得从位置 n 开始的字符串
`s[:m]`	取得从位置 0 到位置 m-1 处的字符串
`len(s)`	获得字符串 s 的字节数
`len([]rune(s))`	获得字符串 s 中的字符个数，可以使用 `utf8.RuneCountInString()` 代替
`[]rune(s)`	将字符串 s 转换成一个 Unicode 码点数组
`string(chars)`	将一个 `[]rune` 或者 `[]int32` 转换成字符串
`[]byte(s)`	将字符串 s 转换成一个原始字节切片数组，但不保证得到的字节是合法的 Unicode 码点
`string(bytes)`	将 `[]byte` （字节数组）或者 `[]uint8` 转换成一个字符串类型，但不保证转换的字节数组是合法的 UTF-8 编码字节
`string(i)`	将任意整数类型 i 转换成字符串，会自动判断其为 Unicode 码点或者 ASCII 码
`strconv.Itoa(i)`	将整型 i 转换为字符串，同时返回一个错误值，如果成功转换，错误值返回 `nil`
`fmt.Sprint(x)`	返回任意类型 x 的字符串表示

字符串比较

字符串既然是由 Unicode 码点组成的，又可以转换为 `[]byte` 或者 `[]rune`，那就可以对其进行比较操作。字符串比较操作可以使用 `<`、`<=`、`==`、`!=`、`>`、`>=` 运算符来完成，这些运算符会在内存中逐字节的比较两个字符串。

对字符串进行比较时，需要注意以下几个问题，这些问题是所有使用 Unicode 字符串的语言都存在的。

1. 有些 Unicode 编码使用两个或更多的字节序列来表示，这通常会使得比较结果不会按照字面量的期

望得到。Go 中所有的 UTF-8 字节序列都是采用相同的码点到字节的映射，所以只有字符串来自 Go 程序以外的时候，才可能会出现问题。
2. 编程时可能会希望把不同的字符看做相同的字符。这虽然不能被字符串比较操作直接支持，但可以通过标准化操作来实现。
3. 字符的排序是语言相关的。

字符串索引与切片

前面常用操作中列举了字符串索引与切片语法，Go 默认的索引和切片语法在应用到 ASCII 编码字符串时，能够很好的按照预想来运行，但是应用到 Unicode 编码的多字节字符串上，将面临更大的挑战。这种挑战主要来自于字符边界的确定。

当不需要对一个字符串进行切片时，可以使用 `for ... range` 来逐字符迭代字符串，但是需要按照字符边界进行切片得到索引位置时，需要使用 `strings` 包中的函数 `strings.Index()` 或者 `strings.LastIndex()`。当使用索引获取字符串中的字符时，需要注意索引只能按照字节来获取字符串中的内容。例如有字符串 `s`，直接使用 `s[n]` 获取其中的字符是按照 ASCII 编码来获取的，对于使用 Unicode 编码的多字节字符，将会导致 Unicode 字符被拆分。这也就表示对于任意字符串，通过索引来提取字符通常是不可靠的。相比使用索引提取字符，使用切片来返回一个字符串是更安全的提取方法，而索引位置可以使用 `strings` 包中的函数来确定。

```
line := "the quick brown fox jumps over the lazy dog"
i := strings.Index(line, " ")     // 获得第一个空格的索引位置
firstWord := line[:i]             // 从第一个字符切片到第一个空格
```

而使用 `strings`、`utf8` 等字符串处理包中的函数可以更加方便和安全的操作 Unicode 编码的字符串，具体常用方法将在后文给出。

数组与切片

在前面字符串一节中实际上已经见识过 Go 的数组和切片了，例如：`[]rune`。数组和切片虽然看起来相似，但是其功能和原理都不尽相同。Go 的数组是一个元素类型都相同的定长序列，相同元素类型但不同长度的数组不是同一个类型。

数组在声明的时候必须指定长度，可以使用以下语法格式来创建。

```
var arr [N]Type
// 使用大括号对数组中的元素进行初始化
// 当给定的元素值不足以填充数组长度时，剩余的元素将被使用零值初始化
var arr = [N]Type{value1, value2, ..., valueN}
// 方括号中的 ... 表示数组的长度通过初始化元素数量推断
var arr = [...]Type{value1, value2, ..., valueN}
```

与其他的语言不同，Go 在定义数组时，代表数组的 `[]` 要放置在数组元素类型前。二维数组需要用两组 `[]`，例如 `[8][9]int` 表示定义了一个 8×9 的二维数组，多维数组可以看做数组的数组。

数组元素可以通过 `[]` 来进行索引，例如 `arr[1]`。其首元素为 `arr[0]`，末元素为 `arr[len(arr)-1]`。数组虽然是定长的，但是其中元素的值是可以更改的，要更改元素的值只需要对指定索引进行赋值即可，例如 `arr[3] = 1`。数组的长度可以使用 `len()` 获得，由于数组长度固定，所以使用`cap()` 获取其容量与使用 `len()` 获取的长度是一样的。数组可以使用切片操作来进行处理，但返回的结果将是一个切片而不是数组。

切片是 Go 中一个比数组更加灵活、方便、强大的数据类型，切片是元素类型都相同的可变长序列。切片在创建时不需要指定其长度，可以使用以下语法格式来创建。

```
var arr = make([]Type, 长度, 容量)
var arr = make([]Type, 长度)
// 以下两个创建空切片的语法等价
var arr = []Type{}
var arr = make([]Type, 0)
var arr = []Type{value1, value2, ..., valueN}
```

内置函数 `make()` 通常用来创建切片（slice）、字典（map）和通道（channel）。当切片被创建时，运行时会创建一个隐藏的数组，并将其引用给切片；当切片发生长度变化时，运行时会重新创建新的数组。如果在创建切片时指定了容量，那么切片的长度将不能超过所指定的容量。对于切片内元素的访问与数组是一致的。

切片常用的操作有以下这些。

操作	功能
`s[n]`	取得切片中索引位置为 `n` 的元素
`s[n:m]`	用从索引位置 `n` 到 `m-1` 的元素形成新切片
`s[n:]`	用从索引位置 `n` 到 `len(s)-1` 的元素形成新切片
`s[:m]`	用从索引位置 `0` 到 `m-1` 的元素形成新切片
`s[:]`	用从索引位置 `0` 到 `len(s)-1` 的元素形成新切片
`cap(s)`	返回切片的容量，$cap(s) \geq len(s)$ 始终成立
`len(s)`	返回切片中包含元素的数量
`s[:cap(s)]`	增加切片的长度到其容量
`append(s, ...items)`	向切片中添加内容，并返回新切片

`append()` 是一个神奇的函数，可以完成切片中常用的编辑操作。具体可见以下示例。

1. `s = append(s, v1, v2, v3)`，向切片中添加单一的值。
2. `s = append(s, t...)`，向切片中添加另一个切片 `t` 的所有值，操作符 `...` 表示展开。
3. `s = append(s, t[1:4]...)`，向切片中添加另一个切片 `t` 中的子切片。
4. `s = append(s[:4], t[4:]...)`，将两个切片的不同片段合并成一个新的切片。

5. `s = append(s[:1], s[3:]...)`，从切片中删除 `s[1:3]` 的元素。

在 Go 标准库中，所有的公共函数都是采用切片而不是数组，所以在没有特殊要求下应该尽量使用切片来代替数组。对于切片中元素的排序和搜索，是标准库中 `sort` 包提供的，具体常用方法将在后文中给出。

字典

字典在现代语言中是一种十分常用的数据结构，其主要用于保存无序的键值对组合，在其他的 Go 教程中也常被称为映射，本指南为了方便与其他语言共通理解，采用了字典的概念。在字典中，所有的键都是唯一的，并且必须可以使用 `==` 和 `!=` 进行比较。根据字典中键的要求，Go 中的基本类型，如 `int`、`float64`、`string` 等，包括后文将提到的结构体，都可以用作字典的键。对于字典的值则没有过多的要求，基本上任意类型都可以作为字典的值。字典的容量只受到内存限制，所以结合不同的数据类型，可以组合出复杂的数据结构。

字典可以使用以下语法格式创建。

```
var dict = make(map[KeyType]ValueType, 初始容量)
var dict = make(map[KeyType]ValueType)
// 直接使用 map 搭配初始化语法可以创建空集
var dict = map[KeyType]ValueType{}
var dict = map[KeyType]ValueType{key: value}
```

类型 `map` 用来标记字典类型，`[]` 中声明键的类型，紧随着 `[]` 声明值的类型，就形成了字典类型的声明格式 `map[键类型]值类型`。

字典也使用 `[]` 进行元素的访问，但其中书写的不是索引而是键值，并且在使用时也与数组和切片有较大的区别。字典常用的操作主要有以下这些。

操作语法	功能
`m[k] = v`	将 `v` 以键 `k` 保存到字典，如果字典中已经存在键 `k` 则覆盖其原来的值
`delete(m, k)`	从字典中删除键 `k` 及其对应的值，若 `k` 不存在则不作操作
`v := m[k]`	从字典中提取键 `k` 对应的值，如果 `k` 不存在则返回零值
`v, found := m[k]`	从字典中提取键 `k` 对应的值，`found` 表示所请求的 `k` 是否存在
`len(m)`	返回字典中键值对的数量

指针、值类型和引用类型

在所有的编程语言中都始终存在两个概念：值类型和引用类型。

值类型在传递给函数或者方法时，会被复制，这通常都发生在占用内存数量可以被明显预见的类型中。在 Go 中，布尔、整型、浮点、复数、字符串、数组等都是值类型。值类型的值传入函数或者方法的是值类型值的副本，对这个副本的任何修改，都不会影响到副本的来源本体。当函数或者方法结束其生命周期之后，传入函数或方法的值类型值副本将会被垃圾回收（GC）。值类型值在占用内存较小时，进行

复制传递是非常高效的，但如果传递一个元素是复杂结构体的数组，那么就会面临巨大的性能代价。

从基础概念上说，变量就是一个内存块的名字，每个变量对应着一个内存块的地址。变量所对应的内存地址中一般保存的是实际数据，即某种数据类型的实际数值。那如果一个变量所对应的内存地址中保存的是另一个变量的内存地址呢?

前面提到占用字节较少的值类型在传入函数和方法时，系统的开销非常小。那么为了减轻传递占用大量内存的复杂数据结构，能不能只将其内存地址复制传入函数和方法使用呢？答案是可以的。对于字典、切片、通道、结构体这类能够存放复杂数据的类型，Go 会采用传递引用的方式将其传入函数和方法。

要解释引用类型，首先要了解在 Go 中如何获得变量的内存地址和如何通过内存地址获得内存内容。在 Go 中，操作符 `&` 具有多种功能，当其放置在变量前时，它返回的是变量的内存地址，例如 `&a`。操作符 `&` 返回的地址可以被保存在一个指针变量中，也就是前面所说的保存另一个变量内存地址的变量，我们称这个指针变量指向了另一个变量或者指向了一块内存。与 `&` 操作符一样有多种功能的是 `*` 操作符。`*` 操作符可以声明指向指定类型变量的指针类型，例如 `var p *int` 将声明一个指向 `int` 类型变量或者保存 `int` 类型数据的内存地址的指针变量 `p`。要取得指针变量所指向内存位置的内容，同样需要 `*` 操作符，此时 `*` 的操作称为解引用，即 `*p`。由于指针直接指向了一个内存区块，所以对指针进行的操作将直接修改被指向的变量内容。

可以尝试一下以下示例的运行效果来了解指针的基本使用和特性。

```
package main

import "fmt"

func main() {
  var n int = 8
  var p *int
  p = &n
  *p++
  fmt.Println(n)
}
```

指针也可以指向另一个指针，这在声明的时候只需要嵌套 `*` 操作符即可，例如 `var p **int`，而解引用时也需要嵌套 `*` 操作符，例如 `**p`。

> Go 的指针与 C/C++中的指针不同，Go 的指针只能对引用的内存进行操作，而不能对指针变量中保存的内存地址进行运算操作。例如一个指向 `int` 类型的指针 `p`，在 C/C++中可以执行 `*(p++)` 的操作（通常用来遍历数组），但 Go 就不被允许。所以这就是为什么 Go 中的指针不能被用来遍历数组以及字符串。
>
> 此外，Go 支持从函数或者方法中返回局部变量的内存地址，这会使函数外部的指针变量指向被返回的内部变量，按照传统作用域管理的概念，这种指向会形成悬垂指针（无效指向的指针）而造成内存泄漏，但在 Go 中，这种使用方法会使局部变量在内存中保留下来直到不再使用为止。这种用法在 Go 标准库中非常常见。

指针变量的初始化零值为 `nil`，与其他语言中的 `NULL` 的含义类似，表示指针指向虚空。

> 指针变量中保存的内存地址不会是 0，也就是说指针不会指向编号为 0 的内存地址，这个地址是系统保留内存，对这个地址进行操作会导致系统崩溃。

在了解了指针以后，再弄懂引用类型就不难了。引用类型在传入函数和方法时，不会进行复制，而是将其内存地址传入。但是由于 Go 的内部支持，在函数和方法内部，对于引用类型还是按照普通变量来使用，而不是需要像指针一样需要解引用。在 Go 中，切片、字典、通道、结构体等都是引用类型，所以在使用这些引用类型作为函数和方法的参数的时候要尤其注意。由于引用类型值在函数和方法外部和内部都是相同的内存区块，所以在函数和方法内部对引用类型值进行了修改以后，修改结果会传递到函数和方法以外。

不要想当然的认为值类型值一定被分配在栈内存中，引用类型值一定被分配在堆内存中。Go 的编译器和运行时十分聪明，会自行选择对程序运行有利的分配方法，所以本指南中不会提示或者暗示栈内存或者堆内存的使用。

类型转换

类型转换是一种在相互兼容的不同类型之间相互转换的操作，非数值类型之间的转换不会丢失精度，但数值类型之间的转换，则根据转换方向的不同，会发生精度丢失现象。在 Go 中，转换类型可以采用以下语法格式。

```
新变量名 := 目标类型名(待转换变量)
```

除常量、别名类型与未命名类型以外，Go 要求所有的类型转换都是显式的。所以不要期望 Go 能够提供隐式类型转换供使用。这里所提到的 **未命名类型** 是指数组、切片、字典、通道等虽然有明确类型标识符，但实际类型与其长度、元素类型相关的类型，这些类型在使用时不必像 `int` 、`float64` 等一样提供具体的类型命名，所以称之为 **未命名类型** 。未命名类型可以通过自定义类型语法为其命名。

自定义类型

与 C/C++一样，Go 也允许自定义类型。自定义类型使用关键字 `type` 定义。类型自定义的语法格式为：`type 自定义类型名 原始类型名`。与 `var` 和 `const` 一样，多个自定义类型也可以合并到一个 `()` 中，例如：

```
type (
  Age int
  Name string
)
```

自定义类型虽然与原始类型拥有相同的底层结构，但两者属于完全不同的两种类型。自定义类型除了会继承原始类型的操作符以外，自定义类型不会继承原始类型的所有其他信息，所以自定义类型不是原始类型的别名，而不能进行隐式类型转换和比较。

分支结构

分支结构是进行逻辑判断的基础，Go 提供了三种分支语句：`if`、`switch` 和 `select`。其中 `if` 和 `switch` 通常用于条件判断，`select` 则多用于并发。这里将仅介绍 `if` 和 `switch` 语句，`select` 语句将在并发通信部分介绍。

`if` 语句

`if` 语句的语法很简单，可按照以下语法格式书写。

```
if 域内赋值语句; 条件表达式 {
  语句块
} else if 域内赋值语句; 条件表达式 {
  语句块
} else {
  语句块
}
```

从 `if` 的语法格式可以看出，`if` 的条件表达式不需要使用 `()`，并且 `{}` 是强制必需的。与其他语言不同的一点是，`if` 可以使用一个域内赋值语句来初始化一个仅可以在 `if` 语句块内使用的变量。例如以下示例。

```
if a := someOperate(); a < 0 {
  fmt.Printf("(%d)\n", -a)
} else {
  fmt.Println(a)
}
```

这种用法在异常处理和某些元素的存在性判断中比较常见。

> `if` 是一个语句，不是表达式，不能返回任何值。

`switch` 语句

Go 中的 `switch` 有两种类型：表达式 `switch` 和类型 `switch`。其中类型 `switch` 是 Go 专有的，并且所有类型的 `switch` 语句都无法向下贯穿，也就是不需要在每个分支中使用 `break`，相反，需要贯穿的位置需要使用关键字 `fallthrough` 明确指示。

表达式 `switch` 的语法格式如下：

```
switch 域内赋值语句; 表达式 {
case 表达式列表 : 语句块
default: 语句块
}
```

与 `if` 语句一样，`switch` 语句也支持域内赋值语句，在此处完成初始化的变量只能够在 `switch` 语句的域内使用。用于进行判断的表达式是可选的，这样在 `case` 的表达式列表中就需要使用布尔表达式进行真值判断。例如以下函数：

```go
func max(a, b int) int {
  switch {
    case a < b:
      return b
    case a >= b:
      return a
  }
}
```

使用进行判断的表达式将使 `switch` 与 `case` 后的表达式进行相等判断，并执行满足条件的 `case` 后的语句。

类型 `switch` 的语法格式与表达式 `switch` 的语法格式相同，但必须要提供用于判断的表达式，`switch` 将自动对其类型进行判断。例如以下函数：

```go
func classifier(a interface{}) {
  switch a.(type) {     // 这是类型 switch 获取类型的固定写法
  case int:
    fmt.Println("is int")
  case bool:
    fmt.Println("is bool")
  case nil:
    fmt.Println("is nil")
  default:
    fmt.Println("unkown")
  }
}
```

循环结构

Go 提供的循环结构只有 `for` 语句，但是 Go 中的 `for` 语句功能强大，通过与不同格式表达式的组合，`for` 可以实现多种操作。以下通过其语法格式来说明 `for` 能够完成的功能。

```go
// 无限循环
for { }
// 有条件循环，即 while 循环
for 终止条件表达式 { }
// 传统 for 循环，其中后置表达式在每次循环结束都会执行
for 初始表达式; 终止条件表达式; 后置表达式 { }
// 逐字符迭代字符串
for index, char := range 字符串 { }
// 迭代数组或者切片
for index, item := range 数组或切片 { }
// 迭代字典
for key, value := range 字典 { }
// 迭代通道
for item := range 通道 { }
```

在迭代字符串、数组、切片和字典时，一定要注意 `range` 关键字会返回两个值，第一个值为索引或者键，第二个值才是元素的值。如果使用 `for v := range a { }` 的格式来迭代字符串、数组等内容，变量 `v` 中存储的内容将是索引或者键，而不是元素的值。

`for` 语句可以使用 `break` 和 `continue` 来控制循环进行，这两个关键字的行为与其他语言中的行为基本一致。但是 Go 为其增加了更加强大的跳转能力：按标签跳转。标签的定义，允许 Go 在进行嵌套循环时，在整个循环体中的任意位置跳出任意一层循环。例如：

```
// 标签使用 标识符: 的格式定义
OUTER:
for row := range table {
  for column := range row {
    // 执行一些逻辑
    break OUTER // 这将直接退出嵌套的两层循环
  }
}
```

标签可以用在 `for`、`switch`、`select` 中，并且 `break` 和 `continue` 都可以使用标签。

> 对于标签的使用，Go 还支持 `goto` 语句，但这种随意跳转并不受到推荐。

函数

函数是面向过程编程的根本，Go 的函数融合了许多语言的特性。函数使用关键字 `func` 定义，接受一个参数列表，可以返回一个或者多个值。函数的定义语法格式如下。

```
func 函数标识符(参数列表) 可选返回值类型 {
  函数体
}
// 如果需要返回多个值，则需要使用括号来定义返回值类型
func 函数标识符(参数列表) (返回值类型) {
  函数体
}
// 还可以对返回值进行命名
func 函数标识符(参数列表) (返回值名称 返回值类型, 返回值名称 返回值类型) {
  函数体
}
// 利用匿名函数和变量定义来声明函数
var 函数标识符 = func(参数列表) (返回值名称 返回值类型) {
  函数体
}
```

相比其他语言中的函数，Go 的函数拥有以下特点：

1. 支持不定长参数。
2. 支持多返回值和命名返回值。
3. 支持匿名函数和闭包。
4. 不支持默认参数。

5. 不支持函数重载。
6. 不支持函数嵌套。但可以使用匿名函数来作为替代实现。
7. 不需要声明函数原型。Go 编译器会自动扫描所有文件，对函数出现的顺序没有要求。
8. Go 的第一类对象，可在运行期创建，可以用作函数的参数或者返回值，并且可以存入变量。

在 Go 中，函数也是一个类型，具备相同签名（参数列表和返回值列表）的函数视作同一类型。函数的类型一般如下例中所示。

```
type FuncType func(a, b int) (c int, err error)
```

函数的调用方式是直接使用函数标识符，格式为 `函数标识符(传入函数的参数)`，例如：`Add(a, b)`。函数通常用在赋值表达式的右值，其返回值将赋予左值。

函数标识符命名原则

函数标识符的命名规则与普通标识符的命名规则相同，但是由于函数一般是一组操作的集合用于实现固定的目的，所以在命名时推荐采用以下命名原则。

1. 精悍简短，望文知意。精确描述用途，避免歧义。
2. 有动词或介词与名词组合而成，例如：`generateSample`、`searchInSamples`。
3. 除了有特定含义的缩写以外，避免使用缩写。
4. 避免使用类型关键字，例如 `struct`、`string` 等。
5. 不使用仅有大小写区别的同名函数。
6. 避免与内置函数同名。
7. 尽量不使用数字作为函数标识符的组成部分。
8. 使用驼峰法命名风格。
9. 使用习惯用语表示固定动作和行为。例如 `init`、`is`、`has`、`get`、`set` 等。

参数定义与使用

Go 的函数不支持默认参数，也就不支持可选参数，也同样不支持命名参数，所以在调用时，必须严格按照函数定义时的参数声明顺序传递指定类型和数量的实参。函数的参数可以被认为是函数的局部变量，所以在函数内声明同名变量，将使得函数的参数被遮蔽（Shadowed）。

> 这里提到了一个名词：**实参**，实参是指传递给函数的实际参数。相对应的，函数在定义时使用的参数，被称作 **形参**，即形式参数。函数的形参类似于变量，而实参则可以是多种形式，例如常量、变量、表达式等。

函数形参在定义时，如果相邻的形参类型相同，则可以合并书写，这与 Go 中变量、常量等声明的习惯是相同的。例如。

```
func Add(a int, b int) int { }
// 可以简写为
func Add(a, b int) int { }
```

所有传递给函数的参数，都是采用值拷贝的形式传入函数的，其中值类型参数是通过复制其实际值传入，而引用类型参数则是通过复制其指针传入。在调用函数时，运行时会为形参和返回值分配内存并将实参复制进来。

在函数中使用指针参数，可以在一定程度上提升程序性能，但是由于指针的使用会延长被引用对象的生命周期，所以可能会造成内存消耗的暴涨，提升垃圾回收的成本。所以选择使用何种方式传递参数，需要根据所传递的参数的大小与复杂程度来确定。

利用指针可以实现出参（Output parameter），由于指针类型传递的是被引用对象的内存地址，所以使用指针类型作为参数，并在函数中对其进行修改，可以使函数内部的操作结果传出到函数外。但是在实际编码中并不推荐这种用法。

虽然 Go 的函数不能使用默认参数和可选参数，但是 Go 的函数提供了可变参数的支持。可变参数实质上是一个切片，其中包含一个或多个同类型的参数。可变参数必须放置在参数列表尾部，并且它会捕获函数调用时所有额外提供的实参。可变参数的定义采用 `...` 展开操作符，定义格式为 `func 函数标识符(参数标识符 ...可变参数类型)`，具体可参考以下示例。

```
func Add(a ...int) int {
  var sum int
  for _, v := range a {
    sum += a
  }
  return sum
}
```

可变参数在函数中实际上是一个切片，但传入函数的实参却不能是一个切片。如果实参是一个切片，则必须要先展开，如果实参是数组，则必须先转换为切片再展开。例如以上示例的应用。

```
func main() {
  num1 := []int{1, 2, 3}
  num2 := [3]int{1, 2, 3}      // 注意 num2 在定义时与 num1 的区别，num2 是一个数组
  sum1 := Add(num1...)         // ... 操作符用于将切片展开
  sum2 := Add(num2[:]...)      // num2 需要先转换为一个切片
}
```

返回值

当函数明确定义有返回值时，必须要使用关键字 `return` 输出返回值，除非函数中出现 `panic` 或者死循环。Go 中的函数支持多返回值，多返回值函数的定义方法在前面章节中已经明确说明，函数在返回多个返回值时，返回值类型列表需要与定义时的返回值类型列表相同，并且不能有遗漏的返回值。

以下是一个 Go 中常见的一个多返回值函数。

```
func Div(x, y int) (int, error) {
  if y == 0 {
    return 0, errors.New("divided by zero")
  }
  return x / y, nil
}

func log(r int, e error) {
```

```
  // 省略这个函数的定义，这里只需要函数参数列表
}

func main() {
  f, err := Div(1, 0)
  if err != nil {
    fmt.Println(err)
  }
  g, _ := Div(8, 4)
  log(Div(7, 3))
}
```

由于 Go 中没有元组，所以多返回值只需要用逗号隔开即可。示例中返回了一个 `error` 类型的返回值，这种用法在 Go 中非常常见。当返回的 `error` 是一个 `nil` 值时，说明函数执行顺利没有出现错误；但是如果返回的 `error` 不是 `nil` 值时，则需要进行错误处理。

> 这是 Go 被诟病为开火后需要检查的发令枪的原因，虽然这种错误检查十分繁琐，但是对于程序处理错误非常有效，并且节省了其他语言中出现异常时需要栈展开的资源和时间。在适应这种编程思想后，你会发现要是想编写一个会因为异常而中止的程序，必须得忽略这些返回的错误。

多个返回值在接收的时候需要逐一接收，不能使用数组或者切片来接收，如果其中某个返回值不想要，可以使用 `_` 丢弃。多返回值的函数调用可以直接作为其他函数的实参。具体可参考上面的示例。

在返回值中可以返回一个定义在函数内部的指针，一般说来当函数内部定义的变量在超出函数作用域后就会被垃圾回收。但是在函数中定义的指针如果传出到了函数外，则会被运行时标记为逃逸，从而延长其生命周期，提升其作用域，直到其完全不再被使用为止。在后面的结构体章节中，常常利用这个特性来定义结构体的构造函数。

返回值是可以命名的。命名返回值可以当做局部变量使用，在返回时直接使用空白的 `return` 即可。例如前面的示例可以改写为以下形式。

```
func Div(x, y int) (z int, e error) {
  if y == 0 {
    e = errors.New("divided by zero")
    return
  }
  z = x / y
  return
}
```

在使用命名返回值时需要注意，这些可以被当做局部变量的命名返回值会被其他作用域中的同名变量遮蔽。而且，一个函数的返回值如果使用命名返回值，则它的所有返回值都必须命名，不能存在混用的情况。在编码过程中，推荐尽量使用明确的类型去标识返回值用途，而不使用命名返回值。

闭包

闭包（Closure）就是一个函数，但与其他函数不同的是这个函数捕获了与它位于同一作用域的其他变量和常量。在 Go 中，所有的匿名函数都是闭包。这也就是说，闭包的创建方法与普通函数一致，只是没有函数标识符。

闭包常常用来作为函数参数传递给其他函数供调用实现更高级的功能。还可以用在工厂函数中，根据传入的参数构造并返回一个函数。例如以下示例。

```
func MakeWeight(w int) func(int, int) int {
  return func(x, y int) int {
    return (x + y) * w
  }
}

func main() {
  weight3 := MakeWeight(3)
  result := weight3(4, 5)
}
```

以上示例中的工厂函数 `MakeWeight` 会根据给定的参数 `w` 的值不同，返回可以执行不同功能的函数。这就像是一个制造函数的工厂，所以称它为工厂函数。工厂函数返回的函数可以保存在变量中以待使用。

当了解了闭包之后，就可以引入另一个概念：**高阶函数**。高阶函数实际上就是一个将其他函数作为自己参数，并在函数体中调用它们的函数。高阶函数通常可以通过传入不同的参数来改变其行为，达到不同需求的目的。例如以下常见的过滤函数。

```
func First(arr []int, predicate func(int) bool) int {
  for _, v := range arr {
    if predicate(v) {
      return v
    }
  }
  return -1
}

func main() {
  a := []int{1, 2, 3, 4, 5, 6}
  firstGt4 := First(a, func(x int) bool { return x > 4 })
}
```

由于闭包可以捕获调用位置的上下文状态，虽然十分方便，但是对于性能影响稍大，在有强烈性能要求的场合中使用要谨慎。

延迟调用

延迟调用是 Go 中的特色功能，通过关键字 `defer` 在函数中注册函数调用，可以声明该函数调用需要延迟到函数结束之前执行。延迟调用常用于资源释放、异常处理等场合。

以下是一个文件操作的示例。

```
func main() {
  f, err := os.Open("./data.txt")
  if err != nil {
    log.Fatal(err)
  }
  defer f.Close()
  // 以下可以做其他文件访问操作，不必再担心文件关闭
}
```

关键字 `defer` 注册的是函数调用，而不是函数本身，这在使用时需要注意，如果注册的函数调用需要参数，需要在注册位置给出。一个函数中的多个延迟调用，会根据采用先入后出（FILO）的原则在函数结束之前逐一调用。

延迟调用不受 `return` 和 `panic` 的影响，只要函数即将结束执行，都会逐一调用注册的延迟调用。但是需要注意，`return` 语句只会启动函数执行流程的终止，但函数的返回值只有在函数完全结束执行后才会返回，如果在延迟调用中修改了函数的返回值，那么函数所返回的将是被修改后的返回值。

在 Go 1.14 版本中，`defer` 的性能获得了几大的提升，对于常见的大部分延迟调用操作，几乎都能达到零开销的程度。所以在几乎所有对性能有较高要求的场景中，延迟调用也可以放心大胆的使用。

特殊函数

Go 保留了两个函数名：`main()` 和 `init()`。这两个函数不接受任何参数，不返回任何值，并且其调用是由 Go 运行时控制的。

`main()` 是 Go 程序的主函数，表示程序功能的开始执行，通常存在于程序的 `main` 包中，每个程程序最多只能有一个 `main()` 。`init()` 用于包的初始化，Go 运行时支持每个包中最多只有一个 `init()` 函数。如果在 `main` 包中引用了其他的包，那么程序执行的流程将如下图所示。

`main` 包中也同样可以存在一个 `init()` 函数，并且 `main()` 函数会在所有的 `init()` 函数都执行完毕后开始执行。

异常处理

Go 中没有其他语言中的 `try ... catch ...` 异常处理结构，对于异常的处理，Go 采用了 `panic` 和 `recover` 的形式，但最常用也是官方所推荐的处理形式，是在函数中返回一个 `error` 对象。

使用 `error` 表示错误

Go 在标准库中将 `error` 定义为了一个接口，其形式如下：

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

我们可以利用标准库中定义的接口轻松的实现自定义错误类型。

> 对于接口的使用可参考下一章内容。

从函数中返回一个 `error` 对象的形式，通常都将 `error` 放在返回值列表的最后。当函数没有出错时，`error` 返回 `nil` 。`error` 对象可以通过标准库 `errors` 中的函数来创建，例如：

```
var errDivByZero = errors.New("divided by zero")
```

对于错误类别的判定，应该通过类似上例中的错误变量来完成，而不是通过错误中携带的字符串内容。构建一个错误时，其中携带的字符串需要全部为小写字母，并且不存在任何标点，这样可以方便将其插入到格式化字符串中输出。

自定义错误类型

在日常编码中，常常通过使用结构体来实现 `error` 接口完成自定义错误类型，这样就可以使用错误类型来进行错误的判断。例如上面的被零除错误可以优化成以下实现。

```
type DivByZeroError struct {
  x, y int
}

func (DivByZeroError) Error() string {
  return "divided by zero"
}

func Div(x, y int) (int, error) {
  if y == 0 {
    return 0, DivError{x, y}
  }
  return x / y, nil
}

func main() {
  f, err := Div(1, 0)

  if err != nil {
    switch eType := err.(type) {
    case DivByZeroError:
      fmt.Println(eType, eType.x, eType.y)
```

```
    default:
      fmt.Println(eType)
    }
    // 还可以使用以下方式进行判断，具体可参考 类型断言 一节
    if e, ok := err.(DivByZeroError), ok {
      // 出错后的处理
    }
  }
}
```

自定义错误一般以 `Error` 为后缀，在进行比对判断时，应将自定义错误类型和更加具体的错误类型放在前面。

使用 `error` 对象进行编码时，通常会使函数调用代码看起来很不爽，错误检查代码填充在了函数调用代码的间隙，也就是被饱为诟病的发令枪式代码。通过许多人的努力和总结，目前针对这种情况主要有以下几种方法进行改善。

1. 使用专门的检查函数处理错误，简化检查代码。
2. 在不影响逻辑的情况下，使用 `defer` 延后处理错误。
3. 在不中断逻辑的情况下，将错误作为内部状态保存，等待最终处理结构处理。

`panic`

`panic` 和 `recover` 的组合就与 `try ... catch ...` 结构比较相像了。`panic` 和 `recover` 是 Go 的内置函数，其中 `panic` 会立刻中断当前函数流程并开始执行延迟调用；`recover` 则用在延迟调用中用来捕获 `panic` 提交的错误对象。

`panic` 函数接受任何对象作为错误状态，但 `recover` 获取的状态则需要进行类型转换才能获得其中信息。一般情况下会使用 `panic` 传递字符串信息作为错误状态。

以下是 `panic` 和 `recover` 的使用示例。

```
func main() {
  defer func() {
    if err := recover(); err != nil {
      log.Fatal(err)
    }
  }

  panic("something happens")
  fmt.Println("never reaches")    // 程序不会执行到这里
}
```

在程序中发生 `panic` 后，所有的延迟调用都将开始执行，`panic` 提交的错误将沿着调用栈向回传递，直到被最近的 `recover` 捕获。如果没有 `recover` 捕获 `panic` 提交的错误，那么整个进程最终将会崩溃。

`recover` 必须在延迟函数中才能正常工作，不再延迟函数中或者嵌套在延迟函数中都不会起到任何作用。

`try ... catch ...` 结构可以通过 `panic` 和 `recover` 来实现，但是看起来可能比较不美观。

```go
func Div(x, y int) int {
  z := 0

  func() {        // 利用匿名函数定义并同时执行来保护会抛出panic的语句
    defer func() {
      if recover != nil {
        z = 0
      }
    }()

    z = x / y     // 这里当y=0时会产生panic
  }

  return z
}
```

> 这里需要注意的是，延迟调用是不能返回任何值的，要让延迟调用修改函数返回值，必须采用类似上例中的做法。
>
> `panic` 一般用在系统无法正常工作的错误下，不建议在一般错误中也使用 `panic` 。

错误包装

自 Go 1.13 版本开始引进了错误包装，自此对于错误的处理又增加了新的方式。

首先，一个错误可以通过 `fmt.Errorf("其他错误提示: %w", err)` 进行包装。新得到的错误对象，可以通过 `errors.Unwrap()` 函数来解包装获得其原始错误。

其次，对于错误的判断，`errors` 包提供了 `Is()` 和 `As()` 两个函数来进行比较。其中 `Is()` 函数用于错误类型的值比较，`As()` 用于测试错误是否为指定类型。比如前节中判断 `DivByZeroError` 的示例，其判断语句可以进行以下改写。

```go
// 前哨错误判断
var DivByZeroError = errors.New("divided by zero")
if err == DivByZeroError { ... }
// 使用 errors.Is()
if errors.Is(err, DivByZeroError) { ... }

// 类型判断语句
if e, ok := err.(DivByZeroError), ok { ... }
// 使用 errors.As()
var e DivByZeroError
if errors.As(err, &e) { ... }
```

面向对象

虽然 Go 是面向过程编程的语言，但这并非指示 Go 不支持面向对象编程。Go 对于面向对象的实现是基于面向过程的，这使得 Go 的面向对象不像 Java、C++那样严谨，并且在编程方式上有着显著的不同。Go 中没有类的概念，对于面向对象的封装特性主要依靠结构体来实现；并且由于面向对象中的继承在复杂系统维护时的复杂度，Go 直接选择了不支持继承，而是通过组合来实现面向对象的扩展性。

> 通过组合来组装类不存在继承带来的父子依赖，使整体与局部松耦合。整体中的各个单元遵循单一职责原则，彼此无关联，可以使面向对象的实现和维护更加简单。所以在目前面向对象的程序设计中，由于对松耦合的追求，更加提倡使用组合来替代继承。

所以，在 Go 中面向对象是以结构体为基础实现的。

结构体

结构体是一个自定义类型，通过组合 Go 的内置类型和其他自定义类型创建，并且其中可以包含一些自定义的方法。Go 的结构体与 C 的结构体更加相似，如果对 C 比较熟悉，那么应该可以很容易的理解 Go 的结构体。

结构体通过关键字 `struct` 创建，格式为：

```
struct {
  字段标识符 字段类型
}
// 通常会将结构体类型定义与自定义类型语句合并使用
// 以直接定义一个新的类型
type 自定义类型标识符 struct {
  字段表示符 字段类型
}
```

结构体是一个复合类型，其中的字段名必须是唯一的，并且字段排列顺序会影响到其初始化。

结构体的初始化可以通过对其中字段顺序赋值或者命名赋值的方式完成，其初始化方式主要有以下几种形式。

```
// 例如有以下结构体
type Person struct {
  Name string
  Age int
}
// 使用关键字 new 初始化，承载变量类型为 *Person，其中字段均为零值
p0 := new(Person)
// 使用顺序初始化，承载变量的类型为 Person
p1 := Person{"John", 20}
// 使用命名初始化，承载变量的类型为 Person
p2 := Person{
  Name: "John",
  Age: 20
```

```
}
// 常见的返回指针的初始化，承载变量的类型为 *Person
p3 := &Person{"John", 20}
p4 := &Person{
  Name: "John",
  Age: 20
}
```

> 关键字 `new` 始终返回指针，也就是 `*T` 类型的实例，而函数 `make()` 始终返回 `T` 类型的实例。

对于结构来说，指针变量可以直接操作其中的字段，但二级指针就不可以。例如：

```
// 接上例中的变量
p4.Age++    // 这是可以的，*Person 类型指针变量可以与 . 操作符一起工作
pp := &p4   // 将 *Person 类型的变量 p4 的地址赋予新变量
*pp.Age++   // 理论上 *pp 指向 p4，是可以完成该操作的，但是这实际上是不允许的
```

字段声明

Go 的结构体中的字段声明遵守 Go 的导出规则，以大写字母开头的成员将被导出，可以从外部进行访问。结构体不能嵌套包含自身类型成员，但是可以包含自身类型的指针成员，例如下例中定义的二叉树。

```
type Tree struct {
  Value int
  Left, Right *Tree     // 如果不使用指针是不可以的
}
```

结构体中可以包含匿名字段，这种匿名字段只有类型，没有字段标识符，这种字段被称为嵌入字段。嵌入字段是 Go 实现组合的主要方式。以以下两个结构体为例。

```
type Score struct {
  Math int
  Physics int
  Chemistry int
}

type Student struct {
  Name string
  Class string
  Score         // 仅使用类型名称定义嵌入字段
}
```

嵌入字段默认使用类型的名称作为字段名，在初始化时需要作为独立字段显式书写。例如：

```
john := Student{
  Name: "John",
  Class: "2-A",
  Score: Score{
    Math: 8,
    Physics: 7,
    Chemistry: 7
  }
}
```

但是嵌入字段在访问或赋值时，则不需要书写字段名，可以直接访问嵌入字段中的字段成员。

```
fmt.Println(john.Physics)
```

> 嵌入其他包中的字段，字段名称将不包含包名，只包含类型名。并且出接口指针和多级指针之外的所有命名类型都可以作为匿名字段，但基础类型和它的指针类型不能同时作为匿名字段嵌入。

匿名字段中的成员如果与结构体中的具名字段重名时将会被遮蔽，这时如果需要访问匿名字段中被遮蔽的成员，需要书写匿名字段的类型名称来进行显式访问。

字段标签

结构体的字段在声明时可以在其类型后使用反引号 ``` 来加入标签。字段标签是用来对字段进行描述的元数据，是类型的组成部分，通常在其中编写校验信息、数据库关联映射信息等。

字段标签可以在运行时通过反射获取。 并且可以按照一定格式自动分解。可运行以下示例来观察。

```
package main

import {
  "fmt"
  "reflect"
}

type Person struct {
  Name string `field:"姓名" type:"varchar"`
  Age int `field:"年龄" type:"int4"`
}

func main() {
  // 注意，john 现在是一个 *Person 类型的变量
  john := &Person{
    Name: "John",
    Age: 20
  }
  t := reflect.TypeOf(john)
  // 注意 .Elem() 的使用，由于 john 是一个指针变量，所以需要使用 .Elem()
  // 获取其指向的实际内容
```

```
  for i := 0; i < t.Elem().NumField(); i++ {
    // .Field() 系列方法用于获取结构体的字段描述
    f := t.Elem().Field(i)
    // .Tag 可以返回字段描述中的标签，.Get() 方法可以返回标签中按照格式规则解析的结果
    fmt.Printf("%s, %s, %s\n", f.Name, f.Tag.Get("field"),
f.Tag.Get("type"))
  }
}
```

字段标签中可以被解析的内容格式是使用空格或逗号分隔的 `域名称:"域值"` 的键值对，其中域值只可以使用字符串形式，不接受其他任何类型。

方法

结构体上除了可以承载字段成员以外，还可以承载方法成员。结构体上的方法是一个特殊的函数，使用普通函数的声明方法声明，但其中要加入指示其归属的接收者（receiver）。

```
func (接收者变量 接收者类型) 函数标识符(参数列表) 返回值类型列表 {
  方法体
}
```

在定义方法时，如果方法中不需要访问结构体中的字段或者其他方法，接收者变量可以省略，但接收者类型不可省略。

> 方法实际上不止可以定义在结构体上，在一个包中，除了接口与指针类型以外，都可以为其定义方法。但这种做法一般并不推荐，因为 Go 不支持重载，一旦定义了相同名称的方法，可能会导致编译失败，花费较大的纠错成本。

以下给出一个为结构体定义方法的示例。

```
type Person struct {
  Name string
  Age int
}

func (p *Person) AgeUp() {
  p.Age++
}

func (p Person) String() string {
  return fmt.Sprintf("%s %d", p.Name, p.Age)
}
```

在阅读一些示例代码后，你会发现接收者类型的书写方法有类型（`T`）和指针类型（`*T`）两种，对于在编写方法时，如何选择接收者类型可以参考以下规则：

1. 如果需要修改实例的状态，接收者类型选择 `*T`。
2. 无需修改实例状态的小对象或者固定值，接收者类型选择 `T`。

3. 如果实例类型较为复杂或庞大，接收者类型选择 `*T`，以提高性能。
4. 接收者类型为引用类型、字符串、函数等，接收者类型选择 `T`。
5. 包含同步锁等同步字段或者其他无法确定的情况，接收者类型选择 `*T`。

前面提到过结构体是可以嵌入其他类型来进行组合的，如果结构体（`T`）嵌入了另一个携带有方法的结构体（`S`），那么对于结构体（`T`）则根据其实例类型其方法集会出现以下几种组合情况，在实际工程编码中要注意。

1. 实例类型 `T` 的方法集包含所有接收者类型为 `T` 的方法。
2. 实例类型 `*T` 的方法集包含所有接受这类型为 `T` 和 `*T` 的方法。
3. 匿名嵌入 `S`，则实例类型 `T` 的方法集包含所有 `T` 和 `S` 的方法。
4. 匿名嵌入 `*S`，则实例类型 `T` 的方法集包含所有 `T` 和 `S` 以及 `*S` 的方法。
5. 匿名嵌入 `S` 或者 `*S`，则实例类型 `*T` 的方法集包含所有 `T`、`*T`、`S` 和 `*S` 的方法。

结构体的方法集仅影响后文接口的实现和方法表达式的转换，与实例如何调用方法无关。

方法表达式

结构体的方法和函数一样，可以直接赋给变量或者作为参数传递。根据方法的引用方式不同，可以分为表达式状态和值状态。

采用类型引用的方法表达式会以普通函数形式出现，在调用时需要显式提供接收者参数，格式为 `方法变量标识符(接收者变量, 其他参数列表)`。在调用时，接收者变量的类型方法集中需要包含相应的方法。

以下给出一个使用类型引用的方法表达式的示例。

```
type Name string

func (n Name) Format() {
  fmt.Printf("The person is %s.\n", n)
}

func main() {
  var john Name = "John"

  fv := Name.Format
  fv(john)

  fp := (*Name).Format
  fp(&john)

  // 或者可以直接进行调用
  Name.Format(john)
  (*Name).Format(&john)
}
```

接收者变量需要是函数调用的第一个参数，并且要与类型引用方式相匹配。

方法的值状态来自于实例或者是实例的指针引用。在使用值状态的方法时，函数签名不会改变，因为此时函数调用已经固化了其所对应的实例。这里继续由以上示例给出一个值状态的示例。

```
func main() {
  var john Name = "John"

  f := john.Format
  f()
}
```

构造函数

结构体在默认情况下，Go 会使用零值来初始化其中所有字段，不需要使用构造函数。但是在零值构造函数不能满足要求的情况下，我们也可以创建一个“构造函数”。这个构造函数实际上就是一个普通的函数，只是利用返回局部变量地址会提升局部变量生命周期的特性实现了复杂结构体的构建操作。

以下给出一个常见的构造函数示例。

```
type Cat struct {
  Name string
  Kind string
}

func NewCat(name string, kind string) *Cat {
  return &Cat{
    Name: name,
    Kind: kind
  }
}
```

接口

Go 中的接口是一个完全抽象不可实例化的自定义类型，其中包含了一个或多个方法的签名。接口可以被看做是一个契约或者协议，结构体不需要显式实现接口，只要结构体中包含接口中声明的全部方法，就可以看做这个结构体实现了这个接口。Go 中的接口完全实现了鸭子类型的特点。

> **鸭子类型** 是指如果一个动物（类型）看起来像是鸭子，行为像是鸭子，那么它就是鸭子。这个原则引申到接口上，就是如果一个类型能够提供接口要求的所有方法，那么就可以说这个类型实现了这个接口。

接口的出现解除了代码中对于具体类型的强依赖，使用户只需要关心所使用实例能够提供的行为，而对实例中对于行为的实现进行了屏蔽。接口虽然不能实例化，但是可以用其来声明变量，这个变量可以被赋值任何满足该接口类型的实际类型的实例。接口变量的初始化值为 `nil`。

接口一般使用后缀 `er` 来声明，通常采用自定义类型定义，格式如下：

```
type 接口名称 interface {
  方法签名
}
```

以下给出一个接口的应用示例。

```
type Animal interface {
  Run()
}

type Dog struct {
  Kind string
}

func (d Dog) Run() {
  fmt.Printf("The %s dog runs", d.Kind)
}

type Cat struct {
  Kind string
}

func (c Cat) Run() {
  fmt.Println("The %s cat runs", c.Kind)
}

func main() {
  var a Animal = Dog{"Husky"}
  a.Run()

  a = Cat{"Ragdoll"}
  a.Run()
}
```

从以上示例可以看出，变量 `a` 调用了两次 `Run()` 方法，但是两次输出的内容不相同。

接口在声明时需要遵守以下约束：

1. 接口中不能存在字段。
2. 只允许存在方法声明，不允许存在已经实现的方法，即默认方法。
3. 可以嵌入其他接口类型，嵌入的声明方法与结构体类型嵌入相同。

一个接口在嵌入其他接口后，被视为接口实现的类型必须拥有接口和嵌入接口的全部方法。并且接口在嵌入时，接口与被嵌入的接口不能有同名方法，并且接口不能嵌入自身或者循环嵌入。以下是一个不正确的循环嵌入的示例。

```go
type InterfaceMaster interface {
  InterfaceSlave
}

type InterfaceSlave interface {
  InterfaceMaster
}
```

> 自 Go 1.14开始，具有相同签名的方法可以被内嵌进一个接口中，但是在之前的版本中，这样做将会报错。

空接口

空接口是一种没有声明任何方法的接口，即 `interface{}`，在 Go 中会经常看到空接口的使用。在 Go 中，空接口的作用类似于 Java 中的 `Object` 根类型，一个空接口类型的变量可以被赋值为任何类型的对象。

```go
func main() {
  var v interface{}

  v = 100
  fmt.Println(v)

  v = "Hello world"
  fmt.Println(v)
}
```

类型转换

接口的使用，使具体类型的类型信息和成员被隐藏，我们只能访问接口提供的方法。但是很多时候我们需要将接口变量转换回原始具体类型，这就需要使用类型转换。对一个接口变量进行类型转换，一般使用以下两种模式。

```go
func main() {
  var d Data = 5
  var x interface{} = d

  // 最常见的 ok-idiom 模式
  if v, ok := x.(fmt.Stringer); ok {
    fmt.Println(v)
  }

  // 转换回原类型
  if d2, ok := x.(Data); ok {
    fmt.Println(d2)
  }

  // 使用 switch 完成转换判断
```

```
  switch v := x.(type) {
  case fmt.Stringer:
    fmt.Println(v)
  case Data:
    fmt.Println(v)
  }
}
```

以上示例中的语法结构 `x.(T)` 是类型断言语法。类型断言接受一个接口类型值，并从中抽取出显式指定类型的值。示例中的 `x` 即是类型断言所接受的接口类型值，而其后的 `.(T)` 则是要抽取出的指定类型。类型断言除了可以转换为具体类型以外，还可以将接口值转换为另一个接口类型。如果类型断言语法执行失败，将会抛出一个运行时错误，但类型断言语法提供了返回两个返回值的方法来避免产生错误，这就是上例中所采用的的 ok-idiom 模式。当类型断言返回的第二个值为 `false` 时，表示类型断言执行失败，所得到的新变量为零值。

> 示例中的语法结构 `x.(type)` 表示获取实例 `x` 的类型并同时返回类型转换后的实例，所以该用法可以使用 `switch` 进行类型判断，转换后的实例的类型将根据 `case` 语句匹配的类型确定。

并发

Go 被设计为从更高级别支持高并发功能，但是有一点需要明确，并发能力是程序同时处理多个任务的能力，而不是同时执行多个并发任务的能力。这两种说法有什么区别呢?

在单核处理器上，每时每刻只会有一个任务在运行，所有的任务按照调度，轮流在处理器上运行，这种执行方式被称为“时间片轮转”。在多核处理器上，虽然可以实现多个任务同时执行，但同时执行的任务数量不会超过处理器核心的数量。并行运行是并发设计最理想的执行方式，但是我们的程序不可能占据所有的处理器核心。

传统的多线程是在一个进程中开辟多个线程来实现多任务并发，随着技术的发展，目前引入了协程来完成多任务的并发。多个协程工作在一个线程中，多个协程的主动切换要比多个线程的切换更加有效率，损失的性能也更小。但是实际上，多个协程在执行时，多个任务的执行依旧是串行的。Go 中提供的 goroutine 并不是完全的协程，而是会根据任务单元和系统资源来创建线程或者协程，所以更像是两者的结合体。

创建并发任务

在 Go 语言中，每个并发任务被称为一个 goroutine。goroutine 的构建实际上非常简单，使用关键字 `go` 调用一个普通函数就可以完成一个 goroutine 的创建。使用关键字 `go` 调用的函数由于运行在新创建的 goroutine 中，所以不会阻塞当前程序的运行。实际上，在程序启动时，主函数就运行在一个 goroutine 上。但是需要注意的是，当主函数运行结束时，所有的 goroutine 都会直接中断。Go 没有提供从一个 goroutine 控制其他 goroutine 运行状态的功能，但是我们可以通过后一节中介绍的 goroutine 通信功能来控制 goroutine 的活动状态。

> 在设计并发程序时，由于所有的子 goroutine 在主 goroutine 退出后，都会中断工作并直接退出，所以会造成一些工作未能执行完毕，所以有必要通过一些同步的方式使主 goroutine 在所有子 goroutine 都退出后结束，具体实现可见后面“同步”一节。

作为 goroutine 的函数，可以接受参数，但没有返回值。goroutine 在启动时会立刻计算并捕获所有传入参数的值。以下给出一个没什么意义的简单示例。

```
1  func main() {
2      a := 1
3      go func(x, y int) {
4          time.Sleep(time.Second)          // 延迟 goroutine 的执行
5          fmt.Println(x + y)
6      }(a, 3)                              // goroutine 会立刻复制参数，固定状态
7
8      a += 10
9      fmt.Println(a + 3)
10     time.Sleep(time.Second * 3)          // 等待 goroutine 执行结束
11 }
```

通信

由于 goroutin e没有返回值，所以在一些需要子 goroutine 返回数据的情况下，就需要用到 goroutine 间的通信。goroutine 之间的通信除了可以在 goroutine 之间传递数据以外，还可以控制 goroutine 的运行状态。虽然 Go 提供了全局变量、指针、引用类型可以供内存共享操作使用，但还是鼓励使用 CSP 通信来代替不安全的内存共享。

前面所提到过的另一种引用类型——通道（Channel）——是 CSP 通信概念的核心。通道显式声明了通信双方在进行通信时所必须要知道的一切信息，但其中不包含通信双方的具体身份。Go提供的通道可以被想象为两根管道，一根用于传输上行数据，一根用于传输下行数据，所有在其中的传递的信息都将在管道中排队等待取出处理。通道有同步和异步两种运行模式。

通道与切片一样，都使用函数 `make()` 创建，其类型为 `chan`，但是与其他类型不一样的是，通道在创建时需要声明其中传递的内容类型，类型格式为 `chan 内容类型`。通道的创建语法格式如下。

```
// 通道变量声明，以下声明了一个传递字符串内容的通道变量
var a chan string
// 创建带缓冲队列的通道，通常用于支持异步操作
// 这个通道可以缓存三个消息
c := make(chan int, 3)
// 创建无缓冲队列的通道，通常用于支持同步操作
c := make(chan int)
```

通道主要有发送消息、取出消息和关闭三种操作，向通道发送消息和从通道取出消息都是使用 `<-` 操作符，只是发送和取出操作 `<-` 操作符的位置不同。通道的常用操作可以参考以下示例。

```
// 假设有传输整型值的通道 ch
ch := make(chan int)
// 向通道中发送数据
// 如果通道已经关闭，发送操作将产生一个 panic
ch <- 8
// 从通道中取出数据
// 当通道中没有缓存数据时，取出操作将使当前的 goroutine 阻塞
// 如果通道已经关闭，接受操作将返回已经缓冲数据或者零值
x := <-ch
// 从通道中取出数据，并测试是否取出成功
// 当通道关闭后，变量 ok 的值为 false
x, ok := <-ch
// 关闭通道
// 关闭一个已经关闭的通道将产生一个 panic
close(ch)
// 通道可以使用 for 来迭代收取其中的数据
for x := range ch { ... }
```

同步模式与异步模式

不带缓存的通道在发送时将会导致发送端 goroutine 阻塞，除非另一个 goroutine 在这个通道上执行接收语句。同理，当一个 goroutine 接收一个不带缓存的通道时，如果没有相应的发送端，接收者 goroutine 也会阻塞。这样就形成了一个多 goroutine 结构的同步模式。

以下是一个同步模式的示例。

```
func main() {
    exitSignal := make(chan struct{})       // 空结构体的长度为0，一般用作一个信号
    ch := make(chan int)

    go func() {
        defer close(exitSignal)

        for x := range ch {
            fmt.Println(x)
        }
    }()

    ch <- 1
    ch <- 2
    ch <- 3
    close(ch)       // 如果同步通道 ch 不关闭，那么子goroutine中的 for 将一直循环下去

    <-exitSignal    // 等待子goroutine结束
}
```

异步模式通道就相对较为简单了，任何带有缓冲队列的通道都是以异步模式工作的。

单向通道

通道在默认情况下都是双向的，但是可以通过定义单向通道变量来限制对通道的操作。单向通道变量的声明如下例所示。

```
// 创建一个作为源的双向通道
ch := make(chan int)
// 定义单向发送通道变量，通过将双向通道赋值给单向通道完成方向分离
var send chan<- int = ch
// 定义单向接收通道变量，同样使用双向通道赋值
var recv <-chan int = ch
```

> 单向通道可以通过 `make()` 来创建，例如 `make(chan<- int)` 和 `make(<-chan int)`，但是创建这样的通道并没有什么实际意义。

单向通道通常用于传递给函数用以限制函数的操作逻辑。如果函数在单向通道上做了逆向操作，就会抛出panic，这样就提升了操作的严谨性。单向通道无法转换回双向通道，并且单向接收的通道不能使用 `close()` 来关闭。

通道选择

普通的通道接收语法是阻塞性的，在如果一个通道上没有数据，那么通道取出语句将会阻塞 goroutine 的运行，如果一个 goroutine 要同时操作和响应多个通道，那么这种阻塞是我们不需要的。为了同时操作多个通道，Go 提供了通道选择语法，这个语法与 `switch` 十分相似。通道选择会随机选择一个可用的通道进行收发操作。

以下是通道选择语法的简单示例。

```
select {
case x, ok := <-chA:
    // 执行从通道 chA 中取出数据的处理
case x, ok := <-chB:
    // 执行从通道 chB 中取出数据的处理
default:
    // 当所有通道都不可用时的操作
    // 如果 select 不带有 default 分支，那么在没有通道可用时，select 将会阻塞
}
```

由于通道选择语句 `select` 每次只执行一次通道选择，所以在进行连续处理时，常常将其与 `for` 语句同时使用。如果一个通道的值被设为 `nil`，那么这个通道将不会再被 `select` 选中。

通道的一些常见应用

利用通道，可以实现许多多线程中的常见控制方法。以下通过几个示例来展示以下通道的常见用途。

实现信号量（Semaphore）

信号量是用来解决并发时的任务调度问题和共享资源访问的，其初始值一般是资源的可用量或者可并发任务数量，通过信号量值的变化可以控制并发任务的唤醒和等待。

```
func main() {
    var wg sync.WaitGroup
    semaphore := make(chan struct{}, 2)     // 允许2个并发同时执行

    for i := 0; i < 20; i++ {
        wg.Add(1)

        go func(id int) {
            defer wg.Done()
            semaphore <- struct{}           // 获取信号量，启动自身执行
            defer func() {
                <-semaphore                 // 释放信号量，结束自身执行
            }()
            // 这里可以做一些其他操作
```

```
        }(i)
    }

    wg.Wait()
}
```

> 关于 `sync.WaitGroup` 的用法见后面“同步”一节。

定时任务

`time` 包中提供的 `Tick` 函数会返回一个通道，这个通道可以按照固定时间间隔发出整型信号，因此可以使用 `time.Tick()` 来完成定时任务。

```
func CycleTask() {
    tick := time.Tick(time.Second)      // 时间间隔设为1秒

    for {
        select {
        case <-tick:
            // 做定时要做的操作
        }
    }
}
```

同步

通道并不能完全替代传统使用锁的并发管理。通道是通过逻辑层次来解决并发管理问题，而锁是用来保护一定范围内的数据安全。Go 标准库 `sync` 中提供了常用的锁，包括互斥锁（`Mutex`）、读写锁（`RWMutex`）、原子操作等。这一节将对常见的几种 `sync` 中的功能进行简单说明。

互斥锁

顾名思义，互斥锁是用来进行排他性访问的，一个任务获得了互斥锁，那么其他的任务就必须等待，直到获得互斥锁的任务释放锁为止。

以下是一个互斥锁使用的简单示例。

```
var (
    m sync.Mutex
    resource int
)

func Increase(amount int) {
    m.Lock()            // 尝试获得锁，成功后锁定，不成功时阻塞等待
    resource += amount
    m.Unlock()          // 释放已经获得的锁
}

func Decrease(amount int) {
```

```
    m.Lock()
    resource -= amount
    m.Unlock()
}
```

位于 `m.Lock()` 和 `m.Unlock()` 之间的代码被称为临界区，其中可以对共享资源进行随意访问。但是在临界区结束时必须要释放锁，否则程序将进入死锁状态。`sync.Mutex` 在使用时最好控制在最小的锁定范围内，并且不要递归使用锁和重复锁定，以防止出现死锁。

`sync.Mutex` 可以直接作为结构体的嵌入类型，但是所有使用锁的方法的接收者类型必须为 `*T`，否则将会使锁失效。

读写锁

`sync.Mutex` 互斥锁会同时锁定写入和读取状态，所以在并发操作时实际上效率比较低。`sync.RWMutex` 读写锁将共享资源的读操作和写操作分开分别进行锁定，在一定程度上就提升了并发操作的性能。`sync.RWMutex` 的使用方法与 `sync.Mutex` 基本相同，但是锁定和解锁的方法增加了。

以下是 `sync.RWMutex` 中常用的方法。

1. `.Lock()`，尝试获取写入锁，在未获得写入锁时阻塞。
2. `.RLock()`，尝试获取读取锁，在未获得读取锁时阻塞。
3. `.RUnlock()`，释放已经获得的读取锁。
4. `.Unlock()`，释放已经获得的写入锁。

单次任务

单次任务比较容易理解，主要目标是支持在多次调用指定函数时，只执行第一次调用的功能。单次任务多用在一些初始化功能上，可以防止多个goroutine调用同一初始化任务导致的重复初始化问题。类型 `sync.Once` 的使用非常简单，其中只提供了一个方法：`Do()`，使用格式为 `.Do(f func())`，通过 `.Do()` 调用的方法只会运行一次。

等待组

等待组 `sync.WaitGroup` 常用来在主 goroutine 中等待全部子 goroutine 运行完成的。`sync.WaitGroup` 采用计数法控制主 goroutine 的执行。`sync.WaitGroup` 主要提供了以下方法供使用。

1. `.Wait()`，使当前 goroutine 等待，直到 `WaitGroup` 实例中的计数器值为0时。
2. `.Add(delta int)`，向 `WaitGroup` 实例的计数器中添加指定数值，当计数器值为0时，所有调用 `.Wait()` 的 goroutine 都会被释放。如果计数器值变为负值，则 `.Add()` 方法会抛出 panic。
3. `.Done()`，使 `WaitGroup` 实例的计数器值减1。

具体 `sync.WaitGroup` 的使用可以参考前面“信号量”一节的示例。

常用标准库功能

Go的标准库中提供了大量的包，几乎覆盖了常用的所有功能。这里只拣选一些比较稳定而常用的包来进行简单说明。

`time` 包

`time` 包用来进行时间的解析、格式化输出和计算。其中提供的函数与其他语言没有太大差异，主要是用于日期时间的获取、计算和比较。但是Go中比较有特色的，是时间的格式化输出和格式化解析。

Go中对于时间的格式化，不是采用常见的字母表示法，而是索引表示法。根据美式时间格式，时间排列起来索引格式为：`01/02 03:04:05PM '06 -0700`，即 `1` 表示月，`2` 表示日，`3` 表示时，`4` 表示分，`5` 表示秒，`06` 表示年（需要前面的0，表示两位），`-0700` 表示时区（需要全部书写）。其中月、日、时、分、秒可以添加前导0来使其填充为两位数字，年至少为两位，可以添加 `20` 作为前缀将其扩展为4位年份数字。在格式串中可以使用 `.000` 来输出毫秒，`.000000` 来输出微秒，`.000000000` 来输出纳秒，其中 `0` 的数量可以根据所需要精度自由在 1 到 9 之间调整。

`fmt` 包

`fmt` 包是一个Go程序中最常用的包，主要用于进行输入输出的格式化。`fmt` 包中提供的常用函数主要有以下这些。

- `Errorf(format string, a ...interface{}) error`，用指定信息包装出一个新的 `error` 实例。
- `Print`，用于输出一组内容，不同的后缀表示采用不同的输出方式，不同的前缀表示输出到不同的位置。
  - 无后缀：`Print(a ...interface{}) (int, error)`，将参数拼合为一个字符串输出。
  - 后缀为 `f`：`Printf(format string, a ...interface{}) (int, error)`，根据指定格式化字符串输出全部参数。
  - 后缀为 `ln`：`Println(a ...interface{}) (int, error)`，功能与无后缀版基本相同，但自动在输出结束后输出一个换行符。
  - 前缀为 `S`：`Sprint(a ...interface{}) string`，返回一个字符串，同样可以搭配后缀版使用。
  - 前缀为 `F`：`Fprint(w io.Writer, a ...interface{}) (int, error)`，向指定 `Writer` 输出字符串，同样可以搭配后缀版使用。
- `Scan`，用于读取指定内容到变量，不同后缀与前缀版的功能与 `Print` 函数族相同。输入与输出不同，在读入内容时，需要指定要读入的变量的地址，例如：`fmt.Scan(&i, &j, &k)`。

`fmt` 包中最常用的函数是后缀为 `f` 的 `Printf`、`Sprintf`、`Fprintf`、`Scanf`、`Sscanf` 和 `Fscanf` 这几个函数，它们都是采用格式化字符串来输出给定的参数的。Go中的格式化字符串与C语言中的格式化字符串十分相似，但是要更简单，例如：`"%s %d %5.2f"`。格式化字符串采用一系列转换词来描述如何格式化相应的参数。

Go中可用的转换词主要有以下这些。

- `%v`，类型默认的输出格式。

- `%#v`，使用Go语法输出参数的值。
- `%T`，使用Go语法输出参数的类型。
- `%%`，输出 `%` 字符。
- `%t`，输出布尔值。
- `%b`，以二进制形式输出整型值。
- `%c`，以Unicode码点对应字符形式输出整型值。
- `%d`，以十进制形式输出整型值。
- `%o`，以八进制形式输出整型值。
- `%O`，以 `0o` 的八进制形式输出整型值。
- `%q`，输出单引号括起来的整型值对应的字符，或输出双引号括起来的字符串。
- `%x`，以小写字母的十六进制输出整型值，或使用小写的十六进制计数法输出浮点数，或以小写十六进制输出字符串字节。
- `%X`，以大写字母的十六进制输出整型值，或使用大写的十六进制计数法输出浮点数，或以大写十六进制输出字符串字节。
- `%b`，输出以2为底的科学计数法浮点数。
- `%e`，输出小写的科学计数法浮点数。
- `%E`，输出大写的科学计数法浮点数。
- `%f` 或 `%F`，以自然小数形式输出浮点数。
    - `%.nf`，可以指定小数形式的精度，可省略。
- `%g`，自动决定使用 `%e` 或 `%f`。
- `%G`，自动决定使用 `%E` 或 `%F`。
- `%s`，输出字符串。
- `%p`，以十六进制输出切片的首元素地址，或以十六进制输出指针的地址。

对于整型和浮点型，可以在 `%` 与描述符（浮点型描述符包含宽度和精度说明）之间使用以下字符来进行额外的格式指定。

- `n`，设定输出宽度，输出内容会在指定宽度内右对齐，例如：`%5d`。
- `+`，始终输出正负号，例如：`%+5.2f`。
- `_`，使用左对齐输出。
- `#`，使用额外输出格式，例如输出前导 `0x`。
- ` `（一个空格），在数字前留下一个符号位。
- `0`，左侧使用0填充，例如：`%08d`。

所有的转换词，都可以在 `%` 与描述符之间使用 `[n]` 的形式来指定当前描述符引用第几个参数，如果使用 `[n]*`，则会将指定参数拼入格式描述词而不是输出。具体可通过尝试以下示例来体会。

```go
fmt.Printf("%[2]d %[1]d", 100, 200)          // 会输出 200 100
fmt.Printf("%[3]*.[2]*[1]f", 54.7, 2, 6)     // 会输出 54.70
// 上面这一语句与下面的语句效果是一致的，请仔细体会
fmt.Printf("%6.2f", 54.7)
```

> 格式化输入操作也是按照格式化字符串来解析输入的。

`reflect` 包

反射包主要提供运行时反射功能，使程序可以在运行时访问和操作任意类型的值，并且还提供了元编程支持。反射包提供了两个主要的类型： `Type` 和 `Value`。

`Type` 表示一个Go类型，是一个接口，其中定义了许多方法来检查类型和组成。`Type` 类型实例可以通过 `reflect.TypeOf()` 方法获得， `.TypeOf()` 方法可以接受任意类型的值，并返回其对应的 `reflect.Type` 实例。

`reflect` 包中提供了以下函数来对类型进行操作。

- `ArrayOf(count int, elem Type) Type`，构建一个数组类型，其中 `elem` 表示元素类型。
- `ChanOf(dir ChanDir, t Type) Type`，构建一个通道类型。
- `FuncOf(in, out []Type, variadic bool) Type`，构建一个函数类型。
- `MapOf(key, elem Type) Type`，构建一个字典类型。
- `PtrTo(t Type) Type`，构建一个指针类型。
- `SliceOf(t Type) Type`，构建一个切片类型。
- `StructOf(fields []StructField) Type`，构建一个结构体类型。

`Value` 可以持有一个任意类型的值，其实例可以通过 `reflect.ValueOf()` 来获得。`Value` 类型提供了许多方法，例如 `.Bool()`、`.Complex()` 等来获取所持有值的底层类型值。`Value` 还提供了许多用来操作集合类型的方法。

`reflect` 包中提供了以下常用函数来对 `Value` 类型实例进行操作，其中以 `.` 开头的函数为 `Value` 类型中的方法。

- `Append(s Value, x ...Value) Value`，向切片 `s` 中添加值。
- `Indirect(v Value) Value`，返回指针 `v` 指向的值。
- `MakeChan(t Type, buffer int) Value`，创建一个通道。
- `MakeSlice(t Type, len, cap int) Value`，创建一个切片。
- `New(t Type) Value`，创建一个指定类型的实例，并返回指向它的指针。
- `NewAt(t Type, p unsafe.Pointer) Value`，将指定位置的内存按照指定类型解析并返回指向它的指针。
- `Select(cases []SelectCase) (chosen int, recv Value, recvOk bool)`，启动一个 `select` 选择语句，其中分支由 `cases` 指定。
- `Zero(t Type) Value`，返回指定类型的零值。
- `.CanAddr() bool`，判断实例是否可以取得被指向的地址。
- `.Addr() Value`，返回可以被指针指向的地址，相当于使用 `&` 取地址。
- `.Call(in []Value) []Value`，调用函数实例。

- `.CanSet() bool`，判断实例是否可以被赋值。
- `.Close()`，关闭通道实例。
- `.Convert(t Type) Value`，将实例转换为指定类型。
- `.Elem() Value`，返回实例所指向的值。
- `.Field(i int) Value`，取得结构体第 `i` 个字段的值。
- `.FieldByName(n string) Value`，取得结构体中名称为 `n` 的字段的值。
- `.FieldByNameFunc(match func(string) bool) Value`，返回结构体中字段名满足 `match` 函数的字段的值。
- `.Index(i int) Value`，返回数组、切片、字符串中索引为 `i` 位置的值。
- `.IsNil() bool`，判断当前实例是否是空。
- `.IsZero() bool`，判断当前实例是否是零值。
- `.Kind() Kind`，取得当前实例的底层类型。
- `.Len() int`，取得当前实例的长度，只应用于数组、通道、字典、切片和字符串。
- `.MapIndex(k Value) Value`，取得字典中键 `k` 对应的值。
- `.MapKeys() []Value`，取得字典中的所有键。
- `.Method(i int) Value`，取得结构体中第 `i` 个方法。
- `.MethodByName(n string) Value`，取得结构体中名称为 `n` 的方法。
- `.NumField() int`，取得结构体中字段的数量。
- `.NumMethod() int`，取得结构体中方法的数量。
- `.Pointer() uintptr`，获得一个指向当前值的指针。
- `.Recv() (x Value, ok bool)`，从通道中收取一个值。
- `.Send(x Value)`，向通道中发送一个值。
- `.Set(x Value)`，给实例赋一个新值。
- `.Slice(i, j int) Value`，用实例创建一个新的切片。
- `.String() string`，返回一个用于表示实例的字符串。
- `.TryRecv() (x Value, ok bool)`，尝试从通道中收取一个值，但不阻塞当前goroutine。
- `.TrySend(x Value) bool`，尝试向通道中发送一个值，但不阻塞当前goroutine。
- `.Type() Type`，获取实例的类型。

`context`包

在并发程序中，由于超时、取消或者一些异常的出现，往往需要执行抢占式操作或者中断后续操作，这种操作需求一般可以使用一个`chan bool`类型的通道来处理，但是这样做的效果是模板代码变得更多了，不仅啰嗦而且难以管理。`context`包的出现提供了一种编程模式，允许使用更加优雅的解决方案。

`context.Context`接口的定义如下：

```
type Context interface {
  Deadline() (deadline time.Time, ok bool)
  Done() <-chan struct{}
  Err() error
  Value(key interface{}) interface{}
}
```

`context.Context`的定义可以解释为以下常用场景。

- `Deadline()`返回绑定当前`context`的任务被取消的截止时间，也就是任务执行的最后期限，如果没有设定这个期限，那么返回值`ok`的值为`false`。
- `Done()`返回一个容纳单个值的通道，如果当前的`context`不会被取消，那么这个通道将会返回`nil`。`Done()`可以作为一个广播通知来使用，通知所有使用`context`的函数停止工作。
- `Err()`用于返回错误信息，需要与`Done()`返回的通道结合使用。
    - 如果`Done`返回的通道没有关闭，那么`Err`将返回`nil`；
    - 如果`Done`返回的通道已经关闭了，那么`Err`将返回一个非空的值表示任务结束原因。
        - 如果`context`被取消，那么`Err`将会返回`Canceled`；
        - 如果`context`超时，`Err`将返回`DeadlineExceeded`。
- `Value()`返回`context`中存储的键值对中`key`对应的值，如果`key`不存在，则会返回`nil`。

`context`包提供了两个顶级`Context`，分别使用其对应的函数来构建：

- `Background`是所有`Context`的根，从来不会被取消，也没有终止期限，也不携带任何值，一般用于主进程中使用。
- `TODO`与`Background`的特性基本上一样，只是用在不知道该使用什么`Context`类型的时候。

在实际使用的时候，是需要使用以下函数来中父级`Context`中派生：

- `WithCancel()`返回一个父`Context`被取消时，也同样关闭自己的`Done`通道的`Context`。
  与`Context`实例一同被返回的还有一个用于取消`Context`的函数。
- `WithDeadline()`返回一个Deadline为指定时间的`Context`，随`Context`一同返回的同样也有一个用于取消`Context`的函数。
- `WithTimeout()`返回一个Deadline为当前时间加上指定时间间隔的`Context`，同样也会返回一个用于取消`Context`的函数。
- `WithValue()`返回一个附加了指定键值对的`Context`。

所有的`Context`实例会形成一棵树，每个`Context`实例仅会包含一个键值对，如果使用`Value()`获取其中所携带的值时，`Context`将会沿着整棵`Context`树从子节点像根节点检索。

工程管理

Go是一门工程语言，在设计之初就已经考虑了如何解决当前工程管理中的问题，并且Go自带的工具链已经提供了完善的工程管理功能。

在安装完Go的安装包之后，Go工具链就被自动安装好了，可以在CMD、PowerShell、终端等命令行界面中通过命令 `go` 来使用。在命令行中执行 `go help` 可以列出Go工具链的功能说明。总起来说，Go工具链可以完成以下这些工作。

- 代码格式化。
- 代码质量分析和修复。
- 单元测试。
- 工程构建。
- 代码文档提取和展示。
- 管理依赖。

提取文档

文档是程序的一部分，也是软件产品重要的组成部分。代码是告诉计算机如何完成工作，而文档则是告诉使用和维护软件的人如何完成工作。对于编程者来说，文档更多指的是代码中的注释、函数、接口的输入输出、功能和参数说明等，这些文档都影响着软件后续的维护和使用。

在理想状态下，设计文档和代码应该是同步的，但是在实际情况中，两者之间的差距往往随时间逐渐变大并最终导致设计文档被废弃。自从Java引入了Javadoc之后，这种设计文档与代码的差距逐渐缩小，而Javadoc可以直接将代码中的注释提取出来形成HTML格式的文档。Javadoc的编码规则较为复杂，为注释的编写增加了不少的工作量，所以Go提供了一种更加简单的注释规范，使编写者更加专注于内容。

通过Go工具链中的 `go doc` 命令可以提取项目中的注释内容，并将其格式化输出到终端中。如果要在浏览器窗口中查看，则可以添加选项 `-http=:端口号`，这样就可以通过访问 `http://localhost:端口号` 来访问 `go doc` 提取得到的效果。如果添加了 `-play` 选项，则可以在网页中使用 playground 试验代码。

Go自动提取的文档一般包含三大部分：

1. Overview 总览，包含包的 import 语句和概要说明，一般放置在 `package` 之前。
2. Index 目录，包含在包中可见（首字母为大写）的常量、变量、方法、函数的总目录和说明。
3. Examples 示例，包含文档中所有示例的快速跳转。

编写可以被自动提取的注释一般要遵循以下格式要求。

- 注释符 `//` 或者 `/*` 后面要加一个空格。
- 注释应该在关键字 `package`、`const`、`type`、`func` 等上方且紧邻关键字逐行连续书写。这种注释被称为关键字注释
- 针对 `type`、`const`、`func` 以标识符为注释开头， `package` 以 包名或者 `Package 包名` 为注释开头。
- 每个关键字注释最好不要超过三行。
- 包的Overview如果小于三行，可以放置在包的任意组成文件中，如果超过三行，则应该使用独立的 `doc.go` 来存放包的Overview。
- 注释如果需要换行，则需要保留一个空行。

- 一个包中的多个文件，注释将按照文件名的字母顺序排列显示。
- 行首字母为大写的，会被提取为段落。行首存在缩进的，会自动转换为代码段。
- 以 `BUG(人名):` 开头的注释，将被识别为已知bug，并显示在 `bugs` 区域。

示例的编写则比较特殊，需要起一个新的文件来存放，存放示例的文件和其中内容需要遵循以下要求。

- 存放示例的文件必须放置在对应的包下，文件名以 `example` 开头，使用 `_` 连接其他内容，并以 `test` 结尾。例如：`example_control_test.go`。
- 存放示例的文件其包名不能与对应的包相同，一般习惯命名为 `当前包名_test`。
- 用作示例的函数名称需要使用以下格式，以指明示例函数所对应的原函数。`func Example[所属类型_][对应函数名][_标签名]()`。具体命名可参考以下示例。
    - `func Example()`，展示在Overview区域。
    - `func Example_someTag()`，展示在Overview区域，并配以 `someTag` 标签。
    - `func ExampleFound()`，展示在Function区域 `Found()` 函数部分。
    - `func ExampleControl_At_someTag()`，展示在Function区域 `Control` 结构体的 `At` 方法部分，并配以 `someTag` 标签。
- 函数结尾可以使用 `// Output:` 注释，以说明函数逦返回值。返回值的注释从 `// Output:` 下一行开始分行书写，每行书写一个输出。

单元测试

Go的工具链提供了一套轻量级的测试框架，可以执行功能测试和性能测试两种单元测试。单元测试命令会自动在当前包中寻找符合命名规则的单元测试文件并执行其中的测试函数。单元测试的创建规则如下。

- 在需要测试的包下创建以 `_test` 结尾的Go文件。要注意与前面示例文档的文件有所区分。
- 功能测试函数以 `Test` 为前缀命名，性能测试函数以 `Benchmark` 为前缀命名。
- 功能测试函数只能接受一个类型为 `*testing.T` 的单一参数。
- 性能测试函数只能接受一个类型为 `*testing.B` 的单一参数。
- 功能测试函数通过代码在执行过程中是否会发生错误来返回不同的结果。可以通过传入参数的 `.Errorf()` 方法抛出错误。

以下分别给出一个功能测试函数和一个性能测试函数的示例供参考。

```
1  func TestAdd(t *testing.T) {
2      result := Add(4, 5)
3      if result != 9 {
4          t.Errorf("A format string", result) // 测试的输出，可以像
   fmt.Printf() 一样使用
5      }
6  }
7
8  func BenchmarkAdd(b *testing.B) {
9      // 操作计时器的操作不是必须的，只是为了不让耗时的准备过程影响评估
10     b.StopTimer()        // 暂停计时器
11     // 完成比较费时的准备工作
```

```
    b.StartTimer()      // 恢复计时器

    // for 循环比较重要，是用于让测试运行足够长的时间的
    for i := 0; i < b.N; i++ {
        Add(4, 5)
    }
}
```

在包目录下执行 `go test` 将会直接开始包中所有的功能测试，如果要进行性能测试，需要添加 `-test.bench` 选项。

项目编译

在一个项目完成编码后，就需要进行编译。Go项目根据是否有 `main` 包，可一个分为库项目和可执行项目两种，但编译方法是一致的，都是通过 `go build` 命令。`go build` 命令一般在项目根目录中执行，编译得到的可执行文件或者库文件将会放置在 `main` 包所在的目录中。

如果需要将编译结果放置在其他目录中，可以使用选项 `-o` 来指定存放位置。

Go工具链对跨平台开发也有很强悍的支持，允许进行交叉编译。PC与服务器通常都是x86或者x86_64架构，而移动设备则通常为ARM架构；而不同的操作系统其可执行文件的格式也不相同，例如Windows采用PE格式，而Linux采用ELF格式，这就导致在Windows上编译得到的可执行文件不能在Linux中运行，反之亦然。在编译指定架构的可执行文件时，一般不会选择构建一个指定架构的环境专门用于项目编译，而是采用交叉编译的方法。

由于Go对Linux的支持是最好的，所以在Linux下，Go工具链的交叉编译功能是最强的，可以生成x86 ELF、AMD64 ELF、ARM ELF、x86 PE和AMD64 PE格式。而在Windows下，则只能生成x86 PE和AMD64 PE格式。所以对于使用Windows 10的用户，可以通过安装Linux子系统来在Windows环境下使用基于Linux的交叉编译功能。

要执行交叉编译，可以通过设定 `GOOS` 和 `GOARCH` 两个环境变量来设置交叉编译的目标格式。例如在Linux系统上编译64位Windows可执行文件的命令可以是以下命令。

```
GOOS=windows GOARCH=amd64 go build
```

其中环境变量 `GOOS` 可以取 `darwin`（macOS）、`freebsd`、`linux` 和 `windows`，而 `GOARCH` 则可以取 `386`（32位）、`amd64`（64位）和 `arm`，可以通过组合这两个环境变量不同的值，来完成交叉编译功能。

打包发布

Go目前的项目组织形式，并不适合使用二进制方式分发功能包，由于Go对于兼容性的控制非常严格，一个版本号的不同都会导致链接失败。所以在进行分发时，应该尽量选择打包源码进行分发，或者将源码托管到Github等位置，由使用者自行编译。

对于二进制可执行文件的分发则没有这种约束，可以在编译后直接分发。