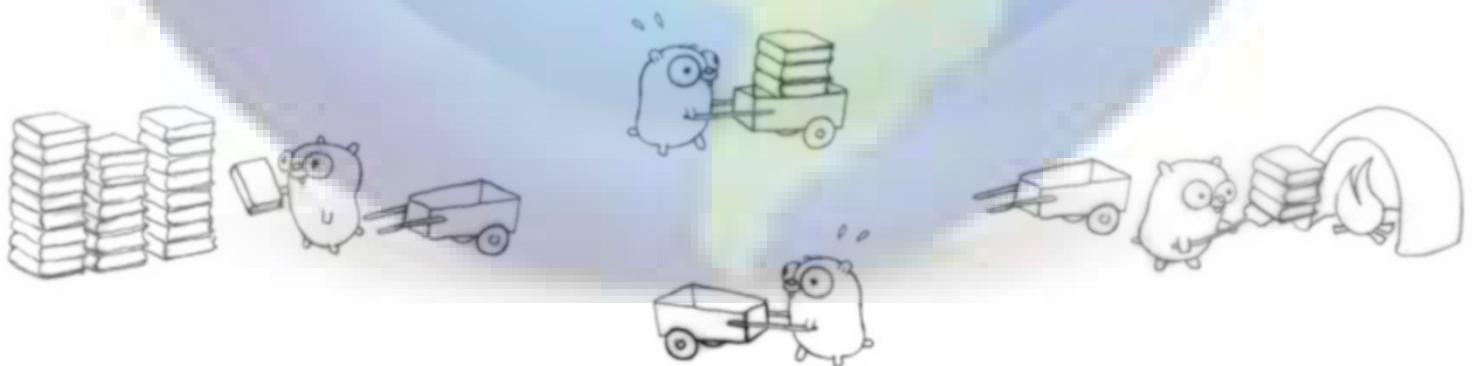


Go 101

= v1.21.0-745652d =



Tapir Liu

Contents

- §0. [About Go 101](#) - why this book is written.
- §1. [Acknowledgments](#)
- §2. [An Introduction of Go](#) - why Go is worth learning.
- §3. [The Go Toolchain](#) - how to compile and run Go programs.
- Become Familiar With Go Code
 - §4. [Introduction of Source Code Elements](#)
 - §5. [Keywords and Identifiers](#)
 - §6. [Basic Types and Their Value Literals](#)
 - §7. [Constants and Variables](#) - also introduces untyped values and type deductions.
 - §8. [Common Operators](#) - also introduces more type deduction rules.
 - §9. [Function Declarations and Calls](#)
 - §10. [Code Packages and Package Imports](#)
 - §11. [Expressions, Statements and Simple Statements](#)
 - §12. [Basic Control Flows](#)
 - §13. [Goroutines, Deferred Function Calls and Panic/Recover](#)
- Go Type System
 - §14. [Go Type System Overview](#) - a must read to master Go programming.
 - §15. [Pointers](#)
 - §16. [Structs](#)
 - §17. [Value Parts](#) - to gain a deeper understanding into Go values.
 - §18. [Arrays, Slices and Maps](#) - first-class citizen container types.
 - §19. [Strings](#)
 - §20. [Functions](#) - function types and values, including variadic functions.
 - §21. [Channels](#) - the Go way to do concurrency synchronizations.
 - §22. [Methods](#)
 - §23. [Interfaces](#) - value boxes used to do reflection and polymorphism.
 - §24. [Type Embedding](#) - type extension in the Go way.
 - §25. [Type-Unsafe Pointers](#)
 - §26. [Generics](#) - use and read composite types
 - §27. [Reflections](#) - the `reflect` standard package.
- Some Special Topics
 - §28. [Line Break Rules](#)
 - §29. [More About Deferred Function Calls](#)
 - §30. [Some Panic/Recover Use Cases](#)

- §31. [Explain Panic/Recover Mechanism in Detail](#) - also explains exiting phases of function calls.
- §32. [Code Blocks and Identifier Scopes](#)
- §33. [Expression Evaluation Orders](#)
- §34. [Value Copy Costs in Go](#)
- §35. [Bounds Check Elimination](#)
- Concurrent Programming
 - §36. [Concurrency Synchronization Overview](#)
 - §37. [Channel Use Cases](#)
 - §38. [How to Gracefully Close Channels](#)
 - §39. [Other Concurrency Synchronization Techniques](#) - the sync standard package.
 - §40. [Atomic Operations](#) - the sync/atomic standard package.
 - §41. [Memory Order Guarantees in Go](#)
 - §42. [Common Concurrent Programming Mistakes](#)
- Memory Related
 - §43. [Memory Blocks](#)
 - §44. [Memory Layouts](#)
 - §45. [Memory Leaking Scenarios](#)
- Some Summaries
 - §46. [Some Simple Summaries](#)
 - §47. [nil in Go](#)
 - §48. [Value Conversion, Assignment and Comparison Rules](#)
 - §49. [Syntax/Semantics Exceptions](#)
 - §50. [Go Details 101](#)
 - §51. [Go FAQ 101](#)
 - §52. [Go Tips 101](#)
- §53. [More Go Related Topics](#)

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

About Go 101

I feel it is hard to describe the contents in this article in the general description manner. So this article will use the interview manner to make descriptions instead.

Hi Tapir, when and why did you plan to write this book?

At about July 2016, after (not very intensively) using Go for two years, I felt that Go is a simple language and I had mastered Go programming. At that time, I had collected many details in Go programming. I thought I can archive these details into a book. I thought it should be an easy job.

I was wrong. I was overconfident. In trying to make explanations for some details, I found I couldn't explain them clearly. With more and more confusions being gathered, I felt my Go knowledge was so limited that I was still a newbie Go programmer.

I gave up writing that book.

Gave up? Isn't this book already finished now?

It was that book being cancelled, not the book **Go 101**. I eventually cleared almost all the confusions by reading many official Go documentation and all kinds of Go articles on Internet, and by finding answers from some Go forums and the Go project issue tracker.

I spent about one year clearing the confusions. During the period, from time to time, once I had cleared most confusions on a topic and regained the confidence on explaining that topic, I wrote one blog article for that topic. In the end, I had written about twenty Go articles. And I had collected more Go details than before. It was the time to restart the plan of writing a Go book.

I wrote another ten basic tutorial articles and twenty more articles on all kinds of other Go topics. So now **Go 101** has about 50 articles.

What were your ever confusions?

Some of the confusions were a few syntax and semantics design details, some of them involved values of certain kinds of types (mainly slices, interfaces and channels), and a few of them were

related to standard package APIs.

What are the causes of your ever confusions do you think?

Thinking Go is easy to master is considered harmful. Holding such opinion (thinking Go is easy to master) will make you understand Go shallowly and prevent you from mastering Go.

Go is a feature rich language. Its syntax set is surely not large, but we also can't say it is small. Some syntax and semantics designs in Go are straightforward, some are a little counter-intuitive or inconsistent with others. There are several trade-offs in Go syntax and semantics designs. A programmer needs certain Go programming experiences to comprehend the trade-offs.

Go provides several first-citizen non-essential kinds of types. Some encapsulations are made in implementing these types to hide the internal structures of these types. On one hand, the encapsulations bring much convenience to Go programming. On the other hand, the encapsulations make some obstacles to understand the behaviors of values of these types more deeply.

Many official and unofficial Go tutorials are very simple and only cover the general use cases by ignoring many details. This may be good to encourage new Go programmers to learn and use Go. On the other hand, this also makes many Go programmers overconfident on the extent of their Go knowledge.

Several functions and types declared in some standard packages do not have detailed explanations. This is understandable, for many details are so subtle that it is hard to find proper wordings to explain them clearly. Saying a few accurate words is better than saying lots of words with inaccuracies. But this really leaves some confusions for the package users.

So do you think simplicity is not a selling point of Go?

I think, at least, simplicity is not a main selling point of Go. After all, there are several other languages simpler than Go. On the other hand, Go, as a feature rich language, is also not a complicated language. A new Go programmer with right attitudes can master Go programming in one year.

Then what are the selling points of Go do you think?

Personally, I think the fact that, as a static language, Go is flexible as many dynamic script languages is the main selling point of Go language.

Memory saving, fast program warming-up, fast code execution speed and fast compilations combined is another main selling point of Go. Although this is a common selling point of many C family languages. But for web development area, seldom languages own the four characteristics at the same time. In fact, this is the main reason why I switched to Go from Java for web development.

Built-in concurrent programming support is also a selling point of Go, though personally I don't think it is the main selling point of Go.

Great code readability is another important selling point of Go. I feel readability is the most important factor considered in designing Go.

Great cross-platform support is also a selling point of Go, though this selling point is not much unique nowadays.

A stable core design and development team and an active community together can also be viewed as a selling point of Go.

What does Go 101 do to clear these confusions?

Go 101 tries to clear many confusions by doing the followings.

1. Emphasizes on basic concepts and terminologies. Without understanding these basic concepts and terminologies, it is hard to fully understand many rules and high level concepts.
2. Adds the **value part** terminology and use one special article to explain value parts. This article uncovers the underlying structures of some kinds of types, so that Go programmers could understand Go values of those types more deeply. I think knowing a little possible underlying implementations is very helpful to clear some confusions about all kinds of Go values.
3. Explains memory blocks in detail. Knowing the relations between Go values and memory blocks is very helpful to understand how a garbage collector works and how to avoid memory leaking.
4. Views interface values as boxes for wrapping non-interface values. I found thinking interface values as value boxes is very helpful to clear many interface related confusions.
5. Clarifies several ambiguities which exist in the Go specification, such as embedding rules, promoted method value evaluation, and panic/recover mechanism.
6. Makes several summary articles and special topic articles by aggregating many knowledge points and details, which would save Go programmers much learning time.

Is there anything else worth mentioning?

This book doesn't touch custom generics in depth. Please read the [Go Generics 101](#) book to learn about custom generics.

In addition, type parameter types (used frequently in custom generics) are deliberately ignored in the descriptions of conversion, assignability and comparison rules. In other words, this book doesn't consider the situations in which custom generics are involved.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Acknowledgments

Firstly, thanks to the whole Go community. Without an active and responsive community, this book couldn't have been finished.

I'd like to give special thanks to Ian Lance Taylor. Ian kindly answered my countless boring questions on go-nuts group and the Go project issue tracker. Ian's answers helped me clear many of my ever confusions in using Go.

I want to give thanks to the following people who participated in my question threads on go-nuts and go-dev groups: Axel Wagner, Keith Randall, Russ Cox, Robert Griesemer, Jan Mercl, Konstantin Khomoutov, Brad Fitzpatrick, Alan Donovan, Minux Ma, Dave Cheney, Volker Dobler, Caleb Spare, Matt Harden, Roger Peppe, Michael Jones, peterGo, Pietro Gagliardi, Paul Jolly, Oleg Sidorov, and Rob 'Commander' Pike, etc.

Also I learned so much from the Go project issue tracker. I give my thanks to the following people who are active on the Go project issue tracker, including: Ian Lance Taylor, Robert Griesemer, Brad Fitzpatrick, Russ Cox, Matthew Dempsky, Keith Randall, Bryan C. Mills, Joe Tsai, Minux Ma, Josh Bleecher Snyder, Axel Wagner, Daniel Martí, Dave Cheney, Austin Clements, Andrew Bonventre, Damian Gryski, Alberto Donizetti, Emmanuel T Odeke, Filippo Valsorda, Dominik Honnef, and Rob Pike, etc.

I thank Martin Möhrmann for pointing out a mistake in one of my Go articles on reddit.

I also would like to thank all gophers who ever made influences on the Go 101 book, be it directly or indirectly, intentionally or unintentionally.

I thank all Go 101 contributors for improving Go 101 articles, including: ipinak, Amir Khazaie, Ziqi Zhao, Artur Kirillov, Arinto Murdopo, Andreas Pannewitz, Jason-Huang, Joel Rebello, Julia Wong, Wenbin Zhang, Farid Gh, Valentin Deleplace, nofrish, Peter Matseykanets, KimMachineGun, Yoshinori Kawasaki, mooncaker816, Sangwon Seo, Shaun (wrfly), Stephan Renatus, Yang Shu, stemar94, berkant (yadkit), Thomas Bower, Patryk Małek, kucharskim, Rodrigue Koffi, Jhuliano Skittberg Moreno, youmoo, ahadc, Kevin, jojo (hjb912), Gordon Wang, Steve Zhang, cerlestes, Bai Kai, Gleb Sozinov, Jake Montgomery, Erik Dubbelboer, Afriza N. Arief, Maxim Neverov, Asim Himani, sulinehk, Grachev Mikhail, halochou, HaiwenZhu, Alisha Sonawalla, SheldonZhong, wens.wq, xamenyap, Emmanuel Hayford, Sam Berry, binderclip, Oleg Atamanenko, Amargin, Tony Bai, lanastasov, liukaifei, Abirdcfly, Tahir Raza, Bububuger, lniwn, GFZRZK, bhakiyalimuthu, 黎显图, Crazy Yang, Ilya Markin, Yang Yang, sdjdd, N0mansky, Samir Ettali, GeXiao, ZXH, bestgopher, 9r0k, Streppel, wieghx, SourceLink, Oleh, Ibrahim Mohammed, shu-ming, Genaro-

§1. Acknowledgments

Chris, Ying-Han Chen, Sina-Ghaderi, Kaijie Chen, huydang284, Soule Bah, Alex Pashkov, Dai Jie, Souhail, Phan Phu Thanh, Frank Meyer, hhoke, V0idk, LiuHe, Rathawut Lertsuksakda, ffmiylo, Arun Kumar, Bo Tao, KONY, Michael Winser, eNV25, opennota, luozhiyun, Huang Chao, Guilherme C. Matuella, Boy Baukema, Cuong Manh Le, chenxu zhao, Roman Ilchyshyn, Andy Wong, Luo Peng, Vitaly Zdanovich, Harsh Mangalam, Djim Molenkamp, ch0ngsheng, Alexandre Primak, cortes-, Yussif Mohammed, Wei-Cheng Yeh, W.T. Chang, darkCavalier11, jordancurve, 1Mark, Caio Leonhardt, Aleksandr Shalimov, DashJay, Gabriel Crispino, Ammar Kasem, nekidb, ursatong, etc.

I'm sorry if I forgot mentioning somebody in above lists. There are so many kind and creative gophers in the Go community that I must have missed out on someone.

Thanks to the authors of the [Bootstrap framework](#), [jQuery library](#), [code prettify](#) and [prism syntax highlighting](#) JavaScript packages, for supporting the go101.org website. Thanks to the authors of the [go-epub](#) and [calibre](#) projects for building Go 101 ebooks.

Thanks to Adam Chalkley for a good suggestion in making the ebooks.

Special thanks to [Renee French](#) and Rob Pike. The vivid picture used in the covers of the digital and paper versions of this book is copied from [one of from Rob's slides](#). Renee is the author of the lovely gopher iconic mascots used in the picture.

(The **Go 101** book is still being improved frequently from time to time. Please visit [go101.org](#) or follow [@go100and1](#) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit [tapirgames.com](#) to get more information about these games. Hope you enjoy them.)

An Introduction of Go

Go is a compiled and static typed programming language born from Google. Many of the core Go design and tool development team members have many years of experience in the field of programming language research.

Go has many features. Some are unique, some are borrowed from other programming languages:

- built-in concurrent programming support
 - goroutines (green threads) and start new goroutines easily.
 - channels (based on CSP model) and select mechanisms to do synchronizations between goroutines.
- the container types `map` and `slice` are first-class citizens.
- polymorphism through interfaces.
- value boxing and reflection through interfaces.
- pointers.
- function closures.
- methods.
- deferred function calls.
- type embedding.
- type deduction.
- memory safety.
- automatic garbage collection.
- great cross-platform compatibility.
- custom generics (since Go 1.18).

Besides the above features, further highlights are:

- The syntax of Go is deliberately designed to be simple, clean, and similar to other popular programming languages. This makes Go programming easy to pick up.
- Go comes with a great set of standard code packages which provide all kinds of common functionalities. Most of the packages are cross-platform.
- Go also has an active community, and there are [plenty of high quality third party Go packages and projects](#) to import and use.

Go programmers are often called gophers.

In fact, although Go is a compiled and static typed programming language, Go also has many features which are usually only available in dynamic script languages. It is hard to combine these

two kinds into one language, but Go did it. In other words, Go owns both the strictness of static languages and the flexibility of dynamic languages. I can't say there are not any compromises between the two, but the effect of the compromises is much weaker than the benefits of the combination in Go.

Readability is an important factor which affects the design of Go heavily. It is not hard for a gopher to understand the Go code written by other gophers.

Currently, the most popular Go compiler is written in Go and maintained by the Go design team.

Later we shall call it the standard Go compiler, or `gc` (an abbreviation for Go compiler, not for garbage collection GC). The Go design team also maintains a second Go compiler, `gccgo`.

Nowadays its use is less popular than `gc`, but it always serves as a reference, and both compilers are in active development. As of now the Go team focuses on the improvement of `gc`.

`gc` is provided in Go Toolchain (a set of tools for Go development maintained by Go team). Go Toolchain 1.0 was released in March, 2012. The version of Go is consistent with the version of Go Toolchain. There were/are two major versions released each year.

Since the release of Go 1.0, the syntax of Go has changed a little, but there were/are many improvements for the tools in Go Toolchain, from version to version, especially for `gc`. For example, noticeable lags caused by garbage collecting is a common criticism for languages with automatic memory management. But since Go 1.8, improvements made for the concurrent garbage collection implementation in `gc` basically eliminated the lag problem.

`gc` supports cross-platform compilation. For example, we can build a Windows executable on a Linux OS, and vice versa.

Programs written in go language mostly compile very fast. Compilation time is an important factor for the happiness in development. Short build time is one reason why many programmers like programming with Go.

Advantages of Go executables are:

- small memory footprint
- fast code execution
- short warm-up duration (so great deployment experience)

Some other compiled languages, such as C/C++/Rust may also have these three advantages (and they may have their respective advantages compared to Go), but they lack three important characteristics of Go:

- fast compilation results in happy local development experience and short deployment iteration cycles
- flexible, like dynamic languages
- built-in concurrent programming support

All the above advantages combined make Go an outstanding language and a good choice for many kinds of projects. Currently, Go is popularly used in network, system tools, database development and block chain development areas. With the introduction of custom generics in Go 1.18, it is expected that Go will be used more and more in some other areas, such as gui/game, big data and AI.

Finally, Go is not perfect in all aspects. There are certain trade-offs in Go design. And Go really has some shortcomings. For example, Go doesn't support arbitrary immutable values now, which leads to that many values which are not intended to be modified in standard packages are declared as variables. This is a potential security weak point for some Go programs.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

The Go Toolchain

Currently, the tools in the official Go Toolchain (called Go Toolchain later) are the most used tools to develop Go projects. In Go 101 article series, all examples are compiled and verified with the standard Go compiler provided in Go Toolchain.

This article will introduce how to setup the Go development environment and how to use the `go` command provided in Go Toolchain. Some other tools out of Go Toolchain will also be introduced.

Install Go Toolchain

Please [download ↗](#) Go Toolchain and install it by following the instructions shown in the download page.

We can also use the tool [GoTV](<https://go101.org/apps-and-libs/gotv.html>) to manage installations of multiple Go toolchain versions and use them harmoniously and conveniently.

The version of a Go Toolchain release is consistent with the highest Go language version the release supports. For example, Go Toolchain 1.21.x supports all Go language versions from 1.0 to Go 1.21.

The path to the `bin` subfolder in Go Toolchain installation root path must be put in the `PATH` environment variable to execute the tools (through the `go` command) provided in Go Toolchain without inputting their full paths. If your Go Toolchain is installed through an installer or with a package manager, the path to the `bin` subfolder may have been already set in the `PATH` environment variable automatically for you.

The recent Go Toolchain versions support a feature called modules to manage project dependencies. The feature was introduced experimentally in version 1.11 and has been enabled by default since version 1.16.

There is an environment variable, `GOPATH`, we should be aware of, though we don't need to care too much about it. It is defaulted to the path to the `go` folder under the home directory of the current user. The `GOPATH` environment variable may list multiple paths if it is specified manually. Later, when the `GOPATH` folder is mentioned, it means the folder at the first path in the `GOPATH` environment variable.

- The `pkg` subfolder under the `GOPATH` folder is used to store cached versions of Go modules (a Go module is a collection of Go packages) depended by local Go projects.

- There is a `GOBIN` environment variable which controls where the Go program binary files generated by the `go install` subcommand will be stored. The value of the `GOBIN` environment variable is defaulted to the path to `bin` subfolder under the `GOPATH` folder. The `GOBIN` path should be set in the `PATH` environment variable if you would run the generated Go program binary files without specifying their full paths.

The Simplest Go Program

Let's write a simple example and learn how to run simple Go programs.

The following program is the simplest Go program.

```
1| package main
2|
3| func main() {
4| }
```

The words `package` and `func` are two keywords. The two `main` words are two identifiers.

Keywords and identifiers are introduced in [a coming article](#) (§5).

The first line `package main` specifies the package name (`main` here) of the containing source file.

The second line is a blank line for better readability.

The remaining code declares a function which is also called `main`. This `main` function in a `main` package specifies the entry point of a program. (Note that some other user code might be executed before the `main` function gets invoked.)

Run Go Programs

Go Toolchain requires that Go source code file to have the extension `.go`. Here, we assume the above source code is saved in a file named `simplest-go-program.go`.

Open a terminal and change the current directory to the directory which contains the above source file, then run

```
$ go run simplest-go-program.go
```

Nothing is output? Yes, this program outputs nothing.

If there are some syntax errors in the source code, then these errors will be reported as compilation errors.

If multiple source files are in the `main` package of a program, then we should run the program with the following command

```
$ go run .
```

Note,

- the `go run` command is not recommended to compile and run large Go projects. It is just a convenient way to run simple Go programs, like the ones in the Go 101 articles. For large Go projects, please use the commands `go build` or `go install` to build then run executable binary files instead.
- each formal Go project supporting Go modules needs a `go.mod` file located in the root folder of that project. The `go.mod` file can be generated by the `go mod init` subcommand (see below).
- source files starting with `_` or `.` are ignored by Go Toolchain.

More go Subcommands

The three commands, `go run`, `go build` and `go install`, only output code syntax errors (if any). They don't (try to) output code warnings (a.k.a., possible code logic mistakes). We can use the `go vet` command to check and report such warnings.

We can (and should often) use the `go fmt` command to format Go source code with a consistent coding style.

We can use the `go test` command to run tests and benchmarks.

We can use the `go doc` command to view Go documentation in terminal windows.

It is highly recommended to let your Go projects support the Go modules feature to ease dependency management. For projects supporting Go modules,

- the `go mod init example.com/myproject` command is used to generate a `go.mod` file in the current directory, which will be viewed as the root directory of a module called `example.com/myproject`. The `go.mod` file will be used to record module dependencies. We can edit the `go.mod` file manually or let the `go` subcommands to update it.
- the `go mod tidy` command is used to add missing module dependencies into and remove unused module dependencies from the `go.mod` file by analyzing all the source code of the

current project.

- the `go get` command is used to add/upgrade/downgrade/remove a single dependency. This command is used less frequently than the `go mod tidy` command.

Since Go Toolchain 1.16, we can run `go install example.com/program@latest` to install the latest version of a third-party Go program (into the `GOBIN` folder). Before Go Toolchain 1.16, the corresponding command is `go get -u example.com/program`, which has been deprecated now.

We can use the `go help aSubCommand` command to view the help message for a specified sub command.

Run the `go` command without any arguments to show the supported subcommands.

The Go 101 article series will not explain much more on how to use the tools provided in Go Toolchain. Please read the [official documentation ↗](#) for details.

View Go Package Docs in Browsers

We could use the docs and source view tool [**Golds** ↗](#) (`go101.org/golds`) to locally view the docs of a specified Go project. This is not an official tool. It is developed by me (Tapir Liu). **Golds** supports rich code reading experiences (such as listing type implementation relations). After installing **Golds**, we can run `golds std ./...` to view the docs of standard packages and packages in the current folder.

For official Go documentations, please visit [go.dev ↗](#).

(The **Go 101** book is still being improved frequently from time to time. Please visit [go101.org ↗](#) or follow [@go100and1 ↗](#) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit [tapirgames.com ↗](#) to get more information about these games. Hope you enjoy them.)

Introduction to Source Code Elements

Go is known for its simple and clean syntax. This article introduces the common source code elements in programming through a simple example. This will help new gophers (Go programmers) get a basic idea of the usage of Go elements.

Programming and Source Code Elements

Programming can be viewed as manipulating operations in all kinds of ways to reach certain goals. Operations write data to and read data from hardware devices to complete tasks. For modern computers, elemental operations are low-level CPU and GPU instructions. Common hardware devices include memory, disk, network card, graphics card, monitor, keyboard and mouse, etc.

Programming by manipulating low-level instructions directly is tedious and error-prone. High-level programming languages make some encapsulations for low-level operations, and make some abstracts for data, to make programming more intuitive and human-friendly.

In popular high-level programming languages, operations are mainly achieved by calling **functions** and using **operators**. Most popular high-level programming languages support several kinds of conditional and loop **control flows**, we can think of them as special operations. The syntax of these control flows is close to human language so that the code written by programmers is easy to understand.

Data is abstracted as **types** and **values** in most high-level programming languages. Types can be viewed as value templates, and values can be viewed as type instances. Most languages support several built-in types, and also support custom types. The type system of a programming language is the spirit of the language.

There may be a large number of values used in programming. Some of them can be represented with their **literals** (text representations) directly, but others can't. To make programming flexible and less error-prone, many values are named. Such values include **variables** and named **constants**.

Named functions, named values (including variables and named constants), defined types and type alias are called **code elements**. The names of code elements must be [identifiers \(§5\)](#). Package names and package import names shall also be identifiers.

High-level programming code will be translated to low-level CPU instructions by compilers to get executed. To help compilers parse high-level programming code, many words are reserved to prevent them from being used as identifiers. Such words are called [keywords \(§5\)](#).

Many modern high-level programming languages use **packages** to organize code. A package must **import** another package to use the exported (public) code elements in the other package. Package names and package import names shall also be identifiers.

Although the code written in high-level programming languages is more understandable than low-level machine languages, we still need some comments for some code to explain the logic. The example program in the next section contains many comments.

A Simple Go Demo Program

Let's view a short Go demo program to know all kinds of code elements in Go. Like some other languages, in Go, line comments start with `//`, and each block comment is enclosed in a pair of `/*` and `*/`.

Below is the demo Go program. Please read the comments for explanations. More explanations are following the program.

```

1| package main // specify the source file's package
2|
3| import "math/rand" // import a standard package
4|
5| const MaxRnd = 16 // a named constant declaration
6|
7| // A function declaration
8| /*
9| StatRandomNumbers produces a certain number of
10| non-negative random integers which are less than
11| MaxRnd, then counts and returns the numbers of
12| small and large ones among the produced randoms.
13| n specifies how many randoms to be produced.
14| */
15| func StatRandomNumbers(n int) (int, int) {
16|     // Declare two variables (both as 0).
17|     var a, b int
18|     // A for-loop control flow.
19|     for i := 0; i < n; i++ {
20|         // An if-else control flow.
21|         if rand.Intn(MaxRnd) < MaxRnd/2 {
22|             a = a + 1
23|         } else {
24|             b++ // same as: b = b + 1
25|         }
26|     }

```

```

27|     return a, b // this function return two results
28|
29|
30| // "main" function is the entry function of a program.
31| func main() {
32|     var num = 100
33|     // Call the declared StatRandomNumbers function.
34|     x, y := StatRandomNumbers(num)
35|     // Call two built-in functions (print and println).
36|     print("Result: ", x, " + ", y, " = ", num, "? ")
37|     println(x+y == num)
38|

```

Save above source code to a file named `basic-code-element-demo.go` and run this program by:

```
$ go run basic-code-element-demo.go
Result: 46 + 54 = 100? true
```

In the above program, `package`, `import`, `const`, `func`, `var`, `for`, `if`, `else`, and `return` are all keywords. Most other words in the program are identifiers. Please read [keywords and identifiers](#) (§5) for more information about keywords and identifiers.

The four `int` words at line 15 and line 17 denote the built-in `int` type, one of many kinds of integer types in Go. The `16` at line 5, `0` at line 19, `1` at line 22 and `100` at line 32 are some integer literals. The `"Result: "` at line 36 is a string literal. Please read [basic types and their value literals](#) (§6) for more information about above built-in basic types and their value literals. Some other types (composite types) will be introduced later in other articles.

Line 22 is an assignment. Line 5 declares a named constant, `MaxRnd`. Line 17 and line 32 declare three variables, with the standard variable declaration form. Variables `i` at line 19, `x` and `y` at line 34 are declared with the short variable declaration form. We have specified the type for variables `a` and `b` as `int`. Go compiler will deduce that the types of `i`, `num`, `x` and `y` are all `int`, because they are initialized with integer literals. Please read [constants and variables](#) (§7) for more information about untyped values, type deduction, value assignments, and how to declare variables and named constants.

There are many operators used in the program, such as the less-than comparison operator `<` at line 19 and 21, the equal-to operator `==` at line 37, and the addition operator `+` at line 22 and line 37. Yes, `+` at line 36 is not an operator, it is one character in a string literal. The values involved in an operator operation are called operands. Please read [common operators](#) (§8) for more information. More operators will be introduced in other articles later.

At line 36 and line 37, two built-in functions, `print` and `println`, are called. A custom function `StatRandomNumbers` is declared from line 15 to line 28, and is called at line 34. Line 21 also calls a function, `Intn`, which is a function declared in the `math/rand` standard package. A function call is a function operation. The input values used in a function call are called arguments. Please read [function declarations and calls](#) (§9) for more information.

(Note, the built-in `print` and `println` functions are not recommended to be used in formal Go programming. The corresponding functions in the `fmt` standard packages should be used instead in formal Go projects. In Go 101, the two functions are only used in the several starting articles.)

Line 1 specifies the package name of the current source file. The `main` entry function must be declared in a package which is also called `main`. Line 3 imports a package, the `math/rand` standard code package. Its import name is `rand`. The function `Intn` declared in this standard package is called at line 21. Please read [code packages and package imports](#) (§10) for more information about how to organize code packages and import packages.

The article [expressions, statements and simple statements](#) (§11) will introduce what are expressions and statements. In particular, all kinds of simple statements, which are special statements, are listed. Some portions of all kinds of control flows must be simple statements, and some portions must be expressions.

In the `StatRandomNumbers` function body, two control flows are used. One is a `for` loop control flow, which nests the other one, an `if-else` conditional control flow. Please read [basic control flows](#) (§12) for more information about all kinds of basic control flows. Some other special control flows will be introduced in other articles later.

Blank lines have been used in the above program to improve the readability of the code. And as this program is for code elements introduction purpose, there are many comments in it. Except the documentation comment for the `StatRandomNumbers` function, other comments are for demonstration purpose only. We should try to make code self-explanatory and only use necessary comments in formal projects.

About Line Breaks

Like many other languages, Go also uses a pair of braces (`{` and `}`) to form an explicit code block. However, in Go programming, coding style can't be arbitrary. For example, many of the starting curly braces (`{`) can't be put on the next line. If we modify the `StatRandomNumbers` function declaration in the above program as the following, the program will fail to compile.

```

1| func StatRandomNumbers(n int) (int, int)
2| { // syntax error
3|     var a, b int
4|     for i := 0; i < n; i++
5|     { // syntax error
6|         if rand.Intn(MaxRnd) < MaxRnd/2
7|             { // syntax error
8|                 a = a + 1
9|             } else {
10|                 b++
11|             }
12|         }
13|     return a, b
14| }
```

Some programmers may not like the line break restrictions. But the restrictions have two benefits:

1. they make code compilations become faster.
2. they make the coding styles written by different gophers look similar, so that it is more easily for gophers to read and understand the code written by other gophers.

We can learn more about line break rules in [a later article](#) (§28). At present, we should avoid putting a starting curly brace on a new line. In other words, generally, the first non-blank character of a code line should not be the starting curly brace character. (But, please remember, this is not a universal rule.)

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Keywords and Identifiers in Go

This article will introduce keywords and identifiers in Go.

Keywords

Keywords are the special words which help compilers understand and parse user code.

Up to now (Go 1.21), Go has only 25 keywords:

1	<code>break</code>	<code>default</code>	<code>func</code>	<code>interface</code>	<code>select</code>
2	<code>case</code>	<code>defer</code>	<code>go</code>	<code>map</code>	<code>struct</code>
3	<code>chan</code>	<code>else</code>	<code>goto</code>	<code>package</code>	<code>switch</code>
4	<code>const</code>	<code>fallthrough</code>	<code>if</code>	<code>range</code>	<code>type</code>
5	<code>continue</code>	<code>for</code>	<code>import</code>	<code>return</code>	<code>var</code>

They can be categorized as four groups:

- `const`, `func`, `import`, `package`, `type` and `var` are used to declare all kinds of code elements in Go programs.
- `chan`, `interface`, `map` and `struct` are used as parts in some composite type denotations.
- `break`, `case`, `continue`, `default`, `else`, `fallthrough`, `for`, `goto`, `if`, `range`, `return`, `select` and `switch` are used to control flow of code.
- `defer` and `go` are also control flow keywords, but in other specific manners. They modify function calls, which we'll talk about in [this article](#) (§13).

These keywords will be explained in details in other articles.

Identifiers

An identifier is a token which must be composed of Unicode letters, Unicode digits (Number category *Nd* in Unicode Standard 8.0) and `_` (underscore), and start with either an Unicode letter or `_`. Here,

- Unicode letters mean the characters defined in the Letter categories *Lu*, *Ll*, *Lt*, *Lm*, or *Lo* of [The Unicode Standard 8.0](#).
- Unicode digits mean the characters defined in the Number category *Nd* of The Unicode Standard 8.0.

keywords can not be used as identifiers.

Identifier `_` is a special identifier, it is called **blank identifier**.

Later we will learn that all names of types, variables, constants, labels, package names and package import names must be identifiers.

An identifier starting with an [Unicode upper case letter ↗](#) is called an **exported identifier**. The word **exported** can be interpreted as **public** in many other languages. The identifiers which don't start with an Unicode upper case letter are called non-exported identifiers. The word **non-exported** can be interpreted as **private** in many other languages. Currently (Go 1.21), eastern characters are viewed as non-exported letters. Sometimes, non-exported identifiers are also called unexported identifiers.

Here are some legal exported identifiers:

```
1| Player_9
2| DoSomething
3| VERSION
4| Go
5| Π
```

Here are some legal non-exported identifiers:

```
1| _
2| _status
3| memStat
4| book
5| π
6| 一个类型
7| 변수
8| エラー
```

And here are some tokens which are illegal to be used as identifiers:

```
1| // Starting with a Unicode digit.
2| 123
3| 3apples
4|
5| // Containing Unicode characters not
6| // satisfying the requirements.
7| a.b
8| *ptr
9| $name
10| a@b.c
```

```
11|  
12| // These are keywords.  
13| type  
14| range
```

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Basic Types and Basic Value Literals

Types can be viewed as value templates, and values can be viewed as type instances. This article will introduce the built-in basic types and their value literals in Go. Composite types will not get introduced in this article.

Built-in Basic Types in Go

Go supports following built-in basic types:

- one boolean built-in boolean type: `bool`.
- 11 built-in integer numeric types (basic integer types): `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `int`, `uint`, and `uintptr`.
- two built-in floating-point numeric types: `float32` and `float64`.
- two built-in complex numeric types: `complex64` and `complex128`.
- one built-in string type: `string`.

Each of the 17 built-in basic types belongs to one different kind of type in Go. We can use the above built-in types in code without importing any packages, though all the names of these types are non-exported identifiers.

15 of the 17 built-in basic types are numeric types. Numeric types include integer types, floating-point types and complex types.

Go also support two built-in type aliases,

- `byte` is a built-in alias of `uint8`. We can view `byte` and `uint8` as the same type.
- `rune` is a built-in alias of `int32`. We can view `rune` and `int32` as the same type.

The integer types whose names starting with an `u` are unsigned types. Values of unsigned types are always non-negative. The number in the name of a type means how many binary bits a value of the type will occupy in memory at run time. For example, every value of the `uint8` occupies 8 bits in memory. So the largest `uint8` value is 255 (2^8-1), the largest `int8` value is 127 (2^7-1), and the smallest `int8` value is -128 (-2^7).

If a value occupies **N** bits in memory, we say the size of the value is **N** bits. The sizes of all values of a type are always the same, so value sizes are often called as type sizes.

We often measure the size of a value based on the number of bytes it occupies in memory. One byte contains 8 bits. So the size of the `uint32` type is four bytes.

The size of `uintptr`, `int` and `uint` values in memory are implementation-specific. Generally, the size of `int` and `uint` values are 4 on 32-bit architectures, and 8 on 64-bit architectures. The size of `uintptr` value must be large enough to store the uninterpreted bits of any memory address.

The real and imaginary parts of a `complex64` value are both `float32` values, and the real and imaginary parts of a `complex128` value are both `float64` values.

In memory, all floating-point numeric values in Go are stored in [IEEE-754 format](#).

A boolean value represents a truth. There are only two possible boolean values in memory, they are denoted by the two predeclared named constants, `false` and `true`. Name constants will be introduced in [the next article](#) (§7).

In logic, a string value denotes a piece of text. In memory, a string value stores a sequence of bytes, which is the UTF-8 encoding representation of the piece of text denoted by the string value. We can learn more facts on strings from the article [strings in Go](#) (§19) later.

Although there is only one built-in type for each of boolean and string types, we can define custom boolean and string types for the built-in boolean and string types. So there can be many boolean and string types. The same is for any kinds of numeric types. The following are some type declaration examples. In these declarations, the word `type` is a keyword.

```

1| /* Some type definition declarations */
2|
3| // status and bool are two different types.
4| type status bool
5| // MyString and string are two different types.
6| type MyString string
7| // Id and uint64 are two different types.
8| type Id uint64
9| // real and float32 are two different types.
10| type real float32
11|
12| /* Some type alias declarations */
13|
14| // boolean and bool denote the same type.
15| type boolean = bool
16| // Text and string denote the same type.
17| type Text = string
18| // U8, uint8 and byte denote the same type.
19| type U8 = uint8

```

```
20| // char, rune and int32 denote the same type.  
21| type char = rune
```

We can call the custom `real` type defined above and the built-in `float32` type both as `float32` types. Note, the second `float32` word in the last sentence is a general reference, whereas the first one is a specified reference. Similarly, `MyString` and `string` are both string types, `status` and `bool` are both bool types, etc.

We can learn more on custom types in the article [Go type system overview](#) (§14) later.

Zero Values

Each type has a zero value. The zero value of a type can be viewed as the default value of the type.

- The zero value of a boolean type is `false`.
- The zero value of a numeric type is zero, though zeros of different numeric types may have different sizes in memory.
- The zero value of a string type is an empty string.

Basic Value Literals

A literal of a value is a text representation of the value in code. A value may have many literals. The literals denoting values of basic types are called basic value literals.

Boolean value literals

Go specification doesn't define boolean literals. However, in general programming, we can view the two predeclared identifiers, `false` and `true`, as boolean literals. But we should know that the two are not literals in the strict sense.

As mentioned above, zero values of boolean types are denoted with the predeclared `false` constant.

Integer value literals

There are four integer value literal forms, the decimal (base 10) form, the octal (base 8) form, the hex (base 16) form and the binary form (base 2). For example, the following four integer literals all denote `15` in decimal.

```

0xF // the hex form (starts with a "0x" or "0X")
0XF

017 // the octal form (starts with a "0", "0o" or "0O")
0o17
0017

0b1111 // the binary form (starts with a "0b" or "0B")
0B1111

15 // the decimal form (starts without a "0")

```

(Note: the binary form and the octal form starting with 0o or 0O are supported since Go 1.13.)

The following program will print two `true` texts.

```

1| package main
2|
3| func main() {
4|     println(15 == 017) // true
5|     println(15 == 0xF) // true
6| }

```

Note, the two `==` are the equal-to comparison operator, which will be introduced in [common operators](#) (§8).

Generally, zero values of integer types are denoted as `0` in literal, though there are many other legal literals for integer zero values, such as `00` and `0x0`. In fact, the zero value literals introduced in the current article for other kinds of numeric types can also represent the zero value of any integer type.

Floating-point value literals

A decimal floating-point value literal may contain a decimal integer part, a decimal point, a decimal fractional part, and an integer exponent part (10-based). Such an integer exponent part starts with a letter `e` or `E` and suffices with a decimal integer literal (`xEn` is equivalent to `x` is multiplied by 10^n , and `xE-n` is equivalent to `x` is divided by 10^n). Some examples:

```

1.23
01.23 // == 1.23
.23
1.
// An "e" or "E" starts the exponent part (10-based).
1.23e2 // == 123.0

```

```
123E2    // == 12300.0
123.E+2  // == 12300.0
1e-1     // == 0.1
.1e0     // == 0.1
0010e-2 // == 0.1
0e+5     // == 0.0
```

Since Go 1.13, Go also supports another floating point literal form: hexadecimal floating point literal form.

- A hexadecimal floating point literal must end with 2-based exponent part, which starts with a letter `p` or `P` and suffixes with a decimal integer literal (`yPn` is equivalent to `y` is multiplied by 2^n , and `yP-n` is equivalent to `y` is divided by 2^n).
- Same as hex integer literals, a hexadecimal floating point literal also must start with `0x` or `0X`. Different from hex integer literals, a hexadecimal floating point literal may contain a decimal point and a decimal fractional part.

The followings are some legal hexadecimal floating point literals:

```
0x1p-2    // == 1.0/4 == 0.25
0x2.p10   // == 2.0 * 1024 == 2048.0
0x1.Fp+0  // == 1+15.0/16 == 1.9375
0X.8p1    // == 8.0/16 * 2 == 1.0
0X1FFFFP-16 // == 0.1249847412109375
```

However, the following ones are illegal:

```
0x.p1    // mantissa has no digits
1p-2    // p exponent requires hexadecimal mantissa
0x1.5e-2 // hexadecimal mantissa requires p exponent
```

Note: the following literal is legal, but it is not a floating point literal. It is a subtraction arithmetic expression actually. The `e` in it means 14 in decimal. `0x15e` is a hex integer literal, `-` is the subtraction operator, and `2` is a decimal integer literal. (Arithmetic operators will be introduced in the article [common operators](#) (§8).)

```
0x15e-2 // == 0x15e - 2 // a subtraction expression
```

The standard literals for zero value of floating-point types are `0.0`, though there are many other legal literals, such as `0.`, `.0`, `0e0`, `0x0p0`, etc. In fact, the zero value literals introduced in the current article for other kinds of numeric types can also represent the zero value of any floating-point type.

Imaginary value literals

An imaginary literal consists of a floating-point or integer literal and a lower-case letter `i`.

Examples:

```
1.23i
1.i
.23i
123i
0123i    // == 123i (for backward-compatibility. See below.)
1.23E2i // == 123i
1e-1i
011i    // == 11i (for backward-compatibility. See below.)
00011i // == 11i (for backward-compatibility. See below.)
// The following lines only compile okay since Go 1.13.
0o11i    // == 9i
0x11i    // == 17i
0b11i    // == 3i
0X.8p-0i // == 0.5i
```

Note, before Go 1.13, in an imaginary literal, the letter `i` can only be prefixed with a floating-point literal. To be compatible with the older versions, since Go 1.13, the integer literals appearing as octal integer forms not starting with `0o` and `0O` are still viewed as floating-point literals, such as `011i`, `0123i` and `00011i` in the above example.

Imaginary literals are used to represent the imaginary parts of complex values. Here are some literals to denote complex values:

```
1 + 2i      // == 1.0 + 2.0i
1. - .1i    // == 1.0 + -0.1i
1.23i - 7.89 // == -7.89 + 1.23i
1.23i      // == 0.0 + 1.23i
```

The standard literals for zero values of complex types are `0.0+0.0i`, though there are many other legal literals, such as `0i`, `.0i`, `0+0i`, etc. In fact, the zero value literals introduced in the current article for other kinds of numeric types can also represent the zero value of any complex type.

Use `_` in numeric literals for better readability

Since Go 1.13, underscores `_` can appear in integer, floating-point and imaginary literals as digit separators to enhance code readability. But please note, in a numeric literal,

- any `_` is not allowed to be used as the first or the last character of the literal,
- the two sides of any `_` must be either literal prefixes (such as `0X`) or legal digit characters.

Some legal and illegal numeric literals which contain underscores:

```
// Legal ones:
6_9          // == 69
0_33_77_22   // == 0337722
0x_Bad_Face // == 0xBAdFAcE
0X_1F_FFP-16 // == 0X1FFFFP-16
0b1011_0111 + 0xA_B.Fp2i

// Illegal ones:
_69          // _ can't appear as the first character
69_          // _ can't appear as the last character
6__9         // one side of _ is an illegal character
0_xBadFace  // "x" is not a legal octal digit
1_.5         // "." is not a legal octal digit
1._5         // "." is not a legal octal digit
```

Rune value literals

Rune types, including custom defined rune types and the built-in `rune` type (a.k.a., `int32` type), are special integer types, so all rune values can be denoted by the integer literals introduced above. On the other hand, many values of all kinds of integer types can also be represented by rune literals introduced below in the current subsection.

A rune value is intended to store a Unicode code point. Generally, we can view a code point as a Unicode character, but we should know that some Unicode characters are composed of more than one code points each.

A rune literal is expressed as one or more characters enclosed in a pair of quotes. The enclosed characters denote one Unicode code point value. There are some minor variants of the rune literal form. The most popular form of rune literals is just to enclose the characters denoted by rune values between two single quotes. For example

```
'a' // an English character
'π'
'众' // a Chinese character
```

The following rune literal variants are equivalent to `'a'` (the Unicode value of character `a` is 97).

```
// 141 is the octal representation of decimal number 97.
'\141'
// 61 is the hex representation of decimal number 97.
'\x61'
'\u0061'
'\U00000061'
```

Please note, \ must be followed by exactly three octal digits to represent a byte value, \x must be followed by exactly two hex digits to represent a byte value, \u must be followed by exactly four hex digits to represent a rune value, and \U must be followed by exactly eight hex digits to represent a rune value. Each such octal or hex digit sequence must represent a legal Unicode code point, otherwise, it fails to compile.

The following program will print 7 true texts.

```
1| package main
2|
3| func main() {
4|     println('a' == 97)
5|     println('a' == '\141')
6|     println('a' == '\x61')
7|     println('a' == '\u0061')
8|     println('a' == '\U00000061')
9|     println(0x61 == '\x61')
10|    println('\u4f17' == '众')
11| }
```

In fact, the four variant rune literal forms just mentioned are rarely used for rune values in practice. They are occasionally used in interpreted string literals (see the next subsection for details).

If a rune literal is composed by two characters (not including the two quotes), the first one is the character \ and the second one is not a digital character, x, u and U, then the two successive characters will be escaped as one special character. The possible character pairs to be escaped are:

\a	(Unicode value 0x07) alert or bell
\b	(Unicode value 0x08) backspace
\f	(Unicode value 0x0C) form feed
\n	(Unicode value 0x0A) line feed or newline
\r	(Unicode value 0x0D) carriage return
\t	(Unicode value 0x09) horizontal tab
\v	(Unicode value 0x0b) vertical tab
\\\	(Unicode value 0x5c) backslash
\'	(Unicode value 0x27) single quote

\n is the most used escape character pair.

An example:

```

1|     println('\n') // 10
2|     println('\r') // 13
3|     println('\\') // 39
4|
5|     println('\n' == 10)    // true
6|     println('\n' == '\x0A') // true

```

There are many literals which can denote the zero values of rune types, such as '\000', '\x00', '\u0000', etc. In fact, we can also use any numeric literal introduced above to represent the values of rune types, such as 0, 0x0, 0.0, 0e0, 0i, etc.

String value literals

String values in Go are UTF-8 encoded. In fact, all Go source files must be UTF-8 encoding compatible.

There are two forms of string value literals, interpreted string literal (double quotes form) and raw string literal (back quotes form). For example, the following two string literals are equivalent:

```

// The interpreted form.
"Hello\nworld!\n\"你好世界\""

// The raw form.
`Hello
world!
"你好世界"`

```

In the above interpreted string literal, each \n character pair will be escaped as one newline character, and each \" character pair will be escaped as one double quote character. Most of such escape character pairs are the same as the escape character pairs used in rune literals introduced above, except that \" is only legal in interpreted string literals and ` is only legal in rune literals.

The character sequence of \, \x, \u and \U followed by several octal or hex digits introduced in the last section can also be used in interpreted string literals.

```

// The following interpreted string literals are equivalent.
"\141\142\143"
"\x61\x62\x63"
"\x61b\x63"

```

```

"abc"

// The following interpreted string literals are equivalent.
"\u4f17\xe4\xba\xba"
    // The Unicode of 众 is 4f17, which is
    // UTF-8 encoded as three bytes: e4 bc 97.
"\xe4\xbc\x97\u4eba"
    // The Unicode of 人 is 4eba, which is
    // UTF-8 encoded as three bytes: e4 ba ba.
"\xe4\xbc\x97\xe4\xba\xba"
"众人"

```

Please note that each English character (code point) is represented with one byte, but each Chinese character (code point) is represented with three bytes.

In a raw string literal, no character sequences will be escaped. The back quote character is not allowed to appear in a raw string literal. To get better cross-platform compatibility, carriage return characters (Unicode code point 0x0D) inside raw string literals will be discarded.

Zero values of string types can be denoted as "" or `` in literal.

Representability of Basic Numeric Value Literals

A numeric literal can be used to represent as an integer value only if it needn't be rounded. For example, 1.23e2 can represent as values of any basic integer types, but 1.23 can't represent as values of any basic integer types. Rounding is allowed when using a numeric literal to represent a non-integer basic numeric values.

Each basic numeric type has a representable value range. So, if a literal overflows the value range of a type, then the literal is not representable as values of the type.

Some examples:

The Literal	Types Which Values the Literal Can Represent
256	All basic numeric types except int8 and uint8 types.
255	All basic numeric types except int8 types.
-123	All basic numeric types except the unsigned ones.
123	All basic numeric types.
123.000	
1.23e2	
'a'	

The Literal	Types Which Values the Literal Can Represent
<code>1.0+0i</code>	
<code>1.23</code>	
<code>0x1000000000000000</code> (16 zeros)	All basic floating-point and complex numeric types.
<code>3.5e38</code>	All basic floating-point and complex numeric types except float32 and complex64 types.
<code>1+2i</code>	All basic complex numeric types.
<code>2e+308</code>	None basic types.

Notes:

- Because no values of the basic integer types provided in Go can hold `0x1000000000000000`, so the literal is not representable as values of any basic integer types.
- The maximum IEEE-754 float32 value which can be represented accurately is `3.40282346638528859811704183484516925440e+38`, so `3.5e38` is not representable as values of any float32 and complex64 types.
- The max IEEE-754 float64 value which can be represented accurately is `1.797693134862315708145274237317043567981e+308`, so `2e+308` is not representable as values of any float64 and complex128 types.
- In the end, please note, although `0x1000000000000000` can represent values of float32 types, however it can't represent any float32 values accurately in memory. In other words, it will be rounded to the closest float32 value which can be represented accurately in memory when it is used as values of float32 types.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Constants and Variables

This article will introduce constant and variable declarations in Go. The concept of untyped values and explicit conversions will also be introduced.

The literals introduced in [the last article](#) (§6) are all called unnamed constants (or literal constants), except `false` and `true`, which are two predeclared (built-in) named constants. Custom named constant declarations will be introduced below in this article.

Untyped Values and Typed Values

In Go, some values are untyped. An untyped value means the type of the value has not been confirmed yet. On the contrary, the type of a typed value is determined.

For most untyped values, each of them has one default type. The predeclared `nil` is the only untyped value which has no default type. We will learn more about `nil` in other Go 101 articles later.

All literal constants (unnamed constants) are untyped values. In fact, in Go, most untyped values are literal constants and named constants (which will be introduced below in the current article). The other untyped values include the just mentioned `nil` and some boolean results returned by some operations which will be introduced in other articles later.

The default type of a literal constant is determined by its literal form.

- The default type of a string literal is `string`.
- The default type of a boolean literal is `bool`.
- The default type of an integer literal is `int`.
- The default type of a rune literal is `rune` (a.k.a., `int32`).
- The default type of a floating-point literal is `float64`.
- If a literal contains an imaginary part, then its default type is `complex128`.

Explicit Conversions of Untyped Constants

Like many other languages, Go also supports value conversions. We can use the form `T(v)` to convert a value `v` to the type denoted by `T` (or simply speaking, type `T`). If the conversion `T(v)` is

legal, Go compilers view $T(v)$ as a typed value of type T . Surely, for a certain type T , to make the conversion $T(v)$ legal, the value v can't be arbitrary.

The following mentioned rules apply for both the literal constants introduced in the last article and the untyped named constants which will be introduced soon.

For an untyped constant value v , there are two scenarios where $T(v)$ is legal.

1. v (or the literal denoted by v) is [representable](#) (§6) as a value of a basic type T . The result value is a typed constant of type T .
2. The default type of v is an integer type (`int` or `rune`) and T is a string type. The result of $T(v)$ is a string of type T and contains the UTF-8 representation of the integer as a Unicode code point. Integer values outside the range of valid Unicode code points result strings represented by "`\uFFFD`" (a.k.a., "`\xef\xbf\xbd`"). `0xFFFFD` is the code point for the Unicode replacement character. The result string of a conversion from an integer always contains one and only one rune. (Note, later Go version might [only allow converting rune or byte integers to strings](#) . Since Go Toolchain 1.15, the `go vet` command warns on conversions from non-rune and non-byte integers to strings.)

In fact, the second scenario doesn't require v to be a constant. If v is a constant, then the result of the conversion is also a constant, otherwise, the result is not a constant.

For example, the following conversions are all legal.

```
// Rounding happens in the following 3 lines.
complex128(1 + -1e-1000i) // 1.0+0.0i
float32(0.49999999)      // 0.5
float32(17000000000000000)
// No rounding in the these lines.
float32(123)
uint(1.0)
int8(-123)
int16(6+0i)
complex128(789)

string(65)                // "A"
string('A')                // "A"
string('\u68ee')           // "森"
string(-1)                 // "\uFFFD"
string(0xFFFFD)             // "\uFFFD"
string(0x2FFFFFFF)          // "\uFFFD"
```

And the following conversions are all illegal.

```
// 1.23 is not representable as a value of int.
int(1.23)
// -1 is not representable as a value of uint8.
uint8(-1)
// 1+2i is not representable as a value of float64.
float64(1+2i)

// Constant -1e+1000 overflows float64.
float64(-1e1000)
// Constant 0x1000000000000000 overflows int.
int(0x1000000000000000)

// The default type of 65.0 is float64,
// which is not an integer type.
string(65.0)
// The default type of 66+0i is complex128,
// which is not an integer type.
string(66+0i)
```

From the above examples, we know that an untyped constant, (for example `-1e1000` and `0x1000000000000000`), may even not be able to represent as a value of its default type.

Please note, sometimes, the form of explicit conversions must be written as `(T)(v)` to avoid ambiguities. Such situations often happen in case of `T` is not an identifier.

We will learn more explicit conversion rules later in other Go 101 articles.

Introduction of Type Deductions in Go

Go supports type deduction. In other words, in many circumstances, programmers don't need to explicitly specify the types of some values in code. Go compilers will deduce the types for these values by context.

Type deduction is also often called type inference.

In Go code, if a place needs a value of a certain type and an untyped value (often a constant) is representable as a value of the certain type, then the untyped value can be used in the place. Go compilers will view the untyped value as a typed value of the certain type. Such places include an operand in an operator operation, an argument in a function call, a destination value or a source value in an assignment, etc.

Some circumstances have no requirements on the types of the used values. If an untyped value is used in such a circumstance, Go compilers will treat the untyped value as a typed value of its

default type.

The two type deduction cases can be viewed as implicit conversions.

The below constant and variable declaration sections will show some type deduction cases. More type deduction rules and cases will be introduced in other articles.

Constant Declarations

Unnamed constants are all boolean, numeric and string values. Like unnamed constants, named constants can also be only boolean, numeric and string values. The keyword `const` is used to declare named constants. The following program contains some constant declarations.

```

1| package main
2|
3| // Declare two individual constants. Yes,
4| // non-ASCII letters can be used in identifiers.
5| const π = 3.1416
6| const Pi = π // <=> const Pi = 3.1416
7|
8| // Declare multiple constants in a group.
9| const (
10|     No      = !Yes
11|     Yes     = true
12|     MaxDegrees = 360
13|     Unit      = "radian"
14| )
15|
16| func main() {
17|     // Declare multiple constants in one line.
18|     const TwoPi, HalfPi, Unit2 = π * 2, π * 0.5, "degree"
19| }
```

Go specification calls each of the lines containing a `=` symbol in the above constant declaration group as a ***constant specification***.

In the above example, the `*` symbol is the multiplication operator and the `!` symbol is the boolean-not operator. Operators will be introduced in the next article, [common operators](#) (§8).

The `=` symbol means "bind" instead of "assign". We should interpret each constant specification as a declared identifier is bound to a corresponding basic value literal. Please read the last section in the current article for more explanations.

In the above example, the name constants `π` and `Pi` are both bound to the literal `3.1416`. The two named constants may be used at many places in code. Without constant declarations, the literal `3.1416` would be populated at those places. If we want to change the literal to `3.14` later, many places need to be modified. With the help of constant declarations, the literal `3.1416` will only appear in one constant declaration, so only one place needs to be modified. This is the main purpose of constant declarations.

Later, we use the terminology ***non-constant*** values to denote the values who are not constants. The to be introduced variables below, all belong to one kind of non-constant values.

Please note that, constants can be declared both at package level (out of any function body) and in function bodies. The constants declared in function bodies are called local constants. The variables declared out of any function body are called package-level constants. We also often call package-level constants as global constants.

The declaration orders of two package-level constants are not important. In the above example, the declaration orders of `No` and `Yes` can be exchanged.

All constants declared in the last example are untyped. The default type of a named untyped constant is the same as the literal bound to it.

Typed named constants

We can declare typed constants, typed constants are all named. In the following example, all the four declared constants are typed values. The types of `X` and `Y` are both `float32` and the types of `A` and `B` are both `int64`.

```
1| const X float32 = 3.14
2|
3| const (
4|     A, B int64    = -3, 5
5|     Y      float32 = 2.718
6| )
```

If multiple typed constants are declared in the same constant specification, then their types must be the same, just as the constants `A` and `B` in the above example.

We can also use explicit conversions to provide enough information for Go compilers to deduce the types of typed named constants. The above code snippet is equivalent to the following one, in which `X`, `Y`, `A` and `B` are all typed constants.

```

1| const X = float32(3.14)
2|
3| const (
4|     A, B = int64(-3), int64(5)
5|     Y     = float32(2.718)
6| )

```

If a basic value literal is bound to a typed constant, the basic value literal must be representable as a value of the type of the constant. The following typed constant declarations are invalid.

```

1| // error: 256 overflows uint8
2| const a uint8 = 256
3| // error: 256 overflows uint8
4| const b = uint8(255) + uint8(1)
5| // error: 128 overflows int8
6| const c = int8(-128) / int8(-1)
7| // error: -1 overflows uint
8| const MaxUint_a = uint(^0)
9| // error: -1 overflows uint
10| const MaxUint_b uint = ^0

```

In the above and following examples `^` is bitwise-not operator.

The following typed constant declaration is valid on 64-bit OSes, but invalid on 32-bit OSes. For each `uint` value has only 32 bits on 32-bit OSes. $(1 \ll 64) - 1$ is not representable as 32-bit values. (Here, `<<` is bitwise-left-shift operator.)

```
1| const MaxUint uint = (1 << 64) - 1
```

Then how to declare a typed `uint` constant and bind the largest `uint` value to it? Use the following way instead.

```
1| const MaxUint = ^uint(0)
```

Similarly, we can declare a typed `int` constant and bind the largest `int` value to it. (Here, `>>` is bitwise-right-shift operator.)

```
1| const MaxInt = int(^uint(0) >> 1)
```

A similar method can be used to get the number of bits of a native word, and check the current OS is 32-bit or 64-bit.

```

1| // NativeWordBits is 64 or 32.
2| const NativeWordBits = 32 << (^uint(0) >> 63)

```

```
3| const Is64bitOS = ^uint(0) >> 63 != 0
4| const Is32bitOS = ^uint(0) >> 32 == 0
```

Here, `!=` and `==` are not-equal-to and equal-to operators.

| Autocomplete in constant declarations

In a group-style constant declaration, except the first constant specification, other constant specifications can be incomplete. An incomplete constant specification only contains an identifier list. Compilers will autocomplete the incomplete lines for us by copying the missing part from the first preceding complete constant specification. For example, at compile time, compilers will automatically complete the following code

```
1| const (
2|     X float32 = 3.14
3|     Y           // here must be one identifier
4|     Z           // here must be one identifier
5|
6|     A, B = "Go", "language"
7|     C, _         // In the above line, the blank identifier
8|     // is required to be present.
9|
10| )
```

as

```
1| const (
2|     X float32 = 3.14
3|     Y float32 = 3.14
4|     Z float32 = 3.14
5|
6|     A, B = "Go", "language"
7|     C, _ = "Go", "language"
8| )
```

| iota in constant declarations

The autocomplete feature plus the `iota` constant generator feature brings much convenience to Go programming. `iota` is a predeclared constant which can only be used in other constant declarations. It is declared as

```
1| const iota = 0
```

But the value of an `iota` in code may be not always `0`. When the predeclared `iota` constant is used in a custom constant declaration, at compile time, within the custom constant declaration, its value will be reset to `0` at the first constant specification of each group of constants and will increase `1` constant specification by constant specification. In other words, in the `n`th constant specification of a constant declaration, the value of `iota` is `n` (starting from zero). So `iota` is only useful in group-style constant declarations.

Here is an example using both the autocomplete and the `iota` constant generator features. Please read the comments to get what will happen at compile time. The `+` symbol in this example is the addition operator.

```

1| package main
2|
3| func main() {
4|     const (
5|         k = 3 // now, iota == 0
6|
7|         m float32 = iota + .5 // m float32 = 1 + .5
8|         n                     // n float32 = 2 + .5
9|
10|        p = 9                // now, iota == 3
11|        q = iota * 2         // q = 4 * 2
12|        _                   // _ = 5 * 2
13|        r                   // r = 6 * 2
14|        s, t = iota, iota // s, t = 7, 7
15|        u, v               // u, v = 8, 8
16|        _, w               // _, w = 9, 9
17|    )
18|
19|    const x = iota // x = 0
20|    const (
21|        y = iota // y = 0
22|        z       // z = 1
23|    )
24|
25|    println(m)           // +1.500000e+000
26|    println(n)           // +2.500000e+000
27|    println(q, r)         // 8 12
28|    println(s, t, u, v, w) // 7 7 8 8 9
29|    println(x, y, z)       // 0 0 1
30| }
```

The above example is just to demo the rules of the `iota` constant generator feature. Surely, in practice, we should use it in more meaningful ways. For example,

```
1| const (
2|     Failed = iota - 1 // == -1
3|     Unknown          // == 0
4|     Succeeded        // == 1
5| )
6|
7| const (
8|     Readable = 1 << iota // == 1
9|     Writable          // == 2
10|    Executable       // == 4
11| )
```

Here, the `-` symbol is the subtraction operator, and the `<<` symbol is the left-shift operator. Both of these operators will be introduced in the next article.

Variables, Variable Declarations and Value Assignments

Variables are named values. Variables are stored in memory at run time. The value represented by a variable can be modified at run time.

All variables are typed values. When declaring a variable, there must be sufficient information provided for compilers to deduce the type of the variable.

The variables declared within function bodies are called local variables. The variables declared out of any function body are called package-level variables. We also often call package-level variables as global variables.

There are two basic variable declaration forms, the standard one and the short one. The short form can only be used to declare local variables.

Standard variable declaration forms

Each standard declaration starts with the `var` keyword, which is followed by the declared variable name. Variable names must be [identifiers](#) (§5).

The following are some full standard declaration forms. In these declarations, the types and initial values of the declared variables are all specified.

```

1| var lang, website string = "Go", "https://golang.org"
2| var compiled, dynamic bool = true, false
3| var announceYear int = 2009

```

As we have found, multiple variables can be declared together in one variable declaration. Please note, there can be just one type specified in a variable declaration. So the types of the multiple variables declared in the same declaration line must be identical.

Full standard variable declaration forms are seldom used in practice, since they are verbose. In practice, the two standard variable declaration variant forms introduced below are used more often. In the two variants, either the types or the initial values of the declared variables are absent.

The following are some standard variable declarations without specifying variable types. Compilers will deduce the types of the declared variables as the types (or default types) of their respective initial values. The following declarations are equivalent to the above ones in fact. Please note, in the following declarations, the types of the multiple variables declared in the same declaration line can be different.

```

1| // The types of the lang and dynamic variables
2| // will be deduced as built-in types "string"
3| // and "bool" by compilers, respectively.
4| var lang, dynamic = "Go", false
5|
6| // The types of the compiled and announceYear
7| // variables will be deduced as built-in
8| // types "bool" and "int", respectively.
9| var compiled, announceYear = true, 2009
10|
11| // The types of the website variable will be
12| // deduced as the built-in type "string".
13| var website = "https://golang.org"

```

The type deductions in the above example can be viewed as implicit conversions.

The following are some standard declarations without specifying variable initial values. In these declarations, all declared variables are initialized as the zero values of their respective types.

```

1| // Both are initialized as blank strings.
2| var lang, website string
3| // Both are initialized as false.
4| var interpreted, dynamic bool
5| // n is initialized as 0.
6| var n int

```

Multiple variables can be grouped into one standard form declaration by using (). For example:

```
1| var (
2|     lang, bornYear, compiled      = "Go", 2007, true
3|     announceAt, releaseAt      int = 2009, 2012
4|     createdBy, website        string
5| )
```

The above example is formatted by using the `go fmt` command provided in Go Toolchain. In the above example, each of the three lines are enclosed in () this is known as variable specification.

Generally, declaring related variables together will make code more readable.

Pure value assignments

In the above variable declarations, the sign `=` means assignment. Once a variable is declared, we can modify its value by using pure value assignments. Like variable declarations, multiple values can be assigned in a pure assignment.

The expression items at the left of `=` symbol in a pure assignment are called destination or target values. They must be addressable values, map index expressions, or the blank identifier. Value addresses and maps will be introduced in later articles.

Constants are immutable, so a constant can't show up at the left side of a pure assignment as a destination value, it can only appear at the right side as a source value. Variables can be used as both source values and destination values, so they can appear at both sides of pure value assignments.

Blank identifiers can also appear at the left side of pure value assignments as destination values, in which case, it means we ignore the destination values. Blank identifiers can't be used as source values in assignments.

Example:

```
1| const N = 123
2| var x int
3| var y, z float32
4|
5| N = 9 // error: constant N is not modifiable
6| y = N // ok: N is deduced as a float32 value
7| x = y // error: type mismatch
8| x = N // ok: N is deduced as an int value
9| y = x // error: type mismatch
10| z = y // ok
```

```

11| _ = y // ok
12|
13| x, y = y, x // error: type mismatch
14| x, y = int(y), float32(x) // ok
15| z, y = y, z           // ok
16| _, y = y, z          // ok
17| z, _ = y, z          // ok
18| _, _ = y, z          // ok
19| x, y = 69, 1.23     // ok

```

The code at last line in the above example uses explicit conversions to make the corresponding destination and source values matched. The explicit conversion rules for non-constant numeric values are introduced below.

Go doesn't support assignment chain. For example, the following code is illegal.

```

1| var a, b int
2| a = b = 123 // syntax error

```

Short variable declaration forms

We can also use short variable declaration forms to declare variables. Short variable declarations can only be used to declare local variables. Let's view an example which uses some short variable declarations.

```

1| package main
2|
3| func main() {
4|     // Both lang and year are newly declared.
5|     lang, year := "Go language", 2007
6|
7|     // Only createdBy is a new declared variable.
8|     // The year variable has already been
9|     // declared before, so here its value is just
10|    // modified, or we can say it is redeclared.
11|    year, createdBy := 2009, "Google Research"
12|
13|    // This is a pure assignment.
14|    lang, year = "Go", 2012
15|
16|    print(lang, " is created by ", createdBy)
17|    println(", and released at year", year)
18| }

```

Each short variable declaration must declare at least one new variable.

There are several differences between short and standard variable declarations.

1. In the short declaration form, the `var` keyword and variable types must be omitted.
2. The assignment sign must be `:=` instead of `=`.
3. In the short variable declaration, old variables and new variables can mix at the left of `:=`. But there must be at least one new variable at the left.

Please note, comparing to pure assignments, there is a limit for short variable declarations. **In a short variable declaration, all items at the left of the `:=` sign must pure identifiers.** This means some other items which can be assigned to, which will be introduced in other articles, can't appear at the left of `:=`. These items include qualified identifiers, container elements, pointer dereferences and struct field selectors. Pure assignments have no such limit.

About the terminology "assignment"

Later, when the word "assignment" is mentioned, it may mean a pure assignment, a short variable declaration, or a variable specification with initial values in a standard variable declaration. In fact, a more general definition also includes [function argument passing](#) (§9) introduced in a follow-up article.

We say x is **assignable to** y if $y = x$ is a legal statement (compiles okay). Assume the type of y is Ty , sometimes, for description convenience, we can also say x is **assignable to type** Ty .

Generally, if x is assignable to y , then y should be mutable, and the types of x and y are identical or x can be implicitly converted to the type of y . Surely, y can also be the blank identifier `_`.

Each local declared variable must be used at least once effectively

Please note, the standard Go compiler and gccgo both don't allow local variables declared but not used. Package-level variables have no such limit.

If a local variable is only ever used as destination values, it will also be viewed as unused.

For example, in the following program, `r` is only used as destination.

```
1| package main
2|
3| // Some package-level variables.
4| var x, y, z = 123, true, "foo"
```

```

5|
6| func main() {
7|     var q, r = 789, false
8|     r, s := true, "bar"
9|     r = y // r is unused.
10|    x = q // q is used.
11| }
```

Compiling the above program will result to the following compilation errors (assume the source file is name `example-unused.go`):

```

./example-unused.go:6:6: r declared and not used
./example-unused.go:7:16: s declared and not used
```

The fix is easy, we can assign `r` and `s` to blank identifiers to avoid compilation errors.

```

1| package main
2|
3| var x, y, z = 123, true, "foo"
4|
5| func main() {
6|     var q, r = 789, false
7|     r, s := true, "bar"
8|     r = y
9|     x = q
10|
11|     _, _ = r, s // make r and s used.
12| }
```

Generally, the above fix is not recommended to be used in production code. It should be used in development/debug phase only. It is not a good habit to leave unused local variables in code, for unused local variables have negative effects on both code readability and program execution performance.

Dependency relations of package-Level variables affect their initialization order

For the following example,

```

1| var x, y = a+1, 5          // 8 5
2| var a, b, c = b+1, c+1, y // 7 6 5
```

the initialization order of the package-level variables are `y = 5`, `c = y`, `b = c+1`, `a = b+1`, and `x = a+1`.

Here, the `+` symbol is the addition operator, which will be introduced in the next article.

Package-level variables can't be depended circularly in their declaration. The following code fails to compile.

```
1| var x, y = y, x
```

Value Addressability

In Go, some values are addressable (there is an address to find them). All variables are addressable and all constants are unaddressable. We can learn more about addresses and pointers from the article [pointers in Go](#) (§15) and learn other addressable and unaddressable values from other articles later.

Explicit Conversions on Non-Constant Numeric Values

In Go, two typed values of two different basic types can't be assigned to each other. In other words, the types of the destination and source values in an assignment must be identical if the two values are both basic values. If the type of the source basic value is not same as the type of the destination basic value, then the source value must be explicitly converted to the type of the destination value.

As mentioned above, non-constant integer values can be converted to strings. Here we introduce two more legal non-constant numeric values related conversion cases.

- Non-constant floating-point and integer values can be explicitly converted to any other floating-point and integer types.
- Non-constant complex values can be explicitly converted to any other complex types.

Unlike constant number conversions, overflows are allowed in non-constant number conversions. And when converting a non-constant floating-point value to an integer, rounding is also allowed. If a non-constant floating-point value doesn't overflow an integer type, the fraction part of the floating-point value will be discarded (towards zero) when it is converted to the integer type.

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

In the following example, the intended implicit conversions at line 7 and line 18 both don't work. The explicit conversions at line 5 and line 16 are also disallowed.

```

1| const a = -1.23
2| // The type of b is deduced as float64.
3| var b = a
4| // error: constant 1.23 truncated to integer.
5| var x = int32(a)
6| // error: cannot assign float64 to int32.
7| var y int32 = b
8| // okay: z == -1, and the type of z is int32.
9| //       The fraction part of b is discarded.
10| var z = int32(b)
11|
12| const k int16 = 255
13| // The type of n is deduced as int16.
14| var n = k
15| // error: constant 256 overflows uint8.
16| var f = uint8(k + 1)
17| // error: cannot assign int16 to uint8.
18| var g uint8 = n + 1
19| // okay: h == 0, and the type of h is uint8.
20| //       n+1 overflows uint8 and is truncated.
21| var h = uint8(n + 1)
22|

```

We can think that value `a` at line 3 is implicitly converted to its default type (`float64`), so that the type of `b` is deduced as `float64`. More implicit conversion rules will be introduced in other articles later.

Scopes of Variables and Named Constants

In Go, we can use a pair of `{` and `}` to form a code block. A code block can nest other code blocks. A variable or a named constant declared in an inner code block will shadow the variables and constants declared with the same name in outer code blocks. For examples, the following program declares three distinct variables, all of them are called `x`. An inner `x` shadows an outer one.

```

1| package main
2|
3| const y = 789
4| var x int = 123
5|
6| func main() {
7|     // The x variable shadows the above declared
8|     // package-level variable x.
9|     var x = true

```

```

10|
11|    // A nested code block.
12|    {
13|        // Here, the left x and y are both
14|        // new declared variable. The right
15|        // ones are declared in outer blocks.
16|        x, y := x, y
17|
18|        // In this code block, the just new
19|        // declared x and y shadow the outer
20|        // declared same-name identifiers.
21|        x, z := !x, y/10 // only z is new declared
22|        y /= 100
23|        println(x, y, z) // false 7 78
24|    }
25|    println(x) // true
26|    println(z) // error: z is undefined.
27|

```

The scope (visibility range in code) of a package-level variable (or a named constant) is the whole package of the variable (or the named constant) is declared in. The scope of a local variable (or a named constant) begins at the end of its declaration and ends at the end of its innermost containing code block. This is why the last line in the `main` function of the above example doesn't compile.

Code blocks and identifier scopes will be explained in detail in [blocks and scopes](#) (§32) later.

More About Constant Declarations

The value denoted by an untyped constant can overflow its default type

For example, the following code compiles okay.

```

1| // 3 untyped named constants. Their bound
2| // values all overflow their respective
3| // default types. This is allowed.
4| const n = 1 << 64          // overflows int
5| const r = 'a' + 0xFFFFFFFF // overflows rune
6| const x = 2e+308           // overflows float64
7|
8| func main() {
9|     _ = n >> 2

```

```

10|     _ = r - 0x7FFFFFFF
11|     _ = x / 2
12| }
```

But the following code does't compile, for the constants are all typed.

```

1| // 3 typed named constants. Their bound
2| // values are not allowed to overflow their
3| // respective default types. The 3 lines
4| // all fail to compile.
5| const n int = 1 << 64           // overflows int
6| const r rune = 'a' + 0x7FFFFFFF // overflows rune
7| const x float64 = 2e+308       // overflows float64
```

Each named constant identifier will be replaced with its bound literal value at compile time

Constant declarations can be viewed as enhanced `#define` macros in C. A constant declaration defines a named constant which represents a literal. All the occurrences of a named constant will be replaced with the literal it represents at compile time.

If the two operands of an operator operation are both constants, then the operation will be evaluated at compile time. Please read the next article [common operators](#) (§8) for details.

For example, at compile time, the following code

```

1| package main
2|
3| const X = 3
4| const Y = X + X
5| var a = X
6|
7| func main() {
8|     b := Y
9|     println(a, b, X, Y)
10| }
```

will be viewed as

```

1| package main
2|
3| var a = 3
4|
5| func main() {
```

```
6|     b := 6
7|     println(a, b, 3, 6)
8| }
```

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Common Operators

Operator operations are the operations using all kinds of operators. This article will introduce common operators in Go. More operators will be introduced in other articles later.

About Some Descriptions in Operator Explanations

This article will only introduce arithmetic operators, bitwise operators, comparison operators, boolean operators and string concatenation operator. These operators are either binary operators or unary operators. A binary operator operation takes two operands and a unary operator operation takes only one operand.

All the operator operations introduced in this article each returns one result.

This article doesn't pursue the accuracy of some descriptions. For example, when it says that a binary operator requires the types of its two operands must be the same, what it means is:

- if both of the two operands are typed values, then their types must be the same one, or one operand can be implicitly converted to the type of the other.
- if only one of the two operands is typed, then the other (untyped) operand must be representable as a value of the type of the typed operand, or the values of the default type of the other (untyped) operand can be implicitly converted to the typed of the typed operand.
- if both operands are untyped values, then they must be both boolean values, both string values or both basic numeric values.

Similarly, when it says an operator, either a binary operator or a unary operator, requires the type of one of its operands to be of a certain type, what it means is:

- if the operand is typed, then its type must be, or can be implicitly converted to, that certain type.
- if the operand is untyped, then the untyped value must be representable as a value of that certain type, or the values of the default type of the operand can be implicitly converted to that certain type.

Constant Expressions

Before introducing all kinds of operators, we should know what are constant expressions and a fact in the evaluations of constant expressions. Expressions will get explained in a later article

[expressions and statements](#) (§11). At present, we just should know that most of the operations mentioned in the current article are expressions.

If all the operands involved in an expression are constants, then this expression is called a constant expression. All constant expressions are evaluated at compile time. The evaluation result of a constant expression is still a constant.

Only if one operand in an expression is not a constant, the expression is called a non-constant expression.

Arithmetic Operators

Go supports five basic binary arithmetic operators:

Operator	Name	Requirements for the Two Operands
+	addition	
-	subtraction	
*	multiplication	The two operands must be both values of the same basic numeric type.
/	division	
%	remainder	The two operands must be both values of the same basic integer type.

The five operators are also often called **sum**, **difference**, **product**, **quotient** and **modulo** operators, respectively. Go 101 will not explain how these operator operations work in detail.

Go supports six bitwise binary arithmetic operators:

Operator	Name	Requirements for the Two Operands and Mechanism Explanations
&	bitwise and	The two operands must be both values of the same integer type. Mechanism explanations (a value with the subscript 2 is the binary literal form of the value): <ul style="list-style-type: none">• $1100_2 \& 1010_2$ results in 1000_2• $1100_2 1010_2$ results in 1110_2• $1100_2 ^ 1010_2$ results in 0110_2• $1100_2 \&^ 1010_2$ results in 0100_2
	bitwise or	
^	bitwise xor	

Operator	Name	Requirements for the Two Operands and Mechanism Explanations
<code>&^</code>	bitwise clear	
<code><<</code>	bitwise left shift	<p>The left operand must be an integer and the right operand must be also an integer (if it is a constant, then it must be non-negative), their types are not required to be identical. (Note, before Go 1.13, the right operand must be an unsigned integer or an untyped (§7) integer constant which is representable as an <code>uint</code> value.)</p> <p>A negative right operand (must be a non-constant) will cause a panic at run time.</p>
<code>>></code>	bitwise right shift	<p>Mechanism explanations:</p> <ul style="list-style-type: none"> $1100_2 \ll 3$ results in 1100000_2 $1100_2 \gg 3$ results in 1_2 <p>Note: in a bitwise-right-shift operation, all the freed-up bits at left are filled with the sign bit (the highest bit) of the left operand. For example, if the left operand is an <code>int8</code> value -128, or 10000000_2 in the binary literal form, then $10000000_2 \gg 2$ results 11100000_2, a.k.a., -32.</p>

Go also supports three unary arithmetic operators:

Operator	Name	Explanations
<code>+</code>	positive	<code>+n</code> is equivalent to <code>0 + n</code> .
<code>-</code>	negative	<code>-n</code> is equivalent to <code>0 - n</code> .
<code>^</code>	bitwise complement (bitwise not)	<code>^n</code> is equivalent to <code>m ^ n</code> , where <code>m</code> is a value all of whose bits are 1. For example, if the type of <code>n</code> is <code>int8</code> , then <code>m</code> is -1, and if the type of <code>n</code> is <code>uint8</code> , then <code>m</code> is <code>0xFF</code> .

Note,

- in many other languages, bitwise-complement operator is denoted as `~`.
- like many other languages, the addition binary operator `+` can also be used as **string concatenation** operator, which will be introduced below.

- like C and C++ languages, the multiplication binary operator `*` can also be used as **pointer dereference** operator, and the bitwise-and operator `&` can also be used as **pointer address** operator. Please read [pointers in Go](#) (§15) for details later.
- unlike Java language, Go supports unsigned integer types, so the unsigned shift operator `>>>` doesn't exist in Go.
- there is no power operator in Go, please use `Pow` function in the `math` standard package instead. Code package and package import will be introduced in the next article [packages and imports](#) (§10).
- the bitwise-clear operator `&n` is a unique operator in Go. `m &n n` is equivalent to `m & (^n)`.

Example:

```

1| func main() {
2|     var (
3|         a, b float32 = 12.0, 3.14
4|         c, d int16    = 15, -6
5|         e    uint8    = 7
6|     )
7|
8|     // The ones compile okay.
9|     _ = 12 + 'A' // two numeric untyped operands
10|    _ = 12 - a   // one untyped and one typed operand
11|    _ = a * b   // two typed operands
12|    _ = c % d
13|    _, _ = c + int16(e), uint8(c) + e
14|    _, _, _, _ = a / b, c / d, -100 / -9, 1.23 / 1.2
15|    _, _, _, _ = c | d, c & d, c ^ d, c &n d
16|    _, _, _, _ = d << e, 123 >> e, e >> 3, 0xF << 0
17|    _, _, _, _ = -b, +c, ^e, ^-1
18|
19|     // The following ones fail to compile.
20|     _ = a % b   // error: a and b are not integers
21|     _ = a | b   // error: a and b are not integers
22|     _ = c + e   // error: type mismatching
23|     _ = b >> 5 // error: b is not an integer
24|     _ = c >> -5 // error: -5 is not representable as uint
25|
26|     _ = e << uint(c) // compiles ok
27|     _ = e << c       // only compiles ok since Go 1.13
28|     _ = e << -c      // only compiles ok since Go 1.13,
29|                           // will cause a panic at run time.
30|     _ = e << -1      // error: right operand is negative
31| }
```

About overflows

Overflows are not allowed for typed constant values but are allowed for non-constant and untyped constant values, either the values are intermediate or final results. Overflows will be truncated (or wrapped around) for non-constant values, but overflows (for default types) on untyped constant value will not be truncated (or wrapped around).

Example:

```

1| // Results are non-constants.
2| var a, b uint8 = 255, 1
3| // Compiles ok, higher overflowed bits are truncated.
4| var c = a + b // c == 0
5| // Compiles ok, higher overflowed bits are truncated.
6| var d = a << b // d == 254
7|
8| // Results are untyped constants.
9| const X = 0x1FFFFFFF * 0x1FFFFFFF // overflows int
10| const R = 'a' + 0x7FFFFFFF // overflows rune
11| // The above two lines both compile ok, though the
12| // two untyped value X and R both overflow their
13| // respective default types.
14|
15| // Operation results or conversion results are
16| // typed values. These lines all fail to compile.
17| var e = X // error: untyped constant X overflows int
18| var h = R // error: constant 2147483744 overflows rune
19| const Y = 128 - int8(1) // error: 128 overflows int8
20| const Z = uint8(255) + 1 // error: 256 overflow uint8

```

About the results of arithmetic operator operations

Except bitwise shift operations, the result of a binary arithmetic operator operation

- is a typed value of the same type of the two operands if the two operands are both typed values of the same type.
- is a typed value of the same type of the typed operand if only one of the two operands is a typed value. In the computation, the other (untyped) value will be deduced as a value of the type of the typed operand. In other words, the untyped operand will be implicitly converted to the type of the typed operand.
- is still an untyped value if both of the two operands are untyped. The default type of the result value is one of the two default types and it is the one appears latter in this list: `int`, `rune`,

`float64`, `complex128`. For example, if the default type of one untyped operand is `int`, and the other one is `rune`, then the default type of the result untyped value is `rune`.

The rules for the result of a bitwise shift operator operation is a little complicated. Firstly, the result value is always an integer value. Whether it is typed or untyped depends on specific scenarios.

- If the left operand is a typed value (an integer value), then the type of the result is the same as the type of the left operand.
- If the left operand is an untyped value and the right operand is a constant, then the left operand will be always treated as an integer value, if its default type is not an integer type, it must be representable as an untyped integer and its default type will be viewed as `int`. For such cases, the result is also an untyped value and the default type of the result is the same as the left operand.
- If the left operand is an untyped value and the right operand is a non-constant integer, then the left operand will be first converted to the type it would assume if the bitwise shift operator operation were replaced by its left operand alone. The result is a typed value whose type is the assumed type.

Example:

```

1| func main() {
2|     // Three untyped values. Their default
3|     // types are: int, rune(int32), complex64.
4|     const X, Y, Z = 2, 'A', 3i
5|
6|     var a, b int = X, Y // two typed values.
7|
8|     // The type of d is the default type of Y: rune.
9|     d := X + Y
10|    // The type of e is the type of a: int.
11|    e := Y - a
12|    // The type of f is the types of a and b: int.
13|    f := a * b
14|    // The type of g is Z's default type: complex64.
15|    g := Z * Y
16|
17|    // Output: 2 65 (+0.000000e+000+3.000000e+000i)
18|    println(X, Y, Z)
19|    // Output: 67 63 130 (+0.000000e+000+1.950000e+002i)
20|    println(d, e, f, g)
21| }
```

Another example (bitwise shift operations):

```

1| const N = 2
2| // A is an untyped value (default type as int).
3| const A = 3.0 << N // A == 12
4| // B is typed value (type is int8).
5| const B = int8(3.0) << N // B == 12
6|
7| var m = uint(32)
8| // The following three lines are equivalent to
9| // each other. In the following two lines, the
10| // types of the two "1" are both deduced as
11| // int64, instead of int.
12| var x int64 = 1 << m
13| var y = int64(1 << m)
14| var z = int64(1) << m
15|
16| // The following line fails to compile.
17| /*
18| var _ = 1.23 << m // error: shift of type float64
19| */

```

The last rule for bitwise shift operator operation is to avoid the cases that some bitwise shift operations return different results on different architectures but the differences will not be detected in time. For example, if the operand `1` is deduced as `int` instead of `int64`, the bitwise operation at line 13 (or line 12) will return different results between 32-bit architectures (`0`) and 64-bit architectures (`0x100000000`), which may produce some bugs hard to detect.

One interesting consequence of the last rule for bitwise shift operator operation is shown in the following code snippet:

```

1| const n = uint(2)
2| var m = uint(2)
3|
4| // The following two lines compile okay.
5| var _ float64 = 1 << n
6| var _ = float64(1 << n)
7|
8| // The following two lines fail to compile.
9| var _ float64 = 1 << m
10| var _ = float64(1 << m)

```

The reason of the last two lines failing to compile is they are both equivalent to the followings two line:

```
1| var _ = float64(1) << m
2| var _ = 1.0 << m // error: shift of type float64
```

Another example:

```
1| package main
2|
3| const n = uint(8)
4| var m = uint(8)
5|
6| func main() {
7|     println(a, b) // 2 0
8| }
9|
10| var a byte = 1 << n / 128
11| var b byte = 1 << m / 128
```

The above program prints 2 0, because the last two lines are equivalent to

```
1| var a = byte(int(1) << n / 128)
2| var b = byte(1) << m / 128
```

About integer division and remainder operations

Assume x and y are two operands of the same integer type, the integer quotient q ($= x / y$) and remainder r ($= x \% y$) satisfy $x == q * y + r$, where $|r| < |y|$. If r is not zero, its sign is the same as x (the dividend). The result of x / y is truncated towards zero.

If the divisor y is a constant, it must not be zero. If the divisor is zero at run time and it is an integer, a run-time panic occurs. Panics are like exceptions in some other languages. We can learn more about panics in [this article](#) (§13).

Example:

```
1| println( 5/3,    5%3) // 1 2
2| println( 5/-3,   5%-3) // -1 2
3| println(-5/3,   -5%3) // -1 -2
4| println(-5/-3,  -5%-3) // 1 -2
5|
6| println(5.0 / 3.0)      // 1.666667
7| println((1-1i)/(1+1i)) // -1i
8|
9| var a, b = 1.0, 0.0
```

```

10| println(a/b, b/b) // +Inf NaN
11|
12| _ = int(a)/int(b) // compiles okay but panics at run time.
13|
14| // The following two lines fail to compile.
15| println(1.0/0.0) // error: division by zero
16| println(0.0/0.0) // error: division by zero

```

Using op= for binary arithmetic operators

For a binary arithmetic operator `op`, `x = x op y` can be shortened to `x op= y`. In the short form, `x` will be only evaluated once.

Example:

```

1| var a, b int8 = 3, 5
2| a += b
3| println(a) // 8
4| a *= a
5| println(a) // 64
6| a /= b
7| println(a) // 12
8| a %= b
9| println(a) // 2
10| b <= uint(a)
11| println(b) // 20

```

The increment ++ and decrement -- operators

Like many other popular languages, Go also supports the increment `++` and decrement `--` operators. However, operations using the two operators don't return any results, so such operations can not be used as [expressions](#) (§11). The only operand involved in such an operation must be a numeric value, the numeric value must not be a constant, and the `++` or `--` operator must follow the operand.

Example:

```

1| package main
2|
3| func main() {
4|     a, b, c := 12, 1.2, 1+2i
5|     a++ // ok. <=> a += 1 <=> a = a + 1

```

```

6|     b-- // ok. <=> b -= 1 <=> b = b - 1
7|     c++ // ok
8|
9|     // The following lines fail to compile.
10|    /*
11|     _ = a++
12|     _ = b--
13|     _ = c++
14|     ++a
15|     --b
16|     ++c
17|     */
18| }
```

String Concatenation Operator

As mentioned above, the addition operator can also be used as string concatenation.

Operator	Name	Requirements for the Two Operands
+	string concatenation	The two operands must be both values of the same string type.

The `op=` form also applies for the string concatenation operator.

Example:

```

1| println("Go" + "lang") // Golang
2| var a = "Go"
3| a += "lang"
4| println(a) // Golang
```

If one of the two operands of a string concatenation operation is a typed string, then the type of the result string is the same as the type of the typed string. If both of the two operands are untyped (constant) strings, the result is also an untyped string value.

Boolean (a.k.a. Logical) Operators

Go supports two boolean binary operators and one boolean unary operator:

Operator	Name	Requirements for Operand(s)
<code>&&</code>	boolean and (binary) a.k.a. conditional and	The two operands must be both values of the same boolean type.

Operator	Name	Requirements for Operand(s)
<code> </code>	boolean or (binary) a.k.a. conditional or	
<code>!</code>	boolean not (unary)	The type of the only operand must be a boolean type.

We can use the `!=` operator introduced in the next sub-section as the **boolean xor** operator.

Mechanism explanations:

// x	y	x && y	x y	!x	!y
true	true	true	true	false	false
true	false	false	true	false	true
false	true	false	true	true	false
false	false	false	false	true	true

If one of the two operands is a typed boolean, then the type of the result boolean is the same as the type of the typed boolean. If both of the two operands are untyped booleans, the result is also an untyped boolean value.

Comparison Operators

Go supports six comparison binary operators:

Operator	Name	Requirements for the Two Operands
<code>==</code>	equal to	Generally, the types of its two operands must be the same. For detailed rules, please read comparison rules in Go (§48).
<code>!=</code>	not equal to	
<code><</code>	less than	
<code><=</code>	less than or equal to	The two operands must be both values of the same integer type, floating-point type or string type.
<code>></code>	larger than	
<code>>=</code>	larger than or equal to	

The type of the result of any comparison operation is always an untyped boolean value. If both of the two operands of a comparison operation are constant, the result is also a constant (boolean) value.

Later, if we say two values are comparable, we mean they can be compared with the `==` and `!=` operators. We will learn that values of which types are not comparable later. Values of basic types are all comparable.

Please note that, not all real numbers can be accurately represented in memory, so comparing two floating-point (or complex) values may be not reliable. We should check whether or not the absolute value of the difference of two floating-point values is smaller than a small threshold to judge whether or not the two floating-point values are equal.

Operator Precedence

The following is the operator precedence in Go. Top ones have higher precedence. The operators in the same line have the same precedence. Like many other languages, `()` can be used to promote precedence.

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

One obvious difference to some other popular languages is that the precedence of `<<` and `>>` is higher than `+` and `-` in Go.

More About Constant Expressions

The following declared variable will be initialized as `2.2` instead of `2.7`. The reason is the precedence of the division operation is higher than the addition operation, and in the division operation, both `3` and `2` are viewed as integers. The evaluation result of `3/2` is `1`.

```
1| var x = 1.2 + 3/2
```

The two named constants declared in the following program are not equal. In the first declaration, both `3` and `2` are viewed as integers, however, they are both viewed as floating-point numbers in the second declaration.

```
1| package main
2|
3| const x = 3/2*0.1
4| const y = 0.1*3/2
```

```
5|  
6| func main() {  
7|     println(x) // +1.000000e-001  
8|     println(y) // +1.500000e-001  
9| }
```

More Operators

Same as C/C++, there are two pointer related operators, `*` and `&`. Yes the same operator symbols as the multiplication and bitwise-and operators. `&` is used to take the address of an addressable value, and `*` is used to dereference a pointer value. Unlike C/C++, in Go, values of pointer types don't support arithmetic operations. For more details, please read [pointers in Go](#) (§15) later.

There are some other operators in Go. They will be introduced and explained in other Go 101 articles.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Function Declarations and Function Calls

Except the operator operations introduced in the last article, function operations are another kind of popular operations in programming. Function operations are often called function calls. This article will introduce how to declare functions and call functions.

Function Declarations

Let's view a function declaration.

```

1| func SquaresOfSumAndDiff(a int64, b int64) (s int64, d int64) {
2|     x, y := a + b, a - b
3|     s = x * x
4|     d = y * y
5|     return // <=> return s, d
6| }
```

We can find that, a function declaration is composed of several portions. From left to right,

1. the first portion is the `func` keyword.
2. the next portion is the function name, which must be an identifier. Here the function name is `SquareOfSumAndDiff`.
3. the third portion is the input parameter declaration list, which is enclosed in a pair of `()`.
4. the fourth portion is the output (or return) result declaration list. Go functions can return multiple results. For this specified example, the result declaration list is also enclosed in a pair of `()`. However, if the list is blank or it is composed of only one anonymous result declaration, then the pair of `()` in result declaration list is optional (see below for details).
5. the last portion is the function body, which is enclosed in a pair of `{}`. In a function body, the `return` keyword is used to end the normal forward execution flow and enter the exiting phase (see the section after next) of a call of the function.

In the above example, each parameter and result declaration is composed of a name and a type (the type follows the name). We can view parameter and result declarations as standard variable declarations without the `var` keywords. The above declared function has two parameters, `a` and `b`, and has two results, `s` and `d`. All the types of the parameters and results are `int64`. Parameters and results are treated as local variables within their corresponding function bodies.

The names in the result declaration list of a function declaration can/must be present or absent all together. Either case is used common in practice. If a result is defined with a name, then the result is called a named result, otherwise, it is called an anonymous result.

When all the results in a function declaration are anonymous, then, within the corresponding function body, the `return` keyword must be followed by a sequence of return values, each of them corresponds to a result declaration of the function declaration. For example, the following function declaration is equivalent to the above one.

```
1| func SquaresOfSumAndDiff(a int64, b int64) (int64, int64) {
2|     return (a+b) * (a+b), (a-b) * (a-b)
3| }
```

In fact, if all the parameters are never used within the corresponding function body, the names in the parameter declaration list of a function declaration can be also be omitted all together. However, anonymous parameters are rarely used in practice.

Although it looks the parameter and result variables are declared outside of the body of a function declaration, they are viewed as general local variables within the function body. The difference is that local variables with non-blank names declared within a function body must be ever used in the function body. Non-blank names of top-level local variables, parameters and results in a function declaration can't be duplicated.

Go doesn't support default parameter values. The initial value of each result is the zero value of its type. For example, the following function will always print (and return) `0 false`.

```
1| func f() (x int, y bool) {
2|     println(x, y) // 0 false
3|     return
4| }
```

If the type portions of some successive parameters in a parameter declaration list are identical, then these parameters could share the same type portion in the parameter declaration list. The same is for result declaration lists. For example, the above two function declarations with the name `SquaresOfSumAndDiff` are equivalent to

```
1| func SquaresOfSumAndDiff(a, b int64) (s, d int64) {
2|     return (a+b) * (a+b), (a-b) * (a-b)
3|     // The above line is equivalent
4|     // to the following line.
5|     /*
6|     s = (a+b) * (a+b); d = (a-b) * (a-b); return
```

```

7|      */
8| }
```

Please note, even if both the two results are named, the `return` keyword can be followed with return values.

If the result declaration list in a function declaration only contains one anonymous result declaration, then the result declaration list doesn't need to be enclosed in a `()`. If the function declaration has no return results, then the result declaration list portion can be omitted totally. The parameter declaration list portion can never be omitted, even if the number of parameters of the declared function is zero.

Here are more function declaration examples.

```

1| func CompareLower4bits(m, n uint32) (r bool) {
2|     r = m&0xF > n&0xF
3|     return
4|     // The above two lines is equivalent to
5|     // the following line.
6|     /*
7|     return m&0xF > n&0xF
8|     */
9| }
10|
11| // This function has no parameters. The result
12| // declaration list is composed of only one
13| // anonymous result declaration, so it is not
14| // required to be enclosed within () .
15| func VersionString() string {
16|     return "go1.0"
17| }
18|
19| // This function has no results. And all of
20| // its parameters are anonymous, for we don't
21| // care about them. Its result declaration
22| // list is blank (so omitted).
23| func doNothing(string, int) {
24| }
```

One fact we have learned from the earlier articles in Go 101 is that the `main` entry function in each Go program is declared without parameters and results.

Please note that, functions must be directly declared at package level. In other words, a function can't be declared within the body of another function. In a later section, we will learn that we can

define anonymous functions in bodies of other functions. But anonymous functions are not function declarations.

Function Calls

A declared function can be called through its name plus an argument list. The argument list must be enclosed in a `()`. We often call this as argument passing (or parameter passing). Each single-value argument corresponds to (is passed to) a parameter.

Note: argument passing is also value assignments.

The type of an argument is not required to be identical with the corresponding parameter type. The only requirement for the argument is it must be [assignable](#) (§7) (a.k.a., implicitly convertible) to the corresponding parameter type.

If a function has return results, then each of its calls is viewed as an expression. If it returns multiple results, then each of its calls is viewed as a multi-value expression. A multi-value expression may be assigned to a list of destination values with the same count.

The following is a full example to show how to call some declared functions.

```

1| package main
2|
3| func SquaresOfSumAndDiff(a int64, b int64) (int64, int64) {
4|     return (a+b) * (a+b), (a-b) * (a-b)
5| }
6|
7| func CompareLower4bits(m, n uint32) (r bool) {
8|     r = m&0xF > n&0xF
9|     return
10| }
11|
12| // Initialize a package-level variable
13| // with a function call.
14| var v = VersionString()
15|
16| func main() {
17|     println(v) // v1.0
18|     x, y := SquaresOfSumAndDiff(3, 6)
19|     println(x, y) // 81 9
20|     b := CompareLower4bits(uint32(x), uint32(y))
21|     println(b) // false
22|     // "Go" is deduced as a string,

```

```

23|     // and 1 is deduced as an int32.
24|     doNothing("Go", 1)
25| }
26|
27| func VersionString() string {
28|     return "v1.0"
29| }
30|
31| func doNothing(string, int32) {
32| }
```

From the above example, we can learn that a function can be either declared before or after any of its calls.

Function calls can be deferred or invoked in new goroutines (green threads) in Go. Please read [a later article](#) (§13) for details.

Exiting (or Returning) Phase of a Function Call

In Go, besides the normal forward execution phase, a function call may undergo an exiting phase (also called returning phase). The exiting phase of a function call starts when the called function is returned. In other words, when a function call is returned, it is possible that it hasn't exited yet.

More detailed explanations for exiting phases of function calls can be found in [this article](#) (§31).

Anonymous Functions

Go supports anonymous functions. The definition of an anonymous function is almost the same as a function declaration, except there is no function name portion in the anonymous function definition.

An anonymous function can be called right after it is defined. Example:

```

1| package main
2|
3| func main() {
4|     // This anonymous function has no parameters
5|     // but has two results.
6|     x, y := func() (int, int) {
7|         println("This function has no parameters.")
8|         return 3, 4
9|     }()
10|    // Call it. No arguments are needed.
```

```

11| // The following anonymous function have no results.
12|
13| func(a, b int) {
14|     // The following line prints: a*a + b*b = 25
15|     println("a*a + b*b =", a*a + b*b)
16| }(x, y) // pass argument x and y to parameter a and b.
17|
18| func(x int) {
19|     // The parameter x shadows the outer x.
20|     // The following line prints: x*x + y*y = 32
21|     println("x*x + y*y =", x*x + y*y)
22| }(y) // pass argument y to parameter x.
23|
24| func() {
25|     // The following line prints: x*x + y*y = 25
26|     println("x*x + y*y =", x*x + y*y)
27| }()
28| }
```

Please note that, the last anonymous function is in the scope of the `x` and `y` variables declared above, it can use the two variables directly. Such functions are called closures. In fact, all custom functions in Go can be viewed as closures. This is why Go functions are as flexible as many dynamic languages.

Later, we will learn that an anonymous function can be assigned to a function value and can be called at any time later.

Built-in Functions

There are some built-in functions in Go, for example, the `println` and `print` functions. We can call these functions without importing any packages.

We can use the built-in `real` and `imag` functions to get the real and imaginary parts of a complex value. We can use the built-in `complex` function to produce a complex value. Please note, if any of the arguments of a call to any of the two functions are all constants, then the call will be evaluated at compile time, and the result value of the call is also a constant. In particular, if any of the arguments is an untyped constant, then the result value is also an untyped constant. The call is viewed as a constant expression.

Example:

```

1| // c is a untyped complex constant.
2| const c = complex(1.6, 3.3)
3|
4| // The results of real(c) and imag(c) are both
5| // untyped floating-point values. They are both
6| // deduced as values of type float32 below.
7| var a, b float32 = real(c), imag(c)
8|
9| // d is deduced as a typed value of type complex64.
10| // The results of real(d) and imag(d) are both
11| // typed values of type float32.
12| var d = complex(a, b)
13|
14| // e is deduced as a typed value of type complex128.
15| // The results of real(e) and imag(e) are both
16| // typed values of type float64.
17| var e = c

```

More built-in functions will be introduced in other Go 101 articles later.

More About Functions

There are more about function related concepts and details which are not touched in the current article. We can learn those concepts and details in the article [function types and values](#) (§20) later.

(The **Go 101** book is still being improved frequently from time to time. Please visit [go101.org](#) or follow [@go100and1](#) to get the latest news of this book. BTW,

Tapir, the author of the book, has developed several fun games. You can visit [tapirgames.com](#) to get more information about these games. Hope you enjoy them.)

Code Packages and Package Imports

Like many modern programming languages, Go code is also organized as code packages. To use the exported code elements (functions, types, variables and named constants, etc) in a specified package, the package must first be imported, except the `builtin` standard code package (which is a universe package). This article will explain code packages and package imports in Go.

Introduction of Package Import

Let's view a small program which imports a standard code package. (Assume the source code of this program is stored in a file named `simple-import-demo.go`.)

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     fmt.Println("Go has", 25, "keywords.")
7| }
```

Some explanations:

- The first line specifies the name of the package containing the source file `simple-import-demo.go`. The `main` entry function of a program must be put in a package named `main`.
- The third line imports the `fmt` standard package by using the `import` keyword. The identifier `fmt` is the package name. It is also used as the import name of, and represents, this standard package in the scope of containing source file. (Import names will be explained a below section.) There are many format functions declared in this standard package for other packages to use. The `Println` function is one of them. It will print the string representations of an arbitrary number of arguments to the standard output.
- The sixth line calls the `Println` function. Note that the function name is prefixed with a `fmt.` in the call, where `fmt` is the name of the package which contains the called function. The form `aImportName.AnExportedIdentifier` is called a [qualified identifier](#) . `AnExportedIdentifier` is called an unqualified identifier.
- A `fmt.Println` function call has no requirements for its arguments, so in this program, its three arguments will be deduced as values of their respective default types, `string`, `int` and `string`.

- For each `fmt.Println` call, a space character is inserted between each two consecutive string representations and a newline character is printed at the end.

Running this program, you will get the following output:

```
$ go run simple-import-demo.go
Go has 25 keywords.
```

Please note, only exported code elements in a package can be used in the source file which imports the package. An exported code element uses an [exported identifier](#) (§5) as its name. For example, the first character of the identifier `Println` is an upper case letter (so the `Println` function is exported), which is why the `Println` function declared in the `fmt` standard package can be used in the above example program.

The built-in functions, `print` and `println`, have similar functionalities as the corresponding functions in the `fmt` standard package. Built-in functions can be used without importing any packages.

Note, the two built-in functions, `print` and `println`, are not recommended to be used in the production environment, for they are not guaranteed to stay in the future Go versions.

All standard packages are listed [here ↗](#). We can also [run a local server](#) (§3) to view Go documentation.

A package import is also called an import declaration formally in Go. An import declaration is only visible to the source file which contains the import declaration. It is not visible to other source files in the same package.

Let's view another example:

```
1| package main
2|
3| import "fmt"
4| import "math/rand"
5|
6| func main() {
7|     fmt.Printf("Next pseudo-random number is %v.\n", rand.Uint32())
8| }
```

This example imports one more standard package, the `math/rand` package, which is a sub-package of the `math` standard package. This package provides some functions to produce pseudo-random numbers.

Some explanations:

- In this example, the package name `rand` is used as the import name of the imported `math/rand` standard package. A `rand.Uint32()` call will return a random `uint32` integer number.
- `Printf` is another commonly used function in the `fmt` standard package. A call to the `Printf` function must take at least one argument. The first argument of a `Printf` function call must be a `string` value, which specifies the format of the printed result. The `%v` in the first argument is called a format verb, it will be replaced with the string representation of the second argument. As we have learned in the article [basic types and their literals](#) (§6), the `\n` in a double-quoted string literal will be escaped as a newline character.

The above program will always output:

```
Next pseudo-random number is 2596996162.
```

Note: before Go 1.20, if we expect the above program to produce a different random number at each run, we should set a different seed by calling the `rand.Seed` function when the program just starts.

If multiple packages are imported into a source file, we can group them in one import declaration by enclosing them in a `()`.

Example:

```

1| package main
2|
3| // Multiple packages can be imported together.
4| import (
5|     "fmt"
6|     "math/rand"
7|     "time"
8| )
9|
10| func main() {
11|     // Set the random seed (only needed before Go 1.20).
12|     rand.Seed(time.Now().UnixNano())
13|     fmt.Printf("Next pseudo-random number is %v.\n", rand.Uint32())
14| }
```

Some explanations:

- this example imports one more package, the `time` standard package, which provides many time related utilities.

- function `time.Now()` returns the current time, as a value of type `time.Time`.
- `UnixNano` is a method of the `time.Time` type. The method call `aTime.UnixNano()` returns the number of nanoseconds elapsed since January 1, 1970 UTC to the time denoted by `aTime`. The return result is a value of type `int64`, which is also the parameter type of the `rand.Seed` function (note: the `rand.Seed` function has been deprecated since Go 1.20). Methods are special functions. We can learn methods in the article [methods in Go](#) (§22) for details later.

More About `fmt.Printf` Format Verbs

As the above has mentioned, if there is one format verb in the first argument of a `fmt.Printf` call, it will be replaced with the string representation of the second argument. In fact, there can be multiple format verbs in the first `string` argument. The second format verb will be replaced with the string representation of the third argument, and so on.

In Go 101, only the following listed format verbs will be used.

- `%v`, which will be replaced with the general string representation of the corresponding argument.
- `%T`, which will be replaced with the type name or type literal of the corresponding argument.
- `%x`, which will be replaced with the hex string representation of the corresponding argument. Note, the hex string representations for values of some kinds of types are not defined. Generally, the corresponding arguments of `%x` should be strings, integers, integer arrays or integer slices (arrays and slices will be explained in a later article).
- `%s`, which will be replaced with the string representation of the corresponding argument. The corresponding argument should be a string or byte slice.
- Format verb `%%` represents a percent sign.

An example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     a, b := 123, "Go"
7|     fmt.Printf("a == %v == 0x%x, b == %s\n", a, a, b)
8|     fmt.Printf("type of a: %T, type of b: %T\n", a, b)
9|     fmt.Printf("1% 50% 99%\n")
10| }
```

Output:

```
a == 123 == 0x7b, b == Go
type of a: int, type of b: string
1% 50% 99%
```

For more `Printf` format verbs, please read the online [fmt package documentation](#), or view the same documentation by running a local documentation server. We can also run `go doc fmt` to view the documentation of the `fmt` standard package, and run `go doc fmt.Printf` to view the documentation of the `fmt.Printf` function, in a terminal.

Package Folder, Package Import Path and Package Dependencies

A code package may consist of several source files. These source files are located in the same folder. The source files in a folder (not including subfolders) must belong to the same package. So, a folder corresponds to a code package, and vice versa. The folder containing the source files of a code package is called the folder of the package.

For Go Toolchain, a package whose import path containing an `internal` folder name is viewed as a special package. It can only be imported by the packages in and under the direct parent directory of the `internal` folder. For example, package `.../a/b/c/internal/d/e/f` and `.../a/b/c/internal` can only be imported by the packages whose import paths have a `.../a/b/c` prefix.

When one source file in a package imports another package, we say the importing package depends on the imported package.

Go doesn't support circular package dependencies. If package `a` depends on package `b` and package `b` depends on package `c`, then source files in package `c` can't import package `a` and `b`, and source files in package `b` can't import package `a`.

Surely, source files in a package can't, and don't need to, import the package itself.

Later, we will call the packages named with `main` and containing `main` entry functions as **program packages** (or **command packages**), and call other packages as **library packages**. Program packages are not importable. Each Go program should contain one and only one program package.

The name of the folder of a package is not required to be the same as the package name. However, for a library package, it will make package users confused if the name of the package is different

from the name of the folder of the package. The cause of the confusion is that the default import path of a package is the name of the package but what is contained in the import path of the package is the folder name of the package. So please try to make the two names identical for each library package.

On the other hand, it is recommended to give each program package folder a meaningful name other than its package name, `main`.

The `init` Functions

There can be multiple functions named as `init` declared in a package, even in a source code file. The functions named as `init` must have not any input parameters and return results.

Note, at the top package-level block, the `init` identifier can only be used in function declarations. We can't declare package-level variable/constants/types which names are `init`.

At run time, each `init` function will be (sequentially) invoked once and only once (before invoking the `main` entry function). So the meaning of the `init` functions are much like the static initializer blocks in Java.

Here is a simple example which contains two `init` functions:

```

1| package main
2|
3| import "fmt"
4|
5| func init() {
6|     fmt.Println("hi,", bob)
7| }
8|
9| func main() {
10|     fmt.Println("bye")
11| }
12|
13| func init() {
14|     fmt.Println("hello,", smith)
15| }
16|
17| func titledName(who string) string {
18|     return "Mr. " + who
19| }
```

```

20|
21| var bob, smith = titledName("Bob"), titledName("Smith")

```

The output of this program:

```

hi, Mr. Bob
hello, Mr. Smith
bye

```

Resource Initialization Order

At run time, a package will be loaded after all its dependency packages. Each package will be loaded once and only once.

All `init` functions in all involved packages in a program will be invoked sequentially. An `init` function in an importing package will be invoked after all the `init` functions declared in the dependency packages of the importing package for sure. All `init` functions will be invoked before invoking the `main` entry function.

The invocation order of the `init` functions in the same source file is from top to bottom. Go specification recommends, but doesn't require, to invoke the `init` functions in different source files of the same package by the alphabetical order of filenames of their containing source files. So it is not a good idea to have dependency relations between two `init` functions in two different source files.

All package-level variables declared in a package are initialized before any `init` function declared in the same package is invoked.

Go runtime will try to initialize package-level variables in a package by their declaration order, but a package-level variable will be initialized after all of its depended variables for sure. For example, in the following code snippet, the initializations the four package-level variables happen in the order `y`, `z`, `x`, and `w`.

```

1| func f() int {
2|     return z + y
3| }
4|
5| func g() int {
6|     return y/2
7| }
8|
9| var (

```

```

10|     w      = x
11|     x, y, z = f(), 123, g()
12| )

```

About more detailed rule of the initialization order of package-level variables, please read the article [expression evaluation order](#) (§33).

Full Package Import Forms

In fact, the full form of an import declaration is

```
import importname "path/to/package"
```

where `importname` is optional, its default value is the name (not the folder name) of the imported package.

In fact, in the above used import declarations, the `importname` portions are all omitted, for they are identical to the respective package names. These import declarations are equivalent to the following ones:

```

import fmt           // <=> import "fmt"
import rand "math/rand" // <=> import "math/rand"
import time "time"      // <=> import "time"

```

If the `importname` portion presents in an import declaration, then the prefix tokens used in qualified identifiers must be `importname` instead of the name of the imported package.

The full import declaration form is not used widely. However, sometimes we must use it. For example, if a source file imports two packages with the same name, to avoid making compiler confused, we must use the full import form to set a custom `importname` for at least one package in the two.

Here is an example of using full import declaration forms.

```

1| package main
2|
3| import (
4|     format "fmt"
5|     random "math/rand"
6|     "time"
7| )
8|
9| func main() {

```

```

10|     random.Seed(time.Now().UnixNano())
11|     format.Println("A random number: ", random.Uint32(), "\n")
12|
13|     // The following line fails to compile,
14|     // for "rand" is not identified.
15|     /*
16|         fmt.Println("A random number: ", rand.Uint32(), "\n")
17|     */
18| }
```

Some explanations:

- we must use `format` and `random` as the prefix token in qualified identifiers, instead of the real package names `fmt` and `rand`.
- `Print` is another function in the `fmt` standard package. Like `Println` function calls, a `Print` function call can take an arbitrary number of arguments. It will print the string representations of the passed arguments, one by one. If two consecutive arguments are both not string values, then a space character will be automatically inserted between them in the print result.

The `importname` in the full form import declaration can be a dot (.). Such imports are called dot imports. To use the exported elements in the packages being dot imported, the prefix part in qualified identifiers must be omitted.

Example:

```

1| package main
2|
3| import (
4|     . "fmt"
5|     . "time"
6| )
7|
8| func main() {
9|     println("Current time:", Now())
10| }
```

In the above example, `Println` instead of `fmt.Println`, and `Now` instead of `time.Now` must be used.

Generally, dot imports reduce code readability, so they are not recommended to be used in formal projects.

The `importname` in the full form import declaration can be the blank identifier (`_`). Such imports are called anonymous imports (some articles elsewhere also call them blank imports). The importing source files can't use the exported code elements in anonymously imported packages. The purpose of anonymous imports is to initialize the imported packages (each of `init` functions in the anonymously imported packages will be called once).

In the following example, all `init` functions declared in [the net/http/pprof standard package](#) will be called before the `main` entry function is called.

```
1| package main
2|
3| import _ "net/http/pprof"
4|
5| func main() {
6|     ... // do somethings
7| }
```

Each Non-Anonymous Import Must Be Used at Least Once

Except anonymous imports, other imports must be used at least once. For example, the following example fails to compile.

```
1| package main
2|
3| import (
4|     "net/http" // error: imported and not used
5|     . "time"   // error: imported and not used
6| )
7|
8| import (
9|     format "fmt" // okay: it is used once below
10|    _ "math/rand" // okay: it is not required to be used
11| )
12|
13| func main() {
14|     format.Println() // use the imported "fmt" package
15| }
```

Modules

A module is a collection of several packages. After being downloaded to local, these packages are all contained in the same folder, which is called the root folder of the module. A module may have many versions, which follow [Semantic Versioning](#) specification. About more modules related concepts and how to manage and use modules, please read the [official documentation](#).

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Expressions, Statements and Simple Statements

This article will introduce expressions and statements in Go.

Simply speaking, an expression represents a value and a statement represents an operation. However, in fact, some special expressions may be composed of and represent several values, and some statements may be composed of several sub operations/statements. By context, some statements can be also viewed as expressions.

Simple statements are some special statements. In Go, some portions of all kinds of control flows must be simple statements, and some portions must be expressions. Control flows will be introduced in the next Go 101 article.

This article will not make accurate definitions for expressions and statements. It is hard to achieve this. This article will only list some expression and statement cases. Not all kinds of expressions and statements will be covered in this article, but all kinds of simple statements will be listed.

Some Expression Cases

Most expressions in Go are single-value expressions. Each of them represents one value. Other expressions represent multiple values and they are named multi-value expressions.

In the scope of this document, when an expression is mentioned, we mean it is a single-value expression, unless otherwise specified.

Value literals, variables, and named constants are all single-value expressions, also called elementary expressions.

Operations (without the assignment parts) using the operators introduced in the article [common operators](#) (§8) are all single-value expressions.

If a function returns at least one result, then its calls (without the assignment parts) are expressions. In particular, if a function returns more than one results, then its calls belong to multi-value expressions. Calls to functions without results are not expressions.

Methods can be viewed as special functions. So the aforementioned function cases also apply to methods. Methods will be explained in detail in the article [method in Go](#) (§22) later.

In fact, later we will learn that custom functions, including methods, are all function values, so they are also (single-value) expressions. We will learn more about [function types and values](#) (§20) later.

Channel receive operations (without the assignment parts) are also expressions. Channel operations will be explained in the article [channels in Go](#) (§21) later.

Some expressions in Go, including channel receive operations, may have optional results in Go. Such expressions can present as both single-value and multi-value expressions, depending on different contexts. We can learn such expressions in other Go 101 articles later.

Simple Statement Cases

There are six kinds of simple statements.

1. short variable declaration forms
2. pure value assignments (not mixing with variable declarations), including `x op= y` operations.
3. function/method calls and channel receive operations. As mentioned in the last section, these simple statements can also be used as expressions.
4. channel send operations.
5. nothing (a.k.a., blank statements). We will learn some uses of blank statements in the next article.
6. `x++` and `x--`.

Again, channel receive and sent operations will be introduced in the article [channels in Go](#) (§21).

Note, `x++` and `x--` can't be used as expressions. And Go doesn't support the `++x` and `--x` syntax forms.

Some Non-Simple Statement Cases

An incomplete non-simple statements list:

- standard variable declaration forms. Yes, short variable declarations are simple statements, but standard ones are not.
- named constant declarations.
- custom type declarations.
- package import declarations.
- explicit code blocks. An explicit code block starts with a `{` and ends with a `}`. A code block may contain many sub-statements.

- function declarations. A function declaration may contain many sub-statements.
- control flows and code execution jumps. Please read [the next article](#) (§12) for details.
- return lines in function declarations.
- deferred function calls and goroutine creations. The two will be introduced in [the article after next](#) (§13).

Examples of Expressions and Statements

```

1| // Some non-simple statements.
2| import "time"
3| var a = 123
4| const B = "Go"
5| type Choice bool
6| func f() int {
7|     for a < 10 {
8|         break
9|     }
10|
11|     // This is an explicit code block.
12|     {
13|         // ...
14|     }
15|     return 567
16| }
17|
18| // Some simple statements:
19| c := make(chan bool) // channels will be explained later
20| a = 789
21| a += 5
22| a = f() // here f() is used as an expression
23| a++
24| a--
25| c <- true // a channel send operation
26| z := <-c // a channel receive operation used as the
27|           // source value in an assignment statement.
28|
29| // Some expressions:
30| 123
31| true
32| B
33| B + " language"
34| a - 789
35| a > 0 // an untyped boolean value

```

```
36| f      // a function value of type "func ()"  
37|  
38| // The following ones can be used as both  
39| // simple statements and expressions.  
40| f()  
41| <-c // a channel receive operation
```

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Basic Control Flows

The control flow code blocks in Go are much like other popular programming languages, but there are also many differences. This article will show these similarities and differences.

An Introduction of Control Flows in Go

There are three kinds of basic control flow code blocks in Go:

- `if-else` two-way conditional execution block.
- `for` loop block.
- `switch-case` multi-way conditional execution block.

There are also some control flow code blocks which are related to some certain kinds of types in Go.

- `for-range` loop block for [container](#) (§18) types.
- type-`switch` multi-way conditional execution block for [interface](#) (§23) types.
- `select-case` block for [channel](#) (§21) types.

Like many other popular languages, Go also supports `break`, `continue` and `goto` code execution jump statements. Besides these, there is a special code jump statement in Go, `fallthrough`.

Among the six kinds of control flow blocks, except the `if-else` control flow, the other five are called **breakable control flow blocks**. We can use `break` statements to make executions jump out of breakable control flow blocks.

`for` and `for-range` loop blocks are called **loop control flow blocks**. We can use `continue` statements to end a loop step in advance in a loop control flow block, i.e. `continue` to the next iteration of the loop.

Please note, each of the above mentioned control flow blocks is a statement, and it may contain many other sub-statements.

Above mentioned control flow statements are all the ones in narrow sense. The mechanisms introduced in the next article, [goroutines, deferred function calls and panic/recover](#) (§13), and the concurrency synchronization techniques introduced in the later article [concurrency synchronization overview](#) (§36) can be viewed as control flow statements in broad sense.

Only the basic control flow code blocks and code jump statements will be explained in the current article, other ones will be explained in many other Go 101 articles later.

if-else Control Flow Blocks

The full form of a `if-else` code block is like

```
1| if InitSimpleStatement; Condition {
2|     // do something
3| } else {
4|     // do something
5| }
```

`if` and `else` are keywords. Like many other programming languages, the `else` branch is optional.

The `InitSimpleStatement` portion is also optional. It must be a [simple statement](#) (§11) if it is present. If it is absent, we can view it as a blank statement (one kind of simple statements). In practice, `InitSimpleStatement` is often a short variable declaration or a pure assignment. A `Condition` must be an [expression](#) (§11) which results to a boolean value. The `Condition` portion can be enclosed in a pair of `()` or not, but it can't be enclosed together with the `InitSimpleStatement` portion.

If the `InitSimpleStatement` in a `if-else` block is present, it will be executed before executing other statements in the `if-else` block. If the `InitSimpleStatement` is absent, then the semicolon following it is optional.

Each `if-else` control flow forms one implicit code block, one `if` branch explicit code block and one optional `else` branch code block. The two branch code blocks are both nested in the implicit code block. Upon execution, if `Condition` expression results in `true`, then the `if` branch block will get executed, otherwise, the `else` branch block will get executed.

Example:

```
1| package main
2|
3| import (
4|     "fmt"
5|     "math/rand"
6|     "time"
7| )
8| 
```

```

9| func main() {
10|     rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
11|
12|     if n := rand.Int(); n%2 == 0 {
13|         fmt.Println(n, "is an even number.")
14|     } else {
15|         fmt.Println(n, "is an odd number.")
16|     }
17|
18|     n := rand.Int() % 2 // this n is not the above n.
19|     if n % 2 == 0 {
20|         fmt.Println("An even number.")
21|     }
22|
23|     if ; n % 2 != 0 {
24|         fmt.Println("An odd number.")
25|     }
26| }
```

If the `InitSimpleStatement` in a `if-else` code block is a short variable declaration, then the declared variables will be viewed as being declared in the top nesting implicit code block of the `if-else` code block.

An `else` branch code block can be implicit if the corresponding `else` is followed by another `if-else` code block, otherwise, it must be explicit.

Example:

```

1| package main
2|
3| import (
4|     "fmt"
5|     "time"
6| )
7|
8| func main() {
9|     if h := time.Now().Hour(); h < 12 {
10|         fmt.Println("Now is AM time.")
11|     } else if h > 19 {
12|         fmt.Println("Now is evening time.")
13|     } else {
14|         fmt.Println("Now is afternoon time.")
15|         h := h // the right one is declared above
16|         // The just new declared "h" variable
17|         // shadows the above same-name one.
```

```

18|     _ = h
19| }
20|
21| // h is not visible here.
22| }
```

for Loop Control Flow Blocks

The full form of a `for` loop block is:

```

1| for InitSimpleStatement; Condition; PostSimpleStatement {
2|     // do something
3| }
```

`for` is a keyword. The `InitSimpleStatement` and `PostSimpleStatement` portions must be both simple statements, and the `PostSimpleStatement` portion must not be a short variable declaration. `Condition` must be an expression which result is a boolean value. The three portions are all optional.

Unlike many other programming languages, the just mentioned three parts following the `for` keyword can't be enclosed in a pair of `()`.

Each `for` control flow forms at least two code blocks, one is implicit and one is explicit. The explicit one is nested in the implicit one.

The `InitSimpleStatement` in a `for` loop block will be executed (only once) before executing other statements in the `for` loop block.

The `Condition` expression will be evaluated at each loop step. If the evaluation result is `false`, then the loop will end. Otherwise the body (a.k.a., the explicit code block) of the loop will get executed.

The `PostSimpleStatement` will be executed at the end of each loop step.

A `for` loop example. The example will print the integers from `0` to `9`.

```

1| for i := 0; i < 10; i++ {
2|     fmt.Println(i)
3| }
```

If the `InitSimpleStatement` and `PostSimpleStatement` portions are both absent (just view them as blank statements), their nearby two semicolons can be omitted. The form is called as

condition-only `for` loop form. It is the same as the `while` loop in other languages.

```

1| var i = 0
2| for ; i < 10; {
3|     fmt.Println(i)
4|     i++
5|
6| for i < 20 {
7|     fmt.Println(i)
8|     i++
9|

```

If the Condition portion is absent, compilers will view it as `true`.

```

1| for i := 0; ; i++ { // <=> for i := 0; true; i++ {
2|     if i >= 10 {
3|         // "break" statement will be explained below.
4|         break
5|
6|     fmt.Println(i)
7|
8|
9| // The following 4 endless loops are
10| // equivalent to each other.
11| for ; true; {
12| }
13| for true {
14| }
15| for ; ; {
16| }
17| for {
18| }

```

If the `InitSimpleStatement` in a `for` block is a short variable declaration statement, then the declared variables will be viewed as being declared in the top nesting implicit code block of the `for` block. For example, the following code snippet prints `012` instead of `0`.

```

1| for i := 0; i < 3; i++ {
2|     fmt.Print(i)
3|     // The left i is a new declared variable,
4|     // and the right i is the loop variable.
5|     i := i
6|     // The new declared variable is modified, but
7|     // the old one (the loop variable) is not yet.
8|     i = 10

```

```

9|     _ = i
10| }
```

A `break` statement can be used to make execution jump out of the `for` loop control flow block in advance, if the `for` loop control flow block is the innermost breakable control flow block containing the `break` statement.

```

1| i := 0
2| for {
3|     if i >= 10 {
4|         break
5|     }
6|     fmt.Println(i)
7|     i++
8| }
```

A `continue` statement can be used to end the current loop step in advance (`PostSimpleStatement` will still get executed), if the `for` loop control flow block is the innermost loop control flow block containing the `continue` statement. For example, the following code snippet will print `13579`.

```

1| for i := 0; i < 10; i++ {
2|     if i % 2 == 0 {
3|         continue
4|     }
5|     fmt.Print(i)
6| }
```

switch-case Control Flow Blocks

`switch-case` control flow block is one kind of conditional execution control flow blocks.

The full form a `switch-case` block is

```

1| switch InitSimpleStatement; CompareOperand0 {
2| case CompareOperandList1:
3|     // do something
4| case CompareOperandList2:
5|     // do something
6| ...
7| case CompareOperandListN:
8|     // do something
9| default:
```

```

10| // do something
11| }
```

In the full form,

- `switch`, `case` and `default` are three keywords.
- The `InitSimpleStatement` portion must be a simple statement. The `CompareOperand0` portion is an expression which is viewed as a typed value (if it is an untyped value, then it is viewed as a type value of its default type), hence it can't be an untyped `nil`. `CompareOperand0` is called as switch expression in Go specification.
- Each of the `CompareOperandListX` (`X` may represent from 1 to N) portions must be a comma separated expression list. Each of these expressions shall be comparable with `CompareOperand0`. Each of these expressions is called as a case expression in Go specification. If a case expression is an untyped value, then it must be implicitly convertible to the type of the switch expression in the same switch-case control flow. If the conversion is impossible to achieve, compilation fails.

Each `case CompareOperandListX:` or `default:` opens (and is followed by) an implicit code block. The implicit code block and that `case CompareOperandListX:` or `default:` forms a branch. Each such branch is optional to be present. We call an implicit code block in such a branch as a branch code block later.

There can be at most one `default` branch in a switch-case control flow block.

Besides the branch code blocks, each switch-case control flow forms two code blocks, one is implicit and one is explicit. The explicit one is nested in the implicit one. All the branch code blocks are nested in the explicit one (and nested in the implicit one indirectly).

switch-case control flow blocks are breakable, so `break` statements can also be used in any branch code block in a switch-case control flow block to make execution jump out of the switch-case control flow block in advance.

The `InitSimpleStatement` will get executed firstly when a switch-case control flow gets executed. It will get executed only once during executing the switch-case control flow. After the `InitSimpleStatement` gets executed, the switch expression `CompareOperand0` will be evaluated and only evaluated once. The evaluation result is always a typed value. The evaluation result will be compared (by using the `==` operator) with the evaluation result of each case expression in the `CompareOperandListX` expression lists, from top to down and from left to right. If a case expression is found to be equal to `CompareOperand0`, the comparison process stops and the corresponding branch code block of the expression will be executed. If none case expressions

are found to be equal to `CompareOperand0`, the default branch code block (if it is present) will get executed.

A switch-case control flow example:

```

1| package main
2|
3| import (
4|     "fmt"
5|     "math/rand"
6|     "time"
7| )
8|
9| func main() {
10|     rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
11|     switch n := rand.Intn(100); n%9 {
12|     case 0:
13|         fmt.Println(n, "is a multiple of 9.")
14|
15|         // Different from many other languages,
16|         // in Go, the execution will automatically
17|         // jumps out of the switch-case block at
18|         // the end of each branch block.
19|         // No "break" statement is needed here.
20|     case 1, 2, 3:
21|         fmt.Println(n, "mod 9 is 1, 2 or 3.")
22|         // Here, this "break" statement is nonsense.
23|         break
24|     case 4, 5, 6:
25|         fmt.Println(n, "mod 9 is 4, 5 or 6.")
26|     // case 6, 7, 8:
27|         // The above case line might fail to compile,
28|         // for 6 is duplicate with the 6 in the last
29|         // case. The behavior is compiler dependent.
30|     default:
31|         fmt.Println(n, "mod 9 is 7 or 8.")
32|     }
33| }
```

The `rand.Intn` function returns a non-negative `int` random value which is smaller than the specified argument.

Note, if any two case expressions in a switch-case control flow can be detected to be equal at compile time, then a compiler may reject the latter one. For example, the standard Go compiler

thinks the case 6, 7, 8 line in the above example is invalid if that line is not commented out. But other compilers may think that line is okay. In fact, the current standard Go compiler (version 1.21.n) [allows duplicate boolean case expressions](#), and gccgo (v8.2) allows both duplicate boolean and string case expressions.

As the comments in the above example describes, unlike many other languages, in Go, at the end of each branch code block, the execution will automatically break out of the corresponding switch-case control block. Then how to let the execution slip into the next branch code block? Go provides a `fallthrough` keyword to do this task. For example, in the following example, every branch code block will get executed, by their orders, from top to down.

```

1| rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
2| switch n := rand.Intn(100) % 5; n {
3| case 0, 1, 2, 3, 4:
4|     fmt.Println("n =", n)
5|     // The "fallthrough" statement makes the
6|     // execution slip into the next branch.
7|     fallthrough
8| case 5, 6, 7, 8:
9|     // A new declared variable also called "n",
10|    // it is only visible in the current
11|    // branch code block.
12|    n := 99
13|    fmt.Println("n =", n) // 99
14|    fallthrough
15| default:
16|     // This "n" is the switch expression "n".
17|     fmt.Println("n =", n)
18| }
```

Please note,

- a `fallthrough` statement must be the final statement in a branch.
- a `fallthrough` statement can't show up in the final branch in a switch-case control flow block.

For example, the following `fallthrough` uses are all illegal.

```

1| switch n := rand.Intn(100) % 5; n {
2| case 0, 1, 2, 3, 4:
3|     fmt.Println("n =", n)
4|     // The if-block, not the fallthrough statement,
5|     // is the final statement in this branch.
6|     if true {
```

```

7|     fallthrough // error: not the final statement
8| }
9| case 5, 6, 7, 8:
10|    n := 99
11|    fallthrough // error: not the final statement
12|    _ = n
13| default:
14|    fmt.Println(n)
15|    fallthrough // error: show up in the final branch
16| }
```

The `InitSimpleStatement` and `CompareOperand0` portions in a `switch-case` control flow are both optional. If the `CompareOperand0` portion is absent, it will be viewed as `true`, a typed value of the built-in type `bool`. If the `InitSimpleStatement` portion is absent, the semicolon following it can be omitted.

And as above has mentioned, all branches are optional. So the following code blocks are all legal, all of them can be viewed as no-ops.

```

1| switch n := 5; n {
2| }
3|
4| switch 5 {
5| }
6|
7| switch _ = 5; {
8| }
9|
10| switch {
11| }
```

For the latter two `switch-case` control flow blocks in the last example, as above has mentioned, each of the absent `CompareOperand0` portions is viewed as a typed value `true` of the built-in type `bool`. So the following code snippet will print `hello`.

```

1| switch { // <=> switch true {
2| case true: fmt.Println("hello")
3| default: fmt.Println("bye")
4| }
```

Another obvious difference from many other languages is the order of the `default` branch in a `switch-case` control flow block can be arbitrary. For example, the following three `switch-case` control flow blocks are equivalent to each other.

```

1| switch n := rand.Intn(3); n {
2| case 0: fmt.Println("n == 0")
3| case 1: fmt.Println("n == 1")
4| default: fmt.Println("n == 2")
5|
6|
7| switch n := rand.Intn(3); n {
8| default: fmt.Println("n == 2")
9| case 0: fmt.Println("n == 0")
10| case 1: fmt.Println("n == 1")
11|
12|
13| switch n := rand.Intn(3); n {
14| case 0: fmt.Println("n == 0")
15| default: fmt.Println("n == 2")
16| case 1: fmt.Println("n == 1")
17|

```

goto Statement and Label Declaration

Like many other languages, Go also supports `goto` statement. A `goto` keyword must be followed by a label to form a statement. A label is declared with the form `LabelName:`, where `LabelName` must be an identifier. A label which name is not the blank identifier must be used at least once.

A `goto` statement will make the execution jump to the next statement following the declaration of the label used in the `goto` statement. So a label declaration must be followed by one statement.

A label must be declared within a function body. A use of a label can appear before or after the declaration of the label. But a label is not visible (and can't appear) outside the innermost code block the label is declared in.

The following example uses a `goto` statement and a label to implement a loop control flow.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     i := 0
7|
8|     Next: // here, a label is declared.
9|     fmt.Println(i)
10|    i++

```

```

11|     if i < 5 {
12|         goto Next // execution jumps
13|     }
14| }
```

As mentioned above, a label is not visible (and can't appear) outside the innermost code block the label is declared in. So the following example fails to compile.

```

1| package main
2|
3| func main() {
4|     goto Label1 // error
5|     {
6|         Label1:
7|         goto Label2 // error
8|     }
9|     {
10|         Label2:
11|     }
12| }
```

Note that, if a label is declared within the scope of a variable, then the uses of the label can't appear before the declaration of the variable. Identifier scopes will be explained in the article [blocks and scopes in Go](#) (§32) later.

The following example also fails to compile.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     i := 0
7|     Next:
8|     if i >= 5 {
9|         // error: jumps over declaration of k
10|        goto Exit
11|    }
12|
13|    k := i + i
14|    fmt.Println(k)
15|    i++
16|    goto Next
17|
18| // This label is declared in the scope of k,
```

```

19| // but its use is outside of the scope of k.
20| Exit:
21| }
```

The just mentioned rule [may change later ↗](#). Currently, to make the above code compile okay, we must adjust the scope of the variable `k`. There are two ways to fix the problem in the last example.

One way is to shrink the scope of the variable `k`.

```

1| func main() {
2|     i := 0
3| Next:
4|     if i >= 5 {
5|         goto Exit
6|     }
7|     // Create an explicit code block to
8|     // shrink the scope of k.
9|     {
10|         k := i + i
11|         fmt.Println(k)
12|     }
13|     i++
14|     goto Next
15| Exit:
16| }
```

The other way is to enlarge the scope of the variable `k`.

```

1| func main() {
2|     var k int // move the declaration of k here.
3|     i := 0
4| Next:
5|     if i >= 5 {
6|         goto Exit
7|     }
8|
9|     k = i + i
10|    fmt.Println(k)
11|    i++
12|    goto Next
13| Exit:
14| }
```

break and continue Statements With Labels

A `goto` statement must contain a label. A `break` or `continue` statement can also contain a label, but the label is optional. Generally, `break` containing labels are used in nested breakable control flow blocks and `continue` statements containing labels are used in nested loop control flow blocks.

If a `break` statement contains a label, the label must be declared just before a breakable control flow block which contains the `break` statement. We can view the label name as the name of the breakable control flow block. The `break` statement will make execution jump out of the breakable control flow block, even if the breakable control flow block is not the innermost breakable control flow block containing `break` statement.

If a `continue` statement contains a label, the label must be declared just before a loop control flow block which contains the `continue` statement. We can view the label name as the name of the loop control flow block. The `continue` statement will end the current loop step of the loop control flow block in advance, even if the loop control flow block is not the innermost loop control flow block containing the `continue` statement.

The following is an example of using `break` and `continue` statements with labels.

```

1| package main
2|
3| import "fmt"
4|
5| func FindSmallestPrimeLargerThan(n int) int {
6| Outer:
7|     for n++; ; n++{
8|         for i := 2; ; i++ {
9|             switch {
10|                 case i * i > n:
11|                     break Outer
12|                 case n % i == 0:
13|                     continue Outer
14|                 }
15|             }
16|         }
17|     return n
18| }
19|
20| func main() {
21|     for i := 90; i < 100; i++ {
22|         n := FindSmallestPrimeLargerThan(i)
23|         fmt.Println("The smallest prime number larger than ")
24|         fmt.Println(i, "is", n)

```

25 | }

26 | }

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

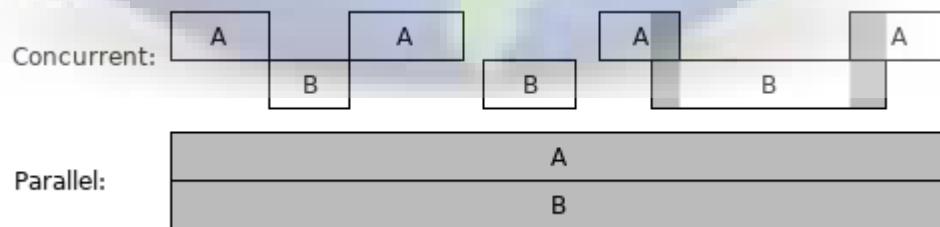
Goroutines, Deferred Function Calls and Panic/Recover

This article will introduce goroutines and deferred function calls. Goroutine and deferred function call are two unique features in Go. This article also explains panic and recover mechanism. Not all knowledge relating to these features is covered in this article, more will be introduced in future articles.

Goroutines

Modern CPUs often have multiple cores, and some CPU cores support hyper-threading. In other words, modern CPUs can process multiple instruction pipelines simultaneously. To fully use the power of modern CPUs, we need to do concurrent programming in coding our programs.

Concurrent computing is a form of computing in which several computations are executed during overlapping time periods. The following picture depicts two concurrent computing cases. In the picture, A and B represent two separate computations. The second case is also called parallel computing, which is special concurrent computing. In the first case, A and B are only in parallel during a small piece of time.



Concurrent computing may happen in a program, a computer, or a network. In Go 101, we only talk about program-scope concurrent computing. Goroutine is the Go way to create concurrent computations in Go programming.

Goroutines are also often called green threads. Green threads are maintained and scheduled by the language runtime instead of the operating systems. The cost of memory consumption and context switching, of a goroutine is much lesser than an OS thread. So, it is not a problem for a Go program to maintain tens of thousands goroutines at the same time, as long as the system memory is sufficient.

Go doesn't support the creation of system threads in user code. So, using goroutines is the only way to do (program scope) concurrent programming in Go.

Each Go program starts with only one goroutine, we call it the main goroutine. A goroutine can create new goroutines. It is super easy to create a new goroutine in Go, just use the keyword `go` followed by a function call. The function call will then be executed in a newly created goroutine. The newly created goroutine will exit alongside the exit of the called function.

All the result values of a goroutine function call (if the called function returns values) must be discarded in the function call statement. The following is an example which creates two new goroutines in the main goroutine. In the example, `time.Duration` is a custom type defined in the `time` standard package. Its underlying type is the built-in type `int64`. Underlying types will be explained in [the next article](#) (§14).

```

1| package main
2|
3| import (
4|     "log"
5|     "math/rand"
6|     "time"
7| )
8|
9| func SayGreetings(greeting string, times int) {
10|     for i := 0; i < times; i++ {
11|         log.Println(greeting)
12|         d := time.Second * time.Duration(rand.Intn(5)) / 2
13|         time.Sleep(d) // sleep for 0 to 2.5 seconds
14|     }
15| }
16|
17| func main() {
18|     rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
19|     log.SetFlags(0)
20|     go SayGreetings("hi!", 10)
21|     go SayGreetings("hello!", 10)
22|     time.Sleep(2 * time.Second)
23| }
```

Quite easy. Right? We do concurrent programming now! The above program may have three user-created goroutines running simultaneously at its peak during run time. Let's run it. One possible output result:

```

hi!
hello!
hello!
hello!
```

```
hello!
hi!
```

When the main goroutine exits, the whole program also exits, even if there are still some other goroutines which have not exited yet.

Unlike previous articles, this program uses the `Println` function in the `log` standard package instead of the corresponding function in the `fmt` standard package. The reason is the print functions in the `log` standard package are synchronized (the next section will explain what synchronizations are), so the texts printed by the two goroutines will not be messed up in one line (though the chance of the printed texts being messed up by using the print functions in the `fmt` standard package is very small for this specific program).

Concurrency Synchronization

Concurrent computations may share resources, generally memory resource. The following are some circumstances that may occur during concurrent computing:

- In the same period that one computation is writing data to a memory segment, another computation is reading data from the same memory segment. Then the integrity of the data read by the other computation might be not preserved.
- In the same period that one computation is writing data to a memory segment, another computation is also writing data to the same memory segment. Then the integrity of the data stored at the memory segment might be not preserved.

These circumstances are called data races. One of the duties in concurrent programming is to control resource sharing among concurrent computations, so that data races will never happen. The ways to implement this duty are called concurrency synchronizations, or data synchronizations, which will be introduced one by one in later Go 101 articles.

Other duties in concurrent programming include

- determine how many computations are needed.
- determine when to start, block, unblock and end a computation.
- determine how to distribute workload among concurrent computations.

The program shown in the last section is not perfect. The two new goroutines are intended to print ten greetings each. However, the main goroutine will exit in two seconds, so many greetings don't have a chance to get printed. How to let the main goroutine know when the two new goroutines have both finished their tasks? We must use something called concurrency synchronization techniques.

Go supports several [concurrency synchronization techniques](#) (§36). Among them, [the channel technique](#) (§21) is the most unique and popularly used one. However, for simplicity, here we will use another technique, the `WaitGroup` type in the `sync` standard package, to synchronize the executions between the two new goroutines and the main goroutine.

The `WaitGroup` type has three methods (special functions, will be explained later): `Add`, `Done` and `Wait`. This type will be explained in detail later in another article. Here we can simply think

- the `Add` method is used to register the number of new tasks.
- the `Done` method is used to notify that a task is finished.
- and the `Wait` method makes the caller goroutine become blocking until all registered tasks are finished.

Example:

```

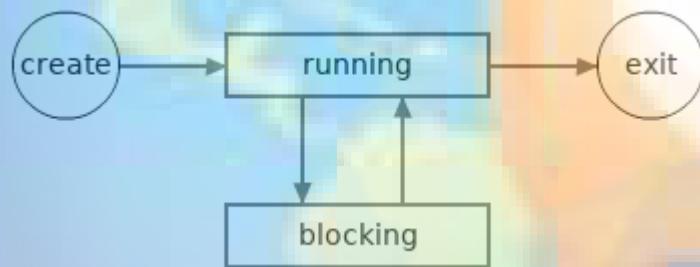
1| package main
2|
3| import (
4|     "log"
5|     "math/rand"
6|     "time"
7|     "sync"
8| )
9|
10| var wg sync.WaitGroup
11|
12| func SayGreetings(greeting string, times int) {
13|     for i := 0; i < times; i++ {
14|         log.Println(greeting)
15|         d := time.Second * time.Duration(rand.Intn(5)) / 2
16|         time.Sleep(d)
17|     }
18|     // Notify a task is finished.
19|     wg.Done() // <=> wg.Add(-1)
20| }
21|
22| func main() {
23|     rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
24|     log.SetFlags(0)
25|     wg.Add(2) // register two tasks.
26|     go SayGreetings("hi!", 10)
27|     go SayGreetings("hello!", 10)
28|     wg.Wait() // block until all tasks are finished.
29| }
```

Run it, we can find that, before the program exits, each of the two new goroutines prints ten greetings.

Goroutine States

The last example shows that a live goroutine may stay in (and switch between) two states, **running** and **blocking**. In that example, the main goroutine enters the blocking state when the `wg.Wait` method is called, and enter running state again when the other two goroutines both finish their respective tasks.

The following picture depicts a possible lifecycle of a goroutine.



Note, a goroutine is still considered to be 'running' if it is asleep (after calling `time.Sleep` function) or awaiting the response of a system call or a network connection.

When a new goroutine is created, it will enter the 'running' state automatically. Goroutines can only exit from running state, and never from blocking state. If, for any reason, a goroutine stays in blocking state forever, then it will never exit. Such cases, except some rare ones, should be avoided in concurrent programming.

A blocking goroutine can only be unblocked by an operation made in another goroutine. If all goroutines in a Go program are in blocking state, then all of them will stay in blocking state forever. This can be viewed as an overall deadlock. When this happens in a program, the standard Go runtime will try to crash the program.

The following program will crash, after two seconds:

```

1| package main
2|
3| import (
4|     "sync"
5|     "time"
6| )
7|
8| var wg sync.WaitGroup
  
```

```

9|
10| func main() {
11|     wg.Add(1)
12|     go func() {
13|         time.Sleep(time.Second * 2)
14|         wg.Wait()
15|     }()
16|     wg.Wait()
17| }

```

The output:

```

fatal error: all goroutines are asleep - deadlock!
...

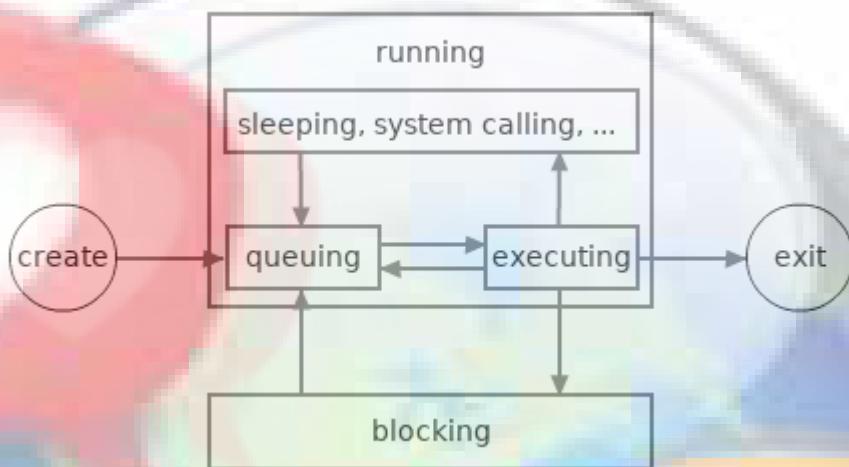
```

Later, we will learn more operations which will make goroutines enter blocking state.

Goroutine Schedule

Not all goroutines in running state are being executed at a given time. At any given time, the maximum number of goroutines being executed will not exceed the number of logical CPUs available for the current program. We can call the [runtime.NumCPU](#) function to get the number of logical CPUs available for the current program. Each logical CPU can only execute one goroutine at any given time. Go runtime must frequently switch execution contexts between goroutines to let each running goroutine have a chance to execute. This is similar to how operating systems switch execution contexts between OS threads.

The following picture depicts a more detailed possible lifecycle for a goroutine. In the picture, the running state is divided into several more sub-states. A goroutine in the queuing sub-state is waiting to be executed. A goroutine in the executing sub-state may enter the queuing sub-state again when it has been executed for a while (a very small piece of time).



Please note, for simplicity, the sub-states shown in the above picture will be not mentioned in other articles in Go 101. And again, in Go 101, the sleeping and system calling sub-states are not viewed as sub-states of the blocking state.

The standard Go runtime adopts the [M-P-G model](#) to do the goroutine schedule job, where **M** represents OS threads, **P** represents logical/virtual processors (not logical CPUs) and **G** represents goroutines. Most schedule work is made by logical processors (**Ps**), which act as brokers by attaching goroutines (**Gs**) to OS threads (**Ms**). Each OS thread can only be attached to at most one goroutine at any given time, and each goroutine can only be attached to at most one OS thread at any given time. A goroutine can only get executed when it is attached to an OS thread. A goroutine which has been executed for a while will try to detach itself from the corresponding OS thread, so that other running goroutines can have a chance to get attached and executed.

At runtime, we can call the [`runtime.GOMAXPROCS`](#) function to get and set the number of logical processors (**Ps**). For the standard Go runtime, before Go 1.5, the default initial value of this number is 1, but since Go 1.5, the default initial value of this number is equal to the number of logical CPUs available for the current running program. The default initial value (the number of logical CPUs) is the best choice for most programs. But for some file IO heavy programs, a `GOMAXPROCS` value larger than `runtime.NumCPU()` may be helpful.

The default initial value of `runtime.GOMAXPROCS` can also be set through the `GOMAXPROCS` environment variable.

At any time, the number of goroutines in the executing sub-state is no more than the smaller one of `runtime.NumCPU` and `runtime.GOMAXPROCS`.

Deferred Function Calls

A deferred function call is a function call which follows a `defer` keyword. The `defer` keyword and the deferred function call together form a `defer` statement. Like goroutine function calls, all the results of the function call (if the called function has return results) must be discarded in the function call statement.

When a `defer` statement is executed, the deferred function call is not executed immediately. Instead, it is pushed into a deferred call queue maintained by its caller goroutine. After a function call `fc(...)` returns (but has not fully existed yet) and enters its [exiting phase](#) (§9), all the deferred function calls pushed into the deferred call queue during executing the function call will be removed from the deferred call queue and executed, in first-in, last-out order, that is, the reverse of the order in which they were pushed into the deferred call queue. Once all these deferred calls are executed, the function call `fc(...)` fully exits.

Here is a simple example to show how to use deferred function calls.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     defer fmt.Println("The third line.")
7|     defer fmt.Println("The second line.")
8|     fmt.Println("The first line.")
9| }
```

The output:

```

The first line.
The second line.
The third line.
```

Here is another example which is a little more complex. The example will print 0 to 9, each per line, by their natural order.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     defer fmt.Println("9")
7|     fmt.Println("0")
8|     defer fmt.Println("8")
9|     fmt.Println("1")
10|    if false {
```

```

11|     defer fmt.Println("not reachable")
12| }
13| defer func() {
14|     defer fmt.Println("7")
15|     fmt.Println("3")
16|     defer func() {
17|         fmt.Println("5")
18|         fmt.Println("6")
19|     }()
20|     fmt.Println("4")
21| }()
22| fmt.Println("2")
23| return
24| defer fmt.Println("not reachable")
25| }
```

Deferred Function Calls Can Modify the Named Return Results of Nesting Functions

For example,

```

1| package main
2|
3| import "fmt"
4|
5| func Triple(n int) (r int) {
6|     defer func() {
7|         r += n // modify the return value
8|     }()
9|
10|    return n + n // <=> r = n + n; return
11| }
12|
13| func main() {
14|     fmt.Println(Triple(5)) // 15
15| }
```

The Evaluation Moment of the Arguments of Deferred Function Calls

The arguments of a deferred function call are all evaluated at the moment when the corresponding defer statement is executed (a.k.a. when the deferred call is pushed into the deferred call queue). The evaluation results are used when the deferred call is executed later during the existing phase of the surrounding call (the caller of the deferred call).

The expressions within the body of an anonymous function call, whether the call is a general call or a deferred/goroutine call, are evaluated during the anonymous function call is executed.

Here is an example.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     func() {
7|         for i := 0; i < 3; i++ {
8|             defer fmt.Println("a:", i)
9|         }
10|    }()
11|    fmt.Println()
12|    func() {
13|        for i := 0; i < 3; i++ {
14|            defer func() {
15|                fmt.Println("b:", i)
16|            }()
17|        }
18|    }()
19| }
```

Run it. The output:

```

a: 2
a: 1
a: 0

b: 3
b: 3
b: 3
```

The first loop prints `i` as 2, 1 and 0 as a sequence. The second loop always prints `i` as 3, because when the three `fmt.Println` calls within the anonymous function are invoked, the value of the loop variable `i` has changed to 3.

To make the second loop print the same result as the first one, we can modify the second loop as

```

1|     for i := 0; i < 3; i++ {
2|         defer func(i int) {
3|             // The "i" is the input parameter.
4|             fmt.Println("b:", i)
5|         }(i)
6|     }

```

or

```

1|     for i := 0; i < 3; i++ {
2|         i := i // <=> var i = i
3|         defer func() {
4|             // The "i" is not the loop variable.
5|             fmt.Println("b:", i)
6|         }()
7|     }

```

The same argument valuation moment rules also apply to goroutine function calls. The following program will output 123 789.

```

1| package main
2|
3| import "fmt"
4| import "time"
5|
6| func main() {
7|     var a = 123
8|     go func(x int) {
9|         time.Sleep(time.Second)
10|        fmt.Println(x, a) // 123 789
11|    }(a)
12|
13|    a = 789
14|
15|    time.Sleep(2 * time.Second)
16| }

```

By the way, it is not a good idea to do synchronizations by using `time.Sleep` calls in formal projects. If the program runs on a computer which CPUs are occupied by many other programs running on the computer, the newly created goroutine may never get a chance to execute before the program exits. We should use the concurrency synchronization techniques introduced in the article [concurrency synchronization overview](#) (§36) to do synchronizations in formal projects.

The Necessity of the Deferred Function Feature

In the above examples, the deferred function calls are not absolutely necessary. However, the deferred function call feature is a necessary feature for the panic and recover mechanism which will be introduced below.

Deferred function calls can also help us write cleaner and more robust code. We can read more code examples that make use of deferred function calls and learn more details on deferred function calls in the article [more about deferred functions](#) (§29) later. For now, we will explore the importance of deferred functions for panic and recovery.

Panic and Recover

Go doesn't support exception throwing and catching, instead explicit error handling is preferred to use in Go programming. In fact, Go supports an exception throw/catch alike mechanism. The mechanism is called panic/recover.

We can call the built-in `panic` function to create a panic to make the current goroutine enter panicking status.

Panicking is another way to make a function return. Once a panic occurs in a function call, the function call returns immediately and enters its exiting phase.

By calling the built-in `recover` function in a deferred call, an alive panic in the current goroutine can be removed so that the current goroutine will enter normal calm status again.

If a panicking goroutine exits without being recovered, it will make the whole program crash.

The built-in `panic` and `recover` functions are declared as

```
1| func panic(v interface{})  
2| func recover() interface{}
```

Interface types and values will be explained in the article [interfaces in Go](#) (§23) later. Here, we just need to know that the blank interface type `interface{}` can be viewed as the `any` type or the `Object` type in many other languages. In other words, we can pass a value of any type to a `panic` function call.

The value returned by a `recover` function call is the value a `panic` function call consumed.

The example below shows how to create a panic and how to recover from it.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     defer func() {
7|         fmt.Println("exit normally.")
8|     }()
9|     fmt.Println("hi!")
10|    defer func() {
11|        v := recover()
12|        fmt.Println("recovered:", v)
13|    }()
14|    panic("bye!")
15|    fmt.Println("unreachable")
16| }

```

The output:

```

hi!
recovered: bye!
exit normally.

```

Here is another example which shows a panicking goroutine exits without being recovered. So the whole program crashes.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "time"
6| )
7|
8| func main() {
9|     fmt.Println("hi!")
10|
11|    go func() {
12|        time.Sleep(time.Second)
13|        panic(123)
14|    }()
15|
16|    for {
17|        time.Sleep(time.Second)
18|    }
19| }

```

The output:

```
hi!
panic: 123

goroutine 5 [running]:
...

```

Go runtime will create panics for some circumstances, such as dividing an integer by zero. For example,

```
1| package main
2|
3| func main() {
4|     a, b := 1, 0
5|     _ = a/b
6| }
```

The output:

```
panic: runtime error: integer divide by zero

goroutine 1 [running]:
...

```

More runtime panic circumstances will be mentioned in later Go 101 articles.

Generally, panics are used for logic errors, such as careless human errors. Logic errors should never happen at run time. If they happen, there must be bugs in the code. On the other hand, non-logic errors are hard to absolutely avoid at run time. In other words, non-logic errors are errors happening in reality. Such errors should not cause panics and should be explicitly returned and handled properly.

We can learn [some panic/recover use cases](#) (§30) and [more about panic/recover mechanism](#) (§31) later.

Some Fatal Errors Are Not Panics and They Are Unrecoverable

For the standard Go compiler, some fatal errors, such as stack overflow and out of memory are not recoverable. Once they occur, program will crash.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Go Type System Overview

This article will introduce all kinds of types in Go and the concepts regarding Go type system. Without knowing these fundamental concepts, it is hard to have a thorough understanding of Go.

Concept: Basic Types

Built-in basic types in Go have been also introduced in [built-in basic types and basic value literals](#) (§6). For completeness of the current article, these built-in basic types are re-listed here.

- Built-in string type: `string`.
- Built-in boolean type: `bool`.
- Built-in numeric types:
 - `int8`, `uint8` (`byte`), `int16`, `uint16`, `int32` (`rune`), `uint32`, `int64`, `uint64`, `int`, `uint`, `uintptr`.
 - `float32`, `float64`.
 - `complex64`, `complex128`.

Note, `byte` is a built-in alias of `uint8`, and `rune` is a built-in alias of `int32`. We can also declare custom type aliases (see below).

Except [string types](#) (§19), Go 101 article series will not try to explain more on other basic types.

The 17 built-in basic types are predeclared types.

Concept: Composite Types

Go supports the following composite types:

- [pointer types](#) (§15) - C pointer alike.
- [struct types](#) (§16) - C struct alike.
- [function types](#) (§20) - functions are first-class types in Go.
- [container types](#) (§18):
 - array types - fixed-length container types.
 - slice type - dynamic-length and dynamic-capacity container types.
 - map types - maps are associative arrays (or dictionaries). The standard Go compiler implements maps as hashtables.

- [channel types](#) (§21) - channels are used to synchronize data among goroutines (the green threads in Go).
- [interface types](#) (§23) - interfaces play a key role in reflection and polymorphism.

Unnamed composite types may be denoted by their respective type literals. Following are some literal representation examples of all kinds of unnamed composite types (name and unnamed types will be explained below).

```

1| // Assume T is an arbitrary type and TKey is
2| // a type supporting comparison (== and !=).
3|
4| *T          // a pointer type
5| [5]T        // an array type
6| []T         // a slice type
7| map[Tkey]T // a map type
8|
9| // a struct type
10| struct {
11|     name string
12|     age  int
13| }
14|
15| // a function type
16| func(int) (bool, string)
17|
18| // an interface type
19| interface {
20|     Method0(string) int
21|     Method1() (int, bool)
22| }
23|
24| // some channel types
25| chan T
26| chan<- T
27| <-chan T

```

[Comparable and incomparable types](#) will be explained below.

Fact: Kinds of Types

Each of the above mentioned basic and composite types corresponds to one kind of types. Besides these kinds, the unsafe pointer types introduced in the [unsafe standard package ↗](#) also belong to one kind of types in Go. So, up to now (Go 1.21), Go has 26 kinds of types.

Syntax: Type Definitions

(**Type definition**, or *type definition declaration*, initially called **type declaration**, was the only type declaration way before Go 1.9. Since Go 1.9, type definition has become one of two forms of type declarations. The new form is called **type alias declaration**, which will be introduced in a section below.)

In Go, we can define new types by using the following form. In the syntax, `type` is a keyword.

```

1| // Define a solo new type.
2| type NewTypeName SourceType
3|
4| // Define multiple new types together.
5| type (
6|     NewTypeName1 SourceType1
7|     NewTypeName2 SourceType2
8| )

```

New type names must be identifiers. But please note that, type names declared at package level can't be [init](#) (§10). This is the same for the following introduced type alias names.

The second type declaration in the above example includes two type specifications. If a type declaration contains more than one type specification, the type specifications must be enclosed within a pair of `()`.

Note,

- a new defined type and its respective source type in type definitions are two distinct types.
- two types defined in two type definitions are always two distinct types.
- the new defined type and the source type will share the same underlying type (see below for the definition of underlying types), and their values can be converted to each other.
- types can be defined within function bodies.

Some type definition examples:

```

1| // The following new defined and source types are all
2| // basic types. The source types are all predeclared.
3| type (
4|     MyInt int
5|     Age    int
6|     Text   string
7| )
8|

```

```

9| // The following new defined and source types are all
10| // composite types. The source types are all unnamed.
11| type IntPtr *int
12| type Book struct{author, title string; pages int}
13| type Convert func(in0 int, in1 bool)(out0 int, out1 string)
14| type StringArray [5]string
15| type StringSlice []string
16|
17| func f() {
18|     // The names of the three defined types
19|     // can be only used within the function.
20|     type PersonAge map[string]int
21|     type MessageQueue chan string
22|     type Reader interface{Read([]byte) int}
23| }
```

Please note that, from Go 1.9 to Go 1.17, the Go specification ever thought predeclared built-in types are defined types. But since Go 1.18, the Go specificaiton clarifies that predeclared built-in types are not defined types.

Concept: Custom Generic Types and Instantiated Types

Since Go 1.18, Go has supported custom generic types (and functions). A generic type must be instantiated to be used as value types.

A generic type is a defined type and its instantiated types are named types (named types are explained in the next section).

The other two important concepts in custom generics are constraints and type parameters.

This book doesn't talk about custom generics in detail. Please read the [Go Generics 101](#) book on how to declare and use generic types and functions.

Concept: Named Types vs. Unnamed Types

Before Go 1.9, the terminology **named type** is defined accurately in Go specification. A named type was defined as a type that is represented by an identifier. Along with the custom type alias feature introduced in Go 1.9 (see the next section), the **named type** terminology was ever removed from Go specification and it was replaced by the **defined type** terminology. Since Go 1.18, along with

With the introduction of custom generics, the **named type** terminology has been added back to Go specification.

A named type may be

- a predeclared type;
- a defined (non-custom-generic) type;
- an instantiated type (of a generic type);
- a type parameter type (used in custom generics).

Other value types are called unnamed types. An unnamed type must be a composite type (not vice versa).

Syntax: Type Alias Declarations

Since Go 1.9, we can declare custom type aliases by using the following syntax. The syntax of alias declaration is much like type definition, but please note there is a `=` in each type alias specification.

```

1| type (
2|     Name = string
3|     Age  = int
4| )
5|
6| type table = map[string]int
7| type Table = map[Name]Age

```

Type alias names must be identifiers. Like type definitions, type aliases can also be declared within function bodies.

A type name (or literal) and its aliases all denote an identical type. By the above declarations, `Name` is an alias of `string`, so both denote the same type. The same applies to the other three pairs of type notations (either names or literals):

- `Age` and `int`
- `table` and `map[string]int`
- `Table` and `map[Name]Age`

In fact, the literals `map[string]int` and `map[Name]Age` also both denote the same type. So, similarly, aliases `table` and `Table` also denote the same type.

Note, although a type alias always has a name, it might denote an unnamed type. For example, the `table` and `Table` aliases both denote the same unnamed type `map[string]int`.

Concept: Underlying Types

In Go, each type has an underlying type. Rules:

- for built-in types, the respective underlying types are themselves.
- for the `Pointer` type defined in the `unsafe` standard code package, its underlying type is itself (at least we can think so. In fact, the underlying type of the `unsafe.Pointer` type is not well documented. We can also think the underlying type is `*T`, where `T` represents an arbitrary type). `unsafe.Pointer` is also viewed as a built-in type.
- the underlying type of an unnamed type, which must be a composite type, is itself.
- in a type declaration, the newly declared type and the source type have the same underlying type.

Examples:

```

1| // The underlying types of the following ones are both int.
2| type (
3|     MyInt int
4|     Age    MyInt
5| )
6|
7| // The following new types have different underlying types.
8| type (
9|     IntSlice    []int    // underlying type is []int
10|    MyIntSlice []MyInt // underlying type is []MyInt
11|    AgeSlice   []Age   // underlying type is []Age
12| )
13|
14| // The underlying types of []Age, Ages, and AgeSlice
15| // are all the unnamed type []Age.
16| type Ages AgeSlice

```

How can an underlying type be traced given a user declared type? The rule is, when a built-in basic type or an unnamed type is met, the tracing should be stopped. Take the type declarations shown above as examples, let's trace their underlying types.

```

MyInt → int
Age → MyInt → int
IntSlice → []int
MyIntSlice → []MyInt → []int
AgeSlice → []Age → []MyInt → []int
Ages → AgeSlice → []Age → []MyInt → []int

```

In Go,

- types whose underlying types are `bool` are called **boolean types**;
- types whose underlying types are any of the built-in integer types are called **integer types**;
- types whose underlying types are either `float32` or `float64` are called **floating-point types**;
- types whose underlying types are either `complex64` or `complex128` are called **complex types**;
- integer, floating-point and complex types are also called **numeric types**;
- types whose underlying types are `string` are called **string types**.

The concept of underlying type plays an important role in [value conversions, assignments and comparisons in Go](#) (§48).

Concept: Values

An instance of a type is called a 'value' of the type. Values of the same type share some common properties. A type may have many values. One of them is the zero value of the type.

Each type has a zero value, which can be viewed as the default value of the type. The predeclared `nil` identifier can be used to represent zero values of slices, maps, functions, channels, pointers (including type-unsafe pointers) and interfaces. For more information on `nil`, please read [nil in Go](#) (§47).

There are several kinds of value representation forms in code, including [literals](#) (§6), [named constants](#) (§7), [variables](#) (§7) and [expressions](#) (§11), though the former three can be viewed as special cases of the latter one.

A value can be [typed or untyped](#) (§7).

All kinds of basic value literals have been introduced in the article [basic types and basic value literals](#) (§6). There are two more kinds of literals in Go, composite literals and function literals.

Function literals, as the name implies, are used to represent function values. A [function declaration](#) (§9) is composed of a function literal and an identifier (the function name).

Composite literals are used to represent values of struct types and container types (arrays, slices and maps). Please read [structs in Go](#) (§16) and [containers in Go](#) (§18) for more details.

There are no literals to represent values of pointers, channels and interfaces.

Concept: Value Parts

At run time, many values are stored somewhere in memory. In Go, each of such values has a direct part. However, some of them have one or more indirect parts. Each value part occupies a continuous memory segment. The indirect underlying parts of a value are referenced by its direct part through ([safe](#) (§15) or [unsafe](#) (§25)) pointers.

The terminology [value part](#) (§17) is not defined in Go specification. It is just used in Go 101 to make some explanations simpler and help Go programmers understand Go types and values better.

Concept: Value Sizes

When a value is stored in memory, the number of bytes occupied by the direct part of the value is called the size of the value. As all values of the same type have the same value size, we often simply call this the size of the type.

We can use the `Sizeof` function in the `unsafe` standard package to get the size of any value.

Go specification doesn't specify value size requirements for non-numeric types. The requirements for value sizes of all kinds of basic numeric types are listed in the article [basic types and basic value literals](#) (§6).

Concept: Base Type of a Pointer Type

For a pointer type, assume its underlying type can be denoted as `*T` in literal, then `T` is called the base type of the pointer type.

More information on pointer types and values can be found in the article [pointers in Go](#) (§15).

Concept: Fields of a Struct Type

A struct type consists a collection of member variable declarations. Each of the member variable declarations is called a "field" of the struct type. For example, the following struct type `Book` has three fields, `author`, `title` and `pages`.

```
1| struct {
2|     author string
3|     title  string
```

```
4|     pages int  
5| }
```

More information on struct types and values can be found in the article [structs in Go](#) (§16).

Concept: Signature of Function Types

The signature of a function type is composed of the input parameter definition list and the output result definition list of the function.

The function name and body are not parts of a function signature. Parameter and result types are important for a function signature, but parameter and result names are not important.

Please read [functions in Go](#) (§20) for more details on function types and function values.

Concept: Methods and Method Set of a Type

In Go, some types can have [methods](#) (§22). Methods can also be called member functions. The method set of a type is composed of all the methods of the type.

Concept: Dynamic Type and Dynamic Value of an Interface Value

Interface values are values whose types are interface types.

Each interface value can box a non-interface value in it. The value boxed in an interface value is called the dynamic value of the interface value. The type of the dynamic value is called the dynamic type of the interface value. An interface value boxing nothing is a zero interface value. A zero interface value has neither a dynamic value nor a dynamic type.

An interface type can specify zero or several methods, which form the method set of the interface type.

If the method set of a type, which is either an interface type or a non-interface type, is the super set of the method set of an interface type, we say the type [implements](#) (§23) the interface type.

For more about interface types and values, please read [interfaces in Go](#) (§23).

Concept: Concrete Value and Concrete Type of a Value

For a (typed) non-interface value, its concrete value is itself and its concrete type is the type of the value.

A zero interface value has neither concrete type nor concrete value. For a non-zero interface value, its concrete value is its dynamic value and its concrete type is its dynamic type.

Concept: Container Types

Arrays, slices and maps can be viewed as formal container types.

Sometimes, string and channel types can also be viewed as container types informally.

Each value of a formal or informal container type has a length.

More information on formal container types and values can be found in the article [containers in Go](#) (§18).

Concept: Key Type of a Map Type

If the underlying type of a map type can be denoted as `map[Tkey]T`, then `Tkey` is called the key type of the map type. `Tkey` must be a comparable type (see below).

Concept: Element Type of a Container Type

The types of the elements stored in values of a container type must be identical. The identical type of the elements is called the element type of the container type.

- For an array type, if its underlying type is `[N]T`, then its element type is `T`.
- For a slice type, if its underlying type is `[]T`, then its element type is `T`.
- For a map type, if its underlying type is `map[Tkey]T`, then its element type is `T`.
- For a channel type, if its underlying type is `chan T`, `chan<- T` or `<-chan T`, then its element type is `T`.
- The element type of any string type is always `byte` (a.k.a. `uint8`).

Concept: Directions of Channel Types

Channel values can be viewed as synchronized first-in-first-out (FIFO) queues. Channel types and values have directions.

- A channel value which is both sendable and receivable is called a bidirectional channel. Its type is called a bidirectional channel type. The underlying types of bidirectional channel types are denoted by the `chan T` literal.
- A channel value which is only sendable is called a send-only channel. Its type is called a send-only channel type. Send-only channel types are denoted by the `chan<- T` literal.
- A channel value which is only receivable is called a receive-only channel. Its type is called a receive-only channel type. Receive-only channel types are denoted by the `<-chan T` literal.

More information on channel types and values can be found in the article [channels in Go](#) (§21).

Fact: Types Which Support or Don't Support Comparisons

Currently (Go 1.21), Go doesn't support comparisons (with the `==` and `!=` operators) for values of the following types:

- slice types
- map types
- function types
- any struct type with a field whose type is incomparable and any array type which element type is incomparable.

Above listed types are called incomparable types. All other types are called comparable types. Compilers forbid comparing two values of incomparable types.

Note, incomparable types are also called uncomparable types in many articles.

The key type of any map type must be a comparable type.

We can learn more about the detailed rules of comparisons from the article [value conversions, assignments and comparisons in Go](#) (§48).

Fact: Object-Oriented Programming in Go

Go is not a full-featured object-oriented programming language, but Go really supports some object-oriented programming elements. Please read the following listed articles for details:

- [methods in Go](#) (§22).
- [implementations in Go](#) (§23).
- [type embedding in Go](#) (§24).

Fact: Generics in Go

Before Go 1.18, the generic functionalities in Go were limited to built-in types and functions. Since Go 1.18, custom generics has already been supported. Please read the [generics in Go](#) (§26) article for built-in generics and the [Go Generics 101](#) book for custom generics.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Pointers in Go

Although Go absorbs many features from all kinds of other languages, Go is mainly viewed as a C family language. One evidence is Go also supports pointers. Go pointers and C pointers are much similar in many aspects, but there are also some differences between Go pointers and C pointers. This article will list all kinds of concepts and details related to pointers in Go.

Memory Addresses

A memory address means a specific memory location in programming.

Generally, a memory address is stored as an unsigned native (integer) word. The size of a native word is 4 (bytes) on 32-bit architectures and 8 (bytes) on 64-bit architectures. So the theoretical maximum memory space size is 2^{32} bytes, a.k.a. 4GB (1GB == 2^{30} bytes), on 32-bit architectures, and is 2^{64} bytes a.k.a 16EB (1EB == 1024PB, 1PB == 1024TB, 1TB == 1024GB), on 64-bit architectures.

Memory addresses are often represented with hex integer literals, such as `0x1234CDEF`.

Value Addresses

The address of a value means the start address of the memory segment occupied by the [direct part](#) (§17) of the value.

What Are Pointers?

Pointer is one kind of type in Go. A pointer value is used to store a memory address, which is generally the address of another value.

Unlike C language, for safety reason, there are some restrictions made for Go pointers. Please read the following sections for details.

Go Pointer Types and Values

In Go, an unnamed pointer type can be represented with `*T`, where `T` can be an arbitrary type. Type `T` is called the base type of pointer type `*T`.

We can declare named pointer types, but generally, it's not recommended to use named pointer types, for unnamed pointer types have better readability.

If the [underlying type](#) (§14) of a named pointer type is `*T`, then the base type of the named pointer type is `T`.

Two unnamed pointer types with the same base type are the same type.

Example:

```
1| *int // An unnamed pointer type whose base type is int.  
2| **int // An unnamed pointer type whose base type is *int.  
3|  
4| // Ptr is a named pointer type whose base type is int.  
5| type Ptr *int  
6| // PP is a named pointer type whose base type is Ptr.  
7| type PP *Ptr
```

Zero values of any pointer types are represented with the predeclared `nil`. No addresses are stored in nil pointer values.

A value of a pointer type whose base type is `T` can only store the addresses of values of type `T`.

About the Word "Reference"

In Go 101, the word "reference" indicates a relation. For example, if a pointer value stores the address of another value, then we can say the pointer value (directly) references the other value, and the other value has at least one reference. The uses of the word "reference" in Go 101 are consistent with Go specification.

When a pointer value references another value, we also often say the pointer value points to the other value.

How to Get a Pointer Value and What Are Addressable Values?

There are two ways to get a non-nil pointer value.

1. The built-in `new` function can be used to allocate memory for a value of any type. `new(T)` will allocate memory for a `T` value (an anonymous variable) and return the address of the `T` value. The allocated value is a zero value of type `T`. The returned address is viewed as a pointer value of type `*T`.
2. We can also take the addresses of values which are addressable in Go. For an addressable value `t` of type `T`, we can use the expression `&t` to take the address of `t`, where `&` is the operator to take value addresses. The type of `&t` is viewed as `*T`.

Generally speaking, an addressable value means a value which is hosted at somewhere in memory. Currently, we just need to know that all variables are addressable, whereas constants, function calls and explicit conversion results are all unaddressable. When a variable is declared, Go runtime will allocate a piece of memory for the variable. The starting address of that piece of memory is the address of the variable.

We will learn other addressable and unaddressable values from other articles later. If you have already been familiar with Go, you can read [this summary](#) (§46) to get the lists of addressable and unaddressable values in Go.

The next section will show an example on how to get pointer values.

Pointer Dereference

Given a pointer value `p` of a pointer type whose base type is `T`, how can you get the value at the address stored in the pointer (a.k.a., the value being referenced by the pointer)? Just use the expression `*p`, where `*` is called dereference operator. `*p` is called the dereference of pointer `p`. Pointer dereference is the inverse process of address taking. The result of `*p` is a value of type `T` (the base type of the type of `p`).

Dereferencing a nil pointer causes a runtime panic.

The following program shows some address taking and pointer dereference examples:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     p0 := new(int)    // p0 points to a zero int value.
7|     fmt.Println(p0)  // (a hex address string)
8|     fmt.Println(*p0) // 0
9|

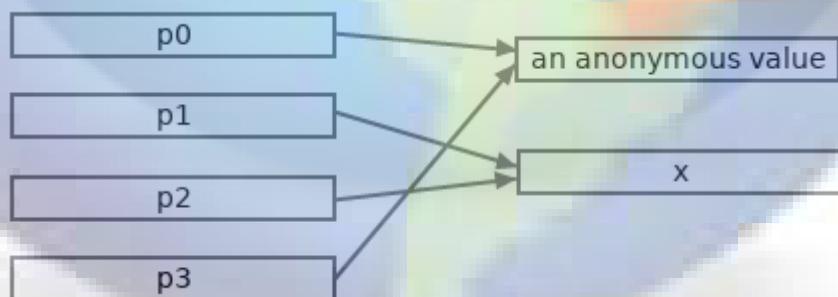
```

```

10| // x is a copy of the value at
11| // the address stored in p0.
12| x := *p0
13| // Both take the address of x.
14| // x, *p1 and *p2 represent the same value.
15| p1, p2 := &x, &x
16| fmt.Println(p1 == p2) // true
17| fmt.Println(p0 == p1) // false
18| p3 := &*p0 // <=> p3 := &(*p0) <=> p3 := p0
19| // Now, p3 and p0 store the same address.
20| fmt.Println(p0 == p3) // true
21| *p0, *p1 = 123, 789
22| fmt.Println(*p2, x, *p3) // 789 789 123
23|
24| fmt.Printf("%T, %T \n", *p0, x) // int, int
25| fmt.Printf("%T, %T \n", p0, p1) // *int, *int
26| }

```

The following picture depicts the relations of the values used in the above program.



Why Do We Need Pointers?

Let's view an example firstly.

```

1| package main
2|
3| import "fmt"
4|
5| func double(x int) {
6|     x += x
7| }
8|
9| func main() {
10|     var a = 3
11|     double(a)

```

```

12|     fmt.Println(a) // 3
13| }
```

The `double` function in the above example is expected to modify the input argument by doubling it. However, it fails. Why? Because all value assignments, including function argument passing, are value copying in Go. What the `double` function modified is a copy (x) of variable `a` but not variable `a`.

One solution to fix the above `double` function is let it return the modification result. This solution doesn't always work for all scenarios. The following example shows another solution, by using a pointer parameter.

```

1| package main
2|
3| import "fmt"
4|
5| func double(x *int) {
6|     *x += *x
7|     x = nil // the line is just for explanation purpose
8| }
9|
10| func main() {
11|     var a = 3
12|     double(&a)
13|     fmt.Println(a) // 6
14|     p := &a
15|     double(p)
16|     fmt.Println(a, p == nil) // 12 false
17| }
```

We can find that, by changing the parameter to a pointer type, the passed pointer argument `&a` and its copy `x` used in the function body both reference the same value, so the modification on `*x` is equivalent to a modification on `*p`, a.k.a., variable `a`. In other words, the modification in the `double` function body can be reflected out of the function now.

Surely, the modification of the copy of the passed pointer argument itself still can't be reflected on the passed pointer argument. After the second `double` function call, the local pointer `p` doesn't get modified to `nil`.

In short, pointers provide indirect ways to access some values. Many languages do not have the concept of pointers. However, pointers are just hidden under other concepts in those languages.

Return Pointers of Local Variables Is Safe in Go

Unlike C language, Go is a language supporting garbage collection, so return the address of a local variable is absolutely safe in Go.

```

1| func newInt() *int {
2|     a := 3
3|     return &a
4|

```

Restrictions on Pointers in Go

For safety reasons, Go makes some restrictions to pointers (comparing to pointers in C language). By applying these restrictions, Go keeps the benefits of pointers, and avoids the dangerousness of pointers at the same time.

Go pointer values don't support arithmetic operations

In Go, pointers can't do arithmetic operations. For a pointer `p`, `p++` and `p-2` are both illegal.

If `p` is a pointer to a numeric value, compilers will view `*p++` is a legal statement and treat it as `(*p)++`. In other words, the precedence of the pointer dereference operator `*` is higher than the increment operator `++` and decrement operator `--`.

Example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     a := int64(5)
7|     p := &a
8|
9|     // The following two lines don't compile.
10|    /*
11|     p++
12|     p = (&a) + 8
13|     */
14|
15|     *p++

```

```

16|     fmt.Println(*p, a)    // 6 6
17|     fmt.Println(p == &a) // true
18|
19|     *&a++
20|     *&*&a++
21|     **&p++
22|     *&*p++
23|     fmt.Println(*p, a) // 10 10
24| }
```

A pointer value can't be converted to an arbitrary pointer type

In Go, a pointer value of pointer type `T1` can be directly and explicitly converted to another pointer type `T2` only if either of the following two conditions is satisfied.

1. The underlying types of type `T1` and `T2` are identical (ignoring struct tags), in particular if either `T1` and `T2` is a [unnamed](#) (§14) type and their underlying types are identical (considering struct tags), then the conversion can be implicit. Struct types and values will be explained in [the next article](#) (§16).
2. Type `T1` and `T2` are both unnamed pointer types and the underlying types of their base types are identical (ignoring struct tags).

For example, for the below shown pointer types:

```

1| type MyInt int64
2| type Ta      *int64
3| type Tb      *MyInt
```

the following facts exist:

1. values of type `*int64` can be implicitly converted to type `Ta`, and vice versa, for their underlying types are both `*int64`.
2. values of type `*MyInt` can be implicitly converted to type `Tb`, and vice versa, for their underlying types are both `*MyInt`.
3. values of type `*MyInt` can be explicitly converted to type `*int64`, and vice versa, for they are both unnamed and the underlying types of their base types are both `int64`.
4. values of type `Ta` can't be directly converted to type `Tb`, even if explicitly. However, by the just listed first three facts, a value `pa` of type `Ta` can be indirectly converted to type `Tb` by nesting three explicit conversions, `Tb((*MyInt)((*int64)(pa)))`.

None values of these pointer types can be converted to type `*uint64`, in any safe ways.

A pointer value can't be compared with values of an arbitrary pointer type

In Go, pointers can be compared with `==` and `!=` operators. Two Go pointer values can only be compared if either of the following three conditions are satisfied.

1. The types of the two Go pointers are identical.
2. One pointer value can be implicitly converted to the pointer type of the other. In other words, the underlying types of the two types must be identical and either of the two types of the two Go pointers is an unnamed type.
3. One and only one of the two pointers is represented with the bare (untyped) `nil` identifier.

Example:

```

1| package main
2|
3| func main() {
4|     type MyInt int64
5|     type Ta    *int64
6|     type Tb    *MyInt
7|
8|     // 4 nil pointers of different types.
9|     var pa0 Ta
10|    var pa1 *int64
11|    var pb0 Tb
12|    var pb1 *MyInt
13|
14|    // The following 6 lines all compile okay.
15|    // The comparison results are all true.
16|    _ = pa0 == pa1
17|    _ = pb0 == pb1
18|    _ = pa0 == nil
19|    _ = pa1 == nil
20|    _ = pb0 == nil
21|    _ = pb1 == nil
22|
23|    // None of the following 3 lines compile ok.
24|    /*
25|     _ = pa0 == pb0
26|     _ = pa1 == pb1
27|     _ = pa0 == Tb(nil)
28|    */
29| }
```

A pointer value can't be assigned to pointer values of other pointer types

The conditions to assign a pointer value to another pointer value are the same as the conditions to compare a pointer value to another pointer value, which are listed above.

It's Possible to Break the Go Pointer Restrictions

As the start of this article has mentioned, the mechanisms (specifically, the `unsafe.Pointer` type) provided by [the unsafe standard package](#) (§25) can be used to break the restrictions made for pointers in Go. The `unsafe.Pointer` type is like the `void*` in C. In general the unsafe ways are not recommended to use.

(The **Go 101** book is still being improved frequently from time to time. Please visit [go101.org](#) or follow [@go100and1](#) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit [tapirgames.com](#) to get more information about these games. Hope you enjoy them.)

Structs in Go

Same as C, Go also supports struct types. This article will introduce the basic knowledge of struct types and values in Go.

Struct Types and Struct Type Literals

Each unnamed struct type literal starts with a `struct` keyword which is followed by a sequence of field definitions enclosed in a `{}`. Generally, each field definition is composed of a name and a type. The number of fields of a struct type can be zero.

The following is an unnamed struct type literal:

```
1| struct {
2|     title  string
3|     author string
4|     pages  int
5| }
```

The above struct type has three fields. The types of the two fields `title` and `author` are both `string`. The type of the `pages` field is `int`.

Some articles also call fields as member variables.

Consecutive fields with the same type can be declared together.

```
1| struct {
2|     title, author string
3|     pages         int
4| }
```

The size of a struct type is the sum of the sizes of all its field types plus the number of some padding bytes. The padding bytes are used to align the memory addresses of some fields. We can learn padding and memory address alignments in [a later article](#) (§44).

The size of a zero-field struct type is zero.

A tag may be bound to a struct field when the field is declared. Field tags are optional, the default value of each field tag is a blank string. The syntax allows either string literal forms for field tags. However, in practice, struct filed tags should present as key-value pairs, and each tag should present

as raw string literals (`...`), whereas each value in a tag should present as interpreted string literals ("..."). For example:

```
1| struct {
2|     Title  string `json:"title" myfmt:"s1"`
3|     Author string `json:"author,omitempty" myfmt:"s2"`
4|     Pages   int    `json:"pages,omitempty" myfmt:"n1"`
5|     X, Y    bool   `myfmt:"b1"`
6| }
```

Note, the tags of the `X` and `Y` fields in the above example are identical (though using field tags as this way is a bad practice).

We can use the [reflection](#) (§27) way to inspect field tag information.

The purpose of each field tag is application dependent. In the above example, the field tags can help the functions in the `encoding/json` standard package to determine the field names in JSON texts, in the process of encoding struct values into JSON texts or decoding JSON texts into struct values. The functions in the `encoding/json` standard package will only encode and decode the exported struct fields, which is why the first letters of the field names in the above example are all upper cased.

It is not a good idea to use field tags as comments.

Unlike C language, Go structs don't support unions.

All above shown struct types are unnamed. In practice, named struct types are more popular.

Only exported fields of struct types shown up in a package can be used in other packages by importing the package. We can view non-exported struct fields as private/protected member variables.

The field tags and the order of the field declarations in a struct type matter for the identity of the struct type. Two unnamed struct types are identical only if they have the same sequence of field declarations. Two field declarations are identical only if their respective names, their respective types and their respective tags are all identical. Please note, **two non-exported struct field names from different packages are always viewed as two different names.**

A struct type can't have a field of the struct type itself, neither directly nor recursively.

Struct Value Literals and Struct Value Manipulations

In Go, the form `T{...}`, where `T` must be a type literal or a type name, is called a **composite literal** and is used as the value literals of some kinds of types, including struct types and the container types introduced later.

Note, a type literal `T{...}` is a typed value, its type is `T`.

Given a struct type `S` whose [underlying type](#) (§14) is `struct{x int; y bool}`, the zero value of `S` can be represented by the following two variants of struct composite literal forms:

1. `S{0, false}`. In this variant, no field names are present but all field values must be present by the field declaration orders.
2. `S{x: 0, y: false}`, `S{y: false, x: 0}`, `S{x: 0}`, `S{y: false}` and `S{}`. In this variant, each field item is optional and the order of the field items is not important. The values of the absent fields will be set as the zero values of their respective types. But if a field item is present, it must be presented with the `FieldName: FieldValue` form. The order of the field items in this form doesn't matter. The form `S{}` is the most used zero value representation of type `S`.

If `S` is a struct type imported from another package, it is recommended to use the second form, to maintain compatibility. Consider the case where the maintainer of the package adds a new field for type `S`, this will make the use of first form invalid.

Surely, we can also use the struct composite literals to represent non-zero struct value.

For a value `v` of type `S`, we can use `v.x` and `v.y`, which are called selectors (or selector expressions), to represent the field values of `v`. `v` is called the receiver of the selectors. Later, we call the dot `.` in a selector as the property selection operator.

An example:

```

1| package main
2|
3| import (
4|     "fmt"
5| )
6|
7| type Book struct {
8|     title, author string
9|     pages         int
10| }
11|
12| func main() {
13|     book := Book{"Go 101", "Tapir", 256}

```

```

14|     fmt.Println(book) // {Go 101 Tapir 256}
15|
16|     // Create a book value with another form.
17|     // All of the three fields are specified.
18|     book = Book{author: "Tapir", pages: 256, title: "Go 101"}
19|
20|     // None of the fields are specified. The title and
21|     // author fields are both "", pages field is 0.
22|     book = Book{}
23|
24|     // Only specify the author field. The title field
25|     // is "" and the pages field is 0.
26|     book = Book{author: "Tapir"}
27|
28|     // Initialize a struct value by using selectors.
29|     var book2 Book // <=> book2 := Book{}
30|     book2.author = "Tapir Liu"
31|     book2.pages = 300
32|     fmt.Println(book2.pages) // 300
33| }
```

The last , in a composite literal is optional if the last item in the literal and the closing } are at the same line. Otherwise, the last , is required. For more details, please read [line break rules in Go](#) (§28).

```

1| var _ = Book {
2|     author: "Tapir",
3|     pages: 256,
4|     title: "Go 101", // here, the "," must be present
5| }
6|
7| // The last "," in the following line is optional.
8| var _ = Book{author: "Tapir", pages: 256, title: "Go 101",}
```

About Struct Value Assignments

When a struct value is assigned to another struct value, the effect is the same as assigning each field one by one.

```

1| func f() {
2|     book1 := Book{pages: 300}
3|     book2 := Book{"Go 101", "Tapir", 256}
4| }
```

```

5|     book2 = book1
6|     // The above line is equivalent to the
7|     // following lines.
8|     book2.title = book1.title
9|     book2.author = book1.author
10|    book2.pages = book1.pages
11| }
```

Two struct values can be assigned to each other only if their types are identical or the types of the two struct values have an identical underlying type (considering field tags) and at least one of the two types is an [unnamed type](#) (§14).

Struct Field Addressability

The fields of an addressable struct are also addressable. The fields of an unaddressable struct are also unaddressable. The fields of unaddressable structs can't be modified. All composite literals, including struct composite literals are unaddressable.

Example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     type Book struct {
7|         Pages int
8|     }
9|     var book = Book{} // book is addressable
10|    p := &book.Pages
11|    *p = 123
12|    fmt.Println(book) // {123}
13|
14|    // The following two lines fail to compile, for
15|    // Book{} is unaddressable, so is Book{}.Pages.
16|    /*
17|     Book{}.Pages = 123
18|     p = &(Book{}.Pages) // <=> p = &Book{}.Pages
19|     */
20| }
```

Note that the precedence of the property selection operator `.` in a selector is higher than the address-taking operator `&`.

Composite Literals Are Unaddressable But Can Take Addresses

Generally, only addressable values can take addresses. But there is a syntactic sugar in Go, which allows us to take addresses on composite literals. A syntactic sugar is an exception in syntax to make programming convenient.

For example,

```

1| package main
2|
3| func main() {
4|     type Book struct {
5|         Pages int
6|     }
7|     // Book{100} is unaddressable but can
8|     // be taken address.
9|     p := &Book{100} // <=> tmp := Book{100}; p := &tmp
10|    p.Pages = 200
11| }
```

In Field Selectors, Dereferences of Receivers Can Be Implicit

In the following example, for simplicity, `(*bookN).pages` could be written as `bookN.pages`. In other words, `bookN` is dereferenced in the simplified selectors.

```

1| package main
2|
3| func main() {
4|     type Book struct {
5|         pages int
6|     }
7|     book1 := &Book{100} // book1 is a struct pointer
8|     book2 := new(Book) // book2 is another struct pointer
9|     // Use struct pointers as structs.
10|    book2.pages = book1.pages
11|    // This last line is equivalent to the above line.
12|    // In other words, if the receiver is a pointer,
13|    // it will be implicitly dereferenced.
```

```
14|     (*book2).pages = (*book1).pages
15| }
```

About Struct Value Comparisons

A struct type is comparable only if none of the types of its fields (including the fields with names as the blank identifier `_`) are [incomparable](#) (§14).

Two struct values are comparable only if they can be assigned to each other and their types are both comparable. In other words, two struct values can be compared with each other only if the (comparable) types of the two struct values are identical or their underlying types are identical (considering field tags) and at least one of the two types is unnamed.

When comparing two struct values of the same type, each pair of their corresponding fields will be compared (in the order shown in source code). The two struct values are equal only if all of their corresponding fields are equal. The comparison stops in advance when a pair of fields is found unequal or [a panic occurs](#) (§23). In comparisons, fields with names as the blank identifier `_` will be ignored.

About Struct Value Conversions

Values of two struct types `S1` and `S2` can be converted to each other's types, if `S1` and `S2` share the identical underlying type (by ignoring field tags). In particular if either `S1` or `S2` is an [unnamed type](#) (§14) and their underlying types are identical (considering field tags), then the conversions between the values of them can be implicit.

Given struct types `S0`, `S1`, `S2`, `S3` and `S4` in the following code snippet,

- values of type `S0` can't be converted to the other four types, and vice versa, because the corresponding field names are different.
- two values of two different types among `S1`, `S2`, `S3` and `S4` can be converted to each other's type.

In particular,

- values of the type denoted by `S2` can be implicitly converted to type `S3`, and vice versa.
- values of the type denoted by `S2` can be implicitly converted to type `S4`, and vice versa.

But,

- values of the type denoted by S2 must be explicitly converted to type S1, and vice versa.
- values of type S3 must be explicitly converted to type S4, and vice versa.

```

1| package main
2|
3| type S0 struct {
4|     y int "foo"
5|     x bool
6| }
7|
8| // S1 is an alias of an unnamed type.
9| type S1 = struct {
10|    x int "foo"
11|    y bool
12| }
13|
14| // S2 is also an alias of an unnamed type.
15| type S2 = struct {
16|     x int "bar"
17|     y bool
18| }
19|
20| // If field tags are ignored, the underlying
21| // types of S3(S4) and S1 are same. If field
22| // tags are considered, the underlying types
23| // of S3(S4) and S1 are different.
24| type S3 S2 // S3 is a defined (so named) type
25| type S4 S3 // S4 is a defined (so named) type
26|
27| var v0, v1, v2, v3, v4 = S0{}, S1{}, S2{}, S3{}, S4{}
28| func f() {
29|     v1 = S1(v2); v2 = S2(v1)
30|     v1 = S1(v3); v3 = S3(v1)
31|     v1 = S1(v4); v4 = S4(v1)
32|     v2 = v3; v3 = v2 // the conversions can be implicit
33|     v2 = v4; v4 = v2 // the conversions can be implicit
34|     v3 = S3(v4); v4 = S4(v3)
35| }
```

In fact, two struct values can be assigned (or compared) to each other only if one of them can be implicitly converted to the type of the other.

Anonymous Struct Types Can Be Used in Field Declarations

Anonymous struct types are allowed to be used as the types of the fields of another struct type. Anonymous struct type literals are also allowed to be used in composite literals.

An example:

```

1| var aBook = struct {
2|     // The type of the author field is
3|     // an anonymous struct type.
4|     author struct {
5|         firstName, lastName string
6|         gender               bool
7|     }
8|     title string
9|     pages int
10| }{
11|     author: struct { // an anonymous struct type
12|         firstName, lastName string
13|         gender               bool
14|     }{
15|         firstName: "Mark",
16|         lastName: "Twain",
17|     },
18|     title: "The Million Pound Note",
19|     pages: 96,
20| }
```

Generally, for better readability, it is not recommended to use anonymous struct type literals in composite literals.

More About Struct Types

There are some advanced topics which are related to struct types. They will be explained in [type embedding](#) (§24) and [memory layouts](#) (§44) later.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit

tapirgames.com  to get more information about these games. Hope you enjoy them.)

Value Parts

The articles following the current one will introduce more kinds of Go types. To easily and deeply understand those articles, it is best to read the following contents in the current article firstly before reading those articles.

Two Categories of Go Types

Go can be viewed as a C-family language, which can be confirmed from the two previous articles [pointers in Go](#) (§15) and [structs in Go](#) (§16). The memory structures of struct types and pointer types in Go and C are much alike.

On the other hand, Go can be also viewed as a C language framework. This is mainly reflected from the fact that Go supports several kinds of types whose value memory structures are not totally transparent, whereas the main characteristic of C types is the memory structures of C values are transparent. Each C value in memory occupies [one memory block](#) (§43) (one continuous memory segment). However, a value of some kinds of Go types may often be hosted on more than one memory block.

Later, we call the parts (being distributed on different memory blocks) of a value as value parts. A value hosting on more than one memory blocks is composed of one direct value part and several underlying indirect parts which are [referenced](#) (§15) by that direct value part.

The above paragraphs describe two categories of Go types:

Types whose values each is only hosted on one single memory block	Types whose values each may be hosted on multiple memory blocks
Solo Direct Value Part	Direct Part → Underlying Part
boolean types numeric types pointer types unsafe pointer types struct types array types	slice types map types channel types function types interface types string types

The following Go 101 articles will make detailed explanations for many kinds of types listed in the above table. The current article is just to make a preparation to understand those explanations more easily.

Note,

- whether or not interface and string values may contain underlying parts is compiler dependent. For the standard Go compiler implementation, interface and string values may contain underlying parts.
- whether or not functions values may contain underlying parts is hardly, even impossible, to prove. In Go 101, we will view functions values may contain underlying parts.

The kinds of types in the second category bring much convenience to Go programming by encapsulating many implementation details. Different Go compilers may adopt different internal implementations for these types, but the external behaviors of values of these types must satisfy the requirements specified in Go specification.

The types in the second category are not very fundamental types for a language, we can implement them from scratch by using the types from the first category. However, by encapsulating some common or unique functionalities and supporting these types as the first-class citizens in Go, the experiences of Go programming become enjoyable and productive.

On the other hand, these encapsulations adopted in implementing the types in the second category hide many internal definitions of these types. This prevents Go programmers from viewing the whole pictures of these types, and sometimes makes some obstacles to understand Go better.

To help gophers better understand the types in the second category and their values, the following contents of this article will introduce the internal structure definitions of these kinds of types. The detailed implementations of these types will not be explained here. The explanations in this article are based on, but not exactly the same as, the implementations used by the standard Go compiler.

Two Kinds of Pointer Types in Go

Before showing the internal structure definitions of the kinds of types in the second category, let's clarify more on pointers and references.

We have learned [Go pointers](#) (§15) in the article before the last. The pointer types in that article are type-safe pointer types. In fact, Go also supports [type-unsafe pointer types](#) (§25). The `unsafe.Pointer` type provided in the `unsafe` standard package is like `void*` in C language.

In most other articles in Go 101, if not specially specified, when a pointer type is mentioned, it means a type-safe pointer type. However, in the following parts of the current article, when a pointer is mentioned, it might be either a type-safe pointer or a type-unsafe pointer.

A pointer value stores a memory address of another value, unless the pointer value is a nil pointer. We can say the pointer value [references](#) (§15) the other value, or the other value is referenced by the

pointer value. Values can also be referenced indirectly.

- If a struct value `a` has a pointer field `b` which references a value `c`, then we can say the struct value `a` also references value `c`.
- If a value `x` references (either directly or indirectly) a value `y`, and the value `y` references (either directly or indirectly) a value `z`, then we can also say the value `x` (indirectly) references value `z`.

Below, we call a struct type with fields of pointer types as a **pointer wrapper type**, and call a type whose values may contains (either directly or indirectly) pointers a **pointer holder type**. Pointer types and pointer wrapper types are all pointer holder types. Array types with pointer holder element types are also pointer holder types. (Array types will be explained in the next article.)

(Possible) Internal Definitions of the Types in the Second Category

To better understand the runtime behaviors of values of the second category, it is not a bad idea that we could think these types are internally defined as types in the first category, which are shown below. If you haven't used all kinds of Go types much, currently you don't need to try to comprehend these definitions clearly. Instead, it is okay to just get a rough impression on these definitions and reread this article when you get more experience in Go programming later. Knowing the definitions roughly is good enough to help Go programmers understand the types explained in the following articles.

Internal definitions of map, channel and function types

The internal definitions of map, channel and function types are similar:

```

1| // map types
2| type _map *hashtableImpl
3|
4| // channel types
5| type _channel *channelImpl
6|
7| // function types
8| type _function *functionImpl

```

So, internally, types of the three kinds are just pointer types. In other words, the direct parts of values of these types are pointers internally. For each non-zero value of these types, its direct part (a pointer) references its indirect underlying implementation part.

BTW, the standard Go compiler uses hashtables to implement maps.

Internal definition of slice types

The internal definition of slice types is like:

```
1| type _slice struct {
2|     // referencing underlying elements
3|     elements unsafe.Pointer
4|     // number of elements and capacity
5|     len, cap int
6| }
```

So, internally, slice types are pointer wrapper struct types. Each non-zero slice value has an indirect underlying part which stores the element values of the slice value. The `elements` field of the direct part references the indirect underlying part of the slice value.

Internal definition of string types

Below is the internal definition for string types:

```
1| type _string struct {
2|     elements *byte // referencing underlying bytes
3|     len      int   // number of bytes
4| }
```

So string types are also pointer wrapper struct types internally. Each string value has an indirect underlying part storing the bytes of the string value, the indirect part is referenced by the `elements` field of that string value.

Internal definition of interface types

Below is the internal definition for general interface types:

```
1| type _interface struct {
2|     dynamicType  *_type          // the dynamic type
3|     dynamicValue unsafe.Pointer // the dynamic value
4| }
```

Internally, interface types are also pointer wrapper struct types. The internal definition of an interface type has two pointer fields. Each non-zero interface value has two indirect underlying

parts which store the dynamic type and dynamic value of that interface value. The two indirect parts are referenced by the `dynamicType` and `dynamicValue` fields of that interface value.

In fact, for the standard Go compiler, the above definition is only used for blank interface types. Blank interface types are the interface types which don't specify any methods. We can learn more about interfaces in the article [interfaces in Go](#) (§23) later. For non-blank interface types, the definition like the following one is used.

```

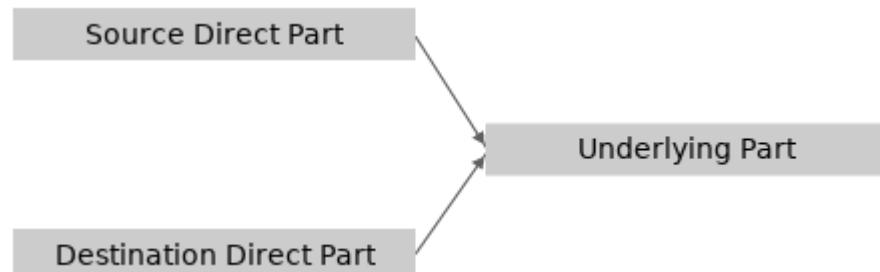
1| type _interface struct {
2|     dynamicTypeInfo *struct {
3|         dynamicType *_type      // the dynamic type
4|         methods      []*function // method table
5|     }
6|     dynamicValue unsafe.Pointer // the dynamic value
7| }
```

The `methods` field of the `dynamicTypeInfo` field of an interface value stores the implemented methods of the dynamic type of the interface value for the (interface) type of the interface value.

Underlying Value Parts Are Not Copied in Value Assignments

Now we have learned that the internal definitions of the types in the second category are pointer holder (pointer or pointer wrapper) types. Knowing this is very helpful to understand value copy behaviors in Go.

In Go, each value assignment (including parameter passing, etc) is a shallow value copy if the involved destination and source values have the same type (if their types are different, we can think that the source value will be implicitly converted to the destination type before doing that assignment). In other words, only the direct part of the source value is copied to the destination value in a value assignment. If the source value has underlying value part(s), then the direct parts of the destination and source values will reference the same underlying value part(s), in other words, the destination and source values will share the same underlying value part(s).



In fact, the above descriptions are not 100% correct in theory, for strings and interfaces. The [official Go FAQ ↗](#) says the underlying dynamic value part of an interface value should be copied as well when the interface value is copied. However, as the dynamic value of an interface value is read only, the standard Go compiler/runtime doesn't copy the underlying dynamic value parts in copying interface values. This can be viewed as a compiler optimization. The same situation is for string values and the same optimization (made by the standard Go compiler/runtime) is made for copying string values. So, for the standard Go compiler/runtime, the descriptions in the last section are 100% correct, for values of any type.

Since an indirect underlying part may not belong to any value exclusively, it doesn't contribute to the size returned by the `unsafe.Sizeof` function.

About the "Reference Type" and "Reference Value" Terminologies

The word ***reference*** in Go world is a big mess. It brings many confusions to Go community. Some articles, including some [official ones ↗](#), use ***reference*** as qualifiers of types and values, or treat ***reference*** as the opposite of ***value***. This is strongly discouraged in Go 101. I really don't want to dispute on this point. Here I just list some absolutely misuses of ***reference***:

- only slice, map, channel and function types are reference types in Go. (If we do need the ***reference type*** terminology in Go, then we shouldn't exclude any pointer holder types from reference types).
- references are opposites of values. (If we do need the ***reference value*** terminology in Go, then please view reference values as special values, instead of opposites of values.)
- some parameters are passed by reference. (Sorry, all parameters are passed by copy, **of direct parts**, in Go.)

I don't mean the ***reference type*** or ***reference value*** terminologies are totally useless for Go, I just think they are not very essential, and they bring many confusions in using Go. If we do need these terminologies, I prefer to define them as pointer holders. And, my personal opinion is it is best to limit the ***reference*** word to only representing relations between values by using it as a verb or a noun, and never use it as an adjective. This will avoid many confusions in learning, teaching and using Go.

(The **Go 101** book is still being improved frequently from time to time. Please visit [go101.org ↗](#) or follow [@go100and1 ↗](#) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit

tapirgames.com  to get more information about these games. Hope you enjoy them.)

Arrays, Slices and Maps in Go

Strictly speaking, there are three kinds of first-class citizen container types in Go, array, slice and map. Sometimes, strings and channels can also be viewed as container types, but this article will not touch the two kinds of types. All container types talked about in the current article are arrays, slices and maps.

There are many container related details in Go. This article will list them one by one.

Simple Overview of Container Types and Values

Each value of the three kinds of types is used to store a collection of element values. The types of all the elements stored in a container value are identical. The identical type is called the element type of (the container type of) the container value.

Each element in a container has an associated key. An element value can be accessed or modified through its associated key. The key types of map types must be [comparable types](#) (§14). The key types of array and slice types are all the built-in type `int`. The keys of the elements of an array or slice are non-negative integers which mark the positions of these elements in the array or slice. The non-negative integer keys are often called indexes.

Each container value has a length property, which indicates how many elements are stored in that container. The valid range of the integer keys of an array or slice value is from zero (inclusive) to the length (exclusive) of the array or slice. For each value of a map type, the key values of that map value can be an arbitrary value of the key type of the map type.

There are also many differences between the three kinds of container types. Most of the differences originate from the differences between the value memory layouts of the three kinds of types. From the last article, [value parts](#) (§17), we learned that an array value consists of only one direct part, however a slice or map value may have an underlying part, which is referenced by the direct part of the slice or map value.

Elements of an array or a slice are both stored contiguously in a continuous memory segment. For an array, the continuous memory segment hosts the direct part of the array. For a slice, the continuous memory segment hosts the underlying indirect part of the slice. The map implementation of the standard Go compiler/runtime adopts the hashtable algorithm. So all elements of a map are also stored in an underlying continuous memory segment, but they may be not contiguous. There may be many holes (gaps) within the continuous memory segment. Another common map

implementation algorithm is the binary tree algorithm. Whatever algorithm is used, the keys associated with the elements of a map are also stored in (the underlying parts of) the map.

We can access an element through its key. The time complexities of element accesses on all container values are all $O(1)$, though, generally map element accesses are several times slower than array and slice element accesses. But maps have two advantages over arrays and slices:

- the key types of maps can be any comparable types.
- maps consume much less memory than arrays and slices if most elements are zero values.

From the last article, we have learned that the underlying parts of a value will not get copied when the value is copied. In other words, if a value has underlying parts, a copy of the value will share the underlying parts with the value. This is the root reason of many behavior differences between array and slice/map values. These behavior differences will be introduced below.

Literal Representations of Unnamed Container Types

The literal representations of the three kinds of unnamed container types:

- array types: $[N]T$
- slice types: $[]T$
- map types: $\text{map}[K]T$

where

- T is an arbitrary type. It specifies the element type of a container type. Only values of the specified element type can be stored as element values of values of the container type.
- N must be a non-negative integer constant. It specifies the number of elements stored in any value of an array type, and it can be called the length of the array type. This means the length of an array type is the inherent part of the array type. For example, $[5]\text{int}$ and $[8]\text{int}$ are two distinct array types.
- K is an arbitrary comparable type. It specifies the key type of a map type. Most types in Go are comparable, incomparable types are [listed here](#) (§14).

Here are some container type literal representation examples:

```

1| const Size = 32
2|
3| type Person struct {
4|     name string
5|     age   int
6| }
```

```

7|
8| /* Array types */
9|
10| [5]string
11| [Size]int
12| // Element type is a slice type: []byte
13| [16][]byte
14| // Element type is a struct type: Person
15| [100]Person
16|
17| /* Slice types */
18|
19| []bool
20| []int64
21| // Element type is a map type: map[int]bool
22| []map[int]bool
23| // Element type is a pointer type: *int
24| []*int
25|
26| /* Map types */
27|
28| map[string]int
29| map[int]bool
30| // Element type is an array type: [6]string
31| map[int16][6]string
32| // Element type is a slice type: []string
33| map[bool][]string
34| // Element type is a pointer type: *int8,
35| // and key type is a struct type.
36| map[struct{x int}]*int8

```

The [sizes](#) (§14) of all slice types are the same. The sizes of all map types are also the same. The size of an array type depends on its length and the size of its element type. The size of a zero-length array type or an array type with a zero-size element type is zero.

Container Value Literals

Like struct values, container values can also be represented with composite literals, $T\{\dots\}$, where T denotes container type (except the zero values of slice and map types). Here are some examples:

```

1| // An array value containing four bool values.
2| [4]bool{false, true, true, false}
3|

```

```

4| // A slice value which contains three words.
5| []string{"break", "continue", "fallthrough"}
6|
7| // A map value containing some key-value pairs.
8| map[string]int{"C": 1972, "Python": 1991, "Go": 2009}

```

Each key-element pair between the braces of a map composite literal is also called an entry.

There are several variants for array and slice composite literals:

```

1| // The followings slice composite literals
2| // are equivalent to each other.
3| []string{"break", "continue", "fallthrough"}
4| []string{0: "break", 1: "continue", 2: "fallthrough"}
5| []string{2: "fallthrough", 1: "continue", 0: "break"}
6| []string{2: "fallthrough", 0: "break", "continue"}
7|
8| // The followings array composite literals
9| // are equivalent to each other.
10| [4]bool{false, true, true, false}
11| [4]bool{0: false, 1: true, 2: true, 3: false}
12| [4]bool{1: true, true}
13| [4]bool{2: true, 1: true}
14| [...]bool{false, true, true, false}
15| [...]bool{3: false, 1: true, true}

```

In the last two literals, the `...`s mean we want to let compilers deduce the lengths for the corresponding array values.

From the above examples, we know that element indexes (keys) are optional in array and slice composite literals. In an array or slice composite literal,

- if an index is present, it is not needed to be a typed value of the key type `int`, but it must be a non-negative constant representable as a value of type `int`. And if it is typed, its type must be a [basic integer type](#) (§6).
- **an element which index is absent uses the previous element's index plus one as its index.**
- if the index of the first element is absent, its index is zero.

The keys of entries in a map literal must not be absent, they can be non-constants.

```

1| var a uint = 1
2| var _ = map[uint]int {a : 123} // okay
3|
4| // The following two lines fail to compile,
5| // for "a" is not a constant key/index.

```

```

6| var _ = []int{a: 100} // error
7| var _ = [5]int{a: 100} // error

```

Constant keys in one specific composite literal [can't be duplicate](#) (§50).

Literal Representations of Zero Values of Container Types

Like structs, the zero value of an array type A can be represented with the composite literal A{}.

For example, the zero value of type [100]int can be denoted as [100]int{}. All elements stored in the zero value of an array type are zero values of the element type of the array type.

Like pointer types, zero values of all slice and map types are represented with the predeclared nil.

BTW, there are some other kinds of types whose zero values are also represented by nil, including later to be introduced function, channel and interface types.

When an array variable is declared without being specified an initial value, memory has been allocated for the elements of the zero array value. The memory for the elements of a nil slice or map value has not been allocated yet.

Please note, []T{} represents a blank slice value (with zero elements) of slice type []T, it is different from []T(nil). The same situation is for map[K]T{} and map[K]T(nil).

Composite Literals Are Unaddressable but Can Be Taken Addresses

We have learned that [struct composite literals can be taken addresses directly](#) (§16) before. Container composite literals have no exceptions here.

Example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     pm := &map[string]int{"C": 1972, "Go": 2009}
7|     ps := &[]string{"break", "continue"}
8|     pa := &[...]bool{false, true, true, false}
9|     fmt.Printf("%T\n", pm) // *map[string]int

```

```

10|     fmt.Printf("%T\n", ps) // *[]string
11|     fmt.Printf("%T\n", pa) // *[4]bool
12| }
```

Nested Composite Literals Can Be Simplified

If a composite literal nested many other composite literals, then those nested composed literals can simplified to the form `{...}`.

For example, the slice value literal

```

1| // A slice value of a type whose element type is
2| // *[4]byte. The element type is a pointer type
3| // whose base type is [4]byte. The base type is
4| // an array type whose element type is "byte".
5| var heads = []*[4]byte{
6|     &[4]byte{'P', 'N', 'G', ' '},
7|     &[4]byte{'G', 'I', 'F', ' '},
8|     &[4]byte{'J', 'P', 'E', 'G'},
9| }
```

can be simplified to

```

1| var heads = []*[4]byte{
2|     {'P', 'N', 'G', ' '},
3|     {'G', 'I', 'F', ' '},
4|     {'J', 'P', 'E', 'G'},
5| }
```

The array value literal in the following example

```

1| type language struct {
2|     name string
3|     year int
4| }
5|
6| var _ = [...]language{
7|     language{"C", 1972},
8|     language{"Python", 1991},
9|     language{"Go", 2009},
10| }
```

can be simplified to

```

1| var _ = [...]language{
2|     {"C", 1972},
3|     {"Python", 1991},
4|     {"Go", 2009},
5| }
```

And the map value literal in the following example

```

1| type LangCategory struct {
2|     dynamic bool
3|     strong   bool
4| }
5|
6| // A value of map type whose key type is
7| // a struct type and whose element type
8| // is another map type "map[string]int".
9| var _ = map[LangCategory]map[string]int{
10|     LangCategory{true, true}: map[string]int{
11|         "Python": 1991,
12|         "Erlang": 1986,
13|     },
14|     LangCategory{true, false}: map[string]int{
15|         "JavaScript": 1995,
16|     },
17|     LangCategory{false, true}: map[string]int{
18|         "Go": 2009,
19|         "Rust": 2010,
20|     },
21|     LangCategory{false, false}: map[string]int{
22|         "C": 1972,
23|     },
24| }
```

can be simplified to

```

1| var _ = map[LangCategory]map[string]int{
2|     {true, true}: {
3|         "Python": 1991,
4|         "Erlang": 1986,
5|     },
6|     {true, false}: {
7|         "JavaScript": 1995,
8|     },
9|     {false, true}: {
10|         "Go": 2009,
```

```

11|     "Rust": 2010,
12|   },
13|   {false, false}: {
14|     "C": 1972,
15|   },
16| }
```

Please notes, in the above several examples, the comma following the last item in each composite literal can't be omitted. Please read [the line break rules in Go](#) (§28) for more information later.

Compare Container Values

As which has mentioned in the article [overview of Go type system](#) (§14), map and slice types are incomparable types. So map and slice types can't be used as map key types.

Although a slice or map value can't be compared with another slice or map value (or itself), it can be compared to the bare untyped `nil` identifier to check whether or not the slice or map value is a zero value.

Most array types are comparable, except the ones whose element types are incomparable types.

When comparing two array values, each pair of the corresponding elements will be compared. We can think element pairs are compared by their index order. The two array values are equal only if all of their corresponding elements are equal. The comparison stops in advance when a pair of elements is found unequal or [a panic occurs](#) (§23).

Example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|   var a [16]byte
7|   var s []int
8|   var m map[string]int
9|
10|  fmt.Println(a == a) // true
11|  fmt.Println(m == nil) // true
12|  fmt.Println(s == nil) // true
13|  fmt.Println(nil == map[string]int{}) // false
14|  fmt.Println(nil == []int{}) // false
15| }
```

```

16| // The following lines fail to compile.
17| /*
18| _ = m == m
19| _ = s == s
20| _ = m == map[string]int(nil)
21| _ = s == []int(nil)
22| var x [16][]int
23| _ = x == x
24| var y [16]map[int]bool
25| _ = y == y
26| */
27| }
```

Check Lengths and Capacities of Container Values

Besides the length property, each container value also has a capacity property. The capacity of an array is always equal to the length of the array. The capacity of a non-nil map can be viewed as unlimited. So, in practice, only capacities of slice values are meaningful. The capacity of a slice is always equal to or larger than the length of the slice. The meaning of slice capacities will be explained in the section after next.

We can use the built-in `len` function to get the length of a container value, and use the built-in `cap` function to get the capacity of a container value. Each of the two functions returns an `int` typed result or an untyped result which default type is `int`, depending on whether or not the passed argument is a constant expression. As the capacity of any map value is unlimited, the built-in `cap` function doesn't apply to map values.

The length and capacity of an array value can never change. The lengths and capacities of all values of an array type always equal to the length of the array type. The length and capacity of a slice value may change at run time. So slices can be viewed as dynamic arrays. Slices are much more flexible than arrays and are used more popularly than arrays in practice.

Example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     var a [5]int
7|     fmt.Println(len(a), cap(a)) // 5 5
8|     var s []int
```

```

9|     fmt.Println(len(s), cap(s)) // 0 0
10|    s, s2 := []int{2, 3, 5}, []bool{}
11|    fmt.Println(len(s), cap(s)) // 3 3
12|    fmt.Println(len(s2), cap(s2)) // 0 0
13|    var m map[int]bool
14|    fmt.Println(len(m)) // 0
15|    m, m2 := map[int]bool{1: true, 0: false}, map[int]int{}
16|    fmt.Println(len(m), len(m2)) // 2 0
17| }
```

The length and capacity of each slice shown in the above specified example value are equal. This is not true for every slice value. We will use some slices whose respective lengths and capacities are not equal in the following sections.

Retrieve and Modify Container Elements

The element associated to key k stored in a container value v is represented with the element indexing syntax form $v[k]$.

For a use of $v[k]$, assume v is an array or slice,

- if k is a constant, then it must satisfy [the requirements described above](#) for the indexes in container composite literals. In addition, if v is an array, the k must be smaller than the length of the array.
- if k is a non-constant value, it must be a value of any basic integer type. In addition, it must be larger than or equal to zero and smaller than $\text{len}(v)$, otherwise, a run-time panic will occur.
- if v is a nil slice, a run-time panic will occur.

For a use of $v[k]$, assume v is a map, then k must be assignable to values of the element type of the map type, and

- if k is an interface value whose dynamic type is incomparable, a panic will occur at run time.
- if $v[k]$ is used as a destination value in an assignment and v is a nil map, a panic will occur at run time.
- if $v[k]$ is used to retrieve the element value corresponding key k in map v , then no panics will occur, even if v is a nil map. (Assume the evaluation of k will not panic.)
- if $v[k]$ is used to retrieve the element value corresponding key k in map v , and the map v doesn't contain an entry with key k , $v[k]$ results in a zero value of the element type of the corresponding map type of v . Generally, $v[k]$ is viewed as a single-value expression.

However, when `v[k]` is used as the only source value expression in an assignment, it can be viewed as a multi-value expression and result a second optional untyped boolean value, which indicates whether or not the map `v` contains an entry with key `k`.

An example of container element accesses and modifications:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     a := [3]int{-1, 0, 1}
7|     s := []bool{true, false}
8|     m := map[string]int{"abc": 123, "xyz": 789}
9|     fmt.Println(a[2], s[1], m["abc"])      // retrieve
10|    a[2], s[1], m["abc"] = 999, true, 567 // modify
11|    fmt.Println(a[2], s[1], m["abc"])      // retrieve
12|
13|    n, present := m["hello"]
14|    fmt.Println(n, present, m["hello"]) // 0 false 0
15|    n, present = m["abc"]
16|    fmt.Println(n, present, m["abc"]) // 567 true 567
17|    m = nil
18|    fmt.Println(m["abc"]) // 0
19|
20|    // The two lines fail to compile.
21|    /*
22|     _ = a[3] // index 3 out of bounds
23|     _ = s[-1] // index must be non-negative
24|    */
25|
26|    // Each of the following lines can cause a panic.
27|    _ = a[n]          // panic: index out of range
28|    _ = s[n]          // panic: index out of range
29|    m["hello"] = 555 // panic: assign to entry in nil map
30| }
```

Recall the Internal Structure Definition of Slice Types

To understand slice types and values better and explain slices easier, we need to have an impression on the internal structure of slice types. From the last article, [value parts](#) (§17), we learned that the internal structure of slice types defined by the standard Go compiler/runtime is like

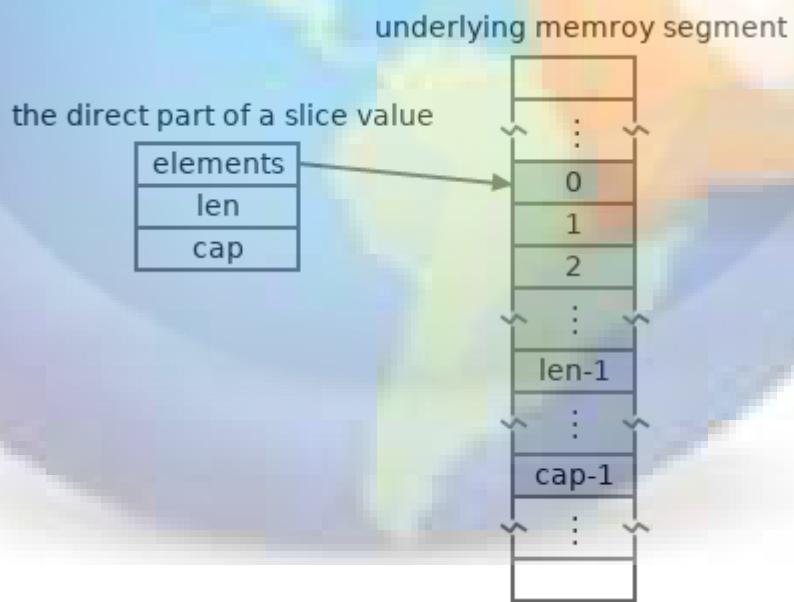
```

1| type _slice struct {
2|     elements unsafe.Pointer // referencing underlying elements
3|     len       int          // length
4|     cap       int          // capacity
5| }

```

The internal structure definitions used by other compilers/runtimes implementations may be not the exact same but would be similar. The following explanations are based on the official slice implementation.

The above shown internal structure explains the memory layouts of the direct parts of slice values. The `len` field of the direct part of a slice indicates the length of the slice, and the `cap` field indicates the capacity of the slice. The following picture depicts one possible memory layout of a slice value.



Although the underlying memory segment which hosts the elements of a slice may be very large, the slice may be only aware of a sub-segment of the memory segment. For example, in the above picture, the slice is only aware of the middle grey sub-segment of the whole memory segment.

For the slice depicted in the above picture, the elements from index `len` to index `cap` (exclusive) don't belong to the elements of the slice. They are just some redundant element slots for the depicted slice, but they may be effective elements of other slices or another array.

The next section will explain how to append elements to a base slice and yield a new slice by using the built-in `append` function. The result slice of an `append` function call may share starting elements with the base slice or not, depending on the capacity (and length) of the base slice and how many elements are appended.

When the slice is used as the base slice in an `append` function call,

- if the number of appended elements is larger than the number of the redundant element slots of the base slice, a new underlying memory segment will be allocated for the result slice, thus the result slice and the base slice will not share any elements.
- otherwise, no new underlying memory segments will be allocated for the result slice, and the elements of the base slice also belong to the elements of the result slice. In other words, the two slices share some elements and all of their elements are hosted on the same underlying memory segment.

The section after next will show a picture which describes both of the two possible cases in appending slice elements.

There are more routes which lead to the elements of two slices are hosted on the same underlying memory segment. Such as assignments and the below to be introduced subslice operations.

Note, generally, we can't modify the three fields of a slice value individually, except through the [reflection](#) and [unsafe](#) (§25) ways. In other words, generally, to modify a slice value, its three fields must be modified together. Generally, this is achieved by assigning another slice value (of the same slice type) to the slice which needs to be modified.

Container Assignments

If a map is assigned to another map, then the two maps will share all (underlying) elements. Appending elements into (or deleting elements from) one map will reflect on the other map.

Like map assignments, if a slice is assigned to another slice, they will share all (underlying) elements. Their respective lengths and capacities equal to each other. However, if the length/capacity of one slice changes later, the change will not reflect on the other slice.

When an array is assigned to another array, all the elements are copied from the source one to the destination one. The two arrays don't share any elements.

Example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     m0 := map[int]int{0:7, 1:8, 2:9}
7|     m1 := m0
8|     m1[0] = 2
9|     fmt.Println(m0, m1) // map[0:2 1:8 2:9] map[0:2 1:8 2:9]

```

```

10|
11|     s0 := []int{7, 8, 9}
12|     s1 := s0
13|     s1[0] = 2
14|     fmt.Println(s0, s1) // [2 8 9] [2 8 9]
15|
16|     a0 := [...]int{7, 8, 9}
17|     a1 := a0
18|     a1[0] = 2
19|     fmt.Println(a0, a1) // [7 8 9] [2 8 9]
20| }
```

Append and Delete Container Elements

The syntax of appending a key-element pair (an entry) to a map is the same as the syntax of modifying a map element. For example, for a non-nil map value `m`, the following line

```
m[k] = e
```

put the key-element pair (`k`, `e`) into the map `m` if `m` doesn't contain an entry with key `k`, or modify the element value associated with `k` if `m` contains an entry with key `k`.

There is a built-in `delete` function which is used to delete an entry from a map. For example, the following line will delete the entry with key `k` from the map `m`. If the map `m` doesn't contain an entry with key `k`, it is a no-op, no panics will occur, even if `m` is a nil map.

```
delete(m, k)
```

An example shows how to append (put) entries to and delete entries from maps:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     m := map[string]int{"Go": 2007}
7|     m["C"] = 1972      // append
8|     m["Java"] = 1995  // append
9|     fmt.Println(m)    // map[C:1972 Go:2007 Java:1995]
10|    m["Go"] = 2009    // modify
11|    delete(m, "Java") // delete
12|    fmt.Println(m)    // map[C:1972 Go:2009]
13| }
```

Please note, before Go 1.12, the entry print order of a map is unspecified.

Array elements can neither be appended nor deleted, though elements of addressable arrays can be modified.

We can use the built-in `append` function to append multiple elements into a base slice and result a new slice. The result new slice contains the elements of the base slice and the appended elements. Please note, the base slice is not modified by the `append` function call. Surely, if we expect (and often in practice), we can assign the result slice to the base slice to modify the base slice.

There is not a built-in way to delete an element from a slice. We must use the `append` function and the subslice feature introduced below together to achieve this goal. Slice element deletions and insertions will be demoed in the below [more slice manipulations](#) section. Here, the following example only shows how to use the `append` function.

An example showing how to use the `append` function:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     s0 := []int{2, 3, 5}
7|     fmt.Println(s0, cap(s0)) // [2 3 5] 3
8|     s1 := append(s0, 7)      // append one element
9|     fmt.Println(s1, cap(s1)) // [2 3 5 7] 6
10|    s2 := append(s1, 11, 13) // append two elements
11|    fmt.Println(s2, cap(s2)) // [2 3 5 7 11 13] 6
12|    s3 := append(s0)        // <=> s3 := s0
13|    fmt.Println(s3, cap(s3)) // [2 3 5] 3
14|    s4 := append(s0, s0...) // double s0 as s4
15|    fmt.Println(s4, cap(s4)) // [2 3 5 2 3 5] 6
16|
17|    s0[0], s1[0] = 99, 789
18|    fmt.Println(s2[0], s3[0], s4[0]) // 789 99 2
19| }
```

Note, the built-in `append` function is a [variadic function](#) (§20). It has two parameters, the second one is a [variadic parameter](#) (§20).

Variadic functions will be explained in the article after next. Currently, we only need to know that there are two manners to pass variadic arguments. In the above example, line 8, line 10 and line 12 use one manner and line 14 uses the other manner. For the former manner, we can pass zero or more element values as the variadic arguments. For the latter manner, we must pass a slice as the only

variadic argument and which must be followed by three dots We can learn how to call variadic functions from the [the article after next](#) (§20).

In the above example, line 14 is equivalent to

```
s4 := append(s0, s0[0], s0[1], s0[2])
```

line 8 is equivalent to

```
s1 := append(s0, []int{7}...)
```

and line 10 is equivalent to

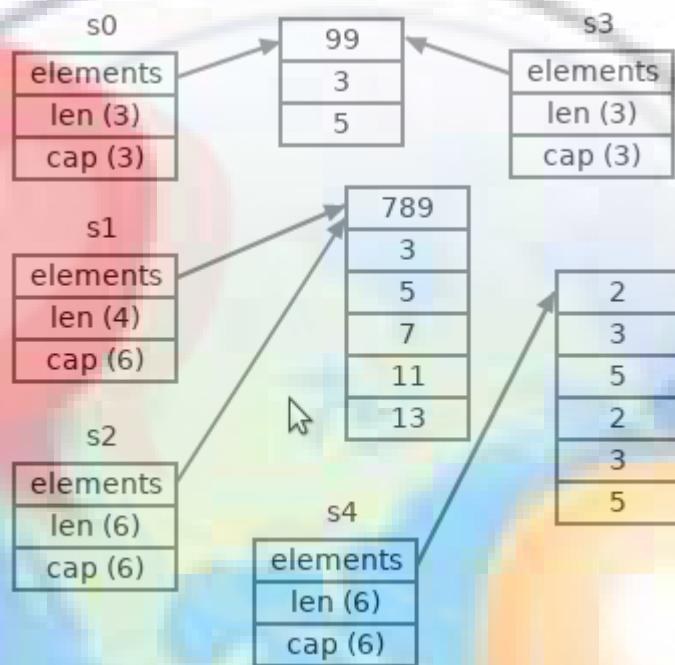
```
s2 := append(s1, []int{11, 13}...)
```

For the three-dot . . . manner, the `append` function doesn't require the variadic argument must be a slice with the same type as the first slice argument, but their element types must be identical. In other words, the two argument slices must share the same [underlying type](#) (§14).

In the above program,

- the `append` call at line 8 will allocate a new underlying memory segment for slice `s1`, for slice `s0` doesn't have enough redundant element slots to store the new appended element. The same situation is for the `append` call at line 14.
- the `append` call at line 10 will not allocate a new underlying memory segment for slice `s2`, for slice `s1` has enough redundant element slots to store the new appended elements.

So, `s1` and `s2` share some elements, `s0` and `s3` share all elements, and `s4` doesn't share elements with others. The following picture depicted the statuses of these slices at the end of the above program.



Please note that, when an `append` call allocate a new underlying memory segment for the result slice, the capacity of the result slice is compiler dependent. For the standard Go compiler, if the capacity of the base slice is small, the capacity of the result slice will be at least the double of the base slice, to avoid allocating underlying memory segments frequently when the result slice is used as the base slices in later possible `append` calls.

As mentioned above, we can assign the result slice to the base slice in an `append` call to append elements into the base slice. For example,

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     var s = append([]string(nil), "array", "slice")
7|     fmt.Println(s)      // [array slice]
8|     fmt.Println(cap(s)) // 2
9|     s = append(s, "map")
10|    fmt.Println(s)      // [array slice map]
11|    fmt.Println(cap(s)) // 4
12|    s = append(s, "channel")
13|    fmt.Println(s)      // [array slice map channel]
14|    fmt.Println(cap(s)) // 4
15| }
```

The first argument of an `append` function call can't be an untyped `nil` (up to Go 1.21).

Create Slices and Maps With the Built-in make Function

Besides using composite literals to create map and slice values, we can also use the built-in `make` function to create map and slice values. The built-in `make` function can't be used to create array values.

BTW, the built-in `make` function can also be used to create channels, which will be explained in the article [channels in Go](#) (§21) later.

Assume `M` is a map type and `n` is an integer, we can use the following two forms to create new maps of type `M`.

```
1| make(M, n)
2| make(M)
```

The first form creates a new empty map which is allocated with enough space to hold at least `n` entries without reallocating memory again. The second form only takes one argument, in which case a new empty map with enough space to hold a small number of entries without reallocating memory again. The small number is compiler dependent.

Note: the second argument `n` may be negative or zero, in which case it will be ignored.

Assume `S` is a slice type, `length` and `capacity` are two non-negative integers, `length` is not larger than `capacity`, we can use the following two forms to create new slices of type `S`. (The types of `length` and `capacity` are not required to be identical.)

```
1| make(S, length, capacity)
2| make(S, length)
```

The first form creates a new slice with the specified `length` and `capacity`. The second form only takes two arguments, in which case the `capacity` of the new created slice is the same as its `length`.

All the elements in the result slice of a `make` function call are initialized as the zero value (of the slice element type).

An example on how to use the built-in `make` function to create maps and slices:

```
1| package main
2|
3| import "fmt"
4|
```

```

5| func main() {
6|     // Make new maps.
7|     fmt.Println(make(map[string]int)) // map[]
8|     m := make(map[string]int, 3)
9|     fmt.Println(m, len(m)) // map[] 0
10|    m["C"] = 1972
11|    m["Go"] = 2009
12|    fmt.Println(m, len(m)) // map[C:1972 Go:2009] 2
13|
14|    // Make new slices.
15|    s := make([]int, 3, 5)
16|    fmt.Println(s, len(s), cap(s)) // [0 0 0] 3 5
17|    s = make([]int, 2)
18|    fmt.Println(s, len(s), cap(s)) // [0 0] 2 2
19| }
```

Allocate Containers With the Built-in `new` Function

From the article [pointers in Go](#) (§15), we learned that we can also call the built-in `new` function to allocate a value of any type and get a pointer which references the allocated value. The allocated value is a zero value of its type. For this reason, it is a nonsense to use `new` function to create map and slice values.

It is not totally a nonsense to allocate a zero value of an array type with the built-in `new` function. However, it is seldom to do this in practice, for it is more convenient to use composite literals to allocate arrays.

Example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     m := *new(map[string]int)    // <=> var m map[string]int
7|     fmt.Println(m == nil)        // true
8|     s := *new([]int)           // <=> var s []int
9|     fmt.Println(s == nil)        // true
10|    a := *new([5]bool)          // <=> var a [5]bool
11|    fmt.Println(a == [5]bool{}) // true
12| }
```

Addressability of Container Elements

Following are some facts about the addressabilities of container elements.

- Elements of addressable array values are also addressable. Elements of unaddressable array values are also unaddressable. The reason is each array value only consists of one [direct part](#) (§17).
- Elements of any slice value are always addressable, whether or not that slice value is addressable. This is because the elements of a slice are stored in the underlying (indirect) value part of the slice and the underlying part is always hosted on an allocated memory segment.
- Elements of map values are always unaddressable. Please read [this FAQ item](#) (§51) for reasons.

For example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     a := [5]int{2, 3, 5, 7}
7|     s := make([]bool, 2)
8|     pa2, ps1 := &a[2], &s[1]
9|     fmt.Println(*pa2, *ps1) // 5 false
10|    a[2], s[1] = 99, true
11|    fmt.Println(*pa2, *ps1) // 99 true
12|    ps0 := &[]string{"Go", "C"}[0]
13|    fmt.Println(*ps0) // Go
14|
15|    m := map[int]bool{1: true}
16|    _ = m
17|    // The following lines fail to compile.
18|    /*
19|     _ = &[3]int{2, 3, 5}[0]
20|     _ = &map[int]bool{1: true}[1]
21|     _ = &m[1]
22|     */
23| }
```

Unlike most other unaddressable values, which direct parts can not be modified, the direct part of a map element values can be modified, but can only be modified (overwritten) as a whole. For most

kinds of element types, this is not a big issue. However, if the element type of map type is an array type or struct type, things become some counter-intuitive.

From the last article, [value parts \(§17\)](#), we learned that each of struct and array values only consists of one direct part. So

- if the element type of a map is a struct type, we can not individually modify each field of an element (which is a struct) of the map.
- if the element type of a map is an array type, we can not individually modify each element of an element (which is an array) of the map.

Example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     type T struct{age int}
7|     mt := map[string]T{}
8|     mt["John"] = T{age: 29} // modify it as a whole
9|     ma := map[int][5]int{}
10|    ma[1] = [5]int{1: 789} // modify it as a whole
11|
12|    // The following two lines fail to compile,
13|    // for map elements can be modified partially.
14|    /*
15|     ma[1][1] = 123      // error
16|     mt["John"].age = 30 // error
17|    */
18|
19|    // Accesses are okay.
20|    fmt.Println(ma[1][1])      // 789
21|    fmt.Println(mt["John"].age) // 29
22| }
```

To make any expected modification work in the above example, the corresponding map element should be saved in a temporary variable firstly, then the temporary variable is modified as needed, in the end the corresponding map element is overwritten by the temporary variable. For example,

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
```

```

6| type T struct{age int}
7| mt := map[string]T{}
8| mt["John"] = T{age: 29}
9| ma := map[int][5]int{}
10| ma[1] = [5]int{1: 789}
11|
12| t := mt["John"] // a temporary copy
13| t.age = 30
14| mt["John"] = t // overwrite it back
15|
16| a := ma[1] // a temporary copy
17| a[1] = 123
18| ma[1] = a // overwrite it back
19|
20| fmt.Println(ma[1][1], mt["John"].age) // 123 30
21| }
```

Note: the just mentioned limit [might be lifted later](#).

Derive Slices From Arrays and Slices

We can derive a new slice from another (base) slice or a base addressable array by using the subslice syntax forms (Go specification calls them as slice syntax forms). The process is also often called as reslicing. The elements of the derived slice and the base array or slice are hosted on the same memory segment. In other words, the derived slice and the base array or slice may share some contiguous elements.

There are two subslice syntax forms (`baseContainer` is an array or slice):

```

1| baseContainer[low : high]           // two-index form
2| baseContainer[low : high : max] // three-index form
```

The two-index form is equivalent to

```
baseContainer[low : high : cap(baseContainer)]
```

So the two-index form is a special case of the three-index form. The two-index form is used much more popularly than the three-index form in practice.

Note, the three-index form is only supported since Go 1.2.

In a subslice expression, the `low`, `high` and `max` indexes must satisfy the following relation requirements.

```
// two-index form
0 <= low <= high <= cap(baseContainer)

// three-index form
0 <= low <= high <= max <= cap(baseContainer)
```

Indexes not satisfying these requirements may make the subslice expression fail to compile at compile time or panic at run time, depending on the base container type kind and whether or not the indexes are constants.

Note,

- the `low` and `high` indexes can be both larger than `len(baseContainer)`, as long as the above relations are all satisfied. But the two indexes must not be larger than `cap(baseContainer)`.
- a subslice expression will not cause a panic if `baseContainer` is a nil slice and all indexes used in the expression are zero. The result slice derived from a nil slice is still a nil slice.

The length of the result derived slice is equal to `high - low`, and the capacity of the result derived slice is equal to `max - low`. The length of a derived slice may be larger than the base container, but the capacity will never be larger than the base container.

In practice, for simplicity, we often omitted some indexes in subslice syntax forms. The omission rules are:

- if the `low` index is equal to zero, it can be omitted, either for two-index or three-index forms.
- if the `high` is equal to `len(baseContainer)`, it can be omitted, but only for two-index forms.
- the `max` can never be omitted in three-index forms.

For example, the following expressions are equivalent.

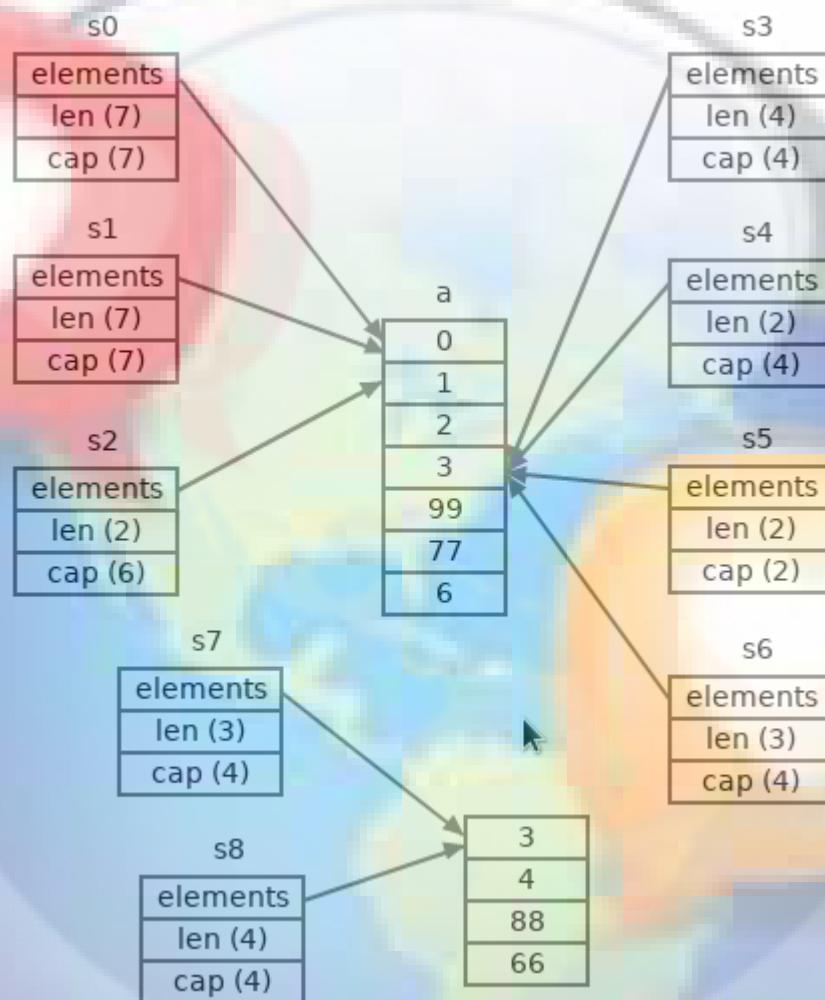
```
1| baseContainer[0 : len(baseContainer)]
2| baseContainer[: len(baseContainer)]
3| baseContainer[0 :]
4| baseContainer[:]
5| baseContainer[0 : len(baseContainer) : cap(baseContainer)]
6| baseContainer[: len(baseContainer) : cap(baseContainer)]
```

An example of using subslice syntax forms:

```
1| package main
2|
3| import "fmt"
```

```
4|  
5| func main() {  
6|     a := [...]int{0, 1, 2, 3, 4, 5, 6}  
7|     s0 := a[:]      // <=> s0 := a[0:7:7]  
8|     s1 := s0[:]     // <=> s1 := s0  
9|     s2 := s1[1:3]   // <=> s2 := a[1:3]  
10|    s3 := s1[3:]    // <=> s3 := s1[3:7]  
11|    s4 := s0[3:5]   // <=> s4 := s0[3:5:7]  
12|    s5 := s4[:2:2]  // <=> s5 := s0[3:5:5]  
13|    s6 := append(s4, 77)  
14|    s7 := append(s5, 88)  
15|    s8 := append(s7, 66)  
16|    s3[1] = 99  
17|    fmt.Println(len(s2), cap(s2), s2) // 2 6 [1 2]  
18|    fmt.Println(len(s3), cap(s3), s3) // 4 4 [3 99 77 6]  
19|    fmt.Println(len(s4), cap(s4), s4) // 2 4 [3 99]  
20|    fmt.Println(len(s5), cap(s5), s5) // 2 2 [3 99]  
21|    fmt.Println(len(s6), cap(s6), s6) // 3 4 [3 99 77]  
22|    fmt.Println(len(s7), cap(s7), s7) // 3 4 [3 4 88]  
23|    fmt.Println(len(s8), cap(s8), s8) // 4 4 [3 4 88 66]  
24| }
```

The following picture depicts the final memory layouts of the array and slice values used in the above example.



From the picture, we know that the elements of slice `s7` and `s8` are hosted on a different underlying memory segment than the other containers. The elements of the other slices are hosted on the same memory segment hosting the array `a`.

Please note that, subslice operations may cause kind-of memory leaking. For example, half of the memory allocated for the return slice of a call to the following function will be wasted unless the returned slice becomes unreachable (if no other slices share the underlying element memory segment with the returned slice).

```

1| func f() []int {
2|     s := make([]int, 10, 100)
3|     return s[50:60]
4|

```

Please note that, in the above function, the lower index (50) is larger than the length (10) of `s`, which is allowed.

Convert Slice to Array Pointer

Since Go 1.17, a slice may be converted to an array pointer. In such a conversion, if the length of the base array type of the pointer type is larger than the length of the slice, a panic occurs.

An example:

```

1| package main
2|
3| type S []int
4| type A [2]int
5| type P *A
6|
7| func main() {
8|     var x []int
9|     var y = make([]int, 0)
10|    var x0 = (*[0]int)(x) // okay, x0 == nil
11|    var y0 = (*[0]int)(y) // okay, y0 != nil
12|    _, _ = x0, y0
13|
14|    var z = make([]int, 3, 5)
15|    var _ = (*[3]int)(z) // okay
16|    var _ = (*[2]int)(z) // okay
17|    var _ = (*A)(z)      // okay
18|    var _ = P(z)         // okay
19|
20|    var w = S(z)
21|    var _ = (*[3]int)(w) // okay
22|    var _ = (*[2]int)(w) // okay
23|    var _ = (*A)(w)      // okay
24|    var _ = P(w)         // okay
25|
26|    var _ = (*[4]int)(z) // will panic
27| }
```

Convert Slice to Array

Since Go 1.20, a slice may be converted to an array. In such a conversion, if the length of the array type is larger than the length of the slice, a panic occurs. The slice and the result array don't share any element.

An example:

```

1| package main
2|
3| import "fmt"
```

```

4|
5| func main() {
6|     var s = []int{0, 1, 2, 3}
7|     var a = [3]int(s[1:])
8|     s[2] = 9
9|     fmt.Println(s) // [0 1 9 3]
10|    fmt.Println(a) // [1 2 3]
11|
12|    _ = [3]int(s[:2]) // panic
13|

```

Copy Slice Elements With the Built-in copy Function

We can use the built-in `copy` function to copy elements from one slice to another, the types of the two slices are not required to be identical, but their element types must be identical. In other words, the two argument slices must share the same underlying type. The first parameter of the `copy` function is the destination slice and the second one is the source slice. The two parameters can overlap some elements. `copy` function returns the number of elements copied, which will be the smaller one of the lengths of the two parameters.

With the help of the subslice syntax, we can use the `copy` function to copy elements between two arrays or between an array and a slice.

An example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     type Ta []int
7|     type Tb []int
8|     dest := Ta{1, 2, 3}
9|     src := Tb{5, 6, 7, 8, 9}
10|    n := copy(dest, src)
11|    fmt.Println(n, dest) // 3 [5 6 7]
12|    n = copy(dest[1:], dest)
13|    fmt.Println(n, dest) // 2 [5 5 6]
14|
15|    a := [4]int{} // an array
16|    n = copy(a[:], src)
17|    fmt.Println(n, a) // 4 [5 6 7 8]
18|    n = copy(a[:], a[2:])

```

```

19|     fmt.Println(n, a) // 2 [7 8 7 8]
20| }
```

Note, as a special case, the built-in `copy` function can be used to [copy bytes from a string to a byte slice](#) (§19).

Neither of the two arguments of a `copy` function call can be an untyped `nil` value (up to Go 1.21).

Container Element Iterations

In Go, keys and elements of a container value can be iterated with the following syntax:

```

for key, element = range aContainer {
    // use key and element ...
}
```

where `for` and `range` are two keywords, `key` and `element` are called iteration variables. If `aContainer` is a slice or an array (or an array pointer, see below), then the type of `key` must be built-in type `int`.

The assignment sign `=` can be a short variable declaration sign `:=`, in which case the two iteration variables are both two new declared variables which are only visible within the `for-range` code block body, if `aContainer` is a slice or an array (or an array pointer), then the type of `key` is deduced as `int`.

Like the traditional `for` loop block, each `for-range` loop block creates two code blocks, an implicit one and an explicit one which is formed by using `{}`. The explicit one is nested in the implicit one.

Like `for` loop blocks, `break` and `continue` statements can also be used in `for-range` loop blocks,

Example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     m := map[string]int{"C": 1972, "C++": 1983, "Go": 2009}
7|     for lang, year := range m {
8|         fmt.Printf("%v: %v \n", lang, year)
```

```

9| }
10|
11| a := [...]int{2, 3, 5, 7, 11}
12| for i, prime := range a {
13|     fmt.Printf("%v: %v \n", i, prime)
14| }
15|
16| s := []string{"go", "defer", "goto", "var"}
17| for i, keyword := range s {
18|     fmt.Printf("%v: %v \n", i, keyword)
19| }
20|

```

The form `for-range` code block syntax has several variants:

```

1| // Ignore the key iteration variable.
2| for _, element = range aContainer {
3|     // ...
4| }
5|
6| // Ignore the element iteration variable.
7| for key, _ = range aContainer {
8|     element = aContainer[key]
9|     // ...
10| }
11|
12| // The element iteration variable is omitted.
13| // This form is equivalent to the last one.
14| for key = range aContainer {
15|     element = aContainer[key]
16|     // ...
17| }
18|
19| // Ignore both the key and element iteration variables.
20| for _, _ = range aContainer {
21|     // This variant is not much useful.
22| }
23|
24| // Both the key and element iteration variables are
25| // omitted. This form is equivalent to the last one.
26| for range aContainer {
27|     // This variant is not much useful.
28| }

```

Iterating over nil maps or nil slices is allowed. Such iterations are no-ops.

Some details about iterations over maps are listed here.

- For a map, the entry order in an iteration is not guaranteed to be the same as the next iteration, even if the map is not modified between the two iterations. By Go specification, the order is unspecified (kind-of randomized).
- If a map entry (a key-element pair) which has not yet been reached is removed during an iteration, then the entry will not be iterated in the same iteration for sure.
- If a map entry is created during an iteration, that entry may be iterated during the same iteration, or not.

If it is promised that there are no other goroutines manipulating a map `m`, then the following code is guaranteed to clear all entries (but the ones with keys as `NaN`) stored in the map `m`:

```
1| for key := range m {
2|     delete(m, key)
3| }
```

(Note: Go 1.21 introduced a [clear builtin function](#), which may be used to clear all entries of a map, including those with keys as `NaN`.)

Surely, array and slice elements can also be iterated by using the traditional `for` loop block:

```
1| for i := 0; i < len(anArrayOrSlice); i++ {
2|     // ... use anArrayOrSlice[i]
3| }
```

For a `for-range` loop block (whether `=` or `:=` before `range`)

```
for key, element = range aContainer {...}
```

there are two important facts.

1. The ranged container is **a copy** of `aContainer`. Please note, [only the direct part of aContainer is copied](#) (§17). The container copy is anonymous, so there are no ways to modify it.
 - If the `aContainer` is an array, then the modifications made on the array elements during the iteration will not be reflected to the iteration variables. The reason is that the copy of the array doesn't share elements with the array.
 - If the `aContainer` is a slice or map, then the modifications made on the slice or map elements during the iteration will be reflected to the iteration variables. The reason is that the clone of the slice (or map) shares all elements (entries) with the slice (or map).

2. A key-element pair of the copy of `aContainer` will be assigned (copied) to the iteration variable pair at each iteration step, so the modifications made on **the direct parts** of the iteration variables will not be reflected to the elements (and keys for maps) stored in `aContainer`. (For this fact, and as using `for-range` loop blocks is the only way to iterate map keys and elements, it is recommended not to use large-size types as map key and element types, to avoid large copy burdens.)

An example which proves the first and second facts.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     type Person struct {
7|         name string
8|         age   int
9|     }
10|    persons := [2]Person {"Alice", 28}, {"Bob", 25}
11|    for i, p := range persons {
12|        fmt.Println(i, p)
13|
14|        // This modification has no effects on
15|        // the iteration, for the ranged array
16|        // is a copy of the persons array.
17|        persons[1].name = "Jack"
18|
19|        // This modification has not effects on
20|        // the persons array, for p is just a
21|        // copy of a copy of one persons element.
22|        p.age = 31
23|    }
24|    fmt.Println("persons:", &persons)
25| }
```

The output:

```

0 {Alice 28}
1 {Bob 25}
persons: &[{Alice 28} {Jack 25}]
```

If we change the array in the above to a slice, then the modification on the slice during the iteration has effects on the iteration, but the modification on the iteration variable still has no effects on the slice.

```

1| ...
2|
3|     // A slice.
4|     persons := []Person {"Alice", 28}, {"Bob", 25}
5|     for i, p := range persons {
6|         fmt.Println(i, p)
7|
8|             // Now this modification has effects
9|             // on the iteration.
10|            persons[1].name = "Jack"
11|
12|            // This modification still has not
13|            // any real effects.
14|            p.age = 31
15|        }
16|        fmt.Println("persons:", &persons)
17|    }

```

The output becomes to:

```

0 {Alice 28}
1 {Jack 25}
persons: &[{Alice 28} {Jack 25}]

```

The following example proves the second facts.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     m := map[int]struct{ dynamic, strong bool } {
7|         0: {true, false},
8|         1: {false, true},
9|         2: {false, false},
10|     }
11|
12|     for _, v := range m {
13|         // This following line has no effects on the map m.
14|         v.dynamic, v.strong = true, true
15|     }
16|
17|     fmt.Println(m[0]) // {true false}
18|     fmt.Println(m[1]) // {false true}

```

```

19|     fmt.Println(m[2]) // {false false}
20| }
```

The cost of a slice or map assignment is small, but the cost of an array assignment is large if the size of the array type is large. So, generally, it is not a good idea to range over a large array. We can range over a slice derived from the array, or range over a pointer to the array (see the next section for details).

For an array or slice, if the size of its element type is large, then, generally, it is also not a good idea to use the second iteration variable to store the iterated element at each loop step. For such arrays and slices, we should use the one-iteration-variable `for-range` loop variant or the traditional `for` loop to iterate their elements. In the following example, the loop in function `fa` is much less efficient than the loop in function `fb`.

```

1| type Buffer struct {
2|     start, end int
3|     data        [1024]byte
4| }
5|
6| func fa(buffers []Buffer) int {
7|     numUnreads := 0
8|     for _, buf := range buffers {
9|         numUnreads += buf.end - buf.start
10|    }
11|    return numUnreads
12| }
13|
14| func fb(buffers []Buffer) int {
15|     numUnreads := 0
16|     for i := range buffers {
17|         numUnreads += buffers[i].end - buffers[i].start
18|     }
19|     return numUnreads
20| }
```

Before Go 1.22, for a `for-range` loop block (note the sign before before `range` is `:=`)

```

1| for key, element := range aContainer {...}
```

all key-element pairs will be assigned to **the same** iteration variable pair. However, since Go 1.22, each key-element pair will be assigned to a **distinctive** iteration variable pair (a.k.a. a distinctive instance will be created for each iteration variable in each loop step).

The following example shows the behavior differences between 1.21- and 1.22+ Go versions.

```

1| // forrange1.go
2| package main
3|
4| import "fmt"
5|
6| func main() {
7|     for i, n := range []int{0, 1, 2} {
8|         defer func() {
9|             fmt.Println(i, n)
10|         }()
11|     }
12| }
```

Use different versions of Go Toolchain to run the code ([gotv](#) 是一个工具，用于管理和使用多个并存的官方 Go 工具链版本；即将到来的 Go 1.22 版本将基于当前 tip 版本)，我们将得到不同的输出：

```

1| $ gotv 1.21. run forrange1.go
2| [Run]: $HOME/.cache/gotv/tag_go1.21.2/bin/go run forrange1.go
3| 2 3
4| 2 3
5| 2 3
6| $ gotv :tip run forrange1.go
7| [Run]: $HOME/.cache/gotv/bra_master/bin/go run forrange1.go
8| 2 3
9| 1 2
10| 0 1
```

Another example：

```

1| // forrange2.go
2| package main
3|
4| import "fmt"
5|
6| func main() {
7|     var m = map[*int]uint32{}
8|     for i, n := range []int{1, 2, 3} {
9|         m[&i]++
10|        m[&n]++
11|    }
12|    fmt.Println(len(m))
13| }
```

使用不同的 Go 工具链版本，我们将得到以下输出：

```

1| $ gotv 1.21. run forrange2.go
2| [Run]: $HOME/.cache/gotv/tag_go1.21.2/bin/go run forrange2.go
3| 2
4| $ gotv :tip run forrange2.go
5| [Run]: $HOME/.cache/gotv/bra_master/bin/go run forrange2.go
6| 6

```

Therefore, this is a semantic change that breaks backward compatibility. But the new semantic is actually more in line with people's expectations; and in theory, no old logic-correct code has been found will be broken by this change.

Use Array Pointers as Arrays

In many scenarios, we can use a pointer to an array as the array.

We can range over a pointer to an array to iterate the elements of the array. For arrays with large lengths, this way is much more efficient, for copying a pointer is much more efficient than copying a large-size array. In the following example, the two loop blocks are equivalent and both are efficient.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     var a [100]int
7|
8|     // Copying a pointer is cheap.
9|     for i, n := range &a {
10|         fmt.Println(i, n)
11|     }
12|
13|     // Copying a slice is cheap.
14|     for i, n := range a[:] {
15|         fmt.Println(i, n)
16|     }
17| }

```

If the second iteration in a `for-range` loop is neither ignored nor omitted, then range over a nil array pointer will panic. In the following example, each of the first two loop blocks will print five indexes, however, the last one will produce a panic.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     var p *[5]int // nil
7|
8|     for i, _ := range p { // okay
9|         fmt.Println(i)
10|    }
11|
12|     for i := range p { // okay
13|         fmt.Println(i)
14|    }
15|
16|     for i, n := range p { // panic
17|         fmt.Println(i, n)
18|    }
19| }
```

Array pointers can also be used to index array elements. Indexing array elements through a nil array pointer produces a panic.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     a := [5]int{2, 3, 5, 7, 11}
7|     p := &a
8|     p[0], p[1] = 17, 19
9|     fmt.Println(a) // [17 19 5 7 11]
10|    p = nil
11|    _ = p[0] // panic
12| }
```

We can also derive slices from array pointers. Deriving slices from a nil array pointer produce a panic.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     pa := &[5]int{2, 3, 5, 7, 11}
```

```

7|     s := pa[1:3]
8|     fmt.Println(s) // [3 5]
9|     pa = nil
10|    s = pa[0:0] // panic
11|    // Should this line execute, it also panics.
12|    _ = (*[0]byte)(nil)[:]
13|

```

We can also pass array pointers as the arguments of the built-in `len` and `cap` functions. Nil array pointer arguments for the two functions will not produce panics.

```

1| var pa *[5]int // == nil
2| fmt.Println(len(pa), cap(pa)) // 5 5

```

The `memclr` Optimization

Assume `t0` is a literal presentation of the zero value of type `T`, and `a` is an array or slice which element type is `T`, then the standard Go compiler will translate the following one-iteration-variable `for-range` loop block

```

1| for i := range a {
2|     a[i] = t0
3|

```

to an [internal `memclr` call](#), generally which is faster than resetting each element one by one.

The optimization was adopted in the standard Go compiler 1.5.

Since Go Toolchain 1.19, the optimization also works if the ranged value is an array pointer.

Note: Go 1.21 introduced a `clear` builtin function, which can be used to clear map entries and reset slice elements. We should try to use that function instead of relying on this optimization to reset slice/array elements. Please read the next section for the `clear` builtin function.

Use the built-in `clear` function to clear map entries and reset slice elements

Go 1.21 introduced `clear` builtin function. This function is used to clear map entries and reset slice elements.

An example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     s := []int{1, 2, 3}
7|     clear(s)
8|     fmt.Println(s) // [0 0 0]
9|
10|    a := [4]int{5, 6, 7, 8}
11|    clear(a[1:3])
12|    fmt.Println(a) // [5 0 0 8]
13|
14|    m := map[float64]float64{}
15|    x := 0.0
16|    m[x] = x
17|    x /= x // x is NaN now
18|    m[x] = x
19|    fmt.Println(len(m)) // 2
20|    for k := range m {
21|        delete(m, k)
22|    }
23|    fmt.Println(len(m)) // 1
24|    clear(m)
25|    fmt.Println(len(m)) // 0
26| }
```

We can find that the `clear` function can even delete map entries with keys as `Nan`.

Calls to the Built-in `len` and `cap` Functions May Be Evaluated at Compile Time

If the argument passed to a built-in function `len` or `cap` function call is an array or an array pointer value, then the call is evaluated at compile time and the result of the call is a typed constant with type as the built-in type `int`. The result can be bound to named constants.

Example:

```

1| package main
2|
3| import "fmt"
4|
5| var a [5]int
```

```

6| var p *[7]string
7|
8| // N and M are both typed constants.
9| const N = len(a)
10| const M = cap(p)
11|
12| func main() {
13|     fmt.Println(N) // 5
14|     fmt.Println(M) // 7
15| }
```

Modify the Length and Capacity Properties of a Slice Individually

Above has mentioned, generally, the length and capacity of a slice value can't be modified individually. A slice value can only be overwritten as a whole by assigning another slice value to it. However, we can modify the length and capacity of a slice individually by using reflections. Reflection will be explained in [a later article](#) (§27) in detail.

Example:

```

1| package main
2|
3| import (
4|     "fmt"
5|     "reflect"
6| )
7|
8| func main() {
9|     s := make([]int, 2, 6)
10|    fmt.Println(len(s), cap(s)) // 2 6
11|
12|    reflect.ValueOf(&s).Elem().SetLen(3)
13|    fmt.Println(len(s), cap(s)) // 3 6
14|
15|    reflect.ValueOf(&s).Elem().SetCap(5)
16|    fmt.Println(len(s), cap(s)) // 3 5
17| }
```

The second argument passed to the `reflect.SetLen` function must not be larger than the current capacity of the argument slice `s`. The second argument passed to the `reflect.SetCap` function

must not be smaller than the current length of the argument slice `s` and larger than the current capacity of the argument slice `s`. Otherwise, a panic will occur.

The reflection way is very inefficient, it is slower than a slice assignment.

More Slice Manipulations

Go doesn't support more built-in slice operations, such as slice clone, element deletion and insertion. We must compose the built-in ways to achieve those operations.

In the following examples in the current section, assume `s` is the talked slice, `T` is its element type and `t0` is a zero value literal representation of `T`.

Clone slices

For the current Go verison (1.21), the simplest way to clone a slice is:

```
sClone := append(s[:0:0], s...)
```

We can also use the following line instead, but comparing with the above way, it has one imperfection. If the source slice `s` is blank non-nil slice, then the result slice is nil.

```
sClone := append([]T(nil), s...)
```

The above append ways have one drawback. They both might allocate (then initialize) more elements than needed. We can also use the following ways to avoid this drawback:

```
// The two-line make+copy way.
sClone := make([]T, len(s))
copy(sClone, s)

// Or the following one-line make+append way.
// As of Go Toolchain v1.21.n, this way is a bit
// slower than the above two-line make+copy way.
sClone := append(make([]T, 0, len(s)), s...)
```

The "make" ways have a drawback that if `s` is a nil slice, then they both result a non-nil blank slice. However, in practice, we often needn't to pursue the perfection. If we do, then three more lines are needed:

```
var sClone []T
if s != nil {
```

```
sClone = make([]T, len(s))
copy(sClone, s)
}
```

Delete a segment of slice elements

Above has mentioned that the elements a slice are stored contiguously in memory and there are no gaps between any two adjacent elements of the slice. So when a slice element is removed,

- if the element order must be preserved, then each of the subsequent elements followed the removed elements must be moved forwards.
- if the element order doesn't need to be preserved, then we can move the last elements in the slice to the removed indexes.

In the following example, assume `from` and `to` are two legal indexes, `from` is not larger than `to`, and the `to` index is exclusive.

```
1| // way 1 (preserve element orders):
2| s = append(s[:from], s[to:]...)
3|
4| // way 2 (preserve element orders):
5| s = s[:from + copy(s[from:], s[to:])]
6|
7| // Don't preserve element orders:
8| if n := to-from; len(s)-to < n {
9|     copy(s[from:to], s[to:])
10| } else {
11|     copy(s[from:to], s[len(s)-n:])
12| }
13| s = s[:len(s)-(to-from)]
```

If the slice elements reference other values, we should reset tail elements (on the just freed-up slots) to avoid memory leaking.

```
1| // "len(s)+to-from" is the old slice length.
2| temp := s[len(s):len(s)+to-from]
3| for i := range temp {
4|     temp[i] = t0
5| }
```

As mentioned above, the `for-range` loop code block will be optimized as a `memclr` call by the standard Go compiler.

Delete one slice element

Deleting one element is similar to, and also simpler than, deleting a segment of elements.

In the following example, assume `i` the index of the element to be removed and `i` is a legal index.

```

1| // Way 1 (preserve element orders):
2| s = append(s[:i], s[i+1:]...)
3|
4| // Way 2 (preserve element orders):
5| s = s[:i] + copy(s[i:], s[i+1:])
6|
7| // There will be len(s)-i-1 elements being
8| // copied in either of the above two ways.
9|
10| // Don't preserve element orders:
11| s[i] = s[len(s)-1]
12| s = s[:len(s)-1]
```

If the slice elements contain pointers, then after the deletion action, we should reset the last element of the old slice value to avoid memory leaking:

```

1| s[len(s):len(s)+1][0] = t0
2| // or
3| s[:len(s)+1][len(s)] = t0
```

Delete slice elements conditionally

Sometimes, we may need to delete slice elements by some conditions.

```

1| // Assume T is a small-size type.
2| func DeleteElements(s []T, keep func(T) bool, clear bool) []T {
3|     //result := make([]T, 0, len(s))
4|     result := s[:0] // without allocating a new slice
5|     for _, v := range s {
6|         if keep(v) {
7|             result = append(result, v)
8|         }
9|     }
10|    if clear { // avoid memory leaking
11|        temp := s[len(result):]
12|        for i := range temp {
13|            // t0 is a zero value literal of T.
```

```

14|     temp[i] = t0
15| }
16| }
17| return result
18| }
```

Please note, if `T` is not a small-size type, then generally we should [try to](#) (§34) avoid using `T` as function parameter types and using two-iteration-variable `for`-range block form to iterate slices with element types as `T`.

Insert all elements of a slice into another slice

Assume the insertion position is a legal index `i` and `elements` is the slice whose elements are to be inserted.

```

1| // One-line implementation:
2| s = append(s[:i], append(elements, s[i:]...)...)
3|
4| // The one-line way always copies s[i:] twice and
5| // might make at most two allocations.
6| // The following verbose way always copies s[i:]
7| // once and will only make at most one allocation.
8| // However, as of Go Toolchain v1.21.n, the "make"
9| // call will clear partial of just allocated elements,
10| // which is actually unnecessary. So the verbose
11| // way is more efficient than the one-line way
12| // only for small slices now.
13|
14| if cap(s) >= len(s) + len(elements) {
15|     s = s[:len(s)+len(elements)]
16|     copy(s[i+len(elements):], s[i:])
17|     copy(s[i:], elements)
18| } else {
19|     x := make([]T, 0, len(elements)+len(s))
20|     x = append(x, s[:i]...)
21|     x = append(x, elements...)
22|     x = append(x, s[i:]...)
23|     s = x
24| }
25|
26| // Push:
27| s = append(s, elements...)
28| }
```

```
29| // Unshift (insert at the beginning):
30| s = append(elements, s...)
```

|Insert several individual elements

Inserting several individual elements is similar to inserting all elements of a slice. We can construct a slice with a slice composite literal with the elements to be inserted, then use the above ways to insert these elements.

|Special deletions and insertions: push front/back, pop front/back

Assume the pushed or popped element is `e` and slice `s` has at least one element.

```
1| // Pop front (shift):
2| s, e = s[1:], s[0]
3| // Pop back:
4| s, e = s[:len(s)-1], s[len(s)-1]
5| // Push front
6| s = append([]T{e}, s...)
7| // Push back:
8| s = append(s, e)
```

Please note, using `append` to insert elements is often inefficient, for all the elements after the insertion position need to be move backwards, and a larger space needs to be allocated to store all the elements if the free capacity is insufficient. These might be not serious problems for small slices, but for large slices, they often are. So if the number of the elements needing to be moved is large, it is best to use a linked list to do the insertions.

|More slice operations

In reality, the needs are varied. For some specific cases, it is possible none of the above ways shown in the above examples are the most efficient way. And sometimes, the above ways may not satisfy some specific requirements in practice. So, please learn and apply elastically. This may be the reason why Go doesn't support the more operations introduced above in the built-in way.

|Use Maps to Simulate Sets

Go doesn't support built-in set types. However, it is easy to use a map type to simulate a set type. In practice, we often use the map type `map[K]struct{}` to simulate a set type with element type `K`. The size of the map element type `struct{}` is zero, elements of values of such map types don't occupy memory space.

Container Related Operations Are Not Synchronized Internally

Please note that, all container operations are not synchronized internally. Without making use of any data synchronization technique, it is okay for multiple goroutines to read a container concurrently, but it is not okay for multiple goroutines to manipulate a container concurrently and at least one goroutine modifies the container. The latter case will cause data races, even make goroutines panic. We must synchronize the container operations manually. Please read the articles on [data synchronizations](#) (§36) for details.

verbose

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Strings in Go

Like many other programming languages, string is also one important kind of types in Go. This article will list all the facts of strings.

The Internal Structure of String Types

For the standard Go compiler, the internal structure of any string type is declared like:

```
1| type _string struct {
2|     elements *byte // underlying bytes
3|     len       int    // number of bytes
4| }
```

From the declaration, we know that a string is actually a byte sequence wrapper. In fact, we can really view a string as an (element-immutable) byte slice.

Note, in Go, byte is a built-in alias of type uint8.

Some Simple Facts About Strings

We have learned the following facts about strings from previous articles.

- String values can be used as constants (along with boolean and all kinds of numeric values).
- Go supports [two styles of string literals](#) (§6), the double-quote style (or interpreted literals) and the back-quote style (or raw string literals).
- The zero values of string types are blank strings, which can be represented with "" or `` in literal.
- Strings can be concatenated with + and += operators.
- String types are all comparable (by using the == and != operators). And like integer and floating-point values, two values of the same string type can also be compared with >, <, >= and <= operators. When comparing two strings, their underlying bytes will be compared, one byte by one byte. If one string is a prefix of the other one and the other one is longer, then the other one will be viewed as the larger one.

Example:

```
1| package main
2|
```

```

3| import "fmt"
4|
5| func main() {
6|     const World = "world"
7|     var hello = "hello"
8|
9|     // Concatenate strings.
10|    var helloworld = hello + " " + World
11|    helloworld += "!"
12|    fmt.Println(helloworld) // hello world!
13|
14|     // Compare strings.
15|    fmt.Println(hello == "hello") // true
16|    fmt.Println(hello > helloworld) // false
17| }
```

More facts about string types and values in Go.

- Like Java, the contents (underlying bytes) of string values are immutable. The lengths of string values also can't be modified separately. An addressable string value can only be overwritten as a whole by assigning another string value to it.
- The built-in `string` type has no methods (just like most other built-in types in Go), but we can
 - use functions provided in the [strings standard package](#) to do all kinds of string manipulations.
 - call the built-in `len` function to get the length of a string (number of bytes stored in the string).
 - use the element access syntax `aString[i]` introduced in [container element accesses](#) (§18) to get the *ith* byte value stored in `aString`. The expression `aString[i]` is not addressable. In other words, value `aString[i]` can't be modified.
 - use [the subslice syntax](#) (§18) `aString[start:end]` to get a substring of `aString`.
Here, `start` and `end` are both indexes of bytes stored in `aString`.
- For the standard Go compiler, the destination string variable and source string value in a string assignment will share the same underlying byte sequence in memory. The result of a substring expression `aString[start:end]` also shares the same underlying byte sequence with the base string `aString` in memory.

Example:

```

1| package main
2|
3| import (
```

```

4|     "fmt"
5|     "strings"
6| )
7|
8| func main() {
9|     var helloworld = "hello world!"
10|
11|     var hello = helloworld[:5] // substring
12|     // 104 is the ASCII code (and Unicode) of char 'h'.
13|     fmt.Println(hello[0])      // 104
14|     fmt.Printf("%T \n", hello[0]) // uint8
15|
16|     // hello[0] is unaddressable and immutable,
17|     // so the following two lines fail to compile.
18| /*
19|     hello[0] = 'H'           // error
20|     fmt.Println(&hello[0]) // error
21| */
22|
23|     // The next statement prints: 5 12 true
24|     fmt.Println(len(hello), len(helloworld),
25|                 strings.HasPrefix(helloworld, hello))
26| }
```

Note, if `aString` and the indexes in expressions `aString[i]` and `aString[start:end]` are all constants, then out-of-range constant indexes will make compilations fail. And please note that the evaluation results of such expressions are always non-constants ([this might or might not change since a later Go version](#) ). For example, the following program will print `4 0`.

```

1| package main
2|
3| import "fmt"
4|
5| const s = "Go101.org" // len(s) == 9
6|
7| // len(s) is a constant expression,
8| // whereas len(s[:]) is not.
9| var a byte = 1 << len(s) / 128
10| var b byte = 1 << len(s[:]) / 128
11|
12| func main() {
13|     fmt.Println(a, b) // 4 0
14| }
```

About why variables `a` and `b` are evaluated to different values, please read [the special type deduction rule in bitwise shift operator operation](#) (§8) and [which function calls are evaluated at compile time](#) (§46).

String Encoding and Unicode Code Points

Unicode standard specifies a unique value for each character in all kinds of human languages. But the basic unit in Unicode is not character, it is code point instead. For most code points, each of them corresponds to a character, but for a few characters, each of them consists of several code points.

Code points are represented as [rune values](#) (§6) in Go. In Go, `rune` is a built-in alias of type `int32`.

In applications, there are several encoding methods to represent code points, such as UTF-8 encoding and UTF-16 encoding. Nowadays, the most popularly used encoding method is UTF-8 encoding. In Go, all string constants are viewed as UTF-8 encoded. At compile time, illegal UTF-8 encoded string constants will make compilation fail. However, at run time, Go runtime can't prevent some strings from being illegally UTF-8 encoded.

For UTF-8 encoding, each code point value may be stored as one or more bytes (up to four bytes). For example, each English code point (which corresponds to one English character) is stored as one byte, however each Chinese code point (which corresponds to one Chinese character) is stored as three bytes.

String Related Conversions

In the article [constants and variables](#) (§7), we have learned that integers can be explicitly converted to strings (but not vice versa).

Here introduces two more string related conversions rules in Go:

1. a string value can be explicitly converted to a byte slice, and vice versa. A byte slice is a slice with element type's underlying type as `[]byte`.
2. a string value can be explicitly converted to a rune slice, and vice versa. A rune slice is a slice whose element type's underlying type as `[]rune`.

In a conversion from a rune slice to string, each slice element (a rune value) will be UTF-8 encoded as from one to four bytes and stored in the result string. If a slice rune element value is outside the range of valid Unicode code points, then it will be viewed as `0xFFFFD`, the code point for the

Unicode replacement character. 0xFFFFD will be UTF-8 encoded as three bytes (0xef 0xbf 0xbd).

When a string is converted to a rune slice, the bytes stored in the string will be viewed as successive UTF-8 encoding byte sequence representations of many Unicode code points. Bad UTF-8 encoding representations will be converted to a rune value 0xFFFFD.

When a string is converted to a byte slice, the result byte slice is just a deep copy of the underlying byte sequence of the string. When a byte slice is converted to a string, the underlying byte sequence of the result string is also just a deep copy of the byte slice. A memory allocation is needed to store the deep copy in each of such conversions. The reason why a deep copy is essential is slice elements are mutable but the bytes stored in strings are immutable, so a byte slice and a string can't share byte elements.

Please note, for conversions between strings and byte slices,

- illegal UTF-8 encoded bytes are allowed and will keep unchanged.
- the standard Go compiler makes some optimizations for some special cases of such conversions, so that the deep copies are not made. Such cases will be introduced below.

Conversions between byte slices and rune slices are not supported directly in Go. We can use the following ways to achieve this goal:

- use string values as a hop. This way is convenient but not very efficient, for two deep copies are needed in the process.
- use the functions in [unicode/utf8](#) standard package.
- use [the Runes function in the bytes standard package](#) to convert a []byte value to a []rune value. There is not a function in this package to convert a rune slice to byte slice.

Example:

```

1| package main
2|
3| import (
4|     "bytes"
5|     "unicode/utf8"
6| )
7|
8| func Runes2Bytes(rs []rune) []byte {
9|     n := 0
10|    for _, r := range rs {
11|        n += utf8.RuneLen(r)
12|    }

```

```

13|     n, bs := 0, make([]byte, n)
14|     for _, r := range rs {
15|         n += utf8.EncodeRune(bs[n:], r)
16|     }
17|     return bs
18| }
19|
20| func main() {
21|     s := "Color Infection is a fun game."
22|     bs := []byte(s) // string -> []byte
23|     s = string(bs) // []byte -> string
24|     rs := []rune(s) // string -> []rune
25|     s = string(rs) // []rune -> string
26|     rs = bytes.Runes(bs) // []byte -> []rune
27|     bs = Runes2Bytes(rs) // []rune -> []byte
28| }
```

Compiler Optimizations for Conversions Between Strings and Byte Slices

Above has mentioned that the underlying bytes in the conversions between strings and byte slices will be copied. The standard Go compiler makes some optimizations, which are proven to still work in Go Toolchain 1.21.n, for some special scenarios to avoid the duplicate copies. These scenarios include:

- a conversion (from string to byte slice) which follows the `range` keyword in a `for-range` loop.
- a conversion (from byte slice to string) which is used as a map key in map element retrieval indexing syntax.
- a conversion (from byte slice to string) which is used in a comparison.
- a conversion (from byte slice to string) which is used in a string concatenation, and at least one of concatenated string values is a non-blank string constant.

Example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     var str = "world"
7|     // Here, the []byte(str) conversion will
```

```

8|     // not copy the underlying bytes of str.
9|     for i, b := range []byte(str) {
10|         fmt.Println(i, ":", b)
11|     }
12|
13|     key := []byte{'k', 'e', 'y'}
14|     m := map[string]string{}
15|     // The string(key) conversion copys the bytes in key.
16|     m[string(key)] = "value"
17|     // Here, this string(key) conversion doesn't copy
18|     // the bytes in key. The optimization will be still
19|     // made, even if key is a package-level variable.
20|     fmt.Println(m[string(key)]) // value (very possible)
21| }
```

Note, the last line might not output `value` if there are data races in evaluating `string(key)`. However, such data races will never cause panics.

Another example:

```

1| package main
2|
3| import "fmt"
4| import "testing"
5|
6| var s string
7| var x = []byte{1023: 'x'}
8| var y = []byte{1023: 'y'}
9|
10| func fc() {
11|     // None of the below 4 conversions will
12|     // copy the underlying bytes of x and y.
13|     // Surely, the underlying bytes of x and y will
14|     // be still copied in the string concatenation.
15|     if string(x) != string(y) {
16|         s = (" " + string(x) + string(y))[1:]
17|     }
18| }
19|
20| func fd() {
21|     // Only the two conversions in the comparison
22|     // will not copy the underlying bytes of x and y.
23|     if string(x) != string(y) {
24|         // Please note the difference between the
25|         // following concatenation and the one in fc.
```

```

26|     s = string(x) + string(y)
27| }
28| }
29|
30| func main() {
31|     fmt.Println(testing.AllocsPerRun(1, fc)) // 1
32|     fmt.Println(testing.AllocsPerRun(1, fd)) // 3
33| }
```

for-range on Strings

The `for-range` loop control flow applies to strings. But please note, `for-range` will iterate the Unicode code points (as `rune` values), instead of bytes, in a string. Bad UTF-8 encoding representations in the string will be interpreted as `rune` value `0xFFFFD`.

Example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     s := "é    aπ圆"
7|     for i, rn := range s {
8|         fmt.Printf("%2v: 0x%04x %v \n", i, rn, string(rn))
9|     }
10|    fmt.Println(len(s))
11| }
```

The output of the above program:

```

0: 0x65 e
1: 0x301 '
3: 0x915
6: 0x94d
9: 0x937
12: 0x93f
15: 0x61 a
16: 0x3c0 π
18: 0x56e7 圆
21
```

From the output result, we can find that

1. the iteration index value may be not continuous. The reason is the index is the byte index in the ranged string and one code point may need more than one byte to represent.
2. the first character, é, is composed of two runes (3 bytes total)
3. the second character, , is composed of four runes (12 bytes total).
4. the English character, a, is composed of one rune (1 byte).
5. the character, π, is composed of one rune (2 bytes).
6. the Chinese character, 圖, is composed of one rune (3 bytes).

Then how to iterate bytes in a string? Do this:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     s := "é aπ圖"
7|     for i := 0; i < len(s); i++ {
8|         fmt.Printf("The byte at index %v: 0x%02x \n", i, s[i])
9|     }
10| }
```

Surely, we can also make use of the compiler optimization mentioned above to iterate bytes in a string. For the standard Go compiler, this way is a little more efficient than the above one.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     s := "é aπ圖"
7|     // Here, the underlying bytes of s are not copied.
8|     for i, b := range []byte(s) {
9|         fmt.Printf("The byte at index %v: 0x%02x \n", i, b)
10|    }
11| }
```

From the above several examples, we know that `len(s)` will return the number of bytes in string `s`. The time complexity of `len(s)` is $O(1)$. How to get the number of runes in a string? Using a `for-range` loop to iterate and count all runes is a way, and using the [RuneCountInString](#) function in the `unicode/utf8` standard package is another way. The efficiencies of the two ways are almost the same. The third way is to use `len([]rune(s))` to get the count of runes in string `s`. Since Go Toolchain 1.11, the standard Go compiler makes an optimization for the third way to

avoid an unnecessary deep copy so that it is as efficient as the former two ways. Please note that the time complexities of these ways are all $O(n)$.

More String Concatenation Methods

Besides using the `+` operator to concatenate strings, we can also use following ways to concatenate strings.

- The `Sprintf/Sprint/Sprintln` functions in the `fmt` standard package can be used to concatenate values of any types, including string types.
- Use the `Join` function in the `strings` standard package.
- The `Buffer` type in the `bytes` standard package (or the built-in `copy` function) can be used to build byte slices, which afterwards can be converted to string values.
- Since Go 1.10, the `Builder` type in the `strings` standard package can be used to build strings. Comparing with `bytes.Buffer` way, this way avoids making an unnecessary duplicated copy of underlying bytes for the result string.

The standard Go compiler makes optimizations for string concatenations by using the `+` operator. So generally, using `+` operator to concatenate strings is convenient and efficient if the number of the concatenated strings is known at compile time.

Sugar: Use Strings as Byte Slices

From the article [arrays, slices and maps](#) (§18), we have learned that we can use the built-in `copy` and `append` functions to copy and append slice elements. In fact, as a special case, if the first argument of a call to either of the two functions is a byte slice, then the second argument can be a string (if the call is an `append` call, then the string argument must be followed by three dots `...`). In other words, a string can be used as a byte slice for the special case.

Example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     hello := []byte("Hello ")
7|     world := "world!"
8|
9|     // The normal way:

```

```

10| // helloworld := append(hello, []byte(world)... )
11| helloworld := append(hello, world...) // sugar way
12| fmt.Println(string(helloworld))
13|
14| helloworld2 := make([]byte, len(hello) + len(world))
15| copy(helloworld2, hello)
16| // The normal way:
17| // copy(helloworld2[len(hello):], []byte(world))
18| copy(helloworld2[len(hello):], world) // sugar way
19| fmt.Println(string(helloworld2))
20| }

```

More About String Comparisons

Above has mentioned that comparing two strings is comparing their underlying bytes actually. Generally, Go compilers will make the following optimizations for string comparisons.

- For `==` and `!=` comparisons, if the lengths of the compared two strings are not equal, then the two strings must be also not equal (no needs to compare their bytes).
- If their underlying byte sequence pointers of the compared two strings are equal, then the comparison result is the same as comparing the lengths of the two strings.

So for two equal strings, the time complexity of comparing them depends on whether or not their underlying byte sequence pointers are equal. If the two are equal, then the time complexity is $O(1)$, otherwise, the time complexity is $O(n)$, where n is the length of the two strings.

As above mentioned, for the standard Go compiler, in a string value assignment, the destination string value and the source string value will share the same underlying byte sequence in memory. So the cost of comparing the two strings becomes very small.

Example:

```

1| package main
2|
3| import (
4|     "fmt"
5|     "time"
6| )
7|
8| func main() {
9|     bs := make([]byte, 1<<26)
10|    s0 := string(bs)
11|    s1 := string(bs)

```

```

12|     s2 := s1
13|
14|     // s0, s1 and s2 are three equal strings.
15|     // The underlying bytes of s0 is a copy of bs.
16|     // The underlying bytes of s1 is also a copy of bs.
17|     // The underlying bytes of s0 and s1 are two
18|     // different copies of bs.
19|     // s2 shares the same underlying bytes with s1.
20|
21|     startTime := time.Now()
22|     _ = s0 == s1
23|     duration := time.Now().Sub(startTime)
24|     fmt.Println("duration for (s0 == s1):", duration)
25|
26|     startTime = time.Now()
27|     _ = s1 == s2
28|     duration = time.Now().Sub(startTime)
29|     fmt.Println("duration for (s1 == s2):", duration)
30| }
```

Output:

```

duration for (s0 == s1): 10.462075ms
duration for (s1 == s2): 136ns
```

1ms is 1000000ns! So please try to avoid comparing two long strings if they don't share the same underlying byte sequence.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Functions in Go

[Function declarations and calls](#) (§9) have been explained before. The current article will touch more function related concepts and details in Go.

In fact, function is one kind of first-class citizen types in Go. In other words, we can use functions as values. Although Go is a static language, Go functions are very flexible. The feeling of using Go functions is much like using many dynamic languages.

There are some built-in functions in Go. These functions are documented in `builtin` and `unsafe` standard code packages. Built-in functions have several differences from custom functions. These differences will be mentioned below.

Function Signatures and Function Types

The literal of a function type is composed of the `func` keyword and a function signature literal. A function signature is composed of two type lists, one is the input parameter type list, the other is the output result type lists. Parameter and result names can appear in function type and signature literals, but the names are not important.

In practice, the `func` keyword can be presented in signature literals, or not. For this reason, we can think function type and function signature as the same concept.

Here is a literal of a function type:

```
func (a int, b string, c string) (x int, y int, z bool)
```

From the article [function declarations and calls](#) (§9), we have learned that consecutive parameters and results of the same type can be declared together. So the above literal is equivalent to

```
func (a int, b, c string) (x, y int, z bool)
```

As parameter names and result names are not important in the literals (as long as there are no duplicate non-blank names), the above ones are equivalent to the following one.

```
func (x int, y, z string) (a, b int, c bool)
```

Variable (parameter and result) names can be blank identifier `_`. The above ones are equivalent to the following one.

```
func (_ int, _, _ string) (_, _ int, _ bool)
```

The parameter names must be either all present or all absent (anonymous). The same rule is for result names. The above ones are equivalent to the following ones.

```
func (int, string, string) (int, int, bool) // the standard form
func (a int, b string, c string) (int, int, bool)
func (x int, _ string, z string) (int, int, bool)
func (int, string, string) (x int, y int, z bool)
func (int, string, string) (a int, b int, _ bool)
```

All of the above function type literals denote the same (unnamed) function type.

Each parameter list must be enclosed in a () in a literal, even if the parameter list is blank. If a result list of a function type is blank, then it can be omitted from literal of the function type. When a result list has most one result, then the result list doesn't need to be enclosed in a () if the literal of the result list contains no result names.

```
// The following three function types are identical.
func () (x int)
func () (int)
func () int

// The following two function types are identical.
func (a int, b string) ()
func (a int, b string)
```

Variadic parameters and variadic function types

The last parameter of a function can be a variadic parameter. Each function can have at most one variadic parameter. The type of a variadic parameter is always a slice type. To indicate the last parameter is variadic, just prefix three dots ... to the element type of its (slice) type in its declaration. Example:

```
func (values ...int64) (sum int64)
func (sep string, tokens ...string) string
```

A function type with variadic parameter can be called a variadic function type. A variadic function type and a non-variadic function type are absolutely not identical.

Some variadic functions examples will be shown in a below section.

Function types are incomparable types

It has been [mentioned](#) (§14) several times in Go 101 that function types are incomparable types. But like map and slice values, function values can compare with the untyped bare `nil` identifier. (Function values will be explained in the last section of the current article.)

As function types are incomparable types, they can't be used as the key types of map types.

Function Prototypes

A function prototype is composed of a function name and a function type (or signature). Its literal is composed of the `func` keyword, a function name and the literal of a function signature literal.

A function prototype literal example:

```
func Double(n int) (result int)
```

In other words, a function prototype is a function declaration without the body portion. A function declaration is composed of a function prototype and a function body.

Variadic Function Declarations and Variadic Function Calls

General function declarations and calls have been explained in [function declarations and calls](#) (§9). Here introduces how to declare and call variadic functions.

Variadic function declarations

Variadic function declarations are similar to general function declarations. The difference is that the last parameter of a variadic function must be variadic parameter. Note, the variadic parameter of a variadic function will be treated as a slice within the body of the variadic function.

```
1| // Sum and return the input numbers.
2| func Sum(values ...int64) (sum int64) {
3|     // The type of values is []int64.
4|     sum = 0
5|     for _, v := range values {
6|         sum += v
7|     }
```

```

8|     return
9| }
10|
11| // An inefficient string concatenation function.
12| func Concat(sep string, tokens ...string) string {
13|     // The type of tokens is []string.
14|     r := ""
15|     for i, t := range tokens {
16|         if i != 0 {
17|             r += sep
18|         }
19|         r += t
20|     }
21|     return r
22| }
```

From the above two variadic function declarations, we can find that if a variadic parameter is declared with type portion as `...T`, then the type of the parameter is `[]T` actually.

In fact, the `Print`, `Println` and `Printf` functions in the `fmt` standard package are all variadic functions.

```

1| func Print(a ...interface{}) (n int, err error)
2| func Printf(format string, a ...interface{}) (n int, err error)
3| func Println(a ...interface{}) (n int, err error)
```

The variadic parameter types of the three functions are all `[]interface{}`, which element type `interface{}` is an interface types. Interface types and values will be explained [interfaces in Go](#) (§23) later.

Variadic function calls

There are two manners to pass arguments to a variadic parameter of type `[]T`:

1. pass a slice value as the only argument. The slice must be assignable to values of type `[]T`, and the slice must be followed by three dots `...`. The passed slice is called as a variadic argument.
2. pass zero or more arguments which are assignable to values of type `T`. These arguments will be copied (or converted) as the elements of a new allocated slice value of type `[]T`, then the new allocated slice will be passed to the variadic parameter.

Note, the two manners can't be mixed in the same variadic function call.

An example program which uses some variadic function calls:

```

1| package main
2|
3| import "fmt"
4|
5| func Sum(values ...int64) (sum int64) {
6|     sum = 0
7|     for _, v := range values {
8|         sum += v
9|     }
10|    return
11| }
12|
13| func main() {
14|     a0 := Sum()
15|     a1 := Sum(2)
16|     a3 := Sum(2, 3, 5)
17|     // The above three lines are equivalent to
18|     // the following three respective lines.
19|     b0 := Sum([]int64{}...)
20|     b1 := Sum([]int64{2}...)
21|     b3 := Sum([]int64{2, 3, 5}...)
22|     fmt.Println(a0, a1, a3) // 0 2 10
23|     fmt.Println(b0, b1, b3) // 0 2 10
24| }
```

Another example:

```

1| package main
2|
3| import "fmt"
4|
5| func Concat(sep string, tokens ...string) (r string) {
6|     for i, t := range tokens {
7|         if i != 0 {
8|             r += sep
9|         }
10|        r += t
11|    }
12|    return
13| }
14|
15| func main() {
16|     tokens := []string{"Go", "C", "Rust"}
```

```

17|     // manner 1
18|     langsA := Concat("", "", tokens...)
19|     // manner 2
20|     langsB := Concat("", "", "Go", "C", "Rust")
21|     fmt.Println(langsA == langsB) // true
22| }
```

The following example doesn't compile, for the two variadic function call manners are mixed.

```

1| package main
2|
3| // See above examples for the full declarations
4| // of the following two functions.
5| func Sum(values ...int64) (sum int64)
6| func Concat(sep string, tokens ...string) string
7|
8| func main() {
9|     // The following two lines both fail
10|    // to compile, for the same error:
11|    // too many arguments in call.
12|    _ = Sum(2, []int64{3, 5}...)
13|    _ = Concat("", "Go", []string{"C", "Rust"}...)
14| }
```

More About Function Declarations and Calls

Functions whose names can be duplicate

Generally, the names of the functions declared in the same code package can't be duplicate. But there are two exceptions.

1. One exception is each code package can declare several functions with [the same name init and the same type func \(\)](#). (§10).
2. The other exception is multiple functions can be declared with names as the blank identifier `_`, in which cases, the declared functions can never be called.

Some function calls are evaluated at compile time

Most function calls are evaluated at run time. But calls to the functions of the `unsafe` standard package are always evaluated at compile time. Calls to some other built-in functions, such as `len`

and `cap`, [may be evaluated at either compile time or run time](#) (§46), depending on the passed arguments. The results of the function calls evaluated at compile time can be assigned to constants.

| All function arguments are passed by copy

Let's repeat it again, like all value assignments in Go, all function arguments are passed by copy in Go. When a value is copied, [only its direct part is copied](#) (§17) (a.k.a., a shallow copy).

| Function declarations without bodies

We can implement a function in [Go assembly](#). Go assembly source files are stored in `*.a` files. A function implemented in Go assembly is still needed to be declared in a `*.go` file, but the only the prototype of the function is needed to be present. The body portion of the declaration of the function must be omitted in the `*.go` file.

| Some functions with results are not required to return

If a function has return results, then the last statement in its declaration body must be a [terminating statement](#). Other than `return` terminating statement, there are some other kinds of terminating statements. So a function body is not required to contain a return statement. For example,

```

1| func fa() int {
2|     a:
3|     goto a
4|
5|
6| func fb() bool {
7|     for {}
8|

```

| The results of calls to custom function can be discarded, not true for calls to some built-in functions

The return results of a custom function call can be all discarded together. However, the return results of calls to built-in functions, except `recover` and `copy`, can't be discarded, though they can be ignored by assigning them to some blank identifiers. Function calls whose results can't be discarded can't be used as deferred function calls or goroutine calls.

Use function calls as expressions

A call to a function with single return result can always be used as a single value. For example, it can be nested in another function call as an argument, and can also be used as a single value to appear in any other expressions and statements.

If the return results of a call to a multi-result function are not discarded, then the call can only be used as a multi-value expression in two scenarios.

1. The call can be used in an assignment as source values. But the call can't mix with other source values in the assignment.
2. The call can be nested in another function call as arguments. But the call can't mix with other arguments.

An example:

```

1| package main
2|
3| func HalfAndNegative(n int) (int, int) {
4|     return n/2, -n
5| }
6|
7| func AddSub(a, b int) (int, int) {
8|     return a+b, a-b
9| }
10|
11| func Dummy(values ...int) {}
12|
13| func main() {
14|     // These lines compile okay.
15|     AddSub(HalfAndNegative(6))
16|     AddSub(AddSub(AddSub(7, 5)))
17|     AddSub(AddSub(HalfAndNegative(6)))
18|     Dummy(HalfAndNegative(6))
19|     _, _ = AddSub(7, 5)
20|
21|     // The following lines fail to compile.
22|     /*
23|     _, _, _ = 6, AddSub(7, 5)
24|     Dummy(AddSub(7, 5), 9)
25|     Dummy(AddSub(7, 5), HalfAndNegative(6))
26|     */
27| }
```

Note, for the standard Go compiler, [some built-in functions break the universality](#) (§49) of the just described rules above.

Function Values

As mentioned above, function types are one kind of types in Go. A value of a function type is called a function value. The zero values of function types are represented with the predeclared `nil`.

When we declare a custom function, we also declared an immutable function value actually. The function value is identified by the function name. The type of the function value is represented as the literal by omitting the function name from the function prototype literal.

Note, built-in functions can't be used as values. `init` functions also can't be used as values.

Any function value can be invoked just like a declared function. It is fatal error to call a nil function to start a new goroutine. The fatal error is not recoverable and will make the whole program crash. For other situations, calls to nil function values will produce recoverable panics, including deferred function calls.

From the article [value parts](#) (§17), we know that non-nil function values are multi-part values. After one function value is assigned to another, the two functions share the same underlying part(s). In other words, the two functions represent the same internal function object. The effects of invoking two functions are the same.

An example:

```

1| package main
2|
3| import "fmt"
4|
5| func Double(n int) int {
6|     return n + n
7| }
8|
9| func Apply(n int, f func(int) int) int {
10|    return f(n) // the type of f is "func(int) int"
11| }
12|
13| func main() {
14|     fmt.Printf("%T\n", Double) // func(int) int
15|     // Double = nil // error: Double is immutable.
16|
17|     var f func(n int) int // default value is nil.

```

```

18|     f = Double
19|     g := Apply // let compile deduce the type of g
20|     fmt.Printf("%T\n", g) // func(int, func(int) int) int
21|
22|     fmt.Println(f(9))      // 18
23|     fmt.Println(g(6, Double)) // 12
24|     fmt.Println(Apply(6, f)) // 12
25| }
```

In the above example, `g(6, Double)` and `Apply(6, f)` are equivalent.

In practice, we often assign anonymous functions to function variables, so that we can call the anonymous functions multiple times.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     // This function returns a function (a closure).
7|     isMultipleOfX := func (x int) func(int) bool {
8|         return func(n int) bool {
9|             return n%x == 0
10|        }
11|    }
12|
13|    var isMultipleOf3 = isMultipleOfX(3)
14|    var isMultipleOf5 = isMultipleOfX(5)
15|    fmt.Println(isMultipleOf3(6)) // true
16|    fmt.Println(isMultipleOf3(8)) // false
17|    fmt.Println(isMultipleOf5(10)) // true
18|    fmt.Println(isMultipleOf5(12)) // false
19|
20|    isMultipleOf15 := func(n int) bool {
21|        return isMultipleOf3(n) && isMultipleOf5(n)
22|    }
23|    fmt.Println(isMultipleOf15(32)) // false
24|    fmt.Println(isMultipleOf15(60)) // true
25| }
```

All functions in Go can be viewed as closures. This is why user experiences of all kinds of Go functions are so uniform and why Go functions are as flexible as those in dynamic languages.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Channels in Go

Channel is an important built-in feature in Go. It is one of the features that makes Go unique. Along with another unique feature, [goroutine](#) (§13), channel makes concurrent programming convenient, fun and lowers the difficulties of concurrent programming.

Channel mainly acts as a [concurrency synchronization](#) (§13) technique. This article will list all the channel related concepts, syntax and rules. To understand channels better, the internal structure of channels and some implementation details by the standard Go compiler/runtime are also simply described.

The information in this article may be slightly challenging for new gophers. Some parts of this article may need to be read several times to be fully understood.

Channel Introduction

One suggestion (made by *Rob Pike*) for concurrent programming is **don't (let computations) communicate by sharing memory, (let them) share memory by communicating (through channels)**. (We can view each computation as a goroutine in Go programming.)

Communicating by sharing memory and sharing memory by communicating are two programming manners in concurrent programming. When goroutines communicate by sharing memory, we use traditional concurrency synchronization techniques, such as mutex locks, to protect the shared memory to prevent data races. We can use channels to implement sharing memory by communicating.

Go provides a unique concurrency synchronization technique, channel. Channels make goroutines share memory by communicating. We can view a channel as an internal FIFO (first in, first out) queue within a program. Some goroutines send values to the queue (the channel) and some other goroutines receive values from the queue.

Along with transferring values (through channels), the ownership of some values may also be transferred between goroutines. When a goroutine sends a value to a channel, we can view the goroutine releases the ownership of some values (which could be accessed through the sent value). When a goroutine receives a value from a channel, we can view the goroutine acquires the ownership of some values (which could be accessed through the received value).

Surely, there may also not be any ownership transferred along with channel communications.

The values (whose ownerships are transferred) are often referenced (but are not required to be referenced) by the transferred value. Please note, here, when we talk about ownership, we mean the ownership from the logic view. Go channels can help programmers write data races free code easily, but Go channels can't prevent programmers from writing bad concurrent code from the syntax level.

Although Go also supports traditional concurrency synchronization techniques, only channel is first-class citizen in Go. Channel is one kind of types in Go, so we can use channels without importing any packages. On the other hand, those traditional concurrency synchronization techniques are provided in the `sync` and `sync/atomic` standard packages.

Honestly, each concurrency synchronization technique has its own best use scenarios. But channel has [a wider application range and has more variety in using](#) (§37). One problem of channels is, the experience of programming with channels is so enjoyable and fun that programmers often even prefer to use channels for the scenarios which channels are not best for.

Channel Types and Values

Like array, slice and map, each channel type has an element type. A channel can only transfer values of the element type of the channel.

Channel types can be bi-directional or single-directional. Assume `T` is an arbitrary type,

- `chan T` denotes a bidirectional channel type. Compilers allow both receiving values from and sending values to bidirectional channels.
- `chan<- T` denotes a send-only channel type. Compilers don't allow receiving values from send-only channels.
- `<-chan T` denotes a receive-only channel type. Compilers don't allow sending values to receive-only channels.

`T` is called the element type of these channel types.

Values of bidirectional channel type `chan T` can be implicitly converted to both send-only type `chan<- T` and receive-only type `<-chan T`, but not vice versa (even if explicitly). Values of send-only type `chan<- T` can't be converted to receive-only type `<-chan T`, and vice versa. Note that the `<-` signs in channel type literals are modifiers.

Each channel value has a capacity, which will be explained in the section after next. A channel value with a zero capacity is called unbuffered channel and a channel value with a non-zero capacity is called buffered channel.

The zero values of channel types are represented with the predeclared identifier `nil`. A non-nil channel value must be created by using the built-in `make` function. For example, `make(chan int, 10)` will create a channel whose element type is `int`. The second argument of the `make` function call specifies the capacity of the new created channel. The second parameter is optional and its default value is zero.

Channel Value Comparisons

All channel types are comparable types.

From the article [value parts](#) (§17), we know that non-nil channel values are multi-part values. If one channel value is assigned to another, the two channels share the same underlying part(s). In other words, those two channels represent the same internal channel object. The result of comparing them is `true`.

Channel Operations

There are five channel specified operations. Assume the channel is `ch`, their syntax and function calls of these operations are listed here.

1. Close the channel by using the following function call

```
close(ch)
```

where `close` is a built-in function. The argument of a `close` function call must be a channel value, and the channel `ch` must not be a receive-only channel.

2. Send a value, `v`, to the channel by using the following syntax

```
ch <- v
```

where `v` must be a value which is assignable to the element type of channel `ch`, and the channel `ch` must not be a receive-only channel. Note that here `<-` is a channel-send operator.

3. Receive a value from the channel by using the following syntax

```
<- ch
```

A channel receive operation always returns at least one result, which is a value of the element type of the channel, and the channel `ch` must not be a send-only channel. Note that here `<-` is a channel-receive operator. Yes, its representation is the same as a channel-send operator.

For most scenarios, a channel receive operation is viewed as a single-value expression. However, when a channel operation is used as the only source value expression in an assignment, it can result a second optional untyped boolean value and become a multi-value expression. The untyped boolean value indicates whether or not the first result is sent before the channel is closed. (Below we will learn that we can receive unlimited number of values from a closed channel.)

Two channel receive operations which are used as source values in assignments:

```
v = <-ch  
v, sentBeforeClosed = <-ch
```

4. Query the value buffer capacity of the channel by using the following function call

```
cap(ch)
```

where `cap` is a built-in function which has ever been introduced in [containers in Go \(§18\)](#).

The return result of a `cap` function call is an `int` value.

5. Query the current number of values in the value buffer (or the length) of the channel by using the following function call

```
len(ch)
```

where `len` is a built-in function which also has ever been introduced before. The return value of a `len` function call is an `int` value. The result length is the number of elements which have already been sent successfully to the queried channel but haven't been received (taken out) yet.

Most basic operations in Go are not synchronized. In other words, they are not concurrency-safe. These operations include value assignments, argument passing and container element manipulations, etc. However, all the just introduced channel operations are already synchronized, so no further synchronizations are needed to safely perform these operations.

Like most other operations in Go, channel value assignments are not synchronized. Similarly, assigning the received value to another value is also not synchronized, though any channel receive operation is synchronized.

If the queried channel is a nil channel, both of the built-in `cap` and `len` functions return zero. The two query operations are so simple that they will not get further explanations later. In fact, the two operations are seldom used in practice.

Channel send, receive and close operations will be explained in detail in the next section.

Detailed Explanations for Channel Operations

To make the explanations for channel operations simple and clear, in the remaining of this article, channels will be classified into three categories:

1. nil channels.
2. non-nil but closed channels.
3. not-closed non-nil channels.

The following table simply summarizes the behaviors for all kinds of operations applying on nil, closed and not-closed non-nil channels.

Operation	A Nil Channel	A Closed Channel	A Not-Closed Non-Nil Channel
Close	panic	panic	succeed to close ^(C)
Send Value To	block for ever	panic	block or succeed to send ^(B)
Receive Value From	block for ever	never block ^(D)	block or succeed to receive ^(A)

For the five cases shown without superscripts, the behaviors are very clear.

- Closing a nil or an already closed channel produces a panic in the current goroutine.
- Sending a value to a closed channel also produces a panic in the current goroutine.
- Sending a value to or receiving a value from a nil channel makes the current goroutine enter and stay in blocking state for ever.

The following will make more explanations for the four cases shown with superscripts (A, B, C and D).

To better understand channel types and values, and to make some explanations easier, looking in the raw internal structures of internal channel objects is very helpful.

We can think of each channel consisting of three queues (all can be viewed as FIFO queues) internally:

1. the receiving goroutine queue (generally FIFO). The queue is a linked list without size limitation. Goroutines in this queue are all in blocking state and waiting to receive values from that channel.
2. the sending goroutine queue (generally FIFO). The queue is also a linked list without size limitation. Goroutines in this queue are all in blocking state and waiting to send values to that channel. The value (or the address of the value, depending on compiler implementation) each goroutine is trying to send is also stored in the queue along with that goroutine.
3. the value buffer queue (absolutely FIFO). This is a circular queue. Its size is equal to the capacity of the channel. The types of the values stored in this buffer queue are all the element

type of that channel. If the current number of values stored in the value buffer queue of the channel reaches the capacity of the channel, the channel is called in full status. If no values are stored in the value buffer queue of the channel currently, the channel is called in empty status. For a zero-capacity (unbuffered) channel, it is always in both full and empty status.

Each channel internally holds a mutex lock which is used to avoid data races in all kinds of operations.

Channel operation case A: when a goroutine R tries to receive a value from a not-closed non-nil channel, the goroutine R will acquire the lock associated with the channel firstly, then do the following steps until one condition is satisfied.

1. If the value buffer queue of the channel is not empty, in which case the receiving goroutine queue of the channel must be empty, the goroutine R will receive (by shifting) a value from the value buffer queue. If the sending goroutine queue of the channel is also not empty, a sending goroutine will be shifted out of the sending goroutine queue and resumed to running state again. The value that the just shifted sending goroutine is trying to send will be pushed into the value buffer queue of the channel. The receiving goroutine R continues running. For this scenario, the channel receive operation is called a **non-blocking operation**.
2. Otherwise (the value buffer queue of the channel is empty), if the sending goroutine queue of the channel is not empty, in which case the channel must be an unbuffered channel, the receiving goroutine R will shift a sending goroutine from the sending goroutine queue of the channel and receive the value that the just shifted sending goroutine is trying to send. The just shifted sending goroutine will get unblocked and resumed to running state again. The receiving goroutine R continues running. For this scenario, the channel receive operation is called a **non-blocking operation**.
3. If value buffer queue and the sending goroutine queue of the channel are both empty, the goroutine R will be pushed into the receiving goroutine queue of the channel and enter (and stay in) blocking state. It may be resumed to running state when another goroutine sends a value to the channel later. For this scenario, the channel receive operation is called a **blocking operation**.

Channel rule case B: when a goroutine S tries to send a value to a not-closed non-nil channel, the goroutine S will acquire the lock associated with the channel firstly, then do the following steps until one step condition is satisfied.

1. If the receiving goroutine queue of the channel is not empty, in which case the value buffer queue of the channel must be empty, the sending goroutine S will shift a receiving goroutine from the receiving goroutine queue of the channel and send the value to the just shifted receiving goroutine. The just shifted receiving goroutine will get unblocked and resumed to

running state again. The sending goroutine S continues running. For this scenario, the channel send operation is called a **non-blocking operation**.

2. Otherwise (the receiving goroutine queue is empty), if the value buffer queue of the channel is not full, in which case the sending goroutine queue must be also empty, the value the sending goroutine S trying to send will be pushed into the value buffer queue, and the sending goroutine S continues running. For this scenario, the channel send operation is called a **non-blocking operation**.
3. If the receiving goroutine queue is empty and the value buffer queue of the channel is already full, the sending goroutine S will be pushed into the sending goroutine queue of the channel and enter (and stay in) blocking state. It may be resumed to running state when another goroutine receives a value from the channel later. For this scenario, the channel send operation is called a **blocking operation**.

Above has mentioned, once a non-nil channel is closed, sending a value to the channel will produce a runtime panic in the current goroutine. Note, sending data to a closed channel is viewed as a **non-blocking operation**.

Channel operation case C: when a goroutine tries to close a not-closed non-nil channel, once the goroutine has acquired the lock of the channel, both of the following two steps will be performed by the following order.

1. If the receiving goroutine queue of the channel is not empty, in which case the value buffer of the channel must be empty, all the goroutines in the receiving goroutine queue of the channel will be shifted one by one, each of them will receive a zero value of the element type of the channel and be resumed to running state.
2. If the sending goroutine queue of the channel is not empty, all the goroutines in the sending goroutine queue of the channel will be shifted one by one and each of them will produce a panic for sending on a closed channel. This is the reason why we should avoid concurrent send and close operations on the same channel. In fact, when [the go command's data race detector option ↗ \(-race\)](#) is enabled, concurrent send and close operation cases might be detected at run time and a runtime panic will be thrown.

Note: after a channel is closed, the values which have been already pushed into the value buffer of the channel are still there. Please read the closely following explanations for case D for details.

Channel operation case D: after a non-nil channel is closed, channel receive operations on the channel will never block. The values in the value buffer of the channel can still be received. The accompanying second optional bool return values are still `true`. Once all the values in the value buffer are taken out and received, infinite zero values of the element type of the channel will be received by any of the following receive operations on the channel. As mentioned above, the optional second return result of a channel receive operation is an untyped boolean value which

indicates whether or not the first result (the received value) is sent before the channel is closed. If the second return result is `false`, then the first return result (the received value) must be a zero value of the element type of the channel.

Knowing what are blocking and non-blocking channel send or receive operations is important to understand the mechanism of `select` control flow blocks which will be introduced in a later section.

In the above explanations, if a goroutine is shifted out of a queue (either the sending or the receiving goroutine queue) of a channel, and the goroutine was blocked for being pushed into the queue at a [select control flow code block](#), then the goroutine will be resumed to running state at step 9 of the [select control flow code block execution](#). It may be dequeued from the corresponding goroutine queue of several channels involved in the `select` control flow code block.

According to the explanations listed above, we can get some facts about the internal queues of a channel.

- If the channel is closed, both its sending goroutine queue and receiving goroutine queue must be empty, but its value buffer queue may not be empty.
- At any time, if the value buffer is not empty, then its receiving goroutine queue must be empty.
- At any time, if the value buffer is not full, then its sending goroutine queue must be empty.
- If the channel is buffered, then at any time, at least one of the channel's goroutine queues must be empty (sending, receiving, or both).
- If the channel is unbuffered, most of the time one of its sending goroutine queue and the receiving goroutine queue must be empty, with one exception. The exception is that a goroutine may be pushed into both of the two queues when executing a [select control flow code block](#).

Some Channel Use Examples

Now that you've read the above section, let's view some examples which use channels to enhance your understanding.

A simple request/response example. The two goroutines in this example talk to each other through an unbuffered channel.

```
1| package main
2|
3| import (
4|     "fmt"
```

```

5|     "time"
6| )
7|
8| func main() {
9|     c := make(chan int) // an unbuffered channel
10|    go func(ch chan<- int, x int) {
11|        time.Sleep(time.Second)
12|        // <-ch    // fails to compile
13|        // Send the value and block until the result is received.
14|        ch <- x*x // 9 is sent
15|    }(c, 3)
16|    done := make(chan struct{})
17|    go func(ch <-chan int) {
18|        // Block until 9 is received.
19|        n := <-ch
20|        fmt.Println(n) // 9
21|        // ch <- 123 // fails to compile
22|        time.Sleep(time.Second)
23|        done <- struct{}{}
24|    }(c)
25|    // Block here until a value is received by
26|    // the channel "done".
27|    <-done
28|    fmt.Println("bye")
29| }
```

The output:

```

9
bye
```

A demo of using a buffered channel. This program is not a concurrent one, it just shows how to use buffered channels.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     c := make(chan int, 2) // a buffered channel
7|     c <- 3
8|     c <- 5
9|     close(c)
10|    fmt.Println(len(c), cap(c)) // 2 2
11|    x, ok := <-c
```

```

12|     fmt.Println(x, ok) // 3 true
13|     fmt.Println(len(c), cap(c)) // 1 2
14|     x, ok = <-c
15|     fmt.Println(x, ok) // 5 true
16|     fmt.Println(len(c), cap(c)) // 0 2
17|     x, ok = <-c
18|     fmt.Println(x, ok) // 0 false
19|     x, ok = <-c
20|     fmt.Println(x, ok) // 0 false
21|     fmt.Println(len(c), cap(c)) // 0 2
22|     close(c) // panic!
23| // The send will also panic if the above
24| // close call is removed.
25|     c <- 7
26|

```

A never-ending football game.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "time"
6| )
7|
8| func main() {
9|     var ball = make(chan string)
10|    kickBall := func(playerName string) {
11|        for {
12|            fmt.Println(<-ball, "kicked the ball.")
13|            time.Sleep(time.Second)
14|            ball <- playerName
15|        }
16|    }
17|    go kickBall("John")
18|    go kickBall("Alice")
19|    go kickBall("Bob")
20|    go kickBall("Emily")
21|    ball <- "referee" // kick off
22|    var c chan bool // nil
23|    <-c             // blocking here for ever
24|

```

Please read [channel use cases](#) (§37) for more channel use examples.

Channel Element Values Are Transferred by Copy

When a value is transferred from one goroutine to another goroutine, the value will be copied at least one time. If the transferred value ever stayed in the value buffer of a channel, then two copies will happen in the transfer process. One copy happens when the value is copied from the sender goroutine into the value buffer, the other happens when the value is copied from the value buffer to the receiver goroutine. Like value assignments and function argument passing, when a value is transferred, [only its direct part is copied](#) (§17).

For the standard Go compiler, the size of channel element types must be smaller than 65536. However, generally, we shouldn't create channels with large-size element types, to avoid too large copy cost in the process of transferring values between goroutines. So if the passed value size is too large, it is best to use a pointer element type instead, to avoid a large value copy cost.

About Channel and Goroutine Garbage Collections

Note, a channel is referenced by all the goroutines in either the sending or the receiving goroutine queue of the channel, so if neither of the queues of the channel is empty, the channel cannot be garbage collected. On the other hand, if a goroutine is blocked and stays in either the sending or the receiving goroutine queue of a channel, then the goroutine also cannot be garbage collected, even if the channel is referenced only by this goroutine. In fact, a goroutine can only be garbage collected when it has already exited.

Channel Send and Receive Operations Are Simple Statements

Channel send operations and receive operations are [simple statements](#) (§11). A channel receive operation can be always used as a single-value expression. Simple statements and expressions can be used at certain portions of [basic control flow blocks](#) (§12).

An example in which channel send and receive operations appear as simple statements in two `for` control flow blocks.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "time"
6| )

```

```

7|
8| func main() {
9|     fibonacci := func() chan uint64 {
10|         c := make(chan uint64)
11|         go func() {
12|             var x, y uint64 = 0, 1
13|             for ; y < (1 << 63); c <- y { // here
14|                 x, y = y, x+y
15|             }
16|             close(c)
17|         }()
18|         return c
19|     }
20|     c := fibonacci()
21|     for x, ok := <-c; ok; x, ok = <-c { // here
22|         time.Sleep(time.Second)
23|         fmt.Println(x)
24|     }
25| }
```

for-range on Channels

The `for-range` control flow code block applies to channels. The loop will try to iteratively receive the values sent to a channel, until the channel is closed and its value buffer queue becomes blank. With `for-range` syntax on arrays, slices and maps, multiple iteration variables are allowed. However, for `for-range` blocks applied to channels, you can use at most one iteration variable, which is used to store the received values.

```

1| for v := range aChannel {
2|     // use v
3| }
```

is equivalent to

```

1| for {
2|     v, ok = <-aChannel
3|     if !ok {
4|         break
5|     }
6|     // use v
7| }
```

Surely, here the `aChannel` value must not be a send-only channel. If it is a nil channel, the loop will block there for ever.

For example, the second `for` loop block in the example shown in the last section can be simplified to

```
1| for x := range c {
2|     time.Sleep(time.Second)
3|     fmt.Println(x)
4| }
```

select-case Control Flow Code Blocks

There is a `select-case` code block syntax which is specially designed for channels. The syntax is much like the `switch-case` block syntax. For example, there can be multiple `case` branches and at most one `default` branch in the `select-case` code block. But there are also some obvious differences between them.

- No expressions and statements are allowed to follow the `select` keyword (before `{}`).
- No `fallthrough` statements are allowed to be used in `case` branches.
- Each statement following a `case` keyword in a `select-case` code block must be either a channel receive operation or a channel send operation statement. A channel receive operation can appear as the source value of a simple assignment statement. Later, a channel operation following a `case` keyword will be called a `case` operation.
- If there are one or more non-blocking `case` operations, Go runtime will **randomly select one of these non-blocking operations to execute**, then continue to execute the corresponding `case` branch.
- If all the `case` operations in a `select-case` code block are blocking operations, the `default` branch will be selected to execute if the `default` branch is present. If the `default` branch is absent, the current goroutine will be pushed into the corresponding sending goroutine queue or receiving goroutine queue of every channel involved in all `case` operations, then enter blocking state.

By the rules, a `select-case` code block without any branches, `select{}`, will make the current goroutine stay in blocking state forever.

The following program will enter the `default` branch for sure.

```
1| package main
2|
```

```

3| import "fmt"
4|
5| func main() {
6|     var c chan struct{} // nil
7|     select {
8|         case <-c:           // blocking operation
9|         case c <- struct{}{}: // blocking operation
10|      default:
11|          fmt.Println("Go here.")
12|     }
13| }
```

An example showing how to use try-send and try-receive:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     c := make(chan string, 2)
7|     trySend := func(v string) {
8|         select {
9|             case c <- v:
10|                 default: // go here if c is full.
11|             }
12|     }
13|     tryReceive := func() string {
14|         select {
15|             case v := <-c: return v
16|             default: return "-" // go here if c is empty
17|         }
18|     }
19|     trySend("Hello!") // succeed to send
20|     trySend("Hi!")   // succeed to send
21|     // Fail to send, but will not block.
22|     trySend("Bye!")
23|     // The following two lines will
24|     // both succeed to receive.
25|     fmt.Println(tryReceive()) // Hello!
26|     fmt.Println(tryReceive()) // Hi!
27|     // The following line fails to receive.
28|     fmt.Println(tryReceive()) // -
29| }
```

The following example has 50% possibility to panic. Both of the two case operations are non-blocking in this example.

```

1| package main
2|
3| func main() {
4|     c := make(chan struct{})
5|     close(c)
6|     select {
7|         case c <- struct{}{}:
8|             // Panic if the first case is selected.
9|         case <-c:
10|     }
11| }
```

The Implementation of the Select Mechanism

The select mechanism in Go is an important and unique feature. Here the steps of [the select mechanism implementation by the official Go runtime](#) are listed.

There are several steps to execute a `select`-case block:

1. evaluate all involved channel expressions and value expressions to be potentially sent in case operations, from top to bottom and left to right. Destination values for receive operations (as source values) in assignments needn't be evaluated at this time.
2. randomize the branch orders for polling in step 5. The `default` branch is always put at the last position in the result order. Channels may be duplicate in the `case` operations.
3. sort all involved channels in the `case` operations to avoid deadlock (with other goroutines) in the next step. No duplicate channels stay in the first N channels of the sorted result, where N is the number of involved channels in the `case` operations. Below, the ***channel lock order*** is a concept for the first N channels in the sorted result.
4. lock (a.k.a., acquire the locks of) all involved channels by the channel lock order produced in last step.
5. poll each branch in the select block by the randomized order produced in step 2:
 1. if this is a `case` branch and the corresponding channel operation is a send-value-to-closed-channel operation, unlock all channels by the inverse channel lock order and make the current goroutine panic. Go to step 12.
 2. if this is a `case` branch and the corresponding channel operation is non-blocking, perform the channel operation and unlock all channels by the inverse channel lock order,

then execute the corresponding case branch body. The channel operation may wake up another goroutine in blocking state. Go to step 12.

3. if this is the `default` branch, then unlock all channels by the inverse channel lock order and execute the `default` branch body. Go to step 12.

(Up to here, the `default` branch is absent and all `case` operations are blocking operations.)

6. push (enqueue) the current goroutine (along with the information of the corresponding case branch) into the receiving or sending goroutine queue of the involved channel in each `case` operation. The current goroutine may be pushed into the queues of a channel for multiple times, for the involved channels in multiple cases may be the same one.
7. make the current goroutine enter blocking state and unlock all channels by the inverse channel lock order.
8. wait in blocking state until other channel operations wake up the current goroutine, ...
9. the current goroutine is waken up by another channel operation in another goroutine. The other operation may be a channel close operation or a channel send/receive operation. If it is a channel send/receive operation, there must be a `case` channel receive/send operation (in the current being explained `select-case` block) cooperating with it (by transferring a value). In the cooperation, the current goroutine will be dequeued from the receiving/sending goroutine queue of the channel.
10. lock all involved channels by the channel lock order.
11. dequeue the current goroutine from the receiving goroutine queue or sending goroutine queue of the involved channel in each `case` operation,
 1. if the current goroutine is waken up by a channel close operation, go to step 5.
 2. if the current goroutine is waken up by a channel send/receive operation, the corresponding case branch of the cooperating receive/send operation has already been found in the dequeuing process, so just unlock all channels by the inverse channel lock order and execute the corresponding case branch.
12. done.

From the implementation, we know that

- a goroutine may stay in the sending goroutine queues and the receiving goroutine queues of multiple channels at the same time. It can even stay in the sending goroutine queue and the receiving goroutine queue of the same channel at the same time.
- when a goroutine currently being blocked at a `select-case` code block gets resumed later, it will be removed from all the sending goroutine queues and the receiving goroutine queues of all channels involved in the channel operations following `case` keywords in the `select-case` code block.

More

We can find more channel use cases in [this article](#) (§37).

Although channels can help us write [correct concurrent code easily](#) (§38), like other data synchronization techniques, channels will not prevent us from [writing improper concurrent code](#) (§42).

Channel may be not always the best solution for all use cases for data synchronizations. Please read [this article](#) (§39) and [this article](#) (§40) for more synchronization techniques in Go.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Methods in Go

Go supports some object-oriented programming features. Method is one of these features. This article will introduce method-related concepts in Go.

Method Declarations

In Go, we can (explicitly) declare a method for type `T` and `*T`, where `T` must satisfy 4 conditions:

1. `T` must be a [defined type](#) (§14);
2. `T` must be defined in the same package as the method declaration;
3. `T` must not be a pointer type;
4. `T` must not be an interface type. Interface types will be explained in [the next article](#) (§23).

Type `T` and `*T` are called the receiver type of the respective methods declared for them. Type `T` is called the receiver base types of all methods declared for both type `T` and `*T`.

Note, we can also declare methods for [type aliases](#) (§14) of the `T` and `*T` types specified above. The effect is the same as declaring methods for the `T` and `*T` types themselves.

If a method is declared for a type, we can say the type has (or owns) the method.

From the above listed conditions, we will get the conclusions that we can never (explicitly) declare methods for:

- predeclared types, such as `int` and `string`, for we can't declare methods in the `builtin` standard package.
- interface types. But an interface type can own methods. Please read [the next article](#) (§23) for details.
- [unnamed types](#) (§14) except the pointer types with the form `*T` which are described above.

A method declaration is similar to a function declaration, but it has an extra parameter declaration part. The extra parameter part can contain one and only one parameter of the receiver type of the method. The only one parameter is called a receiver parameter of the method declaration. The receiver parameter must be enclosed in a `()` and declared between the `func` keyword and the method name.

Here are some method declaration examples:

```

1| // Age and int are two distinct types. We
2| // can't declare methods for int and *int,
3| // but can for Age and *Age.
4| type Age int
5| func (age Age) LargerThan(a Age) bool {
6|     return age > a
7| }
8| func (age *Age) Increase() {
9|     *age++
10| }
11|
12| // Receiver of custom defined function type.
13| type FilterFunc func(in int) bool
14| func (ff FilterFunc) Filte(in int) bool {
15|     return ff(in)
16| }
17|
18| // Receiver of custom defined map type.
19| type StringSet map[string]struct{}
20| func (ss StringSet) Has(key string) bool {
21|     _, present := ss[key]
22|     return present
23| }
24| func (ss StringSet) Add(key string) {
25|     ss[key] = struct{}{}
26| }
27| func (ss StringSet) Remove(key string) {
28|     delete(ss, key)
29| }
30|
31| // Receiver of custom defined struct type.
32| type Book struct {
33|     pages int
34| }
35|
36| func (b Book) Pages() int {
37|     return b.pages
38| }
39|
40| func (b *Book) SetPages(pages int) {
41|     b.pages = pages
42| }

```

From the above examples, we know that the receiver base types not only can be struct types, but also can be other kinds of types, such as basic types and container types, as long as the receiver base

types satisfy the 4 conditions listed above.

In some other programming languages, the receiver parameter names are always the implicit `this`, which is not a recommended identifier for receiver parameter names in Go.

The receiver of type `*T` is called ***pointer receiver***, non-pointer receivers are called ***value receivers***. Personally, I don't recommend to view the terminology ***pointer*** as an opposite of the terminology ***value***, because pointer values are just special values. But, I am not against using the pointer receiver and value receiver terminologies here. The reason will be explained below.

Method names can be the blank identifier `_`. A type can have multiple methods with the blank identifier as name. But such methods can never be called. Only exported methods can be called from other packages. Method calls will be introduced in a later section.

Each Method Corresponds to an Implicit Function

For each method declaration, compiler will declare a corresponding implicit function for it. For the last two methods declared for type `Book` and type `*Book` in the last example in the last section, two following functions are implicitly declared by compiler:

```

1| func Book.Pages(b Book) int {
2|     // The body is the same as the Pages method.
3|     return b.pages
4|
5|
6| func (*Book).SetPages(b *Book, pages int) {
7|     // The body is the same as the SetPages method.
8|     b.pages = pages
9|

```

In each of the two implicit function declarations, the receiver parameter is removed from its corresponding method declaration and inserted into the normal parameter list as the first one. The function bodies of the two implicitly declared functions is the same as their corresponding method explicit bodies.

The implicit function names, `Book.Pages` and `(*Book).SetPages`, are both of the form `TypeDenotation.MethodName`. As identifiers in Go can't contain the period special characters, the two implicit function names are not legal identifiers, so the two functions can't be declared explicitly. They can only be declared by compilers implicitly, but they can be called in user code:

```

1| package main
2|

```

```

3| import "fmt"
4|
5| type Book struct {
6|     pages int
7| }
8|
9| func (b Book) Pages() int {
10|     return b.pages
11| }
12|
13| func (b *Book) SetPages(pages int) {
14|     b.pages = pages
15| }
16|
17| func main() {
18|     var book Book
19|     // Call the two implicit declared functions.
20|     (*Book).SetPages(&book, 123)
21|     fmt.Println(book.Pages()) // 123
22| }
```

In fact, compilers not only declare the two implicit functions, they also rewrite the two corresponding explicit declared methods to let the two methods call the two implicit functions in the method bodies (at least, we can think this happens), just like the following code shows:

```

1| func (b Book) Pages() int {
2|     return Book.Pages(b)
3| }
4|
5| func (b *Book) SetPages(pages int) {
6|     (*Book).SetPages(b, pages)
7| }
```

Implicit Methods With Pointer Receivers

For each method declared for value receiver type `T`, a corresponding method with the same name will be implicitly declared by compiler for type `*T`. By the example above, the `Pages` method is declared for type `Book`, so a method with the same name `Pages` is implicitly declared for type `*Book`:

```

1| // Note: this is not a legal Go syntax.
2| // It is shown here just for explanation purpose.
3| // It indicates that the expression (&aBook).Pages
```

```

4| // is evaluated as aBook.Pages (see below sections).
5| func (b *Book) Pages = (*b).Pages

```

This is why I don't reject the use the value receiver terminology (as the opposite of the pointer receiver terminology). After all, when we explicitly declare a method for a non-pointer type, in fact two methods are declared, the explicit one is for the non-pointer type and the implicit one is for the corresponding pointer type.

As mentioned at the last section, for each declared method, compilers will also declare a corresponding implicit function for it. So for the implicitly declared method, the following implicit function is declared by compiler.

```

1| func (*Book).Pages(b *Book) int {
2|     return Book.Pages(*b)
3|

```

In other words, for each explicitly declared method with a value receiver, two implicit functions and one implicit method will also be declared at the same time.

Method Specifications and Method Sets

A method specification can be viewed as a [function prototype](#) (§20) without the `func` keyword. We can view each method declaration is composed of the `func` keyword, a receiver parameter declaration, a method specification and a method (function) body.

For example, the method specifications of the `Pages` and `SetPages` methods shown above are

```

1| Pages() int
2| SetPages(pages int)

```

Each type has a method set. The method set of a non-interface type is composed of all the method specifications of the methods declared, either explicitly or implicitly, for the type, except the ones whose names are the blank identifier `_`. Interface types will be explained in [the next article](#) (§23).

For example, the method sets of the `Book` type shown in the previous sections is

```
1| Pages() int
```

and the method set of the `*Book` type is

```

1| Pages() int
2| SetPages(pages int)

```

The order of the method specifications in a method set is not important for the method set.

For a method set, if every method specification in it is also in another method set, then we say the former method set is a subset of the latter one, and the latter one is a superset of the former one. If two method sets are subsets (or supersets) of each other, then we say the two method sets are identical.

Given a type T , assume it is neither a pointer type nor an interface type, for [the reason](#) mentioned in the last section, the method set of a type T is always a subset of the method set of type $*T$. For example, the method set of the `Book` type shown above is a subset of the method set of the `*Book` type.

Please note, **non-exported method names, which start with lower-case letters, from different packages will be always viewed as two different method names, even if the two method names are the same in literal.**

Method sets play an important role in the polymorphism feature of Go. About polymorphism, please read [the next article](#) (§23) (interfaces in Go) for details.

The method sets of the following types are always blank:

- built-in basic types.
- defined pointer types.
- pointer types whose base types are interface or pointer types.
- unnamed array, slice, map, function and channel types.

Method Values and Method Calls

Methods are special functions actually. Methods are often called member functions. When a type owns a method, each value of the type will own an immutable member of a function type. The member name is the same as the method name and the type of the member is the same as the function declared with the form of the method declaration but without the receiver part.

A method call is just a call to such a member function. For a value v , its method m can be represented with the selector form $v.m$, which is a function value.

An example containing some method calls:

```
1| package main
2|
3| import "fmt"
4|
```

```

5| type Book struct {
6|     pages int
7| }
8|
9| func (b Book) Pages() int {
10|    return b.pages
11| }
12|
13| func (b *Book) SetPages(pages int) {
14|    b.pages = pages
15| }
16|
17| func main() {
18|     var book Book
19|
20|     fmt.Printf("%T \n", book.Pages)           // func() int
21|     fmt.Printf("%T \n", (&book).SetPages) // func(int)
22|     // &book has an implicit method.
23|     fmt.Printf("%T \n", (&book).Pages) // func() int
24|
25|     // Call the three methods.
26|     (&book).SetPages(123)
27|     book.SetPages(123) // equivalent to the last line
28|     fmt.Println(book.Pages())      // 123
29|     fmt.Println((&book).Pages()) // 123
30| }
```

(Different from C language, there is not the `->` operator in Go to call methods with pointer receivers, so `(&book)->SetPages(123)` is illegal in Go.)

Wait! Why does the line `book.SetPages(123)` in the above example compile okay? After all, the method `SetPages` is not declared for the `Book` type. On one hand, this can be viewed as a syntactic sugar to make programming convenient. This sugar only works for addressable value receivers. Compiler will implicitly take the address of the addressable value `book` when it is passed as the receiver argument of a `SetPages` method call. On the other hand, we should also think `aBookExpression.SetPages` is always a legal selector (from the syntax view), even if the expression `aBookExpression` is evaluated as an unaddressable `Book` value, for which case, the selector `aBookExpression.SetPages` is invalid (but legal).

As above just mentioned, when a method is declared for a type, each value of the type will own a member function. Zero values are not exceptions, whether or not the zero values of the types are represented by `nil`.

Example:

```

1| package main
2|
3| type StringSet map[string]struct{}
4| func (ss StringSet) Has(key string) bool {
5|     // Never panic here, even if ss is nil.
6|     _, present := ss[key]
7|     return present
8| }
9|
10| type Age int
11| func (age *Age) IsNil() bool {
12|     return age == nil
13| }
14| func (age *Age) Increase() {
15|     *age++ // If age is a nil pointer, then
16|             // dereferencing it will panic.
17| }
18|
19| func main() {
20|     _ = (StringSet(nil)).Has    // will not panic
21|     _ = ((*Age)(nil)).IsNil   // will not panic
22|     _ = ((*Age)(nil)).Increase // will not panic
23|
24|     _ = (StringSet(nil)).Has("key") // will not panic
25|     _ = ((*Age)(nil)).IsNil()      // will not panic
26|
27|     // This following line will panic. But the
28|     // panic is not caused by invoking the method.
29|     // It is caused by the nil pointer dereference
30|     // within the method body.
31|     ((*Age)(nil)).Increase()
32| }
```

Receiver Arguments Are Passed by Copy

Same as general function arguments, the receiver arguments are also passed by copy. So, the modifications on the [direct part](#) (§17) of a receiver argument in a method call will not be reflected to the outside of the method.

An example:

```
1| package main
2|
3| import "fmt"
4|
5| type Book struct {
6|     pages int
7| }
8|
9| func (b Book) SetPages(pages int) {
10|     b.pages = pages
11| }
12|
13| func main() {
14|     var b Book
15|     b.SetPages(123)
16|     fmt.Println(b.pages) // 0
17| }
```

Another example:

```
1| package main
2|
3| import "fmt"
4|
5| type Book struct {
6|     pages int
7| }
8|
9| type Books []Book
10|
11| func (books Books) Modify() {
12|     // Modifications on the underlying part of
13|     // the receiver will be reflected to outside
14|     // of the method.
15|     books[0].pages = 500
16|     // Modifications on the direct part of the
17|     // receiver will not be reflected to outside
18|     // of the method.
19|     books = append(books, Book{789})
20| }
21|
22| func main() {
23|     var books = Books{{123}, {456}}
24|     books.Modify()
```

```
25|     fmt.Println(books) // [{500} {456}]
26| }
```

Some off topic, if the two lines in the orders of the above `Modify` method are exchanged, then both of the modifications will not be reflected to outside of the method body.

```
1| func (books Books) Modify() {
2|     books = append(books, Book{789})
3|     books[0].pages = 500
4|
5|
6| func main() {
7|     var books = Books{{123}, {456}}
8|     books.Modify()
9|     fmt.Println(books) // [{123} {456}]
10| }
```

The reason here is that the `append` call will allocate a new memory block to store the elements of the copy of the passed slice receiver argument. The allocation will not reflect to the passed slice receiver argument itself.

To make both of the modifications be reflected to outside of the method body, the receiver of the method must be a pointer one.

```
1| func (books *Books) Modify() {
2|     *books = append(*books, Book{789})
3|     (*books)[0].pages = 500
4|
5|
6| func main() {
7|     var books = Books{{123}, {456}}
8|     books.Modify()
9|     fmt.Println(books) // [{500} {456} {789}]
10| }
```

Method Value Normalization

At compile time, compilers will normalize each method value expression, by changing implicit address taking and pointer dereference operations into explicit ones in that method value expression.

Assume `v` is a value of type `T` and `v.m` is a legal method value expression,

- if `m` is a method explicitly declared for type `*T`, then compilers will normalize it as `(&v).m`;

- if m is a method explicitly declared for type T , then the method value expression $v.m$ is already normalized.

Assume p is a value of type $*T$ and $p.m$ is a legal method value expression,

- if m is a method explicitly declared for type T , then compilers will normalize it as $(*p).m$;
- if m is a method explicitly declared for type $*T$, then the method value expression $p.m$ is already normalized.

Promoted method value Normalization will be explained in the following [type embedding](#) (§24) article.

Method Value Evaluation

Assume $v.m$ is a normalized method value expression, at run time, when the method value $v.m$ is evaluated, the receiver argument v is evaluated and a copy of the evaluation result is saved and used in later calls to the method value.

For example, in the following code,

- the method value expression $b.Pages$ is already normalized. At run time, a copy of the receiver argument b is saved. The copy is the same as `Book{pages: 123}`, the subsequent modification of value b has no effects on this copy. That is why the call `f1()` prints `123`.
- the method value expression $p.Pages$ is normalized as $(*p).Pages$ at compile time. At run time, the receiver argument $*p$ is evaluated to the current b value, which is `Book{pages: 123}`. A copy of the evaluation result is saved and used in later calls of the method value, that is why the call `f2()` also prints `123`.
- the method value expression $p.Pages2$ is already normalized. At run time, a copy of the receiver argument p is saved. The saved value is the address of the value b , thus any changes to b will be reflected through dereferencing of the saved value, that is why the call `g1()` prints `789`.
- the method value expression $b.Pages2$ is normalized as $(&b).Pages2$ at compile time. At run time, a copy of the evaluation result of $&b$ is saved. The saved value is the address of the value b , thus any changes to b will be reflected through dereferencing of the saved value, that is why the call `g2()` prints `789`.

```
1| package main
2|
3| import "fmt"
4|
```

```

5| type Book struct {
6|     pages int
7| }
8|
9| func (b Book) Pages() int {
10|    return b.pages
11| }
12|
13| func (b *Book) Pages2() int {
14|    return (*b).Pages()
15| }
16|
17| func main() {
18|     var b = Book{pages: 123}
19|     var p = &b
20|     var f1 = b.Pages
21|     var f2 = p.Pages
22|     var g1 = p.Pages2
23|     var g2 = b.Pages2
24|     b.pages = 789
25|     fmt.Println(f1()) // 123
26|     fmt.Println(f2()) // 123
27|     fmt.Println(g1()) // 789
28|     fmt.Println(g2()) // 789
29| }
```

A Defined Type Doesn't Obtain the Methods Declared Explicitly for the Source Type Used in Its Definition

For example, in the following code, unlike the defined type `MyInt`, the defined type `Age` has not an `IsOdd` method.

```

1| package main
2|
3| type MyInt int
4| func (mi MyInt) IsOdd() bool {
5|     return mi%2 == 1
6| }
7|
8| type Age MyInt
9|
10| func main() {
11|     var x MyInt = 3
```

```

12|     _ = x.IsOdd() // okay
13|
14|     var y Age = 36
15|     // _ = y.IsOdd() // error: y.IsOdd undefined
16|     _ = y
17| }
```

Should a Method Be Declared With Pointer Receiver or Value Receiver?

Firstly, from the last section, we know that sometimes we must declare methods with pointer receivers.

In fact, we can always declare methods with pointer receivers without any logic problems. It is just a matter of program performance that sometimes it is better to declare methods with value receivers.

For the cases value receivers and pointer receivers are both acceptable, here are some factors needed to be considered to make decisions.

- Too many pointer copies may cause heavier workload for garbage collector.
- If the size of a value receiver type is large, then the receiver argument copy cost may be not negligible. Pointer types are all [small-size](#) (§34) types.
- Declaring methods of both value receivers and pointer receivers for the same base type is more likely to cause data races if the declared methods are called concurrently in multiple goroutines.
- Values of the types in the `sync` standard package should not be copied, so declaring methods with value receivers for struct types which [embedding](#) (§24) the types in the `sync` standard package is problematic.

If it is hard to make a decision whether a method should use a pointer receiver or a value receiver, then just choose the pointer receiver way.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Interfaces in Go

Interface types are one special kind of type in Go. Interfaces play several important roles in Go. Fundamentally, interface types make Go support value boxing. Consequently, through value boxing, reflection and polymorphism get supported.

Since version 1.18, Go has already supported custom generics. In custom generics, an interface type could be (always) also used as type constraints. In fact, all type constraints are actually interface types. Before Go 1.18, all interface types may be used as value types. But since Go 1.18, some interface types may be only used as type constraints. Interface types which may be used as value types are called basic interface types.

This article was mainly written before Go supports custom generics, so it mainly talks about basic interfaces. About constraint-only interface types, please read the [Go generics 101](#) book for details.

Interface Types and Type Sets

An interface type defines some (type) requirements. All non-interface types satisfying these requirements form a type set, which is called the type set of the interface type.

The requirements defined for an interface type are expressed by embedding some interface elements in the interface type. Currently (Go 1.21), there are two kinds of interface elements, method elements and type elements.

- A method element presents as a [method specification](#) (§22). A method specification embedded in an interface type may not use the blank identifier `_` as its name.
- A type element may be a type name, a type literal, an approximation type, or a type union. The current article doesn't talk much about the latter two and only talks about type names and literals which denote interface types.

For example, the predeclared [error interface type](#), which definition is shown below, embeds a method specification `Error() string`. In the definition, `interface{...}` is called an interface type literal and the word `interface` are a keyword in Go.

```
1| type error interface {
2|     Error() string
3| }
```

We may also say the `error` interface type (directly) specified a method `Error() string`. Its type set is composed of all non-interface types which have a [method](#) (§22) with the specification `Error() string`. In theory, the type set is infinite. Surely, for a specified Go project, it is finite.

The following are some other interface type definitions and alias declarations.

```

1| // This interface directly specifies two methods and
2| // embeds two other interface types, one of which
3| // is a type name and the other is a type literal.
4| type ReadWriteCloser = interface {
5|     Read(buf []byte) (n int, err error)
6|     Write(buf []byte) (n int, err error)
7|     error           // a type name
8|     interface{ Close() error } // a type literal
9| }
10|
11| // This interface embeds an approximation type. Its type
12| // set includes all types whose underlying type is []byte.
13| type AnyByteSlice = interface {
14|     ~[]byte
15| }
16|
17| // This interface embeds a type union. Its type set includes
18| // 6 types: uint, uint8, uint16, uint32, uint64 and uintptr.
19| type Unsigned interface {
20|     uint | uint8 | uint16 | uint32 | uint64 | uintptr
21| }
```

Embedding an interface type (denoted by either a type name or a type literal) in another one is equivalent to (recursively) expanding the elements in the former into the latter. For example, the interface type denoted by the type alias `ReadWriteCloser` is equivalent to the interface type denoted by the following literal, which directly specifies four methods.

```

1| interface {
2|     Read(buf []byte) (n int, err error)
3|     Write(buf []byte) (n int, err error)
4|     Error() string
5|     Close() error
6| }
```

The type set of the above interface type is composed of all non-interface types which at least have the four methods specified by the interface type. The type set is also infinite. It is definitely a subset of the type set of `error` interface type.

Please note that, before Go 1.18, only interface type names may be embedded in interface types.

The interface types shown in the following code are all called blank interface types, which embeds nothing.

```

1| // The unnamed blank interface type.
2| interface{}
3|
4| // Nothing is a defined blank interface type.
5| type Nothing interface{}
```

In fact, Go 1.18 introduced a predeclared alias, `any`, which denotes the blank interface type `interface{}`.

The type set of a blank interface type is composed of all non-interface types.

Method Sets of Types

Each type has a [method set](#) (§22) associated with it.

- For a non-interface type, its method set is composed of the specifications of all [the methods \(either explicit or implicit ones\) declared](#) (§22) for it.
- For an interface type, its method set is composed of all the method specifications it specifies, either directly or indirectly through embedding other types.

In the examples shown in the last section,

- the method set of the interface type denoted by `ReadWriteCloser` contains four methods.
- the method set of the predeclared interface type `error` contains only one method.
- the method set of a blank interface type is empty.

For convenience, the method set of a type is often also called the method set of any value of the type.

Basic Interface Types

Basic interface types are the interface types which may be used as value types. A non-basic interface type is also called a constraint-only interface type.

Currently (Go 1.21), every basic interface type could be defined entirely by a method set (may be empty). In other words, a basic interface type doesn't need type elements to be defined.

In the examples shown in the section before the last, the interface type denoted by alias `ReadWriteCloser` is a basic type, but the `Unsigned` interface type and the type denoted by alias `AnyByteSlice` are not. The latter two are both constraint-only interface types.

Blank interface types and the predeclared `error` interface type are also all basic interface types.

Two unnamed basic interface types are identical if their method sets are identical. Please note, non-exported method names (which start with lower-case letters), from different packages will be always viewed as two different method names, even if the two method names themselves are the same.

Implementations

If a non-interface type is contained in the type set of an interface type, then we say the non-interface type implements the interface type. If the type set of an interface type is a subset of another interface type, then we say the former one implements the latter one.

An interface type always implements itself, as a type set is always a subset (or superset) of itself. Similarly, two interface types with the same method set implement each other. In fact, two unnamed interface types are identical if their type sets are identical.

If a type `T` implements an interface type `X`, then the method set of `T` must be superset of `X`, whether `T` is an interface type or a non-interface type. Generally, not vice versa. But if `X` is a basic interface, then vice versa. For example, in the examples provided in a previous section, the interface type denoted by `ReadWriteCloser` implements the `error` interface type.

Implementations are all implicit in Go. The compiler does not require implementation relations to be specified in code explicitly. There is not an `implements` keyword in Go. Go compilers will check the implementation relations automatically as needed.

For example, in the following example, the method sets of struct pointer type `*Book`, integer type `MyInt` and pointer type `*MyInt` all contain the method specification `About() string`, so they all implement the above mentioned interface type `Aboutable`.

```

1| type Aboutable interface {
2|     About() string
3| }
4|
5| type Book struct {
6|     name string
7|     // more other fields ...

```

```

8| }
9|
10| func (book *Book) About() string {
11|     return "Book: " + book.name
12| }
13|
14| type MyInt int
15|
16| func (MyInt) About() string {
17|     return "I'm a custom integer value"
18| }
```

The implicit implementation design makes it possible to let types defined in other library packages, such as standard packages, passively implement some interface types declared in user packages. For example, if we declare an interface type as the following one, then the type `DB` and type `Tx` declared in [the database/sql standard package](#) will both implement the interface type automatically, for they both have the three corresponding methods specified in the interface.

```

1| import "database/sql"
2|
3| ...
4|
5| type DatabaseStorer interface {
6|     Exec(query string, args ...interface{}) (sql.Result, error)
7|     Prepare(query string) (*sql.Stmt, error)
8|     Query(query string, args ...interface{}) (*sql.Rows, error)
9| }
```

Note, as the type set of a blank interface type is composed of all non-interface types, so all types implement any blank interface type. This is an important fact in Go.

Value Boxing

Again, currently (Go 1.21), the types of interface values must be basic interface types. In the remaining contents of the current article, when a value type is mentioned, the value type may be non-interface type or a basic interface type. It is never a constraint-only interface type.

We can view each interface value as a box to encapsulate a non-interface value. To box/encapsulate a non-interface value into an interface value, the type of the non-interface value must implement the type of the interface value.

If a type T implements a (basic) interface type I , then any value of type T can be implicitly converted to type I . In other words, any value of type T is [assignable](#) (§7) to (modifiable) values of type I . When a T value is converted (assigned) to an I value,

- if type T is a non-interface type, then a copy of the T value is boxed (or encapsulated) into the result (or destination) I value. The time complexity of the copy is $O(n)$, where n is the size of copied T value.
- if type T is also an interface type, then a copy of the value boxed in the T value is boxed (or encapsulated) into the result (or destination) I value. The standard Go compiler makes an optimization here, so the time complexity of the copy is $O(1)$, instead of $O(n)$.

The type information of the boxed value is also stored in the result (or destination) interface value. (This will be further explained below.)

When a value is boxed in an interface value, the value is called the ***dynamic value*** of the interface value. The type of the dynamic value is called the ***dynamic type*** of the interface value.

The direct part of the dynamic value of an interface value is immutable, though we can replace the dynamic value of an interface value with another dynamic value.

In Go, the zero values of any interface type are represented by the predeclared `nil` identifier. Nothing is boxed in a `nil` interface value. Assigning an untyped `nil` to an interface value will clear the dynamic value boxed in the interface value.

(Note, the zero values of many non-interface types in Go are also represented by `nil` in Go. Non-interface nil values can also be boxed in interface values. An interface value boxing a nil non-interface value still boxes something, so it is not a nil interface value.)

As any type implements all blank interface types, so any non-interface value can be boxed in (or assigned to) a blank interface value. For this reason, blank interface types can be viewed as the `any` type in many other languages.

When an untyped value (except untyped `nil` values) is assigned to a blank interface value, the untyped value will be first converted to its default type. (In other words, we can think the untyped value is deduced as a value of its default type).

Let's view an example which demonstrates some assignments with interface values as the destinations.

```
1| package main
2|
3| import "fmt"
```

```

4|
5| type Aboutable interface {
6|   About() string
7| }
8|
9| // Type *Book implements Aboutable.
10| type Book struct {
11|   name string
12| }
13| func (book *Book) About() string {
14|   return "Book: " + book.name
15| }
16|
17| func main() {
18|   // A *Book value is boxed into an
19|   // interface value of type Aboutable.
20|   var a Aboutable = &Book{"Go 101"}
21|   fmt.Println(a) // &{Go 101}
22|
23|   // i is a blank interface value.
24|   var i interface{} = &Book{"Rust 101"}
25|   fmt.Println(i) // &{Rust 101}
26|
27|   // Aboutable implements interface{}.
28|   i = a
29|   fmt.Println(i) // &{Go 101}
30| }
```

Please note, the prototype of the `fmt.Println` function used many times in previous articles is

```
func Println(a ...interface{}) (n int, err error)
```

This is why a `fmt.Println` function calls can take arguments of any types.

The following is another example which shows how a blank interface value is used to box values of any non-interface type.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|   var i interface{}
7|   i = []int{1, 2, 3}
8|   fmt.Println(i) // [1 2 3]
```

```

9|     i = map[string]int{"Go": 2012}
10|    fmt.Println(i) // map[Go:2012]
11|    i = true
12|    fmt.Println(i) // true
13|    i = 1
14|    fmt.Println(i) // 1
15|    i = "abc"
16|    fmt.Println(i) // abc
17|
18|    // Clear the boxed value in interface value i.
19|    i = nil
20|    fmt.Println(i) // <nil>
21| }
```

Go compilers will build a global table which contains the information of each type at compile time. The information includes what [kind](#) (§14) a type is, what methods and fields a type owns, what the element type of a container type is, type sizes, etc. The global table will be loaded into memory when a program starts.

At run time, when a non-interface value is boxed into an interface value, the Go runtime (at least for the standard Go runtime) will analyze and build the implementation information for the type pair of the two values, and store the implementation information in the interface value. The implementation information for each non-interface type and interface type pair will only be built once and cached in a global map for execution efficiency consideration. The number of entries of the global map never decreases. In fact, a non-nil interface value just uses [an internal pointer field which references a cached implementation information entry](#) (§17).

The implementation information for each (interface type, dynamic type) pair includes two pieces of information:

1. the information of the dynamic type (a non-interface type)
2. and a method table (a slice) which stores all the corresponding methods specified by the interface type and declared for the non-interface type (the dynamic type).

These two pieces of information are essential for implementing two important features in Go:

1. The dynamic type information is the key to implement [reflection](#) in Go.
2. The method table information is the key to implement polymorphism (polymorphism will be explained in the next section).

Polymorphism

Polymorphism is one key functionality provided by interfaces, and it is an important feature of Go.

When a non-interface value `t` of a type `T` is boxed in an interface value `i` of type `I`, calling a method specified by the interface type `I` on the interface value `i` will call the corresponding method declared for the non-interface type `T` on the non-interface value `t` actually. In other words, **calling the method of an interface value will actually call the corresponding method of the dynamic value of the interface value**. For example, calling method `i.m` will call method `t.m` actually. With different dynamic values of different dynamic types boxed into the interface value, the interface value behaves differently. This is called polymorphism.

When method `i.m` is called, the method table in the implementation information stored in `i` will be looked up to find and call the corresponding method `t.m`. The method table is a slice and the lookup is just a slice element indexing, so this is quick.

(Note, calling methods on a nil interface value will panic at run time, for there are no available declared methods to be called.)

An example:

```

1| package main
2|
3| import "fmt"
4|
5| type Filter interface {
6|     About() string
7|     Process([]int) []int
8| }
9|
10| // UniqueFilter is used to remove duplicate numbers.
11| type UniqueFilter struct{}
12| func (UniqueFilter) About() string {
13|     return "remove duplicate numbers"
14| }
15| func (UniqueFilter) Process(inputs []int) []int {
16|     outs := make([]int, 0, len(inputs))
17|     pusheds := make(map[int]bool)
18|     for _, n := range inputs {
19|         if !pusheds[n] {
20|             pusheds[n] = true
21|             outs = append(outs, n)
22|         }
23|     }
24|     return outs
25| }
```

```

26|
27| // MultipleFilter is used to keep only
28| // the numbers which are multiples of
29| // the MultipleFilter as an int value.
30| type MultipleFilter int
31| func (mf MultipleFilter) About() string {
32|     return fmt.Sprintf("keep multiples of %v", mf)
33| }
34| func (mf MultipleFilter) Process(inputs []int) []int {
35|     var outs = make([]int, 0, len(inputs))
36|     for _, n := range inputs {
37|         if n % int(mf) == 0 {
38|             outs = append(outs, n)
39|         }
40|     }
41|     return outs
42| }
43|
44| // With the help of polymorphism, only one
45| // "filterAndPrint" function is needed.
46| func filterAndPrint(fltr Filter, unfiltered []int) []int {
47|     // Calling the methods of "fltr" will call the
48|     // methods of the value boxed in "fltr" actually.
49|     filtered := fltr.Process(unfiltered)
50|     fmt.Println(fltr.About() + ":\n\t", filtered)
51|     return filtered
52| }
53|
54| func main() {
55|     numbers := []int{12, 7, 21, 12, 12, 26, 25, 21, 30}
56|     fmt.Println("before filtering:\n\t", numbers)
57|
58|     // Three non-interface values are boxed into
59|     // three Filter interface slice element values.
60|     filters := []Filter{
61|         UniqueFilter{},
62|         MultipleFilter(2),
63|         MultipleFilter(3),
64|     }
65|
66|     // Each slice element will be assigned to the
67|     // local variable "fltr" (of interface type
68|     // Filter) one by one. The value boxed in each
69|     // element will also be copied into "fltr".
70|     for _, fltr := range filters {

```

```

71|     numbers = filterAndPrint(fltr, numbers)
72| }
73| }
```

The output:

```

before filtering:
[12 7 21 12 12 26 25 21 30]
remove duplicate numbers:
[12 7 21 26 25 30]
keep multiples of 2:
[12 26 30]
keep multiples of 3:
[12 30]
```

In the above example, polymorphism makes it unnecessary to write one `filterAndPrint` function for each filter type.

Besides the above benefit, polymorphism also makes it possible for the developers of a library code package to declare an exported interface type and declare a function (or method) which has a parameter of the interface type, so that a user of the package can declare a type, which implements the interface type, in user code and pass arguments of the user type to calls to the function (or method). The developers of the code package don't need to care about how the user type is declared, as long as the user type satisfies the behaviors specified by the interface type declared in the library code package.

In fact, polymorphism is not an essential feature for a language. There are alternative ways to achieve the same job, such as callback functions. But the polymorphism way is cleaner and more elegant.

Reflection

The dynamic type information stored in an interface value can be used to inspect the dynamic value of the interface value and manipulate the values referenced by the dynamic value. This is called reflection in programming.

This article will not explain the functionalities provided by [the reflect standard package ↗](#). Please read [reflections in Go](#) (§27) to get how to use that package. Below will only introduce the built-in reflection functionalities in Go. In Go, built-in reflections are achieved with type assertions and `type-switch` control flow code blocks.

Type assertion

There are four kinds of interface-value-involving value conversion cases in Go:

1. convert a non-interface value to an interface value, where the type of the non-interface value must implement the type of the interface value.
2. convert an interface value to an interface value, where the type of the source interface value must implement the type of the destination interface value.
3. convert an interface value to a non-interface value, where the type of the non-interface value must implement the type of the interface value.
4. convert an interface value to an interface value, where the type of the source interface value doesn't implement the destination interface type, but the dynamic type of the source interface value might implement the destination interface type.

We have already explained the first two kinds of cases. They both require that the source value type implements the destination interface type. The convertibility for the first two are verified at compile time.

Here will explain the later two kinds of cases. The convertibility for the later two are verified at run time, by using a syntax called ***type assertion***. In fact, the syntax also applies to the second kind of conversion in our above list.

The form of a type assertion expression is `i.(T)`, where `i` is an interface value and `T` is a type name or a type literal. Type `T` must be

- either an arbitrary non-interface type,
- or an arbitrary interface type.

In a type assertion `i.(T)`, `i` is called the asserted value and `T` is called the asserted type. A type assertion might succeed or fail.

- In the case of `T` being a non-interface type, if the dynamic type of `i` exists and is identical to `T`, then the assertion will succeed, otherwise, the assertion will fail. When the assertion succeeds, the evaluation result of the assertion is a copy of the dynamic value of `i`. We can view assertions of this kind as value unboxing attempts.
- In the case of `T` being an interface type, if the dynamic type of `i` exists and implements `T`, then the assertion will succeed, otherwise, the assertion will fail. When the assertion succeeds, a copy of the dynamic value of `i` will be boxed into a `T` value and the `T` value will be used as the evaluation result of the assertion.

When a type assertion fails, its evaluation result is a zero value of the asserted type.

By the rules described above, if the asserted value in a type assertion is a nil interface value, then the assertion will always fail.

For most scenarios, a type assertion is used as a single-value expression. However, when a type assertion is used as the only source value expression in an assignment, it can result in a second optional untyped boolean value and be viewed as a multi-value expression. The second optional untyped boolean value indicates whether or not the type assertion succeeds.

Note, if a type assertion fails and the type assertion is used as a single-value expression (the second optional bool result is absent), then a panic will occur.

An example which shows how to use type assertions (asserted types are non-interface types):

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     // Compiler will deduce the type of 123 as int.
7|     var x interface{} = 123
8|
9|     // Case 1:
10|    n, ok := x.(int)
11|    fmt.Println(n, ok) // 123 true
12|    n = x.(int)
13|    fmt.Println(n) // 123
14|
15|     // Case 2:
16|    a, ok := x.(float64)
17|    fmt.Println(a, ok) // 0 false
18|
19|     // Case 3:
20|    a = x.(float64) // will panic
21| }
```

Another example which shows how to use type assertions (asserted types are interface types):

```

1| package main
2|
3| import "fmt"
4|
5| type Writer interface {
6|     Write(buf []byte) (int, error)
7| }
8| 
```

```

9| type DummyWriter struct{}
10| func (DummyWriter) Write(buf []byte) (int, error) {
11|     return len(buf), nil
12| }
13|
14| func main() {
15|     var x interface{} = DummyWriter{}
16|     var y interface{} = "abc"
17|     // Now the dynamic type of y is "string".
18|     var w Writer
19|     var ok bool
20|
21|     // Type DummyWriter implements both
22|     // Writer and interface{}.
23|     w, ok = x.(Writer)
24|     fmt.Println(w, ok) // {} true
25|     x, ok = w.(interface{})
26|     fmt.Println(x, ok) // {} true
27|
28|     // The dynamic type of y is "string",
29|     // which doesn't implement Writer.
30|     w, ok = y.(Writer)
31|     fmt.Println(w, ok) // <nil> false
32|     w = y.(Writer)    // will panic
33| }
```

In fact, for an interface value `i` with a dynamic type `T`, the method call `i.m(...)` is equivalent to the method call `i.(T).m(...)`.

type-switch control flow block

The `type-switch` code block syntax may be the weirdest syntax in Go. It can be viewed as the enhanced version of type assertion. A `type-switch` code block is in some way similar to a `switch-case` control flow code block. It looks like:

```

1| switch aSimpleStatement; v := x.(type) {
2| case TypeA:
3|     ...
4| case TypeB, TypeC:
5|     ...
6| case nil:
7|     ...
8| default:
```

```

9| ...
10| }
```

The `aSimpleStatement;` portion is optional in a type-switch code block.

`aSimpleStatement` must be a [simple statement](#) (§11). `x` must be an interface value and it is called the asserted value. `v` is called the assertion result, it must be present in a short variable declaration form.

Each `case` keyword in a type-switch block can be followed by the predeclared `nil` identifier or a comma-separated list composed of at least one type name and type literal. None of such items (`nil`, type names and type literals) may be duplicate in the same type-switch code block.

If the type denoted by a type name or type literal following a `case` keyword in a type-switch code block is not an interface type, then it must implement the interface type of the asserted value.

Here is an example in which a type-switch control flow code block is used.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     values := []interface{}{
7|         456, "abc", true, 0.33, int32(789),
8|         []int{1, 2, 3}, map[int]bool{}, nil,
9|     }
10|    for _, x := range values {
11|        // Here, v is declared once, but it denotes
12|        // different variables in different branches.
13|        switch v := x.(type) {
14|            case []int: // a type literal
15|                // The type of v is "[]int" in this branch.
16|                fmt.Println("int slice:", v)
17|            case string: // one type name
18|                // The type of v is "string" in this branch.
19|                fmt.Println("string:", v)
20|            case int, float64, int32: // multiple type names
21|                // The type of v is "interface{}", 
22|                // the same as x in this branch.
23|                fmt.Println("number:", v)
24|            case nil:
25|                // The type of v is "interface{}",
26|                // the same as x in this branch.
27|                fmt.Println(v)
```

```

28|     default:
29|         // The type of v is "interface{}",
30|         // the same as x in this branch.
31|         fmt.Println("others:", v)
32|     }
33|     // Note, each variable denoted by v in the
34|     // last three branches is a copy of x.
35| }
36| }
```

The output:

```

number: 456
string: abc
others: true
number: 0.33
number: 789
int slice: [1 2 3]
others: map[]
<nil>
```

The above example is equivalent to the following in logic:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     values := []interface{}{
7|         456, "abc", true, 0.33, int32(789),
8|         []int{1, 2, 3}, map[int]bool{}, nil,
9|     }
10|    for _, x := range values {
11|        if v, ok := x.([]int); ok {
12|            fmt.Println("int slice:", v)
13|        } else if v, ok := x.(string); ok {
14|            fmt.Println("string:", v)
15|        } else if x == nil {
16|            v := x
17|            fmt.Println(v)
18|        } else {
19|            _, isInt := x.(int)
20|            _, isFloat64 := x.(float64)
21|            _, isInt32 := x.(int32)
22|            if isInt || isFloat64 || isInt32 {
```

```

23|     v := x
24|     fmt.Println("number:", v)
25| } else {
26|     v := x
27|     fmt.Println("others:", v)
28| }
29|
30| }
31| }
```

`type-switch` code blocks are similar to `switch-case` code blocks in some aspects.

- Like `switch-case` blocks, in a `type-switch` code block, there can be at most one `default` branch.
- Like `switch-case` blocks, in a `type-switch` code block, if the `default` branch is present, it can be the last branch, the first branch, or a middle branch.
- Like `switch-case` blocks, a `type-switch` code block may not contain any branches, it will be viewed as a no-op.

But, unlike `switch-case` code blocks, `fallthrough` statements can't be used within branch blocks of a `type-switch` code block.

More About Interfaces in Go

Comparisons involving interface values

There are two cases of comparisons involving interface values:

1. comparisons between a non-interface value and an interface value.
2. comparisons between two interface values.

For the first case, the type of the non-interface value must implement the type (assume it is `I`) of the interface value, so the non-interface value can be converted to (boxed into) an interface value of `I`. This means a comparison between a non-interface value and an interface value can be translated to a comparison between two interface values. So below only comparisons between two interface values will be explained.

Comparing two interface values is comparing their respective dynamic types and dynamic values actually.

The steps of comparing two interface values (with the `==` operator):

1. if one of the two interface values is a nil interface value, then the comparison result is whether or not the other interface value is also a nil interface value.
2. if the dynamic types of the two interface values are two different types, then the comparison result is `false`.
3. in the case where the dynamic types of the two interface values are the same type,
 - if the same dynamic type is an [incomparable type](#) (§48), a panic will occur.
 - otherwise, the comparison result is the result of comparing the dynamic values of the two interface values.

In short, two interface values are equal only if one of the following conditions are satisfied.

1. They are both nil interface values.
2. Their dynamic types are identical and comparable, and their dynamic values are equal to each other.

By the rules, two interface values which dynamic values are both `nil` may be not equal. An example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     var a, b, c interface{} = "abc", 123, "a"+"b"+"c"
7|     // A case of step 2.
8|     fmt.Println(a == b) // false
9|     // A case of step 3.
10|    fmt.Println(a == c) // true
11|
12|    var x *int = nil
13|    var y *bool = nil
14|    var ix, iy interface{} = x, y
15|    var i interface{} = nil
16|    // A case of step 2.
17|    fmt.Println(ix == iy) // false
18|    // A case of step 1.
19|    fmt.Println(ix == i) // false
20|    // A case of step 1.
21|    fmt.Println(iy == i) // false
22|
23|    // []int is an incomparable type
24|    var s []int = nil
25|    i = s
26|    // A case of step 1.

```

```

27|     fmt.Println(i == nil) // false
28|     // A case of step 3.
29|     fmt.Println(i == i) // will panic
30| }
```

The internal structure of interface values

For the official Go compiler/runtime, blank interface values and non-blank interface values are represented with two different internal structures. Please read [value parts](#) (§17) for details.

Pointer dynamic value vs. non-pointer dynamic value

The official Go compiler/runtime makes an optimization which makes boxing pointer values into interface values more efficient than boxing non-pointer values. For [small size values](#) (§34), the efficiency differences are small, but for large size values, the differences may be not small. For the same optimization, type assertions with a pointer type are also more efficient than type assertions with the base type of the pointer type if the base type is a large size type.

So please try to avoid boxing large size values, box their pointers instead.

Values of `[]T` can't be directly converted to `[]I`, even if type `T` implements interface type `I`.

For example, sometimes, we may need to convert a `[]string` value to `[]interface{}` type. Unlike some other languages, there is no direct way to make the conversion. We must make the conversion manually in a loop:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     words := []string{
7|         "Go", "is", "a", "high",
8|         "efficient", "language.",
9|     }
10|
11|    // The prototype of fmt.Println function is
12|    // func Println(a ...interface{}) (n int, err error).
13|    // So words... can't be passed to it as the argument.
```

```

14|
15|     // fmt.Println(words...) // not compile
16|
17|     // Convert the []string value to []interface{}.
18|     iw := make([]interface{}, 0, len(words))
19|     for _, w := range words {
20|         iw = append(iw, w)
21|     }
22|     fmt.Println(iw...) // compiles okay
23|

```

Each method specified in an interface type corresponds to an implicit function

For each method with name `m` in the method set defined by an interface type `I`, compilers will implicitly declare a function named `I.m`, which has one more input parameter, of type `I`, than method `m`. The extra parameter is the first input parameter of function `I.m`. Assume `i` is an interface value of `I`, then the method call `i.m(...)` is equivalent to the function call `I.m(i, ...)`.

An example:

```

1| package main
2|
3| import "fmt"
4|
5| type I interface {
6|     m(int)bool
7| }
8|
9| type T string
10| func (t T) m(n int) bool {
11|     return len(t) > n
12| }
13|
14| func main() {
15|     var i I = T("gopher")
16|     fmt.Println(i.m(5))           // true
17|     fmt.Println(I.m(i, 5))        // true
18|     fmt.Println(interface{m(int)bool}.m(i, 5)) // true
19|
20|     // The following lines compile okay,
21|     // but will panic at run time.

```

```
22|     I(nil).m(5)
23|     I.m(nil, 5)
24|     interface {m(int) bool}.m(nil, 5)
25| }
```

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Type Embedding

From the article [structs in Go](#) (§16), we know that a struct type can have many fields. Each field is composed of one field name and one field type. In fact, sometimes, a struct field can be composed of a field type only. The way to declare struct fields is called type embedding.

This article will explain the purpose of type embedding and all kinds of details in type embedding.

What Does Type Embedding Look Like?

Here is an example demonstrating type embedding:

```

1| package main
2|
3| import "net/http"
4|
5| func main() {
6|     type P = *bool
7|     type M = map[int]int
8|     var x struct {
9|         string // a named non-pointer type
10|        error // a named interface type
11|        *int    // an unnamed pointer type
12|        P       // an alias of an unnamed pointer type
13|        M       // an alias of an unnamed type
14|
15|         http.Header // a named map type
16|     }
17|     x.string = "Go"
18|     x.error = nil
19|     x.int = new(int)
20|     x.P = new(bool)
21|     x.M = make(M)
22|     x.Header = http.Header{}
23| }
```

In the above example, six types are embedded in the struct type. Each type embedding forms an embedded field.

Embedded fields are also called as anonymous fields. However, each embedded field has a name specified implicitly. The [unqualified](#) [type](#) name of an embedded field acts as the name of the

field. For example, the names of the six embedded fields in the above examples are `string`, `error`, `int`, `P`, `M`, and `Header`, respectively.

Which Types Can be Embedded?

The current Go specification (version 1.21) [says](#)

An embedded field 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 above description was accurate before Go 1.9. However, with the introduction of type aliases in Go 1.9, the description [has become a little outdated and inaccurate](#). For example, the description doesn't include the case of the `P` field in the example in the last section.

Here, the article tries to provide more accurate descriptions.

- A type name `T` can be embedded as an embedded field unless `T` denotes a named pointer type or a pointer type whose base type is either a pointer or an interface type.
- A pointer type `*T`, where `T` is a type name denoting the base type of the pointer type, can be embedded as an embedded field unless type name `T` denotes a pointer or interface type.

The following lists some example types which can and can't be embedded:

```

1| type Encoder interface {Encode([]byte) []byte}
2| type Person struct {name string; age int}
3| type Alias = struct {name string; age int}
4| type AliasPtr = *struct {name string; age int}
5| type IntPtr *int
6| type AliasPP = *IntPtr
7|
8| // These types and aliases can be embedded.
9| Encoder
10| Person
11| *Person
12| Alias
13| *Alias
14| AliasPtr
15| int
16| *int
17|
18| // These types and aliases can't be embedded.
19| AliasPP          // base type is a pointer type
20| *Encoder         // base type is an interface type

```

```

21| *AliasPtr           // base type is a pointer type
22| IntPtr               // named pointer type
23| *IntPtr              // base type is a pointer type
24| *chan int             // base type is an unnamed type
25| struct {age int}     // unnamed non-pointer type
26| map[string]int        // unnamed non-pointer type
27| []int64                // unnamed non-pointer type
28| func()                 // unnamed non-pointer type

```

No two fields are allowed to have the same name in a struct, there are no exceptions for anonymous struct fields. By the embedded field naming rules, an unnamed pointer type can't be embedded along with its base type in the same struct type. For example, `int` and `*int` can't be embedded in the same struct type.

A struct type can't embed itself or its aliases, recursively.

Generally, it is only meaningful to embed types who have fields or methods (the following sections will explain why), though some types without any field and method can also be embedded.

What Is the Meaningfulness of Type Embedding?

The main purpose of type embedding is to extend the functionalities of the embedded types into the embedding type, so that we don't need to re-implement the functionalities of the embedded types for the embedding type.

Many other object-oriented programming languages use inheritance to achieve the same goal of type embedding. Both mechanisms have their own [benefits and drawbacks ↗](#). Here, this article will not discuss which one is better. We should just know Go chose the type embedding mechanism, and there is a big difference between the two:

- If a type `T` inherits another type, then type `T` obtains the abilities of the other type. At the same time, each value of type `T` can also be viewed as a value of the other type.
- If a type `T` embeds another type, then type other type becomes a part of type `T`, and type `T` obtains the abilities of the other type, but none values of type `T` can be viewed as values of the other type.

Here is an example to show how an embedding type extends the functionalities of the embedded type.

```

1| package main
2|
3| import "fmt"

```

```

4|
5| type Person struct {
6|     Name string
7|     Age  int
8| }
9| func (p Person) PrintName() {
10|    fmt.Println("Name:", p.Name)
11| }
12| func (p *Person) SetAge(age int) {
13|     p.Age = age
14| }
15|
16| type Singer struct {
17|     Person // extends Person by embedding it
18|     works  []string
19| }
20|
21| func main() {
22|     var gaga = Singer{Person: Person{"Gaga", 30}}
23|     gaga.PrintName() // Name: Gaga
24|     gaga.Name = "Lady Gaga"
25|     (&gaga).SetAge(31)
26|     (&gaga).PrintName() // Name: Lady Gaga
27|     fmt.Println(gaga.Age) // 31
28| }
```

From the above example, it looks that, after embedding type `Person`, the type `Singer` obtains all methods and fields of type `Person`, and type `*Singer` obtains all methods of type `*Person`. Are the conclusions right? The following sections will answer this question.

Please note that, a `Singer` value is not a `Person` value, the following code doesn't compile:

```

1| var gaga = Singer{}
2| var _ Person = gaga
```

Does the Embedding Type Obtain the Fields and Methods of the Embedded Types?

Let's list all the fields and methods of type `Singer` and the methods of type `*Singer` used in the last example by using [the reflection functionalities](#) (§27) provided in the `reflect` standard package.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "reflect"
6| )
7|
8| ... // the types declared in the last example
9|
10| func main() {
11|     t := reflect.TypeOf(Singer{}) // the Singer type
12|     fmt.Println(t, "has", t.NumField(), "fields:")
13|     for i := 0; i < t.NumField(); i++ {
14|         fmt.Print(" field#", i, ": ", t.Field(i).Name, "\n")
15|     }
16|     fmt.Println(t, "has", t.NumMethod(), "methods:")
17|     for i := 0; i < t.NumMethod(); i++ {
18|         fmt.Print(" method#", i, ": ", t.Method(i).Name, "\n")
19|     }
20|
21|     pt := reflect.TypeOf(&Singer{}) // the *Singer type
22|     fmt.Println(pt, "has", pt.NumMethod(), "methods:")
23|     for i := 0; i < pt.NumMethod(); i++ {
24|         fmt.Print(" method#", i, ": ", pt.Method(i).Name, "\n")
25|     }
26| }
```

The result:

```

main.Singer has 2 fields:
field#0: Person
field#1: works
main.Singer has 1 methods:
method#0: PrintName
*main.Singer has 2 methods:
method#0: PrintName
method#1: SetAge
```

From the result, we know that the type `Singer` really owns a `PrintName` method, and the type `*Singer` really owns two methods, `PrintName` and `SetAge`. But the type `Singer` doesn't own a `Name` field. Then why is the selector expression `gaga.Name` legal for a `Singer` value `gaga`? Please read the next section to get the reason.

Shorthands of Selectors

From the articles [structs in Go](#) (§16) and [methods in Go](#) (§22), we have learned that, for a value `x`, `x.y` is called a selector, where `y` is either a field name or a method name. If `y` is a field name, then `x` must be a struct value or a struct pointer value. A selector is an expression, which represents a value. If the selector `x.y` denotes a field, it may also have its own fields (if `x.y` is a struct value) and methods. Such as `x.y.z`, where `z` can also be either a field name or a method name.

In Go, (without considering selector colliding and shadowing explained in a later section), ***if a middle name in a selector corresponds to an embedded field, then that name can be omitted from the selector.*** This is why embedded fields are also called anonymous fields.

For example:

```

1| package main
2|
3| type A struct {
4|     FieldX int
5| }
6|
7| func (a A) MethodA() {}
8|
9| type B struct {
10|     *A
11| }
12|
13| type C struct {
14|     B
15| }
16|
17| func main() {
18|     var c = &C{B: B{A: &A{FieldX: 5}}}
19|
20|     // The following 4 lines are equivalent.
21|     _ = c.B.A.FieldX
22|     _ = c.B.FieldX
23|     _ = c.A.FieldX // A is a promoted field of C
24|     _ = c.FieldX   // FieldX is a promoted field
25|
26|     // The following 4 lines are equivalent.
27|     c.B.A.MethodA()
28|     c.B.MethodA()
29|     c.A.MethodA()
30|     c.MethodA() // MethodA is a promoted method of C
31| }
```

This is why the expression `gaga.Name` is legal in the example in the last section. For it is just the shorthand of `gaga.Person.Name`.

Similarly, the selector `gaga.PrintName` can be viewed as a shorthand of `gaga.Person.PrintName`. But, it is also okay if we think it is not a shorthand. After all, the type `Singer` really has a `PrintName` method, though the method is declared implicitly (please read the section after next for details). For the similar reason, the selector `(&gaga).PrintName` and `(&gaga).SetAge` can also be viewed as, or not as, shorthands of `(&gaga.Person).PrintName` and `(&gaga.Person).SetAge`.

`Name` is called a promoted field of type `Singer`. `PrintName` is called a promoted method of type `Singer`.

Note, we can also use the selector `gaga.SetAge`, only if `gaga` is an addressable value of type `Singer`. It is just syntactical sugar of `(&gaga).SetAge`. Please read [method calls](#) (§22) for details.

In the above examples, `c.B.A.FieldX` is called the full form of selectors `c.FieldX`, `c.B.FieldX` and `c.A.FieldX`. Similarly, `c.B.A.MethodA` is called the full form of selectors `c.MethodA`, `c.B.MethodA` and `c.A.MethodA`.

If every middle name in the full form of a selector corresponds to an embedded field, then the number of middle names in the selector is called the depth of the selector. For example, the depth of the selector `c.MethodA` used in an above example is 2, for the full form of the selector is `c.B.A.MethodA`.

Selector Shadowing and Colliding

For a value `x` (we should always assume it is addressable, even if it is not), it is possible that many of its full-form selectors have the same last item `y` and every middle name of these selectors represents an embedded field. For such cases,

- only the full-form selector with the shallowest depth (assume it is the only one) can be shortened as `x.y`. In other words, `x.y` denotes the full-form selector with the shallowest depth. Other full-form selectors are **shadowed** by the one with the shallowest depth.
- if there are more than one full-form selectors with the shallowest depth, then none of those full-form selectors can be shortened as `x.y`. We say those full-form selectors with the shallowest depth are **colliding** with each other.

If a method selector is shadowed by another method selector, and the two corresponding method signatures are identical, we say the first method is overridden by the other one.

For example, assume A, B and C are three [defined types](#) (§14).

```

1| type A struct {
2|   x string
3|
4| func (A) y(int) bool {
5|   return false
6| }
7|
8| type B struct {
9|   y bool
10|
11| func (B) x(string) {}
12|
13| type C struct {
14|   B
15| }
```

The following code doesn't compile. The reason is the depths of the selectors `v1.A.x` and `v1.B.x` are equal, so the two selectors collide with each other and neither of them can be shortened to `v1.x`. The same situation is for the selectors `v1.A.y` and `v1.B.y`.

```

1| var v1 struct {
2|   A
3|   B
4| }
5|
6| func f1() {
7|   _ = v1.x // error: ambiguous selector v1.x
8|   _ = v1.y // error: ambiguous selector v1.y
9| }
```

The following code compiles okay. The selector `v2.C.B.x` is shadowed by `v2.A.x`, so the selector `v2.x` is a shortened form of `v2.A.x` actually. For the same reason, the selector `v2.y` is a shortened form of `v2.A.y`, not of `v2.C.B.y`.

```

1| var v2 struct {
2|   A
3|   C
4| }
5|
```

```

6| func f2() {
7|     fmt.Printf("%T \n", v2.x) // string
8|     fmt.Printf("%T \n", v2.y) // func(int) bool
9| }
```

Colliding or shadowed selectors don't prevent their deeper selectors being promoted. For example, the `.M` and `.z` selectors still get promoted in the following example.

```

1| package main
2|
3| type x string
4| func (x) M() {}
5|
6| type y struct {
7|     z byte
8| }
9|
10| type A struct {
11|     x
12| }
13| func (A) y(int) bool {
14|     return false
15| }
16|
17| type B struct {
18|     y
19| }
20| func (B) x(string) {}
21|
22| func main() {
23|     var v struct {
24|         A
25|         B
26|     }
27|     //_ = v.x // error: ambiguous selector v.x
28|     //_ = v.y // error: ambiguous selector v.y
29|     _ = v.M // ok. <=> v.A.x.M
30|     _ = v.z // ok. <=> v.B.y.z
31| }
```

One detail which is unusual but should be noted is that two unexported methods (or fields) from two different packages are always viewed as two different identifiers, even if their names are identical. So they will not never collide with or shadow each other when their owner types are embedded in the same struct type. For example, a program comprising two packages as the following shows will

§24. Type Embedding

compile and run okay. But if all the `m()` occurrences are replaced with `M()`, then the program will fail to compile for `A.M` and `B.M` collide with each other, so `c.M` is not a valid selector.

```
1| package foo // import "x.y/foo"
2|
3| import "fmt"
4|
5| type A struct {
6|     n int
7| }
8|
9| func (a A) m() {
10|     fmt.Println("A", a.n)
11| }
12|
13| type I interface {
14|     m()
15| }
16|
17| func Bar(i I) {
18|     i.m()
19| }
```

```
1| package main
2|
3| import "fmt"
4| import "x.y/foo"
5|
6| type B struct {
7|     n bool
8| }
9|
10| func (b B) m() {
11|     fmt.Println("B", b.n)
12| }
13|
14| type C struct{
15|     foo.A
16|     B
17| }
18|
19| func main() {
20|     var c C
21|     c.m()      // B false
```

```
22|     foo.Bar(c) // A 0
23| }
```

Implicit Methods for Embedding Types

As mentioned above, both of type `Singer` and type `*Singer` have a `PrintName` method each, and the type `*Singer` also has a `SetAge` method. However, we never explicitly declare these methods for the two types. Where do these methods come from?

In fact, assume a struct type `S` embeds a type (or a type alias) `T` and the embedding is legal,

- for each method of the embedded type `T`, if the selectors to that method neither collide with nor are shadowed by other selectors, then compilers will implicitly declare a corresponding method with the same specification for the embedding struct type `S`. And consequently, compilers will also [implicitly declare a corresponding method](#) (§22) for the pointer type `*S`.
- for each method of the pointer type `*T`, if the selectors to that method neither collide with nor are shadowed by other selectors, then compilers will implicitly declare a corresponding method with the same specification for the pointer type `*S`.

Simply speaking,

- type `struct{T}` and type `*struct{T}` both obtain all the methods of the type denoted by `T`.
- type `*struct{T}`, type `struct{*T}`, and type `*struct{*T}` all obtain all the methods of type `*T`.

The following (promoted) methods are implicitly declared by compilers for type `Singer` and type `*Singer`.

```
1| // Note: these declarations are not legal Go syntax.
2| // They are shown here just for explanation purpose.
3| // They indicate how implicit method values are
4| // evaluated (see the next section for more).
5| func (s Singer) PrintName = s.Person.PrintName
6| func (s *Singer) PrintName = (*s).Person.PrintName
7| func (s *Singer) SetAge = (&(*s).Person).SetAge
```

The right parts are the corresponding full form selectors.

From the article [methods in Go](#) (§22), we know that we can't explicitly declare methods for unnamed struct types and unnamed pointer types whose base types are unnamed struct types. But

through type embedding, such unnamed types can also own methods.

If a struct type embeds a type which implements an interface type (the embedded type may be the interface type itself), then generally the struct type also implements the interface type, exception there is a method specified by the interface type shadowed by or colliding other methods or fields. For example, in the above example program, both the embedding struct type and the pointer type whose base type is the embedding struct type implement the interface type `I`.

Please note, a type will only obtain the methods of the types it embeds directly or indirectly. In other words, the method set of a type is composed of the methods declared directly (either explicitly or implicitly) for the type and the method set of the type's underlying type. For example, in the following code,

- the type `Age` has no methods, for it doesn't embed any types.
- the type `X` has two methods, `IsOdd` and `Double`. `IsOdd` is obtained by embedding the type `MyInt`.
- the type `Y` has no methods, for its embedded the type `Age` has not methods.
- the type `Z` has only one method, `IsOdd`, which is obtained by embedding the type `MyInt`. It doesn't obtain the method `Double` from the type `X`, for it doesn't embed the type `X`.

```

1| type MyInt int
2| func (mi MyInt) IsOdd() bool {
3|     return mi%2 == 1
4|
5|
6| type Age MyInt
7|
8| type X struct {
9|     MyInt
10| }
11| func (x X) Double() MyInt {
12|     return x.MyInt + x.MyInt
13| }
14|
15| type Y struct {
16|     Age
17| }
18|
19| type Z X

```

Normalization and Evaluation of Promoted Method Values

Assume `v.m` is a legal promoted method value expression, compilers will normalize it as the result of changing implicit address taking and pointer dereference operations into explicit ones in the corresponding full form selector of `v.m`.

The same as any other [method value evaluation](#) (§22), for a normalized method value expression `v.m`, at run time, when the method value `v.m` is evaluated, the receiver argument `v` is evaluated and a copy of the evaluation result is saved and used in later calls to the method value.

For example, in the following code

- the full form selector of the promoted method expression `s.M1` is `s.T.X.M1`. After changing the implicit address taking and pointer dereference operations in it, it becomes `(*s.T).X.M1`. At run time, the receiver argument `(*s.T).X` is evaluated and a copy of the evaluation result is saved and used in later calls to the promoted method value. The evaluation result is `1`, that is why the call `f()` always prints `1`.
- the full form selector of the promoted method expression `s.M2` is `s.T.X.M2`. After changing the implicit address taking and pointer dereference operations in it, it becomes `(&(*s.T).X).M2`. At run time, the receiver argument `&(*s.T).X` is evaluated and a copy of the evaluation result is saved and used in later calls to the promoted method value. The evaluation result is the address of the field `s.X` (a.k.a. `(*s.T).X`). Any change of the value `s.X` will be reflected through the dereference of the address, but the changes of the value `s.T` have no effects on the evaluation result, that is why the two `g()` calls both print `2`.

```

1| package main
2|
3| import "fmt"
4|
5| type X int
6|
7| func (x X) M1() {
8|     fmt.Println(x)
9| }
10|
11| func (x *X) M2() {
12|     fmt.Println(*x)
13| }
14|
15| type T struct { X }
16|
17| type S struct { *T }
18|
19| func main() {

```

```

20|     var t = &T{X: 1}
21|     var s = S{T: t}
22|     var f = s.M1 // <=> (*s.T).X.M1
23|     var g = s.M2 // <=> (&(*s.T).X).M2
24|     s.X = 2
25|     f() // 1
26|     g() // 2
27|     s.T = &T{X: 3}
28|     f() // 1
29|     g() // 2
30| }
```

Interface Types Embed Interface Types

Not only can struct types embed other types, but also can interface types, but interface types can only embed interface types. Please read [interfaces in Go](#) (§23) for details.

An Interesting Type Embedding Example

In the end, let's view an interesting example. The example program will dead loop and stack overflow. If you have understood the above contents and [polymorphism](#) (§23) and type embedding, it is easy to understand why it will dead loop.

```

1| package main
2|
3| type I interface {
4|     m()
5| }
6|
7| type T struct {
8|     I
9| }
10|
11| func main() {
12|     var t T
13|     var i = &t
14|     t.I = i
15|     i.m() // will call t.m(), then call i.m() again, ...
16| }
```

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Type-Unsafe Pointers

We have learned Go pointers from the article [pointers in Go](#) (§15). From that article, we know that, comparing to C pointers, there are many restrictions made for Go pointers. For example, Go pointers can't participate arithmetic operations, and for two arbitrary pointer types, it is very possible that their values can't be converted to each other.

The pointers explained in that article are called type-safe pointers actually. Although the restrictions on type-safe pointers really make us be able to write safe Go code with ease, they also make some obstacles to write efficient code for some scenarios.

In fact, Go also supports type-unsafe pointers, which are pointers without the restrictions made for safe pointers. Type-unsafe pointers are also called unsafe pointers in Go. Go unsafe pointers are much like C pointers, they are powerful, and also dangerous. For some cases, we can write more efficient code with the help of unsafe pointers. On the other hand, by using unsafe pointers, it is easy to write bad code which is too subtle to detect in time.

Another big risk of using unsafe pointers comes from the fact that the unsafe mechanism is not protected by [the Go 1 compatibility guidelines](#). Code depending on unsafe pointers works today could break since a later Go version.

If you really desire the code efficient improvements by using unsafe pointers for any reason, you should not only know the above mentioned risks, but also follow the instructions written in the official Go documentation and clearly understand the effect of each unsafe pointer use, so that you can write safe Go code with unsafe pointers.

About the `unsafe` Standard Package

Go provides a special [kind of types](#) (§14) for unsafe pointers. We must import [the unsafe standard package](#) to use unsafe pointers. The `unsafe.Pointer` type is defined as

```
type Pointer *ArbitraryType
```

Surely, it is not a usual type definition. Here the `ArbitraryType` just hints that a `unsafe.Pointer` value can be converted to any safe pointer values in Go (and vice versa). In other words, `unsafe.Pointer` is like the `void*` in C language.

Go unsafe pointers mean the types whose underlying types are `unsafe.Pointer`.

The zero values of unsafe pointers are also represented with the predeclared identifier `nil`.

Before Go 1.17, the `unsafe` standard package has already provided three functions.

- `func Alignof(variable ArbitraryType) uintptr`, which is used to get the address alignment of a value. Please note, the aligns for struct-field values and non-field values of the same type may be different, though for the standard Go compiler, they are always the same. For the `gccgo` compiler, they may be different.
- `func Offsetof(selector ArbitraryType) uintptr`, which is used to get the address offset of a field in a struct value. The offset is relative to the address of the struct value. The return results should be always the same for the same corresponding field of values of the same struct type in the same program.
- `func Sizeof(variable ArbitraryType) uintptr`, which is used to get the size of a value (a.k.a., the size of the type of the value). The return results should be always the same for all values of the same type in the same program.

Note,

- the types of the return results of the three functions are all `uintptr`. Below we will learn that `uintptr` values can be converted to unsafe pointers (and vice versa).
- although the return results of calls of any of the three functions are consistent in the same program, they might be different across operating systems, across architectures, across compilers, and across compiler versions.
- calls to the three functions are always evaluated at compile time. The evaluation results are typed constants with type `uintptr`.
- the argument passed to a call to the `unsafe.Offsetof` function must be the struct field selector from `value.field`. The selector may denote an embedded field, but the field must be reachable without implicit pointer indirections.

An example of using the three functions.

```

1| package main
2|
3| import "fmt"
4| import "unsafe"
5|
6| func main() {
7|     var x struct {
8|         a int64
9|         b bool
10|        c string
11|    }

```

```

12|     const M, N = unsafe.Sizeof(x.c), unsafe.Sizeof(x)
13|     fmt.Println(M, N) // 16 32
14|
15|     fmt.Println(unsafe.Alignof(x.a)) // 8
16|     fmt.Println(unsafe.Alignof(x.b)) // 1
17|     fmt.Println(unsafe.Alignof(x.c)) // 8
18|
19|     fmt.Println(unsafe.Offsetof(x.a)) // 0
20|     fmt.Println(unsafe.Offsetof(x.b)) // 8
21|     fmt.Println(unsafe.Offsetof(x.c)) // 16
22| }
```

An example which demonstrates the last note mentioned above.

```

1| package main
2|
3| import "fmt"
4| import "unsafe"
5|
6| func main() {
7|     type T struct {
8|         c string
9|     }
10|    type S struct {
11|        b bool
12|    }
13|    var x struct {
14|        a int64
15|        *S
16|        T
17|    }
18|
19|    fmt.Println(unsafe.Offsetof(x.a)) // 0
20|
21|    fmt.Println(unsafe.Offsetof(x.S)) // 8
22|    fmt.Println(unsafe.Offsetof(x.T)) // 16
23|
24|    // This line compiles, for c can be reached
25|    // without implicit pointer indirections.
26|    fmt.Println(unsafe.Offsetof(x.c)) // 16
27|
28|    // This line doesn't compile, for b must be
29|    // reached with the implicit pointer field S.
30|    //fmt.Println(unsafe.Offsetof(x.b)) // error
31| }
```

```

32| // This line compiles. However, it prints
33| // the offset of field b in the value x.S.
34| fmt.Println(unsafe.Offsetof(x.S.b)) // 0
35| }
```

Please note, the print results shown in the comments are for the standard Go compiler version 1.21.n on Linux AMD64 architecture.

The three functions provided in the `unsafe` package don't look much dangerous. The signatures of these functions are very [impossible to be changed in future Go 1 versions](#) [↳](#). Rob Pike even ever [made a proposal to move the three functions to elsewhere](#) [↳](#). Most of the unsafety of the `unsafe` package comes from unsafe pointers. They are as dangerous as C pointers, what is Go safe pointers always try to avoid.

Go 1.17 introduces one new type and two new functions into the `unsafe` package. The new type is `IntegerType`, The following is its definition. This type doesn't denote a specified type. It just represents any arbitrary integer type. We can view it as a generic type.

```
type IntegerType int
```

The two functions introduced in Go 1.17 are:

- `func Add(ptr Pointer, len IntegerType) Pointer`. This function adds an offset to the address represented by an unsafe pointer and return a new unsafe pointer which represents the new address. This function partially covers the usages of the below introduced unsafe pointer use pattern 3.
- `func Slice(ptr *ArbitraryType, len IntegerType) []ArbitraryType`. This function is used to derive a slice with the specified length from a safe pointer, where `ArbitraryType` is the element type of the result slice.

Go 1.20 further introduced three more functions:

- `func String(ptr *byte, len IntegerType) string`. This function is used to derive a string with the specified length from a safe `byte` pointer.
- `func StringData(str string) *byte`. This function is used to get the pointer to the underlying byte sequence of a string. Please note, don't pass empty strings as arguments to this function.
- `func SliceData(slice []ArbitraryType) *ArbitraryType`. This function is used to get the pointer to the underlying element sequence of a slice.

These functions introduced since Go 1.17 have certain dangerousness. They need to be used with caution. This following is an example using the two functions introduced in Go 1.17.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "unsafe"
6| )
7|
8| func main() {
9|     a := [16]int{3: 3, 9: 9, 11: 11}
10|    fmt.Println(a)
11|    eleSize := int(unsafe.Sizeof(a[0]))
12|    p9 := &a[9]
13|    up9 := unsafe.Pointer(p9)
14|    p3 := (*int)(unsafe.Add(up9, -6 * eleSize))
15|    fmt.Println(*p3) // 3
16|    s := unsafe.Slice(p9, 5)[:3]
17|    fmt.Println(s) // [9 0 11]
18|    fmt.Println(len(s), cap(s)) // 3 5
19|
20|    t := unsafe.Slice((*int)(nil), 0)
21|    fmt.Println(t == nil) // true
22|
23|    // The following two calls are dangerous.
24|    // They make the results reference
25|    // unknown memory blocks.
26|    _ = unsafe.Add(up9, 7 * eleSize)
27|    _ = unsafe.Slice(p9, 8)
28| }

```

The following two functions may be used to do conversions between strings and byte slices, in type unsafe manners. Comparing with type safe manners, the type unsafe manners don't duplicate underlying byte sequences of strings and byte slices, so they are more performant.

```

1| import "unsafe"
2|
3| func String2ByteSlice(str string) []byte {
4|     if str == "" {
5|         return nil
6|     }
7|     return unsafe.Slice(unsafe.StringData(str), len(str))
8| }
9|
10| func ByteSlice2String(bs []byte) string {
11|     if len(bs) == 0 {
12|         return ""

```

```

13|     }
14|     return unsafe.String(unsafe.SliceData(bs), len(bs))
15|

```

Unsafe Pointers Related Conversion Rules

Currently (Go 1.21), Go compilers allow the following explicit conversions.

- A safe pointer can be explicitly converted to an unsafe pointer, and vice versa.
- An uintptr value can be explicitly converted to an unsafe pointer, and vice versa. But please note, a nil unsafe.Pointer shouldn't be converted to uintptr and back with arithmetic.

By using these conversions, we can convert a safe pointer value to an arbitrary safe pointer type.

However, although these conversions are all legal at compile time, not all of them are valid (safe) at run time. These conversions defeat the memory safety the whole Go type system (except the unsafe part) tries to maintain. We must follow the instructions listed in a later section below to write valid Go code with unsafe pointers.

Some Facts in Go We Should Know

Before introducing the valid unsafe pointer use patterns, we need to know some facts in Go.

Fact 1: unsafe pointers are pointers and uintptr values are integers

Each of non-nil safe and unsafe pointers references another value. However uintptr values don't reference any values, they are just plain integers, though often each of them stores an integer which can be used to represent a memory address.

Go is a language supporting automatic garbage collection. When a Go program is running, Go runtime will [check which memory blocks are not used by any value any more and collect the memory](#) (§43) allocated for these unused blocks, from time to time. Pointers play an important role in the check process. If a memory block is unreachable from (referenced by) any values still in use, then Go runtime thinks it is an unused value and it can be safely garbage collected.

As uintptr values are integers, they can participate arithmetic operations.

The example in the next subsection shows the differences between pointers and uintptr values.

Fact 2: unused memory blocks may be collected at any time

At run time, the garbage collector may run at an uncertain time, and each garbage collection process may last an uncertain duration. So when a memory block becomes unused, it may be [collected at an uncertain time](#) (§43).

For example:

```

1| import "unsafe"
2|
3| // Assume createInt will not be inlined.
4| //go:noinline
5| func createInt() *int {
6|     return new(int)
7| }
8|
9| func foo() {
10|     p0, y, z := createInt(), createInt(), createInt()
11|     var p1 = unsafe.Pointer(y)
12|     var p2 = uintptr(unsafe.Pointer(z))
13|
14|     // At the time, even if the address of the int
15|     // value referenced by z is still stored in p2,
16|     // the int value has already become unused, so
17|     // garbage collector can collect the memory
18|     // allocated for it now. On the other hand, the
19|     // int values referenced by p0 and p1 are still
20|     // in use.
21|
22|     // uintptr can participate arithmetic operations.
23|     p2 += 2; p2--; p2--
24|
25|     *p0 = 1                         // okay
26|     *(*int)(p1) = 2                 // okay
27|     *(*int)(unsafe.Pointer(p2)) = 3 // dangerous!
28| }
```

In the above example, the fact that value `p2` is still in use can't guarantee that the memory block ever hosting the `int` value referenced by `z` has not been garbage collected yet. In other words, when `*(*int)(unsafe.Pointer(p2)) = 3` is executed, the memory block may be collected, or not. It is dangerous to dereference the address stored in value `p2` to an `int` value, for it is possible that the memory block has been already reallocated for another value (even for another program).

Fact 3: the addresses of some values might change at run time

Please read the article [memory blocks](#) (§43) for details (see the end of the hyperlinked section). Here, we should just know that when the size of the stack of a goroutine changes, the memory blocks allocated on the stack will be moved. In other words, the addresses of the values hosted on these memory blocks will change.

Fact 4: the life range of a value at run time may be not as large as it looks in code

In the following example, the fact value `t` is still in use can't guarantee that the values referenced by value `t.y` are still in use.

```

1| type T struct {
2|     x int
3|     y *[1<<23]byte
4| }
5|
6| func bar() {
7|     t := T{y: new([1<<23]byte)}
8|     p := uintptr(unsafe.Pointer(&t.y[0]))
9|
10|    ... // use T.x and T.y
11|
12|    // A smart compiler can detect that the value
13|    // t.y will never be used again and think the
14|    // memory block hosting t.y can be collected now.
15|
16|    // Using *(*byte)(unsafe.Pointer(p)) is
17|    // dangerous here.
18|
19|    // Continue using value t, but only use its x field.
20|    println(t.x)
21| }
```

Fact 5: `*unsafe.Pointer` is a general safe pointer type

Yes, `*unsafe.Pointer` is a safe pointer type. Its base type is `unsafe.Pointer`. As it is a safe pointer, according the conversion rules listed above, it can be converted to `unsafe.Pointer` type, and vice versa.

For example:

```

1| package main
2|
3| import "unsafe"
4|
5| func main() {
6|     x := 123           // of type int
7|     p := unsafe.Pointer(&x) // of type unsafe.Pointer
8|     pp := &p           // of type *unsafe.Pointer
9|     p = unsafe.Pointer(pp)
10|    pp = (*unsafe.Pointer)(p)
11| }
```

How to Use Unsafe Pointers Correctly?

The `unsafe` standard package documentation lists [six unsafe pointer use patterns](#). Following will introduce and explain them one by one.

Pattern 1: convert a `*T1` value to `unsafe.Pointer`, then convert the `unsafe pointer` value to `*T2`.

As mentioned above, by using the unsafe pointer conversion rules above, we can convert a value of `*T1` to type `*T2`, where `T1` and `T2` are two arbitrary types. However, we should only do such conversions if the size of `T1` is no smaller than `T2`, and only if the conversions are meaningful.

As a result, we can also achieve the conversions between type `T1` and `T2` by using this pattern.

One example is the `math.Float64bits` function, which converts a `float64` value to an `uint64` value, without changing any bit in the `float64` value. The `math.Float64frombits` function does reverse conversions.

```

1| func Float64bits(f float64) uint64 {
2|     return *(*uint64)(unsafe.Pointer(&f))
3| }
4|
5| func Float64frombits(b uint64) float64 {
6|     return *(*float64)(unsafe.Pointer(&b))
7| }
```

Please note, the return result of the `math.Float64bits(aFloat64)` function call is different from the result of the explicit conversion `uint64(aFloat64)`.

In the following example, we use this pattern to convert a `[]MyString` slice to type `[]string`, and vice versa. The result slice and the original slice share the underlying elements. Such conversions are impossible through safe ways,

```

1| package main
2|
3| import (
4|     "fmt"
5|     "unsafe"
6| )
7|
8| func main() {
9|     type MyString string
10|    ms := []MyString{"C", "C++", "Go"}
11|    fmt.Printf("%s\n", ms) // [C C++ Go]
12|    // ss := ([]string)(ms) // compiling error
13|    ss := *(*[]string)(unsafe.Pointer(&ms))
14|    ss[1] = "Zig"
15|    fmt.Printf("%s\n", ms) // [C Zig Go]
16|    // ms = []MyString(ss) // compiling error
17|    ms = *(*[]MyString)(unsafe.Pointer(&ss))
18|
19|    // Since Go 1.17, we may also use the
20|    // unsafe.Slice function to do the conversions.
21|    ss = unsafe.Slice((*string)(&ms[0]), len(ms))
22|    ms = unsafe.Slice((*MyString)(&ss[0]), len(ss))
23| }
```

By the way, since Go 1.17, we may also use the `unsafe.Slice` function to do the conversions:

```

1| func main() {
2|     ...
3|
4|     ss = unsafe.Slice((*string)(&ms[0]), len(ms))
5|     ms = unsafe.Slice((*MyString)(&ss[0]), len(ss))
6| }
```

A practice by using the pattern is to convert a byte slice, which will not be used after the conversion, to a string, as the following code shows. In this conversion, a duplication of the underlying byte sequence is avoided.

```

1| func ByteSlice2String(bs []byte) string {
2|     return *(*string)(unsafe.Pointer(&bs))
3|

```

This is the implementation adopted by the `String` method of the `Builder` type supported since Go 1.10 in the `strings` standard package. The size of a byte slice is larger than a string, and [their internal structures](#) (§17) are similar, so the conversion is valid (for main stream Go compilers). However, despite the implementation may be safely used in standard packages now, it is not recommended to be used in general user code. Since Go 1.20, in general user code, we should try to use the implementation which uses the `unsafe.String` function, mentioned above in this article.

The converse, converting a string to a byte slice in the similar way, is invalid, for the size of a string is smaller than a byte slice.

```

1| func String2ByteSlice(s string) []byte {
2|     return *(*[]byte)(unsafe.Pointer(&s)) // dangerous!
3|

```

In the pattern 6 section below, a valid implementation to do the same job is introduced.

Note: when using the just introduced unsafe way to convert a byte slice to a string, please make sure not to modify the bytes in the byte slice if the result string still survives.

Pattern 2: convert unsafe pointer to uintptr, then use the uintptr value.

This pattern is not very useful. Usually, we print the result `uintptr` values to check the memory addresses stored in them. However, there are other both safe and less verbose ways to this job. So this pattern is not much useful.

Example:

```

1| package main
2|
3| import "fmt"
4| import "unsafe"
5|
6| func main() {
7|     type T struct{a int}
8|     var t T
9|     fmt.Printf("%p\n", &t)           // 0xc6233120a8
10|    println(&t)                  // 0xc6233120a8

```

```

11|     fmt.Printf("%x\n", uintptr(unsafe.Pointer(&t))) // c6233120a8
12|

```

The outputted addresses might be different for each run.

Pattern 3: convert unsafe pointer to uintptr, do arithmetic operations with the uintptr value, then convert it back

In this pattern, the result unsafe pointer must continue to point into the original allocated memory block. For example:

```

1| package main
2|
3| import "fmt"
4| import "unsafe"
5|
6| type T struct {
7|     x bool
8|     y [3]int16
9| }
10|
11| const N = unsafe.Offsetof(T{}.y)
12| const M = unsafe.Sizeof(T{}.y[0])
13|
14| func main() {
15|     t := T{y: [3]int16{123, 456, 789}}
16|     p := unsafe.Pointer(&t)
17|     // "uintptr(p)+N+M+M" is the address of t.y[2].
18|     ty2 := (*int16)(unsafe.Pointer(uintptr(p)+N+M+M))
19|     fmt.Println(*ty2) // 789
20|

```

In fact, since Go 1.17, it is more recommended to use the above introduced `unsafe.Add` function to do such address offset operations.

Please note, in this specified example, the conversion `unsafe.Pointer(uintptr(p) + N + M + M)` shouldn't be split into two lines, like the following code shows. Please read the comments in the code for the reason.

```

1| func main() {
2|     t := T{y: [3]int16{123, 456, 789}}
3|     p := unsafe.Pointer(&t)
4|     // ty2 := (*int16)(unsafe.Pointer(uintptr(p)+N+M+M))

```

```

5|     addr := uintptr(p) + N + M + M
6|
7|     // ... (some other operations)
8|
9|     // Now the t value becomes unused, its memory may be
10|    // garbage collected at this time. So the following
11|    // use of the address of t.y[2] may become invalid
12|    // and dangerous!
13|    // Another potential danger is, if some operations
14|    // make the stack grow or shrink here, then the
15|    // address of t might change, so that the address
16|    // saved in addr will become invalid (fact 3).
17|    ty2 := (*int16)(unsafe.Pointer(addr))
18|    fmt.Println(*ty2)
19| }
```

Such bugs are very subtle and hard to detect, which is why the uses of unsafe pointers are dangerous.

The intermediate uintptr value may also participate in &^ bitwise clear operations to do address alignment, as long as the result unsafe pointer and the original one point into the same allocated memory block.

Another detail which should be also noted is that, it is not recommended to store the end boundary of a memory block in a pointer (either safe or unsafe one). Doing this will prevent another memory block which closely follows the former memory block from being garbage collected, or crash program if that boundary address is not valid for any allocated memory blocks (depending on compiler implementations). Please read [this FAQ item](#) (§51) to get more explanations.

Pattern 4: convert unsafe pointers to uintptr values as arguments of syscall.Syscall calls.

From the explanations for the last pattern, we know that the following function is dangerous.

```

1| // Assume this function will not inlined.
2| func DoSomething(addr uintptr) {
3|     // read or write values at the passed address ...
4| }
```

The reason why the above function is dangerous is that the function itself can't guarantee the memory block at the passed argument address is not garbage collected yet. If the memory block is

collected or is reallocated for other values, then the operations made in the function body are dangerous.

However, the prototype of the `Syscall` function in the `syscall` standard package is as

```
func Syscall(trap, a1, a2, a3 uintptr) (r1, r2 uintptr, err Errno)
```

How does this function guarantee that the memory blocks at the passed addresses `a1`, `a2` and `a3` are still not garbage collected yet within the function internal? The function can't guarantee this. In fact, compilers will make the guarantee. It is the privilege of calls to `syscall.Syscall` alike functions.

We can think that, compilers will automatically insert some instructions for each of the unsafe pointer arguments who are converted to `uintptr`, like the third argument in the following `syscall.Syscall` call, to prevent the memory block referenced by that argument from being garbage collected or moved.

Please note that, before Go 1.15, it was okay the conversion expressions `uintptr(anUnsafePointer)` act as sub-expressions of the talked arguments. Since Go 1.15, the requirement becomes a bit stricter: the talked arguments must present exactly as the `uintptr(anUnsafePointer)` form.

The following call is safe:

```
1| syscall.Syscall(SYS_READ, uintptr(fd),
2|                   uintptr(unsafe.Pointer(p)), uintptr(n))
```

But the following calls are dangerous:

```
1| u := uintptr(unsafe.Pointer(p))
2| // At this time, the value referenced by p might
3| // have become unused and been collected already,
4| // or the address of the value has changed.
5| syscall.Syscall(SYS_READ, uintptr(fd), u, uintptr(n))
6|
7| // Arguments must be in the "uintptr(anUnsafePointer)"
8| // form. In fact, the call was safe before Go 1.15.
9| // But Go 1.15 changes the rule a bit.
10| syscall.Syscall(SYS_XXX, uintptr(uintptr(fd)),
11|                  uint(uintptr(unsafe.Pointer(p))), uintptr(n))
```

Note: this pattern also applies to the [syscall.Proc.Call](#) and [syscall.LazyProc.Call](#) methods on Windows.

Again, never use this pattern when calling other functions.

Pattern 5: convert the `uintptr` result of `reflect.Value.Pointer` or `reflect.Value.UnsafeAddr` method call to unsafe pointer

The methods `Pointer` and `UnsafeAddr` of the `Value` type in the `reflect` standard package both return a result of type `uintptr` instead of `unsafe.Pointer`. This is a deliberate design, which is to avoid converting the results of calls (to the two methods) to any safe pointer types without importing the `unsafe` standard package.

The design requires the return result of a call to either of the two methods must be converted to an unsafe pointer immediately after making the call. Otherwise, there will be small time window in which the memory block allocated at the address stored in the result might lose all references and be garbage collected.

For example, the following call is safe.

```
p := (*int)(unsafe.Pointer(reflect.ValueOf(new(int)).Pointer()))
```

On the other hand, the following call is dangerous.

```
1| u := reflect.ValueOf(new(int)).Pointer()
2| // At this moment, the memory block at the address
3| // stored in u might have been collected already.
4| p := (*int)(unsafe.Pointer(u))
```

Please note that, Go 1.19 introduces a new method, `reflect.Value.UnsafePointer()`, which returns a `unsafe.Pointer` value and is preferred over the two just mentioned functions. That means, the old deliberate design is thought as not good now.

Pattern 6: convert a `reflect.SliceHeader.Data` or `reflect.StringHeader.Data` field to unsafe pointer, and the inverse.

For the same reason mentioned for the last subsection, the `Data` fields of the struct type `SliceHeader` and `StringHeader` in the `reflect` standard package are declared with type `uintptr` instead of `unsafe.Pointer`.

We can convert a string pointer to a `*reflect.StringHeader` pointer value, so that we can manipulate the internal of the string. The same, we can convert a slice pointer to a `*reflect.SliceHeader` pointer value, so that we can manipulate the internal of the slice.

An example of using `reflect.StringHeader`:

```

1| package main
2|
3| import "fmt"
4| import "unsafe"
5| import "reflect"
6|
7| func main() {
8|     a := [...]byte{'G', 'o', 'l', 'a', 'n', 'g'}
9|     s := "Java"
10|    hdr := (*reflect.StringHeader)(unsafe.Pointer(&s))
11|    hdr.Data = uintptr(unsafe.Pointer(&a))
12|    hdr.Len = len(a)
13|    fmt.Println(s) // Golang
14|    // Now s and a share the same byte sequence, which
15|    // makes the bytes in the string s become mutable.
16|    a[2], a[3], a[4], a[5] = 'o', 'g', 'l', 'e'
17|    fmt.Println(s) // Google
18| }
```

An example of using `reflect.SliceHeader`:

```

1| package main
2|
3| import (
4|     "fmt"
5|     "unsafe"
6|     "reflect"
7| )
8|
9| func main() {
10|     a := [6]byte{'G', 'o', 'l', 'a', 'n', 'g'}
11|     bs := []byte("Golang")
12|     hdr := (*reflect.SliceHeader)(unsafe.Pointer(&bs))
13|     hdr.Data = uintptr(unsafe.Pointer(&a))
14|
15|     hdr.Len = 2
16|     hdr.Cap = len(a)
17|     fmt.Printf("%s\n", bs) // Go
18|     bs = bs[:cap(bs)]
```

```

19|     fmt.Printf("%s\n", bs) // Go101
20| }
```

In general, we should only get a `*reflect.StringHeader` pointer value from an actual (already existed) string, or get a `*reflect.SliceHeader` pointer value from an actual (already existed) slice. We shouldn't do the contrary, such as creating a string from a new allocated `StringHeader`, or creating a slice from a new allocated `SliceHeader`. For example, the following code is dangerous.

```

1| var hdr reflect.StringHeader
2| hdr.Data = uintptr(unsafe.Pointer(new([5]byte)))
3| // Now the just allocated byte array has lose all
4| // references and it can be garbage collected now.
5| hdr.Len = 5
6| s := *(*string)(unsafe.Pointer(&hdr)) // dangerous!
```

The following is an example which shows how to convert a string to a byte slice, by using the unsafe way. Different from the safe conversion from a string to a byte slice, the unsafe way doesn't allocate a new underlying byte sequence for the result slice in each conversion.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "reflect"
6|     "strings"
7|     "unsafe"
8| )
9|
10| func String2ByteSlice(str string) (bs []byte) {
11|     strHdr := (*reflect.StringHeader)(unsafe.Pointer(&str))
12|     sliceHdr := (*reflect.SliceHeader)(unsafe.Pointer(&bs))
13|     sliceHdr.Data = strHdr.Data
14|     sliceHdr.Cap = strHdr.Len
15|     sliceHdr.Len = strHdr.Len
16|     return
17| }
18|
19| func main() {
20|     // str := "Golang"
21|     // For the official standard compiler, the above
22|     // line will make the bytes in str allocated on
23|     // an immutable memory zone.
24|     // So we use the following line instead.
```

```

25|     str := strings.Join([]string{"Go", "land"}, "")
26|     s := String2ByteSlice(str)
27|     fmt.Printf("%s\n", s) // Golang
28|     s[5] = 'g'
29|     fmt.Println(str) // Golang
30| }
```

Note, when using the just introduced unsafe way to convert a string to a byte slice, please make sure not to modify the bytes in the result byte slice if the string still survives (for demonstration purpose, the above example violates this principle).

It is also possible to convert a byte slice to a string in a similar way, which is a bit safer (but a bit slower) than the way shown in pattern 1.

```

1| func ByteSlice2String(bs []byte) (str string) {
2|     sliceHdr := (*reflect.SliceHeader)(unsafe.Pointer(&bs))
3|     strHdr := (*reflect.StringHeader)(unsafe.Pointer(&str))
4|     strHdr.Data = sliceHdr.Data
5|     strHdr.Len = sliceHdr.Len
6|     return
7| }
```

Similarly, please make sure not to modify the bytes in the argument byte slice if the result string still survives.

BTW, let's view a bad example which violates the principle of pattern 3 (the example is borrowed from one slack comment posted by Bryan C. Mills):

```

1| package main
2|
3| import (
4|     "fmt"
5|     "reflect"
6|     "unsafe"
7| )
8|
9| func Example_Bad() *byte {
10|     var str = "godoc"
11|     hdr := (*reflect.StringHeader)(unsafe.Pointer(&str))
12|     pbyte := (*byte)(unsafe.Pointer(hdr.Data + 2))
13|     return pbyte // *pbyte == 'd'
14| }
15|
16| func main() {
```

```
17|     fmt.Println(string(*Example_Bad()))
18| }
```

Two correct implementations:

```
1| func Example_Good1() *byte {
2|     var str = "godoc"
3|     hdr := (*reflect.StringHeader)(unsafe.Pointer(&str))
4|     pbyte := (*byte)(unsafe.Pointer(
5|         uintptr(unsafe.Pointer(hdr.Data)) + 2))
6|     return pbyte
7| }
8|
9| // Works since Go 1.17.
10| func Example_Good2() *byte {
11|     var str = "godoc"
12|     hdr := (*reflect.StringHeader)(unsafe.Pointer(&str))
13|     pbyte := (*byte)(unsafe.Add(unsafe.Pointer(hdr.Data), 2))
14|     return pbyte
15| }
```

Tricky? Yes.

The docs [↗](#) of the `SliceHeader` and `StringHeader` types in the `reflect` standard package are similar in that they say the representations of the two struct types may change in a later release. So the above valid examples using the two types may become invalid even if the unsafe rules keep unchanged. Fortunately, at present (Go 1.21), the two available mainstream Go compilers (the standard Go compiler and the `gccgo` compiler) both recognize the representations of the two types declared in the `reflect` standard package.

The Go core development team also realized that the two types are inconvenient and error-prone, so the two types have been not recommended any more since Go 1.20 (they have been deprecated since Go 1.21). Instead, we should try to use the `unsafe.String`, `unsafe.StringData`, `unsafe.Slice` and `unsafe.SliceData` functions described earlier in this article.

Final Words

From the above contents, we know that, for some cases, the unsafe mechanism can help us write more efficient Go code. However, it is very easy to introduce some subtle bugs which have very low possibilities to produce when using the unsafe mechanism. A program with these bugs may run well for a long time, but suddenly behave abnormally and even crash at a later time. Such bugs are very hard to detect and debug.

We should only use the `unsafe` mechanism when we have to, and we must use it with extreme care. In particular, we should follow the instructions described above.

And again, we should aware that the unsafe mechanism introduced above may change and even become invalid totally in later Go versions, though no evidences this will happen soon. If the unsafe mechanism rules change, the above introduced valid unsafe pointer use patterns may become invalid. So please keep it easy to switch back to the safe implementations for you code depending on the unsafe mechanism.

In the end, it is worth mentioning that a dynamic analysis compiler option `-gcflags=all=-d=checkptr` is supported since Go Toolchain 1.14 (it is recommended to use this option on Windows with Go Toolchain 1.15+). When this option is used, some (but not all) incorrect unsafe pointer uses will be detected at run time. Once such an incorrect use is detected, a panic will occur. Thanks to Matthew Dempsky for implementing this [great feature](#) !

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Generics

Before Go 1.18, Go only supported built-in generics. Since Go 1.18, Go also supports custom generics. This article only introduces built-in generics.

Go built-in generic types are supported through first-class citizen composite types. We can use composite types to create infinite custom types by using the composite types. This article will show some type composition examples and explain how to read these composed types.

Type Composition Examples

Type compositions in Go are designed very intuitive and easy to interpret. It is hardly to get lost in understanding Go composite types, even if for some very complex ones. The following will list several type composition examples, from simpler ones to more complex ones.

Let's view an simple composite type literal.

```
1| [3][4]int
```

When interpreting a composite type, we should look at it from left to right. The [3] on the left in the above type literal indicates that this type is an array type. The whole right part following the [4]int is another array type, which is the element type of the first array type. The element type of the element type (an array type) of the first array type is built-in type int . The first array type can be viewed as a two-dimensional array type.

An example on using this two-dimensional array type.

```
1| package main
2|
3| import (
4|     "fmt"
5| )
6|
7| func main() {
8|     matrix := [3][4]int{
9|         {1, 0, 0, 1},
10|        {0, 1, 0, 1},
11|        {0, 0, 1, 1},
12|    }
13| }
```

```

14|     matrix[1][1] = 3
15|     a := matrix[1] // type of a is [4]int
16|     fmt.Println(a) // [0 3 0 1]
17| }
```

Similarly,

- `[][]string` is a slice type whose element type is another slice type `[]string`.
- `**bool` is a pointer type whose base type is another pointer type `*bool`.
- `chan chan int` is a channel type whose element type is another channel type `chan int`.
- `map[int]map[int]string` is a map type whose element type is another map type `map[int]string`. The key types of the two map types are both `int`.
- `func(int32) func(int32)` is a function type whose only return result type is another function type `func(int32)`. The two function types both have only one input parameter with type `int32`.

Let's view another type.

```
1| chan *[16]byte
```

The `chan` keyword at the left most indicates this type is a channel type. The whole right part `*[16]byte`, which is a pointer type, denotes the element type of this channel type. The base type of the pointer type is `[16]byte`, which is an array type. The element type of the array type is `byte`.

An example on using this channel type.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "time"
6|     "crypto/rand"
7| )
8|
9| func main() {
10|     c := make(chan *[16]byte)
11|
12|     go func() {
13|         // Use two arrays to avoid data races.
14|         var dataA, dataB = new([16]byte), new([16]byte)
15|         for {
16|             _, err := rand.Read(dataA[:])
17|             if err != nil {
```

```

18|         close(c)
19|     } else {
20|         c <- dataA
21|         dataA, dataB = dataB, dataA
22|     }
23| }
24| }()
25|
26| for data := range c {
27|     fmt.Println((*data)[])
28|     time.Sleep(time.Second / 2)
29| }
30| }
```

Similarly, type `map[string][]func(int) int` is a map type. The key type of this map type is `string`. The remaining right part `[]func(int) int` denotes the element type of the map type. The `[]` indicates the element type is a slice type, whose element type is a function type `func(int) int`.

An example on using the just explained map type.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     addone := func(x int) int {return x + 1}
7|     square := func(x int) int {return x * x}
8|     double := func(x int) int {return x + x}
9|
10|    transforms := map[string][]func(int) int {
11|        "inc,inc,inc": {addone, addone, addone},
12|        "sqr,inc dbl": {square, addone, double},
13|        "dbl,sqr,sqr": {double, double, square},
14|    }
15|
16|    for _, n := range []int{2, 3, 5, 7} {
17|        fmt.Println("">>>>", n)
18|        for name, transfers := range transforms {
19|            result := n
20|            for _, xfer := range transfers {
21|                result = xfer(result)
22|            }
23|            fmt.Printf(" %v: %v \n", name, result)
24|        }
```

```
25|     }
26| }
```

Below is a type which looks some complex.

```
1| []map[struct {
2|     a int
3|     b struct {
4|         x string
5|         y bool
6|     }
7| }]interface {
8|     Build([]byte, struct {x string; y bool}) error
9|     Update(dt float64)
10|    Destroy()
11| }
```

Let's read it from left to right. The starting `[]` at the left most indicates this type is a slice type. The following `map` keyword shows the element type of the slice type is a map type. The `struct` type denoted by the `struct` literal enclosed in the `[]` following the `map` keyword is the key type of the map type. The element type of the map type is an interface type which specifies three methods. The key type, a `struct` type, has two fields, one field `a` is of `int` type, and the other field `b` is of another `struct` type `struct {x string; y bool}`.

Please note that the second `struct` type is also used as one parameter type of one method specified by the just mentioned interface type.

To get a better readability, we often decompose such a type into multiple type declarations. The type alias `T` declared in the following code and the just explained type above denote the identical type.

```
1| type B = struct {
2|     x string
3|     y bool
4| }
5|
6| type K = struct {
7|     a int
8|     b B
9| }
10|
11| type E = interface {
12|     Build([]byte, B) error
13|     Update(dt float64)
14|     Destroy()
```

```
15| }
16|
17| type T = []map[K]E
```

Built-in Generic Functions

Besides the built-in generics for composite types, there are several built-in functions which also support generics. Such as the built-in `len` function can be used to get the length of values of arrays, slices, maps, strings and channels. Generally, the functions in the `unsafe` standard package are also viewed as built-in functions. The built-in generic functions have been introduced in previous articles,

Custom Generics

Since version 1.18, Go has already supported custom generics. Please read the [Go Generics 101](#) book to get how to use custom generics.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Reflections in Go

Go is a static language with well reflection support. The remaining of this article will explain the reflection functionalities provided in the `reflect` standard package.

It is very helpful to read the [overview of Go type system](#) (§14) and [interfaces in Go](#) (§23) articles before reading the remaining of the current article.

Overview of Go Reflection

Go reflection brings many dynamic functionalities to Go programming. Many standard code packages, such as the `fmt` and `encoding` packages, heavily rely on the reflection functionalities.

We can inspect Go values through the values of the `Type` and `Value` types defined in the `reflect` standard package. The remaining of this article will show some examples on how to use values of the two types.

One of the Go reflection design goals is any non-reflection operation should be also possible to be applied through the reflection ways. For all kinds of reasons, this goal is not 100 percent achieved. However, most non-reflection operations can be applied through the reflection ways now. On the other hand, through the reflection ways, we can do some operations which are impossible to be achieved through non-reflection ways. The operations which can't and can only be achieved through the reflection ways will be mentioned in the following sections.

The `reflect.Type` Type and Values

In Go, we can create a `reflect.Type` value from an arbitrary non-interface value by calling the `reflect.TypeOf` function. The result `reflect.Type` value represents the type of the non-interface value. Surely, we can also pass an interface value to a `reflect.TypeOf` function call, but the call will return a `reflect.Type` value which represents the dynamic type of the interface value. In fact, the `reflect.TypeOf` function has only one parameter of type `interface{}` and always returns a `reflect.Type` value which represents the dynamic type of the only interface parameter. Then how to get a `reflect.Type` value which represents an interface type? We must use indirect ways which will be introduced below to achieve this goal.

The `reflect.Type` type is an interface type. It [specifies several methods ↗](#). We can call these methods to inspect the information of the type represented by a `reflect.Type` receiver value. Some of these methods apply for all [kinds of types ↗](#), some of them are one kind or several kinds specific. Please read the documentation of each method for details. Calling one of the methods through an improper `reflect.Type` receiver value will produce a panic.

An example:

```

1| package main
2|
3| import "fmt"
4| import "reflect"
5|
6| func main() {
7|     type A = [16]int16
8|     var c <-chan map[A][]byte
9|     tc := reflect.TypeOf(c)
10|    fmt.Println(tc.Kind()) // chan
11|    fmt.Println(tc.ChanDir()) // <-chan
12|    tm := tc.Elem()
13|    ta, tb := tm.Key(), tm.Elem()
14|    // The next line prints: map array slice
15|    fmt.Println(tm.Kind(), ta.Kind(), tb.Kind())
16|    tx, ty := ta.Elem(), tb.Elem()
17|
18|    // byte is an alias of uint8
19|    fmt.Println(tx.Kind(), ty.Kind()) // int16 uint8
20|    fmt.Println(tx.Bits(), ty.Bits()) // 16 8
21|    fmt.Println(tx.ConvertibleTo(ty)) // true
22|    fmt.Println(tb.ConvertibleTo(ta)) // false
23|
24|    // Slice and map types are incomparable.
25|    fmt.Println(tb.Comparable()) // false
26|    fmt.Println(tm.Comparable()) // false
27|    fmt.Println(ta.Comparable()) // true
28|    fmt.Println(tc.Comparable()) // true
29| }
```

There are [26 kinds of types ↗](#) in Go.

In the above example, we use the method `Elem` to get the element types of some container types (a channel type, a map type, a slice type and an array type). In fact, we can also use this method to get the base type of a pointer type. For example,

```

1| package main
2|
3| import "fmt"
4| import "reflect"
5|
6| type T []interface{m()}
7| func (T) m() {}
8|
9| func main() {
10|     tp := reflect.TypeOf(new(interface{}))
11|     tt := reflect.TypeOf(T{})
12|     fmt.Println(tp.Kind(), tt.Kind()) // ptr slice
13|
14|     // Get two interface Types indirectly.
15|     ti, tim := tp.Elem(), tt.Elem()
16|     // The next line prints: interface interface
17|     fmt.Println(ti.Kind(), tim.Kind())
18|
19|     fmt.Println(tt.Implements(tim)) // true
20|     fmt.Println(tp.Implements(tim)) // false
21|     fmt.Println(tim.Implements(ti)) // true
22|
23|     // All types implement any blank interface type.
24|     fmt.Println(tp.Implements(ti)) // true
25|     fmt.Println(tt.Implements(ti)) // true
26|     fmt.Println(tim.Implements(ti)) // true
27|     fmt.Println(ti.Implements(ti)) // true
28| }
```

The above example also shows how to (indirectly) get a `reflect.Type` value which represents an interface type.

We can get all of the field types (of a struct type) and the method information of a type through reflection. We can also get the parameter and result type information of a function type through reflection.

```

1| package main
2|
3| import "fmt"
4| import "reflect"
5|
6| type F func(string, int) bool
7| func (f F) m(s string) bool {
8|     return f(s, 32)
9| }
```

```

10| func (f F) M() {}
11|
12| type I interface{m(s string) bool; M()}
13|
14| func main() {
15|     var x struct {
16|         F F
17|         i I
18|     }
19|     tx := reflect.TypeOf(x)
20|     fmt.Println(tx.Kind())           // struct
21|     fmt.Println(tx.NumField())      // 2
22|     fmt.Println(tx.Field(1).Name)   // i
23|     // Package path is an intrinsic property of
24|     // non-exported selectors (fields or methods).
25|     fmt.Println(tx.Field(0).PkgPath) //
26|     fmt.Println(tx.Field(1).PkgPath) // main
27|
28|     tf, ti := tx.Field(0).Type, tx.Field(1).Type
29|     fmt.Println(tf.Kind())          // func
30|     fmt.Println(tf.IsVariadic())    // false
31|     fmt.Println(tf.NumIn(), tf.NumOut()) // 2 1
32|     t0, t1, t2 := tf.In(0), tf.In(1), tf.Out(0)
33|     // The next line prints: string int bool
34|     fmt.Println(t0.Kind(), t1.Kind(), t2.Kind())
35|
36|     fmt.Println(tf.NumMethod(), ti.NumMethod()) // 1 2
37|     fmt.Println(tf.Method(0).Name)               // M
38|     fmt.Println(ti.Method(1).Name)               // m
39|     _, ok1 := tf.MethodByName("m")
40|     _, ok2 := ti.MethodByName("m")
41|     fmt.Println(ok1, ok2) // false true
42| }
```

From the above example, we could find that,

1. for non-interface types, the `reflect.Type.NumMethod` only returns the number of exported methods (including implicitly declared ones) of a type. We are unable to get the information of a non-exported method by using the `reflect.Type.MethodByName` method. For interface types, the limits don't exist (the fact was not mentioned in the docs of the two methods before Go 1.16). Such situation also applies to the corresponding methods of the `reflect.Value` type introduced in the next section.
2. although a `reflect.Type.NumField` method call returns the number of all fields (including non-exported ones) of a struct type, it is [not a good idea](#) to use the

`reflect.Type.FieldName` method to get the information of a non-exported field.

We may [inspect struct field tags through reflection](#). The types of struct field tags are `reflect.StructTag`, which has two methods, `Get` and `Lookup`, to inspect the key-value pairs specified in field tags. An example of inspecting struct field tags:

```

1| package main
2|
3| import "fmt"
4| import "reflect"
5|
6| type T struct {
7|     X    int `max:"99" min:"0" default:"0"`
8|     Y, Z bool `optional:"yes"`
9| }
10|
11| func main() {
12|     t := reflect.TypeOf(T{})
13|     x := t.Field(0).Tag
14|     y := t.Field(1).Tag
15|     z := t.Field(2).Tag
16|     fmt.Println(reflect.TypeOf(x)) // reflect.StructTag
17|     // v is a string
18|     v, present := x.Lookup("max")
19|     fmt.Println(len(v), present)      // 2 true
20|     fmt.Println(x.Get("max"))        // 99
21|     fmt.Println(x.Lookup("optional")) // false
22|     fmt.Println(y.Lookup("optional")) // yes true
23|     fmt.Println(z.Lookup("optional")) // yes true
24| }
```

Note,

- tag keys may not contain space (Unicode value 32), quote (Unicode value 34) and colon (Unicode value 58) characters.
- to form a valid key-value pair, no space characters are allowed to follow the semicolon in the supposed key-value pair. So
``optional: "yes"`` doesn't form key-value pairs.
- space characters in tag values are important (not be ignored). So
``json:"author, omitempty"`,`
``json:" author,omitempty"`` and
``json:"author,omitempty"'` are different from each other.
- each struct field tag should present as a single line to be wholly meaningful.

Beside the `reflect.TypeOf` function, we can also use some other functions in the `reflect` standard package to create `reflect.Type` values which represent some unnamed composite types.

```

1| package main
2|
3| import "fmt"
4| import "reflect"
5|
6| func main() {
7|     ta := reflect.ArrayOf(5, reflect.TypeOf(123))
8|     fmt.Println(ta) // [5]int
9|     tc := reflect.ChanOf(reflect.SendDir, ta)
10|    fmt.Println(tc) // chan<- [5]int
11|    tp := reflect.PtrTo(ta)
12|    fmt.Println(tp) // *[5]int
13|    ts := reflect.SliceOf(tp)
14|    fmt.Println(ts) // []*[5]int
15|    tm := reflect.MapOf(ta, tc)
16|    fmt.Println(tm) // map[[5]int]chan<- [5]int
17|    tf := reflect.FuncOf([]reflect.Type{ta},
18|                           []reflect.Type{tp, tc}, false)
19|    fmt.Println(tf) // func([5]int) (*[5]int, chan<- [5]int)
20|    tt := reflect.StructOf([]reflect.StructField{
21|        {Name: "Age", Type: reflect.TypeOf("abc")},
22|    })
23|    fmt.Println(tt)           // struct { Age string }
24|    fmt.Println(tt.NumField()) // 1
25| }
```

There are more `reflect.Type` methods which are not used in above examples, please read the `reflect` package documentation for their usages.

Note, up to now (Go 1.21), there are no ways to create interface types through reflection. This is a known limitation of Go reflection.

Another limitation is, although we can create a struct type embedding other types as anonymous fields through reflection, the struct type may or may not obtain the methods of the embedded types, and creating a struct type with anonymous fields even might panic at run time. In other words, the behavior of creating struct types with anonymous fields is partially compiler dependent.

The third limitation is we can't declare new types through reflection.

The `reflect.Value` Type and Values

Similarly, we can create a `reflect.Value` value from an arbitrary non-interface value by calling the `reflect.ValueOf` function. The result `reflect.Value` value represents the non-interface value. Same as the `reflect.TypeOf` function, the `reflect.ValueOf` function also has only one parameter of type `interface{}`. When an interface argument is passed to a `reflect.ValueOf` function call, the call will return a `reflect.Value` value which represents the dynamic value of the interface argument. To get a `reflect.Value` value which represents an interface value, we must use indirect ways which will be introduced below to achieve this goal.

The value represented by a `reflect.Value` value `v` is often called the underlying value of `v`.

There are [plenty of methods ↗](#) declared for the `reflect.Value` type. We can call these methods to inspect the information of (and manipulate) the underlying value of a `reflect.Value` receiver value. Some of these methods apply for all kinds of values, some of them are one kind or several kinds specific. Please read the `reflect` standard package documentation for details. Calling a kind-specific method with an improper `reflect.Value` receiver value will produce a panic.

The `CanSet` method of a `reflect.Value` value returns whether or not the underlying value of the `reflect.Value` value is modifiable (can be assigned to). If the Go value is modifiable, we can call the `Set` method of the corresponding `reflect.Value` value to modify the Go value. Note, the `reflect.Value` values returned directly by `reflect.ValueOf` function calls are always read-only.

An example:

```

1| package main
2|
3| import "fmt"
4| import "reflect"
5|
6| func main() {
7|     n := 123
8|     p := &n
9|     vp := reflect.ValueOf(p)
10|    fmt.Println(vp.CanSet(), vp.CanAddr()) // false false
11|    vn := vp.Elem() // get the value referenced by vp
12|    fmt.Println(vn.CanSet(), vn.CanAddr()) // true true
13|    vn.Set(reflect.ValueOf(789)) // <=> vn.SetInt(789)
14|    fmt.Println(n)           // 789
15| }
```

Non-exported fields of struct values can't be modified through reflections.

```

1| package main
2|
3| import "fmt"
4| import "reflect"
5|
6| func main() {
7|     var s struct {
8|         X interface{} // an exported field
9|         y interface{} // a non-exported field
10|    }
11|    vp := reflect.ValueOf(&s)
12|    // If vp represents a pointer, the following
13|    // line is equivalent to "vs := vp.Elem()".
14|    vs := reflect.Indirect(vp)
15|    // vx and vy both represent interface values.
16|    vx, vy := vs.Field(0), vs.Field(1)
17|    fmt.Println(vx.CanSet(), vx.CanAddr()) // true true
18|    // vy is addressable but not modifiable.
19|    fmt.Println(vy.CanSet(), vy.CanAddr()) // false true
20|    vb := reflect.ValueOf(123)
21|    vx.Set(vb)      // okay, for vx is modifiable
22|    // vy.Set(vb) // will panic, for vy is unmodifiable
23|    fmt.Println(s) // {123 <nil>}
24|    fmt.Println(vx.IsNil(), vy.IsNil()) // false true
25| }
```

From the above two examples, we can learn that there are two ways to get a `reflect.Value` value whose underlying value is referenced by the underlying value (a pointer value) of another `reflect.Value` value.

1. One way is by calling the `Elem` method of a `reflect.Value` value which represents the pointer value.
2. The other way is to pass a `reflect.Value` value which represents the pointer value to a `reflect.Indirect` function call. (If the argument passed to a `reflect.Indirect` function call doesn't represent a pointer value, then the call returns a copy of the argument.)

Note, the `reflect.Value.Elem` method can be also used to get a `reflect.Value` value which represents the dynamic value of an interface value. For example,

```

1| package main
2|
3| import "fmt"
4| import "reflect"
5|
```

```

6| func main() {
7|     var z = 123
8|     var y = &z
9|     var x interface{} = y
10|    v := reflect.ValueOf(&x)
11|    vx := v.Elem()
12|    vy := vx.Elem()
13|    vz := vy.Elem()
14|    vz.Set(reflect.ValueOf(789))
15|    fmt.Println(z) // 789
16| }
```

The `reflect` standard package also declares some `reflect.Value` related functions. Each of these functions corresponds to a built-in function or a non-reflection functionality. The following example demonstrates how to bind a (kind of) custom generic function to different function values.

```

1| package main
2|
3| import "fmt"
4| import "reflect"
5|
6| func InvertSlice(args []reflect.Value) []reflect.Value {
7|     inSlice, n := args[0], args[0].Len()
8|     outSlice := reflect.MakeSlice(inSlice.Type(), 0, n)
9|     for i := n-1; i >= 0; i-- {
10|         element := inSlice.Index(i)
11|         outSlice = reflect.Append(outSlice, element)
12|     }
13|     return []reflect.Value{outSlice}
14| }
15|
16| func Bind(p interface{},
17|           f func ([]reflect.Value) []reflect.Value) {
18|     // invert represents a function value.
19|     invert := reflect.ValueOf(p).Elem()
20|     invert.Set(reflect.MakeFunc(invert.Type(), f))
21| }
22|
23| func main() {
24|     var invertInts func([]int) []int
25|     Bind(&invertInts, InvertSlice)
26|     fmt.Println(invertInts([]int{2, 3, 5})) // [5 3 2]
27|
28|     var invertStrs func([]string) []string
29|     Bind(&invertStrs, InvertSlice)
```

```

30|     fmt.Println(invertStrs([]string{"Go", "C})) // [C Go]
31| }
```

If the underlying value of a `reflect.Value` is a function value, then we can call the `Call` method of the `reflect.Value` to call the underlying function.

```

1| package main
2|
3| import "fmt"
4| import "reflect"
5|
6| type T struct {
7|     A, b int
8| }
9|
10| func (t T) AddSubThenScale(n int) (int, int) {
11|     return n * (t.A + t.b), n * (t.A - t.b)
12| }
13|
14| func main() {
15|     t := T{5, 2}
16|     vt := reflect.ValueOf(t)
17|     vm := vt.MethodByName("AddSubThenScale")
18|     results := vm.Call([]reflect.Value{reflect.ValueOf(3)})
19|     fmt.Println(results[0].Int(), results[1].Int()) // 21 9
20|
21|     neg := func(x int) int {
22|         return -x
23|     }
24|     vf := reflect.ValueOf(neg)
25|     fmt.Println(vf.Call(results[:1])[0].Int()) // -21
26|     fmt.Println(vf.Call([]reflect.Value{
27|         vt.FieldByName("A"), // panic on changing to "b"
28|     })[0].Int()) // -5
29| }
```

Please note that, non-exported fields shouldn't be used as arguments of reflection calls. If the line `vt.FieldName("A")` in the above example is replaced with `vt.FieldName("b")`, a panic will occur.

A reflection example for map values.

```

1| package main
2|
3| import "fmt"
```

```

4| import "reflect"
5|
6| func main() {
7|     valueOf := reflect.ValueOf
8|     m := map[string]int{"Unix": 1973, "Windows": 1985}
9|     v := valueOf(m)
10|    // A zero second Value argument means to delete an entry.
11|    v.SetMapIndex(valueOf("Windows"), reflect.Value{})
12|    v.SetMapIndex(valueOf("Linux"), valueOf(1991))
13|    for i := v.MapRange(); i.Next(); {
14|        fmt.Println(i.Key(), "\t:", i.Value())
15|    }
16| }
```

Please note that, the `MapRange` method is supported since Go 1.12.

A reflection example for channel values.

```

1| package main
2|
3| import "fmt"
4| import "reflect"
5|
6| func main() {
7|     c := make(chan string, 2)
8|     vc := reflect.ValueOf(c)
9|     vc.Send(reflect.ValueOf("C"))
10|    succeeded := vc.TrySend(reflect.ValueOf("Go"))
11|    fmt.Println(succeeded) // true
12|    succeeded = vc.TrySend(reflect.ValueOf("C++"))
13|    fmt.Println(succeeded) // false
14|    fmt.Println(vc.Len(), vc.Cap()) // 2 2
15|    vs, succeeded := vc.TryRecv()
16|    fmt.Println(vs.String(), succeeded) // C true
17|    vs, sentBeforeClosed := vc.Recv()
18|    fmt.Println(vs.String(), sentBeforeClosed) // Go true
19|    vs, succeeded = vc.TryRecv()
20|    fmt.Println(vs.String()) // <invalid Value>
21|    fmt.Println(succeeded) // false
22| }
```

The `TrySend` and `TryRecv` methods correspond to one-case-one-default [select control flow code blocks](#) (§21).

We can use the `reflect.Select` function to simulate a `select` code block with dynamic number of case branches at run time.

```

1| package main
2|
3| import "fmt"
4| import "reflect"
5|
6| func main() {
7|     c := make(chan int, 1)
8|     vc := reflect.ValueOf(c)
9|     succeeded := vc.TrySend(reflect.ValueOf(123))
10|    fmt.Println(succeeded, vc.Len(), vc.Cap()) // true 1 1
11|
12|    vSend, vZero := reflect.ValueOf(789), reflect.Value{}
13|    branches := []reflect.SelectCase{
14|        {Dir: reflect.SelectDefault, Chan: vZero, Send: vZero},
15|        {Dir: reflect.SelectRecv, Chan: vc, Send: vZero},
16|        {Dir: reflect.SelectSend, Chan: vc, Send: vSend},
17|    }
18|    selIndex, vRecv, sentBeforeClosed := reflect.Select(branches)
19|    fmt.Println(selIndex)          // 1
20|    fmt.Println(sentBeforeClosed) // true
21|    fmt.Println(vRecv.Int())     // 123
22|    vc.Close()
23|    // Remove the send case branch this time,
24|    // for it may cause panic.
25|    selIndex, _, sentBeforeClosed = reflect.Select(branches[:2])
26|    fmt.Println(selIndex, sentBeforeClosed) // 1 false
27| }
```

The respective underlying values of some `reflect.Value` values may be nothing. For example, zero `reflect.Value` values.

```

1| package main
2|
3| import "reflect"
4| import "fmt"
5|
6| func main() {
7|     var z reflect.Value // a zero Value value
8|     fmt.Println(z)      // <invalid reflect.Value>
9|     v := reflect.ValueOf((*int)(nil)).Elem()
10|    fmt.Println(v)       // <invalid reflect.Value>
11|    fmt.Println(v == z) // true
```

```

12|     var i = reflect.ValueOf([]interface{}{nil}).Index(0)
13|     fmt.Println(i)           // <nil>
14|     fmt.Println(i.Elem() == z) // true
15|     fmt.Println(i.Elem())    // <invalid reflect.Value>
16|

```

For a Go value, we can use the `reflect.ValueOf` function to create a `reflect.Value` value representing the Go value, through the help of `interface{}`. The inverse process is similar, we can call the `Interface` method of a `reflect.Value` value to get an `interface{}` value, then type assert on the `interface{}` value to get the Go value represented by (a.k.a., the underlying value of) the `reflect.Value` value. But please note that, calling the `Interface` method of a `reflect.Value` value which represents a non-exported field causes a panic.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "reflect"
6|     "time"
7| )
8|
9| func main() {
10|     vx := reflect.ValueOf(123)
11|     vy := reflect.ValueOf("abc")
12|     vz := reflect.ValueOf([]bool{false, true})
13|     vt := reflect.ValueOf(time.Time{})
14|
15|     x := vx.Interface().(int)
16|     y := vy.Interface().(string)
17|     z := vz.Interface().([]bool)
18|     m := vt.MethodByName("IsZero").Interface().(func() bool)
19|     fmt.Println(x, y, z, m()) // 123 abc [false true] true
20|
21|     type T struct {x int}
22|     t := &T{3}
23|     v := reflect.ValueOf(t).Elem().Field(0)
24|     fmt.Println(v)           // 3
25|     fmt.Println(v.Interface()) // panic
26|

```

The method `reflect.Value.IsZero` was introduced in Go 1.13. It is used to check whether or not the underlying value of a `reflect.Value` value is a zero value.

Since Go 1.17, [a slice may be converted to an array pointer](#) (§18). However, such a conversion might panic if the length of the pointer base array type is too large. The method `reflect.Value.CanConvert(T reflect.Type)` introduced in Go 1.17 is used to check whether or not a conversion will succeed.

An example using the `CanConvert` method:

```

1| package main
2|
3| import (
4|     "fmt"
5|     "reflect"
6| )
7|
8| func main() {
9|     s := reflect.ValueOf([]int{1, 2, 3, 4, 5})
10|    ts := s.Type()
11|    t1 := reflect.TypeOf(&[5]int{})
12|    t2 := reflect.TypeOf(&[6]int{})
13|    fmt.Println(ts.ConvertibleTo(t1)) // true
14|    fmt.Println(ts.ConvertibleTo(t2)) // true
15|    fmt.Println(s.CanConvert(t1))    // true
16|    fmt.Println(s.CanConvert(t2))    // false
17| }
```

There are more `reflect.Value` related functions and methods which are not used in above examples, please read the `reflect` package documentation for their usages. In addition, please note that there are [some reflection](#) (§50) related [details](#) (§50) mentioned in [Go details 101](#) (§50).

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Line Break Rules in Go

If you have written go code much, you should have known that we can't use arbitrary code styles in Go programming. Specifically speaking, we can't break a code line at an arbitrary space character position. The remaining of this article will list the detailed line break rules in Go.

Semicolon Insertion Rules

One rule we often obey in practice is, we should not put the a starting curly brace ({) of any explicit code block on a new line. For example, the following `for` loop code block fails to compile.

```
1|     for i := 5; i > 0; i--
2|     { // unexpected newline, expecting { after for clause
3| }
```

To make it compiles okay, the starting curly brace mustn't be put on a new line, like the following:

```
1|     for i := 5; i > 0; i-- {
2| }
```

However, there are some exceptions for the rule mentioned above. For example, the following bare `for` loop block compiles okay.

```
1|     for
2|     {
3|         // do something ...
4|     }
```

Then, what are the fundamental rules to do line breaks in Go programming? Before answering this question, we should know a fact that the formal Go grammar uses semicolons ; as terminators of code lines. However, we seldom use semicolons in our Go code. Why? The reason is most semicolons are optional and can be omitted. Go compilers will insert the omitted semicolons for us automatically in compiling.

For example, the ten semicolons in the following program are all optional.

```
1| package main;
2|
3| import "fmt";
4|
```

```

5| func main() {
6|     var (
7|         i    int;
8|         sum int;
9|     );
10|    for i < 6 {
11|        sum += i;
12|        i++;
13|    };
14|    fmt.Println(sum);
15| };

```

Assume the above program is stored in a file named `semicolons.go`, we can run `go fmt semicolons.go` to remove all the unnecessary semicolons from that file. Compilers will insert the removed semicolons back (in memory) automatically in compiling the source code.

What are the semicolons insertion rules in Go? Let's read [the semicolon rules listed in Go specification ↗](#).

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 immediately after a line's final token if that token is
 - an [identifier](#) (§5)
 - an integer, floating-point, imaginary, rune, or string [literal](#) (§6)
 - one of the keywords `break`, `continue`, `fallthrough`, or `return`
 - one of the operators and punctuation `++`, `--`, `)`, `]`, or `}`
2. To allow complex statements to occupy a single line, a semicolon may be omitted before a closing `)` or `}`.

For the scenarios listed in the first rule, surely, we can also insert the semicolons manually, just like the semicolons in the last code example. In other words, these semicolons are optional in programming.

The second rule means the last semicolon in a multi-item declaration before the closing sign `)` and the last semicolon within a code block or a (struct or interface) type declaration before the closing sign `}` are optional. If the last semicolon is absent, compilers will automatically insert it back.

The second rule lets us be able to write the following valid code.

```

1| import (_ "math"; "fmt")
2| var (a int; b string)
3| const (M = iota; N)
4| type (MyInt int; T struct{x bool; y int32})
5| type I interface{m1(int) int; m2() string}
6| func f() {print("a"); panic(nil)}

```

Compilers will automatically insert the omitted semicolons for us, as the following code shows.

```

1| var (a int; b string);
2| const (M = iota; N);
3| type (MyInt int; T struct{x bool; y int32;});
4| type I interface{m1(int) int; m2() string;};
5| func f() {print("a"); panic(nil);}

```

Compilers will not insert semicolons for any other scenarios. We must insert the semicolons manually as needed for other scenarios. For example, the first semicolon at each line in the last code example are all required. The semicolons in the following example are also required.

```

1| var a = 1; var b = true
2| a++; b = !b
3| print(a); print(b)

```

From the two rules, we know that a semicolon will never be inserted just after the `for` keyword. This is why the bare `for` loop example shown above is valid.

One consequence of the semicolon insertion rules is that the self increment and self decrement operations must appear as statements. They can't be used as expressions. For example, the following code is invalid.

```

1| func f() {
2|     a := 0
3|     // The following two lines both fail to compile.
4|     println(a++) // unexpected ++, expecting comma or )
5|     println(a--) // unexpected --, expecting comma or )
6| }

```

The reason why the above code is invalid is compilers will view it as

```

1| func f() {
2|     a := 0;
3|     println(a++););
4|     println(a--););
5| }

```

Another consequence of the semicolon insertion rules is we can't break a line before the dot `.` in a selector. We can only break a line after the dot, as the following code shows

```
1|     anObject.  
2|         MethodA().  
3|         MethodB().  
4|         MethodC()
```

whereas the following code fails to compile.

```
1|     anObject  
2|         .MethodA()  
3|         .MethodB()  
4|         .MethodC()
```

Compilers will insert a semicolon at the end of each line in the modified version, so the above code is equivalent to the following code which is obviously invalid.

```
1|     anObject;  
2|         .MethodA();  
3|         .MethodB();  
4|         .MethodC();
```

The semicolon insertion rules make us write cleaner code. They also make it is possible to write some valid but a little weird code. For example,

```
1| package main  
2|  
3| import "fmt"  
4|  
5| func alwaysFalse() bool {return false}  
6|  
7| func main() {  
8|     for  
9|         i := 0  
10|        i < 6  
11|        i++ {  
12|            // use i ...  
13|        }  
14|  
15|        if x := alwaysFalse()  
16|        !x {  
17|            // do something ...  
18|        }  
19|    }
```

```

20|     switch alwaysFalse()
21|     {
22|         case true: fmt.Println("true")
23|         case false: fmt.Println("false")
24|     }
25| }
```

All the three control flow blocks are valid. Compilers will insert a semicolon at the end of each of line 9, 10, 15 and 20.

Please note, the `switch-case` block in the above example will print a `true` instead of a `false`. It is different from

```

1|     switch alwaysFalse() {
2|         case true: fmt.Println("true")
3|         case false: fmt.Println("false")
4|     }
```

If we use the `go fmt` command to format the former one, a semicolon will be appended automatically after the `alwaysFalse()` call, so it will become to

```

1|     switch alwaysFalse();
2|     {
3|         case true: fmt.Println("true")
4|         case false: fmt.Println("false")
5|     }
```

The modified version is equivalent to the following one.

```

1|     switch alwaysFalse(); true {
2|         case true: fmt.Println("true")
3|         case false: fmt.Println("false")
4|     }
```

That is why it will print a `true`.

It is a good habit to run `go fmt` and `go vet` often for your code.

For a rare case, the semicolon insertion rules also make some code look valid but invalid actually. For example, the following code snippet fails to compile.

```

1| func f(x int) {
2|     switch x {
3|         case 1:
4|     }
```

```

5|     goto A
6|     A: // compiles okay
7| }
8| case 2:
9|     goto B
10|    B: // syntax error: missing statement after label
11| case 0:
12|     goto C
13|    C: // compiles okay
14| }
15| }
```

The compilation error message indicates that there must be a statement following a label declaration. But it looks none label in the above example is followed by a statement. Why is only the B: label declaration invalid? The reason is, by the second semicolon insertion rule mentioned above, compilers will insert a semicolon before each of the } characters following the A: and C: label declarations. As the following code shows.

```

1| func f(x int) {
2|     switch x {
3|     case 1:
4|     {
5|         goto A
6|         A:
7|     };} // a semicolon is inserted here
8|     case 2:
9|         goto B
10|        B: // syntax error: missing statement after label
11|     case 0:
12|         goto C
13|         C:
14|     };} // a semicolon is inserted here
15| }
```

A solo semicolon represents a [blank statement](#) (§11) actually, which is why the A: and C: label declarations are both valid. On the other hand, the B: label declaration is followed by case 0:, which is not a statement, so the B: label declaration is invalid.

We can manually insert a semicolon (a blank statement) at the end of each of the B: label declaration to make it compile okay.

Comma (,) Will Not Be Inserted Automatically

In some syntax forms containing multiple alike items, commas are used as separators, such as composite literals, function argument lists, function parameter lists and function result lists. In such a syntax form, the last item can always be followed by a comma. If the following comma is the last effective character in its respective code line, then the comma is required, otherwise, it is optional. Compilers will not insert commas automatically for any cases.

For example, the following code snippet is valid.

```

1| func f1(a int, b string,) (x bool, y int,) {
2|     return true, 789
3|
4| var f2 func (a int, b string) (x bool, y int)
5| var f3 func (a int, b string, // the last comma is required
6| ) (x bool, y int,           // the last comma is required
7| )
8| var _ = []int{2, 3, 5, 7, 9,} // the last comma is optional
9| var _ = []int{2, 3, 5, 7, 9, // the last comma is required
10|
11| var _ = []int{2, 3, 5, 7, 9}
12| var _, _ = f1(123, "Go",) // the last comma is optional
13| var _, _ = f1(123, "Go", // the last comma is required
14|
15| var _, _ = f1(123, "Go")
16| // The same for explicit conversions.
17| var _ = string(65,) // the last comma is optional
18| var _ = string(65, // the last comma is required
19|

```

However, the following code snippet is invalid, for compilers will insert a semicolon for each line in the code, except the second line. There are three lines which will cause unexpected newline syntax errors.

```

1| func f1(a int, b string,) (x bool, y int // error
2| ) {
3|     return true, 789
4|
5| var _ = []int{2, 3, 5, 7 // error: unexpected newline
6|
7| var _, _ = f1(123, "Go" // error: unexpected newline
8|

```

Final Words

At the end, let's describe the line break rules in Go according to the above explanations.

In Go, a line break is okay (will not affect code behavior) if:

- it happens immediately after a keyword other than `break`, `continue` and `return`, or after any of the three keywords which are not followed by labels or return results;
- it happens immediately after a semicolon, whether the semicolon is inserted explicitly or implicitly;
- it doesn't lead to an implicit semicolon will be inserted.

Like some other design details in Go, there are both praises and criticisms for the semicolon insertion rules. Some programmers don't like the rules, for they think the rules limit the freedom of code styles. Praisers think the rules make code compile faster, and make the code written by different programmers look similar, so that it is easy to understand the code written by each other.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

More about Deferred Function Calls

Deferred function calls have been [introduced before](#) (§13). Due to the limited Go knowledge at that time, some more details and use cases of deferred functions calls are not touched in that article. These details and use cases will be touched in the remaining of this article.

Calls to Many Built-in Functions With Return Results Can't Be Deferred

In Go, the result values of a call to custom functions can be all absent (discarded). However, for built-in functions with non-blank return result lists, the result values of their calls [mustn't be absent](#) (§49) (up to Go 1.21), except the calls to the built-in `copy` and `recover` functions. On the other hand, we have learned that the result values of a deferred function call must be discarded, so the calls to many built-in functions can't be deferred.

Fortunately, the needs to defer built-in function calls (with non-blank return result lists) are rare in practice. As far as I know, only the calls to the built-in `append` function may needed to be deferred sometimes. For this case, we can defer a call to an anonymous function which wraps the `append` call.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     s := []string{"a", "b", "c", "d"}
7|     defer fmt.Println(s) // [a x y d]
8|     // defer append(s[:1], "x", "y") // error
9|     defer func() {
10|         _ = append(s[:1], "x", "y")
11|     }()
12| }
```

The Evaluation Moment of Deferred Function Values

The called function (value) in a deferred function call is evaluated when the call is pushed into the deferred call queue of the current goroutine. For example, the following program will print `false`.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     var f = func () {
7|         fmt.Println(false)
8|     }
9|     defer f()
10|    f = func () {
11|        fmt.Println(true)
12|    }
13| }
```

The called function in a deferred function call may be a nil function value. For such a case, the panic will occur when the call to the nil function is invoked, instead of when the call is pushed into the deferred call queue of the current goroutine. An example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     defer fmt.Println("reachable 1")
7|     var f func() // f is nil by default
8|     defer f()    // panic here
9|     // The following lines are also reachable.
10|    fmt.Println("reachable 2")
11|    f = func() {} // useless to avoid panicking
12| }
```

The Evaluation Moment of Receiver Arguments of Deferred Method Calls

As explained before, the arguments of a deferred function call are [also evaluated when](#) (§13) the deferred call is pushed into the deferred call queue of the current goroutine.

Method receiver arguments are also not exceptions. For example, the following program prints 1342.

```

1| package main
2|
3| type T int
```

```

4|
5| func (t T) M(n int) T {
6|   print(n)
7|   return t
8| }
9|
10| func main() {
11|   var t T
12|   // "t.M(1)" is the receiver argument of the method
13|   // call ".M(2)", so it is evaluated when the
14|   // ".M(2)" call is pushed into deferred call queue.
15|   defer t.M(1).M(2)
16|   t.M(3).M(4)
17| }
```

Deferred Calls Make Code Cleaner and Less Bug Prone

Example:

```

1| import "os"
2|
3| func withoutDefers(filepath string, head, body []byte) error {
4|   f, err := os.Open(filepath)
5|   if err != nil {
6|     return err
7|   }
8|
9|   _, err = f.Seek(16, 0)
10|  if err != nil {
11|    f.Close()
12|    return err
13|  }
14|
15|  _, err = f.Write(head)
16|  if err != nil {
17|    f.Close()
18|    return err
19|  }
20|
21|  _, err = f.Write(body)
22|  if err != nil {
23|    f.Close()
24|    return err
25|  }
```

```

26|
27|     err = f.Sync()
28|     f.Close()
29|     return err
30| }
31|
32| func withDefers(filepath string, head, body []byte) error {
33|     f, err := os.Open(filepath)
34|     if err != nil {
35|         return err
36|     }
37|     defer f.Close()
38|
39|     _, err = f.Seek(16, 0)
40|     if err != nil {
41|         return err
42|     }
43|
44|     _, err = f.Write(head)
45|     if err != nil {
46|         return err
47|     }
48|
49|     _, err = f.Write(body)
50|     if err != nil {
51|         return err
52|     }
53|
54|     return f.Sync()
55| }
```

Which one looks cleaner? Apparently, the one with the deferred calls, though a little. And it is less bug prone, for there are so many `f.Close()` calls in the function without deferred calls that it has a higher possibility to miss one of them.

The following is another example to show deferred calls can make code less bug prone. If the `doSomething` calls panic in the following example, the function `f2` will exit without unlocking the `Mutex` value. So the function `f1` is less bug prone.

```

1| var m sync.Mutex
2|
3| func f1() {
4|     m.Lock()
5|     defer m.Unlock()
6|     doSomething()
```

```

7| }
8|
9| func f2() {
10|     m.Lock()
11|     doSomething()
12|     m.Unlock()
13| }
```

Performance Losses Caused by Deferring Function Calls

It is not always good to use deferred function calls. For the official Go compiler, before version 1.13, deferred function calls will cause a few performance losses at run time. Since Go Toolchain 1.13, some common defer use cases have got optimized much, so that generally we don't need to care about the performance loss problem caused by deferred calls. Thank Dan Scales for making the great optimizations.

Kind-of Resource Leaking by Deferring Function Calls

A very large deferred call queue may also consume much memory, and some resources might not get released in time if some calls are delayed too much. For example, if there are many files needed to be handled in a call to the following function, then a large number of file handlers will be not get released before the function exits.

```

1| func writeManyFiles(files []File) error {
2|     for _, file := range files {
3|         f, err := os.Open(file.path)
4|         if err != nil {
5|             return err
6|         }
7|         defer f.Close()
8|
9|         _, err = f.WriteString(file.content)
10|        if err != nil {
11|            return err
12|        }
13|
14|        err = f.Sync()
15|        if err != nil {
16|            return err
17|        }
```

```

18| }
19|
20|     return nil
21| }
```

For such cases, we can use an anonymous function to enclose the deferred calls so that the deferred function calls will get executed earlier. For example, the above function can be rewritten and improved as

```

1| func writeManyFiles(files []File) error {
2|     for _, file := range files {
3|         if err := func() error {
4|             f, err := os.Open(file.path)
5|             if err != nil {
6|                 return err
7|             }
8|             // The close method will be called at
9|             // the end of the current loop step.
10|            defer f.Close()
11|
12|             _, err = f.WriteString(file.content)
13|             if err != nil {
14|                 return err
15|             }
16|
17|             return f.Sync()
18|         }(); err != nil {
19|             return err
20|         }
21|     }
22|
23|     return nil
24| }
```

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Some Panic/Recover Use Cases

Panic and recover have been [introduced before](#) (§13). The following of the current article will introduce some (good and bad) panic/recover use cases.

Use Case 1: Avoid Panics Crashing Programs

This should be the most popular use case of panic/recover. The use case is used commonly in concurrent programs, especially client-server programs.

An example:

```

1| package main
2|
3| import "errors"
4| import "log"
5| import "net"
6|
7| func main() {
8|     listener, err := net.Listen("tcp", ":12345")
9|     if err != nil {
10|         log.Fatalln(err)
11|     }
12|     for {
13|         conn, err := listener.Accept()
14|         if err != nil {
15|             log.Println(err)
16|         }
17|         // Handle each client connection
18|         // in a new goroutine.
19|         go ClientHandler(conn)
20|     }
21| }
22|
23| func ClientHandler(c net.Conn) {
24|     defer func() {
25|         if v := recover(); v != nil {
26|             log.Println("capture a panic:", v)
27|             log.Println("avoid crashing the program")
28|         }
29|         c.Close()
30| }()

```

```

31|     panic(errors.New("just a demo.")) // a demo-purpose panic
32| }
```

Start the server and run `telnet localhost 12345` in another terminal, we can observe that the server will not crash down for the panics created in each client handler goroutine.

If we don't recover the potential panic in each client handler goroutine, the potential panic will crash the program.

Use Case 2: Automatically Restart a Crashed Goroutine

When a panic is detected in a goroutine, we can create a new goroutine for it. An example:

```

1| package main
2|
3| import "log"
4| import "time"
5|
6| func shouldNotExit() {
7|     for {
8|         // Simulate a workload.
9|         time.Sleep(time.Second)
10|
11|         // Simulate an unexpected panic.
12|         if time.Now().UnixNano() & 0x3 == 0 {
13|             panic("unexpected situation")
14|         }
15|     }
16| }
17|
18| func NeverExit(name string, f func()) {
19|     defer func() {
20|         if v := recover(); v != nil {
21|             // A panic is detected.
22|             log.Println(name, "is crashed. Restart it now.")
23|             go NeverExit(name, f) // restart
24|         }
25|     }()
26|     f()
27| }
28|
29| func main() {
```

```

30|     log.SetFlags(0)
31|     go NeverExit("job#A", shouldNotExit)
32|     go NeverExit("job#B", shouldNotExit)
33|     select{} // block here for ever
34|

```

Use Case 3: Use panic/recover Calls to Simulate Long Jump Statements

Sometimes, we can use panic/recover as a way to simulate crossing-function long jump statements and crossing-function returns, though generally this way is not recommended to use. This way does harm for both code readability and execution efficiency. The only benefit is sometimes it can make code look less verbose.

In the following example, once a panic is created in an inner function, the execution will jump to the deferred call.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     n := func () (result int) {
7|         defer func() {
8|             if v := recover(); v != nil {
9|                 if n, ok := v.(int); ok {
10|                     result = n
11|                 }
12|             }
13|         }()
14|
15|         func () {
16|             func () {
17|                 func () {
18|                     // ...
19|                     panic(123) // panic on succeeded
20|                 }()
21|                 // ...
22|             }()
23|         }()
24|         // ...
25|         return 0
26|     }()
}

```

```

27|     fmt.Println(n) // 123
28| }
```

Use Case 4: Use panic/recover Calls to Reduce Error Checks

An example:

```

1| func doSomething() (err error) {
2|     defer func() {
3|         err, _ = recover().(error)
4|     }()
5|
6|     doStep1()
7|     doStep2()
8|     doStep3()
9|     doStep4()
10|    doStep5()
11|
12|    return
13| }
14|
15| // In reality, the prototypes of the doStepN functions
16| // might be different. Here, for each of them,
17| // * panic with nil for success and no needs to continue.
18| // * panic with error for failure and no needs to continue.
19| // * will not produce other panics.
20| // * not panic for continuing.
21| func doStepN() {
22|     ...
23|     if err != nil {
24|         panic(err)
25|     }
26|     ...
27|     if done {
28|         panic(nil)
29|     }
30| }
```

The above code is less verbose than the following one.

```

1| func doSomething() (err error) {
2|     shouldContinue, err := doStep1()
```

```

3|     if !shouldContinue {
4|         return err
5|     }
6|     shouldContinue, err = doStep2()
7|     if !shouldContinue {
8|         return err
9|     }
10|    shouldContinue, err = doStep3()
11|    if !shouldContinue {
12|        return err
13|    }
14|    shouldContinue, err = doStep4()
15|    if !shouldContinue {
16|        return err
17|    }
18|    shouldContinue, err = doStep5()
19|    if !shouldContinue {
20|        return err
21|    }
22|
23|    return
24| }
25|
26| // If err is not nil, then shouldContinue must be true.
27| // If shouldContinue is true, err might be nil or non-nil.
28| func doStepN() (shouldContinue bool, err error) {
29|     ...
30|     if err != nil {
31|         return false, err
32|     }
33|     ...
34|     if done {
35|         return false, nil
36|     }
37|     return true, nil
38| }
```

However, usually, this panic/recover use pattern is not recommended to use. It is less Go-idiomatic and less efficient.

And please note that, since Go 1.21, a `panic(nil)` call will become [equivalent to `panic\(new\(runtime.PanicNilError\)\)`](#). So since Go 1.21, the above deferred function call should be written as

```
1| func doSomething() (err error) {
2|     defer func() {
3|         err, _ = recover().(error)
4|         if e := (*runtime.PanicNilError)(nil); errors.As(err, &e) {
5|             err = nil
6|         }
7|     }()
8|
9|     doStep1()
10|    ...
11| }
```

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Explain Panic/Recover Mechanism in Detail

Panic and recover mechanism has been [introduced before](#) (§13), and several panic/recover use cases are shown in [the last article](#) (§30). This current article will explain panic/recover mechanism in detail. Exiting phases of function calls will also be explained in detail.

Exiting Phases of Function Calls

In Go, a function call may undergo an exiting phase before it fully exits. In the exiting phase, the deferred function calls pushed into the deferred call queue during executing the function call will be executed (in the inverse pushing order). When all of the deferred calls fully exit, the exiting phase ends and the function call also fully exits.

Exiting phases might also be called returning phases elsewhere.

A function call may enter its exiting phase (or exit directly) through three ways:

1. after the call returns normally.
2. when a panic occurs in the call.
3. after the `runtime.Goexit` function is called and fully exits in the call.

For example, in the following code snippet,

- a call to the function `f0` or `f1` will enter its existing phase after it returns normally.
- a call to the function `f2` will enter its exiting phase after the divided-by-zero panic happens.
- a call to the function `f3` will enter its exiting phase after the `runtime.Goexit` function call fully exits.

```

1| import (
2|     "fmt"
3|     "runtime"
4| )
5|
6| func f0() int {
7|     var x = 1
8|     defer fmt.Println("exits normally:", x)
9|     x++
10|    return x
11| }
12|
13| func f1() {

```

```

14|     var x = 1
15|     defer fmt.Println("exits normally:", x)
16|     x++
17| }
18|
19| func f2() {
20|     var x, y = 1, 0
21|     defer fmt.Println("exits for panicking:", x)
22|     x = x / y // will panic
23|     x++        // unreachable
24| }
25|
26| func f3() int {
27|     x := 1
28|     defer fmt.Println("exits for Goexit:", x)
29|     x++
30|     runtime.Goexit()
31|     return x+x // unreachable
32| }
```

BTW, the `runtime.Goexit()` function is not intended to be called in the main goroutine of a program.

Associating Panics and Goexit Signals of Function Calls

When a panic occurs directly in a function call, we say the (unrecovered) panic starts associating with the function call. Similarly, when the `runtime.Goexit` function is called in a function call, we say a Goexit signal starts associating with the function call after the `runtime.Goexit` call fully exits. As explained in the last section, associating either a panic or a Goexit signal with a function call will make the function call enter its exiting phase immediately.

We have learned that [panics can be recovered](#) (§13). However, there are no ways to cancel a Goexit signal.

At any give time, a function call may associate with at most one unrecovered panic. If a call is associating with an unrecovered panic, then

- the call will associate with no panics when the unrecovered panic is recovered.
- when a new panic occurs in the function call, the new one will replace the old one to become the associating unrecovered panic of the function call.

For example, in the following program, the recovered panic is panic 3, which is the last panic associating with the `main` function call.

```
1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     defer func() {
7|         fmt.Println(recover()) // 3
8|     }()
9|
10|    defer panic(3) // will replace panic 2
11|    defer panic(2) // will replace panic 1
12|    defer panic(1) // will replace panic 0
13|    panic(0)
14| }
```

As Goexit signals can't be cancelled, arguing whether a function call may associate with at most one or more than one Goexit signal is unnecessary.

Although it is unusual, there might be multiple unrecovered panics coexisting in a goroutine at a time. Each one associates with one non-exited function call in the call stack of the goroutine. When a nested call fully exits and it still associates with an unrecovered panic, the unrecovered panic will spread to the nesting call (the caller of the nested call). The effect is the same as a panic occurs directly in the nesting call. That says,

- if there was an old unrecovered panic associating with the nesting call before, the old one will be replaced by the spread one. For this case, the nesting call must have already entered its exiting phase for sure, so the next deferred function call in its deferred call queue will be invoked.
- if there was not an unrecovered panic associating with the nesting call before, the spread one will associate with the nesting call. For this case, the nesting call might have entered its exiting phase or not. If it hasn't, it will enter its exiting phase immediately.

So, when a goroutine finishes to exit, there may be at most one unrecovered panic in the goroutine. If a goroutine exits with an unrecovered panic, the whole program crashes, and the information of the unrecovered panic will be reported. Otherwise, the goroutine exits normally (peacefully).

When a function is invoked, there is neither a panic nor Goexit signals associating with its call initially, no matter whether its caller (the nesting call) has entered exiting phase or not. Surely, panics might occur or the `runtime.Goexit` function might be called later in the process of executing the call, so panics and Goexit signals might associate with the call later.

The following example program will crash if it runs, because the panic 2 is still not recovered when the new goroutine exits.

```

1| package main
2|
3| func main() {
4|     // The new goroutine.
5|     go func() {
6|         // This is an anonymous deferred call.
7|         // When it fully exits, the panic 2 will spread
8|         // to the entry function call of the new
9|         // goroutine, and replace the panic 0. The
10|        // panic 2 will never be recovered.
11|        defer func() {
12|            // As explained in the last example,
13|            // panic 2 will replace panic 1.
14|            defer panic(2)
15|
16|            // When the anonymous function call fully
17|            // exits, panic 1 will spread to (and
18|            // associate with) the nesting anonymous
19|            // deferred call.
20|            func () {
21|                // Once the panic 1 occurs, there will
22|                // be two unrecovered panics coexisting
23|                // in the new goroutine. One (panic 0)
24|                // associates with the entry function
25|                // call of the new goroutine, the other
26|                // (panic 1) associates with the
27|                // current anonymous function call.
28|                panic(1)
29|            }()
30|        }()
31|        panic(0)
32|    }()
33|
34|    select{}
35| }
```

The output (when the above program is compiled with the standard Go compiler v1.21.n):

```

panic: 0
panic: 1
panic: 2
```

...

The format of the output is not perfect, it is prone to make some people think that the panic 0 is the final unrecovered panic, whereas the final unrecovered panic is actually panic 2.

Similarly, when a nested call fully exits and it is associating with a Goexit signal, then the Goexit signal will also spread to (and associate with) the nesting call. This will make the nesting call enter (if it hasn't entered) its exiting phase immediately.

When a Goexit signal associates with a function call, if the function call is associating with an unrecovered panic, then the panic will be recovered. For example, the following program will exit peacefully and print `<nil>`, because the `bye` panic will be recovered by the Goexit signal.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "runtime"
6| )
7|
8| func f() {
9|     defer func() {
10|         fmt.Println(recover())
11| }()
12|
13|     // The Goexit signal will disable the "bye" panic.
14|     defer runtime.Goexit()
15|     panic("bye")
16| }
17|
18| func main() {
19|     go f()
20|
21|     for runtime.NumGoroutine() > 1 {
22|         runtime.Gosched()
23|     }
24| }
```

Some recover Calls Are No-Ops

The builtin `recover` function must be called at proper places to take effect. Otherwise, the calls are no-ops. For example, none of the `recover` calls in the following example recover the `bye` panic.

```

1| package main
2|
3| func main() {
4|     defer func() {
5|         defer func() {
6|             recover() // no-op
7|         }()
8|     }()
9|     defer func() {
10|         func() {
11|             recover() // no-op
12|         }()
13|     }()
14|     func() {
15|         defer func() {
16|             recover() // no-op
17|         }()
18|     }()
19|     func() {
20|         defer recover() // no-op
21|     }()
22|     func() {
23|         recover() // no-op
24|     }()
25|     recover()      // no-op
26|     defer recover() // no-op
27|     panic("bye")
28| }
```

We have already known that the following `recover` call takes effect.

```

1| package main
2|
3| func main() {
4|     defer func() {
5|         recover() // take effect
6|     }()
7|
8|     panic("bye")
9| }
```

Then why don't those `recover` calls in the first example of the current section take effect? Let's read the current version of [Go specification ↗](#):

The return value of `recover` is `nil` if any of the following conditions holds:

- panic's argument was nil;
- the goroutine is not panicking;
- recover was not called directly by a deferred function.

There is [an example](#) (§30) showing the first condition case in the last article.

Most of the recover calls in the first example of the current section satisfy either the second or the third conditions mentioned in Go specification, except the first call. Yes, here, the current descriptions are not precise yet. The third condition should be described as

- recover was not called directly by a deferred function **call which was called directly by the function call associating with the expected to-be-recovered panic.**

In the first example of the current section, the expected to-be-recovered panic is associating with the main function call. The first recover call is called directly by a deferred function call but the deferred function call is not called directly by the main function call. This is why the first recover call is a no-op.

In fact, the current Go specification also doesn't explain well why the second recover call (by code line order), which is expected to recover panic 1, in the following example doesn't take effect.

```

1| // This program exits without recovering panic 1.
2| package main
3|
4| func demo() {
5|     defer func() {
6|         defer func() {
7|             recover() // this one recovers panic 2
8|         }()
9|
10|        defer recover() // no-op
11|
12|        panic(2)
13|    }()
14|    panic(1)
15| }
16|
17| func main() {
18|     demo()
19| }
```

What Go specification doesn't mention is that, every recover call is viewed as an attempt to recover the newest unrecovered panic in the current goroutine.

Go runtime thinks the second `recover` call in the above example attempts to recover the newest unrecovered panic, panic 2, which is associating with the caller call of the second `recover` call. The second `recover` call is not called directly by a deferred function call which is called by the associating function call. Instead, it is directly called by the associating function call. This is why the second `recover` call is a no-op.

Summary

OK, now, let's try to make a short description on which `recover` calls will take effect:

A `recover` call takes effect only if the direct caller of the `recover` call is a deferred call and the direct caller of the deferred call associates with the newest unrecovered panic in the current goroutine. An effective `recover` call disassociates the newest unrecovered panic from its associating function call, and returns the value passed to the `panic` call which produced the newest unrecovered panic.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW,

Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Code Blocks and Identifier Scopes

This article will explain the code blocks and identifier scopes in Go.

(Please note, the definitions of code block hierarchies in this article are a little different from the viewpoint of go/ standard packages.)*

Code Blocks

In a Go project, there are four kinds of code blocks (also called blocks later):

- the **universe block** contains all project source code.
- each package has a **package block** containing all source code, excluding the package import declarations in that package.
- each file has a **file block** containing all the source code, including the package import declarations, in that file.
- generally, a pair of braces {} encloses a **local block**. However, some local blocks aren't enclosed within {}, such blocks are called implicit local blocks. The local blocks enclosed in {} are called explicit local blocks. The {} in composite literals and type definitions don't form local blocks.

Some keywords in all kinds of control flows are followed by some implicit code blocks.

- An `if`, `switch` or `for` keyword is followed by two nested local blocks. One is implicit, the other is explicit. The explicit one is nested in the implicit one. If such a keyword is followed by a short variable declaration, then the variables are declared in the implicit block.
- An `else` keyword is followed by one explicit or implicit block, which is nested in the implicit block following its `if` counterpart keyword. If the `else` keyword is followed by another `if` keyword, then the code block following the `else` keyword can be implicit, otherwise, the code block must be explicit.
- An `select` keyword is followed by one explicit block.
- Each `case` and `default` keyword is followed by one implicit block, which is nested in the explicit block following its corresponding `switch` or `select` keyword.

The local blocks which aren't nested in any other local blocks are called top-level (or package-level) local blocks. Top-level local blocks are all function bodies.

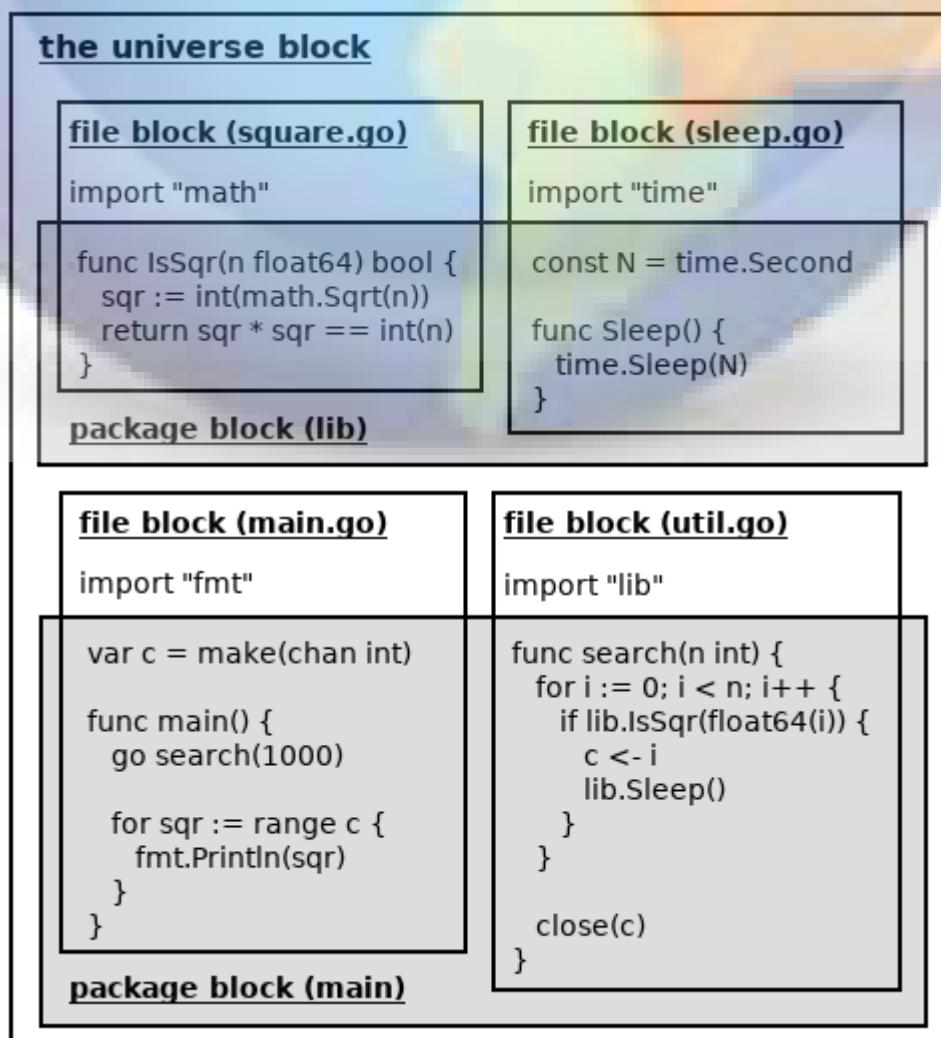
Note, the input parameters and output results of a function are viewed as being declared in explicit body code block of the function, even if their declarations stay out of the pair of braces enclosing the function body block.

Code block hierarchies:

- package blocks are nested in the universe block.
- file blocks are also directly nested in the universe block, instead of package blocks. (This explanation is different from the `go/*` standard packages.)
- each top-level local block is nested in both a package block and a file block. (This explanation is also different from the `go/*` standard packages.)
- a non-top local block must be nested in another local block.

(The differences to Go specification are to make the below explanations for identifier shadowing simpler.)

Here is a picture shows the block hierarchies in a program:



Code blocks are mainly used to explain allowed declaration positions and scopes of source code element identifiers.

Source Code Element Declaration Places

There are six kinds of source code elements which can be declared:

- package imports.
- defined types and type alias.
- named constants.
- variables.
- functions.
- labels.

Labels are used in the `break`, `continue`, and `goto` statements.

A declaration binds a non-blank identifier to a source code element (constant, type, variable, function, label, or package). In other words, in the declaration, the declared source code element is named as the non-blank identifier. After the declaration, we can use the non-blank identifier to represent the declared source code element.

The following table shows which code blocks all kinds of source code elements can be directly declared in:

	the universe block	package blocks	file blocks	local blocks
predeclared (built-in elements) ⁽¹⁾	Yes			
package imports			Yes	
defined types and type alias (non- builtin)		Yes	Yes	Yes
named constants (non-builtin)		Yes	Yes	Yes
variables (non-builtin) ⁽²⁾		Yes	Yes	Yes
functions (non-builtin)		Yes	Yes	
labels				Yes

⁽¹⁾ predeclared elements are documented in [builtin standard package ↗](#).

⁽²⁾ excluding struct field variables.

So,

- package imports can never be declared in package blocks and local blocks.
- functions can never be declared in local blocks. (Anonymous functions can be enclosed in local blocks but they are not declarations.)
- labels can only be declared in local blocks.

Please note,

- if the innermost containing blocks of two code element declarations are the same one, then the names (identifiers) of the two code elements can't be identical.
- the name (identifier) of a package-level code element declared in a package must not be identical to any package import name declared in any source file in the package (a.k.a., a package import name in a package must not be identical to any package-level code element declared in the package). This rule might [be relaxed later ↗](#).
- if the innermost containing function body blocks of two label declarations are the same one, then the names (identifiers) of the two labels can't be identical.
- the references of a label must be within the innermost function body block containing the declaration of the label.
- some special portions in the implicit local blocks in all kinds of control flows have special requirements. Generally, no code elements are allowed to be directly declared in such implicit local blocks, except some short variable declarations.
 - Each `if`, `switch` or `for` keyword can be closely followed by a short variable declaration.
 - Each `case` keyword in a `select` control flow can be closely followed by a short variable declaration.

(BTW, the `go/*` standard packages think file code blocks can only contain package import declarations.)

The source code elements declared in package blocks but outside of any local blocks are called package-level source code elements. Package-level source code elements can be named constants, variables, functions, defined types, or type aliases.

Scopes of Declared Source Code Elements

The scope of a declared identifier means the identifiable range of the identifier (or visible range).

Without considering identifier shadowing which will be explained in the last section of the current article, [the scope definitions ↗](#) for the identifiers of all kinds of source code elements are listed below.

- The scope of a predeclared/built-in identifier is the universe block.
- The scope of the identifier of a package import is the file block containing the package import declaration.
- The scope of an identifier denoting a constant, type, variable, or function (but not method) declared at package level is the package block.

- The scope of an identifier denoting a method receiver, function parameter, or result variable is the corresponding function body (a local block).
- The scope of the identifier of a constant or a variable declared inside a function body begins at the end of the specification of the constant or variable (or the end of the declaration for a short declared variable) and ends at the end of the innermost containing block.
- The scope of the identifier of a type or a type alias declared inside a function body begins at the end of the identifier in the corresponding type specification and ends at the end of the innermost containing block.
- The scope of a label is the body of the innermost function body block containing the label declaration but excludes all the bodies of anonymous functions nested in the containing function.
- About the scopes of type parameters, please read the [Go generics 101 ↗ book](#).

Blank identifiers have no scopes.

(Note, the predeclared iota is only visible in constant declarations.)

You may have noticed the minor difference of identifier scope definitions between local type definitions and local variables, local constants and local type aliases. The difference means a defined type may be able to reference itself in its declaration. Here is an example to show the difference.

```

1| package main
2|
3| func main() {
4|     // var v int = v    // error: v is undefined
5|     // const C int = C // error: C is undefined
6|     /*
7|      type T = struct {
8|          *T      // error: T uses <T>
9|          x []T // error: T uses <T>
10|      }
11|      */
12|
13|      // Following type definitions are all valid.
14|      type T struct {
15|          *T
16|          x []T
17|      }
18|      type A [5]*A
19|      type S []S
20|      type M map[int]M
21|      type F func(F) F
22|      type Ch chan Ch

```

```

23| type P *P
24|
25| // ...
26| var p P
27| p = &p
28| p = *****p
29| *****p = p
30|
31| var s = make(S, 3)
32| s[0] = s
33| s = s[0][0][0][0][0][0][0][0]
34|
35| var m = M{}
36| m[1] = m
37| m = m[1][1][1][1][1][1][1][1]
38| }

```

Note, call `fmt.Println(s)` and call `fmt.Println(m)` both panic (for stack overflow).

And the scope difference between package-level and local declarations:

```

1| package main
2|
3| // Here the two identifiers at each line are the
4| // same one. The right ones are both not the
5| // predeclared identifiers. Instead, they are
6| // same as respective left one. So the two
7| // lines both fail to compile.
8| /*
9| const iota = iota // error: constant definition loop
10| var true = true // error: typechecking loop
11| */
12|
13| var a = b // can reference variables declared later
14| var b = 123
15|
16| func main() {
17|     // The identifiers at the right side in the
18|     // next two lines are the predeclared ones.
19|     const iota = iota // ok
20|     var true = true // ok
21|     _ = true
22|
23|     // The following lines fail to compile, for
24|     // c references a later declared variable d.

```

```

25|  /*
26|  var c = d
27|  var d = 123
28|  _ = c
29|  */
30| }
```

Identifier Shadowing

Ignoring labels, an identifier declared in an outer code block can be shadowed by the same identifier declared in code blocks nested (directly or indirectly) in the outer code block.

Labels can't be shadowed.

If an identifier is shadowed, its scope will exclude the scopes of its shadowing identifiers.

Below is an interesting example. The code contains 6 declared variables named `x`. A `x` declared in a deeper block shadows the `x`s declared in shallower blocks.

```

1| package main
2|
3| import "fmt"
4|
5| var p0, p1, p2, p3, p4, p5 *int
6| var x = 9999 // x#0
7|
8| func main() {
9|     p0 = &x
10|    var x = 888 // x#1
11|    p1 = &x
12|    for x := 70; x < 77; x++ { // x#2
13|        p2 = &x
14|        x := x - 70 // // x#3
15|        p3 = &x
16|        if x := x - 3; x > 0 { // x#4
17|            p4 = &x
18|            x := -x // x#5
19|            p5 = &x
20|        }
21|    }
22|
23|    // 9999 888 77 6 3 -3
24|    fmt.Println(*p0, *p1, *p2, *p3, *p4, *p5)
25| }
```

Another example: the following program prints Sheep Goat instead of Sheep Sheep. Please read the comments for explanations.

```

1| package main
2|
3| import "fmt"
4|
5| var f = func(b bool) {
6|     fmt.Println("Goat")
7| }
8|
9| func main() {
10|     var f = func(b bool) {
11|         fmt.Println("Sheep")
12|         if b {
13|             fmt.Println(" ")
14|             f(!b) // The f is the package-level f.
15|         }
16|     }
17|     f(true) // The f is the local f.
18| }
```

If we modify the above program as the following shown, then it will print Sheep Sheep.

```

1| func main() {
2|     var f func(b bool)
3|     f = func(b bool) {
4|         fmt.Println("Sheep")
5|         if b {
6|             fmt.Println(" ")
7|             f(!b) // The f is also the local f now.
8|         }
9|     }
10|    f(true)
11| }
```

For some circumstances, when identifiers are shadowed by variables declared with short variable declarations, some new gophers may get confused about whether a variable in a short variable declaration is redeclared or newly declared. The following example (which has bugs) shows the famous trap in Go. Almost every gopher has ever fallen into the trap in the early days of using Go.

```

1| package main
2|
3| import "fmt"
```

```

4| import "strconv"
5|
6| func parseInt(s string) (int, error) {
7|     n, err := strconv.Atoi(s)
8|     if err != nil {
9|         // Some new gophers may think err is an
10|        // already declared variable in the following
11|        // short variable declaration. However, both
12|        // b and err are new declared here actually.
13|        // The new declared err variable shadows the
14|        // err variable declared above.
15|     b, err := strconv.ParseBool(s)
16|     if err != nil {
17|         return 0, err
18|     }
19|
20|     // If execution goes here, some new gophers
21|     // might expect a nil error will be returned.
22|     // But in fact, the outer non-nil error will
23|     // be returned instead, for the scope of the
24|     // inner err variable ends at the end of the
25|     // outer if-clause.
26|     if b {
27|         n = 1
28|     }
29| }
30| return n, err
31| }
32|
33| func main() {
34|     fmt.Println(parseInt("TRUE"))
35| }
```

The output:

```
1 strconv.Atoi: parsing "TRUE": invalid syntax
```

Go only has [25 keywords](#) (§5). Keywords can't be used as identifiers. Many familiar words in Go are not keywords, such as `int`, `bool`, `string`, `len`, `cap`, `nil`, etc. They are just predeclared (built-in) identifiers. These predeclared identifiers are declared in the universe block, so custom declared identifiers can shadow them. Here is a weird example in which many predeclared identifiers are shadowed. Its compiles and runs okay.

```

1| package main
2|
```

```
3| import (
4|     "fmt"
5| )
6|
7| // Shadows the built-in function identifier "len".
8| const len = 3
9| // Shadows the built-in const identifier "true".
10| var true = 0
11| // Shadows the built-in variable identifier "nil".
12| type nil struct {}
13| // Shadows the built-in type identifier "int".
14| func int(){}
15|
16| func main() {
17|     fmt.Println("a weird program")
18|     var output = fmt.Println
19|
20|     // Shadows the package import "fmt".
21|     var fmt = [len]nil{{}, {}, {}}
22|     // Sorry, "len" is a constant.
23|     // var n = len(fmt)
24|     // Use the built-in cap function instead, :(
25|     var n = cap(fmt)
26|
27|     // The "for" keyword is followed by one
28|     // implicit local code block and one explicit
29|     // local code block. The iteration variable
30|     // "true" shadows the package-level variable
31|     // "true" declared above.
32|     for true := 0; true < n; true++ {
33|         // Shadows the built-in const "false".
34|         var false = fmt[true]
35|         // The new declared "true" variable
36|         // shadows the iteration variable "true".
37|         var true = true+1
38|         // Sorry, "fmt" is an array, not a package.
39|         // fmt.Println(true, false)
40|         output(true, false)
41|     }
42| }
```

The output:

```
a weird program
1 {}
```

```
2 {}  
3 {}
```

Yes, this example is extreme. It contains many bad practices. Identifier shadowing is useful, but please don't abuse it.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Expression Evaluation Orders

This article will explain [expression](#) (§11) evaluation orders in all kinds of scenarios.

An Expression Is Evaluated After the Expressions It Depends on

This is easy to comprehend. An apparent example is an expression is evaluated later than its sub-expressions. For example, in a function call `f(x, y[n])`,

- `f()` is evaluated later than its depended expressions, including `f`, `x` and `y[n]`.
- the evaluation of the expression `y[n]` is later than the evaluations of `n` and `y`.

Please read [program code element initialization order](#) (§10) for another example on package-level variable initialization orders.

Initialization Order of Package-Level Variables

When a package is loaded at run time, Go runtime will try to initialize uninitialized package-level variables which have no dependencies on uninitialized variables, by their declaration order, until no variables are initialized in such a process. For a successfully compiled Go program, there should be no uninitialized variables after all such processes end.

Package-level variables appearing as blank identifiers are treated like any other variables in the initialization process.

For example, in the following program, variable `a` depends `b`, and variables `c` and `_` depend on `a`. So

1. The first initialized variable is `b`, which is the first package-level variable without dependencies on other package-level variables.
2. The second initialized variable is `a`. After `b` is initialized, `a` is the first package-level variable without dependencies on uninitialized package-level variables.
3. The third and fourth initialized variables are `_` and `c`. After `a` and `b` are initialized, `_` and `c` both don't depend on uninitialized package-level variables.

```

1| package main
2|
3| import "fmt"
4|
5| var (
6|     _ = f()
7|     a = b / 2
8|     b = 6
9|     c = f()
10| )
11|
12| func f() int {
13|     a++
14|     return a
15| }
16|
17| func main() {
18|     fmt.Println(a, b, c) // 5 6 5
19| }
```

The above program prints 5 6 5.

Multiple variables on the left-hand side of a variable specification initialized by single multi-valued expression on the right-hand side are initialized together. For example, for a package-level variable declaration `var x, y = f()`, variables `x` and `y` will be initialized together. In other words, no other variables will be initialized between them.

A package-level variable declaration with multiple source value expressions will be disassembled into multiple single-source-value variable declarations before initializing all package-level variables. For example,

```
1| var m, n = expr1, expr2
```

is equivalent to

```
1| var m = expr1
2| var n = expr2
```

If hidden dependencies exist between variables, the initialization order between those variables is unspecified. In the following example (copied from Go specification),

- the variable `a` will be initialized after `b` for sure,
- but whether `x` is initialized before `b`, between `b` and `a`, or after `a`, is not specified.

- and the moment at which function `sideEffect()` is called (before or after `x` is initialized) is also not specified.

```

1| // x has a hidden dependency on a and b
2| var x = I(T{}).ab()
3| // Assume sideEffect is unrelated to x, a, and b.
4| var _ = sideEffect()
5| var a = b
6| var b = 42
7|
8| type I interface { ab() []int }
9| type T struct{}
10| func (T) ab() []int { return []int{a, b} }

```

Please note that, Go specification doesn't compulsively specify the compilation order of the source files in a package, so try not to put some package-level variables into different source files in a package if there are complicate dependency relations between those variables; otherwise a variable might be initialized to different values by different Go compilers.

Operand Evaluation Orders in Logical Operations

In a bool expression `a && b`, the right operand expression `b` will be evaluated only if the left operand expression `a` is evaluated as `true`. So `b` will be evaluated, if it needs to be evaluated, after the evaluation of `a`.

In a bool expression `a || b`, the right operand expression `b` will be evaluated only if the left operand expression `a` is evaluated as `false`. So `b` will be evaluated, if it needs to be evaluated, after the evaluation of `a`.

The Usual Order

For the evaluations within a function body, Go specification says

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

The just described order is called ***the usual order***.

Please note that an explicit value conversion `T(v)` is not a function call.

For example, in an expression `[]int{x, fa(), fb(), y}`, assume `x` and `y` are two variables, `fa` and `fb` are two functions, then the call `fa()` is guaranteed to be evaluated (executed) before `fb()`. However, the following the evaluation orders are unspecified in Go specification:

- the evaluation order of `x` (or `y`) and `fa()` (or `fb()`).
- the evaluation order of `x, y, fa` and `fb`.

Another example, the following assignment, is demoed in Go specification.

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

where

- `c` is a channel expression and will be evaluated to a channel value.
- `g, h, i, j` and `k` are function expressions.
- `f` is a method of expression `z`.

Also considering the rule mentioned in the last section, compilers should guarantee the following evaluation orders at run time.

- The function calls, method calls and channel communication operations happen in the order `z.f() → h() → i() → j() → <-c → g() → k()`.
- `h()` is evaluated after the evaluations of expressions `h, a` and `b`.
- `y[]` is evaluated after the evaluation of `z.f()`.
- `z.f()` is evaluated after the evaluation of expression `z`.
- `x[]` is evaluated after the evaluation of `j()`.

However, the following orders (and more others) are not specified.

- The evaluation order of `y, z, g, h, a, b, x, i, j, c` and `k`.
- The evaluation order of `y[], x[]` and `<-c`.

By the usual order, we know the following declared variables `x, m` and `n` (also demoed in Go specification) will be initialized with ambiguous values.

```

1|     a := 1
2|     f := func() int { a++; return a }
3|
4|     // x may be [1, 2] or [2, 2]: evaluation order
5|     // between a and f() is not specified.
6|     x := []int{a, f()}
7|
8|     // m may be {2: 1} or {2: 2}: evaluation order

```

```

9| // between the two map element assignments is
10| // not specified.
11| m := map[int]int{a: 1, a: 2}
12|
13| // n may be {2: 3} or {3: 3}: evaluation order
14| // between the key and the value is unspecified.
15| n := map[int]int{a: f()}

```

Evaluation and Assignment Orders in Assignment Statements

Beside the above introduced rules, Go specification specifies more on the expression evaluation order the order of individual assignments in an assignment statement:

The assignment proceeds in two phases. First, the operands of index expressions and pointer indirection (including implicit pointer indirection 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.

Later, we may call the first phase as evaluation phase and the second phase as carry-out phase.

Go specification doesn't specify clearly whether or not the assignments carried-out during the second phase may affect the expression evaluation results got in the first phase, which ever caused [some disputes](#). So, here, this article will explain more on the evaluation orders in value assignments.

Firstly, let's clarify that the assignments carried-out during the second phase don't affect the expression evaluation results got at the end of the first phase.

To make the following explanations convenient, we assume that the container (slice or map) value of an index destination expression in an assignment is always addressable. If it is not, we can think the container value has already been saved in and replaced by a temporary addressable container value before carrying out the second phase.

At the time of the end of the evaluation phase and just before the carry-out phase starts, each destination expression on the left of an assignment has been evaluated as its elementary form. Different destination expressions have different elementary forms.

- If a destination expression is a blank identifier, then its elementary form is still a blank identifier.
- If a destination expression is a container (array, slice or map) index expression `c[k]`, then its elementary form is `(*cAddr)[k]`, where `cAddr` is a pointer pointing to `c`.

- For other cases, the destination expression must result an addressable value, then its elementary form is a dereference to the address of the destination expression evaluation result.

Assume `a` and `b` are two addressable variables of the same type, the following assignment

```
1| a, b = b, a
```

will be executed like the following steps:

```
1| // The evaluation phase:
2| P0 := &a; P1 := &b
3| R0 := b; R1 := a
4|
5| // The elementary form: *P0, *P1 = R0, R1
6|
7| // The carry-out phase:
8| *P0 = R0
9| *P1 = R1
```

Here is another example, in which `x[0]` instead of `x[1]` is modified.

```
1| x := []int{0, 0}
2| i := 0
3| i, x[i] = 1, 2
4| fmt.Println(x) // [2 0]
```

The decomposed execution steps for the line 3 shown below are like:

```
1| // The evaluation phase:
2| P0 := &i; P1 := &x; T2 := i
3| R0 := 1; R1 := 2
4| // Now, T2 == 0
5|
6| // The elementary form: *P0, (*P1)[T2] = R0, R1
7|
8| // The carry-out phase:
9| *P0 = R0
10| (*P1)[T2] = R1
```

An example which is a little more complex.

```
1| package main
2|
3| import "fmt"
4|
```

```

5| func main() {
6|     m := map[string]int{"Go": 0}
7|     s := []int{1, 1, 1}; olds := s
8|     n := 2
9|     p := &n
10|    s, m["Go"], *p, s[n] = []int{2, 2, 2}, s[1], m["Go"], 5
11|    fmt.Println(m, s, n) // map[Go:1] [2 2 2] 0
12|    fmt.Println(olds)    // [1 1 5]
13| }

```

The decomposed execution steps for the line 10 shown below are like:

```

1| // The evaluation phase:
2| P0 := &s; PM1 := &m; K1 := "Go"; P2 := p; PS3 := &s; T3 := 2
3| R0 := []int{2, 2, 2}; R1 := s[1]; R2 := m["Go"]; R3 := 5
4| // now, R1 == 1, R2 == 0
5|
6| // The elementary form:
7| //      *P0, (*PM1)[K1], *P2, (*PS3)[T3] = R0, R1, R2, R3
8|
9| // The carry-out phase:
10| *P0 = R0
11| (*PM1)[K1] = R1
12| *P2 = R2
13| (*PS3)[T3] = R3

```

The following example rotates all elements in a slice for one index.

```

1|     x := []int{2, 3, 5, 7, 11}
2|     t := x[0]
3|     var i int
4|     for i, x[i] = range x {}
5|     x[i] = t
6|     fmt.Println(x) // [3 5 7 11 2]

```

Another example:

```

1|     x := []int{123}
2|     x, x[0] = nil, 456           // will not panic
3|     x, x[0] = []int{123}, 789 // will panic

```

Although it is legal, it is not recommended to use complex multi-value assignments in Go, for their readability is not good and they have negative effects on both compilation speed and execution performance.

As mentioned above, not all orders are specified in Go specification for value assignments, so some bad written code may produce different results. In the following example, the expression order of `x+1` and `f(&x)` is not specified. So the example may print `100 99` or `1 99`.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     f := func (p *int) int {
7|         *p = 99
8|         return *p
9|     }
10|
11|     x := 0
12|     y, z := x+1, f(&x)
13|     fmt.Println(y, z)
14| }
```

The following is another example which will print ambiguous results. It may print `1 7 2`, `1 8 2` or `1 9 2`, depending on different compiler implementations.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     x, y := 0, 7
7|     f := func() int {
8|         x++
9|         y++
10|        return x
11|    }
12|    fmt.Println(f(), y, f())
13| }
```

Expression Evaluation Orders in switch-case Code Blocks

The expression evaluation order in a `switch-case` code block has been [described before](#) (§12). Here just shows an example. Simply speaking, before a branch code block is entered, the case

expressions will be evaluated and compared with the switch expression one by one, until a comparison results in `true`.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     f := func(n int) int {
7|         fmt.Printf("f(%v) is called.\n", n)
8|         return n
9|     }
10|
11|     switch x := f(3); x + f(4) {
12|     default:
13|     case f(5):
14|     case f(6), f(7), f(8):
15|     case f(9), f(10):
16|     }
17| }
```

At run time, the `f()` calls will be evaluated by the order from top to bottom and from left to right, until a comparison results in `true`. So `f(8)`, `f(9)` and `f(10)` will be not evaluated in this example.

The output:

```

f(3) is called.
f(4) is called.
f(5) is called.
f(6) is called.
f(7) is called.
```

Expression Evaluation Orders in select-case Code Blocks

When executing a `select-case` code block, before entering a branch code block, all the channel operands of receive operations and the operands of send statements involved in the `select-case` code block are evaluated exactly once, in source order (from top to bottom, from left to right).

Note, the target expression being assigned to by a receive `case` operation will only be evaluated if that receive operation is selected later.

In the following example, the expression `*fptr("aaa")` will never get evaluated, for its corresponding receive operation `<-fchan("bbb", nil)` will not be selected.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     c := make(chan int, 1)
7|     c <- 0
8|     fchan := func(info string, c chan int) chan int {
9|         fmt.Println(info)
10|        return c
11|    }
12|    fptr := func(info string) *int {
13|        fmt.Println(info)
14|        return new(int)
15|    }
16|
17|    select {
18|        case *fptr("aaa") = <-fchan("bbb", nil): // blocking
19|        case *fptr("ccc") = <-fchan("ddd", c):   // non-blocking
20|        case fchan("eee", nil) <- *fptr("fff"):   // blocking
21|        case fchan("ggg", nil) <- *fptr("hhh"):   // blocking
22|    }
23| }
```

The output of the above program:

```

bbb
ddd
eee
fff
ggg
hhh
ccc
```

Note that the expression `*fptr("ccc")` is the last evaluated expression in the above example. It is evaluated after its corresponding receive operation `<-fchan("ddd", c)` is selected.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit

tapirgames.com  to get more information about these games. Hope you enjoy them.)

Go Value Copy Costs

Value copying happens frequently in Go programming. Values assignments, argument passing and channel value send operations are all value copying involved. This article will talk about the copy costs of values of all kinds of types.

Value Sizes

The size of a value means how many bytes the [direct part](#) (§17) of the value will occupy in memory. The indirect underlying parts of a value don't contribute to the size of the value.

In Go, if the types of two values belong to the same [kind](#) (§14), and the type kind is not string kind, interface kind, array kind and struct kind, then the sizes of the two value are always equal.

In fact, for the standard Go compiler/runtime, the sizes of two string values are also always equal. The same relation is for the sizes of two interface values.

Up to present (Go Toolchain 1.21.n), for the standard Go compiler (and gccgo), values of a specified type always have the same value size. So, often, we call the size of a value as the size of the type of the value.

The size of an array type depends on the element type size and the length of the array type. The array type size is the product of the size of the array element type and the array length.

The size of a struct type depends on all of the sizes and the order of its fields. For there may be some [padding bytes](#) (§44) being inserted between two adjacent struct fields to guarantee certain memory address alignment requirements of these fields, so the size of a struct type must be not smaller than (and often larger than) the sum of the respective type sizes of its fields.

The following table lists the value sizes of all kinds of types (for the standard Go compiler v1.21.n). In the table, one word means one native word, which is 4 bytes on 32-bit architectures and 8 bytes on 64-bit architectures.

Kinds of Types	Value Size	Required by Go Specification
bool	1 byte	not specified
int8, uint8 (byte)	1 byte	1 byte
int16, uint16	2 bytes	2 bytes

Kinds of Types	Value Size	Required by Go Specification
int32 (rune), uint32, float32	4 bytes	4 bytes
int64, uint64, float64, complex64	8 bytes	8 bytes
complex128	16 bytes	16 bytes
int, uint	1 word	architecture dependent, 4 bytes on 32-bit architectures and 8 bytes on 64-bit architectures
uintptr	1 word	large enough to store the uninterpreted bits of a pointer value
string	2 words	not specified
pointer (safe or unsafe)	1 word	not specified
slice	3 words	not specified
map	1 word	not specified
channel	1 word	not specified
function	1 word	not specified
interface	2 words	not specified
struct	(the sum of sizes of all fields) + (the number of padding (§44) bytes)	the size of a struct type is zero if it contains no fields that have a size greater than zero
array	(element value size) * (array length)	the size of an array type is zero if its element type has zero size

Value Copy Costs

Generally speaking, the cost to copy a value is proportional to the size of the value. However, value sizes are not the only factor determining value copy costs. Different CPU models and compiler versions may specially optimize value copying for values with specific sizes.

In practice, we can view struct values with less than 5 fields and with sizes not larger than four native words as small-size values. The costs of copying small-size values are small.

For the standard Go compiler, except values of large-size struct and array types, other types in Go are all small-size types.

To avoid large value copy costs in argument passing and channel value send and receive operations, we should try to avoid using large-size struct and array types as function and method parameter types (including method receiver types) and channel element types. We can use pointer types whose base types are large-size types instead for such scenarios.

On the other hand, we should also consider the fact that too many pointers will increase the pressure of garbage collectors at run time. So whether large-size struct and array types or their corresponding pointer types should be used relies on specific circumstances.

Generally, in practice, we seldom use pointer types whose base types are slice types, map types, channel types, function types, string types and interface types. The costs of copying values of these assumed base types are very small.

We should also try to avoid using the two-iteration-variable forms to iterate array and slice elements if the element types are large-size types, for each element value will be copied to the second iteration variable in the iteration process.

The following is an example which benchmarks different ways to iterate slice elements.

```

1| package main
2|
3| import "testing"
4|
5| type S [12]int64
6| var sX = make([]S, 1000)
7| var sY = make([]S, 1000)
8| var sZ = make([]S, 1000)
9| var sumX, sumY, sumZ int64
10|
11| func Benchmark_Loop(b *testing.B) {
12|     for i := 0; i < b.N; i++ {
13|         sumX = 0
14|         for j := 0; j < len(sX); j++ {
15|             sumX += sX[j][0]
16|         }
17|     }
18| }
19|
20| func Benchmark_Range_OneIterVar(b *testing.B) {
21|     for i := 0; i < b.N; i++ {
22|         sumY = 0
23|         for j := range sY {
24|             sumY += sY[j][0]
25|         }

```

```

26|     }
27| }
28|
29| func Benchmark_Range_TwoIterVar(b *testing.B) {
30|     for i := 0; i < b.N; i++ {
31|         sumZ = 0
32|         for _, v := range sZ {
33|             sumZ += v[0]
34|         }
35|     }
36| }
```

Run the benchmarks in the directory of the test file, we will get a result similar to:

Benchmark_Loop-4	424342 2708 ns/op
Benchmark_Range_OneIterVar-4	407905 2808 ns/op
Benchmark_Range_TwoIterVar-4	214860 5222 ns/op

We can find that the efficiency of the two-iteration-variable form is much lower than the other two. But please note that, some compilers might make special optimizations to remove the performance differences between these forms. The above benchmark result is based on the standard Go compiler v1.21.n.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Bounds Check Elimination

Go is a memory safe language. In array/slice element indexing and subslice operations, Go runtime will check whether or not the involved indexes are out of range. If an index is out of range, a panic will be produced to prevent the invalid index from doing harm. This is called bounds check. Bounds checks make our code run safely, on the other hand, they also make our code run a little slower.

Since Go Toolchain 1.7, the standard Go compiler has used a new compiler backend, which based on SSA (static single-assignment form). SSA helps Go compilers effectively use optimizations like [BCE ↗](#) (bounds check elimination) and [CSE ↗](#) (common subexpression elimination). BCE can avoid some unnecessary bounds checks, and CSE can avoid some duplicate calculations, so that the standard Go compiler can generate more efficient programs. Sometimes the improvement effects of these optimizations are obvious.

This article will list some examples to show how BCE works with the standard Go compiler 1.7+.

For Go Toolchain 1.7+, we can use `-gcflags="-d=ssa/check_bce"` compiler flag to show which code lines still need bounds checks.

Example 1

```

1| // example1.go
2| package main
3|
4| func f1(s []int) {
5|     _ = s[0] // line 5: bounds check
6|     _ = s[1] // line 6: bounds check
7|     _ = s[2] // line 7: bounds check
8| }
9|
10| func f2(s []int) {
11|     _ = s[2] // line 11: bounds check
12|     _ = s[1] // line 12: bounds check eliminated!
13|     _ = s[0] // line 13: bounds check eliminated!
14| }
15|
16| func f3(s []int, index int) {
17|     _ = s[index] // line 17: bounds check
18|     _ = s[index] // line 18: bounds check eliminated!
19| }
```

```

20|
21| func f4(a [5]int) {
22|     _ = a[4] // line 22: bounds check eliminated!
23| }
24|
25| func main() {}

```

```

$ go run -gcflags="-d=ssa/check_bce" example1.go
./example1.go:5: Found IsInBounds
./example1.go:6: Found IsInBounds
./example1.go:7: Found IsInBounds
./example1.go:11: Found IsInBounds
./example1.go:17: Found IsInBounds

```

We can see that there are no needs to do bounds checks for line 12 and line 13 in function `f2`, for the bounds check at line 11 ensures that the indexes in line 12 and line 13 will not be out of range.

But in function `f1`, bounds checks must be performed for all three lines. The bounds check at line 5 can't ensure line 6 and line 7 are safe, and the bounds check at line 6 can't ensure line 7 is safe.

For function `f3`, the compiler knows the second `s[index]` is absolutely safe if the first `s[index]` is safe.

The compiler also correctly thinks the only line (line 22) in function `f4` is safe.

Please note that, up to now (Go Toolchain 1.21.n), the standard compiler doesn't check BCE for an operation in a generic function if the operation involves type parameters and the generic function is never instantiated.

A demo case:

```

1| // example1b.go
2| package main
3|
4| func foo[E any](s []E) {
5|     _ = s[0] // line 5
6|     _ = s[1] // line 6
7|     _ = s[2] // line 7
8|
9|
10| // var _ = foo[bool]
11|
12| func main() {
13| }

```

When the variable declaration line is disabled, the compiler outputs nothing:

```
$ go run -gcflags="-d=ssa/check_bce" example1b.go
```

When the variable declaration line is enabled, the compiler outputs:

```
./aaa.go:5:7: Found IsInBounds
./example1b.go:6:7: Found IsInBounds
./example1b.go:7:7: Found IsInBounds
./example1b.go:4:6: Found IsInBounds
```

Example 2

```
1| // example2.go
2| package main
3|
4| func f5(s []int) {
5|     for i := range s {
6|         _ = s[i]
7|         _ = s[i:len(s)]
8|         _ = s[:i+1]
9|     }
10| }
11|
12| func f6(s []int) {
13|     for i := 0; i < len(s); i++ {
14|         _ = s[i]
15|         _ = s[i:len(s)]
16|         _ = s[:i+1]
17|     }
18| }
19|
20| func f7(s []int) {
21|     for i := len(s) - 1; i >= 0; i-- {
22|         _ = s[i]
23|         _ = s[i:len(s)]
24|     }
25| }
26|
27| func f8(s []int, index int) {
28|     if index >= 0 && index < len(s) {
29|         _ = s[index]
30|         _ = s[index:len(s)]
31|     }
```

```

32| }
33|
34| func f9(s []int) {
35|     if len(s) > 2 {
36|         _, _, _ = s[0], s[1], s[2]
37|     }
38| }
39|
40| func main() {}

```

```
$ go run -gcflags="-d=ssa/check_bce" example2.go
```

Cool! The standard compiler removes all bound checks in this program.

Note: before Go Toolchain version 1.11, the standard compiler is not smart enough to detect line 22 is safe.

Example 3

```

1| // example3.go
2| package main
3|
4| import "math/rand"
5|
6| func fa() {
7|     s := []int{0, 1, 2, 3, 4, 5, 6}
8|     index := rand.Intn(7)
9|     _ = s[:index] // line 9: bounds check
10|    _ = s[index:] // line 10: bounds check eliminated!
11| }
12|
13| func fb(s []int, i int) {
14|     _ = s[:i] // line 14: bounds check
15|     _ = s[i:] // line 15: bounds check, not smart enough?
16| }
17|
18| func fc() {
19|     s := []int{0, 1, 2, 3, 4, 5, 6}
20|     s = s[:4]
21|     i := rand.Intn(7)
22|     _ = s[:i] // line 22: bounds check
23|     _ = s[i:] // line 23: bounds check, not smart enough?
24| }

```

```
25|
26| func main() {}
```

```
$ go run -gcflags="-d=ssa/check_bce" example3.go
./example3.go:9: Found IsSliceInBounds
./example3.go:14: Found IsSliceInBounds
./example3.go:15: Found IsSliceInBounds
./example3.go:22: Found IsSliceInBounds
./example3.go:23: Found IsSliceInBounds
```

Oh, so many places still need to do bounds check!

But wait, why does the standard Go compiler think line 10 is safe but line 15 and line 23 are not? Is the compiler still not smart enough?

In fact, the compiler is right here! Why? The reason is the start index in a subslice expression may be larger than the length of the base slice. Let's view a simple example:

```
1| package main
2|
3| func main() {
4|     s0 := make([]int, 5, 10) // len(s0) == 5, cap(s0) == 10
5|
6|     index := 8
7|
8|     // In Go, for the subslice syntax s[a:b],
9|     // the relations 0 <= a <= b <= cap(s) must
10|    // be ensured to avoid panicking.
11|
12|    _ = s0[:index]
13|    // The above line is safe can't ensure the
14|    // following line is also safe. In fact, the
15|    // following line will panic, for the starting
16|    // index is larger than the end index.
17|    _ = s0[index:] // panic
18| }
```

So the conclusion that **if `s[:index]` is safe then `s[index:]` is also safe** is only right when `len(s)` is equal to `cap(s)`. This is why the code lines in function `fb` and `fc` of example 3 still need to do bounds checks.

Standard Go compiler successfully detects `len(s)` is equal to `cap(s)` in function `fa`. Great work! Go team!

Example 4

```

1| // example4.go
2| package main
3|
4| import "math/rand"
5|
6| func fb2(s []int, index int) {
7|     _ = s[index:] // line 7: bounds check
8|     _ = s[:index] // Line 8: bounds check eliminated!
9| }
10|
11| func fc2() {
12|     s := []int{0, 1, 2, 3, 4, 5, 6}
13|     s = s[:4]
14|     index := rand.Intn(7)
15|     _ = s[index:] // line 15 bounds check
16|     _ = s[:index] // line 16: bounds check eliminated!
17| }
18|
19| func main() {}

```

```

$ go run -gcflags="-d=ssa/check_bce" example4.go
./example4.go:7:7: Found IsSliceInBounds
./example4.go:15:7: Found IsSliceInBounds

```

In this example, The standard Go compiler successfully concludes

- line 8 is also safe if line 7 is safe in function `fb2`.
- line 16 is also safe if line 15 is safe in function `fc2`.

Note: the standard Go compiler in Go Toolchain earlier than version 1.9 fails to detect line 8 doesn't need bounds check.

Example 5

The current version of the standard Go compiler is not smart enough to eliminate all unnecessary bounds checks. Sometimes, we can make some hints to help the compiler eliminate some unnecessary bounds checks.

```

1| // example5.go
2| package main

```

```

3|
4| func fd(is []int, bs []byte) {
5|     if len(is) >= 256 {
6|         for _, n := range bs {
7|             _ = is[n] // line 7: bounds check
8|         }
9|     }
10| }
11|
12| func fd2(is []int, bs []byte) {
13|     if len(is) >= 256 {
14|         is = is[:256] // line 14: a hint
15|         for _, n := range bs {
16|             _ = is[n] // line 16: BCEed!
17|         }
18|     }
19| }
20|
21| func main() {}

```

```
$ go run -gcflags="-d=ssa/check_bce" example5.go
./example5.go:7: Found IsInBounds
```

Summary

There are more BCE optimizations made by the standard Go compiler. They might be not as obvious as the above listed ones, So this article will not show them all.

Although the BCE feature in the standard Go compiler is still not perfect, it really does well for many common cases. It is no doubt that standard Go compiler will do better in later versions so that it is possible the hints made in the above 5th example will become unnecessary. Thank Go team for adding this wonderful feature!

References:

1. [Bounds Check Elimination ↗](#)
2. [Utilizing the Go 1.7 SSA Compiler ↗](#) (and [the second part ↗](#))

(The **Go 101** book is still being improved frequently from time to time. Please visit [go101.org ↗](#) or follow [@go100and1 ↗](#) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit

tapirgames.com  to get more information about these games. Hope you enjoy them.)

Concurrency Synchronization Overview

This article will explain what are synchronizations and list the synchronization techniques supported by Go.

What Are Concurrency Synchronizations?

Concurrency synchronizations means how to control concurrent computations (a.k.a., goroutines in Go)

- to avoid data races between them,
- to avoid them consuming CPU resources when they have nothing to do.

What Synchronization Techniques Does Go Support?

The article [channels in Go](#) (§21) has shown that we can use channels to do synchronizations.

Besides using channels, Go also supports several other common synchronization techniques, such as mutex and atomic operations. Please read the following articles to get how to do synchronizations with all kinds of techniques in Go:

- [Channel Use Cases](#) (§37)
- [How to Gracefully Close Channels](#) (§38)
- [Concurrency Synchronization Techniques Provided in the sync Standard Package](#) (§39)
- [Atomic Operations Provided in the sync/atomic Standard Package](#) (§40)

We can also do synchronizations by making use of network and file IO. But such techniques are very inefficient within a single program process. Generally, they are used for inter-process and distributed synchronizations. Go 101 will not cover such techniques.

To understand these synchronization techniques better, it is recommended to know the [memory order guarantees in Go](#) (§41).

The data synchronization techniques in Go will not prevent programmers from writing [improper concurrent code](#) (§42). However these techniques can help programmers write correct concurrent code easily. And the unique channel related features make concurrent programming flexible and enjoyable.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Channel Use Cases

Before reading this article, please read the article [channels in Go](#) (§21), which explains channel types and values in detail. New gophers may need to read that article and the current one several times to master Go channel programming.

The remaining of this article will show many channel use cases. I hope this article will convince you that

- asynchronous and concurrency programming with Go channels is easy and enjoyable.
- the channel synchronization technique has a wider range of uses and has more variations than the synchronization solutions used in some other languages, such as [the actor model](#) ↗ and the [async/await pattern](#) ↗.

Please note that the intention of this article is to show as many channel use cases as possible. We should know that channel is not the only concurrency synchronization technique supported in Go, and for some cases, the channel way may not be the best solution. Please read [atomic operations](#) (§40) and [some other synchronization techniques](#) (§39) for more concurrency synchronization techniques in Go.

Use Channels as Futures/Promises

Futures and promises are used in many other popular languages. They are often associated with requests and responses.

Return receive-only channels as results

In the following example, the values of two arguments of the `sumSquares` function call are requested concurrently. Each of the two channel receive operations will block until a send operation performs on the corresponding channel. It takes about three seconds instead of six seconds to return the final result.

```

1| package main
2|
3| import (
4|     "time"
5|     "math/rand"
6|     "fmt"
```

```

7| )
8|
9| func longTimeRequest() <-chan int32 {
10|     r := make(chan int32)
11|
12|     go func() {
13|         // Simulate a workload.
14|         time.Sleep(time.Second * 3)
15|         r <- rand.Int31n(100)
16|     }()
17|
18|     return r
19| }
20|
21| func sumSquares(a, b int32) int32 {
22|     return a*a + b*b
23| }
24|
25| func main() {
26|     rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
27|
28|     a, b := longTimeRequest(), longTimeRequest()
29|     fmt.Println(sumSquares(<-a, <-b))
30| }
```

Pass send-only channels as arguments

Same as the last example, in the following example, the values of two arguments of the `sumSquares` function call are requested concurrently. Different to the last example, the `longTimeRequest` function takes a send-only channel as parameter instead of returning a receive-only channel result.

```

1| package main
2|
3| import (
4|     "time"
5|     "math/rand"
6|     "fmt"
7| )
8|
9| func longTimeRequest(r chan<- int32) {
10|     // Simulate a workload.
11|     time.Sleep(time.Second * 3)
```

```

12|     r <- rand.Int31n(100)
13| }
14|
15| func sumSquares(a, b int32) int32 {
16|     return a*a + b*b
17| }
18|
19| func main() {
20|     rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
21|
22|     ra, rb := make(chan int32), make(chan int32)
23|     go longTimeRequest(ra)
24|     go longTimeRequest(rb)
25|
26|     fmt.Println(sumSquares(<-ra, <-rb))
27| }
```

In fact, for the above specified example, we don't need two channels to transfer results. Using one channel is okay.

```

1| ...
2|
3|     // The channel can be buffered or not.
4|     results := make(chan int32, 2)
5|     go longTimeRequest(results)
6|     go longTimeRequest(results)
7|
8|     fmt.Println(sumSquares(<-results, <-results))
9| }
```

This is kind of data aggregation which will be introduced specially below.

The first response wins

This is the enhancement of the using-only-one-channel variant in the last example.

Sometimes, a piece of data can be received from several sources to avoid high latencies. For a lot of factors, the response durations of these sources may vary much. Even for a specified source, its response durations are also not constant. To make the response duration as short as possible, we can send a request to every source in a separated goroutine. Only the first response will be used, other slower ones will be discarded.

Note, if there are N sources, the capacity of the communication channel must be at least $N-1$, to avoid the goroutines corresponding the discarded responses being blocked for ever.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "time"
6|     "math/rand"
7| )
8|
9| func source(c chan<- int32) {
10|     ra, rb := rand.Int31(), rand.Intn(3) + 1
11|     // Sleep 1s/2s/3s.
12|     time.Sleep(time.Duration(rb) * time.Second)
13|     c <- ra
14| }
15|
16| func main() {
17|     rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
18|
19|     startTime := time.Now()
20|     // c must be a buffered channel.
21|     c := make(chan int32, 5)
22|     for i := 0; i < cap(c); i++ {
23|         go source(c)
24|     }
25|     // Only the first response will be used.
26|     rnd := <- c
27|     fmt.Println(time.Since(startTime))
28|     fmt.Println(rnd)
29| }
```

There are some other ways to implement the first-response-win use case, by using the select mechanism and a buffered channel whose capacity is one. Other ways will be introduced below.

More request-response variants

The parameter and result channels can be buffered so that the response sides won't need to wait for the request sides to take out the transferred values.

Sometimes, a request is not guaranteed to be responded back a valid value. For all kinds of reasons, an error may be returned instead. For such cases, we can use a struct type like `struct{v T; err`

`error}` or a blank interface type as the channel element type.

Sometimes, for some reasons, the response may need a much longer time than the expected to arrive, or will never arrive. We can use the timeout mechanism introduced below to handle such circumstances.

Sometimes, a sequence of values may be returned from the response side, this is kind of the data flow mechanism mentioned later below.

Use Channels for Notifications

Notifications can be viewed as special requests/responses in which the responded values are not important. Generally, we use the blank struct type `struct{}` as the element types of the notification channels, for the size of type `struct{}` is zero, hence values of `struct{}` doesn't consume memory.

1-to-1 notification by sending a value to a channel

If there are no values to be received from a channel, then the next receive operation on the channel will block until another goroutine sends a value to the channel. So we can send a value to a channel to notify another goroutine which is waiting to receive a value from the same channel.

In the following example, the channel `done` is used as a signal channel to do notifications.

```

1| package main
2|
3| import (
4|     "crypto/rand"
5|     "fmt"
6|     "os"
7|     "sort"
8| )
9|
10| func main() {
11|     values := make([]byte, 32 * 1024 * 1024)
12|     if _, err := rand.Read(values); err != nil {
13|         fmt.Println(err)
14|         os.Exit(1)
15|     }
16|
17|     done := make(chan struct{}) // can be buffered or not

```

```

18|
19|     // The sorting goroutine
20|     go func() {
21|         sort.Slice(values, func(i, j int) bool {
22|             return values[i] < values[j]
23|         })
24|         // Notify sorting is done.
25|         done <- struct{}{}
26|     }()
27|
28|     // do some other things ...
29|
30|     <- done // waiting here for notification
31|     fmt.Println(values[0], values[len(values)-1])
32|

```

1-to-1 notification by receiving a value from a channel

If the value buffer queue of a channel is full (the buffer queue of an unbuffered channel is always full), a send operation on the channel will block until another goroutine receives a value from the channel. So we can receive a value from a channel to notify another goroutine which is waiting to send a value to the same channel. Generally, the channel should be an unbuffered channel.

This notification way is used much less common than the way introduced in the last example.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "time"
6| )
7|
8| func main() {
9|     done := make(chan struct{})
10|    // The capacity of the signal channel can
11|    // also be one. If this is true, then a
12|    // value must be sent to the channel before
13|    // creating the following goroutine.
14|
15|    go func() {
16|        fmt.Println("Hello")
17|        // Simulate a workload.
18|        time.Sleep(time.Second * 2)

```

```

19|         // Receive a value from the done
20|         // channel, to unblock the second
21|         // send in main goroutine.
22|         <- done
23|
24|     }()
25|
26|     // Blocked here, wait for a notification.
27|     done <- struct{}{}
28|     fmt.Println(" world!")
29| }
```

In fact, there are no fundamental differences between receiving or sending values to make notifications. They can both be summarized as the fasters are notified by the slower.

N-to-1 and 1-to-N notifications

By extending the above two use cases a little, it is easy to do N-to-1 and 1-to-N notifications.

```

1| package main
2|
3| import "log"
4| import "time"
5|
6| type T = struct{}
7|
8| func worker(id int, ready <-chan T, done chan<- T) {
9|     <-ready // block here and wait a notification
10|    log.Println("Worker#", id, " starts.")
11|    // Simulate a workload.
12|    time.Sleep(time.Second * time.Duration(id+1))
13|    log.Println("Worker#", id, " job done .")
14|    // Notify the main goroutine (N-to-1),
15|    done <- T{}
16| }
17|
18| func main() {
19|     log.SetFlags(0)
20|
21|     ready, done := make(chan T), make(chan T)
22|     go worker(0, ready, done)
23|     go worker(1, ready, done)
24|     go worker(2, ready, done)
25| }
```

```

26|     // Simulate an initialization phase.
27|     time.Sleep(time.Second * 3 / 2)
28|     // 1-to-N notifications.
29|     ready <- T{}; ready <- T{}; ready <- T{}
30|     // Being N-to-1 notified.
31|     <-done; <-done; <-done
32|

```

In fact, the ways to do 1-to-N and N-to-1 notifications introduced in this sub-section are not used commonly in practice. In practice, we often use `sync.WaitGroup` to do N-to-1 notifications, and we do 1-to-N notifications by close channels. Please read the next sub-section for details.

Broadcast (1-to-N) notifications by closing a channel

The way to do 1-to-N notifications shown in the last sub-section is seldom used in practice, for there is a better way. By making use of the feature that infinite values can be received from a closed channel, we can close a channel to broadcast notifications.

By the example in the last sub-section, we can replace the three channel send operations `ready <- struct{}{}` in the last example with one channel close operation `close(ready)` to do an 1-to-N notifications.

```

1| ...
2|     close(ready) // broadcast notifications
3| ...

```

Surely, we can also close a channel to do a 1-to-1 notification. In fact, this is the most used notification way in Go.

The feature that infinite values can be received from a closed channel will be utilized in many other use cases introduced below. In fact, the feature is used popularly in the standard packages. For example, the `context` package uses this feature to confirm cancellations.

Timer: scheduled notification

It is easy to use channels to implement one-time timers.

A custom one-time timer implementation:

```

1| package main
2|
3| import (

```

```

4|     "fmt"
5|     "time"
6| )
7|
8| func AfterDuration(d time.Duration) <- chan struct{} {
9|     c := make(chan struct{}, 1)
10|    go func() {
11|        time.Sleep(d)
12|        c <- struct{}{}
13|    }()
14|    return c
15| }
16|
17| func main() {
18|     fmt.Println("Hi!")
19|     <- AfterDuration(time.Second)
20|     fmt.Println("Hello!")
21|     <- AfterDuration(time.Second)
22|     fmt.Println("Bye!")
23| }
```

In fact, the `After` function in the `time` standard package provides the same functionality, with a much more efficient implementation. We should use that function instead to make the code look clean.

Please note, `<-time.After(aDuration)` will make the current goroutine enter blocking state, but a `time.Sleep(aDuration)` function call will not.

The use of `<-time.After(aDuration)` is often used in the timeout mechanism which will be introduced below.

Use Channels as Mutex Locks

One of the above examples has mentioned that one-capacity buffered channels can be used as one-time [binary semaphore](#). In fact, such channels can also be used as multi-time binary semaphores, a.k.a., mutex locks, though such mutex locks are not efficient as the mutexes provided in the `sync` standard package.

There are two manners to use one-capacity buffered channels as mutex locks.

1. Lock through a send, unlock through a receive.
2. Lock through a receive, unlock through a send.

The following is a lock-through-send example.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     // The capacity must be one.
7|     mutex := make(chan struct{}, 1)
8|
9|     counter := 0
10|    increase := func() {
11|        mutex <- struct{}{} // lock
12|        counter++
13|        <-mutex // unlock
14|    }
15|
16|    increase1000 := func(done chan<- struct{}) {
17|        for i := 0; i < 1000; i++ {
18|            increase()
19|        }
20|        done <- struct{}{}
21|    }
22|
23|    done := make(chan struct{})
24|    go increase1000(done)
25|    go increase1000(done)
26|    <-done; <-done
27|    fmt.Println(counter) // 2000
28| }
```

The following is a lock-through-receive example. It just shows the modified part based on the above lock-through-send example.

```

1| ...
2| func main() {
3|     mutex := make(chan struct{}, 1)
4|     mutex <- struct{}{} // this line is needed.
5|
6|     counter := 0
7|     increase := func() {
8|         <-mutex // lock
9|         counter++
10|        mutex <- struct{}{} // unlock
```

```

11|     }
12| ...

```

Use Channels as Counting Semaphores

Buffered channels can be used as [counting semaphores ↗](#). Counting semaphores can be viewed as multi-owner locks. If the capacity of a channel is N , then it can be viewed as a lock which can have most N owners at any time. Binary semaphores (mutexes) are special counting semaphores, each of binary semaphores can have at most one owner at any time.

Counting semaphores are often used to enforce a maximum number of concurrent requests.

Like using channels as mutexes, there are also two manners to acquire one piece of ownership of a channel semaphore.

1. Acquire ownership through a send, release through a receive.
2. Acquire ownership through a receive, release through a send.

An example of acquiring ownership through receiving values from a channel.

```

1| package main
2|
3| import (
4|     "log"
5|     "time"
6|     "math/rand"
7| )
8|
9| type Seat int
10| type Bar chan Seat
11|
12| func (bar Bar) ServeCustomer(c int) {
13|     log.Println("customer#", c, " enters the bar")
14|     seat := <- bar // need a seat to drink
15|     log.Println("++ customer#", c, " drinks at seat#", seat)
16|     time.Sleep(time.Second * time.Duration(2 + rand.Intn(6)))
17|     log.Println("-- customer#", c, " frees seat#", seat)
18|     bar <- seat // free seat and leave the bar
19| }
20|
21| func main() {
22|     rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
23|

```

```

24| // the bar has 10 seats.
25| bar24x7 := make(Bar, 10)
26| // Place seats in an bar.
27| for seatId := 0; seatId < cap(bar24x7); seatId++ {
28|     // None of the sends will block.
29|     bar24x7 <- Seat(seatId)
30| }
31|
32| for customerId := 0; ; customerId++ {
33|     time.Sleep(time.Second)
34|     go bar24x7.ServeCustomer(customerId)
35| }
36|
37| // sleeping != blocking
38| for {time.Sleep(time.Second)}
39| }
```

In the above example, only the customers each of whom get a seat can drink. So there will be most ten customers are drinking at any given time.

The last `for` loop in the `main` function is to avoid the program exiting. There is a better way, which will be introduced below, to do the job.

In the above example, although there will be most ten customers are drinking at any given time, there may be more than ten customers are served at the bar at the same time. Some customers are waiting for free seats. Although each customer goroutine consumes much fewer resources than a system thread, the total resources consumed by a large number of goroutines are not negligible. So it is best to create a customer goroutine only if there is an available seat.

```

1| ... // same code as the above example
2|
3| func (bar Bar) ServeCustomerAtSeat(c int, seat Seat) {
4|     log.Println("++ customer#", c, " drinks at seat#", seat)
5|     time.Sleep(time.Second * time.Duration(2 + rand.Intn(6)))
6|     log.Println("-- customer#", c, " frees seat#", seat)
7|     bar <- seat // free seat and leave the bar
8| }
9|
10| func main() {
11|     rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
12|
13|     bar24x7 := make(Bar, 10)
14|     for seatId := 0; seatId < cap(bar24x7); seatId++ {
15|         bar24x7 <- Seat(seatId)
16|     }
```

```

17|
18|     for customerId := 0; ; customerId++ {
19|         time.Sleep(time.Second)
20|         // Need a seat to serve next customer.
21|         seat := <- bar24x7
22|         go bar24x7.ServeCustomerAtSeat(customerId, seat)
23|     }
24|     for {time.Sleep(time.Second)}
25| }
```

There will be at most about ten live customer goroutines coexisting in the above optimized version (but there will still be lots of customer goroutines to be created in the program lifetime).

In a more efficient implementation shown below, at most ten customer serving goroutines will be created in the program lifetime.

```

1| ... // same code as the above example
2|
3| func (bar Bar) ServeCustomerAtSeat(consumers chan int) {
4|     for c := range consumers {
5|         seatId := <- bar
6|         log.Println("++ customer#", c, " drinks at seat#", seatId)
7|         time.Sleep(time.Second * time.Duration(2 + rand.Intn(6)))
8|         log.Println("-- customer#", c, " frees seat#", seatId)
9|         bar <- seatId // free seat and leave the bar
10|    }
11| }
12|
13| func main() {
14|     rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
15|
16|     bar24x7 := make(Bar, 10)
17|     for seatId := 0; seatId < cap(bar24x7); seatId++ {
18|         bar24x7 <- Seat(seatId)
19|     }
20|
21|     consumers := make(chan int)
22|     for i := 0; i < cap(bar24x7); i++ {
23|         go bar24x7.ServeCustomerAtSeat(consumers)
24|     }
25|
26|     for customerId := 0; ; customerId++ {
27|         time.Sleep(time.Second)
28|         consumers <- customerId
29|     }
30| }
```

```
29|     }
30| }
```

Off-topic: surely, if we don't care about seat IDs (which is common in practice), then the bar24x7 semaphore is not essential at all:

```
1| ... // same code as the above example
2|
3| func ServeCustomer(consumers chan int) {
4|     for c := range consumers {
5|         log.Println("++ customer#", c, " drinks at the bar")
6|         time.Sleep(time.Second * time.Duration(2 + rand.Intn(6)))
7|         log.Println("-- customer#", c, " leaves the bar")
8|     }
9| }
10|
11| func main() {
12|     rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
13|
14|     const BarSeatCount = 10
15|     consumers := make(chan int)
16|     for i := 0; i < BarSeatCount; i++ {
17|         go ServeCustomer(consumers)
18|     }
19|
20|     for customerId := 0; ; customerId++ {
21|         time.Sleep(time.Second)
22|         consumers <- customerId
23|     }
24| }
```

The manner of acquiring semaphore ownership through sending is simpler comparatively. The step of placing seats is not needed.

```
1| package main
2|
3| import (
4|     "log"
5|     "time"
6|     "math/rand"
7| )
8|
9| type Customer struct{id int}
10| type Bar chan Customer
11|
```

```

12| func (bar Bar) ServeCustomer(c Customer) {
13|     log.Println("++ customer#", c.id, " starts drinking")
14|     time.Sleep(time.Second * time.Duration(3 + rand.Intn(16)))
15|     log.Println("-- customer#", c.id, " leaves the bar")
16|     <- bar // leaves the bar and save a space
17| }
18|
19| func main() {
20|     rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
21|
22|     // The bar can serve most 10 customers
23|     // at the same time.
24|     bar24x7 := make(Bar, 10)
25|     for customerId := 0; ; customerId++ {
26|         time.Sleep(time.Second * 2)
27|         customer := Customer{customerId}
28|         // Wait to enter the bar.
29|         bar24x7 <- customer
30|         go bar24x7.ServeCustomer(customer)
31|     }
32|     for {time.Sleep(time.Second)}
33| }
```

Dialogue (Ping-Pong)

Two goroutines can dialogue through a channel. The following is an example which will print a series of Fibonacci numbers.

```

1| package main
2|
3| import "fmt"
4| import "time"
5| import "os"
6|
7| type Ball uint64
8|
9| func Play(playerName string, table chan Ball) {
10|     var lastValue Ball = 1
11|     for {
12|         ball := <- table // get the ball
13|         fmt.Println(playerName, ball)
14|         ball += lastValue
15|         if ball < lastValue { // overflow
16|             os.Exit(0)
17| }
```

```

17|     }
18|     lastValue = ball
19|     table <- ball // bat back the ball
20|     time.Sleep(time.Second)
21|   }
22| }
23|
24| func main() {
25|   table := make(chan Ball)
26|   go func() {
27|     table <- 1 // throw ball on table
28|   }()
29|   go Play("A:", table)
30|   Play("B:", table)
31| }
```

Channel Encapsulated in Channel

Sometimes, we can use a channel type as the element type of another channel type. In the following example, `chan chan<- int` is a channel type which element type is a send-only channel type `chan<- int`.

```

1| package main
2|
3| import "fmt"
4|
5| var counter = func (n int) chan<- chan<- int {
6|   requests := make(chan chan<- int)
7|   go func() {
8|     for request := range requests {
9|       if request == nil {
10|         n++ // increase
11|       } else {
12|         request <- n // take out
13|       }
14|     }
15|   }()
16|
17|   // Implicitly converted to chan<- (chan<- int)
18|   return requests
19| }(0)
20|
21| func main() {
```

```

22|     increase1000 := func(done chan<- struct{}) {
23|         for i := 0; i < 1000; i++ {
24|             counter <- nil
25|         }
26|         done <- struct{}{}
27|     }
28|
29|     done := make(chan struct{})
30|     go increase1000(done)
31|     go increase1000(done)
32|     <-done; <-done
33|
34|     request := make(chan int, 1)
35|     counter <- request
36|     fmt.Println(<-request) // 2000
37| }
```

Although here the encapsulation implementation may be not the most efficient way for the above-specified example, the use case may be useful for some other scenarios.

Check Lengths and Capacities of Channels

We can use the built-in functions `len` and `cap` to check the length and capacity of a channel. However, we seldom do this in practice. The reason for we seldom use the `len` function to check the length of a channel is the length of the channel may have changed after the `len` function call returns. The reason for we seldom use the `cap` function to check the capacity of a channel is the capacity of the channel is often known or not important.

However, there do have some scenarios we need to use the two functions. For example, sometimes, we want to receive all the values buffered in a non-closed channel `c` which no ones will send values to any more, then we can use the following code to receive remaining values.

```

1| // Assume the current goroutine is the only
2| // goroutine tries to receive values from
3| // the channel c at present.
4| for len(c) > 0 {
5|     value := <-c
6|     // use value ...
7| }
```

We can also use the try-receive mechanism introduced below to do the same job. The efficiencies of the two ways are almost the same. The advantage of the try-receive mechanism is the current

goroutine is not required to be the only receiving goroutine.

Sometimes, a goroutine may want to write some values to a buffered channel `c` until it is full without entering blocking state at the end, and the goroutine is the only sender of the channel, then we can use the following code to do this job.

```
1| for len(c) < cap(c) {
2|     c <- aValue
3| }
```

Surely, we can also use the try-send mechanism introduced below to do the same job.

Block the Current Goroutine Forever

The select mechanism is a unique feature in Go. It brings many patterns and tricks for concurrent programming. About the code execution rules of the select mechanism, please read the article [channels in Go](#) (§21).

We can use a blank select block `select{}` to block the current goroutine for ever. This is the simplest use case of the select mechanism. In fact, some uses of `for {time.Sleep(time.Second)}` in some above examples can be replaced with `select{}`.

Generally, `select{}` is used to prevent the main goroutine from exiting, for if the main goroutine exits, the whole program will also exit.

An example:

```
1| package main
2|
3| import "runtime"
4|
5| func DoSomething() {
6|     for {
7|         // do something ...
8|
9|         runtime.Gosched() // avoid being greedy
10|    }
11| }
12|
13| func main() {
14|     go DoSomething()
15|     go DoSomething()
```

```
16|     select{}  
17| }
```

By the way, there are [some other ways](#) (§46) to make a goroutine stay in blocking state for ever. But the `select{}` way is the simplest one.

Try-Send and Try-Receive

A `select` block with one `default` branch and only one `case` branch is called a try-send or try-receive channel operation, depending on whether the channel operation following the `case` keyword is a channel send or receive operation.

- If the operation following the `case` keyword is a send operation, then the `select` block is called as try-send operation. If the send operation would block, then the `default` branch will get executed (fail to send), otherwise, the send succeeds and the only `case` branch will get executed.
- If the operation following the `case` keyword is a receive operation, then the `select` block is called as try-receive operation. If the receive operation would block, then the `default` branch will get executed (fail to receive), otherwise, the receive succeeds and the only `case` branch will get executed.

Try-send and try-receive operations never block.

The standard Go compiler makes special optimizations for try-send and try-receive select blocks, their execution efficiencies are much higher than multi-case select blocks.

The following is an example which shows how try-send and try-receive work.

```
1| package main  
2|  
3| import "fmt"  
4|  
5| func main() {  
6|     type Book struct{id int}  
7|     bookshelf := make(chan Book, 3)  
8|  
9|     for i := 0; i < cap(bookshelf) * 2; i++ {  
10|         select {  
11|             case bookshelf <- Book{id: i}:  
12|                 fmt.Println("succeeded to put book", i)  
13|             default:  
14|                 fmt.Println("failed to put book")  
15|         }  
16|     }  
17| }
```

```

15|     }
16|
17|
18|     for i := 0; i < cap(bookshelf) * 2; i++ {
19|         select {
20|             case book := <-bookshelf:
21|                 fmt.Println("succeeded to get book", book.id)
22|             default:
23|                 fmt.Println("failed to get book")
24|         }
25|     }
26|

```

The output of the above program:

```

succeed to put book 0
succeed to put book 1
succeed to put book 2
failed to put book
failed to put book
failed to put book
succeed to get book 0
succeed to get book 1
succeed to get book 2
failed to get book
failed to get book
failed to get book

```

The following sub-sections will show more try-send and try-receive use cases.

Check if a channel is closed without blocking the current goroutine

Assume it is guaranteed that no values were ever (and will be) sent to a channel, we can use the following code to (concurrently and safely) check whether or not the channel is already closed without blocking the current goroutine, where T the element type of the corresponding channel type.

```

1| func IsClosed(c chan T) bool {
2|     select {
3|         case <-c:
4|             return true
5|         default:

```

```

6|     }
7|     return false
8|

```

The way to check if a channel is closed is used popularly in Go concurrent programming to check whether or not a notification has arrived. The notification will be sent by closing the channel in another goroutine.

Peak/burst limiting

We can implement peak limiting by combining [use channels as counting semaphores](#) and try-send/try-receive. Peak-limit (or burst-limit) is often used to limit the number of concurrent requests without blocking any requests.

The following is a modified version of the last example in the [use channels as counting semaphores](#) section.

```

1| ...
2|     // Can serve most 10 customers at the same time
3|     bar24x7 := make(Bar, 10)
4|     for customerId := 0; ; customerId++ {
5|         time.Sleep(time.Second)
6|         customer := Consumer{customerId}
7|         select {
8|             case bar24x7 <- customer: // try to enter the bar
9|                 go bar24x7.ServeConsumer(customer)
10|             default:
11|                 log.Println("customer#", customerId, " goes elsewhere")
12|         }
13|     }
14| ...

```

Another way to implement the first-response-wins use case

As mentioned above, we can use the select mechanism (try-send) with a buffered channel which capacity is one (at least) to implement the first-response-wins use case. For example,

```

1| package main
2|
3| import (
4|     "fmt"
5|     "math/rand"

```

```

6|     "time"
7| )
8|
9| func source(c chan<- int32) {
10|     ra, rb := rand.Int31(), rand.Intn(3)+1
11|     // Sleep 1s, 2s or 3s.
12|     time.Sleep(time.Duration(rb) * time.Second)
13|     select {
14|         case c <- ra:
15|             default:
16|                 }
17|     }
18|
19| func main() {
20|     rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
21|
22|     // The capacity should be at least 1.
23|     c := make(chan int32, 1)
24|     for i := 0; i < 5; i++ {
25|         go source(c)
26|     }
27|     rnd := <-c // only the first response is used
28|     fmt.Println(rnd)
29| }
```

Please note, the capacity of the channel used in the above example must be at least one, so that the first send won't be missed if the receiver/request side has not gotten ready in time.

The third way to implement the first-response-wins use case

For a first-response-wins use case, if the number of sources is small, for example, two or three, we can use a `select` code block to receive the source responses at the same time. For example,

```

1| package main
2|
3| import (
4|     "fmt"
5|     "math/rand"
6|     "time"
7| )
8|
9| func source() <-chan int32 {
10|     // c must be a buffered channel.
11|     c := make(chan int32, 1)
```

```

12|     go func() {
13|         ra, rb := rand.Int31(), rand.Intn(3)+1
14|         time.Sleep(time.Duration(rb) * time.Second)
15|         c <- ra
16|     }()
17|     return c
18| }
19|
20| func main() {
21|     rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
22|
23|     var rnd int32
24|     // Blocking here until one source responses.
25|     select{
26|     case rnd = <-source():
27|     case rnd = <-source():
28|     case rnd = <-source():
29|     }
30|     fmt.Println(rnd)
31| }
```

Note: if the channel used in the above example is an unbuffered channel, then there will two goroutines hanging for ever after the `select` code block is executed. This is [a memory leak case](#) (§45).

The two ways introduced in the current and the last sub-sections can also be used to do N-to-1 notifications.

|Timeout

In some request-response scenarios, for all kinds of reasons, a request may need a long time to response, sometimes even will never response. For such cases, we should return an error message to the client side by using a timeout solution. Such a timeout solution can be implemented with the `select` mechanism.

The following code shows how to make a request with a timeout.

```

1| func requestWithTimeout(timeout time.Duration) (int, error) {
2|     c := make(chan int)
3|     // May need a long time to get the response.
4|     go doRequest(c)
5|
6|     select {
```

```

7|     case data := <-c:
8|         return data, nil
9|     case <-time.After(timeout):
10|        return 0, errors.New("timeout")
11|    }
12| }

```

Ticker

We can use the try-send mechanism to implement a ticker.

```

1| package main
2|
3| import "fmt"
4| import "time"
5|
6| func Tick(d time.Duration) <-chan struct{} {
7|     // The capacity of c is best set as one.
8|     c := make(chan struct{}, 1)
9|     go func() {
10|         for {
11|             time.Sleep(d)
12|             select {
13|                 case c <- struct{}{}:
14|                 default:
15|             }
16|         }
17|     }()
18|     return c
19| }
20|
21| func main() {
22|     t := time.Now()
23|     for range Tick(time.Second) {
24|         fmt.Println(time.Since(t))
25|     }
26| }

```

In fact, there is a `Tick` function in the `time` standard package provides the same functionality, with a much more efficient implementation. We should use that function instead to make code look clean and run efficiently.

Rate Limiting

One of above section has shown how to use try-send to do [peak limiting](#). We can also use try-send to do rate limiting (with the help of a ticker). In practice, rate-limit is often to avoid quota exceeding and resource exhaustion.

The following shows such an example borrowed from [the official Go wiki](#). In this example, the number of handled requests in any one-minute duration will not exceed 200.

```

1| package main
2|
3| import "fmt"
4| import "time"
5|
6| type Request interface{}
7| func handle(r Request) {fmt.Println(r.(int))}
8|
9| const RateLimitPeriod = time.Minute
10| const RateLimit = 200 // most 200 requests in one minute
11|
12| func handleRequests(requests <-chan Request) {
13|     quotas := make(chan time.Time, RateLimit)
14|
15|     go func() {
16|         tick := time.NewTicker(RateLimitPeriod / RateLimit)
17|         defer tick.Stop()
18|         for t := range tick.C {
19|             select {
20|                 case quotas <- t:
21|                 default:
22|             }
23|         }
24|     }()
25|
26|     for r := range requests {
27|         <-quotas
28|         go handle(r)
29|     }
30| }
31|
32| func main() {
33|     requests := make(chan Request)
34|     go handleRequests(requests)
35|     // time.Sleep(time.Minute)
36|     for i := 0; ; i++ {requests <- i}
37| }
```

In practice, we often use rate-limit and peak/burst-limit together.

Switches

From the article [channels in Go](#) (§21), we have learned that sending a value to or receiving a value from a nil channel are both blocking operations. By making use of this fact, we can change the involved channels in the case operations of a `select` code block to affect the branch selection in the `select` code block.

The following is another ping-pong example which is implemented by using the `select` mechanism. In this example, one of the two channel variables involved in the `select` block is `nil`. The case branch corresponding the `nil` channel will not get selected for sure. We can think such case branches are in off status. At the end of each loop step, the on/off statuses of the two case branches are switched.

```

1| package main
2|
3| import "fmt"
4| import "time"
5| import "os"
6|
7| type Ball uint8
8| func Play(playerName string, table chan Ball, serve bool) {
9|     var receive, send chan Ball
10|    if serve {
11|        receive, send = nil, table
12|    } else {
13|        receive, send = table, nil
14|    }
15|    var lastValue Ball = 1
16|    for {
17|        select {
18|            case send <- lastValue:
19|            case value := <- receive:
20|                fmt.Println(playerName, value)
21|                value += lastValue
22|                if value < lastValue { // overflow
23|                    os.Exit(0)
24|                }
25|                lastValue = value
26|        }
27|        // Switch on/off.
28|        receive, send = send, receive

```

```

29|     time.Sleep(time.Second)
30| }
31| }
32|
33| func main() {
34|     table := make(chan Ball)
35|     go Play("A:", table, false)
36|     Play("B:", table, true)
37| }
```

The following is another (non-concurrent) example which is much simpler and also demos the switch effect. This example will print 1212... when running. It has not much usefulness in practice. It is shown here just for learning purpose.

```

1| package main
2|
3| import "fmt"
4| import "time"
5|
6| func main() {
7|     for c := make(chan struct{}, 1); true; {
8|         select {
9|             case c <- struct{}{}:
10|                 fmt.Println("1")
11|             case <-c:
12|                 fmt.Println("2")
13|         }
14|         time.Sleep(time.Second)
15|     }
16| }
```

Control code execution possibility weights

We can duplicate a `case` branch in a `select` code block to increase the execution possibility weight of the corresponding code.

Example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
```

```

6|     foo, bar := make(chan struct{}), make(chan struct{})
7|     close(foo); close(bar) // for demo purpose
8|     x, y := 0.0, 0.0
9|     f := func(){x++}
10|    g := func(){y++}
11|    for i := 0; i < 1000000; i++ {
12|        select {
13|            case <-foo: f()
14|            case <-foo: f()
15|            case <-bar: g()
16|        }
17|    }
18|    fmt.Println(x/y) // about 2
19| }
```

The possibility of the `f` function being called is about the double of the `g` function being called.

Select from dynamic number of cases

Although the number of branches in a `select` block is fixed, we can use the functionalities provided in the `reflect` standard package to construct a `select` block at run time. The dynamically created `select` block can have an arbitrary number of case branches. But please note, the reflection way is less efficient than the fixed way.

The `reflect` standard package also provides `TrySend` and `TryRecv` functions to implement one-case-plus-default `select` blocks.

Data Flow Manipulations

This section will introduce some data flow manipulation use cases by using channels.

Generally, a data flow application consists of many modules. Different modules do different jobs. Each module may own one or several workers (goroutines), which concurrently do the same job specified for that module. Here is a list of some module job examples in practice:

- data generation/collecting/loading.
- data serving/saving.
- data calculation/analysis.
- data validation/filtering.
- data aggregation/division
- data composition/decomposition.

- data duplication/proliferation.

A worker in a module may receive data from several other modules as inputs and send data to serve other modules as outputs. In other words, a module can be both a data consumer and a data producer. A module which only sends data to some other modules but never receives data from other modules is called a producer-only module. A module which only receives data from some other modules but never sends data to other modules is called a consumer-only module.

Many modules together form a data flow system.

The following will show some data flow module worker implementations. These implementations are for explanation purpose, so they are very simple and they may be not efficient.

|Data generation/collecting/loading

There are all kinds of producer-only modules. A producer-only module worker may produce a data stream

- by loading a file, reading a database, or crawling the web.
- by collecting all kinds of metrics from a software system or all kinds of hardware.
- by generating random numbers.
- etc.

Here, we use a random number generator as an example. The generator function returns one result but takes no parameters.

```

1| import (
2|     "crypto/rand"
3|     "encoding/binary"
4| )
5|
6| func RandomGenerator() <-chan uint64 {
7|     c := make(chan uint64)
8|     go func() {
9|         rnds := make([]byte, 8)
10|        for {
11|             _, err := rand.Read(rnds)
12|             if err != nil {
13|                 close(c)
14|                 break
15|             }
16|             c <- binary.BigEndian.Uint64(rnds)
17|     }
```

```

18|     }()
19|     return c
20| }
```

In fact, the random number generator is a multi-return future/promise.

A data producer may close the output stream channel at any time to end data generating.

Data aggregation

A data aggregation module worker aggregates several data streams of the same data type into one stream. Assume the data type is `int64`, then the following function will aggregate an arbitrary number of data streams into one.

```

1| func Aggregator(inputs ...<-chan uint64) <-chan uint64 {
2|     out := make(chan uint64)
3|     for _, in := range inputs {
4|         go func(in <-chan uint64) {
5|             for {
6|                 out <- <-in // <=> out <- (<-in)
7|             }
8|         }(in)
9|     }
10|    return out
11| }
```

A better implementation should consider whether or not an input stream has been closed. (Also valid for the following other module worker implementations.)

```

1| import "sync"
2|
3| func Aggregator(inputs ...<-chan uint64) <-chan uint64 {
4|     output := make(chan uint64)
5|     var wg sync.WaitGroup
6|     for _, in := range inputs {
7|         wg.Add(1)
8|         go func(int <-chan uint64) {
9|             defer wg.Done()
10|             // If in is closed, then the
11|             // loop will ends eventually.
12|             for x := range in {
13|                 output <- x
14|             }
15|         }(in)
16|     }
17|     wg.Wait()
18|     close(output)
19| }
```

```

16|     }
17|     go func() {
18|         wg.Wait()
19|         close(output)
20|     }()
21|     return output
22| }
```

If the number of aggregated data streams is very small (two or three), we can use `select` block to aggregate these data streams.

```

1| // Assume the number of input stream is two.
2| ...
3|     output := make(chan uint64)
4|     go func() {
5|         inA, inB := inputs[0], inputs[1]
6|         for {
7|             select {
8|                 case v := <- inA: output <- v
9|                 case v := <- inB: output <- v
10|             }
11|         }
12|     }
13| ...
```

Data division

A data division module worker does the opposite of a data aggregation module worker. It is easy to implement a division worker, but in practice, division workers are not very useful and seldom used.

```

1| func Divisor(input <-chan uint64, outputs ...chan<- uint64) {
2|     for _, out := range outputs {
3|         go func(o chan<- uint64) {
4|             for {
5|                 o <- <-input // <=> o <- (<-input)
6|             }
7|         }(out)
8|     }
9| }
```

Data composition

A data composition worker merges several pieces of data from different input data streams into one piece of data.

The following is a composition worker example, in which two `uint64` values from one stream and one `uint64` value from another stream compose one new `uint64` value. Surely, these stream channel element types are different generally in practice.

```

1| func Composer(inA, inB <-chan uint64) <-chan uint64 {
2|     output := make(chan uint64)
3|     go func() {
4|         for {
5|             a1, b, a2 := <-inA, <-inB, <-inA
6|             output <- a1 ^ b & a2
7|         }
8|     }()
9|     return output
10| }
```

Data decomposition

Data decomposition is the inverse process of data composition. A decomposition worker function implementation takes one input data stream parameter and returns several data stream results. No examples will be shown for data decomposition here.

Data duplication/proliferation

Data duplication (proliferation) can be viewed as special data decompositions. One piece of data will be duplicated and each of the duplicated data will be sent to different output data streams.

An example:

```

1| func Duplicator(in <-chan uint64) (<-chan uint64, <-chan uint64) {
2|     outA, outB := make(chan uint64), make(chan uint64)
3|     go func() {
4|         for x := range in {
5|             outA <- x
6|             outB <- x
7|         }
8|     }()
9|     return outA, outB
10| }
```

Data calculation/analysis

The functionalities of data calculation and analysis modules vary and each is very specific. Generally, a worker function of such modules transforms each piece of input data into another piece of output data.

For simple demo purpose, here shows a worker example which inverts every bit of each transferred uint64 value.

```

1| func Calculator(in <-chan uint64, out chan uint64) (<-chan uint64) {
2|     if out == nil {
3|         out = make(chan uint64)
4|     }
5|     go func() {
6|         for x := range in {
7|             out <- ^x
8|         }
9|     }()
10|    return out
11| }
```

Data validation/filtering

A data validation or filtering module discards some transferred data in a stream. For example, the following worker function discards all non-prime numbers.

```

1| import "math/big"
2|
3| func Filter0(input <-chan uint64, output chan uint64) <-chan uint64 {
4|     if output == nil {
5|         output = make(chan uint64)
6|     }
7|     go func() {
8|         bigInt := big.NewInt(0)
9|         for x := range input {
10|             bigInt.SetInt64(x)
11|             if bigInt.ProbablyPrime(1) {
12|                 output <- x
13|             }
14|         }
15|     }()
16|     return output
17| }
```

```

18|
19| func Filter(input <-chan uint64) <-chan uint64 {
20|     return Filter0(input, nil)
21| }

```

Please note that each of the two implementations is used by one of the final two examples shown below.

|Data serving/saving

Generally, a data serving or saving module is the last or final output module in a data flow system. Here just provides a simple worker which prints each piece of data received from the input stream.

```

1| import "fmt"
2|
3| func Printer(input <-chan uint64) {
4|     for x := range input {
5|         fmt.Println(x)
6|     }
7| }

```

|Data flow system assembling

Now, let's use the above module worker functions to assemble several data flow systems.

Assembling a data flow system is just to create some workers of different modules, and specify the input streams for every worker.

Data flow system example 1 (a linear pipeline):

```

1| package main
2|
3| ... // the worker functions declared above.
4|
5| func main() {
6|     Printer(
7|         Filter(
8|             Calculator(
9|                 RandomGenerator(), nil,
10|             ),
11|         ),
12|     )
13| }

```

The above data flow system is depicted in the following diagram.

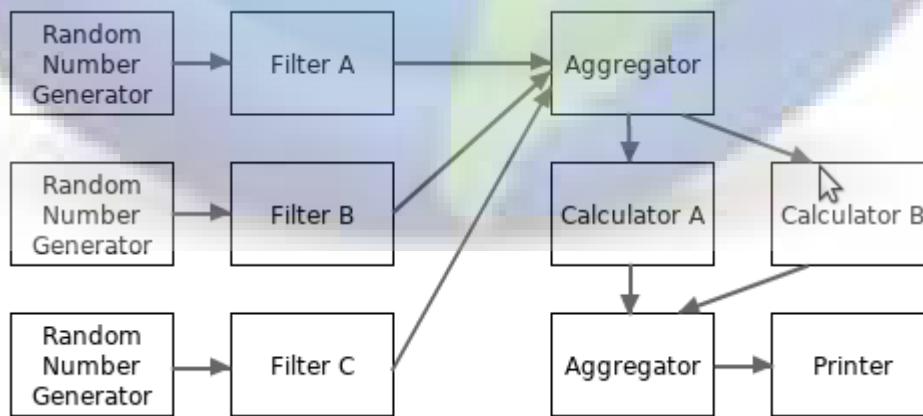


Data flow system example 2 (a directed acyclic graph pipeline):

```

1| package main
2|
3| ... // the worker functions declared above.
4|
5| func main() {
6|     filterA := Filter(RandomGenerator())
7|     filterB := Filter(RandomGenerator())
8|     filterC := Filter(RandomGenerator())
9|     filter := Aggregator(filterA, filterB, filterC)
10|    calculatorA := Calculator(filter, nil)
11|    calculatorB := Calculator(filter, nil)
12|    calculator := Aggregator(calculatorA, calculatorB)
13|    Printer(calculator)
14| }
  
```

The above data flow system is depicted in the following diagram.



More complex data flow topology may be arbitrary graphs. For example, a data flow system may have multiple final outputs. But data flow systems with cyclic-graph topology are seldom used in reality.

From the above two examples, we can find that it is very easy and intuitive to build data flow systems with channels.

From the last example, we can find that, with the help of aggregators, it is easy to implement fan-in and fan-out for the number of workers for a specified module.

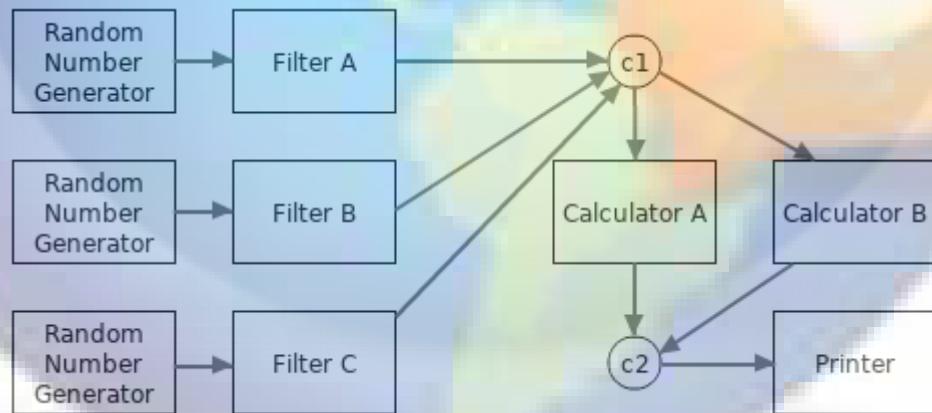
In fact, we can use a simple channel to replace the role of an aggregator. For example, the following example replaces the two aggregators with two channels.

```

1| package main
2|
3| ... // the worker functions declared above.
4|
5| func main() {
6|     c1 := make(chan uint64, 100)
7|     Filter0(RandomGenerator(), c1) // filterA
8|     Filter0(RandomGenerator(), c1) // filterB
9|     Filter0(RandomGenerator(), c1) // filterC
10|    c2 := make(chan uint64, 100)
11|    Calculator(c1, c2) // calculatorA
12|    Calculator(c1, c2) // calculatorB
13|    Printer(c2)
14| }

```

The modified data flow system is depicted in the following diagram.



The above explanations for data flow systems don't consider much on how to close data streams. Please read [this article](#) (§38) for explanations on how to gracefully close channels.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

How to Gracefully Close Channels

Several days ago, I wrote an article which explains [the channel rules in Go](#) (§21). That article got many votes on [reddit](#) and [HN](#), but there are also some criticisms on Go channel design details.

I collected some criticisms on the following designs and rules of Go channels:

1. no easy and universal ways to check whether or not a channel is closed without modifying the status of the channel.
2. closing a closed channel will panic, so it is dangerous to close a channel if the closers don't know whether or not the channel is closed.
3. sending values to a closed channel will panic, so it is dangerous to send values to a channel if the senders don't know whether or not the channel is closed.

The criticisms look reasonable (in fact not). Yes, there is really not a built-in function to check whether or not a channel has been closed.

There is indeed a simple method to check whether or not a channel is closed if you can make sure no values were (and will be) ever sent to the channel. The method has been shown in [the last article](#) (§37). Here, for better coherence, the method is listed in the following example again.

```

1| package main
2|
3| import "fmt"
4|
5| type T int
6|
7| func IsClosed(ch <-chan T) bool {
8|     select {
9|         case <-ch:
10|             return true
11|         default:
12|     }
13|
14|     return false
15| }
16|
17| func main() {
18|     c := make(chan T)
19|     fmt.Println(IsClosed(c)) // false

```

```

20|     close(c)
21|     fmt.Println(IsClosed(c)) // true
22|

```

As above mentioned, this is not a universal way to check whether a channel is closed.

In fact, even if there is a simple built-in `closed` function to check whether or not a channel has been closed, its usefulness would be very limited, just like the built-in `len` function for checking the current number of values stored in the value buffer of a channel. The reason is that the status of the checked channel may have changed just after a call to such functions returns, so that the returned value has already not been able to reflect the latest status of the just checked channel. Although it is okay to stop sending values to a channel `ch` if the call `closed(ch)` returns `true`, it is not safe to close the channel or continue sending values to the channel if the call `closed(ch)` returns `false`.

The Channel Closing Principle

One general principle of using Go channels is **don't close a channel from the receiver side and don't close a channel if the channel has multiple concurrent senders**. In other words, we should only close a channel in a sender goroutine if the sender is the only sender of the channel.

*(Below, we will call the above principle as **channel closing principle**.)*

Surely, this is not a universal principle to close channels. The universal principle is **don't close (or send values to) closed channels**. If we can guarantee that no goroutines will close and send values to a non-closed non-nil channel any more, then a goroutine can close the channel safely. However, making such guarantees by a receiver or by one of many senders of a channel usually needs much effort, and often makes code complicated. On the contrary, it is much easier to hold the **channel closing principle** mentioned above.

Solutions Which Close Channels Rudely

If you would close a channel from the receiver side or in one of the multiple senders of the channel anyway, then you can use [the recover mechanism](#) (§13) to prevent the possible panic from crashing your program. Here is an example (assume the channel element type is `T`).

```

1| func SafeClose(ch chan T) (justClosed bool) {
2|     defer func() {
3|         if recover() != nil {
4|             // The return result can be altered

```

```

5|         // in a defer function call.
6|         justClosed = false
7|     }
8| }()
9|
10| // assume ch != nil here.
11| close(ch) // panic if ch is closed
12| return true // <=> justClosed = true; return
13| }
```

This solution obviously breaks the **channel closing principle**.

The same idea can be used for sending values to a potential closed channel.

```

1| func SafeSend(ch chan T, value T) (closed bool) {
2|     defer func() {
3|         if recover() != nil {
4|             closed = true
5|         }
6|     }()
7|
8|     ch <- value // panic if ch is closed
9|     return false // <=> closed = false; return
10| }
```

Not only does the rude solution break the **channel closing principle**, but data races might also happen in the process.

Solutions Which Close Channels Politely

Many people prefer using `sync.Once` to close channels:

```

1| type MyChannel struct {
2|     C    chan T
3|     once sync.Once
4| }
5|
6| func NewMyChannel() *MyChannel {
7|     return &MyChannel{C: make(chan T)}
8| }
9|
10| func (mc *MyChannel) SafeClose() {
11|     mc.once.Do(func() {
12|         close(mc.C)
13| })
```

```
13|     })
14| }
```

Surely, we can also use `sync.Mutex` to avoid closing a channel multiple times:

```
1| type MyChannel struct {
2|     C      chan T
3|     closed bool
4|     mutex  sync.Mutex
5| }
6|
7| func NewMyChannel() *MyChannel {
8|     return &MyChannel{C: make(chan T)}
9| }
10|
11| func (mc *MyChannel) SafeClose() {
12|     mc.mutex.Lock()
13|     defer mc.mutex.Unlock()
14|     if !mc.closed {
15|         close(mc.C)
16|         mc.closed = true
17|     }
18| }
19|
20| func (mc *MyChannel) IsClosed() bool {
21|     mc.mutex.Lock()
22|     defer mc.mutex.Unlock()
23|     return mc.closed
24| }
```

These ways may be polite, but they may not avoid data races. Currently, Go specification doesn't guarantee that there are no data races happening when a channel close and a channel send operations are executed concurrently. If a `SafeClose` function is called concurrently with a channel send operation to the same channel, data races might happen (though such data races generally don't do any harm).

Solutions Which Close Channels Gracefully

One drawback of the above `SafeSend` function is that its calls can't be used as send operations that follow the `case` keyword in `select` blocks. The other drawback of the above `SafeSend` and `SafeClose` functions is that many people, including me, would think the above solutions by using

`panic/recover` and `sync` package are not graceful. Following, some pure-channel solutions without using `panic/recover` and `sync` package will be introduced, for all kinds of situations.

(In the following examples, `sync.WaitGroup` is used to make the examples complete. It may be not always essential to use it in real practice.)

1. M receivers, one sender, the sender says "no more sends" by closing the data channel

This is the simplest situation, just let the sender close the data channel when it doesn't want to send more.

```

1| package main
2|
3| import (
4|     "time"
5|     "math/rand"
6|     "sync"
7|     "log"
8| )
9|
10| func main() {
11|     rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
12|     log.SetFlags(0)
13|
14|     // ...
15|     const Max = 100000
16|     const NumReceivers = 100
17|
18|     wgReceivers := sync.WaitGroup{}
19|     wgReceivers.Add(NumReceivers)
20|
21|     // ...
22|     dataCh := make(chan int)
23|
24|     // the sender
25|     go func() {
26|         for {
27|             if value := rand.Intn(Max); value == 0 {
28|                 // The only sender can close the
29|                 // channel at any time safely.
30|                 close(dataCh)
31|             return

```

```

32|         } else {
33|             dataCh <- value
34|         }
35|     }
36| }()
37|
38| // receivers
39| for i := 0; i < NumReceivers; i++ {
40|     go func() {
41|         defer wgReceivers.Done()
42|
43|         // Receive values until dataCh is
44|         // closed and the value buffer queue
45|         // of dataCh becomes empty.
46|         for value := range dataCh {
47|             log.Println(value)
48|         }
49|     }()
50| }
51|
52| wgReceivers.Wait()
53| }
```

2. One receiver, N senders, the only receiver says "please stop sending more" by closing an additional signal channel

This is a situation a little more complicated than the above one. We can't let the receiver close the data channel to stop data transferring, since doing this will break the **channel closing principle**. But we can let the receiver close an additional signal channel to notify senders to stop sending values.

```

1| package main
2|
3| import (
4|     "time"
5|     "math/rand"
6|     "sync"
7|     "log"
8| )
9|
10| func main() {
11|     rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
12|     log.SetFlags(0)
13| }
```

```
14| // ...
15| const Max = 100000
16| const NumSenders = 1000
17|
18| wgReceivers := sync.WaitGroup{}
19| wgReceivers.Add(1)
20|
21| // ...
22| dataCh := make(chan int)
23| stopCh := make(chan struct{})
24|     // stopCh is an additional signal channel.
25|     // Its sender is the receiver of channel
26|     // dataCh, and its receivers are the
27|     // senders of channel datach.
28|
29| // senders
30| for i := 0; i < NumSenders; i++ {
31|     go func() {
32|         for {
33|             // The try-receive operation is to try
34|             // to exit the goroutine as early as
35|             // possible. For this specified example,
36|             // it is not essential.
37|             select {
38|                 case <- stopCh:
39|                     return
40|                 default:
41|             }
42|
43|             // Even if stopCh is closed, the first
44|             // branch in the second select may be
45|             // still not selected for some loops if
46|             // the send to dataCh is also unblocked.
47|             // But this is acceptable for this
48|             // example, so the first select block
49|             // above can be omitted.
50|             select {
51|                 case <- stopCh:
52|                     return
53|                 case dataCh <- rand.Intn(Max):
54|             }
55|         }
56|     }()
57| }
```

```

59|     // the receiver
60|     go func() {
61|         defer wgReceivers.Done()
62|
63|         for value := range dataCh {
64|             if value == Max-1 {
65|                 // The receiver of channel dataCh is
66|                 // also the sender of stopCh. It is
67|                 // safe to close the stop channel here.
68|                 close(stopCh)
69|                 return
70|             }
71|
72|             log.Println(value)
73|         }
74|     }()
75|
76|     // ...
77|     wgReceivers.Wait()
78|

```

As mentioned in the comments, for the additional signal channel, its sender is the receiver of the data channel. The additional signal channel is closed by its only sender, which holds the **channel closing principle**.

In this example, the channel `dataCh` is never closed. Yes, channels don't have to be closed. A channel will be eventually garbage collected if no goroutines reference it any more, whether it is closed or not. So the gracefulness of closing a channel here is not to close the channel.

3. M receivers, N senders, any one of them says "let's end the game" by notifying a moderator to close an additional signal channel

This is the most complicated situation. We can't let any of the receivers and the senders close the data channel. And we can't let any of the receivers close an additional signal channel to notify all senders and receivers to exit the game. Doing either will break the **channel closing principle**.

However, we can introduce a moderator role to close the additional signal channel. One trick in the following example is how to use a try-send operation to notify the moderator to close the additional signal channel.

```

1| package main
2|

```

```
3| import (
4|     "time"
5|     "math/rand"
6|     "sync"
7|     "log"
8|     "strconv"
9| )
10|
11| func main() {
12|     rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
13|     log.SetFlags(0)
14|
15|     // ...
16|     const Max = 1000000
17|     const NumReceivers = 10
18|     const NumSenders = 1000
19|
20|     wgReceivers := sync.WaitGroup{}
21|     wgReceivers.Add(NumReceivers)
22|
23|     // ...
24|     dataCh := make(chan int)
25|     stopCh := make(chan struct{})
26|         // stopCh is an additional signal channel.
27|         // Its sender is the moderator goroutine shown
28|         // below, and its receivers are all senders
29|         // and receivers of dataCh.
30|     toStop := make(chan string, 1)
31|         // The channel toStop is used to notify the
32|         // moderator to close the additional signal
33|         // channel (stopCh). Its senders are any senders
34|         // and receivers of dataCh, and its receiver is
35|         // the moderator goroutine shown below.
36|         // It must be a buffered channel.
37|
38|     var stoppedBy string
39|
40|     // moderator
41|     go func() {
42|         stoppedBy = <-toStop
43|         close(stopCh)
44|     }()
45|
46|     // senders
47|     for i := 0; i < NumSenders; i++ {
```

```
48|     go func(id string) {
49|         for {
50|             value := rand.Intn(Max)
51|             if value == 0 {
52|                 // Here, the try-send operation is
53|                 // to notify the moderator to close
54|                 // the additional signal channel.
55|                 select {
56|                     case toStop <- "sender#" + id:
57|                         default:
58|                     }
59|                     return
60|                 }
61|
62|             // The try-receive operation here is to
63|             // try to exit the sender goroutine as
64|             // early as possible. Try-receive and
65|             // try-send select blocks are specially
66|             // optimized by the standard Go
67|             // compiler, so they are very efficient.
68|             select {
69|                 case <- stopCh:
70|                     return
71|                 default:
72|             }
73|
74|             // Even if stopCh is closed, the first
75|             // branch in this select block might be
76|             // still not selected for some loops
77|             // (and for ever in theory) if the send
78|             // to dataCh is also non-blocking. If
79|             // this is unacceptable, then the above
80|             // try-receive operation is essential.
81|             select {
82|                 case <- stopCh:
83|                     return
84|                 case dataCh <- value:
85|             }
86|         }
87|         strconv.Itoa(i))
88|     }
89|
90|     // receivers
91|     for i := 0; i < NumReceivers; i++ {
92|         go func(id string) {
```

```
93|         defer wgReceivers.Done()
94|
95|         for {
96|             // Same as the sender goroutine, the
97|             // try-receive operation here is to
98|             // try to exit the receiver goroutine
99|             // as early as possible.
100|
101|             select {
102|                 case <- stopCh:
103|                     return
104|                 default:
105|             }
106|
107|             // Even if stopCh is closed, the first
108|             // branch in this select block might be
109|             // still not selected for some loops
110|             // (and forever in theory) if the receive
111|             // from dataCh is also non-blocking. If
112|             // this is not acceptable, then the above
113|             // try-receive operation is essential.
114|
115|             select {
116|                 case <- stopCh:
117|                     return
118|                 case value := <-dataCh:
119|                     if value == Max-1 {
```

// Here, the same trick is

```
119 | // used to notify the moderator
120 | // to close the additional
121 | // signal channel.
122 | select {
123 |     case toStop <- "receiver#" + id:
124 |         default:
125 |             }
126 |
127 |     return
128 |
129 |     log.Println(value)
130 |
131 | }
132 |
133 | }(strconv.Itoa(i))
134 |
135 | }
136 |
137 | wgReceivers.Wait()
138 |
| }
```

In this example, the **channel closing principle** is still held.

Please note that the buffer size (capacity) of channel `toStop` is one. This is to avoid the first notification is missed when it is sent before the moderator goroutine gets ready to receive

notification from `toStop`.

We can also set the capacity of the `toStop` channel as the sum number of senders and receivers, then we don't need a try-send `select` block to notify the moderator.

```

1| ...
2| toStop := make(chan string, NumReceivers + NumSenders)
3| ...
4|         value := rand.Intn(Max)
5|         if value == 0 {
6|             toStop <- "sender#" + id
7|             return
8|         }
9| ...
10|        if value == Max-1 {
11|            toStop <- "receiver#" + id
12|            return
13|        }
14| ...

```

4. A variant of the "M receivers, one sender" situation: the close request is made by a third-party goroutine

Sometimes, it is needed that the close signal must be made by a third-party goroutine. For such cases, we can use an extra signal channel to notify the sender to close the data channel. For example,

```

1| package main
2|
3| import (
4|     "time"
5|     "math/rand"
6|     "sync"
7|     "log"
8| )
9|
10| func main() {
11|     rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
12|     log.SetFlags(0)
13|
14|     // ...
15|     const Max = 1000000
16|     const NumReceivers = 100

```

```
17| const NumThirdParties = 15
18|
19| wgReceivers := sync.WaitGroup{}
20| wgReceivers.Add(NumReceivers)
21|
22| // ...
23| dataCh := make(chan int)
24| closing := make(chan struct{}) // signal channel
25| closed := make(chan struct{})
26|
27| // The stop function can be called
28| // multiple times safely.
29| stop := func() {
30|     select {
31|         case closing<-struct{}{}:
32|             <-closed
33|         case <-closed:
34|             }
35|     }
36|
37| // some third-party goroutines
38| for i := 0; i < NumThirdParties; i++ {
39|     go func() {
40|         r := 1 + rand.Intn(3)
41|         time.Sleep(time.Duration(r) * time.Second)
42|         stop()
43|     }()
44| }
45|
46| // the sender
47| go func() {
48|     defer func() {
49|         close(closed)
50|         close(dataCh)
51|     }()
52|
53|     for {
54|         select{
55|             case <-closing: return
56|             default:
57|                 }
58|
59|             select{
60|                 case <-closing: return
61|                 case dataCh <- rand.Intn(Max):
```

```

62| }
63|     }
64| }()
65|
66| // receivers
67| for i := 0; i < NumReceivers; i++ {
68|     go func() {
69|         defer wgReceivers.Done()
70|
71|         for value := range datach {
72|             log.Println(value)
73|         }
74|     }()
75| }
76|
77| wgReceivers.Wait()
78| }
```

The idea used in the `stop` function is learned from [a comment ↗](#) made by Roger Peppe.

5. A variant of the "N sender" situation: the data channel must be closed to tell receivers that data sending is over

In the solutions for the above N-sender situations, to hold the **channel closing principle**, we avoid closing the data channels. However, sometimes, it is required that the data channels must be closed in the end to let receivers know data sending is over. For such cases, we can translate a N-sender situation to a one-sender situation by using a middle channel. The middle channel has only one sender, so that we can close it instead of closing the original data channel.

```

1| package main
2|
3| import (
4|     "time"
5|     "math/rand"
6|     "sync"
7|     "log"
8|     "strconv"
9| )
10|
11| func main() {
12|     rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
13|     log.SetFlags(0)
14| }
```

```
15| // ...
16| const Max = 1000000
17| const NumReceivers = 10
18| const NumSenders = 1000
19| const NumThirdParties = 15
20|
21| wgReceivers := sync.WaitGroup{}
22| wgReceivers.Add(NumReceivers)
23|
24| // ...
25| dataCh := make(chan int)      // will be closed
26| middleCh := make(chan int)    // will never be closed
27| closing := make(chan string) // signal channel
28| closed := make(chan struct{})
29|
30| var stoppedBy string
31|
32| // The stop function can be called
33| // multiple times safely.
34| stop := func(by string) {
35|     select {
36|         case closing <- by:
37|             <-closed
38|         case <-closed:
39|             }
40|     }
41|
42| // the middle layer
43| go func() {
44|     exit := func(v int, needSend bool) {
45|         close(closed)
46|         if needSend {
47|             dataCh <- v
48|         }
49|         close(dataCh)
50|     }
51|
52|     for {
53|         select {
54|             case stoppedBy = <-closing:
55|                 exit(0, false)
56|                 return
57|             case v := <- middleCh:
58|                 select {
59|                     case stoppedBy = <-closing:
```

```
60|         exit(v, true)
61|     return
62|     case dataCh <- v:
63|         }
64|     }
65|   }
66| }()
67|
68| // some third-party goroutines
69| for i := 0; i < NumThirdParties; i++ {
70|     go func(id string) {
71|         r := 1 + rand.Intn(3)
72|         time.Sleep(time.Duration(r) * time.Second)
73|         stop("3rd-party#" + id)
74|     }(strconv.Itoa(i))
75| }
76|
77| // senders
78| for i := 0; i < NumSenders; i++ {
79|     go func(id string) {
80|         for {
81|             value := rand.Intn(Max)
82|             if value == 0 {
83|                 stop("sender#" + id)
84|                 return
85|             }
86|
87|             select {
88|             case <- closed:
89|                 return
90|             default:
91|             }
92|
93|             select {
94|             case <- closed:
95|                 return
96|             case middleCh <- value:
97|                 }
98|             }
99|     }(strconv.Itoa(i))
100|
101| }
```

```

102     |     // receivers
103
104     |     for range [NumReceivers]struct{}{} {
105         |         go func() {
106             |             defer wgReceivers.Done()
107
108             |             for value := range dataCh {
109                 |                 log.Println(value)
110             }
111         }()
112     }
113
114     |     // ...
115     |     wgReceivers.Wait()
116     |     log.Println("stopped by", stoppedBy)
117 }
```

| More situations?

There should be more situation variants, but the above shown ones are the most common and basic ones. By using channels (and other concurrent programming techniques) cleverly, I believe a solution holding the **channel closing principle** for each situation variant can always be found.

| Conclusion

There are no situations which will force you to break the **channel closing principle**. If you encounter such a situation, please rethink your design and rewrite your code.

Programming with Go channels is like making art.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Concurrency Synchronization Techniques Provided in the sync Standard Package

The [channel use cases](#) (§37) article introduces many use cases in which channels are used to do data synchronizations among goroutines. In fact, channels are not the only synchronization techniques provided in Go. There are some other synchronization techniques supported by Go. For some specified circumstances, using the synchronization techniques other than channel are more efficient and readable than using channels. Below will introduce the synchronization techniques provided in the sync standard package.

The sync standard package provides several types which can be used to do synchronizations for some specialized circumstances and guarantee some specialized memory orders. For the specialized circumstances, these techniques are more efficient, and look cleaner, than the channel ways.

(Please note, to avoid abnormal behaviors, it is best never to copy the values of the types in the sync standard package.)

The sync.WaitGroup Type

Each sync.WaitGroup value maintains a counter internally. The initial value of the counter is zero.

The *WaitGroup type has [three methods ↗](#): Add(delta int), Done() and Wait().

For an addressable WaitGroup value wg,

- we can call the wg.Add(delta) method to change the counter value maintained by wg.
- the method call wg.Done() is totally equivalent to the method call wg.Add(-1).
- if a call wg.Add(delta) (or wg.Done()) modifies the counter maintained by wg to negative, panic will happen.
- when a goroutine calls wg.Wait(),
 - if the counter maintained by wg is already zero, then the call wg.Wait() can be viewed as a no-op.
 - otherwise (the counter is positive), the goroutine will enter blocking state. It will enter running state again (a.k.a., the call wg.Wait() returns) when another goroutine modifies the counter to zero, generally by calling wg.Done().

Please note that `wg.Add(delta)`, `wg.Done()` and `wg.Wait()` are shorthands of `(&wg).Add(delta)`, `(&wg).Done()` and `(&wg).Wait()`, respectively.

Generally, a `WaitGroup` value is used for the scenario that one goroutine waits until all of several other goroutines finish their respective jobs. An example:

```

1| package main
2|
3| import (
4|     "log"
5|     "math/rand"
6|     "sync"
7|     "time"
8| )
9|
10| func main() {
11|     rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
12|
13|     const N = 5
14|     var values [N]int32
15|
16|     var wg sync.WaitGroup
17|     wg.Add(N)
18|     for i := 0; i < N; i++ {
19|         i := i
20|         go func() {
21|             values[i] = 50 + rand.Int31n(50)
22|             log.Println("Done:", i)
23|             wg.Done() // <=> wg.Add(-1)
24|         }()
25|     }
26|
27|     wg.Wait()
28|     // All elements are guaranteed to be
29|     // initialized now.
30|     log.Println("values:", values)
31| }
```

In the above example, the main goroutine waits until all other `N` goroutines have populated their respective element value in `values` array. Here is one possible output result:

```

Done: 4
Done: 1
Done: 3
Done: 0
```

```
Done: 2
values: [71 89 50 62 60]
```

We can split the only `Add` method call in the above example into multiple ones.

```
1| ...
2|     var wg sync.WaitGroup
3|     for i := 0; i < N; i++ {
4|         wg.Add(1) // will be invoked N times
5|         i := i
6|         go func() {
7|             values[i] = 50 + rand.Int31n(50)
8|             wg.Done()
9|         }()
10|    }
11| ...
```

The `Wait` method can be called in multiple goroutines. When the counter becomes zero, all of them will be notified, in a broadcast way.

```
1| func main() {
2|     rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
3|
4|     const N = 5
5|     var values [N]int32
6|
7|     var wgA, wgB sync.WaitGroup
8|     wgA.Add(N)
9|     wgB.Add(1)
10|
11|     for i := 0; i < N; i++ {
12|         i := i
13|         go func() {
14|             wgB.Wait() // wait a notification
15|             log.Printf("values[%v]=%v \n", i, values[i])
16|             wgA.Done()
17|         }()
18|     }
19|
20|     // The loop is guaranteed to finish before
21|     // any above wg.Wait calls returns.
22|     for i := 0; i < N; i++ {
23|         values[i] = 50 + rand.Int31n(50)
24|     }
25|     // Make a broadcast notification.
```

```

26|     wgB.Done()
27|     wgA.Wait()
28| }
```

A `WaitGroup` value can be reused after one call to its `Wait` method returns. But please note that each `Add` method call with a positive delta that occurs when the counter is zero must happen before any `Wait` call starts, otherwise, data races may happen.

The `sync.Once` Type

A `*sync.Once` value has a `Do(f func())` method, which takes a solo parameter with type `func()`.

For an addressable `Once` value `o`, the method call `o.Do()`, which is a shorthand of `(&o).Do()`, can be concurrently executed multiple times, in multiple goroutines. The arguments of these `o.Do()` calls should (but are not required to) be the same function value.

Among these `o.Do` method calls, only exact one argument function will be invoked. The invoked argument function is guaranteed to exit before any `o.Do` method call returns. In other words, the code in the invoked argument function is guaranteed to be executed before any `o.Do` method call returns.

Generally, a `Once` value is used to ensure that a piece of code will be executed exactly once in concurrent programming.

An example:

```

1| package main
2|
3| import (
4|     "log"
5|     "sync"
6| )
7|
8| func main() {
9|     log.SetFlags(0)
10|
11|     x := 0
12|     doSomething := func() {
13|         x++
14|         log.Println("Hello")
15|     }
}
```

```

16|
17|     var wg sync.WaitGroup
18|     var once sync.Once
19|     for i := 0; i < 5; i++ {
20|         wg.Add(1)
21|         go func() {
22|             defer wg.Done()
23|             once.Do(doSomething)
24|             log.Println("world!")
25|         }()
26|     }
27|
28|     wg.Wait()
29|     log.Println("x =", x) // x = 1
30|

```

In the above example, `Hello` will be printed once, but `world!` will be printed five times. And `Hello` is guaranteed to be printed before all five `world!`.

The sync.Mutex and sync.RWMutex Types

Both of the `*sync.Mutex` and `*sync.RWMutex` types implement [the sync.Locker interface](#). So they both have two methods, `Lock` and `Unlock`, to prevent multiple data users from using a piece of data concurrently.

Besides the `Lock` and `Unlock` methods, the `*RWMutex` type has two other methods, `RLock` and `RUnlock`, to avoid some data users (either writers or readers) and one data writer using a piece of data at the same time but allow some data readers to access the piece of data at the same time.

*(Note, here the terminologies **data reader** and **data writer** should not be interpreted from literal. They are just used for explanation purpose. A data reader might modify data and a data writer might only read data.)*

A `Mutex` value is often called a mutual exclusion lock. A zero `Mutex` value is an unlocked mutex. A `Mutex` value can only be locked when it is in unlocked status. In other words, once an addressable `Mutex` value `m` is locked successfully (a.k.a., a `m.Lock()` method call returns), a new attempt by a goroutine to lock the `Mutex` value will make the goroutine enter blocking state, until the `Mutex` value is unlocked (through a later `m.Unlock()` call).

Please note that `m.Lock()` and `m.Unlock()` are shorthands of `(&m).Lock()` and `(&m).Unlock()`, respectively.

An example of using `sync.Mutex`:

```
1| package main
2|
3| import (
4|     "fmt"
5|     "runtime"
6|     "sync"
7| )
8|
9| type Counter struct {
10|     m sync.Mutex
11|     n uint64
12| }
13|
14| func (c *Counter) Value() uint64 {
15|     c.m.Lock()
16|     defer c.m.Unlock()
17|     return c.n
18| }
19|
20| func (c *Counter) Increase(delta uint64) {
21|     c.m.Lock()
22|     c.n += delta
23|     c.m.Unlock()
24| }
25|
26| func main() {
27|     var c Counter
28|     for i := 0; i < 100; i++ {
29|         go func() {
30|             for k := 0; k < 100; k++ {
31|                 c.Increase(1)
32|             }
33|         }()
34|     }
35|
36|     // The loop is just for demo purpose.
37|     for c.Value() < 10000 {
38|         runtime.Gosched()
39|     }
40|     fmt.Println(c.Value()) // 10000
41| }
```

In the above example, a `Counter` value uses a `Mutex` field to guarantee that the `n` field of the `Counter` value will be never used by multiple goroutines at the same time.

A `RWMutex` value is often called a reader+writer mutual exclusion lock. It has two locks, the write lock and the read lock. The locks of a zero `RWMutex` value are both unlocked. For an addressable `RWMutex` value `rwm`, data writers can lock the write lock of `rwm` through `rwm.Lock()` method calls, and data readers can lock the read lock of `rwm` through `rwm.RLock()` method calls. Method calls `rwm.Unlock()` and `rwm.RUnlock()` are used to unlock the write and read locks of `rwm`. The read lock of `rwm` maintains a lock count, which increases by one when `rwm.Lock()` is called successfully and decreases by one when `rwm.Unlock()` is called successfully. A zero lock count means the read lock is in unlocked status and a non-zero one (must be larger than one) means the read lock is locked.

Please note that `rwm.Lock()`, `rwm.Unlock()`, `rwm.RLock()` and `rwm.RUnlock()` are shorthands of `(&rwm).Lock()`, `(&rwm).Unlock()`, `(&rwm).RLock()` and `(&rwm).RUnlock()`, respectively.

For an addressable `RWMutex` value `rwm`, the following rules exist.

- The write lock of `rwm` may be locked only if neither of the read lock and write lock of `rwm` is in locked status. In other words, the write lock of `rwm` may only be held by at most one writer at any given time, and the read lock and write lock of `rwm` may not be held at the same time.
- When the write lock of `rwm` is in locked status, any newer attempts to lock the write lock or the read lock of `rwm` will be blocked until the initial write lock is unlocked.
- When the read lock of `rwm` is in locked status, any newer attempts to lock the write lock will be blocked until the read lock is unlocked. However, newer attempts to lock the read lock will succeed as long as the attempts are performed before any blocked attempts to lock the write lock (see the next rule for details). In other words, the read lock may be held by more than one readers at a time. The read lock will return to unlocked status when its lock count returns to zero.
- Assume the read lock of the value `rwm` is in locked status now, to avoid endless read locking, any newer attempts to lock the read lock after the a being blocked attempt to lock the write lock will be blocked.
- Assume the write lock of the value `rwm` is in locked status now, for the official standard Go compiler, to avoid endless write locking, the attempts to lock the read lock before releasing the write lock will succeed for sure once the write lock is unlocked, even if some of the attempts are made after some still being blocked attempts to lock the write lock.

The last two rules are to ensure both readers and writers have chances to acquire locks.

Please note, locks are not bound to goroutines. A lock may be locked in one goroutine and unlocked in another one later. In other words, a lock doesn't know which goroutine successfully locked or unlocked it.

The type of the `m` field of the `Counter` type in the last example can be changed to `sync.RWMutex`, as the following code shows, to get a better performance when the `Value` method is called very frequently but the `Increase` method is called not frequently.

```

1| ...
2| type Counter struct {
3|     //m sync.Mutex
4|     m sync.RWMutex
5|     n uint64
6| }
7|
8| func (c *Counter) Value() uint64 {
9|     //c.m.Lock()
10|    //defer c.m.Unlock()
11|    c.m.RLock()
12|    defer c.m.RUnlock()
13|    return c.n
14| }
15| ...

```

Another use scenario of `sync.RWMutex` values is to slice a write job into several small ones. Please read the next section for an example.

By the last two rules mentioned above, the following program is very possible to output `abdc`.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "time"
6|     "sync"
7| )
8|
9| func main() {
10|     var m sync.RWMutex
11|     go func() {
12|         m.RLock()
13|         fmt.Println("a")
14|         time.Sleep(time.Second)
15|         m.RUnlock()

```

```

16| }()
17| go func() {
18|     time.Sleep(time.Second * 1 / 4)
19|     m.Lock()
20|     fmt.Println("b")
21|     time.Sleep(time.Second)
22|     m.Unlock()
23| }()
24| go func() {
25|     time.Sleep(time.Second * 2 / 4)
26|     m.Lock()
27|     fmt.Println("c")
28|     m.Unlock()
29| }()
30| go func () {
31|     time.Sleep(time.Second * 3 / 4)
32|     m.RLock()
33|     fmt.Println("d")
34|     m.RUnlock()
35| }()
36|     time.Sleep(time.Second * 3)
37|     fmt.Println()
38| }

```

Please note, the above example is only for explanation purpose. It uses `time.Sleep` calls to do concurrency synchronizations, which is [a bad practice for production code](#) (§42).

`sync.Mutex` and `sync.RWMutex` values can also be used to make notifications, though there are many other better ways to do the same job. Here is an example which makes a notification by using a `sync.Mutex` value.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "sync"
6|     "time"
7| )
8|
9| func main() {
10|     var m sync.Mutex
11|     m.Lock()
12|     go func() {
13|         time.Sleep(time.Second)
14|         fmt.Println("Hi")

```

```

15|     m.Unlock() // make a notification
16| }()
17|     m.Lock() // wait to be notified
18|     fmt.Println("Bye")
19| }
```

In the above example, the text `Hi` is guaranteed to be printed before the text `Bye`. About the memory order guarantees made by `sync.Mutex` and `sync.RWMutex` values, please read [memory order guarantees in Go](#) (§41).

The `sync.Cond` Type

The `sync.Cond` type provides an efficient way to do notifications among goroutines.

Each `sync.Cond` value holds a `sync.Locker` field with name `L`. The field value is often a value of type `*sync.Mutex` or `*sync.RWMutex`.

The `*sync.Cond` type has [three methods](#), `Wait()`, `Signal()` and `Broadcast()`.

Each `sync.Cond` value also maintains a FIFO (first in first out) waiting goroutine queue. For an addressable `sync.Cond` value `c`,

- `c.Wait()` must be called when `c.L` is locked, otherwise, a `c.Wait()` will cause panic. A `c.Wait()` call will
 1. first push the current caller goroutine into the waiting goroutine queue maintained by `c`,
 2. then call `c.L.Unlock()` to unlock/unhold the lock `c.L`.
 3. then make the current caller goroutine enter blocking state.

(The caller goroutine will be unblocked by another goroutine through calling `c.Signal()` or `c.Broadcast()` later.)

Once the caller goroutine is unblocked and enters running state again, `c.L.Lock()` will be called (in the resumed `c.Wait()` call) to try to lock and hold the lock `c.L` again. The `c.Wait()` call will exit after the `c.L.Lock()` call returns.

- a `c.Signal()` call will unblock the first goroutine in (and remove it from) the waiting goroutine queue maintained by `c`, if the queue is not empty.
- a `c.Broadcast()` call will unblock all the goroutines in (and remove them from) the waiting goroutine queue maintained by `c`, if the queue is not empty.

Please note that `c.Wait()`, `c.Signal()` and `c.Broadcast()` are shorthands of `(&c).Wait()`, `(&c).Signal()` and `(&c).Broadcast()`, respectively.

`c.Signal()` and `c.Broadcast()` are often used to notify the status of a condition is changed. Generally, `c.Wait()` should be called in a loop of checking whether or not a condition has got satisfied.

In an idiomatic `sync.Cond` use case, generally, one goroutine waits for changes of a certain condition, and some other goroutines change the condition and send notifications. Here is an example:

```

1| package main
2|
3| import (
4|     "fmt"
5|     "math/rand"
6|     "sync"
7|     "time"
8| )
9|
10| func main() {
11|     rand.Seed(time.Now().UnixNano()) // needed before Go 1.20
12|
13|     const N = 10
14|     var values [N]string
15|
16|     cond := sync.NewCond(&sync.Mutex{})
17|
18|     for i := 0; i < N; i++ {
19|         d := time.Second * time.Duration(rand.Intn(10)) / 10
20|         go func(i int) {
21|             time.Sleep(d) // simulate a workload
22|
23|             // Changes must be made when
24|             // cond.L is locked.
25|             cond.L.Lock()
26|             values[i] = string('a' + i)
27|
28|             // Notify when cond.L lock is locked.
29|             cond.Broadcast()
30|             cond.L.Unlock()
31|
32|             // "cond.Broadcast()" can also be put
33|             // here, when cond.L lock is unlocked.

```

```

34|         //cond.Broadcast()
35|     }(i)
36|
37|
38|     // This function must be called when
39|     // cond.L is locked.
40|     checkCondition := func() bool {
41|         fmt.Println(values)
42|         for i := 0; i < N; i++ {
43|             if values[i] == "" {
44|                 return false
45|             }
46|         }
47|         return true
48|     }
49|
50|     cond.L.Lock()
51|     defer cond.L.Unlock()
52|     for !checkCondition() {
53|         // Must be called when cond.L is locked.
54|         cond.Wait()
55|     }
56| }
```

One possible output:

```

[      ]
[   f   ]
[ c   f   ]
[ c   f   h ]
[ b c   f   h ]
[ a b c   f   h   j]
[ a b c   f g h i j]
[ a b c   e f g h i j]
[ a b c d e f g h i j]
```

For there is only one goroutine (the main goroutine) waiting to be unblocked in this example, the `cond.Broadcast()` call can be replaced with `cond.Signal()`. As the comments suggest, `cond.Broadcast()` and `cond.Signal()` are not required to be called when `cond.L` is locked.

To avoid data races, each of the ten parts of the user defined condition should only be modified when `cond.L` is locked. The `checkCondition` function and the `cond.Wait` method should be also called when `cond.L` is locked.

In fact, for the above specified example, the `cond.L` field can also be a `*sync.RWMutex` value, and each of the ten parts of the user defined condition can be modified when the read lock of `cond.L` is held, just as the following code shows:

```

1| ...
2|     cond := sync.NewCond(&sync.RWMutex{})
3|     cond.L.Lock()
4|
5|     for i := 0; i < N; i++ {
6|         d := time.Second * time.Duration(rand.Intn(10)) / 10
7|         go func(i int) {
8|             time.Sleep(d)
9|             cond.L.(*sync.RWMutex).RLock()
10|            values[i] = string('a' + i)
11|            cond.L.(*sync.RWMutex).RUnlock()
12|            cond.Signal()
13|        }(i)
14|    }
15| ...

```

In the above example, the `sync.RWMutex` value is used unusually. Its read lock is held by some goroutines which modify array elements, and its write lock is used by the main goroutine to read array elements.

The user defined condition monitored by a `Cond` value can be a void. For such cases, the `Cond` value is used for notifications purely. For example, the following program will print abc or bac.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "sync"
6| )
7|
8| func main() {
9|     wg := sync.WaitGroup{}
10|    wg.Add(1)
11|    cond := sync.NewCond(&sync.Mutex{})
12|    cond.L.Lock()
13|    go func() {
14|        cond.L.Lock()
15|        go func() {
16|            cond.L.Lock()
17|            cond.Broadcast()

```

```
18|     cond.L.Unlock()
19| }()
20|     cond.Wait()
21|     fmt.Println("a")
22|     cond.L.Unlock()
23|     wg.Done()
24| }()
25|     cond.Wait()
26|     fmt.Println("b")
27|     cond.L.Unlock()
28|     wg.Wait()
29|     fmt.Println("c")
30| }
```

If it needs, multiple `sync.Cond` values can share the same `sync.Locker`. However, such cases are rare in practice.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Atomic Operations Provided in The sync/atomic Standard Package

Atomic operations are more primitive than other synchronization techniques. They are lockless and generally implemented directly at hardware level. In fact, they are often used in implementing other synchronization techniques.

Please note, many examples below are not concurrent programs. They are just for demonstration and explanation purposes, to show how to use the atomic functions provided in the sync/atomic standard package.

Overview of Atomic Operations Provided Before Go 1.19-

The sync/atomic standard package provides the following five atomic functions for an integer type T, where T must be any of int32, int64, uint32, uint64 and uintptr.

```
1| func AddT(addr *T, delta T)(new T)
2| func LoadT(addr *T) (val T)
3| func StoreT(addr *T, val T)
4| func SwapT(addr *T, new T) (old T)
5| func CompareAndSwapT(addr *T, old, new T) (swapped bool)
```

For example, the following five functions are provided for type int32.

```
1| func AddInt32(addr *int32, delta int32)(new int32)
2| func LoadInt32(addr *int32) (val int32)
3| func StoreInt32(addr *int32, val int32)
4| func SwapInt32(addr *int32, new int32) (old int32)
5| func CompareAndSwapInt32(addr *int32,
6|                           old, new int32) (swapped bool)
```

The following four atomic functions are provided for (safe) pointer types. When these functions were introduced into the standard library, Go didn't support custom generics, so these functions are implemented through the [unsafe pointer type](#) (§25) unsafe.Pointer (the Go counterpart of C void*).

```

1| func LoadPointer(addr *unsafe.Pointer) (val unsafe.Pointer)
2| func StorePointer(addr *unsafe.Pointer, val unsafe.Pointer)
3| func SwapPointer(addr *unsafe.Pointer, new unsafe.Pointer,
4|                   ) (old unsafe.Pointer)
5| func CompareAndSwapPointer(addr *unsafe.Pointer,
6|                             old, new unsafe.Pointer) (swapped bool)

```

There is not an `AddPointer` function for pointers, as Go (safe) pointers don't support arithmetic operations.

The `sync/atomic` standard package also provides a type `Value`, which corresponding pointer type `*Value` has four methods (listed below, the later two were introduced by Go 1.17). We may use these methods to do atomic operations for values of any type.

```

1| func (*Value) Load() (x interface{})
2| func (*Value) Store(x interface{})
3| func (*Value) Swap(new interface{}) (old interface{})
4| func (*Value) CompareAndSwap(old, new interface{}) (swapped bool)

```

Overview of New Atomic Operations Provided Since Go 1.19

Go 1.19 introduced several types, each of which owns a set of atomic operation methods, to achieve the same effects made by the package-level functions listed in the last section.

Among these types, `Int32`, `Int64`, `Uint32`, `Uint64` and `Uintptr` are for integer atomic operations. The methods of the `atomic.Int32` type are listed below. The methods of the other four types present in the similar way.

```

1| func (*Int32) Add(delta int32) (new int32)
2| func (*Int32) Load() int32
3| func (*Int32) Store(val int32)
4| func (*Int32) Swap(new int32) (old int32)
5| func (*Int32) CompareAndSwap(old, new int32) (swapped bool)

```

Since Go 1.18, Go has already supported custom generics. And some standard packages started to adopt custom generics since Go 1.19. The `sync/atomic` package is one of these packages. The `Pointer[T any]` type introduced in this package by Go 1.19 is a generic type. Its methods are listed below.

```

1| (*Pointer[T]) Load() *T
2| (*Pointer[T]) Store(val *T)
3| (*Pointer[T]) Swap(new *T) (old *T)
4| (*Pointer[T]) CompareAndSwap(old, new *T) (swapped bool)

```

Go 1.19 also introduced a `Bool` type to do boolean atomic operations.

Atomic Operations for Integers

The remaining of this article shows some examples on how to use the atomic operations provided in Go.

The following example shows how to do the `Add` atomic operation on an `int32` value by using the `AddInt32` function. In this example, 1000 new concurrent goroutines are created by the main goroutine. Each of the new created goroutine increases the integer `n` by one. Atomic operations guarantee that there are no data races among these goroutines. In the end, `1000` is guaranteed to be printed.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "sync"
6|     "sync/atomic"
7| )
8|
9| func main() {
10|     var n int32
11|     var wg sync.WaitGroup
12|     for i := 0; i < 1000; i++ {
13|         wg.Add(1)
14|         go func() {
15|             atomic.AddInt32(&n, 1)
16|             wg.Done()
17|         }()
18|     }
19|     wg.Wait()
20|
21|     fmt.Println	atomic.LoadInt32(&n)) // 1000
22| }

```

If the statement `atomic.AddInt32(&n, 1)` is replaced with `n++`, then the output might be not 1000.

The following code re-implements the above program by using the `atomic.Int32` type and its methods (since Go 1.19). This code looks a bit tidier.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "sync"
6|     "sync/atomic"
7| )
8|
9| func main() {
10|     var n atomic.Int32
11|     var wg sync.WaitGroup
12|     for i := 0; i < 1000; i++ {
13|         wg.Add(1)
14|         go func() {
15|             n.Add(1)
16|             wg.Done()
17|         }()
18|     }
19|     wg.Wait()
20|
21|     fmt.Println(n.Load()) // 1000
22| }
```

The `StoreT` and `LoadT` atomic functions/methods are often used to implement the setter and getter methods of (the corresponding pointer type of) a type if the values of the type need to be used concurrently. For example, the function version:

```

1| type Page struct {
2|     views uint32
3| }
4|
5| func (page *Page) SetViews(n uint32) {
6|     atomic.StoreUint32(&page.views, n)
7| }
8|
9| func (page *Page) Views() uint32 {
10|     return atomic.LoadUint32(&page.views)
11| }
```

And the type+methods version (since Go 1.19):

```

1| type Page struct {
2|     views atomic.Uint32
3|
4|
5| func (page *Page) SetViews(n uint32) {
6|     page.views.Store(n)
7| }
8|
9| func (page *Page) Views() uint32 {
10|    return page.views.Load()
11| }
```

For a signed integer type T (`int32` or `int64`), the second argument for a call to the `AddT` function can be a negative value, to do an atomic decrease operation. But how to do atomic decrease operations for values of an unsigned type T , such as `uint32`, `uint64` and `uintptr`? There are two circumstances for the second unsigned arguments.

1. For an unsigned variable v of type T , $-v$ is legal in Go. So we can just pass $-v$ as the second argument of an `AddT` call.
2. For a positive constant integer c , $-c$ is illegal to be used as the second argument of an `AddT` call (where T denotes an unsigned integer type). We can used $^T(c-1)$ as the second argument instead.

This $^T(v-1)$ trick also works for an unsigned variable v , but $^T(v-1)$ is less efficient than $T(-v)$.

In the trick $^T(c-1)$, if c is a typed value and its type is exactly T , then the form can shortened as $^(c-1)$.

Example:

```

1| package main
2|
3| import (
4|     "fmt"
5|     "sync/atomic"
6| )
7|
8| func main() {
9|     var (
10|         n uint64 = 97
11|         m uint64 = 1
```

```

12|     k int = 2
13| )
14| const (
15|     a      = 3
16|     b uint64 = 4
17|     c uint32 = 5
18|     d int    = 6
19| )
20|
21| show := fmt.Println
22| atomic.AddUint64(&n, -m)
23| show(n) // 96 (97 - 1)
24| atomic.AddUint64(&n, -uint64(k))
25| show(n) // 94 (96 - 2)
26| atomic.AddUint64(&n, ^uint64(a - 1))
27| show(n) // 91 (94 - 3)
28| atomic.AddUint64(&n, ^(b - 1))
29| show(n) // 87 (91 - 4)
30| atomic.AddUint64(&n, ^uint64(c - 1))
31| show(n) // 82 (87 - 5)
32| atomic.AddUint64(&n, ^uint64(d - 1))
33| show(n) // 76 (82 - 6)
34| x := b; atomic.AddUint64(&n, -x)
35| show(n) // 72 (76 - 4)
36| atomic.AddUint64(&n, ^(m - 1))
37| show(n) // 71 (72 - 1)
38| atomic.AddUint64(&n, ^uint64(k - 1))
39| show(n) // 69 (71 - 2)
40| }

```

A `SwapT` function call is like a `StoreT` function call, but returns the old value.

A `CompareAndSwapT` function call only applies the store operation when the current value matches the passed old value. The `bool` return result of the `CompareAndSwapT` function call indicates whether or not the store operation is applied.

Example:

```

1| package main
2|
3| import (
4|     "fmt"
5|     "sync/atomic"
6| )
7|

```

```

8| func main() {
9|     var n int64 = 123
10|    var old = atomic.SwapInt64(&n, 789)
11|    fmt.Println(n, old) // 789 123
12|    swapped := atomic.CompareAndSwapInt64(&n, 123, 456)
13|    fmt.Println(swapped) // false
14|    fmt.Println(n)      // 789
15|    swapped = atomic.CompareAndSwapInt64(&n, 789, 456)
16|    fmt.Println(swapped) // true
17|    fmt.Println(n)      // 456
18| }
```

The following is the corresponding type+methods version (since Go 1.19):

```

1| package main
2|
3| import (
4|     "fmt"
5|     "sync/atomic"
6| )
7|
8| func main() {
9|     var n atomic.Int64
10|    n.Store(123)
11|    var old = n.Swap(789)
12|    fmt.Println(n.Load(), old) // 789 123
13|    swapped := n.CompareAndSwap(123, 456)
14|    fmt.Println(swapped) // false
15|    fmt.Println(n.Load()) // 789
16|    swapped = n.CompareAndSwap(789, 456)
17|    fmt.Println(swapped) // true
18|    fmt.Println(n.Load()) // 456
19| }
```

Please note, up to now (Go 1.21), atomic operations for 64-bit words, a.k.a. int64 and uint64 values, require the 64-bit words must be 8-byte aligned in memory. For Go 1.19 introduced atomic method operations, this requirement is always satisfied, either on 32-bit or 64-bit architectures, but this is not true for atomic function operations on 32-bit architectures. Please read [memory layout](#) (§44) for details.

Atomic Operations for Pointers

Above has mentioned that there are four functions provided in the `sync/atomic` standard package to do atomic pointer operations, with the help of unsafe pointers.

From the article [type-unsafe pointers](#) (§25), we learn that, in Go, values of any pointer type can be explicitly converted to `unsafe.Pointer`, and vice versa. So values of `*unsafe.Pointer` type can also be explicitly converted to `unsafe.Pointer`, and vice versa.

The following example is not a concurrent program. It just shows how to do atomic pointer operations. In this example, `T` can be an arbitrary type.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "sync/atomic"
6|     "unsafe"
7| )
8|
9| type T struct {x int}
10|
11| func main() {
12|     var pT *T
13|     var unsafePPT = (*unsafe.Pointer)(unsafe.Pointer(&pT))
14|     var ta, tb = T{1}, T{2}
15|     // store
16|     atomic.StorePointer(
17|         unsafePPT, unsafe.Pointer(&ta))
18|     fmt.Println(pT) // &{1}
19|     // load
20|     pa1 := (*T)(atomic.LoadPointer(unsafePPT))
21|     fmt.Println(pa1 == &ta) // true
22|     // swap
23|     pa2 := atomic.SwapPointer(
24|         unsafePPT, unsafe.Pointer(&tb))
25|     fmt.Println((*T)(pa2) == &ta) // true
26|     fmt.Println(pT) // &{2}
27|     // compare and swap
28|     b := atomic.CompareAndSwapPointer(
29|         unsafePPT, pa2, unsafe.Pointer(&tb))
30|     fmt.Println(b) // false
31|     b = atomic.CompareAndSwapPointer(
32|         unsafePPT, unsafe.Pointer(&tb), pa2)
33|     fmt.Println(b) // true
34| }
```

Yes, it is quite verbose to use the pointer atomic functions. In fact, not only are the uses verbose, they are also not protected by [Go 1 compatibility guidelines](#), for these uses require to import the unsafe standard package.

On the contrary, the code will be much simpler and cleaner if we use the Go 1.19 introduced generic `Pointer` type and its methods to do atomic pointer operations, as the following code shows.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "sync/atomic"
6| )
7|
8| type T struct {x int}
9|
10| func main() {
11|     var pT atomic.Pointer[T]
12|     var ta, tb = T{1}, T{2}
13|     // store
14|     pT.Store(&ta)
15|     fmt.Println(pT.Load()) // &{1}
16|     // load
17|     pa1 := pT.Load()
18|     fmt.Println(pa1 == &ta) // true
19|     // swap
20|     pa2 := pT.Swap(&tb)
21|     fmt.Println(pa2 == &ta) // true
22|     fmt.Println(pT.Load()) // &{2}
23|     // compare and swap
24|     b := pT.CompareAndSwap(&ta, &tb)
25|     fmt.Println(b) // false
26|     b = pT.CompareAndSwap(&tb, &ta)
27|     fmt.Println(b) // true
28| }
```

More importantly, the implementation using the generic `Pointer` type is protected by Go 1 compatibility guidelines.

Atomic Operations for Values of Arbitrary Types

The `Value` type provided in the `sync/atomic` standard package can be used to atomically load and store values of any type.

Type `*Value` has several methods: `Load`, `Store`, `Swap` and `CompareAndSwap` (The latter two are introduced in Go 1.17). The input parameter types of these methods are all `interface{}`. So any value may be passed to the calls to these methods. But for an addressable `Value` value `v`, once the `v.Store()` (a shorthand of `(&v).Store()`) call has ever been called, then the subsequent method calls on value `v` must also take argument values with the same [concrete type](#) (§14) as the argument of the first `v.Store()` call, otherwise, panics will occur. A `nil` interface argument will also make the `v.Store()` call panic.

An example:

```

1| package main
2|
3| import (
4|     "fmt"
5|     "sync/atomic"
6| )
7|
8| func main() {
9|     type T struct {a, b, c int}
10|    var ta = T{1, 2, 3}
11|    var v atomic.Value
12|    v.Store(ta)
13|    var tb = v.Load().(T)
14|    fmt.Println(tb)      // {1 2 3}
15|    fmt.Println(ta == tb) // true
16|
17|    v.Store("hello") // will panic
18| }
```

Another example (for Go 1.17+):

```

1| package main
2|
3| import (
4|     "fmt"
5|     "sync/atomic"
6| )
7|
8| func main() {
9|     type T struct {a, b, c int}
10|    var x = T{1, 2, 3}
11|    var y = T{4, 5, 6}
12|    var z = T{7, 8, 9}
13|    var v atomic.Value
```

```

14|     v.Store(x)
15|     fmt.Println(v) // {{1 2 3}}
16|     old := v.Swap(y)
17|     fmt.Println(v)      // {{4 5 6}}
18|     fmt.Println(old.(T)) // {1 2 3}
19|     swapped := v.CompareAndSwap(x, z)
20|     fmt.Println(swapped, v) // false {{4 5 6}}
21|     swapped = v.CompareAndSwap(y, z)
22|     fmt.Println(swapped, v) // true {{7 8 9}}
23|

```

In fact, we can also use the atomic pointer functions explained in the last section to do atomic operations for values of any type, with one more level indirection. Both ways have their respective advantages and disadvantages. Which way should be used depends on the requirements in practice.

Memory Order Guarantee Made by Atomic Operations in Go

Please read [Go memory model](#) (§41) for details.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Memory Order Guarantees in Go

About Memory Ordering

Many compilers (at compile time) and CPU processors (at run time) often make some optimizations by adjusting the instruction orders, so that the instruction execution orders may differ from the orders presented in code. Instruction ordering is also often called [memory ordering](#).

Surely, instruction reordering can't be arbitrary. The basic requirement for a reordering inside a specified goroutine is the reordering must not be detectable by the goroutine itself if the goroutine doesn't share data with other goroutines. In other words, from the perspective of such a goroutine, it can think its instruction execution order is always the same as the order specified by code, even if instruction reordering really happens inside it.

However, if some goroutines share some data, then instruction reordering happens inside one of these goroutine may be observed by the others goroutines, and affect the behaviors of all these goroutines. Sharing data between goroutines is common in concurrent programming. If we ignore the results caused by instruction reordering, the behaviors of our concurrent programs might compiler and CPU dependent, and often abnormal.

Here is an unprofessional Go program which doesn't consider instruction reordering. the program is expanded from an example in the official documentation [Go 1 memory model](#).

```

1| package main
2|
3| import "log"
4| import "runtime"
5|
6| var a string
7| var done bool
8|
9| func setup() {
10|     a = "hello, world"
11|     done = true
12|     if done {
13|         log.Println(len(a)) // always 12 once printed
14|     }
15| }
16|
17| func main() {

```

```

18|     go setup()
19|
20|     for !done {
21|         runtime.Gosched()
22|     }
23|     log.Println(a) // expected to print: hello, world
24|

```

The behavior of this program is very possible as we expect, a `hello, world` text will be printed. However, the behavior of this program is compiler and CPU dependent. If the program is compiled with a different compiler, or with a later compiler version, or it runs on a different architecture, the `hello, world` text might not be printed, or a text different from `hello, world` might be printed. The reason is compilers and CPUs may exchange the execution orders of the first two lines in the `setup` function, so the final effect of the `setup` function may become to

```

1| func setup() {
2|     done = true
3|     a = "hello, world"
4|     if done {
5|         log.Println(len(a))
6|     }
7|

```

The `setup` goroutine in the above program is unable to observe the reordering, so the `log.Println(len(a))` line will always print `12` (if this line gets executed before the program exits). However, the main goroutine may observe the reordering, which is why the printed text might be not `hello, world`.

Besides the problem of ignoring memory reordering, there are data races in the program. There are not any synchronizations made in using the variable `a` and `done`. So, the above program is a showcase full of concurrent programming mistakes. A professional Go programmer should not make these mistakes.

We can use the `go build -race` command provided in Go Toolchain to build a program, then we can run the outputted executable to check whether or not there are data races in the program.

Go Memory Model

Sometimes, we need to ensure that the execution of some code lines in a goroutine must happen before (or after) the execution of some code lines in another goroutine (from the view of either of the two goroutines), to keep the correctness of a program. Instruction reordering may cause some

troubles for such circumstances. How should we do to prevent certain possible instruction reordering?

Different CPU architectures provide different fence instructions to prevent different kinds of instruction reordering. Some programming languages provide corresponding functions to insert these fence instructions in code. However, understanding and correctly using the fence instructions raises the bar of concurrent programming.

The design philosophy of Go is to use as fewer features as possible to support as more use cases as possible, at the same time to ensure a good enough overall code execution efficiency. So Go built-in and standard packages don't provide direct ways to use the CPU fence instructions. In fact, CPU fence instructions are used in implementing all kinds of synchronization techniques supported in Go. So, we should use these synchronization techniques to ensure expected code execution orders.

The remaining of the current article will list some guaranteed (and non-guaranteed) code execution orders in Go, which are mentioned or not mentioned in [Go 1 memory model ↗](#) and other official Go documentation.

In the following descriptions, if we say event A is guaranteed to happen before event B, it means any of the goroutines involved in the two events will observe that any of the statements presented before event A in source code will be executed before any of the statements presented after event B in source code. For other irrelevant goroutines, the observed orders may be different from the just described.

The creation of a goroutine happens before the execution of the goroutine

In the following function, the assignment `x, y = 123, 789` will be executed before the call `fmt.Println(x)`, and the call `fmt.Println(x)` will be executed before the call `fmt.Println(y)`.

```

1| var x, y int
2| func f1() {
3|     x, y = 123, 789
4|     go func() {
5|         fmt.Println(x)
6|         go func() {
7|             fmt.Println(y)
8|         }()
9|     }()
10| }
```

However, the execution orders of the three in the following function are not deterministic. There are data races in this function.

```

1| var x, y int
2| func f2() {
3|     go func() {
4|         // Might print 0, 123, or some others.
5|         fmt.Println(x)
6|     }()
7|     go func() {
8|         // Might print 0, 789, or some others.
9|         fmt.Println(y)
10|    }()
11|    x, y = 123, 789
12| }
```

Channel operations related order guarantees

Go 1 memory model lists the following three channel related order guarantees.

1. The n th successful send to a channel happens before the n th successful receive from that channel completes, no matter that channel is buffered or unbuffered.
2. The n th successful receive from a channel with capacity m happens before the $(n+m)$ th successful send to that channel completes. In particular, if that channel is unbuffered ($m == 0$), the n th successful receive from that channel happens before the n th successful send on that channel completes.
3. The closing of a channel happens before a receive completes if the receive returns a zero value because the channel is closed.

In fact, the completion of the n th successful send to a channel and the completion of the n th successful receive from the same channel are the same event.

Here is an example show some guaranteed code execution orders in using an unbuffered channel.

```

1| func f3() {
2|     var a, b int
3|     var c = make(chan bool)
4|
5|     go func() {
6|         a = 1
7|         c <- true
8|         if b != 1 { // impossible
9|             panic("b != 1") // will never happen
10| }
```

```

10|      }
11|  }()
12|
13|  go func() {
14|      b = 1
15|      <-c
16|      if a != 1 { // impossible
17|          panic("a != 1") // will never happen
18|      }
19|  }()
20| }
```

Here, for the two new created goroutines, the following orders are guaranteed:

- the execution of the assignment `b = 1` absolutely ends before the evaluation of the condition `b != 1`.
- the execution of the assignment `a = 1` absolutely ends before the evaluation of the condition `a != 1`.

So the two calls to `panic` in the above example will never get executed. However, the `panic` calls in the following example may get executed.

```

1| func f4() {
2|     var a, b, x, y int
3|     c := make(chan bool)
4|
5|     go func() {
6|         a = 1
7|         c <- true
8|         x = 1
9|     }()
10|
11|     go func() {
12|         b = 1
13|         <-c
14|         y = 1
15|     }()
16|
17|     // Many data races are in this goroutine.
18|     // Don't write code as such.
19|     go func() {
20|         if x == 1 {
21|             if a != 1 { // possible
22|                 panic("a != 1") // may happen
23|             }
24|         }
25|     }()
26| }
```

```

24|         if b != 1 { // possible
25|             panic("b != 1") // may happen
26|         }
27|
28|
29|         if y == 1 {
30|             if a != 1 { // possible
31|                 panic("a != 1") // may happen
32|             }
33|             if b != 1 { // possible
34|                 panic("b != 1") // may happen
35|             }
36|         }
37|     }()
38|

```

Here, for the third goroutine, which is irrelevant to the operations on channel `c`. It will not be guaranteed to observe the orders observed by the first two new created goroutines. So, any of the four `panic` calls may get executed.

In fact, most compiler implementations do guarantee the four `panic` calls in the above example will never get executed, however, the Go official documentation never makes such guarantees. So the code in the above example is not cross-compiler or cross-compiler-version compatible. We should stick to the Go official documentation to write professional Go code.

Here is an example using a buffered channel.

```

1| func f5() {
2|     var k, l, m, n, x, y int
3|     c := make(chan bool, 2)
4|
5|     go func() {
6|         k = 1
7|         c <- true
8|         l = 1
9|         c <- true
10|        m = 1
11|        c <- true
12|        n = 1
13|    }()
14|
15|    go func() {
16|        x = 1
17|        <-c

```

```

18|     y = 1
19| }()
20| }
```

The following orders are guaranteed:

- the execution of `k = 1` ends before the execution of `y = 1`.
- the execution of `x = 1` ends before the execution of `n = 1`.

However, the execution of `x = 1` is not guaranteed to happen before the execution of `l = 1` and `m = 1`, and the execution of `l = 1` and `m = 1` is not guaranteed to happen before the execution of `y = 1`.

The following is an example on channel closing. In this example, the execution of `k = 1` is guaranteed to end before the execution of `y = 1`, but not guaranteed to end before the execution of `x = 1`,

```

1| func f6() {
2|     var k, x, y int
3|     c := make(chan bool, 1)
4|
5|     go func() {
6|         c <- true
7|         k = 1
8|         close(c)
9|     }()
10|
11|    go func() {
12|        <-c
13|        x = 1
14|        <-c
15|        y = 1
16|    }()
17| }
```

Mutex related order guarantees

The followings are the mutex related order guarantees in Go.

1. For an addressable value `m` of type `Mutex` or `RWMutex` in the `sync` standard package, the n th successful `m.Unlock()` method call happens before the $(n+1)$ th `m.Lock()` method call returns.

2. For an addressable value `rw` of type `RWMutex`, if its *n*th `rw.Lock()` method call has returned, then its successful *m*th `rw.Unlock()` method call happens before the return of any `rw.RLock()` method call which is guaranteed to happen after the *n*th `rw.Lock()` method call returns.
3. For an addressable value `rw` of type `RWMutex`, if its *n*th `rw.RLock()` method call has returned, then its *m*th successful `rw.RUnlock()` method call, where $m \leq n$, happens before the return of any `rw.Lock()` method call which is guaranteed to happen after the *n*th `rw.RLock()` method call returns.

In the following example, the following orders are guaranteed:

- the execution of `a = 1` ends before the execution of `b = 1`.
- the execution of `m = 1` ends before the execution of `n = 1`.
- the execution of `x = 1` ends before the execution of `y = 1`.

```

1| func fab() {
2|     var a, b int
3|     var l sync.Mutex // or sync.RWMutex
4|
5|     l.Lock()
6|     go func() {
7|         l.Lock()
8|         b = 1
9|         l.Unlock()
10|    }()
11|    go func() {
12|        a = 1
13|        l.Unlock()
14|    }()
15| }
16|
17| func fmn() {
18|     var m, n int
19|     var l sync.RWMutex
20|
21|     l.RLock()
22|     go func() {
23|         l.Lock()
24|         n = 1
25|         l.Unlock()
26|     }()
27|     go func() {
28|         m = 1

```

```

29|     l.RUnlock()
30| }()
31| }
32|
33| func fxy() {
34|     var x, y int
35|     var l sync.RWMutex
36|
37|     l.Lock()
38|     go func() {
39|         l.RLock()
40|         y = 1
41|         l.RUnlock()
42|     }()
43|     go func() {
44|         x = 1
45|         l.Unlock()
46|     }()
47| }
```

Note, in the following code, by the official Go documentation, the execution of `p = 1` is not guaranteed to end before the execution of `q = 1`, though most compilers do make such guarantees.

```

1| var p, q int
2| func fpq() {
3|     var l sync.Mutex
4|     p = 1
5|     l.Lock()
6|     l.Unlock()
7|     q = 1
8| }
```

Order guarantees made by `sync.WaitGroup` values

At a given time, assume the counter maintained by an addressable `sync.WaitGroup` value `wg` is not zero. If there is a group of `wg.Add(n)` method calls invoked after the given time, and we can make sure that only the last returned call among the group of calls will modify the counter maintained by `wg` to zero, then each of the group of calls is guaranteed to happen before the return of a `wg.Wait` method call which is invoked after the given time.

Note, `wg.Done()` is equivalent to `wg.Add(-1)`.

Please read [the explanations for the sync.WaitGroup type](#) (§39) to get how to use sync.WaitGroup values.

Order guarantees made by sync.Once values

Please read [the explanations for the sync.Once type](#) (§39) to get the order guarantees made by sync.Once values and how to use sync.Once values.

Order guarantees made by sync.Cond values

It is some hard to make a clear description for the order guarantees made by sync.Cond values.

Please read [the explanations for the sync.Cond type](#) (§39) to get how to use sync.Cond values.

Atomic operations related order guarantees

Since Go 1.19, the Go 1 memory model documentation formally specifies that all atomic operations executed in Go programs behave as though executed in some sequentially consistent order. If the effect of an atomic operation A is observed by atomic operation B, then A is synchronized before B.

By the descriptions, in the following code, the atomic write operation on the variable b is guaranteed to happen before the atomic read operation with result 1 on the same variable.

Consequently, the write operation on the variable a is also guaranteed to happen before the read operation on the same variable. So the following program is guaranteed to print 2 .

```
1| package main
2|
3| import (
4|     "fmt"
5|     "runtime"
6|     "sync/atomic"
7| )
8|
9| func main() {
10|     var a, b int32 = 0, 0
11|
12|     go func() {
13|         a = 2
14|         atomic.StoreInt32(&b, 1)
15|     }()
}
```

```
16|  
17|     for {  
18|         if n := atomic.LoadInt32(&b); n == 1 {  
19|             // The following line always prints 2.  
20|             fmt.Println(a)  
21|             break  
22|         }  
23|         runtime.Gosched()  
24|     }  
25| }
```

Please read [this article](#) (§40) to get how to do atomic operations.

Finalizers related order guarantees

A call to `runtime.SetFinalizer(x, f)` happens before the finalization call `f(x)`.

(The **Go 101** book is still being improved frequently from time to time. Please visit [go101.org](#) or follow [@go100and1](#) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit [tapirgames.com](#) to get more information about these games. Hope you enjoy them.)

Common Concurrent Programming Mistakes

Go is a language supporting built-in concurrent programming. By using the `go` keyword to create goroutines (light weight threads) and by [using](#) (§37) [channels](#) (§21) and [other concurrency](#) (§40) [synchronization techniques](#) (§39) provided in Go, concurrent programming becomes easy, flexible and enjoyable.

One the other hand, Go doesn't prevent Go programmers from making some concurrent programming mistakes which are caused by either carelessness or lacking of experience. The remaining of the current article will show some common mistakes in Go concurrent programming, to help Go programmers avoid making such mistakes.

No Synchronizations When Synchronizations Are Needed

Code lines might be [not executed by their appearance order](#) (§41).

There are two mistakes in the following program.

- First, the read of `b` in the main goroutine and the write of `b` in the new goroutine might cause data races.
- Second, the condition `b == true` can't ensure that `a != nil` in the main goroutine.

Compilers and CPUs may make optimizations by [reordering instructions](#) (§41) in the new goroutine, so the assignment of `b` may happen before the assignment of `a` at run time, which makes that slice `a` is still `nil` when the elements of `a` are modified in the main goroutine.

```

1| package main
2|
3| import (
4|     "time"
5|     "runtime"
6| )
7|
8| func main() {
9|     var a []int // nil
10|    var b bool // false
11|
12|    // a new goroutine
13|    go func () {

```

```

14|     a = make([]int, 3)
15|     b = true // write b
16| }()
17|
18|     for !b { // read b
19|         time.Sleep(time.Second)
20|         runtime.Gosched()
21|     }
22|     a[0], a[1], a[2] = 0, 1, 2 // might panic
23| }
```

The above program may run well on one computer, but may panic on another one, or it runs well when it is compiled by one compiler, but panics when another compiler is used.

We should use channels or the synchronization techniques provided in the `sync` standard package to ensure the memory orders. For example,

```

1| package main
2|
3| func main() {
4|     var a []int = nil
5|     c := make(chan struct{})
6|
7|     go func () {
8|         a = make([]int, 3)
9|         c <- struct{}{}
10|    }()
11|
12|     <-c
13|     // The next line will not panic for sure.
14|     a[0], a[1], a[2] = 0, 1, 2
15| }
```

Use `time.Sleep` Calls to Do Synchronizations

Let's view a simple example.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "time"
6| )
7| 
```

```

8| func main() {
9|     var x = 123
10|
11|    go func() {
12|        x = 789 // write x
13|    }()
14|
15|    time.Sleep(time.Second)
16|    fmt.Println(x) // read x
17| }
```

We expect this program to print 789. In fact, it really prints 789, almost always, in running. But is it a program with good synchronization? No! The reason is Go runtime doesn't guarantee the write of `x` happens before the read of `x` for sure. Under certain conditions, such as most CPU resources are consumed by some other computation-intensive programs running on the same OS, the write of `x` might happen after the read of `x`. This is why we should never use `time.Sleep` calls to do synchronizations in formal projects.

Let's view another example.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "time"
6| )
7|
8| func main() {
9|     var n = 123
10|
11|    c := make(chan int)
12|
13|    go func() {
14|        c <- n + 0
15|    }()
16|
17|    time.Sleep(time.Second)
18|    n = 789
19|    fmt.Println(<-c)
20| }
```

What do you expect the program will output? 123, or 789? In fact, the output is compiler dependent. For the standard Go compiler 1.21.n, it is very possible the program will output 123. But in theory, it might also output 789.

Now, let's change `c <- n + 0` to `c <- n` and run the program again, you will find the output becomes to 789 (for the standard Go compiler 1.21.n). Again, the output is compiler dependent.

Yes, there are data races in the above program. The expression `n` might be evaluated before, after, or when the assignment statement `n = 789` is processed. The `time.Sleep` call can't guarantee the evaluation of `n` happens before the assignment is processed.

For this specified example, we should store the value to be sent in a temporary value before creating the new goroutine and send the temporary value instead in the new goroutine to remove the data races.

```

1| ...
2|     tmp := n
3|     go func() {
4|         c <- tmp
5|     }()
6| ...

```

Leave Goroutines Hanging

Hanging goroutines are the goroutines staying in blocking state for ever. There are many reasons leading goroutines into hanging. For example,

- a goroutine tries to receive a value from a channel which no more other goroutines will send values to.
- a goroutine tries to send a value to nil channel or to a channel which no more other goroutines will receive values from.
- a goroutine is dead locked by itself.
- a group of goroutines are dead locked by each other.
- a goroutine is blocked when executing a `select` code block without `default` branch, and all the channel operations following the `case` keywords in the `select` code block keep blocking for ever.

Except sometimes we deliberately let the main goroutine in a program hanging to avoid the program exiting, most other hanging goroutine cases are unexpected. It is hard for Go runtime to judge whether or not a goroutine in blocking state is hanging or stays in blocking state temporarily, so Go runtime will never release the resources consumed by a hanging goroutine.

In the [first-response-wins](#) (§37) channel use case, if the capacity of the channel which is used a future is not large enough, some slower response goroutines will hang when trying to send a result

to the future channel. For example, if the following function is called, there will be 4 goroutines stay in blocking state for ever.

```

1| func request() int {
2|     c := make(chan int)
3|     for i := 0; i < 5; i++ {
4|         i := i
5|         go func() {
6|             c <- i // 4 goroutines will hang here.
7|         }()
8|     }
9|     return <-c
10| }
```

To avoid the four goroutines hanging, the capacity of channel `c` must be at least 4.

In [the second way to implement the first-response-wins](#) (§37) channel use case, if the channel which is used as a future/promise is an unbuffered channel, like the following code shows, it is possible that the channel receiver will miss all responses and hang.

```

1| func request() int {
2|     c := make(chan int)
3|     for i := 0; i < 5; i++ {
4|         i := i
5|         go func() {
6|             select {
7|                 case c <- i:
8|             default:
9|             }
10|         }()
11|     }
12|     return <-c // might hang here
13| }
```

The reason why the receiver goroutine might hang is that if the five try-send operations all happen before the receive operation `<-c` is ready, then all the five try-send operations will fail to send values so that the caller goroutine will never receive a value.

Changing the channel `c` as a buffered channel will guarantee at least one of the five try-send operations succeed so that the caller goroutine will never hang in the above function.

Copy Values of the Types in the `sync` Standard Package

In practice, values of the types (except the `Locker` interface values) in the `sync` standard package should never be copied. We should only copy pointers of such values.

The following is bad concurrent programming example. In this example, when the `Counter.Value` method is called, a `Counter` receiver value will be copied. As a field of the receiver value, the respective `Mutex` field of the `Counter` receiver value will also be copied. The copy is not synchronized, so the copied `Mutex` value might be corrupted. Even if it is not corrupted, what it protects is the use of the copied field `n`, which is meaningless generally.

```

1| import "sync"
2|
3| type Counter struct {
4|     sync.Mutex
5|     n int64
6| }
7|
8| // This method is okay.
9| func (c *Counter) Increase(d int64) (r int64) {
10|    c.Lock()
11|    c.n += d
12|    r = c.n
13|    c.Unlock()
14|    return
15| }
16|
17| // The method is bad. When it is called,
18| // the Counter receiver value will be copied.
19| func (c Counter) Value() (r int64) {
20|    c.Lock()
21|    r = c.n
22|    c.Unlock()
23|    return
24| }
```

We should change the receiver type of the `Value` method to the pointer type `*Counter` to avoid copying `sync.Mutex` values.

The `go vet` command provided in Go Toolchain will report potential bad value copies.

Call the `sync.WaitGroup.Add` Method at Wrong Places

Each `sync.WaitGroup` value maintains a counter internally. The initial value of the counter is zero. If the counter of a `WaitGroup` value is zero, a call to the `Wait` method of the `WaitGroup` value will not block, otherwise, the call blocks until the counter value becomes zero.

To make the uses of `WaitGroup` value meaningful, when the counter of a `WaitGroup` value is zero, the next call to the `Add` method of the `WaitGroup` value must happen before the next call to the `Wait` method of the `WaitGroup` value.

For example, in the following program, the `Add` method is called at an improper place, which makes that the final printed number is not always `100`. In fact, the final printed number of the program may be an arbitrary number in the range $[0, 100]$. The reason is none of the `Add` method calls are guaranteed to happen before the `Wait` method call, which causes none of the `Done` method calls are guaranteed to happen before the `Wait` method call returns.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "sync"
6|     "sync/atomic"
7| )
8|
9| func main() {
10|     var wg sync.WaitGroup
11|     var x int32 = 0
12|     for i := 0; i < 100; i++ {
13|         go func() {
14|             wg.Add(1)
15|             atomic.AddInt32(&x, 1)
16|             wg.Done()
17|         }()
18|     }
19|
20|     fmt.Println("Wait . . .")
21|     wg.Wait()
22|     fmt.Println	atomic.LoadInt32(&x))
23| }
```

To make the program behave as expected, we should move the `Add` method calls out of the new goroutines created in the `for` loop, as the following code shown.

```

1| ...
2|     for i := 0; i < 100; i++ {
```

```

3|     wg.Add(1)
4|     go func() {
5|         atomic.AddInt32(&x, 1)
6|     wg.Done()
7| }
8|
9| ...

```

Use Channels as Futures/Promises Improperly

From the article [channel use cases](#) (§37), we know that some functions will return [channels as futures](#) (§37). Assume `fa` and `fb` are two such functions, then the following call uses future arguments improperly.

```
1| doSomethingWithFutureArguments(<-fa(), <-fb())
```

In the above code line, the generations of the two arguments are processed sequentially, instead of concurrently.

We should modify it as the following to process them concurrently.

```

1| ca, cb := fa(), fb()
2| doSomethingWithFutureArguments(<-ca, <-cb)

```

Close Channels Not From the Last Active Sender Goroutine

A common mistake made by Go programmers is closing a channel when there are still some other goroutines will potentially send values to the channel later. When such a potential send (to the closed channel) really happens, a panic might occur.

This mistake was ever made in some famous Go projects, such as [this bug](#) and [this bug](#) in the kubernetes project.

Please read [this article](#) (§38) for explanations on how to safely and gracefully close channels.

Do 64-bit Atomic Operations on Values Which Are Not Guaranteed to Be 8-byte Aligned

The address of the value involved in a non-method 64-bit atomic operation is required to be 8-byte aligned. Failure to do so may make the current goroutine panic. For the standard Go compiler, such failure can only [happen on 32-bit architectures](#). Since Go 1.19, we can use 64-bit method atomic operations to avoid the drawback. Please read [memory layouts](#) (§44) to get how to guarantee the addresses of 64-bit word 8-byte aligned on 32-bit OSes.

Not Pay Attention to Too Many Resources Are Consumed by Calls to the `time.After` Function

The `After` function in the `time` standard package returns [a channel for delay notification](#) (§37). The function is convenient, however each of its calls will create a new value of the `time.Timer` type. The new created `Timer` value will keep alive in the duration specified by the passed argument to the `After` function. If the function is called many times in a certain period, there will be many alive `Timer` values accumulated so that much memory and computation is consumed.

For example, if the following `longRunning` function is called and there are millions of messages coming in one minute, then there will be millions of `Timer` values alive in a certain small period (several seconds), even if most of these `Timer` values have already become useless.

```

1| import (
2|     "fmt"
3|     "time"
4| )
5|
6| // The function will return if a message
7| // arrival interval is larger than one minute.
8| func longRunning(messages <-chan string) {
9|     for {
10|         select {
11|             case <-time.After(time.Minute):
12|                 return
13|             case msg := <-messages:
14|                 fmt.Println(msg)
15|         }
16|     }
17| }
```

To avoid too many `Timer` values being created in the above code, we should use (and reuse) a single `Timer` value to do the same job.

```

1| func longRunning(messages <-chan string) {
2|     timer := time.NewTimer(time.Minute)
3|     defer timer.Stop()
4|
5|     for {
6|         select {
7|             case <-timer.C: // expires (timeout)
8|                 return
9|             case msg := <-messages:
10|                 fmt.Println(msg)
11|
12|                 // This "if" block is important.
13|                 if !timer.Stop() {
14|                     <-timer.C
15|                 }
16|             }
17|
18|             // Reset to reuse.
19|             timer.Reset(time.Minute)
20|         }
21|     }

```

Note, the `if` code block is used to discard/drain a possible timer notification which is sent in the small period when executing the second branch code block.

Use `time.Timer` Values Incorrectly

An idiomatic use example of `time.Timer` values has been shown in the last section. Some explanations:

- the `Stop` method of a `*Timer` value returns `false` if the corresponding `Timer` value has already expired or been stopped. If the `Stop` method returns `false`, and we know the `Timer` value has not been stopped yet, then the `Timer` value must have already expired.
- after a `Timer` value is stopped, its `C` channel field can only contain most one timeout notification.
- we should take out the timeout notification, if it hasn't been taken out, from a timeout `Timer` value after the `Timer` value is stopped and before resetting and reusing the `Timer` value. This is the meaningfulness of the `if` code block in the example in the last section.

The `Reset` method of a `*Timer` value must be called when the corresponding `Timer` value has already expired or been stopped, otherwise, a data race may occur between the `Reset` call and a

possible notification send to the `C` channel field of the `Timer` value.

If the first case branch of the `select` block is selected, it means the `Timer` value has already expired, so we don't need to stop it, for the sent notification has already been taken out. However, we must stop the timer in the second branch to check whether or not a timeout notification exists. If it does exist, we should drain it before reusing the timer, otherwise, the notification will be fired immediately in the next loop step.

For example, the following program is very possible to exit in about one second, instead of ten seconds. And more importantly, the program is not data race free.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "time"
6| )
7|
8| func main() {
9|     start := time.Now()
10|    timer := time.NewTimer(time.Second/2)
11|    select {
12|        case <-timer.C:
13|        default:
14|            // Most likely go here.
15|            time.Sleep(time.Second)
16|    }
17|    // Potential data race in the next line.
18|    timer.Reset(time.Second * 10)
19|    <-timer.C
20|    fmt.Println(time.Since(start)) // about 1s
21| }
```

A `time.Timer` value can be lefted in non-stopping status when it is not used any more, but it is recommended to stop it in the end.

It is bug prone and not recommended to use a `time.Timer` value concurrently among multiple goroutines.

We should not rely on the return value of a `Reset` method call. The return result of the `Reset` method exists just for compatibility purpose.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Memory Blocks

Go is a language which supports automatic memory management, such as automatic memory allocation and automatic garbage collection. So Go programmers can do programming without handling the underlying verbose memory management. This not only brings much convenience and saves Go programmers lots of time, but also helps Go programmers avoid many careless bugs.

Although knowing the underlying memory management implementation details is not necessary for Go programmers to write Go code, understanding some concepts and being aware of some facts in the memory management implementation by the standard Go compiler and runtime is very helpful for Go programmers to write high quality Go code.

This article will explain some concepts and list some facts of the implementation of memory block allocation and garbage collection by the standard Go compiler and runtime. Other aspects, such as memory apply and memory release in memory management, will not be touched in this article.

Memory Blocks

A memory block is a continuous memory segment to host [value parts](#) (§17) at run time. Different memory blocks may have different sizes, to host different value parts. One memory block may host multiple value parts at the same time, but each value part can only be hosted within one memory block, no matter how large the size of that value part is. In other words, for any value part, it never crosses memory blocks.

There are many reasons when one memory block may host multiple value parts. Some of them:

- a struct value often have several fields. So when a memory block is allocated for a struct value, the memory block will also host (the direct parts of) these field values.
- an array values often have many elements. So when a memory block is allocated for a array value, the memory block will also host (the direct parts of) the array element values.
- the underlying element sequences of two slices may be hosted on the same memory block, the two element sequences even can overlap with each other.

A Value References the Memory Blocks Which Host Its Value Parts

We have known that a value part can reference another value part. Here, we extend the reference definition by saying a memory block is referenced by all the value parts it hosts. So if a value part v

is referenced by another value part, then the other value will also reference the memory block hosting `v`, indirectly.

When Will Memory Blocks Be Allocated?

In Go, memory blocks may be allocated but not limited at following situations:

- explicitly call the `new` and `make` built-in functions. A `new` call will always allocate exactly one memory block. A `make` call will allocate more than one memory blocks to host the direct part and underlying part(s) of the created slice, map or channel value.
- create maps, slices and anonymous functions with corresponding literals. More than one memory blocks may be allocated in each of the processes.
- declare variables.
- assign non-interface values to interface values (when the non-interface value is not a pointer value).
- concatenate non-constant strings.
- convert strings to byte or rune slices, and vice versa, except [some special compiler optimization cases](#) (§19).
- convert integers to strings.
- call the built-in `append` function (when the capacity of the base slice is not large enough).
- add a new key-element entry pair into a map (when the underlying hash table needs to be resized).

Where Will Memory Blocks Be Allocated On?

For every Go program compiled by the official standard Go compiler, at run time, each goroutine will maintain a stack, which is a memory segment. It acts as a memory pool for some memory blocks to be allocated from/on. Before Go Toolchain 1.19, the initial size of a stack is always 2KiB. Since Go Toolchain 1.19, the initial size is [adaptive](#). The stack of a goroutine will grow and shrink as needed in goroutine running. The minimum stack size is 2KiB.

(Please note, there is a global limit of stack size each goroutine may reach. If a goroutine exceeds the limit while growing its stack, the program crashes. As of Go Toolchain 1.21.n, the default maximum stack size is 1 GB on 64-bit systems, and 250 MB on 32-bit systems. We can call the `SetMaxStack` function in the `runtime/debug` standard package to change the size. And please note that, by the current official standard Go compiler implementation, the actual allowed maximum stack size is the largest power of 2 which is not larger than the `MaxStack` setting. So for the default setting, the actual allowed maximum stack size is 512 MiB on 64-bit systems, and 128 MiB on 32-bit systems.)

Memory blocks can be allocated on stacks. Memory blocks allocated on the stack of a goroutine can only be used (referenced) in the goroutine internally. They are goroutine localized resources. They are not safe to be referenced crossing goroutines. A goroutine can access or modify the value parts hosted on a memory block allocated on the stack of the goroutine without using any data synchronization techniques.

Heap is a singleton in each program. It is a virtual concept. If a memory block is not allocated on any goroutine stack, then we say the memory block is allocated on heap. Value parts hosted on memory blocks allocated on heap can be used by multiple goroutines. In other words, they can be used concurrently. Their uses should be synchronized when needed.

Heap is a conservative place to allocate memory blocks on. If compilers detect a memory block will be referenced crossing goroutines or can't easily confirm whether or not the memory block is safe to be put on the stack of a goroutine, then the memory block will be allocated on heap at run time. This means some values which can be safely allocated on stacks may also be allocated on heap.

In fact, stacks are not essential for Go programs. Go compiler/runtime can allocate all memory block on heap. Supporting stacks is just to make Go programs run more efficiently:

- allocating memory blocks on stacks is much faster than on heap.
- memory blocks allocated on a stack don't need to be garbage collected.
- stack memory blocks are more CPU cache friendly than heap ones.

If a memory block is allocated somewhere, we can also say the value parts hosted on the memory block are allocated on the same place.

If some value parts of a local variable declared in a function is allocated on heap, we can say the value parts (and the variable) escape to heap. By using Go Toolchain, we can run `go build -gcflags -m` to check which local values (value parts) will escape to heap at run time. As mentioned above, the current escape analyzer in the standard Go compiler is still not perfect, many local value parts can be allocated on stacks safely will still escape to heap.

An active value part allocated on heap still in use must be referenced by at least one value part allocated on a stack. If a value escaping to heap is a declared local variable, and assume its type is T , Go runtime will create (a memory block for) an implicit pointer of type $*T$ on the stack of the current goroutine. The value of the pointer stores the address of the memory block allocated for the variable on heap (a.k.a., the address of the local variable of type T). Go compiler will also replace all uses of the variable with dereferences of the pointer value at compile time. The $*T$ pointer value on stack may be marked as dead since a later time, so the reference relation from it to the T value on heap will disappear. The reference relation from the $*T$ value on stack to the T value on heap plays an important role in the garbage collection process which will be described below.

Similarly, we can view each package-level variable is allocated on heap, and the variable is referenced by an implicit pointer which is allocated on a global memory zone. In fact, the implicit pointer references the direct part of the package-level variable, and the direct part of the variable references some other value parts.

A memory block allocated on heap may be referenced by multiple value parts allocated on different stacks at the same time.

Some facts:

- if a field of a struct value escapes to heap, then the whole struct value will also escape to heap.
- if an element of an array value escapes to heap, then the whole array value will also escape to heap.
- if an element of a slice value escapes to heap, then all the elements of the slice will also escape to heap.
- if a value (part) v is referenced by a value (part) which escapes to heap, then the value (part) v will also escape to heap.

A memory block created by calling `new` function may be allocated on heap or stacks. This is different to C++.

When the size of a goroutine stack changes (for stack growth or shrinkage), a new memory segment will be allocated for the stack. So the memory blocks allocated on the stack will very likely be moved, or their addresses will change. Consequently, the pointers, which must be also allocated on the stack, referencing these memory blocks also need to be modified accordingly. The following is such an example.

```
1| package main
2|
3| // The following directive is to prevent
4| // calls to the function f being inlined.
5| //go:noinline
6| func f(i int) byte {
7|     var a [1<<20]byte // make stack grow
8|     return a[i]
9| }
10|
11| func main(){
12|     var x int
13|     println(&x)
14|     f(100)
15|     println(&x)
16| }
```

We will find that the two printed addresses are different (as of the standard Go compiler v1.21.n).

When Can a Memory Block Be Collected?

Memory blocks allocated for direct parts of package-level variables will never be collected.

The stack of a goroutine will be collected as a whole when the goroutine exits. So there is no need to collect the memory blocks allocated on stacks, individually, one by one. Stacks are not collected by the garbage collector.

For a memory block allocated on heap, it can be safely collected only if it is no longer referenced (either directly or indirectly) by all the value parts allocated on goroutine stacks and the global memory zone. We call such memory blocks as unused memory blocks. Unused memory blocks on heap will be collected by the garbage collector.

Here is an example to show when some memory blocks can be collected:

```

1| package main
2|
3| var p *int
4|
5| func main() {
6|     done := make(chan bool)
7|     // "done" will be used in main and the following
8|     // new goroutine, so it will be allocated on heap.
9|
10|    go func() {
11|        x, y, z := 123, 456, 789
12|        _ = z // z can be allocated on stack safely.
13|        p = &x // For x and y are both ever referenced
14|        p = &y // by the global p, so they will be both
15|                  // allocated on heap.
16|
17|        // Now, x is not referenced by anyone, so
18|        // its memory block can be collected now.
19|
20|        p = nil
21|        // Now, y is also not referenced by anyone,
22|        // so its memory block can be collected now.
23|
24|        done <- true
25|    }()
26|

```

```

27|     <-done
28|     // Now the above goroutine exits, the done channel
29|     // is not used any more, a smart compiler may
30|     // think it can be collected now.
31|
32|     // ...
33| }
```

Sometimes, smart compilers, such as the standard Go compiler, may make some optimizations so that some references are removed earlier than we expect. Here is such an example.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     // Assume the length of the slice is so large
7|     // that its elements must be allocated on heap.
8|     bs := make([]byte, 1 << 31)
9|
10|    // A smart compiler can detect that the
11|    // underlying part of the slice bs will never be
12|    // used later, so that the underlying part of the
13|    // slice bs can be garbage collected safely now.
14|
15|    fmt.Println(len(bs))
16| }
```

Please read [value parts](#) (§17) to learn the internal structures of slice values.

By the way, sometimes, we may hope the slice `bs` is guaranteed to not being garbage collected until `fmt.Println` is called, then we can use a `runtime.KeepAlive` function call to tell garbage collectors that the slice `bs` and the value parts referenced by it are still in use.

For example,

```

1| package main
2|
3| import "fmt"
4| import "runtime"
5|
6| func main() {
7|     bs := make([]int, 1000000)
8|
9|     fmt.Println(len(bs))
```

```

10|
11| // A runtime.KeepAlive(bs) call is also
12| // okay for this specified example.
13| runtime.KeepAlive(&bs)
14| }

```

How Are Unused Memory Blocks Detected?

The current standard Go compiler (v1.21.n) uses a concurrent, tri-color, mark-sweep garbage collector. Here this article will only make a simple explanation for the algorithm.

A garbage collection (GC) process is divided into two phases, the mark phase and the sweep phase. In the mark phase, the collector (a group of goroutines actually) uses the tri-color algorithm to analyze which memory blocks are unused.

The following quote is taken from [a Go blog article](#) and is modified a bit to make it clearer.

At the start of a GC cycle all heap memory blocks are white. The GC visits all roots, which are objects directly accessible by the application such as globals and things on the stack, and colors these grey. The GC then chooses a grey object, blackens it, and then scans it for pointers to other objects. When this scan finds a pointer to a white memory block, it turns that object grey. This process repeats until there are no more grey objects. At this point, white (heap) memory blocks are known to be unreachable and can be reused.

(About why the algorithm uses three colors instead of two colors, please search "write barrier golang" for details. Here only provides two references: [eliminate STW stack re-scanning](#) and [mbarrier.go](#).)

In the sweep phase, the marked unused memory blocks will be collected.

An unused memory block may not be released to OS immediately after it is collected, so that it can be reused for new some value parts. Don't worry, the official Go runtime is much less memory greedy than most Java runtimes.

The GC algorithm is a non-compacting one, so it will not move memory blocks to rearrange them.

When Will a New Garbage Collection Process Start?

Garbage collection processes will consume much CPU resources and some memory resources. So there is not always a garbage collection process in running. A new garbage collection process will be only triggered when some run-time metrics reach certain conditions. How the conditions are defined is a garbage collection pacer problem.

The garbage collection pacer implementation of the official standard Go runtime is still being improved from version to version. So it is hard to describe the implementation precisely and keep the descriptions up-to-date at the same time. Here, I just list some reference articles on this topic:

- About [the GOGC and GOMEMLIMIT environment variables ↗](#) (note that the GOMEMLIMIT environment variable is only supported since Go 1.19).
- [A Guide to the Go Garbage Collector ↗](#).
- [GC Pacer Redesign ↗](#).
- The "Garbage Collection" chapter of my [Go Optimizations 101 ↗](#) book (it is a paid book).

(The **Go 101** book is still being improved frequently from time to time. Please visit [go101.org ↗](#) or follow [@go100and1 ↗](#) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit [tapirgames.com ↗](#) to get more information about these games. Hope you enjoy them.)

Memory Layouts

This articles will introduce type alignment and size guarantees in Go. It is essential to know the guarantees to estimate the sizes of struct types and properly use the 64-bit functions in `sync/atomic` standard package.

Go is a C family language, so many concepts talked in this article are shared with C language.

Type Alignment Guarantees in Go

To fully utilize CPU directives and get the best performance, the (starting) addresses of the memory blocks allocated for values of a specified type must be aligned as multiples of an integer N, then N is called the value address alignment guarantee of the type, or simply the alignment guarantee of the type. We can also say the addresses of addressable values of the type are guaranteed to be N-byte aligned.

In fact, each type has two alignment guarantees, one is for when it is used as field types of other (struct) types, the other is for other cases (when it is used for a variable declaration, array element type, etc). We call the former one the field alignment guarantee of that type, and call the latter one the general alignment guarantee of that type.

For a type `T`, we can call `unsafe.Alignof(t)` to get its general alignment guarantee, where `t` is a non-field value of type `T`, and call `unsafe.Alignof(x.t)` to get its field alignment guarantee, where `x` is a struct value and `t` is a field value of type `T`.

Calls to the functions in the `unsafe` standard code packages are always evaluated at compile time.

At run time, for a value `t` of type `T`, we can call `reflect.TypeOf(t).Align()` to get the general alignment guarantee of type `T`, and call `reflect.TypeOf(t).FieldAlign()` to get the field alignment guarantee of type `T`.

For the current standard Go compiler (v1.21.n), the field alignment guarantee and the general alignment guarantee of a type are always equal. For `gccgo` compiler, the statement is false.

Go specification only mentions [a little on type alignment guarantees ↗](#):

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 the alignment of a variable of the array's element type.

So Go specification doesn't specify the exact alignment guarantees for any types. It just specifies some minimal requirements.

For the same compiler, the exact type alignment guarantees may be different between different architectures and between different compiler versions. For the current version (v1.21.n) of the standard Go compiler, the alignment guarantees are listed here.

type	alignment guarantee
bool, uint8, int8	1
uint16, int16	2
uint32, int32	4
float32, complex64	4
arrays	depend on element types
structs	depend on field types
other types	size of a native word

Here, the size of a native word (or machine word) is 4-byte on 32-bit architectures and 8-byte on 64-bit architectures.

This means, for the current version of the standard Go compiler, the alignment guarantees of other types may be either 4 or 8, depends on different build target architectures. This is also true for `gccgo`.

Generally, we don't need to care about the value address alignments in Go programming, except that we want to optimizing memory consumption, or write portable programs which using the 64-bit functions from `sync/atomic` package. Please read the following two sections for details.

Type Sizes and Structure Padding

Go specification only makes following [type size guarantees](#) :

type	size in bytes
uint8, int8	1
uint16, int16	2
uint32, int32, float32	4

<code>uint64, int64</code>	8
<code>float64, complex64</code>	8
<code>complex128</code>	16
<code>uint, int</code>	implementation-specific, generally 4 on 32-bit architectures, and 8 on 64-bit architectures.
<code>uintptr</code>	implementation-specific, large enough to store the uninterpreted bits of a pointer value.

Go specification doesn't make value size guarantees for other kinds of types. The full list of sizes of different types settled by the standard Go compiler are listed in [value copy costs](#) (§34).

The standard Go compiler (and `gccgo`) will ensure the size of values of a type is a multiple of the alignment guarantee of the type.

To satisfy type alignment guarantee rules mentioned previously, Go compilers may pad some bytes between fields of struct values. This makes the value size of a struct type may be not a simple sum of the sizes of all fields of the type.

The following is an example showing how bytes are padded between struct fields. We have already learned that

- the alignment guarantee and size of the built-in type `int8` are both one byte.
- the alignment guarantee and size of the built-in type `int16` are both two bytes.
- the size of the built-in type `int64` is 8 bytes, the alignment guarantee of type `int64` is 4 bytes on 32-bit architectures and 8 bytes on 64-bit architectures.
- the alignment guarantees of the types `T1` and `T2` are their respective largest field alignment guarantees, a.k.a., the alignment guarantee of the `int64` field. So their alignment guarantees are both 4 bytes on 32-bit architectures and 8 bytes on 64-bit architectures.
- the sizes of the types `T1` and `T2` must be multiples of their respective alignment guarantees, a.k.a., $4N$ bytes on 32-bit architectures and $8N$ bytes on 64-bit architectures.

```

1| type T1 struct {
2|     a int8
3|
4|     // On 64-bit architectures, to make field b
5|     // 8-byte aligned, 7 bytes need to be padded
6|     // here. On 32-bit architectures, to make
7|     // field b 4-byte aligned, 3 bytes need to be
8|     // padded here.

```

```

9|
10|     b int64
11|     c int16
12|
13|     // To make the size of type T1 be a multiple
14|     // of the alignment guarantee of T1, on 64-bit
15|     // architectures, 6 bytes need to be padded
16|     // here, and on 32-bit architectures, 2 bytes
17|     // need to be padded here.
18|
19| // The size of T1 is 24 (= 1 + 7 + 8 + 2 + 6)
20| // bytes on 64-bit architectures and is 16
21| // (= 1 + 3 + 8 + 2 + 2) on 32-bit architectures.
22|
23| type T2 struct {
24|     a int8
25|
26|     // To make field c 2-byte aligned, one byte
27|     // needs to be padded here on both 64-bit
28|     // and 32-bit architectures.
29|
30|     c int16
31|
32|     // On 64-bit architectures, to make field b
33|     // 8-byte aligned, 4 bytes need to be padded
34|     // here. On 32-bit architectures, field b is
35|     // already 4-byte aligned, so no bytes need
36|     // to be padded here.
37|
38|     b int64
39|
40| // The size of T2 is 16 (= 1 + 1 + 2 + 4 + 8)
41| // bytes on 64-bit architectures, and is 12
42| // (= 1 + 1 + 2 + 8) on 32-bit architectures.

```

Although T1 and T2 have the same field set, their sizes are different.

One interesting fact for the standard Go compiler is that sometimes zero sized fields may affect structure padding. Please read [this question in the unofficial Go FAQ](#) (§51) for details.

The Alignment Requirement for 64-bit Word Atomic Operations

64-bit words mean values of types whose underlying types are `int64` or `uint64`.

The article [atomic operations](#) (§40) mentions a fact that 64-bit atomic operations on a 64-bit word require the address of the 64-bit word must be 8-byte aligned. This is not a problem for the current 64-bit architectures supported by the standard Go compiler, because 64-bit words on these 64-bit architectures are always 8-byte aligned.

However, on 32-bit architectures, the alignment guarantee made by the standard Go compiler for 64-bit words is only 4 bytes. 64-bit atomic operations on a 64-bit word which is not 8-byte aligned will panic at runtime. Worse, on very old CPU architectures, 64-bit atomic functions are not supported.

At the end of the [sync/atomic documentation](#), it mentions:

On x86-32, the 64-bit functions use instructions unavailable before the Pentium MMX.

On non-Linux ARM, the 64-bit functions use instructions unavailable before the ARMv6k core.

On both ARM and x86-32, it is the caller's responsibility to arrange for 64-bit alignment of 64-bit words accessed atomically. The first word in a variable or in an allocated struct, array, or slice can be relied upon to be 64-bit aligned.

So, things are not very bad for two reasons.

1. The very old CPU architectures are not mainstream architectures nowadays. If a program needs to do synchronization for 64-bit words on these architectures, there are [other synchronization techniques](#) (§39) to rescue.
2. On other not-very-old 32-bit architectures, there are some ways to ensure some 64-bit words are relied upon to be 64-bit aligned.

The ways are described as **the first (64-bit) word in a variable or in an allocated struct, array, or slice can be relied upon to be 64-bit aligned**. What does the word **allocated** mean? We can think an **allocated value** as a **declared** variable, a value returned by the built-in `make` function, or the value referenced by a value returned by the built-in `new` function. If a slice value derives from an allocated array and the first element of the slice is the first element of the array, then the slice value can also be viewed as an allocated value.

The description of which 64-bit words can be relied upon to be 64-bit aligned on 32-bit architectures is somewhat conservative. There are more 64-bit words which can be relied upon to be 8-byte aligned. In fact, if the first element of an array or slice whose element type is a 64-bit word type can be relied upon to be 64-bit aligned, then all elements in the array/slice can also be accessed atomically. It is just some subtle and verbose to make a simple clear description to include all the

64-bit words can be relied upon to be 64-bit aligned on 32-bit architectures, so the official documentation just makes a conservative description.

Here is an example which lists some 64-bit words which are safe or unsafe to be accessed atomically on both 64-bit and 32-bit architectures.

```
1| type (
2|     T1 struct {
3|         v uint64
4|     }
5|
6|     T2 struct {
7|         _ int16
8|         x T1
9|         y *T1
10|    }
11|
12|     T3 struct {
13|         _ int16
14|         x [6]int64
15|         y *[6]int64
16|     }
17| )
18|
19| var a int64      // a is safe
20| var b T1        // b.v is safe
21| var c [6]int64 // c[0] is safe
22|
23| var d T2 // d.x.v is unsafe
24| var e T3 // e.x[0] is unsafe
25|
26| func f() {
27|     var f int64          // f is safe
28|     var g = []int64{5: 0} // g[0] is safe
29|
30|     var h = e.x[:] // h[0] is unsafe
31|
32|     // Here, d.y.v and e.y[0] are both safe,
33|     // for *d.y and *e.y are both allocated.
34|     d.y = new(T1)
35|     e.y = &[6]int64{}
36|
37|     _, _, _ = f, g, h
38| }
```

```

40| // In fact, all elements in c, g and e.y.v are
41| // safe to be accessed atomically, though Go
42| // official documentation never makes the guarantees.

```

If a 64-bit word field (generally the first one) of a struct type will be accessed atomically in code, we should always use allocated values of the struct type to guarantee the atomically accessed fields always can be relied upon to be 8-byte aligned on 32-bit architectures. When this struct type is used as the type of a field of another struct type, we should arrange the field as the first field of the other struct type, and always use allocated values of the other struct type.

Sometimes, if we can't make sure whether or not a 64-bit word can be accessed atomically, we can use a value of type `[15]byte` to determine the address for the 64-bit word at run time. For example,

```

1| package mylib
2|
3| import (
4|     "unsafe"
5|     "sync/atomic"
6| )
7|
8| type Counter struct {
9|     x [15]byte // instead of "x uint64"
10| }
11|
12| func (c *Counter) xAddr() *uint64 {
13|     // The return must be 8-byte aligned.
14|     return (*uint64)(unsafe.Pointer(
15|         (uintptr(unsafe.Pointer(&c.x)) + 7)/8*8))
16| }
17|
18| func (c *Counter) Add(delta uint64) {
19|     p := c.xAddr()
20|     atomic.AddUint64(p, delta)
21| }
22|
23| func (c *Counter) Value() uint64 {
24|     return atomic.LoadUint64(c.xAddr())
25| }

```

By using this solution, the `Counter` type can be embedded in other user types freely and safely, even on 32-bit architectures. The drawback of this solution is there are seven bytes being wasted for every value of `Counter` type and it uses unsafe pointers.

Go 1.19 introduced a more elegant way to guarantee 8-byte alignments for some values. Go 1.19 added [several atomic types](#) (§40) in the sync/atomic standard package. The types include `atomic.Int64` and `atomic.Uint64`, which values are guaranteed to be 8-byte aligned, even on 32-bit architectures. We may make use of this fact to make some 64-bit words always 8-byte aligned on 32-bit architectures. For example, in the following code, the `x` field of any value of the type `T` is always 8-byte aligned, in any situations, either on 64-bit or 32-bit architectures.

```
1| type T struct {  
2|     _ [0]atomic.Int64  
3|     x int64  
4| }
```

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Memory Leaking Scenarios

When programming in a language supporting auto garbage collection, generally we don't need care about memory leaking problems, for the runtime will collect unused memory regularly. However, we do need to be aware of some special scenarios which may cause kind-of or real memory leaking. The remaining of the current article will list several such scenarios.

Kind-of Memory Leaking Caused by Substrings

Go specification doesn't specify whether or not the result string and base string involved in a substring expression should share the same underlying [memory block](#) (§43) to host [the underlying byte sequences](#) (§19) of the two strings. The standard Go compiler/runtime does let them share the same underlying memory block. This is a good design, which is both memory and CPU consuming wise. But it may cause kind-of memory leaking sometimes.

For example, after the `demo` function in the following example is called, there will be about 1M bytes memory leaking (kind of), until the package-level variable `s0` is modified again elsewhere.

```

1| var s0 string // a package-level variable
2|
3| // A demo purpose function.
4| func f(s1 string) {
5|     s0 = s1[:50]
6|     // Now, s0 shares the same underlying memory block
7|     // with s1. Although s1 is not alive now, but s0
8|     // is still alive, so the memory block they share
9|     // couldn't be collected, though there are only 50
10|    // bytes used in the block and all other bytes in
11|    // the block become unavailable.
12| }
13|
14| func demo() {
15|     s := createStringWithLengthOnHeap(1 << 20) // 1M bytes
16|     f(s)
17| }
```

To avoid this kind-of memory leaking, we can convert the substring to a `[]byte` value then convert the `[]byte` value back to `string`.

```

1| func f(s1 string) {
2|     s0 = string([]byte(s1[:50]))
3|

```

The drawback of the above way to avoid the kind-of memory leaking is there are two 50-byte duplicates which happen in the conversion process, one of them is unnecessary.

We can make use of one of [the optimizations](#) (§19) made by the standard Go compiler to avoid the unnecessary duplicate, with a small extra cost of one byte memory wasting.

```

1| func f(s1 string) {
2|     s0 = (" " + s1[:50])[1:]
3|

```

The disadvantage of the above way is the compiler optimization may become invalid later, and the optimization may be not available from other compilers.

The third way to avoid the kind-of memory leaking is to utilize the `strings.Builder` supported since Go 1.10.

```

1| import "strings"
2|
3| func f(s1 string) {
4|     var b strings.Builder
5|     b.Grow(50)
6|     b.WriteString(s1[:50])
7|     s0 = b.String()
8|

```

The disadvantage of the third way is it is a little verbose (by comparing to the first two ways). A good news is, since Go 1.12, we can call the `Repeat` function with the `count` argument as `1` in the `strings` standard package to clone a string. Since Go 1.12, the underlying implementation of `strings.Repeat` will make use of `strings.Builder`, to avoid one unnecessary duplicate.

Since Go 1.18, a `Clone` function has been added to the `strings` standard package. It becomes the best way to do this job.

Kind-of Memory Leaking Caused by Subslices

Similarly to substrings, subslices may also cause kind-of memory leaking. In the following code, after the `g` function is called, most memory occupied by the memory block hosting the elements of `s1` will be lost (if no more values reference the memory block).

```

1| var s0 []int
2|
3| func g(s1 []int) {
4|     // Assume the length of s1 is much larger than 30.
5|     s0 = s1[len(s1)-30:]
6| }
```

If we want to avoid the kind-of memory leaking, we must duplicate the 30 elements for `s0`, so that the aliveness of `s0` will not prevent the memory block hosting the elements of `s1` from being collected.

```

1| func g(s1 []int) {
2|     s0 = make([]int, 30)
3|     copy(s0, s1[len(s1)-30:])
4|     // Now, the memory block hosting the elements
5|     // of s1 can be collected if no other values
6|     // are referencing the memory block.
7| }
```

Kind-of Memory Leaking Caused by Not Resetting Pointers in Lost Slice Elements

In the following code, after the `h` function is called, the memory block allocated for the first and the last elements of slice `s` will get lost.

```

1| func h() []*int {
2|     s := []*int{new(int), new(int), new(int), new(int)}
3|     // do something with s ...
4|
5|     return s[1:3:3]
6| }
```

As long as the returned slice is still alive, it will prevent any elements of `s` from being collected, which in consequence prevents the two memory blocks allocated for the two `int` values referenced by the first and the last elements of `s` from being collected.

If we want to avoid such kind-of memory leaking, we must reset the pointers stored in the lost elements.

```

1| func h() []*int {
2|     s := []*int{new(int), new(int), new(int), new(int)}
3|     // do something with s ...
```

```

4|
5|     // Reset pointer values.
6|     s[0], s[len(s)-1] = nil, nil
7|     return s[1:3:3]
8| }
```

We often need to reset the pointers for some old slice elements in [slice element deletion operations](#) (§18).

Real Memory Leaking Caused by Hanging Goroutines

Sometimes, some goroutines in a Go program may stay in blocking state for ever. Such goroutines are called hanging goroutines. Go runtime will not kill hanging goroutines, so the resources allocated for (and the memory blocks referenced by) the hanging goroutines will never get garbage collected.

There are two reasons why Go runtime will not kill hanging goroutines. One is that sometimes it is hard for Go runtime to judge whether or not a blocking goroutine will be blocked for ever. The other is sometimes we deliberately make a goroutine hanging. For example, sometimes we may let the main goroutine of a Go program hang to avoid the program exiting.

We should avoid hanging goroutines which are caused by some logic mistakes in code design.

Real Memory Leaking Caused by Not Stopping time.Ticker Values Which Are Not Used Any More

When a `time.Timer` value is not used any more, it will be garbage collected after some time. But this is not true for a `time.Ticker` value. We should stop a `time.Ticker` value when it is not used any more.

Real Memory Leaking Caused by Using Finalizers Improperly

Setting a finalizer for a value which is a member of a cyclic reference group may [prevent all memory blocks allocated for the cyclic reference group from being collected](#). This is real memory leaking, not kind of.

For example, after the following function is called and exits, the memory blocks allocated for `x` and `y` are not guaranteed to be garbage collected in future garbage collecting.

```

1| func memoryLeaking() {
2|     type T struct {
3|         v [1<<20]int
4|         t *T
5|     }
6|
7|     var finalizer = func(t *T) {
8|         fmt.Println("finalizer called")
9|     }
10|
11|    var x, y T
12|
13|    // The SetFinalizer call makes x escape to heap.
14|    runtime.SetFinalizer(&x, finalizer)
15|
16|    // The following line forms a cyclic reference
17|    // group with two members, x and y.
18|    // This causes x and y are not collectable.
19|    x.t, y.t = &y, &x // y also escapes to heap.
20| }
```

So, please avoid setting finalizers for values in a cyclic reference group.

By the way, we [shouldn't use finalizers as object destructors](#) (§51).

Kind-of Resource Leaking by Deferring Function Calls

Please read [this article](#) (§29) for details.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Some Simple Summaries

Index

- [Types whose values may have indirect underlying parts.](#)
- [Types which values can be used as arguments of built-in `len` function \(and `cap`, `close`, `delete`, `make` functions\).](#)
- [Comparison of built-in container types.](#)
- [Types which values can be represented with composite literals \(`T{...}`\).](#)
- [Value sizes of all kinds of types.](#)
- [Types which zero values can be represented with `nil`.](#)
- [Types we can implement methods for.](#)
- [Types which can be embedded in struct types.](#)
- [Functions whose calls will/may be evaluated at compile time.](#)
- [Values that can't be taken addresses.](#)
- [Types which don't support comparisons.](#)
- [Which code elements are allowed to be declared but not used.](#)
- [Named source code elements which can be declared together within `\(...\)`.](#)
- [Named source code elements which can be declared both inside functions and outside any functions.](#)
- [Expressions which evaluation results may contain optional additional values.](#)
- [Ways to block the current goroutine forever by using the channel mechanism.](#)
- [Ways to concatenate strings.](#)
- [Optimizations made by the standard Go compiler.](#)
- [Run-time panic and crash cases.](#)

Types whose values may have indirect underlying parts

Types whose values may have indirect underlying parts:

- string types
- function types
- slice types
- map types
- channel types
- interface types

The answer is based on the implementation of the standard Go compiler/runtime. In fact, whether or not function values may have indirect underlying parts is hardly to prove, and string values and interface values should be viewed as values without indirect underlying parts in logic. Please read [value parts](#) (§17) for details.

Types which values can be used as arguments of built-in `len` function (and `cap`, `close`, `delete`, `make` functions)

	<code>len</code>	<code>cap</code>	<code>close</code>	<code>delete</code>	<code>make</code>
<code>string</code>	Yes				
<code>array</code> (and array pointer)	Yes	Yes			
<code>slice</code>	Yes	Yes			Yes
<code>map</code>	Yes			Yes	Yes
<code>channel</code>	Yes	Yes	Yes		Yes

Values of above types can also be ranged over in for-range loops.

Types which values can be used as arguments of built-in function `len` can be called container types in broad sense.

Comparison of built-in container types

Type	Can New Elements Be Added into Values?	Are Elements of Values Replaceable?	Are Elements of Values Addressable?	Will Element Accesses Modify Value Lengths?	May Values Have Underlying Parts
<code>string</code>	No	No	No	No	Yes ⁽¹⁾
<code>array</code>	No	Yes ⁽²⁾	Yes ⁽²⁾	No	No
<code>slice</code>	No ⁽³⁾	Yes	Yes	No	Yes
<code>map</code>	Yes	Yes	No	No	Yes
<code>channel</code>	Yes ⁽⁴⁾	No	No	Yes	Yes

⁽¹⁾ For the standard Go compiler/runtime.

⁽²⁾ For addressable array values only.

⁽³⁾ Generally, a slice value are modified by assigned another slice value to it by overwriting it. Here,

such cases are not viewed as "add new elements". In fact, slice lengths can also be modified separately by calling the `reflect.SetLen` function. Increase the length of a slice by this way is kind of adding new elements into the slice. But the `reflect.SetLen` function is slow, so it is rarely used.

(4) For buffered channels which are still not full.

Types which values can be represented with composite literals (`T{...}`)

Values of the following four kinds of types can be represented with composite literals:

Type (<code>T</code>)	Is <code>T{}</code> a Zero Value of <code>T</code> ?
<code>struct</code>	Yes
<code>array</code>	Yes
<code>slice</code>	No (zero value is <code>nil</code>)
<code>map</code>	No (zero value is <code>nil</code>)

Value sizes of all kinds of types

Please read [value copy cost](#) (§34) for details.

Types which zero values can be represented with `nil`

The zero values of the following types can be represented with `nil`.

Type (<code>T</code>)	Size of <code>T(nil)</code>
<code>pointer</code>	1 word
<code>slice</code>	3 words
<code>map</code>	1 word
<code>channel</code>	1 word
<code>function</code>	1 word
<code>interface</code>	2 words

The above listed sizes are for the standard Go compiler. One word means 4 bytes on 32-bit architectures and 8 bytes on 64-bit architectures. and [the indirect underlying parts](#) (§17) of a value don't contribute to the size of the value.

The size of a zero value of a type is the same as any other values of the same type.

Types we can implement methods for

Please read [methods in Go](#) (§22) for details.

Types which can be embedded in struct types

Please read [which types can be embedded](#) (§24) for details.

Functions whose calls will/may be evaluated at compile time

If a function call is evaluated at compile time, its return results must be constants.

Function	Return Type	Are Calls Always Evaluated at Compile Time?
<code>unsafe.Sizeof</code>		Yes, always.
<code>unsafe.Alignof</code>	<code>uintptr</code>	<i>But please note that the result of such a call is not viewed as a constant if the argument type of the call is a type parameter.</i>
<code>unsafe.Offsetof</code>		
<code>len</code>	<code>int</code>	<p>Not always.</p> <p>From Go specification:</p> <ul style="list-style-type: none"> • the expression <code>len(s)</code> is constant if <code>s</code> is a string constant. • the expressions <code>len(s)</code> and <code>cap(s)</code> are constants if the type of <code>s</code> is an array or pointer to an array and the expression <code>s</code> does not contain channel receives or (non-constant) function calls.

Function	Return Type	<small>Please see FAQ item 51 for more information.</small>
<code>cap</code>		<small>Are calls always evaluated at compile time?</small> At compile-time, the evaluation result is not viewed as a constant if the argument type of the call is a type parameter .
<code>real</code>	The result is an untyped value. Its default type is <code>float64</code> .	Not always.
<code>imag</code>		From Go spec : the expressions <code>real(s)</code> and <code>imag(s)</code> are constants if <code>s</code> is a complex constant.
<code>complex</code>	The result is an untyped value. Its default type is <code>complex128</code> .	Not always. From Go spec : the expression <code>complex(sr, si)</code> is constant if both <code>sr</code> and <code>si</code> are numeric constants.

Addressable and unaddressable values

Please read [this FAQ item](#) (§51) to get which values are addressable or unaddressable.

Types which don't support comparisons

Please read [this FAQ item](#) (§51) to get which values are addressable or unaddressable.

Which code elements are allowed to be declared but not used

	Allowed to Be Declared but Not Used?
<code>import</code>	No
<code>type</code>	Yes
<code>variable</code>	Yes for package-level variables. No for local variables (for the standard compiler).
<code>constant</code>	Yes
<code>function</code>	Yes

	Allowed to Be Declared but Not Used?
label	No

Named source code elements which can be declared together within ()

Following source code elements (of the same kind) can be declared together within ():

- import
- type
- variable
- constant

Functions can't be declared together within (). Also labels.

Named source code elements which can be declared both inside functions and outside any functions

Following named source code elements can be declared both inside functions and outside any functions:

- type
- variable
- constant

Imports must be declared before declarations of other elements (and after the package clause).

Functions can only be declared outside any functions. Anonymous functions can be defined inside other function bodies, but such definitions are not function declarations.

Labels must be declared inside functions.

Expressions which evaluation results may contain optional additional values

The evaluation results of the following expressions may contain optional additional values:

	Syntax	Meaning of The Optional Value (ok in the syntax examples)	Will Omitting the Optional Result Affect Program Behavior?
map element access	e, ok = aMap[key]	whether or not the accessed key is present in the map	No
channel value receive	e, ok = <- aChannel	whether or not the received value was sent before the channel was closed	No
type assertion	v, ok = anInterface.(T)	whether or not the dynamic type of the interface value matches the asserted type	Yes (when the optional bool result is omitted, a panic occurs if the assertion fails.)

Ways to block the current goroutine forever by using the channel mechanism

Without importing any package, we can use the following ways to make the current goroutine enter (and stay in) blocking state forever:

1. send a value to a channel which no ones will receive values from

```
make(chan struct{}) <- struct{}{}
// or
make(chan<- struct{}) <- struct{}{}
```

2. receive a value from a never-closed channel which no values have been and will be sent to

```
<-make(chan struct{})
// or
<-make(<-chan struct{})
// or
for range make(<-chan struct{}) {}
```

3. receive a value from (or send a value to) a nil channel

```
chan struct{}(nil) <- struct{}{}
// or
<-chan struct{}(nil)
```

```
// or  
for range chan struct{}(nil) {}
```

4. use a bare select block

```
select{}
```

Ways to concatenate strings

Please read [strings in Go](#) (§19) for details.

Optimizations made by the standard Go compiler

Please read [the Go 101 wiki article](#) ↗ for this summary.

Run-time panic and crash cases

Please read [the Go 101 wiki article](#) ↗ for this summary.

(The **Go 101** book is still being improved frequently from time to time. Please visit [go101.org](#) ↗ or follow [@go100and1](#) ↗ to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit [tapirgames.com](#) ↗ to get more information about these games. Hope you enjoy them.)

nil in Go

`nil` is a frequently used and important predeclared identifier in Go. It is the literal representation of zero values of many kinds of types. Many new Go programmers with experiences of some other popular languages may view `nil` as the counterpart of `null` (or `NULL`) in other languages. This is partly right, but there are many differences between `nil` in Go and `null` (or `NULL`) in other languages.

The remaining of this article will list all kinds of facts and details related to `nil`.

nil Is a Predeclared Identifier in Go

We can use `nil` without declaring it.

nil Can Represent Zero Values of Many Types

In Go, `nil` can represent zero values of the following kinds of types:

- pointer types (including type-unsafe ones).
- map types.
- slice types.
- function types.
- channel types.
- interface types.

Predeclared nil Has Not a Default Type

Each of other predeclared identifiers in Go has a default type. For example,

- the default types of `true` and `false` are both `bool` type.
- the default type of `iota` is `int`.

But the predeclared `nil` has not a default type, though it has many possible types. In fact, the predeclared `nil` is the only untyped value who has not a default type in Go. There must be sufficient information for compiler to deduce the type of a `nil` from context.

Example:

```

1| package main
2|
3| func main() {
4|     // There must be sufficient information for
5|     // compiler to deduce the type of a nil value.
6|     _ = (*struct{}{})(nil)
7|     _ = []int(nil)
8|     _ = map[int]bool(nil)
9|     _ = chan string(nil)
10|    _ = (func())(nil)
11|    _ = interface{}(nil)
12|
13|    // These lines are equivalent to the above lines.
14|    var _ *struct{} = nil
15|    var _ []int = nil
16|    var _ map[int]bool = nil
17|    var _ chan string = nil
18|    var _ func() = nil
19|    var _ interface{} = nil
20|
21|    // This following line doesn't compile.
22|    var _ = nil
23| }
```

Predeclared nil Is Not a Keyword in Go

The predeclared `nil` can be shadowed.

Example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     nil := 123
7|     fmt.Println(nil) // 123
8|
9|     // The following line fails to compile,
10|    // for nil represents an int value now
11|    // in this scope.
```

```

12|     var _ map[string]int = nil
13| }
```

(BTW, `null` and `NULL` in many other languages are also not keywords.)

The Sizes of Nil Values With Types of Different Kinds May Be Different

The memory layouts of all values of a type are always the same. `nil` values of the type are not exceptions (assume the zero values of the type can be represented as `nil`). The size of a `nil` value is always the same as the sizes of non-`nil` values whose types are the same as the `nil` value. But `nil` values of different kinds of types may have different sizes.

Example:

```

1| package main
2|
3| import (
4|     "fmt"
5|     "unsafe"
6| )
7|
8| func main() {
9|     var p *struct{} = nil
10|    fmt.Println( unsafe.Sizeof( p ) ) // 8
11|
12|    var s []int = nil
13|    fmt.Println( unsafe.Sizeof( s ) ) // 24
14|
15|    var m map[int]bool = nil
16|    fmt.Println( unsafe.Sizeof( m ) ) // 8
17|
18|    var c chan string = nil
19|    fmt.Println( unsafe.Sizeof( c ) ) // 8
20|
21|    var f func() = nil
22|    fmt.Println( unsafe.Sizeof( f ) ) // 8
23|
24|    var i interface{} = nil
25|    fmt.Println( unsafe.Sizeof( i ) ) // 16
26| }
```

The sizes are compiler and architecture dependent. The above printed results are for 64-bit architectures and the standard Go compiler. For 32-bit architectures, the printed sizes will be half.

For the standard Go compiler, the sizes of two values of different types of the same kind whose zero values can be represented as the predeclared `nil` are always the same. For example, the sizes of all values of all different slice types are the same.

Two Nil Values of Two Different Types May Be Not Comparable

For example, the two comparisons in the following example both fail to compile. The reason is, in each comparison, neither operand can be implicitly converted to the type of the other.

```
1| // Compilation failure reason: mismatched types.
2| var _ = (*int)(nil) == (*bool)(nil)           // error
3| var _ = (chan int)(nil) == (chan bool)(nil) // error
```

Please read [comparison rules in Go](#) (§48) to get which two values can be compared with each other. Typed `nil` values are not exceptions of the comparison rules.

The code lines in the following example all compile okay.

```
1| type IntPtr *int
2| // The underlying of type IntPtr is *int.
3| var _ = IntPtr(nil) == (*int)(nil)
4|
5| // Every type in Go implements interface{} type.
6| var _ = (interface{})(nil) == (*int)(nil)
7|
8| // Values of a directional channel type can be
9| // converted to the bidirectional channel type
10| // which has the same element type.
11| var _ = (chan int)(nil) == (chan<- int)(nil)
12| var _ = (chan int)(nil) == (<-chan int)(nil)
```

Two Nil Values of the Same Type May Be Not Comparable

In Go, map, slice and function types don't support comparison. Comparing two values, including `nil` values, of an incomparable types is illegal. The following comparisons fail to compile.

```
1| var _ = ([]int)(nil) == ([]int)(nil)
2| var _ = (map[string]int)(nil) == (map[string]int)(nil)
3| var _ = (func())(nil) == (func())(nil)
```

But any values of the above mentioned incomparable types can be compared with the bare `nil` identifier.

```
1| // The following lines compile okay.
2| var _ = ([]int)(nil) == nil
3| var _ = (map[string]int)(nil) == nil
4| var _ = (func())(nil) == nil
```

Two Nil Values May Be Not Equal

If one of the two compared nil values is an interface value and the other is not, assume they are comparable, then the comparison result is always `false`. The reason is the non-interface value will be [converted to the type of the interface value](#) (§23) before making the comparison. The converted interface value has a dynamic type but the other interface value has not. That is why the comparison result is always `false`.

Example:

```
fmt.Println( (interface{})(nil) == (*int)(nil) ) // false
```

Retrieving Elements From Nil Maps Will Not Panic

Retrieving element from a nil map value will always return a zero element value.

For example:

```
1| fmt.Println( (map[string]int)(nil)["key"] ) // 0
2| fmt.Println( (map[int]bool)(nil)[123] )      // false
3| fmt.Println( (map[int]*int64)(nil)[123] )    // <nil>
```

It Is Legal to Range Over Nil Channels, Maps, Slices, and Array Pointers

The number of loop steps by iterate nil maps and slices is zero.

The number of loop steps by iterate a nil array pointer is the length of its corresponding array type. (However, if the length of the corresponding array type is not zero, and the second iteration is neither ignored nor omitted, the iteration will panic at run time.)

Ranging over a nil channel will block the current goroutine for ever.

For example, the following code will print 0, 1, 2, 3 and 4, then block for ever. Hello, world and Bye will never be printed.

```

1| for range []int(nil) {
2|     fmt.Println("Hello")
3|
4|
5| for range map[string]string(nil) {
6|     fmt.Println("world")
7|
8|
9| for i := range (*[5]int)(nil) {
10|    fmt.Println(i)
11|
12|
13| for range chan bool(nil) { // block here
14|     fmt.Println("Bye")
15|

```

Invoking Methods Through Non-Interface Nil Receiver Arguments Will Not Panic

Example:

```

1| package main
2|
3| type Slice []bool
4|
5| func (s Slice) Length() int {
6|     return len(s)
7|
8|
9| func (s Slice) Modify(i int, x bool) {
10|     s[i] = x // panic if s is nil
11|
12|
13| func (p *Slice) DoNothing() {

```

```

14| }
15|
16| func (p *Slice) Append(x bool) {
17|     *p = append(*p, x) // panic if p is nil
18| }
19|
20| func main() {
21|     // The following selectors will not cause panics.
22|     _ = ((Slice)(nil)).Length
23|     _ = ((Slice)(nil)).Modify
24|     _ = ((*Slice)(nil)).DoNothing
25|     _ = ((*Slice)(nil)).Append
26|
27|     // The following two lines will also not panic.
28|     _ = ((Slice)(nil)).Length()
29|     ((*Slice)(nil)).DoNothing()
30|
31|     // The following two lines will panic. But panics
32|     // will not be triggered at the time of invoking
33|     // the methods. They will be triggered on
34|     // dereferencing nil pointers in the method bodies.
35| /*
36|     ((Slice)(nil)).Modify(0, true)
37|     ((*Slice)(nil)).Append(true)
38| */
39| }
```

In fact, the above implementation of the `Append` method is not perfect, it should be modified as the following one.

```

1| func (p *Slice) Append(x bool) {
2|     if p == nil {
3|         *p = []bool{x}
4|         return
5|     }
6|     *p = append(*p, x)
7| }
```

*new(T) Results a Nil T Value if the Zero Value of Type T Is Represented With the Predeclared nil Identifier

Example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     fmt.Println(*new(*int) == nil)           // true
7|     fmt.Println(*new([]int) == nil)          // true
8|     fmt.Println(*new(map[int]bool) == nil)   // true
9|     fmt.Println(*new(chan string) == nil)    // true
10|    fmt.Println(*new(func()) == nil)         // true
11|    fmt.Println(*new(interface{}) == nil)    // true
12| }
```

Summary

In Go, for simplicity and convenience, `nil` is designed as an identifier which can be used to represent the zero values of some kinds of types. It is not a single value. It can represent many values with different memory layouts.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW,

Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Value Conversion, Assignment and Comparison Rules in Go

This article will list all the value comparison, conversion and comparison rules in Go. Please note that type parameter types (used frequently in custom generics) are deliberately ignored in the descriptions of conversion, assignability and comparison rules. In other words, this book doesn't consider the situations in which [custom generics ↗](#) are involved.

Value Conversion Rules

In Go, if a value v can be explicitly converted to type T , the conversion can be represented as the form $(T)(v)$. For most cases, in particular T is a type name (an identifier), the form can be simplified to $T(v)$.

One fact we should know is, when it says a value x can be implicitly converted to a type T , then it means x can also be explicitly converted to type T .

1. the apparent conversion rule

If two types denote the identical type, then their values can be **implicitly** converted to either type of the two.

For example,

- values of type `byte` and `uint8` can be converted to each other.
- values of type `rune` and `int32` can be converted to each other.
- values of type `[]byte` and `[]uint8` can be converted to each other.

Nothing more to explain about this rule, whether you think this case involves conversions or not.

2. underlying type related conversion rules

Given a non-interface value x and a non-interface type T , assume the type of x is Tx ,

- if Tx and T share the same [underlying type](#) (§14) (ignoring struct tags), then x can be explicitly converted to T .

- if either T_x or T is a [unnamed type](#) (§14) and their underlying types are identical (considering struct tags), then x can be **implicitly** converted to T .
- if T_x and T have different underlying types, but both T_x and T are unnamed pointer types and their base types share the same underlying type (ignoring struct tags), then x can be explicitly converted to T .

(Note, the two ***ignoring struct tags*** occurrences have taken effect since Go 1.8.)

An example:

```

1| package main
2|
3| func main() {
4|     // []int, IntSlice and MySlice share
5|     // the same underlying type: []int
6|     type IntSlice []int
7|     type MySlice []int
8|     type Foo = struct{n int `foo`}
9|     type Bar = struct{n int `bar`}
10|
11|    var s  = []int{}
12|    var is = IntSlice{}
13|    var ms = MySlice{}
14|    var x map[Bar]Foo
15|    var y map[foo]Bar
16|
17|    // The two implicit conversions both doesn't work.
18|    /*
19|     is = ms // error
20|     ms = is // error
21|     */
22|
23|    // Must use explicit conversions here.
24|    is = IntSlice(ms)
25|    ms = MySlice(is)
26|    x = map[Bar]Foo(y)
27|    y = map[foo]Bar(x)
28|
29|    // Implicit conversions are okay here.
30|    s = is
31|    is = s
32|    s = ms
33|    ms = s
34| }
```

Pointer related conversion example:

```

1| package main
2|
3| func main() {
4|     type MyInt int
5|     type IntPtr *int
6|     type MyIntPtr *MyInt
7|
8|     var pi = new(int) // the type of pi is *int
9|     // ip and pi have the same underlying type,
10|    // and the type of pi is unnamed, so
11|    // the implicit conversion works.
12|    var ip IntPtr = pi
13|
14|    // var _ *MyInt = pi // can't convert implicitly
15|    var _ = (*MyInt)(pi) // ok, must explicitly
16|
17|    // Values of *int can't be converted to MyIntPtr
18|    // directly, but can indirectly.
19|    /*
20|    var _ MyIntPtr = pi // can't convert implicitly
21|    var _ = MyIntPtr(pi) // can't convert explicitly
22|    */
23|    var _ MyIntPtr = (*MyInt)(pi) // ok
24|    var _ = MyIntPtr((*MyInt)(pi)) // ok
25|
26|    // Values of IntPtr can't be converted to
27|    // MyIntPtr directly, but can indirectly.
28|    /*
29|    var _ MyIntPtr = ip // can't convert implicitly
30|    var _ = MyIntPtr(ip) // can't convert explicitly
31|    */
32|    var _ MyIntPtr = (*MyInt)((*int)(ip)) // ok
33|    var _ = MyIntPtr((*MyInt)((*int)(ip))) // ok
34| }
```

3. channel specific conversion rule

Given a channel value x , assume its type T_x is a bidirectional channel type, T is also a channel type (bidirectional or not). If T_x and T have the identical element type, and either T_x or T is an unnamed type, then x can be **implicitly** converted to T .

Example:

```

1| package main
2|
3| func main() {
4|     type C chan string
5|     type C1 chan<- string
6|     type C2 <-chan string
7|
8|     var ca C
9|     var cb chan string
10|
11|     cb = ca // ok, same underlying type
12|     ca = cb // ok, same underlying type
13|
14|     // The 4 lines compile okay for this 3rd rule.
15|     var _, _ chan<- string = ca, cb // ok
16|     var _, _ <-chan string = ca, cb // ok
17|     var _ C1 = cb // ok
18|     var _ C2 = cb // ok
19|
20|     // Values of C can't be converted
21|     // to C1 and C2 directly.
22|     /*
23|     var _ = C1(ca) // compile error
24|     var _ = C2(ca) // compile error
25|     */
26|
27|     // Values of C can be converted
28|     // to C1 and C2 indirectly.
29|     var _ = C1((chan<- string)(ca)) // ok
30|     var _ = C2((<-chan string)(ca)) // ok
31|     var _ C1 = (chan<- string)(ca) // ok
32|     var _ C2 = (<-chan string)(ca) // ok
33| }
```

4. interface implementation related conversion rules

Given a value x and an interface type I , if the type (or the default type) of x is T_x and T_x implements I , then x can be **implicitly** converted to type I . The conversion result is an interface value (of type I), which boxes

- a copy of x , if T_x is a non-interface type;
- a copy of the dynamic value of x , if T_x is an interface type.

Please read [interfaces in Go](#) (§23) for details and examples.

5. untyped value conversion rule

An untyped value can be **implicitly** converted to type T , if the untyped value can represent as values of type T .

Example:

```

1| package main
2|
3| func main() {
4|     var _ []int = nil
5|     var _ map[string]int = nil
6|     var _ chan string = nil
7|     var _ func()() = nil
8|     var _ *bool = nil
9|     var _ interface{} = nil
10|
11|    var _ int = 123.0
12|    var _ float64 = 123
13|    var _ int32 = 1.23e2
14|    var _ int8 = 1 + 0i
15| }
```

6. constants conversion rule

(This rule is some overlapped with the last one.)

Generally, converting a constant still yields a constant as result. (Except converting a constant string to byte slice or rune slice described in the below 8th rules.)

Given a constant value x and a type T , if x is representable as a value of type T , then x can be explicitly converted to T . In particular if x is an untyped value, then x can be **implicitly** converted to T .

Example:

```

1| package main
2|
3| func main() {
4|     const I = 123
5|     const I1, I2 int8 = 0x7F, -0x80
6|     const I3, I4 int8 = I, 0.0
```

```

7|
8|     const F = 0.123456789
9|     const F32 float32 = F
10|    const F32b float32 = I
11|    const F64 float64 = F
12|    const F64b = float64(I3) // must be explicitly
13|
14|    const C1, C2 complex64 = F, I
15|    const I5 = int(C2) // must be explicitly
16| }
```

7. non-constant number conversion rules

Non-constant floating-point and integer values can be explicitly converted to any floating-point and integer types.

Non-constant complex values can be explicitly converted to any complex types.

Note,

- Complex non-constant values can't be converted to floating-point and integer types.
- Floating-point and integer non-constant values can't be converted to complex types.
- Data overflow and rounding are allowed in non-constant number conversions. When converting a floating-point non-constant number to an integer, the fraction is discarded (truncation towards zero).

An example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     var a, b = 1.6, -1.6 // both are float64
7|     fmt.Println(int(a), int(b)) // 1 -1
8|
9|     var i, j int16 = 0xFFFF, -0x8000
10|    fmt.Println(int8(i), uint16(j)) // -1 32768
11|
12|    var c1 complex64 = 1 + 2i
13|    var _ = complex128(c1)
14| }
```

8. string related conversion rules

If the type (or default type) of a value is an integer type, then the value can be explicitly converted to string types.

A string value can be explicitly converted to a slice type whose underlying type is `[]byte` (a.k.a., `[]uint8`), and vice versa.

A string value can be explicitly converted to a slice type whose underlying type is `[]rune` (a.k.a., `[]int32`), and vice versa.

Please read [strings in Go](#) (§19) for details and examples.

| 9. slices related conversions

Since Go 1.17, a slice may be converted to an array pointer. In such a conversion, if the length of the base array type of the pointer type is larger than the length of the slice, a panic occurs.

Here is [an example](#) (§18).

Since Go 1.20, a slice may be converted to an array. In such a conversion, if the length of the array type is larger than the length of the slice, a panic occurs.

Here is [an example](#) (§18).

| 10. unsafe pointers related conversion rules

A pointer value of any type can be explicitly converted to a type whose underlying type is `unsafe.Pointer`, and vice versa.

An `uintptr` value can be explicitly converted to a type whose underlying type is `unsafe.Pointer`, and vice versa.

Please read [type-unsafe pointers in Go](#) (§25) for details and examples.

Value Assignment Rules

Assignments can be viewed as implicit conversions. Implicit conversion rules are listed among all conversion rules in the last section.

Besides these rules, the destination values in assignments must be addressable values, map index expressions, or the blank identifier.

In an assignment, the source value is copied to the destination value. Precisely speaking, the [direct part](#) (§17) of the source value is copied to the destination value.

Note, parameter passing and result returning are both value assignments actually.

Value Comparison Rules

Go specification [states](#) :

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

So, the comparison rule is much like the assignment rule. In other words, two values are comparable if one of them can be implicitly converted to the type of the other. Right? Almost, for there is another rule which has a higher priority than the above basic comparison rule.

If both of the two operands in a comparison are typed, then their types must be both [comparable types](#) (§14).

By the above rule, if an incomparable type (which must be a non-interface type) implements an interface type, then it is illegal to compare values of the two types, even if values of the former (non-interface) type can be implicitly converted to the latter (interface) type.

Note, although values of slice/map/function types don't support comparisons, they can be compared with untyped nil values (a.k.a., bare `nil` identifiers).

The above described basic rules don't cover all cases. What about if both of the two operands in a comparison are untyped (constant) values? The additional rules are simple:

- untyped boolean values can be compared with untyped boolean values.
- untyped numeric values can be compared with untyped numeric values.
- untyped string values can be compared with untyped string values.

The results of comparing two untyped numeric values obey intuition.

Note, an untyped nil value can't be compared with another untyped nil value.

Any comparison results in an untyped boolean value.

The following example shows some incomparable types related comparisons.

```

1| package main
2|
3| // Some variables of incomparable types.
4| var s []int
5| var m map[int]int

```

```

6| var f func()()
7| var t struct {x []int}
8| var a [5]map[int]int
9|
10| func main() {
11|     // The following lines fail to compile.
12|     /*
13|     _ = s == s
14|     _ = m == m
15|     _ = f == f
16|     _ = t == t
17|     _ = a == a
18|     _ = nil == nil
19|     _ = s == interface{}(nil)
20|     _ = m == interface{}(nil)
21|     _ = f == interface{}(nil)
22|     */
23|
24|     // The following lines compile okay.
25|     _ = s == nil
26|     _ = m == nil
27|     _ = f == nil
28|     _ = 123 == interface{}(nil)
29|     _ = true == interface{}(nil)
30|     _ = "abc" == interface{}(nil)
31| }

```

How Are Two Values Compared?

Assume two values are comparable, and they have the same type T . (If they have different types, one of them must be implicitly convertible to the type of the other. Here we don't consider the cases in which both the two values are untyped.)

1. If T is a boolean type, then the two values are equal only if they are both `true` or both `false`.
2. If T is an integer type, then the two values are equal only if they have the same representation in memory.
3. If T is a floating-point type, then the two values are equal only if any of the following conditions is satisfied:
 - they are both `+Inf`.
 - they are both `-Inf`.
 - each of them is either `-0.0` or `+0.0`.

- they are both not NaN and they have the same bytes representations in memory.
4. If T is a complex type, then the two values are equal only if their real parts (as floating-point values) and imaginary parts (as floating-point values) are both equal.
 5. If T is a pointer type (either safe or unsafe), then the two values are equal only if the memory addresses stored in them are equal.
 6. If T is a channel type, the two channel values are equal if they both reference the same underlying internal channel structure value or they are both nil channels.
 7. If T is a struct type, then each pair of the corresponding fields of the two struct values will be compared (§16).
 8. If T is an array type, then each pair of the corresponding elements of the two array values will be compared (§18).
 9. If T is an interface type, please read how two interface values are compared (§23).
 10. If T is a string type, please read how two string values are compared (§19).

Please note, comparing two interfaces with the same incomparable dynamic type produces a panic. The following is an example in which some panics will occur in comparisons.

```

1| package main
2|
3| func main() {
4|     type T struct {
5|         a interface{}
6|         b int
7|     }
8|     var x interface{} = []int{}
9|     var y = T{a: x}
10|    var z = [3]T{{a: y}}
11|
12|    // Each of the following line can produce a panic.
13|    _ = x == x
14|    _ = y == y
15|    _ = z == z
16| }
```

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Syntax/Semantics Exceptions in Go

This article will list all kinds of syntax/semantics exceptions in Go. Some of these exceptions are syntactic sugars to make programming convenient, some are caused built-in generic privileges, some exists for history reasons, and some exists for other reasons in logic.

Nested function calls

The basic rule:

If the number of the return results of a function call is not zero, and the return results can be used as the whole arguments of another function call, then the former function call can be nested in the latter function call, the former nested call can't mix up with other arguments of the latter nesting call.

Sugar:

If a function call returns exactly one result, then the function call can be always be used as a single-value argument in other function calls, the single-result function call can mix up with other arguments of the nesting function calls.

Example:

```

1| package main
2|
3| import (
4|     "fmt"
5| )
6|
7| func f0() float64 {return 1}
8| func f1() (float64, float64) {return 1, 2}
9| func f2(float64, float64) {}
10| func f3(float64, float64, float64) {}
11| func f4()(x, y []int) {return}
12| func f5()(x map[int]int, y int) {return}
13|
14| type I interface {m()(float64, float64)}
15| type T struct{}
16| func (T) m()(float64, float64) {return 1, 2}
17|
18| func main() {
19|     // These lines compile okay.
20|     f2(f0(), 123)

```

```

21|     f2(f1())
22|     fmt.Println(f1())
23|     _ = complex(f1())
24|     _ = complex(T{}.m())
25|     f2(I(T{}).m())
26|
27|     // These lines don't compile.
28|     /*
29|     f3(123, f1())
30|     f3(f1(), 123)
31|     */
32|
33|     // This line compiles okay only since
34|     // Go Toolchain 1.15.
35|     println(f1())
36|
37|     // The following 3 lines compile okay
38|     // only since Go Toolchain 1.13.
39|     copy(f4())
40|     delete(f5())
41|     _ = complex(I(T{}).m())
42| }
```

Select struct fields

The basic rule:

Pointer values have no fields.

Sugar:

We can select the fields of a struct value through pointers of the struct value.

Example:

```

1| package main
2|
3| type T struct {
4|     x int
5| }
6|
7| func main() {
8|     var t T
9|     var p = &t
10|
11|     p.x *= 2
```

```

12|     // The above line is a sugar of the following line.
13|     (*p).x *= 2
14| }
```

Receiver arguments of method calls

The basic rule:

The methods explicitly declared for type `*T` are not methods of type `T` for sure.

Sugar:

Although the methods explicitly declared for type `*T` are not methods of type `T`, addressable values of type `T` can be used as the receiver arguments of calls to these methods.

Example:

```

1| package main
2|
3| type T struct {
4|     x int
5| }
6|
7| func (pt *T) Double() {
8|     pt.x *= 2
9| }
10|
11| func main() {
12|     // T{3}.Double() // This line fails to compile.
13|
14|     var t = T{3}
15|
16|     t.Double() // t.x == 6 now
17|     // The above line is a sugar of the following line.
18|     (&t).Double() // t.x == 12 now
19| }
```

Take addresses of composite literal values

The basic rule:

Literal values are unaddressable and unaddressable values can't be taken addresses.

Sugar:

Although composite literal values are not addressable, they can be taken addresses explicitly.

Please read [structs](#) (§16) and [containers](#) (§18) for details.

Selectors on named one-Level pointers

The basic rule:

Generally, selectors can't be used on values of [named](#) (§14) pointer types.

Sugar:

If `x` is a value of a named one-level pointer type, and selector `(*x).f` is a legal selector, then the `x.f` is also a legal selector, it can be viewed as a shorthand of `(*x).f`.

Selectors can never be used on values of **multi-level** pointer types, no matter whether the multi-level pointer types are named or not.

Exception of the sugar:

The sugar is only valid if `f` denotes a struct field, it is not valid if `f` denotes a method.

Example:

```

1| package main
2|
3| type T struct {
4|     x int
5| }
6|
7| func (T) y() {
8| }
9|
10| type P *T
11| type PP **T // a multi-level pointer type
12|
13| func main() {
14|     var t T
15|     var p P = &t
16|     var pt = &t    // type of pt is *T
17|     var ppt = &pt // type of ppt is **T
18|     var pp PP = ppt
19|     _ = pp
20|
21|     _ = (*p).x // legal

```

```

22|     _ = p.x    // also legal (for x is a field)
23|
24|     _ = (*p).y // legal
25|     // _ = p.y // illegal (for y is a method)
26|
27|     // Following ones are all illegal.
28|     /*
29|     _ = ppt.x
30|     _ = ppt.y
31|     _ = pp.x
32|     _ = pp.y
33|     */
34| }
```

The addressability of a container and its elements

The basic rule:

If a container is addressable, then its elements are also addressable.

Exception:

Elements of a map are always unaddressable, even if the map itself is addressable.

Sugar:

Elements of a slice are always addressable, even if the slice itself is not addressable.

Example:

```

1| package main
2|
3| func main() {
4|     var m = map[string]int{"abc": 123}
5|     _ = &m // okay
6|
7|     // The exception:
8|     // p = &m["abc"] // error: map elements are unaddressable
9|
10|    // The sugar:
11|    f := func() []int { // return results are unaddressable
12|        return []int{0, 1, 2}
13|    }
14|    // _ = &f() // error: f() is unaddressable
15|    _ = &f()[2] // okay
16| }
```

Modify unaddressable values

The basic rule:

Unaddressable values can't be modified. In other words, unaddressable values shouldn't appear in assignments as destination values.

Exception:

Although map element values are unaddressable, they can be modified and appear in assignments as destination values. (But map elements can't be modified partially, they can only be overwritten wholly, a.k.a., replaced.)

Example:

```

1| package main
2|
3| func main() {
4|     type T struct {
5|         x int
6|     }
7|
8|     var mt = map[string]T{"abc": {123}}
9|     // Map elements are unaddressable.
10|    // _ = &mt["abc"] // error
11|    // Partial modification is not allowed.
12|    // mt["abc"].x = 456 // error
13|    // It is ok to replace a map element as a whole.
14|    mt["abc"] = T{x: 789}
15| }
```

Function Parameters

The basic rule:

Each parameter is a value of some type.

Exception:

The first parameters of the built-in `make` and `new` functions are types.

Function names in one package

The basic rule:

Names of declared functions can't be duplicated in one package.

Exception:

There can be multiple functions declared with names as `init` (and types as `func()`).

Function calls

The basic rule:

Functions whose names are not the blank identifier can be called in user code.

Exception:

`init` functions can't be called in user code.

Functions being used as values

The basic rule:

Declared functions can be used as function values.

Exception:

`init` functions can not be used as function values.

Example:

```
1| package main
2|
3| import "fmt"
4| import "unsafe"
5|
6| func init() {}
7|
8| func main() {
9|     // These ones are okay.
10|    var _ = main
11|    var _ = fmt.Println
12|
13|    // This one fails to compile.
14|    var _ = init
15| }
```

The manners of type argument passing

The basic rule:

In a type argument list, all type arguments are enclosed in square brackets and they are separated by commas in the list.

Exception:

The manners of type argument lists passed to built-in generic type instantiations vary in different forms. Each key type argument of instantiated map types is individually enclosed in square brackets. Other type arguments are not enclosed. The type argument of an instantiation of built-in `new` generic function is enclosed in parentheses. The type argument of an instantiation of the built-in `make` generic function is mixed with value arguments and these type and value arguments are enclosed in parentheses.

Discard return values of function calls

The basic rule:

The return values of a function call can be discarded altogether.

Exception:

The return values of calls to the built-in functions which are documented in the `builtin` and `unsafe` standard packages, can't be discarded, if the called function has return results.

Exception in exception:

The return values of a call to the built-in `copy` and `recover` functions can be all discarded, even if the two functions have return results.

Declared variables

The basic rule:

Declared variables are always addressable.

Exception:

The [predeclared `nil`](#) variable is not addressable.

So, `nil` is an immutable variable.

Argument passing

The basic rule:

An argument can be passed to the corresponding function parameter only if the argument is assignable to the corresponding function parameter type.

Sugar:

If the first slice argument of a copy and append function call is a byte slice, then the second argument can be a string, whereas a string value is not assignable to the second parameter type (also a byte slice). (For an append call, assume the second argument is passed with the form `arg....`)

Example:

```

1| package main
2|
3| func main() {
4|     var bs = []byte{1, 2, 3}
5|     var s = "xyz"
6|
7|     copy(bs, s)
8|     // The above line is a sugar (and an optimization)
9|     // for the following line.
10|    copy(bs, []byte(s))
11|
12|    bs = append(bs, s...)
13|    // The above line is a sugar (and an optimization)
14|    // for the following line.
15|    bs = append(bs, []byte(s)...)
```

Comparisons

The basic rule:

Map, slice and function types don't support comparison.

Exception:

Map, slice and function values can be compared to the predeclared untyped `nil` identifier.

Example:

```

1| package main
2|
3| func main() {
4|     var s1 = []int{1, 2, 3}
5|     var s2 = []int{7, 8, 9}
6|     //_ = s1 == s2 // error: slice values can't be compared
7|     _ = s1 == nil // ok
8|     _ = s2 == nil // ok
9| }
```

```

10|     var m1 = map[string]int{}
11|     var m2 = m1
12|     // _ = m1 == m2 // error: map values can't be compared
13|     _ = m1 == nil
14|     _ = m2 == nil
15|
16|     var f1 = func(){}
17|     var f2 = f1
18|     // _ = f1 == f2 // error: functions can't be compared
19|     _ = f1 == nil
20|     _ = f2 == nil
21| }
```

Comparisons 2

The basic rule:

If a value can be implicitly converted to a comparable type, then the value can be compared to the values of the comparable type.

Exception:

The values of a non-interface incomparable type can't be compared to values of an interface type, even if the non-interface incomparable type implements the interface type (so values of the non-interface incomparable type can be implicitly converted to the interface type).

Please read [comparison rules](#) (§48) for examples.

Blank composite literals

The basic rule:

If the values of a type T can be represented with composite literals, then $T\{\}$ is its zero value.

Exception:

For a map or a slice type T , $T\{\}$ isn't its zero value. Its zero value is represented with `nil`.

Example:

```

1| package main
2|
3| import "fmt"
```

```

4|
5| func main() {
6|     // new(T) returns the address of a zero value of type T.
7|
8|     type T0 struct {
9|         x int
10|    }
11|    fmt.Println( T0{} == *new(T0) ) // true
12|    type T1 [5]int
13|    fmt.Println( T1{} == *new(T1) ) // true
14|
15|    type T2 []int
16|    fmt.Println( T2{} == nil ) // false
17|    type T3 map[int]int
18|    fmt.Println( T3{} == nil ) // false
19| }
```

Container element iterations

The basic rule:

Only container values can be ranged, the iterated values are container elements. The element key/index will also be returned alongside of each iterated element.

Exception 1:

The iterated values are runes if the ranged containers are strings, instead of the byte elements of strings.

Exception 2:

The element index (order) will not be returned alongside of each iterated element when iterating channels.

Sugar:

Array pointers can also be ranged to iterate array elements, though pointers are not containers.

Methods of built-in types

The basic rule:

Generally, built-in types have no methods.

Exception:

The built-in `error` type has a `Error() string` method.

Types of values

The basic rule:

Each value has either a type or a default type.

Exception:

Untyped `nil` has neither a type nor a default type.

Constant values

The basic rule:

Constant values never change. Constants can be assigned to variables.

Exception:

Predeclared `iota` is a built-in constant which is bound with `0`, but its value is not constant. Its value will start from `0` and increase one constant specification by constant specification in a constant declaration, though the increments happen at compile time.

Exception 2:

`iota` can only be used within constant declarations. It can't be assigned to variables in variable declarations.

Behavior change caused by discarding the optional evaluation results of expressions

The basic rule:

Whether or not the optional evaluation result of an expression presents will not affect program behavior.

Exception:

Missing the optional result value in a type assertion will make current goroutine panic if the type assertion fails.

Example:

```
1| package main
2|
3| func main() {
4|     var ok bool
5| }
```

```

6|     var m = map[int]int{}
7|     _, ok = m[123] // will not panic
8|     _ = m[123]      // will not panic
9|
10|    var c = make(chan int, 2)
11|    c <- 123
12|    close(c)
13|    _, ok = <-c // will not panic
14|    _ = <-c      // will not panic
15|
16|    var v interface{} = "abc"
17|    _, ok = v.(int) // will not panic
18|    _ = v.(int)     // will panic!
19| }
```

else keyword followed by a code block

The basic rule:

The `else` keyword must be followed by an explicit code block `{...}`.

Sugar:

The `else` keyword may be followed by a `if` code block (which is implicit).

For example, in the following code, function `foo` compiles okay, but function `bar` doesn't.

```

1| func f() []int {
2|     return nil
3| }
4|
5| func foo() {
6|     if vs := f(); len(vs) == 0 {
7| } else if len(vs) == 1 {
8| }
9|
10|    if vs := f(); len(vs) == 0 {
11| } else {
12|     switch {
13|         case len(vs) == 1:
14|             default:
15|         }
16|     }
17|
18|    if vs := f(); len(vs) == 0 {
19| } else {
```

```
20|     for _, _ = range vs {  
21|     }  
22| }  
23| }  
24|  
25| func bar() {  
26|     if vs := f(); len(vs) == 0 {  
27|     } else switch { // error  
28|     case len(vs) == 1:  
29|     default:  
30|     }  
31|  
32|     if vs := f(); len(vs) == 0 {  
33|     } else for _, _ = range vs { // error  
34|     }  
35| }
```

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Go Details 101

Index:

- Code package related details:
 - [A package can be imported more than once in a source file.](#)
- Control flow related details:
 - [The default branch in switch and select blocks can be put before all case branches, after all case branches, or between case branches.](#)
 - [The numeric constant case expressions in a switch block can't be duplicate, but boolean ones can.](#)
 - [The switch expressions in switch block are always evaluated to typed values.](#)
 - [The default switch expression of a switch block is a typed value true of the predeclared type bool.](#)
 - [Sometimes, the open brace { of an explicit code block can be put on the next line.](#)
 - [Some case branch blocks must be explicit.](#)
 - [Nested deferred function calls can modify return result values of nesting functions.](#)
 - [Some recover calls may be no-ops.](#)
 - [Exit a program with a os.Exit function call and exit a goroutine with a runtime.Goexit function call.](#)
- Operator related details:
 - [The precedence of the increment operator ++ and the decrement -- is lower than the dereference operator * and the address-taking operator &, which are lower than the property selection operator . in selectors.](#)
 - [The type deduction rule for the left untyped operand of a bit-shift operation depends on whether or not the right operand is a constant.](#)
- Pointer related details:
 - [Values of two pointer types with different underlying types can be converted to each other if the base types of their underlying types share the same underlying type.](#)
 - [Addresses of different zero-sized values may be equal, or not.](#)
 - [The base type of a pointer type may be the pointer type itself.](#)
 - [A detail about selector shorthands.](#)
- Container related details:
 - [Sometimes, nested composite literals can be simplified.](#)
 - [In some scenarios, it is ok to use array pointers as arrays.](#)
 - [Retrieving elements from nil maps will not panic. The result is a zero element value.](#)
 - [Deleting an entry from a nil map will not panic. It is a no-op.](#)

- The result slice of an `append` function call may share some elements with the original slice, or not.
- The length of a subslice may be larger than the base slice the subslice derives from.
- Deriving a subslice from a nil slice is ok if all the indexes used in the subslice expression are zero. The result subslice is also a nil slice.
- Ranging over a nil maps or a nil slices is ok, it is a no-op.
- Range over a nil array pointer is ok if the second iteration variable is ignored or omitted.
- The length and capacity of a slice can be modified separately.
- The indexes in slice and array composite literals must be constants and non-negative.
- The constant indexes or keys in slice/array/map composite literals can't be duplicate.
- Elements of unaddressable arrays are also unaddressable, but elements of unaddressable slices are always addressable.
- It is ok to derive subslices from unaddressable slices, but not ok from unaddressable arrays. It is ok to take addresses for elements of unaddressable slices, but not ok for elements of unaddressable arrays.
- Putting entries with `NaN` as keys to a map is like putting the entries in a black hole.
- The capacity of the result slice of a conversion from a string to byte/rune slice may be larger than the length of the result slice.
- For a slice `s`, the loop `for i = range s {....}` is not equivalent to the loop `for i = 0; i < len(s); i++ {....}`.
- The iteration order over maps is not guaranteed to be the same from one iteration to the next.
- If a map entry is created during an iteration of the map, that entry may be iterated during the iteration or may be skipped.
- Function and method related details:
 - A multi-result function call can't mix with other expressions when the call is used as the sources in an assignment or the arguments of another function call.
 - Some function calls are evaluated at compile time.
 - Each method corresponds to an implicit function.
- Interface related details:
 - Comparing two interface values with the same dynamic incomparable type produces a panic.
 - Type assertions can be used to convert a value of an interface type to another interface type, even if the former interface type doesn't implement the latter one.
 - Whether or not the second optional result of a failed type assertion is present will affect the behavior of the type assertion.
 - About the impossible to-interface assertions which can be detected at compile time.
 - Two error values returned by `two_errors.New` calls with the same argument are not equal.

- Channel related details:
 - [Receive-only channels can't be closed.](#)
 - [Sending a value to a closed channel is viewed as a non-blocking operation, and this operation causes a panic.](#)
- More type and value related details:
 - [Types can be declared within function bodies.](#)
 - [For the standard compiler, zero-sized fields in a struct may be treated as one-byte-sized value.](#)
 - [NaN != NaN, Inf == Inf.](#)
 - [Non-exported method names and struct field names from different packages are viewed as different names.](#)
 - [In struct value comparisons, blank fields will be ignored.](#)
- Miscellanies:
 - [Parentheses are required in several rare scenarios to make code compile okay.](#)
 - [Stack overflow is unrecoverable.](#)
 - [Some expression evaluation orders in Go are compiler implementation dependent.](#)
- Standard packages related:
 - [The results of reflect.DeepEqual\(x, y\) and x == y may be different.](#)
 - [The reflect.Value.Bytes\(\) method returns a \[\]byte value, which element type, byte, might be not the same as the Go slice value represented by the receiver parameter.](#)
 - [We should use os.IsNotExist\(err\) instead of err == os.ErrNotExist to check whether or not a file exists.](#)
 - [The flag standard package treats boolean command flags differently than number and string flags.](#)
 - [\[Sp|Ep|P\]rintf functions support positional arguments.](#)

A package can be imported more than once in a source file.

A Go source file can imports the same package multiple times, but the import names must be different. These same-package imports reference the same package instance.

For example:

```
1| package main
2|
3| import "fmt"
4| import "io"
```

```

5| import inout "io"
6|
7| func main() {
8|     fmt.Println(&inout.EOF == &io.EOF) // true
9| }
```

The default branch in switch and select blocks can be put before all case branches, after all case branches, or between case branches.

For example:

```

1|     switch n := rand.Intn(3); n {
2|         case 0: fmt.Println("n == 0")
3|         case 1: fmt.Println("n == 1")
4|         default: fmt.Println("n == 2")
5|     }
6|
7|     switch n := rand.Intn(3); n {
8|         default: fmt.Println("n == 2")
9|         case 0: fmt.Println("n == 0")
10|        case 1: fmt.Println("n == 1")
11|    }
12|
13|     switch n := rand.Intn(3); n {
14|         case 0: fmt.Println("n == 0")
15|         default: fmt.Println("n == 2")
16|         case 1: fmt.Println("n == 1")
17|     }
18|
19|     var x, y chan int
20|
21|     select {
22|         case <-x:
23|         case y <- 1:
24|         default:
25|     }
26|
27|     select {
28|         case <-x:
29|         default:
30|         case y <- 1:
31|     }
```

```

32|     select {
33|     default:
34|     case <-x:
35|     case y <- 1:
36|     }
37|

```

The numeric constant case expressions in a switch block can't be duplicate, but boolean ones can.

For example, the following program fails to compile.

```

1| package main
2|
3| func main() {
4|     switch 123 {
5|     case 123:
6|     case 123: // error: duplicate case
7|     }
8|

```

But the following program compiles okay.

```

1| package main
2|
3| func main() {
4|     switch false {
5|     case false:
6|     case false:
7|     }
8|

```

For reasons, please read [this issue](#). The behavior is compiler dependent. In fact, the standard Go compiler also doesn't allow duplicate string case expressions, but gccgo allows.

The switch expressions in switch block are always evaluated to typed values.

For example, the switch expression 123 in the following switch block is viewed as a value of int instead of an untyped integer. So the following program fails to compile.

```

1| package main
2|
3| func main() {
4|     switch 123 {
5|         case int64(123): // error: mismatched types
6|         case uint32(789): // error: mismatched types
7|     }
8| }
```

The default switch expression of a switch block is a typed value true of the predeclared type bool.

For example, the following program will print true.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     switch { // <=> switch true {
7|         case true: fmt.Println("true")
8|         case false: fmt.Println("false")
9|     }
10| }
```

Sometimes, the open brace { of an explicit code block can be put on the next line.

For example:

```

1| package main
2|
3| func main() {
4|     var i = 0
5| Outer:
6|     for
7|         { // okay on the next line
8|             switch
9|                 { // okay on the next line
10|                     case i == 5:
11|                         break Outer
12|     }
```

```

12|     default:
13|         i++
14|     }
15| }
16| }
```

What result will the following program print? true or false? The answer is true. Please read [line break rules in Go](#) (§28) for reasons.

```

1| package main
2|
3| import "fmt"
4|
5| func False() bool {
6|     return false
7| }
8|
9| func main() {
10|     switch False()
11|     {
12|         case true: fmt.Println("true")
13|         case false: fmt.Println("false")
14|     }
15| }
```

Some case branch blocks must be explicit.

For example, the following program fails to compile.

```

1| func demo(n, m int) (r int) {
2|     switch n {
3|     case 123:
4|         if m > 0 {
5|             goto End
6|         }
7|         r++
8|
9|         End: // syntax error: missing statement after label
10|     default:
11|         r = 1
12|     }
13|     return
14| }
```

To make it compile okay, the case branch code block should be explicit:

```

1| func demo(n, m int) (r int) {
2|     switch n {
3|         case 123: {
4|             if m > 0 {
5|                 goto End
6|             }
7|             r++
8|
9|             End:
10|        }
11|        default:
12|            r = 1
13|        }
14|    return
15| }
```

Alternatively, we can let a semicolon follow the label declaration End::

```

1| func demo(n, m int) (r int) {
2|     switch n {
3|         case 123:
4|             if m > 0 {
5|                 goto End
6|             }
7|             r++
8|
9|             End:;
10|        default:
11|            r = 1
12|        }
13|    return
14| }
```

Please read [line break rules in Go](#) (§28) for reasons.

A nested deferred function calls can modify return result values of its innermost nesting function.

For example:

```

1| package main
2|
3| import "fmt"
4|
5| func F() (r int) {
6|     defer func() {
7|         r = 789
8|     }()
9|
10|    return 123 // <=> r = 123; return
11| }
12|
13| func main() {
14|     fmt.Println(F()) // 789
15| }
```

Some recover calls may be no-ops.

We should call the `recover` function at the right places. Please read [the right places to call the built-in recover function](#) (§31) for details.

Exit a program with a `os.Exit` function call and exit a goroutine with a `runtime.Goexit` function call.

We can exit a program from any function by calling the `os.Exit` function. An `os.Exit` function call takes an `int` code as argument and returns the code to operating system.

An example:

```

1| // exit-example.go
2| package main
3|
4| import "os"
5| import "time"
6|
7| func main() {
8|     go func() {
9|         time.Sleep(time.Second)
10|        os.Exit(1)
11|     }()
12| }
```

```

12|     select{}
13| }
```

Run it:

```

$ go run a.go
exit status 1
$ echo $?
1
```

We can make a goroutine exit by calling the `runtime.Goexit` function. The `runtime.Goexit` function has no parameters.

In the following example, the Java word will not be printed.

```

1| package main
2|
3| import "fmt"
4| import "runtime"
5|
6| func main() {
7|     c := make(chan int)
8|     go func() {
9|         defer func() {c <- 1}()
10|        defer fmt.Println("Go")
11|        func() {
12|            defer fmt.Println("C")
13|            runtime.Goexit()
14|        }()
15|        fmt.Println("Java")
16|    }()
17|    <-c
18| }
```

The precedence of the increment operator `++` and the decrement `--` is lower than the dereference operator `*` and the address-taking operator `&`, which are lower than the property selection operator `.` in selectors.

For example:

```

1| package main
2|
3| import "fmt"
4|
5| type T struct {
6|     x int
7|     y *int
8| }
9|
10| func main() {
11|     var t T
12|     p := &t.x // <=> p := &(t.x)
13|     fmt.Printf("%T\n", p) // *int
14|
15|     *p++ // <=> (*p)++
16|     *p-- // <=> (*p)--
17|
18|     t.y = p
19|     a := *t.y // <=> *(t.y)
20|     fmt.Printf("%T\n", a) // int
21| }
```

The type deduction rule for the left untyped operand of a bit-shift operation depends on whether or not the right operand is a constant.

```

1| package main
2|
3| func main() {
4| }
5|
6| const M = 2
7| // Compiles okay. 1.0 is deduced as an int value.
8| var _ = 1.0 << M
9|
10| var N = 2
11| // Fails to compile. 1.0 is deduced as a float64 value.
12| var _ = 1.0 << N
```

Please read [this article](#) (§8) for reasons.

Values of two pointer types with different underlying types can be converted to each other if the base types of their underlying types share the same underlying type.

An example:

```

1| package main
2|
3| type MyInt int64
4| type Ta     *int64
5| type Tb     *MyInt
6|
7| func main() {
8|     var a Ta
9|     var b Tb
10|
11|    // Direct conversion is not allowed.
12|    //a = Ta(b) // error
13|
14|    // But indirect conversion is possible.
15|    y := (*MyInt)(b)
16|    x := (*int64)(y)
17|    a = x          // <=> the next line
18|    a = (*int64)(y) // <=> the next line
19|    a = (*int64)((*MyInt)(b))
20|    _ = a
21| }
```

Addresses of different zero-sized values may be equal, or not.

Whether or not the addresses of two zero-sized values are equal is compiler and compiler version dependent.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     a := struct{}{}
7|     b := struct{}{}
```

```

8|     x := struct{}{}
9|     y := struct{}{}
10|    m := [10]struct{}{}
11|    n := [10]struct{}{}
12|    o := [10]struct{}{}
13|    p := [10]struct{}{}
14|
15|    fmt.Println(&x, &y, &o, &p)
16|
17|    // For the standard Go compiler (1.21.n),
18|    // x, y, o and p escape to heap, but
19|    // a, b, m and n are allocated on stack.
20|
21|    fmt.Println(&a == &b) // false
22|    fmt.Println(&x == &y) // true
23|    fmt.Println(&a == &x) // false
24|
25|    fmt.Println(&m == &n) // false
26|    fmt.Println(&o == &p) // true
27|    fmt.Println(&n == &p) // false
28| }
```

The outputs indicated in the above code are for the standard Go compiler 1.21.n.

The base type of a pointer type may be the pointer type itself.

An example:

```

1| package main
2|
3| func main() {
4|     type P *P
5|     var p P
6|     p = &p
7|     p = *****p
8| }
```

Similarly,

- the element type of a slice type can be the slice type itself,
- the element type of a map type can be the map type itself,
- the element type of a channel type can be the channel type itself,

- and the argument and result types of a function type can be the function type itself.

```

1| package main
2|
3| func main() {
4|     type S []S
5|     type M map[string]M
6|     type C chan C
7|     type F func(F) F
8|
9|     s := S{0:nil}
10|    s[0] = s
11|    m := M{"Go": nil}
12|    m["Go"] = m
13|    c := make(C, 3)
14|    c <- c; c <- c; c <- c
15|    var f F
16|    f = func(F)f {return f}
17|
18|    _ = s[0][0][0][0][0][0][0][0]
19|    _ = m["Go"]["Go"]["Go"]["Go"]
20|    <-<-<-c
21|    f(f(f(f(f))))
22| }
```

A detail about selector shorthands.

For a pointer value, which type is either named or not, if the base type of its (pointer) type is a struct type, then we can select the fields of the struct value referenced by the pointer value through the pointer value. However, if the type of the pointer value is a named type, then we can't select the methods of the struct value referenced by the pointer value through the pointer value.

```

1| package main
2|
3| type T struct {
4|     x int
5| }
6| func (T) m(){} // T has one method.
7|
8| type P *T // a named one-level pointer type.
9| type PP *P // a named two-level pointer type.
10|
11| func main() {
```

```

12|     var t T
13|     var tp = &t
14|     var tpp = &tp
15|     var p P = tp
16|     var pp PP = &p
17|     tp.x = 12 // okay
18|     p.x = 34 // okay
19|     pp.x = 56 // error: type PP has no field or method x
20|     tpp.x = 78 // error: type **T has no field or method x
21|
22|     tp.m() // okay. Type *T also has a "m" method.
23|     p.m() // error: type P has no field or method m
24|     pp.m() // error: type PP has no field or method m
25|     tpp.m() // error: type **T has no field or method m
26| }
```

Sometimes, nested composite literals can be simplified.

Please read [nested composite literals can be simplified](#) (§18) for details.

In some scenarios, it is ok to use array pointers as arrays.

Please read [use array pointers as arrays](#) (§18) for details.

Retrieving elements from nil maps will not panic. The result is a zero element value.

For example, the `Foo1` and the `Foo2` functions are equivalent, but the function `Foo2` is much tidier than the function `Foo1`.

```

1| func Foo1(m map[string]int) int {
2|     if m != nil {
3|         return m["foo"]
4|     }
5|     return 0
6| }
7|
8| func Foo2(m map[string]int) int {
```

```
9|     return m["foo"]  
10| }
```

Deleting an entry from a nil map will not panic. It is a no-op.

For example, the following program will not panic.

```
1| package main  
2|  
3| func main() {  
4|     var m map[string]int // nil  
5|     delete(m, "foo")  
6| }
```

The result slice of an append function call may share some elements with the original slice, or not.

Please read [append and delete container elements](#) (§18) for details.

The length of a subslice may be larger than the base slice the subslice derives from.

For example,

```
1| package main  
2|  
3| import "fmt"  
4|  
5| func main() {  
6|     s := make([]int, 3, 9)  
7|     fmt.Println(len(s)) // 3  
8|     s2 := s[2:7]  
9|     fmt.Println(len(s2)) // 5  
10| }
```

Please read [derive slices from arrays and slices](#) (§18) for details.

Deriving a subslice from a nil slice is ok if all the indexes used in the subslice expression are zero. The result subslice is also a nil slice.

For example, the following program will not panic at run time.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     var x []int // nil
7|     a := x[:]
8|     b := x[0:0]
9|     c := x[:0:0]
10|    // Print three "true".
11|    fmt.Println(a == nil, b == nil, c == nil)
12| }
```

Please read [derive slices from arrays and slices](#) (§18) for details.

Ranging over a nil maps or a nil slices is ok, it is a no-op.

For example, the following program compiles okay.

```

1| package main
2|
3| func main() {
4|     var s []int // nil
5|     for range s {
6|     }
7|
8|     var m map[string]int // nil
9|     for range m {
10|    }
11| }
```

Range over a nil array pointer is ok if the second iteration variable is ignored or omitted.

For example, the following program will print 01234.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     var a *[5]int // nil
7|     for i, _ := range a {
8|         fmt.Println(i)
9|     }
10| }
```

The length and capacity of a slice can be modified separately.

We can modify the length and capacity of a slice separately through the reflection way. Please read [modify the length and capacity properties of a slice individually](#) (§18) for details.

The indexes in slice and array composite literals must be constants and non-negative.

For example, the following code fails to compile.

```

1| var k = 1
2| // error: index must be non-negative integer constant
3| var x = [2]int{k: 1}
4| // error: index must be non-negative integer constant
5| var y = []int{k: 1}
```

Note, the keys in map composite literals are not required to be constants.

The constant indexes or keys in slice/array/map composite literals can't be duplicate.

For example, the following code fails to compile.

```

1| // error: duplicate index in array literal: 1
2| var a = []bool{0: false, 1: true, 1: true}
```

```

3| // error: duplicate index in array literal: 0
4| var b = [...]string{0: "foo", 1: "bar", 0: "foo"}
5| // error: duplicate key "foo" in map literal
6| var c = map[string]int{"foo": 1, "foo": 2}

```

This feature can be used to [assert some conditions at compile time](#) (§52).

Elements of unaddressable arrays are also unaddressable, but elements of unaddressable slices are always addressable.

The reason is the elements of an array value and the array will be stored in the same memory block when the array is stored in memory. But [the situation is different for slices](#) (§51).

An example:

```

1| package main
2|
3| func main() {
4|     // Container composite literals are unaddressable.
5|
6|     // It is ok to take slice literal element addresses.
7|     _ = &[]int{1}[0] // ok
8|     // Cannot take addresses of array literal elements.
9|     _ = &[5]int{}[0] // error
10|
11|    // It is ok to modify slice literal elements.
12|    []int{1,2,3}[1] = 9 // ok
13|    // Cannot modify array literal elements.
14|    [3]int{1,2,3}[1] = 9 // error
15| }

```

It is ok to derive subslices from unaddressable slices, but not ok from unaddressable arrays.

The reason is the same as the last detail.

An example:

```

1| package main
2|

```

```

3| func main() {
4|     // Map elements are unaddressable in Go.
5|
6|     // The following lines compile okay. Deriving
7|     // slices from unaddressable slices is allowed.
8|     _ = []int{6, 7, 8, 9}[1:3]
9|     var ms = map[string][]int{"abc": {0, 1, 2, 3}}
10|    _ = ms["abc"][1:3]
11|
12|    // The following lines fail to compile. Deriving
13|    // slices from unaddressable arrays is not allowed.
14|    /*
15|     _ = [...]int{6, 7, 8, 9}[1:3] // error
16|     var ma = map[string][4]int{"abc": {0, 1, 2, 3}}
17|     _ = ma["abc"][1:3] // error
18|    */
19| }
```

Putting entries with NaN as keys to a map is like putting the entries in a black hole.

This reason is `NaN != NaN`, which is another detail will be described [below](#). Before Go 1.12, the elements with NaN as keys can only be found out in a `for-range` loop, Since Go 1.12, the elements with NaN as keys can also be printed out by `fmt.Println` alike functions.

```

1| package main
2|
3| import "fmt"
4| import "math"
5|
6| func main() {
7|     var a = math.NaN()
8|     fmt.Println(a) // NaN
9|
10|    var m = map[float64]int{}
11|    m[a] = 123
12|    v, present := m[a]
13|    fmt.Println(v, present) // 0 false
14|    m[a] = 789
15|    v, present = m[a]
16|    fmt.Println(v, present) // 0 false
17|
18|    fmt.Println(m) // map[NaN:789 NaN:123]
```

```

19|     delete(m, a)    // no-op
20|     fmt.Println(m) // map[NaN:789 NaN:123]
21|
22|     for k, v := range m {
23|         fmt.Println(k, v)
24|     }
25|     // the above loop outputs:
26|     // NaN 123
27|     // NaN 789
28| }
```

Please note, before Go 1.12, the two `fmt.Println(m)` calls both printed `map[NaN:<nil> NaN:<nil>]`.

The capacity of the result slice of a conversion from a string to byte/rune slice may be larger than the length of the result slice.

We should not assume the length and the capacity of the result slice are always equal.

In the following example, if the last `fmt.Println` line is removed, the outputs of the two lines before it print the same value 32, otherwise, one print 32 and one print 8 (for the standard Go compiler 1.21.n).

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     s := "a"
7|     x := []byte(s)           // len(s) == 1
8|     fmt.Println(cap([]byte(s))) // 32
9|     fmt.Println(cap(x))       // 8
10|    fmt.Println(x)
11| }
```

[Some buggy code will be written ↗](#) if we assume the length and the capacity of the result slice are always equal.

For a slice `s`, the loop `for i = range s { ... }` is not equivalent to the loop `for i = 0; i < len(s); i++`

{ . . . }.

The respective final values of the iteration variable `i` may be different for the two loops.

```

1| package main
2|
3| import "fmt"
4|
5| var i int
6|
7| func fa(s []int, n int) int {
8|     i = n
9|     for i = 0; i < len(s); i++ {}
10|    return i
11| }
12|
13| func fb(s []int, n int) int {
14|     i = n
15|     for i = range s {}}
16|     return i
17| }
18|
19| func main() {
20|     s := []int{2, 3, 5, 7, 11, 13}
21|     fmt.Println(fa(s, -1), fb(s, -1)) // 6 5
22|     s = nil
23|     fmt.Println(fa(s, -1), fb(s, -1)) // 0 -1
24| }
```

The iteration order over maps is not guaranteed to be the same from one iteration to the next.

For example, the following program will not run infinitely:

```

1| package main
2|
3| import "fmt"
4|
5| func f(m map[byte]byte) string {
6|     bs := make([]byte, 0, 2*len(m))
7|     for k, v := range m {
8|         bs = append(bs, k, v)
```

```

9|     }
10|    return string(bs)
11| }
12|
13| func main() {
14|     m := map[byte]byte{'a':'A', 'b':'B', 'c':'C'}
15|     s0 := f(m)
16|     for i := 1; ; i++{
17|         if s := f(m); s != s0 {
18|             fmt.Println(s0)
19|             fmt.Println(s)
20|             fmt.Println(i)
21|             return
22|         }
23|     }
24| }
```

Please note, the entries in the JSON marshal result on maps are sorted by their keys. And since Go 1.12, printing maps (with the print functions in the standard `fmt` package) also results sorted entries.

If a map entry is created during an iteration of the map, that entry may be iterated during the iteration or may be skipped.

A proof:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     m := map[int]int{0: 0, 1: 100, 2: 200}
7|     r, n, i:= len(m), len(m), 0
8|     for range m {
9|         m[n] = n*100
10|        n++
11|        i++
12|    }
13|    fmt.Printf("%d new entries, iterate %d and skip %d\n",
14|              i, i - r, n - i,
```

```
15|     )
16| }
```

Thanks to Valentin Deleplace for the above [two detail suggestions ↗](#).

A multi-result function call can't mix with other expressions when the call is used as the sources in an assignment or the arguments of another function call.

Please read [use function calls as expressions](#) (§20) for details.

Some function calls are evaluated at compile time.

Please read [some function calls are evaluated at compile time](#) (§20) for details.

Each method corresponds to an implicit function.

Please read [each Method Corresponds to an Implicit Function](#) (§22) for details.

Comparing two interface values with the same dynamic incomparable type produces a panic.

For example:

```
1| package main
2|
3| func main() {
4|     var x interface{} = []int{}
5|     _ = x == x // panic
6| }
```

Type assertions can be used to convert a value of an interface type to another interface type, even if the former interface type doesn't implement the latter one.

For example:

```

1| package main
2|
3| type Foo interface {
4|     foo()
5| }
6|
7| type T int
8| func (T) foo() {}
9|
10| func main() {
11|     var x interface{} = T(123)
12|     // The following two lines fails to compile, for the
13|     // same reason: interface{} does not implement Foo.
14|     /*
15|         var _ Foo = x    // error
16|         var _ = Foo(x) // error
17|     */
18|     // But the following line compiles and runs okay.
19|     var _ = x.(Foo) // okay
20| }
```

Whether or not the second optional result of a type assertion is present will affect the behavior of the type assertion.

If the second optional result presents in a failed type assertion, the type assertion will not produce a panic. Otherwise, a panic will occur. For example:

```

1| package main
2|
3| func main() {
4|     var x interface{} = true
5|
6|     // Assertion fails, but doesn't cause a panic.
7|     _, _ = x.(int)
8|
9|     // Assertion fails, which causes a panic.
10|    _ = x.(int)
11| }
```

About the impossible to-interface assertions which can be detected at compile time.

At compile time, some to-interface assertions can be deduced as impossible to succeed. For example, the assertion shown in the following code:

```

1| package main
2|
3| type Ia interface {
4|     m()
5| }
6|
7| type Ib interface {
8|     m() int
9| }
10|
11| type T struct{}
12|
13| func (T) m() {}
14|
15| func main() {
16|     var x Ia = T{}
17|     _ = x.(Ib) // panic: main.T is not main.Ib
18| }
```

Such assertions will not make code compilations fail (but the program will panic at run time). Since Go Toolchain 1.15, the `go vet` command warns on such assertions.

Two error values returned by two errors.New calls with the same argument are not equal.

The reason is the `errors.New` function will copy the input string argument and use a pointer to the copied string as the dynamic value of the returned `error` value. Two different calls will produce two different pointers.

```

1| package main
2|
3| import "fmt"
4| import "errors"
5|
6| func main() {
```

```

7|     notfound := "not found"
8|     a, b := errors.New(notfound), errors.New(notfound)
9|     fmt.Println(a == b) // false
10| }
```

Receive-only channels can't be closed.

For example, the following code fails to compile.

```

1| package main
2|
3| func main() {
4| }
5|
6| func foo(c <-chan int) {
7|     close(c) // error: cannot close receive-only channel
8| }
```

Sending a value to a closed channel is viewed as a non-blocking operation, and this operation causes a panic.

For example, in the following program, when the second case branch gets selected, it will produce a panic at run time.

```

1| package main
2|
3| func main() {
4|     var c = make(chan bool)
5|     close(c)
6|     select {
7|         case <-c:
8|             case c <- true: // panic: send on closed channel
9|         default:
10|     }
11| }
```

Types can be declared within function bodies.

Types can be declared in function bodies. For example,

```

1| package main
2|
3| func main() {
4|     type T struct{}
5|     type S = []int
6| }
```

For the standard compiler, zero-sized fields in a struct may be treated as one-byte-sized value.

Please read [this FAQ item](#) (§51) for details.

NaN != NaN, Inf == Inf.

This follows IEEE-754 standard and is consistent with most other programming languages:

```

1| package main
2|
3| import "fmt"
4| import "math"
5|
6| func main() {
7|     var a = math.Sqrt(-1.0)
8|     fmt.Println(a)      // NaN
9|     fmt.Println(a == a) // false
10|
11|    var x = 0.0
12|    var y = 1.0 / x
13|    var z = 2.0 * y
14|    fmt.Println(y, z, y == z) // +Inf +Inf true
15| }
```

Non-exported method names and struct field names from different packages are viewed as different names.

For example, if the following types are declared in package `foo`:

```

1| package foo
2|
3| type I = interface {
```

```

4|     about() string
5| }
6|
7| type S struct {
8|     a string
9| }
10|
11| func (s S) about() string {
12|     return s.a
13| }
```

and the following types are declared in package bar:

```

1| package bar
2|
3| type I = interface {
4|     about() string
5| }
6|
7| type S struct {
8|     a string
9| }
10|
11| func (s S) about() string {
12|     return s.a
13| }
```

then,

- values of the two respective types `S` from the two packages can't be converted to each other.
- the two respective interface types `S` from the two packages denote two distinct method sets.
- type `foo.S` doesn't implement the interface type `bar.I`.
- type `bar.S` doesn't implement the interface type `foo.I`.

```

1| package main
2|
3| import "包2/foo"
4| import "包2/bar"
5|
6| func main() {
7|     var x foo.S
8|     var y bar.S
9|     var _ foo.I = x
10|    var _ bar.I = y
11| }
```

```

12| // The following lines fail to compile.
13| x = foo.S(y)
14| y = bar.S(x)
15| var _ foo.I = y
16| var _ bar.I = x
17| }
```

In struct value comparisons, blank fields will be ignored.

Blank fields are those fields whose name are the blank identifier `_`. The following program will print `true`.

```

1| package main
2|
3| import "fmt"
4|
5| type T struct {
6|     _ int
7|     _ bool
8| }
9|
10| func main() {
11|     var t1 = T{123, true}
12|     var t2 = T{789, false}
13|     fmt.Println(t1 == t2) // true
14| }
```

Parentheses are required in several rare scenarios to make code compile okay.

For example:

```

1| package main
2|
3| type T struct{x, y int}
4|
5| func main() {
6|     // Each of the following three lines makes code
7|     // fail to compile. Some "{}'s confuse compilers.
8|     /*
```

```

9|     if T{} == T{123, 789} {}
10|    if T{} == (T{123, 789}) {}
11|    if (T{}) == T{123, 789} {}
12|    var _ = func()(nil) // nil is viewed as a type
13|   */
14|
15|   // We must add parentheses like the following
16|   // two lines to make code compile okay.
17|   if (T{} == T{123, 789}) {}
18|   if (T{}) == (T{123, 789}) {}
19|   var _ = (func())(nil) // nil is viewed as a value
20|

```

Stack overflow is not panic.

For the current main stream Go compilers, stack overflows are fatal errors. Once a stack overflow happens, the whole program will crash without recovery ways.

```

1| package main
2|
3| func f() {
4|     f()
5| }
6|
7| func main() {
8|     defer func() {
9|         recover() // helpless to avoid program crashing
10|    }()
11|    f()
12| }

```

the running result:

```

runtime: goroutine stack exceeds 1000000000-byte limit
fatal error: stack overflow

```

```

runtime stack:
...
```

About more crash cases, please read [this wiki article](#).

Some expression evaluation orders in Go are compiler implementation dependent.

Please read [expression evaluation orders in Go](#) (§33) for details.

The results of `reflect.DeepEqual(x, y)` and `x == y` may be different.

The function call `reflect.DeepEqual(x, y)` will always return `false` if the types of its two arguments are different, whereas `x == y` may return `true` even if the types of the two operands are different.

The second difference is a `DeepEqual` call with two pointer argument values of the same type returns whether or not the two respective values referenced by the two pointers are deep equal. So the call might return `true` even if the two pointers are not equal.

The third difference is the result of a `DeepEqual` call might return `true` if both of its arguments are in cyclic reference chains (to avoid infinite looping in the call).

The fourth difference is, the function call `reflect.DeepEqual(x, y)` is not expected to panic generally, whereas `x == y` will panic if the two operands are both interface values and their dynamic types are identical and incomparable.

An example showing these differences:

```

1| package main
2|
3| import (
4|     "fmt"
5|     "reflect"
6| )
7|
8| func main() {
9|     type Book struct {page int}
10|    x := struct {page int}{123}
11|    y := Book{123}
12|    fmt.Println(reflect.DeepEqual(x, y)) // false
13|    fmt.Println(x == y)                // true
14|
15|    z := Book{123}
16|    fmt.Println(reflect.DeepEqual(&z, &y)) // true

```

```

17|     fmt.Println(&z == &y)                      // false
18|
19|     type Node struct{peer *Node}
20|     var q, r, s Node
21|     q.peer = &q // form a cyclic reference chain
22|     r.peer = &s // form a cyclic reference chain
23|     s.peer = &r
24|     println(reflect.DeepEqual(&q, &r)) // true
25|     fmt.Println(q == r)                  // false
26|
27|     var f1, f2 func() = nil, func(){}
28|     fmt.Println(reflect.DeepEqual(f1, f1)) // true
29|     fmt.Println(reflect.DeepEqual(f2, f2)) // false
30|
31|     var a, b interface{} = []int{1, 2}, []int{1, 2}
32|     fmt.Println(reflect.DeepEqual(a, b)) // true
33|     fmt.Println(a == b)                // panic
34| }
```

Note, if the two arguments of a `.DeepEqual` call are both function values, then the call returns `true` only if the two function arguments are both nil and their types are identical. It is similar to compare container values whose elements contain function values or compare struct values whose fields contain function values. But please also note that the result of comparing two slices (of the same type) is always `true` if the two slices exactly share the same elements (in other words, they have the same length and each pair of their corresponding elements have the same address). An example:

```

1| package main
2|
3| import (
4|     "fmt"
5|     "reflect"
6| )
7|
8| func main() {
9|     a := [1]func(){func(){}}
10|    b := a
11|    fmt.Println(reflect.DeepEqual(a, a))      // false
12|    fmt.Println(reflect.DeepEqual(a[:], a[:])) // true
13|    fmt.Println(reflect.DeepEqual(a[:], b[:])) // false
14|    a[0], b[0] = nil, nil
15|    fmt.Println(reflect.DeepEqual(a[:], b[:])) // true
16| }
```

The `reflect.Value.Bytes()` method returns a `[]byte` value, which element type, `byte`, might be not the same as the Go slice value represented by the receiver parameter.

Assume the underlying type of a defined type `MyByte` is the predeclared type `byte`, we know that Go type system forbids the conversions between `[]MyByte` and `[]byte` values. However, it looks the implementation of the method `Bytes` of the `reflect.Value` type partially violates this restriction unintentionally, by allowing converting a `[]MyByte` value to `[]byte`.

Example:

```

1| package main
2|
3| import "bytes"
4| import "fmt"
5| import "reflect"
6|
7| type MyByte byte
8|
9| func main() {
10|     var mybs = []MyByte{'a', 'b', 'c'}
11|     var bs []byte
12|
13|     // bs = []byte(mybs) // this line fails to compile
14|
15|     v := reflect.ValueOf(mybs)
16|     bs = v.Bytes() // okay. Violating Go type system.
17|     fmt.Println(bytes.HasPrefix(bs, []byte{'a', 'b'})) // true
18|
19|     bs[1], bs[2] = 'r', 't'
20|     fmt.Printf("%s\n", mybs) // art
21| }
```

But it looks the violation is not harmful. On the contrary, it makes some benefits. For example, with this violation, we can use the functions in the `bytes` standard package for the `[]MyByte` values.

Note, the `reflect.Value.Bytes()` method [might be removed later](#).

We should use `os.IsNotExist(err)` instead of `err == os.ErrNotExist` to check whether or not a file exists.

Using `err == os.ErrNotExist` may miss errors.

```

1| package main
2|
3| import (
4|     "fmt"
5|     "os"
6| )
7|
8| func main() {
9|     _, err := os.Stat("a-nonexistent-file.abcxyz")
10|    fmt.Println(os.IsNotExist(err)) // true
11|    fmt.Println(err == os.ErrNotExist) // false
12| }
```

For projects only supporting Go 1.13+, `errors.Is(err, os.ErrNotExist)` is [more recommended to be used](#) to check whether or not a file exists.

```

1| package main
2|
3| import (
4|     "errors"
5|     "fmt"
6|     "os"
7| )
8|
9| func main() {
10|     _, err := os.Stat("a-nonexistent-file.abcxyz")
11|     fmt.Println(errors.Is(err, os.ErrNotExist)) // true
12| }
```

The `flag` standard package treats boolean command flags differently than integer and string flags.

There are three forms to pass flag options.

1. `-flag`, for boolean flags only.

2. `-flag=x`, for any flag.
3. `-flag x`, for non-boolean flags only.

And please note that, a boolean flag with the first form is viewed as the last flag, all items following it are viewed as arguments.

```

1| package main
2|
3| import "fmt"
4| import "flag"
5|
6| var b = flag.Bool("b", true, "a boolean flag")
7| var i = flag.Int("i", 123, "an integer flag")
8| var s = flag.String("s", "hi", "a string flag")
9|
10| func main() {
11|     flag.Parse()
12|     fmt.Println("b=", *b, ", i=", *i, ", s=", *s, "\n")
13|     fmt.Println("arguments:", flag.Args())
14| }
```

If we run this program with the below shown flags and arguments

```
./exampleProgram -b false -i 789 -s bye arg0 arg1
```

the output will be

```
b=true, i=123, s=hi
arguments: [false -i 789 -s bye arg0 arg1]
```

This output is obviously not what we expect.

We should pass the flags and arguments like

```
./exampleProgram -b=false -i 789 -s bye arg0 arg1
```

or

```
./exampleProgram -i 789 -s bye -b arg0 arg1
```

to get the output we expect:

```
b=true, i=789, s=bye
arguments: [arg0 arg1]
```

[Sp|Fp|P]rintf functions support positional arguments.

The following program will print coco.

```
1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     // The next line prints: coco
7|     fmt.Printf("%[2]v%[1]v%[2]v%[1]v", "o", "c")
8| }
```

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

Go FAQ 101

(This is an unofficial Go FAQ. The official one is [here ↗](#).)

Index:

- compiler and runtime
 - [What does the compile error message non-name *** on left side of := mean?](#)
 - [What does the compile error message unexpected newline, expecting { after if clause mean?](#)
 - [What does the compiler error message declared and not used mean?](#)
 - [Does Go runtime maintain the iteration orders for maps?](#)
 - [Will Go compilers do padding to ensure field alignments for struct types?](#)
 - [Why does the final field of a zero-sized type in a struct contribute to the size of the struct sometimes?](#)
 - [Is new\(T\) a sugar of var t T; \(&t\)?](#)
 - [What does the runtime error message all goroutines are asleep - deadlock mean?](#)
 - [Are 64-bit integer values guaranteed to be 64-bit aligned so that they can be accessed atomically?](#)
 - [Are assignments of values atomic operations?](#)
 - [Is every zero value composed of a sequence of zero bytes in memory?](#)
 - [Does the standard Go compiler support function inlining?](#)
 - [Can I use finalizers as object destructors?](#)
- standard packages
 - [How to get the number of days of any month by using as few code lines as possible?](#)
 - [What is the difference between the function call `time.Sleep\(d\)` and the channel receive operation `<- time.After\(d\)`?](#)
 - [Calls of the `TrimLeft` and `TrimRight` functions in the `strings` and `bytes` standard packages often return unexpected results, are there bugs in these function implementations?](#)
 - [What are the differences between the `fmt.Print` and `fmt.Println` functions?](#)
 - [Is there any difference between the `log.Print` and `log.Println` functions?](#)
 - [Are `fmt.Print`, `fmt.Println` and `fmt.Printf` functions synchronized?](#)
 - [What are the differences between the built-in `print/println` functions and the corresponding `print` functions in the `fmt` and `log` standard packages?](#)
 - [What is the difference between the random numbers produced by the `math/rand` standard package and the `crypto/rand` standard package?](#)

- [Why isn't there a `math.Round` function?](#)
- type system
 - [Which types don't support comparisons?](#)
 - [Why aren't two `nil` values equal sometimes?](#)
 - [Why don't type `\[\]T1` and `\[\]T2` share the same underlying type even if the two different types `T1` and `T2` share the same underlying type?](#)
 - [Which values can and which values can't have their addresses taken?](#)
 - [Why are map elements not addressable?](#)
 - [Why elements of a non-nil slice are always addressable, even if the slice is unaddressable?](#)
 - [For any non-pointer non-interface type `T`, why is the method set of `*T` always a super set of the method set of `T`, but not vice versa?](#)
 - [Which types can we implement methods for?](#)
 - [How to declare immutable values in Go?](#)
 - [Why isn't there a built-in `set` container type?](#)
 - [What is `byte`? What is `rune`? How to convert `\[\]byte` and `\[\]rune` values to strings?](#)
 - [How to manipulate pointer values atomically?](#)
- others
 - [What does `iota` mean?](#)
 - [Why isn't there a built-in `closed` function to check whether or not a channel is closed?](#)
 - [Is it safe for a function to return pointers of local variables?](#)
 - [What does the word `gopher` mean in Go community?](#)

What does the compile error message `non-name *** on left side of :=` mean?

Up to now (Go 1.21), there is [a mandatory rule](#) for short variable declarations:

All items at the left side of `:=` must be pure [identifiers](#) and at least one of them must be a new variable name.

This means container elements (`x[i]`), struct fields (`x.f`), pointer dereferences (`*p`) and qualified identifiers (`aPackage.Value`) can't appear at the left side of `:=`.

Currently, there is an [open issue](#) (which was merged with [a more related one](#)) for this problem. It looks Go core team wants to [leave this problem unresolved currently](#).

What does the compile error message `unexpected newline, expecting { . . .` mean?

In Go, we can't break a code line at an arbitrary position. Please read [line break rules in Go](#) (§28) for details. By the rules, generally, it is not okay to break code lines just before the open brackets.

For example, the following code

```

1| if true
2| {
3| }
4|
5| for i := 0; i < 10; i++
6| {
7| }
8|
9| var _ = []int
10| {
11|   1, 2, 3
12| }
```

will be interpreted as

```

1| if true;
2| {
3| }
4|
5| for i := 0; i < 10; i++;
6| {
7| }
8|
9| var _ = []int;
10| {
11|   1, 2, 3;
12| }
```

Go compilers will report an error for each open bracket `{`. To avoid these errors, we should rewrite the above code as the following.

```

1| if true {
2| }
3|
4| for i := 0; i < 10; i++ {
5| }
```

```

6|
7| var _ = []int {
8|   1, 2, 3,
9| }
```

What does the compiler error message declared and not used mean?

For the standard Go compiler, each variable declared in local code blocks must be used as a r-value (right-hand-side value) for at least once.

So the following code fails to compile.

```

1| func f(x bool) {
2|   var y = 1 // y declared but not used (as r-values)
3|   if x {
4|     y = 2 // here y is used as a left-hand-side value
5|   }
6| }
```

Does Go runtime maintain the iteration orders for maps?

No. [Go 1 specification ↗](#) says the iteration order over a map is not specified and is not guaranteed to be the same from one iteration to the next. For the standard Go compiler, the map iteration orders are always partially randomized to varying extent. If you require a stable iteration order for a map you must maintain the order by yourself. Please read [Go maps in action ↗](#) for more information.

However, please note, since Go 1.12, the entry order in the print result of the print functions in standard packages are always ordered.

Will Go compilers do padding to ensure field alignments for struct types?

At least for the standard Go compiler and gccgo, the answer is yes. How many bytes will be padded is OS and compiler dependent. Please read [memory layouts](#) (§44) for details.

Go Compilers will not rearrange struct fields to minimize struct value sizes. Doing this may cause some unexpected results. However, programmers can minimize padding by reordering the fields

manually.

Why does the final field of a zero-sized type in a struct contribute to the size of the struct sometimes?

In the current standard Go runtime implementation, as long as a memory block is referenced by at least one active pointer, that memory block will not be viewed as garbage and will not be collected.

All the fields of an addressable struct value can be taken addresses. If the size of the final field in a non-zero-sized struct value is zero, then taking the address of the final field in the struct value will return an address which is beyond the allocated memory block for the struct value. The returned address may point to another allocated memory block which closely follows the one allocated for the non-zero-sized struct value. As long as the returned address is stored in an active pointer value, the other allocated memory block will not get garbage collected, which may cause memory leaking.

To avoid these kinds of memory leak problems, the standard Go compiler will ensure that taking the address of the final field in a non-zero-sized struct will never return an address which is beyond the allocated memory block for the struct. The standard Go compiler implements this by padding some bytes after the final zero-sized field when needed.

If the types of all fields in a struct type are zero-sized (so the struct is also a zero-sized type), then there is no need to pad bytes in the struct, for the standard Go compiler treats zero-sized memory blocks specially.

An example:

```

1| package main
2|
3| import (
4|     "unsafe"
5|     "fmt"
6| )
7|
8| func main() {
9|     type T1 struct {
10|         a struct{}
11|         x int64
12|     }
13|     fmt.Println(unsafe.Sizeof(T1{})) // 8
14|
15|     type T2 struct {
16|         x int64

```

```

17|     a struct{}}
18|   }
19|   fmt.Println(unsafe.Sizeof(T2{})) // 16
20| }
```

Is new(T) a sugar of var t T; (&t)?

Generally we can think so, though there may be some subtle differences between the two, depending on compiler implementations. The memory block allocated by `new` may be either on stack or on heap.

What does the runtime error message all goroutines are asleep - deadlock mean?

The word *asleep* is not accurate here, it means *in blocking state* in fact.

As a blocking goroutine can only be unblocked by another goroutine, if all goroutines in a program enter blocking state, then all of them will stay in blocking state for ever. This means the program is deadlocked. A normal running program is never expected to be deadlocked, so the standard Go runtime makes the program crash and exit.

Are 64-bit integer values guaranteed to be 64-bit aligned so that they can be accessed atomically?

The addresses passed to the 64-bit functions in `sync/atomic` package must be 64-bit aligned, otherwise, calls to these functions may panic at run time.

For the standard Go compiler and `gccgo` compiler, on 64-bit architectures, 64-bit integers are guaranteed to be 64-bit aligned. So they can be always accessed atomically without any problems.

On 32-bit architectures, 64-bit integers are only guaranteed to be 32-bit aligned. So accessing many 64-bit integers atomically may cause panics. However, there are some ways to guarantee some 64-bit integers to be relied upon to be 64-bit aligned. Please read [memory layouts in Go](#) (§44) for details.

Are assignments of values atomic operations?

No for the standard Go compiler, even if the sizes of the assigned values are native words.

Please read [the official question ↗](#) for more details.

Is every zero value composed of a sequence of zero bytes in memory?

For most types, this is true. In fact, this is compiler dependent. For example, for the standard Go compiler, the statement is wrong for some zero values of string types.

Evidence:

```

1| package main
2|
3| import (
4|     "unsafe"
5|     "fmt"
6| )
7|
8| func main() {
9|     var s1 string
10|    fmt.Println(s1 == "") // true
11|    fmt.Println(*(*uintptr)(unsafe.Pointer(&s1))) // 0
12|    var s2 = "abc"[0:0]
13|    fmt.Println(s2 == "") // true
14|    fmt.Println(*(*uintptr)(unsafe.Pointer(&s2))) // 4869856
15|    fmt.Println(s1 == s2) // true
16| }
```

Inversely, for all the architectures the standard Go compiler currently supports, if all bytes in a value are zero, then the value must be a zero value of its type. However, Go specification doesn't guarantee this. I have heard of that on some very old processors, nil pointers are not zero in memory.

Does the standard Go compiler support function inlining?

Yes, the standard Go compiler supports inlining functions. The compiler will automatically inline some very short functions which satisfy certain requirements. The specific inline requirements may change from version to version.

Currently (Go Toolchain 1.21), for the standard Go compiler:

- There are no explicit ways to specify which functions should be inlined in user programs.
- The `-gcflags "-l"` build option disables inlining globally, which will prevent all functions from being inline expanded.
- In user programs, there are not formal ways to prevent inlining of particular functions. You can add a line `//go:noinline` directive before a function declaration to prevent the function from being inlined, but this way is not guaranteed to work in the future.

Can I use finalizers as object destructors?

In Go programs, we can set a finalizer function for an object by using the `runtime.SetFinalizer` function. Generally, the finalizer function will be called before the object is garbage collected. But finalizers are never intended to be used as destructors of objects. The finalizers set by `runtime.SetFinalizer` are not guaranteed to run. So you shouldn't rely on finalizers for your program correctness.

The main intention of finalizers is to allow the maintainers of a library to compensate for problems caused by incorrect library use. For example, if a programmer uses `os.Open` to open many files but forgets to close them after using them, then the program will hold many file descriptors until the program exits. This is a classic example of resource leak. To avoid the program holding too many file descriptors, the maintainers of the `os` package will set a finalizer on the every created `os.File` object. The finalizer will close the file descriptor stored in the `os.File` object. As mentioned above, the finalizers are not guaranteed to be called. They are just used to make the extent of resource leak as small as possible.

Please note, some finalizers will never get called for sure, and sometimes setting finalizers improperly will prevent some objects from being garbage collected. Please read the [runtime.SetFinalizer function documentation](#) to get more details.

How to get the number of days of any month in as few lines as possible?

Assume the input year and month are from the Gregorian Calendar (January is 1).

```
days := time.Date(year, month+1, 0, 0, 0, 0, 0, time.UTC).Day()
```

For Go time APIs, the usual month range is `[1, 12]` and the start day of each month is 1. The start time of a month `m` in year `y` is `time.Date(y, m, 1, 0, 0, 0, 0, time.UTC)`.

The arguments passed to `time.Date` can be outside their usual ranges and will be normalized during the conversion. For example, January 32 will be converted to February 1.

Here are some `time.Date` use examples in Go:

```

1| package main
2|
3| import (
4|     "time"
5|     "fmt"
6| )
7|
8| func main() {
9|     // 2017-02-01 00:00:00 +0000 UTC
10|    fmt.Println(time.Date(2017, 1, 32, 0, 0, 0, 0, time.UTC))
11|
12|    // 2017-01-31 23:59:59.999999999 +0000 UTC
13|    fmt.Println(time.Date(2017, 1, 32, 0, 0, 0, -1, time.UTC))
14|
15|    // 2017-01-31 00:00:00 +0000 UTC
16|    fmt.Println(time.Date(2017, 2, 0, 0, 0, 0, 0, time.UTC))
17|
18|    // 2016-12-31 00:00:00 +0000 UTC
19|    fmt.Println(time.Date(2016, 13, 0, 0, 0, 0, 0, time.UTC))
20|
21|    // 2017-02-01 00:00:00 +0000 UTC
22|    fmt.Println(time.Date(2016, 13, 32, 0, 0, 0, 0, time.UTC))
23| }
```

What is the difference between the function call `time.Sleep(d)` and the channel receive operation `<- time.After(d)`?

The two will both pause the current goroutine execution for a certain duration. The difference is the function call `time.Sleep(d)` will let the current goroutine enter sleeping sub-state, but still stay in [running state](#) (§13), whereas, the channel receive operation `<- time.After(d)` will let the current goroutine enter blocking state.

Calls of the `TrimLeft` and `TrimRight` functions in the `strings` and `bytes` standard packages often return

unexpected results, are there bugs in these function implementations?

Aha, maybe there are bugs in the implementations, but none are confirmed now. If the return results are unexpected, it is more possible that your expectations are not correct.

There are many trim functions in `strings` and `bytes` standard packages. These functions can be categorized into two groups:

1. `Trim`, `TrimLeft`, `TrimRight`, `TrimSpace`, `TrimFunc`, `TrimLeftFunc`, `TrimRightFunc`. These functions will trim all leading or trailing UTF-8-encoded Unicode code points (a.k.a. runes) which satisfy the specified or implied conditions (`TrimSpace` implies to trim all kinds of white spaces). Each of the leading or trailing runes will be checked until one doesn't satisfy the specified or implied conditions.
2. `TrimPrefix`, `TrimSuffix`. The two functions will trim the specified prefix or suffix substrings (or subslices) as a whole.

[Some](#) [programmers](#) [misused](#) [the](#) [TrimLeft](#) and [TrimRight](#) functions as `TrimPrefix` and `TrimSuffix` functions when they use the trim functions the first time. Certainly, the return results are very possible not as expected.

Example:

```

1| package main
2|
3| import (
4|     "fmt"
5|     "strings"
6| )
7|
8| func main() {
9|     var s = "abaay森z众xbbab"
10|    o := fmt.Println
11|    o(strings.TrimPrefix(s, "ab")) // aay森z众xbbab
12|    o(strings.TrimSuffix(s, "ab")) // abaay森z众xbb
13|    o(strings.TrimLeft(s, "ab"))   // y森z众xbbab
14|    o(strings.TrimRight(s, "ab")) // abaay森z众x
15|    o(strings.Trim(s, "ab"))      // y森z众x
16|    o(strings.TrimFunc(s, func(r rune) bool {
17|         return r < 128 // trim all ascii chars
18|     })) // 森z众
19| }
```

What are the differences between the `fmt.Print` and `fmt.Println` functions?

The `fmt.Println` function will always write a space between two adjacent arguments, whereas the `fmt.Print` function will write a space between two adjacent arguments only if both of (the concrete values of) the two adjacent arguments are not strings.

Another difference is that `fmt.Println` will write a newline character in the end, but the `fmt.Print` function will not.

Is there any difference between the `log.Print` and `log.Println` functions?

The difference between the `log.Print` and `log.Println` functions is the same as the first difference between the `fmt.Print` and `fmt.Println` functions described in the last question.

Both of the two functions will write a newline character in the end.

Are `fmt.Print`, `fmt.Println` and `fmt.Printf` functions synchronized?

No, these functions are not synchronized. Please use the corresponding functions in the `log` standard package instead when synchronizations are needed. You can call `log.SetFlags(0)` to remove the prefix from each log line.

What are the differences between the built-in `print/println` functions and the corresponding `print` functions in the `fmt` and `log` standard packages?

Besides the difference mentioned in the last question, there are some other differences between the three sets of functions.

1. The built-in `print/println` functions will write to standard error. The `print` functions in the `fmt` standard package will write to standard output. The `print` functions in the `log` standard

package will write to standard error by default, though this can be changed using the `log.SetOutput` function.

2. Calls to the built-in `print/println` functions can't take array and struct arguments.
3. For an argument of a composite type, the built-in `print/println` functions write the addresses of the underlying value parts of the argument, whereas the `print` functions in the `fmt` and `log` standard packages try to write the value literal of the dynamic values of the interface arguments.
4. Calls to the built-in `print/println` functions will not make the values referenced by the arguments of the calls escape to heap, whereas the `print` functions in the `fmt` and `log` standard packages will.
5. If an argument has a `String()` `string` or `Error()` `string` method, the `print` functions in the `fmt` and `log` standard packages will try to call that method when writing the argument, whereas the built-in `print/println` functions will ignore methods of arguments.
6. The built-in `print/println` functions are not guaranteed to exist in future Go versions.

What is the difference between the random numbers produced by the `math/rand` standard package and the `crypto/rand` standard package?

The pseudo random numbers produced by the `math/rand` standard package are deterministic for a given seed. The produced random numbers are not good for security-sensitive contexts. For cryptographical security purpose, we should use the pseudo random numbers produced by the `crypto/rand` standard package.

Why isn't there a `math.Round` function?

There is a `math.Round` function, but only since Go 1.10. Two new functions, `math.Round` and `math.RoundToEven` have been added since Go 1.10.

Before Go 1.10, much time and effort was spent [discussing ↗](#) whether or not the `math.Round` function should be added to standard package or not. In the end, the proposal was adopted.

Which types don't support comparisons?

Following types don't support comparisons:

- map
- slice
- function
- struct types containing incomparable fields
- array types with incomparable element types

Types which don't support comparisons can't be used as the key type of map types.

Please note,

- although map, slice and function types don't support comparisons, their values can be compared to the bare `nil` identifier.
- [comparing two interface values](#) (§23) with `==` will panic at run time if the two dynamic types of the two interface values are identical and incomparable.

On why slice, map and function types don't support comparison, please read [this answer ↗](#) in the official Go FAQ.

Why aren't two `nil` values equal sometimes?

([The answer ↗](#) in the official Go FAQ may also answer this question.)

An interface value can be viewed as a box which is used to encapsulate non-interface values. Only values whose types implement the type of the interface value can be boxed (encapsulated) into the interface value. In Go, there are several kinds of types whose zero values are represented as the predeclared identifier `nil`. An interface value boxing nothing is a zero interface value, a.k.a, a nil interface value. However an interface value boxing a nil non-interface value doesn't box nothing, so it is not, and doesn't equal to, a nil interface value.

When comparing a nil interface value and a nil non-interface value (assume they can be compared), the nil non-interface value will be converted to the type of the nil interface value before doing the comparison. The conversion result is an interface value boxing a copy of the non-interface value. The result interface value doesn't box nothing, so it is not, or doesn't equal to, the nil interface value.

Please read [interfaces in Go](#) (§23) and [nils in Go](#) (§47) for detailed explanations.

For example:

```
1| package main
2|
3| import "fmt"
4|
```

```

5| func main() {
6|     var pi *int = nil
7|     var pb *bool = nil
8|     var x interface{} = pi
9|     var y interface{} = pb
10|    var z interface{} = nil
11|
12|    fmt.Println(x == y) // false
13|    fmt.Println(x == nil) // false
14|    fmt.Println(y == nil) // false
15|    fmt.Println(x == z) // false
16|    fmt.Println(y == z) // false
17| }

```

Why don't type $[]T_1$ and $[]T_2$ share the same underlying type even if the two different types T_1 and T_2 share the same underlying type?

(It looks the official Go FAQ also added [a similar question ↗](#) not long ago.)

In Go, values of a slice type can be converted to another slice type without using [the unsafe mechanisms](#) (§25) only if the two slice types share the same [underlying type](#) (§14). ([This article](#) (§48) lists the full list of value conversion rules.)

The underlying type of an unnamed composite type is the composite type itself. So even if two different types T_1 and T_2 share the same underlying type, type $[]T_1$ and $[]T_2$ are still different types, so their underlying types are also different, which means values of one of them can't be converted to the other.

The reasons for the underlying types of $[]T_1$ and $[]T_2$ are not same are:

- the request of converting values of $[]T_1$ and $[]T_2$ to each other is not strong in practice.
- to make the [underlying type tracing rules](#) (§14) simpler.

The same reasons are also valid for other composite types. For example, type $map[T]T_1$ and $map[T]T_2$ also don't share the same underlying type even if T_1 and T_2 share the same underlying type.

It is possible that values of type $[]T_1$ can be converted to $[]T_2$ by using the `unsafe` mechanisms, but generally this is not recommended:

```

1| package main
2|
3| import (
4|     "fmt"
5|     "unsafe"
6| )
7|
8| func main() {
9|     type MyInt int
10|
11|     var a = []int{7, 8, 9}
12|     var b = *(*[]MyInt)(unsafe.Pointer(&a))
13|     b[0] = 123
14|     fmt.Println(a) // [123 8 9]
15|     fmt.Println(b) // [123 8 9]
16|     fmt.Printf("%T\n", a) // []int
17|     fmt.Printf("%T\n", b) // []main.MyInt
18| }

```

Which values can and which values can't have their addresses taken?

We can't take the address of the following values:

- bytes in strings
- map elements
- dynamic values of interface values (exposed by type assertions)
- constant values (including named constants and literals)
- package level functions
- methods (used as function values)
- intermediate values
 - function calls
 - explicit value conversions
 - all sorts of operations, excluding pointer dereference operations, but including:
 - channel receive operations
 - sub-string operations
 - sub-slice operations
 - addition, subtraction, multiplication, and division, etc.

Please note, there is a syntax sugar, `&T{}`, in Go. It is a short form of `tmp := T{};`
`(&tmp)`. However, though `&T{}` is legal, the literal `T{}` is still not addressable.

Following values can have their addresses taken:

- variables
- fields of addressable structs
- elements of addressable arrays
- elements of any slices (whether the slices are addressable or not)
- pointer dereference operations

Why are map elements unaddressable?

The first reason is that it would conflict with maps' internal memory management process. In Go, a map is designed as a container which can contain an unlimited number of entries if memory is available. To ensure good map element indexing efficiency, in the official Go runtime implementation each map value only maintains one continuous memory segment for the entirety of entries stored in that map. Therefore, Go runtime needs to allocate larger memory segments for a map from time to time when more and more entries are being put into the map. In the process, the entries stored on older memory segments will be moved to newer memory segments. There might be also some other reasons causing memory movements of entries. In other words, the addresses of map elements will change from time to time on need. If map elements are allowed to have their addresses taken, then when some map entries are moved, Go runtime would have to update all pointers which are storing the addresses of the moved elements, which brings many difficulties in implementing Go compilers and runtimes and greatly decreases execution performance. So, currently, map elements cannot have their addresses taken.

Secondly, the map index expression `aMap[key]` might return an element stored in map `aMap` or not, which means `aMap[key]` might still result in a zero value after `(&aMap[key]).Modify()` is called. This would confuse many people. (Here `Modify()` refers to a hypothetical method which would modify the value `aMap[key]`).

Why elements of a non-nil slice are always addressable, even if the slice is unaddressable?

The internal type for slices is a struct like

```
1| struct {
2|     // elements references an element sequence.
3|     elements unsafe.Pointer
4|     length    int
```

```

5|     capacity int
6| }
```

Each slice indirectly references an underlying element sequence internally. Although a non-nil slice is not addressable, its internal element sequence is always allocated somewhere and must be addressable. Taking addresses of elements of a slice is taking the addresses of elements of the internal element sequence actually. This is why elements of unaddressable non-nil slices are always addressable.

For any non-pointer non-interface type T , why is the method set of $*T$ always a super set of the method set of T , but not vice versa?

Both of these situations involve sugaring, but only one is an intrinsic rule.

- A value of type T can call methods of type $*T$, but only if the value of T is addressable. Compilers will take the address of the T value automatically before calling the pointer receiver methods. Because type T can have values that are not addressable, not all values of type T are capable of calling methods of type $*T$.

This convenience is just a sugar, not an intrinsic rule.

- A value of type $*T$ can always call methods of type T . This is because it is always legal to dereference a pointer value.

This convenience is not only a sugar, but also an intrinsic rule.

So, it makes sense that the method set of $*T$ is always a super set of the method set of T , but not vice versa.

In fact, you can think that, for every method declared on type T , an implicit method with the same name and the same signature is automatically declared on type $*T$. Please read [methods](#) (§22) for details.

```

1| func (t T) MethodX(v0 ParamType0, ...) (ResultType0, ...) {
2|     ...
3| }
4|
5| // An implicit method of *T is automatically defined as
6| func (pt *T) MethodX(v0 ParamType0, ...) (ResultType0, ...) {
```

```
7|     return (*pt).MethodX(v0, ...)  
8| }
```

Please read [this answer ↗](#) in the official Go FAQ to get more explanations.

Which types can we implement methods for?

Please read [methods in Go](#) (§22) for details.

How to declare immutable values in Go?

There are three ***immutable value*** definitions:

1. the values which have no addresses (so they are not addressable).
2. the values which have addresses but are not addressable (their addresses are not allowed to be taken in syntax).
3. the values which are addressable but their values are not allowed to be modified in syntax.

In Go, up to now (Go 1.21), there are no values satisfy the third definition. In other words, the third definition is not supported. However, variables of zero-size types may be viewed de facto immutable (addressable) values.

Name constant values satisfy the first definition.

Methods and package-level functions can also viewed as declared immutable values. They satisfy the second definition. String elements (bytes) and map entry elements also satisfy the second definition.

There are no ways to declare other custom immutable named values in Go.

Why isn't there a built-in set container type?

Sets are just maps but don't care about element values. In Go, `map[Tkey]struct{}` is often used as a set type.

What is byte? What is rune? How to convert []byte and []rune values to strings?

In Go, `byte` is an alias of type `uint8`. In other words, `byte` and `uint8` are the same identical type. The same relation is for `rune` and `int32`.

A `rune` often is used to store a Unicode code point.

`[]byte` and `[]rune` values can be explicitly and directly converted to strings, and vice versa.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     var s0 = "Go"
7|
8|     var bs = []byte(s0)
9|     var s1 = string(bs)
10|
11|    var rs = []rune(s0)
12|    var s2 = string(rs)
13|
14|    fmt.Println(s0 == s1) // true
15|    fmt.Println(s0 == s2) // true
16| }
```

About more on strings, please read [strings in Go](#) (§19).

How to manipulate pointer values atomically?

Please read [atomic operations for pointers](#) (§40).

What does `iota` mean?

Iota is the ninth letter of the Greek alphabet. In Go, `iota` is used in constant declarations. In each constant declaration group, its value is `N` in the `N`th constant specification in that constant declaration group. This allows for easy [declaration of related constants](#) ↴.

Why isn't there a built-in closed function to check whether or not a channel is closed?

The reason is that the usefulness of such function would be very limited, while the potential for misuse is high. The return result of a call to such function may be not able to reflect the latest status of the input channel argument. So, it is not a good idea to make decisions relying on the return result.

If you do need such a function, it would be effortless to write one by yourself. Please read [this article](#) (§38) to get how to write `closed` functions and how to avoid using such a function.

Is it safe for a function to return pointers of local variables?

Yes, it is absolutely safe in Go.

Go compilers which support stack will do escape analysis. For the standard Go compiler, if the escape analyzer thinks a memory block will only be used in the current goroutine for sure, it will allocate the memory block on stack. If not, the memory block will be allocated on the heap. Please read [memory block](#) (§43) for more information.

What does the word *gopher* mean in Go community?

In the Go community, a *gopher* means a Go programmer. This nickname may originate from the fact that Go language adopted [a cartoon gopher](#) ↗ as the mascot. BTW, the cartoon gopher is designed by *Renee French*, who is the wife of the (first) Go project leader, *Rob Pike*.

(The **Go 101** book is still being improved frequently from time to time. Please visit [go101.org](#) ↗ or follow [@go100and1](#) ↗ to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit [tapirgames.com](#) ↗ to get more information about these games. Hope you enjoy them.)

Go Tips 101

Index

- [How to force package users to use struct composite literals with field names?](#)
- [How to make a struct type incomparable?](#)
- [Don't use value assignments with expressions interacting with each other.](#)
- [How to simulate `for i in 0..N` in some other languages?](#)
- [We should reset the pointers in the element slots which are freed up in all kinds of slice manipulations to avoid memory leaking if we can't make sure if the freed-up element slots will be reused later.](#)
- [Values of some types in standard packages are not expected to be copied.](#)
- [We can use the memclr optimization to reset some contiguous elements in an array or slice.](#)
- [How to check if a value has a method without importing the `reflect` package?](#)
- [How to efficiently and perfectly clone a slice?](#)
- [We should use the three-index subslice form at some scenarios.](#)
- [Use anonymous functions to make some deferred function calls be executed earlier.](#)
- [Make sure and show a custom defined type implements a specified interface type.](#)
- [Some compile-time assertion tricks.](#)
- [How to declare maximum int and uint constants?](#)
- [How to detect native word size at compile time?](#)
- [How to guarantee that the 64-bit value operated by a 64-bit atomic function call is always 64-bit aligned on 32-bit architectures?](#)
- [Avoid boxing large-size values into interface values.](#)
- [Make optimizations by using BCE \(bounds check elimination\).](#)

How to force package users to use struct composite literals with field names?

Package developers can put a non-exported zero-size field in a struct type definition, so that compilers will forbid package users using composite literals with some field items but without field names to create values of the struct type.

An example:

```

1| // foo.go
2| package foo
3|
4| type Config struct {
```

```

5|     _ [0]int
6|     Name string
7|     Size int
8| }
```

```

1| // main.go
2| package main
3|
4| import "foo"
5|
6| func main() {
7|     //_ = foo.Config{[0]int{}, "bar", 123} // error
8|     _ = foo.Config{Name: "bar", Size: 123} // compile ok
9| }
```

Please try not to place the zero-size non-exported field as the last field in the struct, for [doing so might enlarge the size of the struct type](#) (§51).

How to make a struct type incomparable?

Sometimes, we want to avoid a custom struct type being used a map key types, then we can put a field of a non-exported zero-size incomparable type in a struct type to make the struct type incomparable. For example:

```

1| package main
2|
3| type T struct {
4|     dummy [0]func()
5|     AnotherField int
6| }
7|
8| var x map[T]int // compile error: invalid map key type T
9|
10| func main() {
11|     var a, b T
12|     _ = a == b // compile error: invalid operation:
13| }
```

Don't use value assignments with expressions interacting with each other.

Currently (Go 1.21), there are [some evaluation orders in a multi-value assignment are unspecified](#) ↗ when the expressions involved in the multi-value assignment interact with each other. So try to split a multi-value assignment into multiple single value assignments if there are, or you can't make sure whether or not there are, dependencies between the involved expressions.

In fact, in some bad-written single-value assignments, there are also expression evaluation order ambiguities. For example, the following program might print [7 0 9], [0 8 9], or [7 8 9], depending on compiler implementations.

```

1| package main
2|
3| import "fmt"
4|
5| var a = &[]int{1, 2, 3}
6| var i int
7| func f() int {
8|     i = 1
9|     a = &[]int{7, 8, 9}
10|    return 0
11| }
12|
13| func main() {
14|     // The evaluation order of "a", "i"
15|     // and "f()" is unspecified.
16|     (*a)[i] = f()
17|     fmt.Println(*a)
18| }
```

In other words, a function call in a value assignment may the evaluation results of the non-function-call expressions in the same assignment. Please read [evaluation orders in Go](#) (§33) for details.

How to simulate for i in 0..N in some other languages?

We can range over an array with zero-size element or a nil array pointer to simulate such a loop. For example:

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
```

```

6|     const N = 5
7|
8|     for i := range [N]struct{}{} {
9|         fmt.Println(i)
10|    }
11|    for i := range [N][0]int{} {
12|        fmt.Println(i)
13|    }
14|    for i := range (*[N]int)(nil) {
15|        fmt.Println(i)
16|    }
17| }
```

We should reset the pointers in the element slots which are freed up in all kinds of slice manipulations to avoid memory leaking if we can't make sure if the freed-up element slots will be reused later.

Please read how to [delete slice elements](#) (§18) and [kind-of memory leaking caused by not resetting pointers in dead slice elements](#) (§45) for details.

Values of some types in standard packages are not expected to be copied.

Values of the `bytes.Buffer` type, `strings.Builder` type and the types in the `sync` standard package are not recommended to be copied. (They really should not be copied, though it is no problems to copy them under some specified circumstances.)

The implementation of `strings.Builder` will detect invalid `strings.Builder` value copies. Once such a copy is found, panic will occur. For example:

```

1| package main
2|
3| import "strings"
4|
5| func main() {
6|     var b strings.Builder
7|     b.WriteString("hello ")
8|     var b2 = b
```

```

9|     b2.WriteString("world!") // panic here
10| }
```

Copying values of the types in the `sync` standard package will be warned by the `go vet` command provided in Go Toolchain.

```

1| // demo.go
2| package demo
3|
4| import "sync"
5|
6| func f(m sync.Mutex) { // warning
7|     m.Lock()
8|     defer m.Unlock()
9|     // do something ...
10| }
```

```
$ go vet demo.go
./demo.go:5: f passes lock by value: sync.Mutex
```

Copying `bytes.Buffer` values will never be detected at run time nor by the `go vet` command. Just be careful not to do this.

We can use the `memclr` optimization to reset some contiguous elements in an array or slice.

Please read [the `memclr` optimization](#) (§18) for details.

How to check if a value has a method without importing the `reflect` package?

Use the way in the following example. (Assume the method needed to be checked is `M(int)` `string`.)

```

1| package main
2|
3| import "fmt"
4|
5| type A int
6| type B int
7| func (b B) M(x int) string {
```

```

8|     return fmt.Sprint(b, ":", x)
9|
10|
11| func check(v interface{}) bool {
12|     _, has := v.(interface{M(int) string})
13|     return has
14|
15|
16| func main() {
17|     var a A = 123
18|     var b B = 789
19|     fmt.Println(check(a)) // false
20|     fmt.Println(check(b)) // true
21| }
```

How to efficiently and perfectly clone a slice?

Please read [this wiki article ↗](#) and [this wiki article ↗](#) for details.

We should use the three-index subslice form at some scenarios.

Assume a package provides a `func NewX(...Option) *X` function, and the implementation of this function will merge the input options with some internal default options, then the following implementation is not recommended.

```

1| func NewX(opts ...Option) *X {
2|     options := append(opts, defaultOpts...)
3|     // Use the merged options to build and return a X.
4|     //
5| }
```

The reason why the above implementation is not recommended is the `append` call may modify the underlying `Option` sequence of the argument `opts`. For most scenarios, it is not a problem. But for some special scenarios, it may cause some unexpected results.

To avoid modifying the underlying `Option` sequence of the input argument, we should use the following way instead.

```

1| func NewX(opts ...Option) *X {
2|     opts = append(opts[:len(opts):len(opts)], defaultOpts...)
```

```

3|     // Use the merged options to build and return a X.
4|     ...
5|

```

On the other hand, for the callers of the `NewX` function, it is not a good idea to think and rely on the `NewX` function will not modify the underlying elements of the passed slice arguments, so it is best to pass these arguments with the three-index subslice form.

Another scenario at which we should use three-index subslice form is mentioned in [this wiki article ↗](#).

One drawback of three-index subslice forms is they are some verbose. In fact, I ever made [a proposal ↗](#) to make it less verbose, but it was declined.

Use anonymous functions to make some deferred function calls be executed earlier.

Please read [this article](#) (§29) for details.

Make sure and show a custom defined type implements a specified interface type.

We can assign a value of the custom defined type to a variable of type of the specified interface type to make sure the custom type implements the specified interface type, and more importantly, to show the custom type is intended to implement which interface types. Sometimes, writing docs in runnable code is much better than in comments.

```

1| package myreader
2|
3| import "io"
4|
5| type MyReader uint16
6|
7| func NewMyReader() *MyReader {
8|     var mr MyReader
9|     return &mr
10| }
11|
12| func (mr *MyReader) Read(data []byte) (int, error) {
13|     switch len(data) {
14|         default:

```

```

15|     *mr = MyReader(data[0]) << 8 | MyReader(data[1])
16|     return 2, nil
17| case 2:
18|     *mr = MyReader(data[0]) << 8 | MyReader(data[1])
19| case 1:
20|     *mr = MyReader(data[0])
21| case 0:
22| }
23| return len(data), io.EOF
24|
25|
26| // Any of the following three lines ensures
27| // type *MyReader implements io.Reader.
28| var _ io.Reader = NewMyReader()
29| var _ io.Reader = (*MyReader)(nil)
30| func _() {_ = io.Reader(nil).(*MyReader)}

```

Some compile-time assertion tricks.

Besides the above one, there are more compile-time assertion tricks.

Several ways to guarantee a constant N is not smaller than another constant M at compile time:

```

1| // Any of the following lines can guarantee N >= M
2| func _(x []int) {_ = x[N-M]}
3| func _(){_ = []int{N-M: 0}}
4| func _([N-M]int){}
5| var _ [N-M]int
6| const _ uint = N-M
7| type _ [N-M]int
8|
9| // If M and N are guaranteed to be positive integers.
10| var _ uint = N/M - 1

```

One more way which is stolen from [@lukechampine ↗](#). It makes use of the rule that [duplicate constant keys can't appear in the same composite literal](#) (§18).

```
var _ = map[bool]struct{}{false: struct{}{}, N>=M: struct{}{}}
```

The above way looks some verbose but it is more general. It can be used to assert any conditions. It can be less verbose but needs a little more (negligible) memory:

```
var _ = map[bool]int{false: 0, N>=M: 1}
```

Similarly, ways to assert two integer constants are equal to each other:

```

1| var _ [N-M]int; var _ [M-N]int
2| type _ [N-M]int; type _ [M-N]int
3| const _, _ uint = N-M, M-N
4| func _([N-M]int, [M-N]int) {}
5|
6| var _ = map[bool]int{false: 0, M==N: 1}
7|
8| var _ = [1]int{M-N: 0} // the only valid index is 0
9| var _ = [1]int{}[M-N] // the only valid index is 0
10|
11| var _ [N-M]int = [M-N]int{}

```

The last line is also inspired by one of Luke Champine's tweets.

Ways of how to assert a constant string is not blank:

```

1| type _ [len(aStringConstant)-1]int
2| var _ = map[bool]int{false: 0, aStringConstant != "": 1}
3| var _ = aStringConstant[:1]
4| var _ = aStringConstant[0]
5| const _ = 1/len(aStringConstant)

```

The last line is stolen from Jan Mercl's [clever idea](#).

Sometimes, to avoid package-level variables consuming too much memory, we can put assertion code in a function declared with the blank identifier. For example,

```

1| func _() {
2|     var _ = map[bool]int{false: 0, N>=M: 1}
3|     var _ [N-M]int
4| }

```

How to declare maximum int and uint constants?

```

1| const MaxUint = ^uint(0)
2| const MaxInt = int(^uint(0) >> 1)

```

How to detect native word size at compile time?

This tip is Go unrelated.

```

1| const Is64bitArch = ^uint(0) >> 63 == 1
2| const Is32bitArch = ^uint(0) >> 63 == 0
3| const WordBits = 32 << (^uint(0) >> 63) // 64 or 32

```

How to guarantee that the 64-bit value operated by a 64-bit atomic function call is always 64-bit aligned on 32-bit architectures?

Please read [Go value memory layouts](#) (§44) for details.

Avoid boxing large-size values into interface values.

When a non-interface value is assigned to an interface value, a copy of the non-interface value will be boxed into the interface value. The copy cost depends on the size of the non-interface value. The larger the size, the higher the copy cost. So please try to avoid boxing large-size values into interface values.

In the following example, the costs of the latter two print calls are much lower than the former two.

```

1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     var a [1000]int
7|
8|     // This cost of the two lines is high.
9|     fmt.Println(a)           // a is copied
10|    fmt.Printf("Type of a: %T\n", a) // a is copied
11|
12|    // The cost of the two lines is low.
13|    fmt.Printf("%v\n", a[:])
14|    fmt.Println("Type of a:", fmt.Sprintf("%T", &a)[1:])
15| }

```

About value sizes of different types, please read [value copy costs in Go](#) (§34).

Optimize Go code by making use of BCE (bounds check elimination).

Please read [this article](#) (§35) to get what is BCE and how well BCE is supported by the standard Go compiler now.

Here, another example is provided:

```
1| package main
2|
3| import (
4|     "strings"
5|     "testing"
6| )
7|
8| func NumSameBytes_1(x, y string) int {
9|     if len(x) > len(y) {
10|         x, y = y, x
11|     }
12|     for i := 0; i < len(x); i++ {
13|         if x[i] != y[i] {
14|             return i
15|         }
16|     }
17|     return len(x)
18| }
19|
20| func NumSameBytes_2(x, y string) int {
21|     if len(x) > len(y) {
22|         x, y = y, x
23|     }
24|     if len(x) <= len(y) { // more code but more efficient
25|         for i := 0; i < len(x); i++ {
26|             if x[i] != y[i] { // bound check eliminated
27|                 return i
28|             }
29|         }
30|     }
31|     return len(x)
32| }
33|
34| var x = strings.Repeat("hello", 100) + " world!"
35| var y = strings.Repeat("hello", 99) + " world!"
36|
37| func BenchmarkNumSameBytes_1(b *testing.B) {
38|     for i := 0; i < b.N; i++ {
39|         _ = NumSameBytes_1(x, y)
40|     }
41| }
```

```

41| }
42|
43| func BenchmarkNumSameBytes_2(b *testing.B) {
44|     for i := 0; i < b.N; i++ {
45|         _ = NumSameBytes_2(x, y)
46|     }
47| }
```

In the above example, function `NumSameBytes_2` is more efficient than function `NumSameBytes_1`. The benchmark result:

<code>BenchmarkNumSameBytes_1-4</code>	<code>100000000</code>	<code>669 ns/op</code>
<code>BenchmarkNumSameBytes_2-4</code>	<code>200000000</code>	<code>450 ns/op</code>

Please note, there are many small improvements in each main release of the standard Go compiler (gc). The trick used in the above example doesn't work for Go Toolchain versions earlier than 1.11. And future gc versions may become smarter so that the trick will become unnecessary. In fact, since gc v1.11, the bounds check for `y[i]` [has been eliminated](#) if `x` and `y` are slices, even if the above trick is not used.

(The **Go 101** book is still being improved frequently from time to time. Please visit go101.org or follow [@go100and1](https://twitter.com/go100and1) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit tapirgames.com to get more information about these games. Hope you enjoy them.)

More Go Related Topics

Go 101 articles mainly focus on syntax and semantics in Go. There are some other Go related topics which are not covered in Go 101. The remaining of the current article will make simple introductions to those topics and provide some web links for readers to dig more into them.

Profiling, Benchmarks and Unit/Fuzzing Tests

We can use `go test` command in Go Toolchain to run tests and benchmarks. Test source file names must end with `_test.go`. Go Toolchain also supports profiling Go programs. Please read the following articles for more details.

- [Profiling Go programs](#).
- [The testing standard package](#).
- [Write testable examples](#).
- [Using subtests and sub-benchmarks](#).
- [go test command options](#).
- [Go Fuzzing](#).

gccgo

[gccgo](#) is another Go compiler maintained by the Go core team. It is mainly used to verify the correctness of the standard Go compiler (gc). We can use the `-compiler=gccgo` build option in several Go Toolchain commands to use the gccgo compiler instead of the gc compiler. For example, `go run -compiler=gccgo main.go`. This option requires the gccgo program is installed. Once the gccgo program is installed, we can also [use the gccgo command directly to compile Go code](#).

The go/* Standard Packages

The `go/*` standard packages provide functionalities of parsing Go source files, which are very useful to write custom Go tools. Please read [go/types: The Go Type Checker](#) and [package documentation](#) for how to use these packages.

System Calls

We can make system calls by call the functions exported by the `syscall` standard package. Please beware that, different from other standard packages, the functions in the `syscall` standard package are operating system dependent.

Go Assembly

Go functions can be implemented with Go assembly language. Go assembly language is a cross-architectures (though not 100%) assembly language. Go assembly language is often used to implement some functions which are critical for Go program execution performances.

For more details, please follow the following links.

- [A quick guide to Go's assembler ↗](#)
- [The Design of the Go assembler ↗](#)

cgo

We can call C code from Go code, and vice versa, through the cgo mechanism. Please follow the following links for details.

- [cgo official documentation ↗](#)
- [C? Go? Cgo! ↗](#)
- [cgo on Go wiki ↗](#)

It is possible to use C++ libraries through cgo by wrapping C++ libraries as C functions.

Please note that using cgo in code may make it is hard to maintain cross-platform compatibility of Go programs, and the calls between Go and C code are some less efficient than Go-Go and C-C calls.

Cross-Platform Compiling

The standard Go compiler supports cross-platform compiling. By setting the `GOOS` and `GOARCH` environments before running the `go build` command, we can build a Windows executable on a Linux machine, and vice versa. Please read the following articles for details.

- [Building windows go programs on linux ↗](#).
- [The current supported target operating systems and compilation architectures ↗](#).

In particular, since Go 1.11, Go Toolchain starts to support WebAssembly as a new kind of GOARCH. Please read [this wiki article ↗](#) for details.

Compiler Directives

The standard Go compiler supports several [compiler directives ↗](#). A directive appears as a comment line like `//go:DirectiveName args`. For examples, we can use the [go:generate ↗](#) directive to generate code and use the [go:embed ↗](#) directive (introduced in Go 1.16) to embed some data files in code.

Build Constraints (Tags)

We can use [build constraints ↗](#) to let compilers build source files selectively (a.k.a., ignore some source files). A build constraint is also called a build tag. A build constraint can appear as a comment line like `// +build constraints` or appear as the suffix in the base name of a source file. Please note: the new [//go:build directive ↗ introduced in Go 1.17 ↗](#) will retire the old `// +build constraints` lines eventually.

More Build Modes

The `go build` command in Go Toolchain supports several build modes. Please run `go help buildmode` to show the available build modes or read the explanations for [-buildmode option ↗](#) instead. Except the **default** build mode, the most used build mode may be the **plugin** build mode. We can use the functions in [the plugin standard package ↗](#) to load and use the Go plugin files outputted by using the **plugin** build mode.

(The **Go 101** book is still being improved frequently from time to time. Please visit [go101.org ↗](#) or follow [@go100and1 ↗](#) to get the latest news of this book. BTW, Tapir, the author of the book, has developed several fun games. You can visit [tapirgames.com ↗](#) to get more information about these games. Hope you enjoy them.)