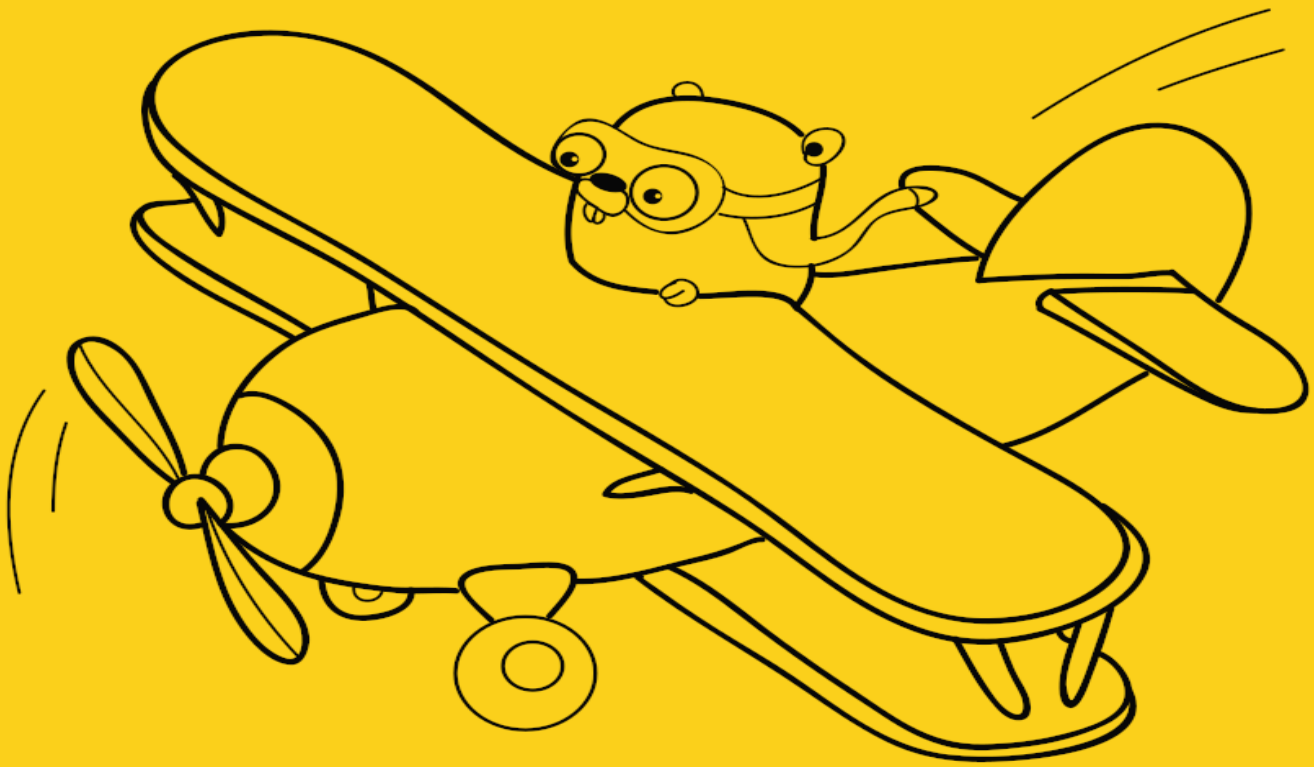
Go Optimizations 101

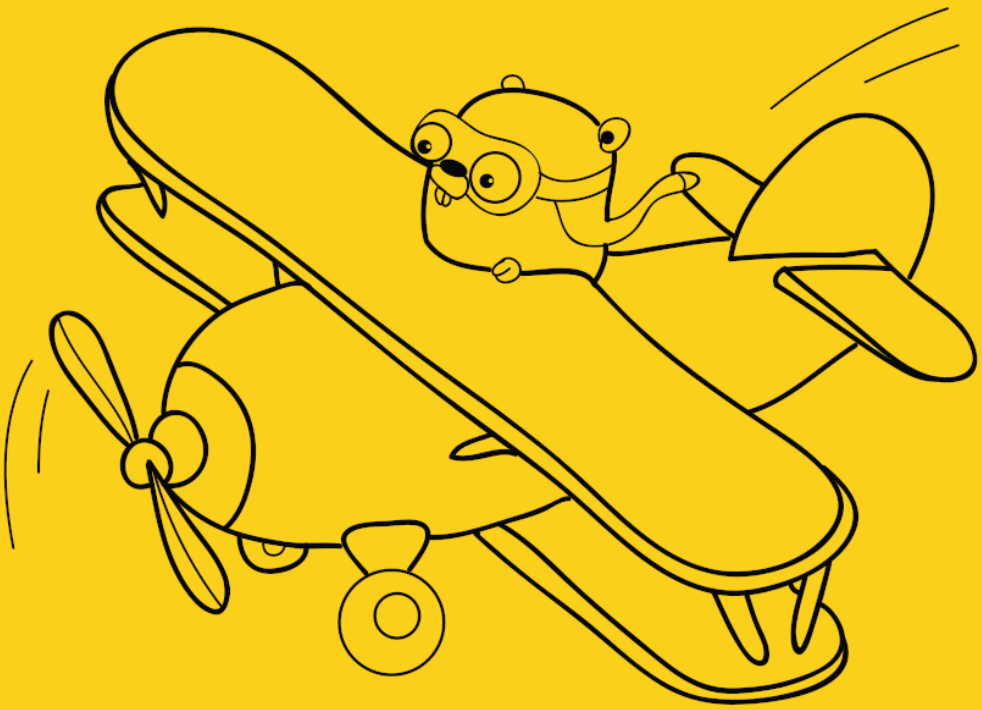
–= v1.19.b–rev–971ee7a–2022/08/29 =–



Tapir Liu

Go Optimizations 101

-= v1.19.b-rev-971ee7a-2022/08/29 =-



Tapir Liu

Acknowledgments

Firstly, thanks to the whole Go community. An active and responsive community guarantees this book is finished in time.

Specially, I want to give thanks to the following people who helped me understand some implementation details in the official standard compiler and runtime: Keith Randall, Ian Lance Taylor, Axel Wagner, Cuong Manh Le, Michael Pratt, Jan Mercl, Matthew Dempsky, Martin Möhrmann, 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.

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.

Thanks to Olexandr Shalakhin for the permission to use one of the wonderful gopher icon designs as the cover image. And thanks to Renee French for designing the lovely gopher cartoon character.

Thanks to the authors of the following open source software and libraries used in building this book:

- golang, https://go.dev/
- gomarkdown, https://github.com/gomarkdown/markdown
- goini, https://github.com/zieckey/goini
- go-epub, https://github.com/bmaupin/go-epub
- pandoc, https://pandoc.org

- calibre, https://calibre-ebook.com/
- GIMP, https://www.gimp.org

Thanks the gophers who ever reported mistakes in this book or made corrections for this book: yingzewen, ivanburak, cortes-, skeeto@reddit, etc.

About Go Optimizations 101

This book provides some code performance optimization tricks, tips, and suggestions. Most of the contents in this book are made based on the official standard Go compiler and runtime implementation.

Life is full of trade-offs, the programming world is, too. In programming, we often need to make trade-offs between code readability, maintainability, development efficiency, and program efficiency, etc. Even for one of the aspects, there are also trade-offs needing to be made. Taking program efficiency for an example, we might need to make trade-offs between memory saving, code execution speed, and implementation difficulty, etc.

In practice, most parts of the code base of a project don't need to be implemented with high performances. Keeping them maintainable and readable is often more important (than making them memory saving and run very fast). The suggestions made in this book are just for the code parts which implementations really need to be high performant. Some of the suggestions often lead to more verbose code. And please note that some of the suggested implementations might be only performant at some certain scenarios.

The contents in this book include:

- how to consume less CPU resources.
- how to consume less memory.
- how to make less memory allocations.

- how to control memory allocation places.
- how to reduce garbage collection pressure.

This book neither explains how to use performance analysis tools, such as pprof, nor tries to study deeply on compiler and runtime implementation details. None of the contents provided in this book make use of unsafe pointers and cgo. And the book doesn't talk about algorithms. In other words, this book tries to provide some optimization suggestions in a way which is clear and easy to understand, for daily general Go programming.

Without particularly indicated, the code examples provided in this book are tested and run on a notebook with the following environment setup:

```
go version go1.19 linux/amd64
goos: linux
goarch: amd64
cpu: Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz
```

Some benchmark times information is removed from benchmark results to keep benchmark lines short.

Please note that:

- some of the talked suggestions in this book work on any platform and for any CPU models, but some others might only work on specified platforms and for specified CPU models. So please benchmark them on the same environments as your production environments before adopting any of them.

- some implementation details of the official standard Go compiler and runtime might change from version to version, which means some of the talked suggestions might not work for future Go toolchain versions.
- the book will be open sourced eventually since a later undetermined time point (about 2023/Jun), in a chapter by chapter way.

About the author

Tapir is the author of this book. He also wrote the Go 101 book. He is planning to write some other **Go 101** series books. Please look forward to.

Tapir was ever (maybe will be again) an indie game developer. You can find his games here: tapirgames.com.

Feedback

Welcome to improve this book by submitting corrections to Go 101 issue list (https://github.com/go101/go101) for all kinds of mistakes, such as typos, grammar errors, wording inaccuracies, wrong explanations, description flaws, code bugs, etc.

It is also welcome to send your feedback to the Go 101 twitter account: @go100and1 (https://twitter.com/go100and1).

Value Parts and Value Sizes

Values and value parts

In Go, a value of some kinds of types always contains only one part (in memory), whereas a value of other kinds of types might contain more than one part. If a value contains more than one part, then one of the parts is called the direct part and the others are called indirect parts. The direct part references the indirect parts.

If a value always contains only one part, then the part may be also called as the direct part of the value, and we say the value has no indirect parts.

In the official standard Go compiler implementation, each value of the following kinds of types always contains only one part:

- boolean types
- numeric types (int8, uint8, int16, uint16, int32, uint32, int64, uint64, int, uint, uintptr, float32, float64, complex64, complex128)
- pointer types
- unsafe pointer types
- struct types
- array types

And a value of the following kinds of types always may contain one or more indirect parts:

- slice types
- map types
- channel types
- function types
- interface types
- string types

When assigning/copying a value, only the direct part of the value is copied. After copying, the direct parts of the destination and source values both are referencing the indirect parts of the source value (if the indirect parts exist).

At run time, each value part is carried on one memory block (memory blocks will be explained in a following chapter). So, if a value contains two parts, the value is very possibly distributed on two memory blocks.

(Note: The terminology `value part` was invented by the Go 101 series books. It is not widely used in Go community. Personally, I think the terminology makes some conveniences when making some explanations.)

Value/type sizes

The size of a value part means how many bytes are needed to be allocated in memory to store the value part at run time.

The size of a value exactly equals to the size of the direct part of the value. In other words, the indirect parts of a value don't contribute to the size of the value. The reason? It has been mentioned above: when assigning/copying a value, only the direct part of the value is copied and the

indirect parts might be shared by multiple values, so it is not a good idea to let the same indirect parts contribute to value sizes.

In the official standard Go compiler implementation, the sizes of all values of a specified type are all the same. So the same value size is also called as the size of that type.

In the official standard Go compiler implementation, the sizes of all types of the same type kind are the same, except struct and array type kinds. From memory allocation point of view,

- A struct values holds all its fields. In other words, a struct value is composed of all its fields. At runtime, the fields of a struct are allocated on the same memory block as the struct itself. Copying a struct value means copying all the fields of the struct value. So all the fields of a struct value contribute to the size of the struct value.
- Like struct values, an array value holds all its elements. In other words, an array is composed of all its elements. At runtime, the elements of an array are allocated on the same memory block as the array itself. Copying an array value means copying all the elements of the array value. So all elements of an array contribute to the size of the array value.

A pointer doesn't hold the value being referenced (pointed) by the pointer. So the value being referenced by the pointer value doesn't contribute to the size of the pointer value (so nil pointers and non-nil pointers have the same size). The two values may be often allocated on two different memory blocks, so copying one of them will not copy the other.

Internally, a slice uses a pointer (on the direct part) to reference all its elements (on the indirect part). The length and capacity information (two `int` values) of a slice is stored on the direct part of the slice. From memory allocation point of view, it doesn't hold its elements. Its elements are allocated on another (indirect) value part other than its direct part. When assigning a slice value to another slice value, none elements of the slice get copied. After assigning, the source slice and the destination slice both reference (but not hold) the same elements. So the elements of a slice don't contribute to the size of a specified slice. This is why the sizes of all slice types are the same.

Like slice values, a map value just references all its entries and a buffered channel value just references its elements being buffered.

In the official standard Go compiler implementation, map values, channel values and function values are all represented as pointers internally. This fact is important to understand some later introduced optimizations made by the official standard Go compiler.

In the official standard Go compiler implementation,

- a string just references all its containing bytes (on the indirect part), though in logic, we can also think a string holds all its containing bytes. The length information of a string is stored on the direct part of the string as an `int` value.
- an interface value just references its dynamic value, though in logic, we can also think an interface value holds its dynamic value.

Detailed type sizes

The following table lists the sizes (used in the official standard Go compiler) of all the 26 kinds of types in Go. In the table, one word means one native word (4 bytes on 32-bit architectures and 8 bytes on 64-bit architectures).

Type Kinds	Value Size	Required by Go Specification
int, uint	1 word	Architecture dependent, 4 bytes on 32-bit architectures and 8 bytes on 64-bit architectures.
int8, uint8 (byte)	1 byte	1 byte
int16, uint16	2 bytes	2 bytes
int32 (rune), uint32, float32	4 bytes	4 bytes
int64, uint64	8 bytes	8 bytes
float64, complex64	8 bytes	8 bytes
complex128	16 bytes	16 bytes
uintptr	1 word	Large enough to store the uninterpreted bits of a pointer value.
bool	1 byte	Unspecified
string	2 words	Unspecified

Type Kinds	Value Size	Required by Go Specification
array	(element value size) * (array length)	The size of an array type is zero if its element type has a zero size.
pointer (safe or unsafe)	1 word	Unspecified
slice	3 words	Unspecified
map	1 word	Unspecified
channel	1 word	Unspecified
function	1 word	Unspecified
interface	2 words	Unspecified
struct	(the sum of sizes of all fields) + (the number of padding bytes)	The size of a struct type is zero if it contains no fields that have a size greater than zero.

Struct padding will be explained in the section after next.

Memory alignments

To fully utilize CPU instructions and get the best performance, the (start) addresses of the memory blocks allocated for (the direct parts of) values of

a specified type must be aligned as multiples of an integer N. Here N is called the alignment guarantee of that type.

For a type `T`, we can call `unsafe.Alignof(t)` to get its general alignment guarantee of type `T`, where `t` is a non-field value of type `T`, and call `unsafe.Alignof(x.t)` to get its field alignment guarantee of type `T`, where `x` is a struct value and `t` is a field value of type `T`. In the current standard Go compiler implementation, the field alignment guarantee and the general alignment guarantee of a type are always equal to each other.

The following table lists the alignment guarantees made by the official standard Go compiler for all kinds of types. Again, one native word is 4 bytes on 32-bit architectures and 8 bytes on 64-bit architectures.

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

The Go specification says:

> For a variable `x` of a 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.

For a variable `x` of an array type: `unsafe.Alignof(x)` is the same as the alignment of a variable of the array's element type.

The official standard Go compiler ensures that the size of a type is a multiple of the alignment guarantee of the type.

Struct padding

To satisfy type alignment guarantee rules mentioned previously, Go compilers may pad some bytes after certain fields of struct values. The padded bytes are counted for struct sizes. So the size of a struct type may be not a simple sum of the sizes of all its fields.

For example, the size of the struct type shown in the following code is 24 on 64-bit architectures.

```
type T1 struct {
   a int8
   // 7 bytes are padded here
   b int64
   c int16
   // 6 bytes are padded here.
}
```

The reason why its size is 24:

- The alignment guarantee of the struct type is the same as its largest alignment guarantee of its filed types. Here is the alignment guarantee

(8, a native word) of type `int64`. This means the distance between the address of the field `b` and `a` of a value of the struct type is a multiple of 8. Clever compilers should choose the minimum possible value: 8. To get the desired alignment, 7 bytes are padded after the field `a`.

- The size of the struct type must be a multiple of the alignment guarantee of the struct type. So considering the existence of the field `c`, the minimum possible size is 24 (8x3), which should be used by clever compilers. To get the desired size, 6 bytes are padded after the field `c`.

Field orders matter in struct type size calculations. If we change the orders of field `b` and `c` of the above struct type, then the size of the struct will become to 16.

```
type T2 struct {
   a int8
   // 1 byte is padded here
   c int16
   // 4 bytes are padded here.
   b int64
}
```

We can use the `unsafe.Sizeof` function to get value/type sizes. For example:

```
package main

import "unsafe"

type T1 struct {
```

```
    a int8
    b int64
    c int16
}

type T2 struct {
    a int8
    c int16
    b int64
}

func main(){
    // The printed values are got on
    // 64-bit architectures.
    println(unsafe.Sizeof(T1{})) // 24
    println(unsafe.Sizeof(T2{})) // 16
}
```

We can view the padding bytes as a form of memory wasting, a trade-off result between program performance, code readability and memory saving.

In practice, generally, we should make related fields adjacent to get good readability, and only order fields in the most memory saving way when it really needs.

Value copy costs and small-size types/values

The cost of copying a value is approximately proportional to the size of the value. In reality, CPU caches, CPU instructions and compiler optimizations might also affect value copy costs.

Value copying operations often happen in value assignments. More value copying operations will be listed in the next section.

To achieve high code execution performance, if it is possible, we should try to avoid

- copying a large quantity of large-size values in a loop.
- copying very-large-size arrays and structs.

Some types in Go belong to small-size types. Copying their values is specially optimized by the official standard Go compiler so that the copy cost is always small.

But what are small-size types? There is not a formal definition. In fact, the definition depends on specific CPU architectures and compiler implementations. In the official standard Go compiler implementation, except large-size struct and array types, all other types in Go could be viewed as small-size types.

What are small-size struct and array values? There is also not a formal definition. The official standard Go compiler tweaks some implementation details from version to version. However, in practice, we can view struct types with no more than 4 native-word-size fields and array types with no more than 4 native-word-size elements as small-size values, such as `struct{a, b, c, d int}`, `struct{element *T; len int; cap int}` and `[4]uint`.

For official standard Go compiler v1.19, a copy cost leap happens between copying 9-element arrays and copying 10-element arrays (the element size is one native word). The similar is for copying 9-field structs and copying 10-field structs (each filed size is one native word).

The proof:

```
package copycost

import "testing"

const N = 1024
type Element = uint64

var a9r [N][9]Element
func Benchmark_CopyArray_9_elements(b *testing.B) {
   var a9s [N][9]Element
   for i := 0; i < b.N; i++ {
      for k := range a9s { a9r[k] = a9s[k] }
   }
}

var a10r [N][10]Element
func Benchmark_CopyArray_10_elements(b *testing.B) {
   var a10s [N][10]Element
   for i := 0; i < b.N; i++ {
      for k := range a10s { a10r[k] = a10s[k] }
   }
}

type S9 struct{ a, b, c, d, e, f, g, h, i Element }
var s9r [N]S9
```

```
func Benchmark_CopyStruct_9_fields(b *testing.B) {
   var s9s [N]S9
   for i := 0; i < b.N; i++ {
      for k := range s9s { s9r[k] = s9s[k] }
   }
}

type S10 struct{ a, b, c, d, e, f, g, h, i, j Element }
var s10r [N]S10
func Benchmark_CopyStruct_10_fields(b *testing.B) {
   var s10s [N]S10
   for i := 0; i < b.N; i++ {
      for k := range s10s { s10r[k] = s10s[k] }
   }
}
```

The benchmark results:

```
Benchmark_CopyArray_9_elements-4   3974 ns/op
Benchmark_CopyArray_10_elements-4  8896 ns/op
Benchmark_CopyStruct_9_fields-4    2970 ns/op
Benchmark_CopyStruct_10_fields-4   8471 ns/op
```

This results indicate copying arrays with less than 10 elements and structs with less than 10 fields might be specially optimized.

The official standard Go compiler might use different criteria for other scenarios to determine what are small struct and array types. For example, in the following benchmark code, the `Add4` function consumes much less CPU resources than the `Add5` function (the official standard Go compiler v1.19).

```
package copycost

import "testing"

type T4 struct{a, b, c, d float32}
type T5 struct{a, b, c, d, e float32}
var t4 T4
var t5 T5

//go:noinline
func Add4(x, y T4) (z T4) {
   z.a = x.a + y.a
   z.b = x.b + y.b
   z.c = x.c + y.c
   z.d = x.d + y.d
   return
}

//go:noinline
func Add5(x, y T5) (z T5) {
   z.a = x.a + y.a
   z.b = x.b + y.b
   z.c = x.c + y.c
   z.d = x.d + y.d
   z.e = x.e + y.e
   return
}

func Benchmark_Add4(b *testing.B) {
   for i := 0; i < b.N; i++ {
      var x, y T4
      t4 = Add4(x, y)
   }
```

```
}

func Benchmark_Add5(b *testing.B) {
   for i := 0; i < b.N; i++ {
      var x, y T5
      t5 = Add5(x, y)
   }
}
```

The benchmark results:

```
Benchmark_Add4-4   2.649 ns/op
Benchmark_Add5-4  19.15 ns/op
```

The `//go:noinline` compiler directives used here are to prevent the calls to the two function from being inlined. If the directives are removed, the `Add4` function will become even more performant.

Value copy scenarios

In Go programming, besides value assignments, there are also several other operations involving value copying:

- value boxing (converting non-interface values into interfaces).
- parameter passing and result returning in function calls.
- receiving values from and sending values to channels.
- put entries into maps.
- append elements to slices.
- iterate container elements/entries with `for-range` loop code blocks.

The following are several examples which show the costs of copying some large values.

Example 1:

```
package copycost

import "testing"

const N = 1024

//go:noinline
func Sum_RangeArray(a [N]int) (r int) {
   for _, v := range a {
      r += v
   }
   return
}

//go:noinline
func Sum_RangeArrayPtr1(a *[N]int) (r int) {
   for _, v := range *a {
      r += v
   }
   return
}

//go:noinline
func Sum_RangeArrayPtr2(a *[N]int) (r int) {
   for _, v := range a {
      r += v
   }
   return
```

```
}

//go:noinline
func Sum_RangeSlice(a []int) (r int) {
   for _, v := range a {
      r += v
   }
   return
}

//===========

var r [128]int

func buildArray() [N]int {
   var a [N]int
   for i := 0; i < N; i ++ {
      a[i] = (N-i)&i
   }
   return a
}

func Benchmark_Sum_RangeArray(b *testing.B) {
   var a = buildArray()
   b.ResetTimer()
   for i := 0; i < b.N; i++ {
      r[i&127] = Sum_RangeArray(a)
   }
}

func Benchmark_Sum_RangeArrayPtr1(b *testing.B) {
   var a = buildArray()
   b.ResetTimer()
```

```
    for i := 0; i < b.N; i++ {
        r[i&127] = Sum_RangeArrayPtr1(&a)
    }
}

func Benchmark_Sum_RangeArrayPtr2(b *testing.B) {
    var a = buildArray()
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        r[i&127] = Sum_RangeArrayPtr2(&a)
    }
}

func Benchmark_Sum_RangeSlice(b *testing.B) {
    var a = buildArray()
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        r[i&127] = Sum_RangeSlice(a[:])
    }
}
```

The benchmark results:

```
Benchmark_Sum_RangeArray-4       897.6 ns/op
Benchmark_Sum_RangeArrayPtr1-4   799.3 ns/op
Benchmark_Sum_RangeArrayPtr2-4   555.3 ns/op
Benchmark_Sum_RangeSlice-4       561.7 ns/op
```

From the results, we could find that the `Sum_RangeArray` function is the slowest one. This is not surprising, because the array value is copied twice in calling this function. One copy happens when passing the array as the argument (arguments are passed by copy in Go), the other happens when

ranging over the array parameter (the direct part of the container following the `range` keyword will be copied if the second iteration variable is used).

The `Sum_RangeArrayPtr1` function is faster than `Sum_RangeArray`, because the array value is only copied once in calling this function. The copy happens when range over the array.

No array copying happens in the calls to the remaining two functions, so those two functions are both the fastest ones.

Example 2:

```
package copycost

import "testing"

type Element [10]int64

//go:noinline
func Sum_PlainForLoop(s []Element) (r int64) {
   for i := 0; i < len(s); i++ {
      r += s[i][0]
   }
   return
}

//go:noinline
func Sum_OneIterationVar(s []Element) (r int64) {
   for i := range s {
      r += s[i][0]
   }
   return
```

```
}

//go:noinline
func Sum_UseSecondIterationVar(s []Element) (r int64) {
   for _, v := range s {
      r += v[0]
   }
   return
}

//==================

func buildSlice() []Element {
   var s = make([]Element, 1000)
   for i := 0; i < len(s); i++ {
      s[i] = Element{0: int64(i)}
   }
   return s
}

var r [128]int64

func Benchmark_PlainForLoop(b *testing.B) {
   var s = buildSlice()
   b.ResetTimer()
   for i := 0; i < b.N; i++ {
      r[i&127] = Sum_PlainForLoop(s)
   }
}

func Benchmark_OneIterationVar(b *testing.B) {
   var s = buildSlice()
   b.ResetTimer()
```

```
    for i := 0; i < b.N; i++ {
        r[i&127] = Sum_OneIterationVar(s)
    }
}


func Benchmark_UseSecondIterationVar(b *testing.B) {
    var s = buildSlice()
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        r[i&127] = Sum_UseSecondIterationVar(s)
    }
}
```

The benchmark results:

```
Benchmark_PlainForLoop-4            911.8 ns/op
Benchmark_OneIterationVar-4         929.3 ns/op
Benchmark_UseSecondIterationVar-4   3753 ns/op
```

From the results, we could learn that the performance of function `Sum_UseSecondIterationVar` is much lower than the `Sum_PlainForLoop` and `Sum_OneIterationVar` functions. The reason is every element is copied to the iteration variable `v` in the function `Sum_UseSecondIterationVar`, and the copy cost is not small (the type `Element` is not a small-size type). The other two functions avoid the copies.

Example 3:

```
package copycost

import "testing"
```

```
type S struct {a, b, c, d, e int}

//go:noinline
func sum_UseSecondIterationVar(s []S) int {
   var sum int
   for _, v := range s {
      sum += v.c
      sum += v.d
      sum += v.e
   }
   return sum
}

//go:noinline
func sum_OneIterationVar_Index(s []S) int {
   var sum int
   for i := range s {
      sum += s[i].c
      sum += s[i].d
      sum += s[i].e
   }
   return sum
}

//go:noinline
func sum_OneIterationVar_Ptr(s []S) int {
   var sum int
   for i := range s {
      v := &s[i]
      sum += v.c
      sum += v.d
      sum += v.e
```

```
    }
    return sum
}

var s = make([]S, 1000)
var r [128]int
func init() {
    for i := range s {
        s[i] = S{i, i, i, i, i}
    }
}

func Benchmark_UseSecondIterationVar(b *testing.B) {
    for i := 0; i < b.N; i++ {
        r[i&127] = sum_UseSecondIterationVar(s)
    }
}

func Benchmark_OneIterationVar_Index(b *testing.B) {
    for i := 0; i < b.N; i++ {
        r[i&127] = sum_OneIterationVar_Index(s)
    }
}

func Benchmark_OneIterationVar_Ptr(b *testing.B) {
    for i := 0; i < b.N; i++ {
        r[i&127] = sum_OneIterationVar_Ptr(s)
    }
}
```

The benchmark results:

```
Benchmark_UseSecondIterationVar-4  2208 ns/op
Benchmark_OneIterationVar_Index-4  1212 ns/op
Benchmark_OneIterationVar_Ptr-4    1182 ns/op
```

Again, the implementation using the second iteration variable is the slowest.

Please note that, if we reduce the field count from 5 to 4 in the above example, then there is no performance differences between the three functions. The reason has been mentioned above. A struct type with no more than 4 fields of native word size is treated as a small-size type.

Memory Allocations

Memory blocks

The basic memory allocation units are called memory blocks. A memory block is a continuous memory segment. As aforementioned, at run time, a value part is carried on a single memory block.

A single memory block might carry multiple value parts. The size of a memory block must not be smaller than the size of any value part it carries.

When a memory block is carrying a value part, we may say the value part is referencing the memory bock.

Memory allocation operations will consume some CPU resources to find the suitable memory blocks. So the more memory blocks are created (for memory allocations), the more CPU resources are consumed. In programming, we should try to avoid unnecessary memory allocations to get better code execution performances.

Memory allocation places

Go runtime might find a memory block on the stack (one kind of memory zone) of a goroutine or the heap (the other kind of memory zone) of the whole program to carry some value parts. The finding-out process is called a (memory) allocation.

The memory management manners of stack and heap are quite different. For most cases, finding a memory block on stack is much cheaper than on heap.

Collecting stack memory blocks is also much cheaper than collecting heap memory blocks. In fact, stack memory blocks don't need to be collected. The stack of a goroutine could be actually viewed as a single memory block, and it will be collected as a whole when the goroutine exits.

On the other hand, when all the value parts being carried on/by a heap memory block are not used any more (in other words, no alive value part is still referencing the memory block), the memory block will be viewed as garbage and automatically collected eventually, during runtime garbage collection cycles, which might consume certain CPU resources (garbage collection will be talked in detail in a later chapter). Generally, the more memory blocks are allocated on heap, the larger pressure is made for garbage collection.

As heap allocations are much more expensive, only heap memory allocations contribute to the allocation metrics in Go code benchmark results. But please note that allocating on stack still has a cost, though it is often comparatively much smaller.

The escape analysis module of a Go compiler could detect some value parts will be only used by one goroutine and try to let those value parts allocated on stack at run time if certain extra conditions are satisfied. Stack memory allocations and escape analysis will be explained with more details in the next chapter.

Memory allocation scenarios

Generally, each of the following operations will make at least one allocation.

- declare variables
- call the builtin `new` function.
- call the builtin `make` function.
- modify slices and maps with composite literals.
- convert integers to strings.
- concatenate strings by using use `+`.
- convert between strings to byte slices, and vice versa.
- convert strings to rune slices.
- box values into interfaces (converting non-interface values into interfaces).
- append elements to a slice and the capacity of the slice is not large enough.
- put new entries into maps and the underlying array (to store entries) of the map is not large enough to store the new entries.

However, the official standard Go compiler makes some special code optimizations so that at certain cases, some of the above listed operations will not make allocations. These optimizations will be introduced later in this book.

Memory wasting caused by allocated memory blocks larger than needed

The sizes of different memory blocks might be different. But they are not arbitrary. In the official standard Go runtime implementation, for memory blocks allocated on heap,

- some memory block size classes (no more than 32768 bytes) are predefined. As of the official standard Go compiler v1.19, the smallest size classes are 8, 16, 24, 32, 48, 64, 80 and 96 bytes.
- For memory blocks larger than 32768 bytes, each of them is always composed of multiple memory pages. The memory page size used by the official standard Go runtime (v1.19) is 8192 bytes.

So,

- to allocate a (heap) memory block for the value which size is in the range `[33, 48]`, the size of the memory block is general (must be at least) 48. In other words, there might be up to 15 bytes wasted (if the value size is 33).
- to create a byte slice with 32769 elements on heap, the size of the memory block carrying the elements of the slice is 40960 (32768 + 8192, 5 memory pages). In other words, 8191 bytes are wasted.

In other words, memory blocks are often larger than needed. The strategies are made to manage memory easily and efficiently, but might cause a bit memory wasting sometimes (yes, a trade-off).

These could be proved by the following program:

```
package main
```

```
import "testing"
import "unsafe"

var t *[5]int64
var s []byte

func f(b *testing.B) {
   for i := 0; i < b.N; i++ {
      t = &[5]int64{}
   }
}

func g(b *testing.B) {
   for i := 0; i < b.N; i++ {
      s = make([]byte, 32769)
   }
}

func main() {
   println(unsafe.Sizeof(*t))      // 40
   rf := testing.Benchmark(f)
   println(rf.AllocedBytesPerOp()) // 48
   rg := testing.Benchmark(g)
   println(rg.AllocedBytesPerOp()) // 40960
}
```

Another example:

```
package main

import "testing"

var s = []byte{32: 'b'} // len(s) == 33
```

```
var r string

func Concat(b *testing.B) {
   for i := 0; i < b.N; i++ {
      r = string(s) + string(s)
   }
}

func main() {
   br := testing.Benchmark(Concat)
   println(br.AllocsPerOp())       // 3
   println(br.AllocedBytesPerOp()) // 176
}
```

There are 3 allocations made within the `Concat` function. Two of them are caused by the byte slice to string conversions `string(s)`, and the sizes of the two memory blocks carrying the underlying bytes of the two result strings are both 48 (which is the smallest size class which is not smaller than 33). The third allocation is caused by the string concatenation, and the size of the result memory block is 80 (the smallest size class which is not smaller than 66). The three allocations allocate 176 (48+48+80) bytes totally. In the end, 14 bytes are wasted. And 44 (15 + 15 + 14) bytes are wasted during executing the `Concat` function.

In the above example, the results of the `string(s)` conversions are used temporarily in the string concatenation operation. By the current official standard Go compiler/runtime implementation (v1.19), the string bytes are allocated on heap (see below sections for details). After the concatenation is

done, the memory blocks carrying the string bytes become into memory garbage and will be collected eventually later.

Reduce memory allocations and save memory

The less memory (block) allocations are made, the less CPU resources are consumed, and the smaller pressure is made for garbage collection.

Memory is cheap nowadays, but this is not true for the memory sold by cloud computing providers. So if we run programs on cloud servers, the more memory is saved by the Go programs, the more money is saved.

The following are some suggestions to reduce memory allocations and save memory in programming.

Avoid unnecessary allocations by allocating enough in advance

We often use the built-in `append` function to push some slice elements. In the statement `r = append(s, elements...)`, if the free capacity of `s` is not large enough to hold all appended elements, Go runtime will allocate a new memory block to hold all the elements of the result slice `r`.

If the `append` function needs to be called multiple times to push some elements, it is best to ensure that the base slice has a large enough capacity, to avoid several unnecessary allocations in the whole pushing process.

For example, to merge some slices into one, the following shown `MergeWithTwoLoops` implementation is more efficient than the

`MergeWithOneLoop` implementation, because the former one makes less allocations and copies less values.

```
package allocations

import "testing"

func getData() [][]int {
   return [][]int{
      {1, 2},
      {9, 10, 11},
      {6, 2, 3, 7},
      {11, 5, 7, 12, 16},
      {8, 5, 6},
   }
}

func MergeWithOneLoop(data ...[]int) []int {
   var r []int
   for _, s := range data {
      r = append(r, s...)
   }
   return r
}

func MergeWithTwoLoops(data ...[]int) []int {
   n := 0
   for _, s := range data {
      n += len(s)
   }
   r := make([]int, 0, n)
   for _, s := range data {
      r = append(r, s...)
```

```
    }
    return r
}

func Benchmark_MergeWithOneLoop(b *testing.B) {
    data := getData()
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        _ = MergeWithOneLoop(data...)
    }
}

func Benchmark_MergeWithTwoLoops(b *testing.B) {
    data := getData()
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        _ = MergeWithTwoLoops(data...)
    }
}
```

The benchmark results:

```
Benchmark_MergeWithOneLoop-4   636.6 ns/op  352 B/op  4
allocs/op
Benchmark_MergeWithTwoLoops-4  268.4 ns/op  144 B/op  1
allocs/op
```

The benchmark results show that allocations affect code execution performance much.

Let's print some logs to see when each of the 4 allocations happens in a `MergeWithOneLoop` function call.

```
package main

import "fmt"

func getData() [][]int {
   return [][]int{
      {1, 2},
      {9, 10, 11},
      {6, 2, 3, 7},
      {11, 5, 7, 12, 16},
      {8, 5, 6},
   }
}

const format = "Allocate from %v to %v (when append
slice#%v).\n"

func MergeWithOneLoop(data [][]int) []int {
   var oldCap int
   var r []int
   for i, s := range data {
      r = append(r, s...)
      if oldCap == cap(r) {
         continue
      }
      fmt.Printf(format, oldCap, cap(r), i)
      oldCap = cap(r)
   }
   return r
}
```

```
func main() {
   MergeWithOneLoop(getData())
}
```

The outputs (for the official standard Go compiler v1.19):

```
Allocate from 0 to 2 (when append slice#0).
Allocate from 2 to 6 (when append slice#1).
Allocate from 6 to 12 (when append slice#2).
Allocate from 12 to 24 (when append slice#3).
```

From the outputs, we could get that only the last `append` call doesn't allocate.

In fact, the `Merge_TwoLoops` function could be faster in theory. As of the official standard Go compiler v1.19, the `make` call in the `Merge_TwoLoop` function will zero all just created elements, which is actually unnecessary. Compiler optimizations in future versions might avoid the zero operation.

BTW, the above implementation of the `Merge_TwoLoops` function has an imperfection. It doesn't handle the integer overflowing case. The following is a better implementation.

```
func Merge_TwoLoops(ss ...[][]byte) []byte {
   n := 0
   for _, s := range data {
      if k := n + len(s); k < n {
         panic("slice length overflows")
      } else {
         n = k
      }
```

```
    }
    r := make([]int, 0, n)
    ...
}
```

Avoid allocations if possible

Allocating less is better, but allocating none is the best.

The following is another example to show the performance differences between two implementations. One of the implementations makes no allocations, the other one makes one allocation.

```
package allocations

import "testing"

func buildOrginalData() []int {
    s := make([]int, 1024)
    for i := range s {
        s[i] = i
    }
    return s
}

func check(v int) bool {
    return v%2 == 0
}

func FilterOneAllocation(data []int) []int {
    var r = make([]int, 0, len(data))
    for _, v := range data {
```

```
      if check(v) {
         r = append(r, v)
      }
   }
   return r
}

func FilterNoAllocations(data []int) []int {
   var k = 0
   for i, v := range data {
      if check(v) {
         data[i] = data[k]
         data[k] = v
         k++
      }
   }
   return data[:k]
}


func Benchmark_FilterOneAllocation(b *testing.B) {
   data := buildOrginalData()
   b.ResetTimer()
   for i := 0; i < b.N; i++ {
      _ = FilterOneAllocation(data)
   }
}

func Benchmark_FilterNoAllocations(b *testing.B) {
   data := buildOrginalData()
   b.ResetTimer()
   for i := 0; i < b.N; i++ {
      _ = FilterNoAllocations(data)
```

```
    }
}
```

The benchmark results:

```
Benchmark_FilterOneAllocation-4  7263 ns/op   8192 B/op  1
allocs/op
Benchmark_FilterNoAllocations-4   903.3 ns/op    0 B/op  0
allocs/op
```

From the benchmark results, we could get that the `FilterNoAllocations` implementation is more performant. (Surely, if the input data is not allowed to be modified, then we have to choose an implementation which makes allocations.)

Save memory and reduce allocations by combining memory blocks

Sometimes, we could allocate one large memory block to carry many value parts instead of allocating a small memory block for each value part. Doing this will reduce many memory allocations, so less CPU resources are consumed and GC pressure is relieved to some extend. Sometimes, doing this could decrease memory wasting, but this is not always true.

Let's view an example:

```
package allocations

import "testing"

const N = 100
```

```
type Book struct {
   Title  string
   Author string
   Pages  int
}

//go:noinline
func CreateBooksOnOneLargeBlock(n int) []*Book {
   books := make([]Book, n)
   pbooks := make([]*Book, n)
   for i := range pbooks {
      pbooks[i] = &books[i]
   }
   return pbooks
}

//go:noinline
func CreateBooksOnManySmallBlocks(n int) []*Book {
   books := make([]*Book, n)
   for i := range books {
      books[i] = new(Book)
   }
   return books
}

func Benchmark_CreateBooksOnOneLargeBlock(b *testing.B) {
   for i := 0; i < b.N; i++ {
      _ = CreateBooksOnOneLargeBlock(N)
   }
}

func Benchmark_CreateBooksOnManySmallBlocks(b *testing.B) {
```

```
    for i := 0; i < b.N; i++ {
        _ = CreateBooksOnManySmallBlocks(N)
    }
}
```

Run the benchmarks, we get:

```
Benchmark_CreateOnOneLargeBlock-4      4372 ns/op  4992 B/op
2 allocs/op
Benchmark_CreateOnManySmallBlocks-4  18017 ns/op  5696 B/op
101 allocs/op
```

From the results, we could get that allocating many small value parts on one large memory block

1. spends much less CPU time.
2. consumes a bit less memory.

The first conclusion is easy to understand. Two allocation operations spend much less time than 101 allocation operations.

The second conclusion has actually already been explained before. As aforementioned, when the size of a small value (part) doesn't exactly match any memory block classes supported by the official standard Go runtime, then a bit larger memory block than needed will be allocated for the small value (part) if the small value (part) is created individually. The size of the `Book` type is 40 bytes (on 64-bit architectures), whereas the size of the smallest memory block size class larger than 40 is 48. So allocating a `Book` value individually will waste 8 bytes.

In fact, the second conclusion is only right under certain conditions. Specifically, the conclusion is not right when the value N is in the range from 820 to 852. In particular, when N == 820, the benchmark results show allocating many small value parts on one large memory block consumes 3.5% more memory.

```
Benchmark_CreateOnOneLargeBlock-4     30491 ns/op  47744 B/op
2 allocs/op
Benchmark_CreateOnManySmallBlocks-4  145204 ns/op  46144 B/op
821 allocs/op
```

Why does the `CreateBooksOnOneLargeBlock` function consume more memory when N == 820? Because it needs to allocate a memory block with the minimum size as 32800 (820 * 40), which is larger than the largest small memory block class 32768. So the memory block needs 5 memory pages, which total size is 40960 (8192 * 5). In other words, 8160 (40960 - 32800) bytes are wasted.

Despite it sometimes wastes more memory, generally speaking, allocating many small value parts on one large memory block is comparatively better than allocating each of them on a separated memory block. This is especially true when the life times of the small value parts are almost the same, in which case allocating many small value parts on one large memory block could often effectively avoid memory fragmentation.

Use value cache pool to avoid some allocations

Sometimes, we need to frequently allocate and discard values of a specified type from time to time. It is a good idea to reuse allocated values to avoid a

large quantity of allocations.

For example, there are many non-player characters (NPC) in RTS games. A large quantity of NPCs will be spawned and destroyed from time to time in a game session. The related code is like

```
type NPC struct {
   name       [64]byte
   nameLen    uint16
   blood      uint16
   properties uint32
   x, y       float64
}

func SpawnNPC(name string, x, y float64) *NPC {
   var npc = newNPC()
   npc.nameLen = uint16(copy(npc.name[:], name))
   npc.x = x
   npc.y = y
   return npc
}

func newNPC() *NPC {
   return &NPC{}
}

func releaseNPC(npc *NPC) {
}
```

As Go supports automatic GC, the `releaseNPC` function may do nothing. However, such implementation will lead to a large quantity of allocations in

game playing and cause large pressure for garbage collection, so that it is hard to guarantee a good game FPS (frames per second).

We could instead use a cache pool to reduce allocations, like the code shown below.

```
import "container/list"

var npcPool = struct {
   sync.Mutex
   *list.List
}{
   List: list.New(),
}

func newNPC() *NPC {
   npcPool.Lock()
   defer npcPool.Unlock()
   if npcPool.Len() == 0 {
      return &NPC{}
   }
   return npcPool.Remove(npcPool.Front()).(*NPC)
}

func releaseNPC(npc *NPC) {
   npcPool.Lock()
   defer npcPool.Unlock()
   *npc = NPC{} // zero the released NPC
   npcPool.PushBack(npc)
}
```

By using the pool (also called free list), allocations of `NPC` values will be reduced much, which is very helpful to keep a smooth FPS (frames per second) in game playing.

If the cache pool is used in only one goroutine, then concurrency synchronizations are not necessary in the implementation.

We could also set a max size for the pool to avoid the pool occupies too much memory.

The standard `sync` package provides a `Pool` type to provide similar functionalities but with several design differences:

- a free object in the a `sync.Pool` will be automatically garbage collected if it is found to be unused in two successive garbage collection cycles. This means the max size of a `sync.Pool` is dynamic and depends on the demand at run time.
- the types and sizes of the objects in a `sync.Pool` could be different. But the best practice is to make sure the objects put in the same `sync.Pool` are of the some type and have the same size.

Personally, I find the design of `sync.Pool` seldom satisfies the needs in practice. So I often use custom value cache pool implementations in my Go projects.

Stack and Escape Analysis

Goroutine stacks

In the last chapter, it is mentioned that some memory blocks will be allocated on stack(s), one of the two kinds of memory zones.

When a new goroutine is created, Go runtime will create a stack memory zone for the goroutine. The current official standard Go runtime uses contiguous stacks (instead of segmented stacks), which means the stack memory zone is a single continuous memory segment.

Allocating memory blocks on stack is generally much faster than on heap, which will be explained in the following sections. Memory blocks allocated on stack don't need to be garbage collected, which is another big advantage over memory blocks allocated on heap. Thirdly, memory blocks allocated on stack are also often more CPU cache friendly than the ones allocated on heap.

So, generally speaking, if it is possible, the official standard Go compiler will let Go runtime try to allocate memory blocks on stack.

Escape analysis

Not all value parts are capable of being allocated on stack. One principle condition to allocate a value part on stack is the value part will be only used in one goroutine (the current one) during its life time. Generally, if the

compiler detects a value part is used by more than one goroutine or it is unable to make sure whether or not the value part is used by only one goroutine, then it will let the value part allocated on heap. We also say the value part escapes (to heap).

The job of determining which value parts could be allocated on stack is handled by the escape analysis module of a Go compiler. However, situations are often complicated in practice so that it is almost impossible for the escape analysis module to find out all such value parts at compile time. And a more powerful escape analysis implementation will increase compilation time much. So, at run time, some value parts will be allocated on heap even if they could be safely allocated on stack.

Please note that a value part could be allocated on stack doesn't mean the value part will be allocated on stack for sure at run time. If the size of the value part is too large, then the compiler will still let the value part allocated on heap anyway. The size thresholds used in the official standard Go compiler will be introduced in later sections of this chapter.

The basic escape analysis units are functions. Only the local values will be escape analyzed. All package-level variables are allocated on heap for sure.

Value parts allocated on heap may be referenced by value parts allocated on either heap or stack, but value parts allocated on a stack may be only referenced by value parts allocated on the same stack. So if a value part is being referenced by another value part allocated in heap, then the former one (the referenced one) must be also allocated on heap. This means value parts being referenced by package-level variables must be heap allocated.

Escape analysis examples

The `-m` compiler option is used to show escape analysis result. Let's view an example.

```
// escape.go
package main

func main() {
   var (
      a = 1 // moved to heap: a
      b = false
      c = make(chan struct{})
   )
   go func() {
      if b {
         a++
      }
      close(c)
   }()
   <-c
   println(a, b) // 1 false
}
```

Run it with the `-m` compiler option:

```
$ go run -gcflags=-m escape.go
# command-line-arguments
./escape.go:10:5: can inline main.func1
./escape.go:6:3: moved to heap: a
./escape.go:10:5: func literal escapes to heap
1 false
```

From the outputs, we know that the variable `a` escapes to heap but the variable `b` is allocated on stack. What about the variable `c`? The direct part of channel `c` is allocated on stack. The indirect parts of channels are always allocated on heap, so escape messages for channel indirect parts are always omitted.

Why the variable `b` is allocated on stack but the variable `a` escapes? Aren't they both used on two goroutines? The reason is that the escape analysis module is so smart that it detects the variable `b` is never modified and thinks it is a good idea to use a (hidden implicit) copy of the variable `b` in the new goroutine.

Let's add one new line `b = !b` before the print line and run it again.

```
// escape.go
package main

func main() {
   var (
      a = 1     // moved to heap: a
      b = false // moved to heap: b
      c = make(chan struct{})
   )
   go func() {
      if b {
         a++
      }
      close(c)
   }()
   <-c
   b = !b
```

```
    println(a, b) // 1 true
}
```

The outputs:

```
./escape.go:10:5: can inline main.func1
./escape.go:6:3: moved to heap: a
./escape.go:7:3: moved to heap: b
./escape.go:10:5: func literal escapes to heap
1 true
```

Now both the variable `a` and the variable `b` escape. In fact, for this specified example, the compiler could still use a copy of variable `b` in the new goroutine. But it is too expensive to let the escape analysis module analyze the concurrency synchronizations used in code.

For the similar reason, the escape analysis module also doesn't try to check whether or not the variable `a` will be really modified. If we change `b` to a constant, then the variable `a` will be allocated on stack, because the line `a++` will be optimized away.

Stack frames of function calls

At compile time, the Go compiler will calculate the maximum possible stack size a function will use at run time. The calculated size is called the stack frame size of the function. In the calculation, all code branches will be considered, even if some of them will not get executed at run time.

We may use the `-S` compiler option to show function stack frame sizes. Let's use the following program as an example:

```
// frame.go
package main

import (
   "fmt"
   "math/rand"
)

func bar(s []uint32, n byte) (r byte) {
   s = s[:1250]
   var a [5000]byte // 5000 == 1250 * 4
   for i, v := range s {
      a[i * 4 + 0] = byte(v >> 0)
      a[i * 4 + 1] = byte(v >> 8)
      a[i * 4 + 2] = byte(v >> 16)
      a[i * 4 + 3] = byte(v >> 24)
   }
   var v = a[0]
   for i := 1; i < len(a); i++ {
      r += v ^ a[i]
      v = a[i]
   }
   return
}

func foo(n byte) uint16 {
   var a [2500]uint32
   for i := range a {
      a[i] = rand.Uint32()
   }
   x := bar(a[:1250], n)
   y := bar(a[1250:], ^n)
   return uint16(x) << 8 | uint16(y)
```

```
}

func main() {
   x := foo(123)
   fmt.Println(x)
   duck()
}

var v interface{}
func duck() {
   type _ int // avoid being inlined
   if v != nil {
      v = [16000]byte{}
      panic("unreachable")
   }
}
```

Note that the `type _ int` line within the `duck` function is to make the function not inline-able. This will be explained in a later chapter.

Run it with the `-S` compiler option, we will get the following outputs (some texts are omitted):

```
$ go run -gcflags=-S frame.go
...
... TEXT  "".bar(SB), ABIInternal, $5024-32
...
... TEXT  "".foo(SB), ABIInternal, $10056-8
...
... TEXT  "".main(SB), ABIInternal, $64-0
...
```

```
... TEXT  "".duck(SB), ABIInternal, $16024-0
...
```

From the outputs, we could get

- the frame size of the `bar` function is 5024 bytes.
- the frame size of the `foo` function is 10056 bytes.
- the frame size of the `main` function is 64 bytes.
- the frame size of the `duck` function is 16024 bytes. Please note that, although the `duck` function is a de facto dummy function, its frame size is not zero. This fact will be made use of in a code optimization trick shown later.

At run time, before entering the execution of a function call, Go runtime will mark out a memory segment on the current stack for the call (to allocate stack memory blocks). The memory segment is called the stack frame of the function call and its size is the stack frame size of the called function. As mentioned above, the frame size is calculated at compile time.

When a value (part) within a function is determined to be allocated on stack, its memory address offset (relative to the start of the stack frame of any call to the function) is also determined, at compile time. At run time, once the stack frame of a call to the function is marked out, the memory addresses of all value parts allocated on the stack within the function call are all determined consequently, which is why allocating memory blocks on stack is much faster than on heap.

Stacks growth and shrinkage

At run time, a cursor is used to mark the boundary between the used part and available part on the stack of a goroutine. The cursor will move forwards (towards the available part) when entering a deeper function call and move backwards (towards the used part) when a function call fully exits. The move-forwards and move-backwards step sizes are equal to the stack frame size of the respective called functions. Go runtime uses this way to mark out stack frames for function calls.

The stack size of a goroutine might be dynamic at run time. In the implementation of the official standard Go runtime, before version 1.19, the initial size of a stack is always 2KiB; since version 1.19, the initial size is adaptive, but the default initial stack size is still 2KiB.

For some goroutines, as function calls go deeper and deeper, more and more function call stack frames are needed on the stack, the current stack size might become insufficient. So stacks will grow on demand during the execution of a goroutine. On the contrary, when the function call depth become shallower and shallower, the stack size of a goroutine will become too large. So stacks might also shrinkage. However, to avoid growing and shrinking frequently, a stack will not shrink immediately as soon as it becomes too large. The stack of a goroutine will shrink only when it is found too large during a garbage collection cycle and the goroutine isn't doing anything or sitting in a system call or a cgo call. Each stack shrinkage halves the old stack. (Special thanks to Ian Lance Taylor for the explanations.)

In the current implementation of the official standard Go compiler and runtime, stack sizes are always 2^n bytes, where n is an integer larger than 10. The minimum stack size is 2KiB.

In the process of a stack growth (shrinkage), Go runtime will create a larger (smaller) stack memory zone to replace the old one, then the already used part of the old stack will be copied to the new stack, which leads to the addresses of all value parts allocated on the stack change. So after the copying, the stack allocated pointers which are referencing stack value parts must be all updated accordingly.

The following is an example to show the effects of stack growth and shrinkage. We will find the address of the variable `x` changes after a stack growth (caused by calling the function `f`) and after a garbage collection cycle (started manually) ends.

```
package main

import "runtime"

//go:noinline
func f(i int) byte {
   var a [1<<13]byte // allocated on stack and make stack grow
   return a[i]
}

func main(){
   var x int
   println(&x)  // <address 1>
   f(1)         // (make stack grow)
```

```
    println(&x)  // <address 2>
    runtime.GC() // (make stack shrink)
    println(&x)  // <address 3>
    runtime.GC() // (make stack shrink)
    println(&x)  // <address 4>
    runtime.GC() // (stack does not shrink)
    println(&x)  // <address 4>
}
```

Note that each of the first two manual `runtime.GC` calls causes a stack shrinkage, but the last one doesn't.

Let's make an analysis on how the stack of the main goroutine grows and shrinks during the execution of the program.

1. At the starting, the initial stack size is 2KiB.
2. The frame size of the `f` function is 8216 bytes. Before entering the `f` function call, the stack grows to 16KiB, which is the smallest 2^n which is larger than the demand (10KiB or so).
3. After the `f` function call fully exits, the stack will not shrink immediately, until a garbage collection cycle happens.
4. The first `runtime.GC` call shrinks the stack to 8KiB.
5. The second `runtime.GC` call shrinks the stack to 4KiB.
6. The third `runtime.GC` call doesn't shrink the stack, though 2KiB is sufficient now. The reason is the current official standard runtime implementation doesn't shrink a stack to a size which is less than 4 times of the demand.

The reason why the array `a` is allocated on stack will be explained in the a later section in this chapter.

Though it happens rarely in practice, we should try to avoid making stack grow and shrink frequently. That is, for some rare cases, allocating some value parts on stack might be not a good idea.

There is a global limit of stack size each goroutine may reach. If a goroutine exceeds the limit while growing its stack, the whole program crashes. As of Go toolchain v1.19, the default maximum stack size is 1 GB (not GiB) on 64-bit systems, and 250 MB (not MiB) on 32-bit systems. Please note that, the actual max stack size allowed by the current stack grow implementation is about the half of the max stack size setting (512 MiB on 64-bit systems and 128 MiB on 32-bit systems).

The following is an example program which will crash for stack exceeds limit.

```
package main

const N = 1024 * 1024 * 10 // 10M

func f(v [N]byte, n int) {
   if n > 0 {
      f(v, n-1)
   }
}

func main() {
   var x [N]byte
```

```
    f(x, 50)
}
```

If the call `f(x, 50)` is changed to `f(x, 48)`, then the program will exit without crashing (on 64-bit systems).

We can call the `runtime/debug.SetMaxStack` function to change the global stack maximum limit setting. There is not a formal way to control the initial stack size of a goroutine, though an informal way will be provided in a later section of this chapter.

Memory wasting caused by unused stack spaces

As aforementioned, Go runtime will create a stack for each goroutine. The size of the memory zone of a stack is often larger than the size the stack really needs, which means there is some memory wasted on most stacks. If the stack of a goroutine grows to a much larger size, then there will be more memory wasted. And, a bit extra memory is needed to store goroutine information. So the more goroutines are alive at the same time, the more memory is wasted. Thousands of simultaneous alive goroutine might be not a problem, millions might be, depending on how much memory is available.

For all kinds of reasons, a value (part) will escape to heap even if it is only used in one goroutine

The following sub-sections will introduce some such cases.

A local variables declared in a loop will escape to heap if it is referenced by a value out of the loop

For example, the variable `n` in the following code will escape to heap. The reason is there might be many coexisting instances of `n` if its containing loop needs many steps to finish. The number of the instances is often hard or impossible to determine at compile time, whereas the stack frame size of a function must be determined at compile time, so the compiler lets every instance of `n` escape to heap.

It is admitted that, for this specified example, the compiler could be smarter to determine that the loop step is one and only one coexisting instance of `n` is needed. The current the official standard Go compiler (v1.19) doesn't consider such special calses.

```
package main

func main() {
   var x *int
   for {
      var n = 1 // moved to heap: n
      x = &n
      break
   }
   _ = x
}
```

The value parts referenced by an argument will escape to heap if the argument is passed to interface method calls

For example, in the following code, the value `x` will be allocated on stack, but the value `y` will be allocated on heap.

```
package main

type I interface {
   M(*int)
}

type T struct{}
func (T) M(*int) {}

var t T
var i I = t

func main() {
   var x int // does not escape
   t.M(&x)
   var y int // moved to heap: y
   i.M(&y)
}
```

It is often impossible or too expensive for compilers to determine the dynamic value (and therefor the concrete method) of an interface value, so the official standard compiler gives up to do so for most cases. Potentially, the concrete method of the interface value could pass its arguments to some other goroutines. So, for safety, the official standard compiler conservatively lets the value parts referenced by the arguments escape to heap.

For some case, the compiler could determine the dynamic value (and therefor the concrete method) of an interface value at compile time. If the compiler finds the concrete method doesn't pass an argument to other goroutines, then it will let the value parts referenced by the argument not escape to heap. For example, in the following code, the value `x` and `y` are both allocated on stack. The reason why the value `y` doesn't escape is the method call `i.M(&y)` is de-virtualized to `t.M(&y)` at compile time.

```
package main

type I interface{
   M(*int)
}

type T struct{}
func (T) M(*int) {}

func main() {
   var t T
   var i I = t

   var x int
   t.M(&x)
   var y int
   i.M(&y)
}
```

A `reflect.ValueOf` function call makes the values referenced by its argument escape to heap

As of now (Go toolchain v1.19), a call to the `reflect.ValueOf` function makes the values references by its argument escape to heap. This could be proved by the following program.

```
package main

import "reflect"

func main() {
   var n = 1 // moved to heap: n
   _ = reflect.ValueOf(&n)

   var k = 1
   _ = reflect.ValueOf(k) // k escapes to heap
}
```

Please note that the message `k escapes to heap` actually means `a copy of k escapes to heap.`

The print functions in the `fmt` standard package all call the `reflect.ValueOf` function internally, so the calls to these functions also make the values references by their arguments escape to heap. For example, in the following code, the variable `x` will escape to heap, but the variable `y` doesn't.

```
package main

import "fmt"

func main() {
   var x = 1 // moved to heap: x
```

```
    fmt.Println(&x)
    var y = 2 // does not escape
    println(&y)
}
```

The values referenced by function return results will escape

It is expensive to trace how the return results of a function are used by the callers of the function. So the compiler lets all value parts referenced by the results escape.

For example, in the following code, assume the function `f` is not inline-able, then the value `*p` will escape to heap, for it is referenced by the return result. Note, the compiler is smart enough to decide the value `t`, which is referenced by the argument `x` passed to the `f` function call, could be safely allocated on stack.

```
package main

//go:noinline
func f(x *int) *int {
    var n = *x + 1 // moved to heap: n
    return &n
}

func main() {
    var t = 1 // does not escape
    var p = f(&t)
    println(*p) // 2
    println(&t) // 0xc000034758
```

```
    println(p)  // 0xc0000140c0
}
```

Function inline might affect escape analysis results

If a function call is inlined in its caller function, then the result variables of the inlined call become into local variables of the caller function, which will ease escape analysis much. After the inlining, compiler might let some value parts which originally escape to heap be allocated on stack instead.

Still use the example in the last section as an example, but remove the `//go:noinline` line to make the function `f` inline-able, then the value `*p` will be allocated on stack.

```
package main

func f(x *int) *int { // x does not escape
    var n = *x + 1 // moved to heap: n
    return &n
}

func main() {
    var t = 1
    var p = f(&t)
    println(*p) // 2
    println(&t) // 0xc000034760
    println(p)  // 0xc000034768
}
```

The printed two addresses show the distance between the value `t` and `*p` is the size of the value size of `*p`, which indicates the two values are both allocated on stack. Please note that the message `"moved to heap: n"` is still there, but it is for the `f` function calls which are not inlined (no such calls in this tiny program).

The following is the rewritten code (by the compiler) after inlining:

```
package main

func main() {
   var t = 1
   var s = &t
   var n = *s + 1
   var p = &n
   println(*p)
   println(&t) // 0xc000034760
   println(p)  // 0xc000034768
}
```

After the rewrite, the compiler easily knows the `n` variable is only used in the current goroutine so lets it not escape.

Function inlining is not always helpful for escape analysis

Currently (Go toolchain v1.19), function inlining is not always helpful for escape analysis. This is caused by some small defects in the compiler implementation. For example, in the following code, even if the `createSlice` function call is inlined, the slice returned by it is still

allocated on heap, because constants don't propagate well in the current implementation.

```
// constinline.go
package main

func createSlice(n int) []byte { // line 4
   return make([]byte, n) // line 5
}

func main() {
   var x = createSlice(32)  // line 9
   var y = make([]byte, 32) // line 10
   _, _ = x, y
}
```

Run it:

```
$ go run -gcflags="-m" constinline.go
# command-line-arguments
./constinline.go:4:6: can inline createSlice
./constinline.go:8:6: can inline main
./constinline.go:9:21: inlining call to createSlice
./constinline.go:5:13: make([]byte, n) escapes to heap
./constinline.go:9:21: make([]byte, n) escapes to heap
./constinline.go:10:14: make([]byte, 32) does not escape
```

Future versions of the official standard Go compiler might make improvements to avoid more unnecessary escapes.

Control memory block allocation places

As aforementioned, sometimes, it is not a good idea to let some very large memory blocks allocated on stack even if these memory blocks are capable of being allocated on stack.

On the other hand, it is generally a good idea to try to let small memory blocks allocated on stack if it is possible.

The following will introduce some facts and tips to control memory block allocation places.

Ensure a value is allocated on heap

For some rare scenarios, we might expect some values which are only used in the current goroutine to be allocated on heap at run time. There are two ways to make sure a value `v` will escape to heap at run time:

1. let another value allocated on heap reference the value `v`.
2. let two goroutines use `v` and at least one goroutine modifies `v` after they both start.

Here, only the first way will be talked about.

The standard packages internally use a trick like the following code shows to make some values escape for sure. The trick uses a package-level variable `sink` (which is allocated on heap for sure) to reference the value which is expected to escape. The line `sink = x` will be optimized away but it will still affect the decisions made by the escape analysis module. Please note that, a value referencing the value expected to escape should be passed

to the `escape` function. Generally, the passed value is a pointer to the value expected to escape.

```
var sink interface{}

//go:noinline
func escape(x interface{}) {
   sink = x
   sink = nil
}
```

the following is an example which uses the trick:

```
package main

var sink interface{}

//go:noinline
func escape(x interface{}) {
   sink = x
   sink = nil
}

func main() {
   var a = 1    // moved to heap: a
   var b = true // moved to heap: b
   escape(&a)
   escape(&b)
   println(a, b)
}
```

As the `//go:noinline` directive is only intended to be used in toolchain and standard packages development, the code for the trick could be modified as the following in user code. The official standard Go compiler will not inline calls to a function which will potentially call itself.

```
var sink interface{}

func escape(x interface{}) {
   if x == nil {
      escape(&x) // avoid being inlined
      panic("x must not nil")
   }

   sink = x
   sink = nil
}
```

Surely, if we know some values will be allocated on heap, then we can just let those values reference the values expected to escape.

Use explicit value copies to help compilers detect some values don't escape

Let's revisit the example used previously (with a small modification, `b` is initialized as `true` now):

```
package main

func main() {
   var (
      a = 1    // moved to heap: a
```

```
      b = true // moved to heap: b
      c = make(chan struct{})
   )
   go func() {
      if b {
         a++
      }
      close(c)
   }()
   <-c
   b = !b
   println(a, b) // 2 false
}
```

In this example, variables `a` and `b` both escape. If we modify the example by making copies of `a` and `b` (in two different ways) like the following, then both `a` and `b` will not escape.

```
package main

func main() {
   var (
      a = 1    // doesn't escape
      b = true // doesn't escape
      c = make(chan int)
   )
   b1 := b
   go func(a int) {
      if b1 {
         a++
         c <- a
      }
```

```
    }(a)
    a = <-c
    b = !b
    println(a, b) // 2 false
}
```

For this specified example, whether or not variables `a` and `b` escape has a very small effect on overall program performance. But the tip introduced here might be helpful elsewhere.

Memory size thresholds used by the compiler to make allocation placement decisions

To avoid crashes caused by goroutine stacks exceed the maximum stack size, even though the compiler makes sure that a value part is used by only one goroutine, it will still let the value part allocated on heap if the size of the value part is larger than a threshold. There are several such thresholds used by the official standard Go compiler (v1.19):

- in a conversion between string and byte slice, it the result string or byte slice contains no more than 32 bytes, then its indirect part (its underlying bytes) will be allocated on stack, otherwise, on heap. However, the threshold of converting a constant string (to byte slice) is relaxed to 64K (64 * 1024) bytes.
- if the size of a type `T` is larger than 64K (64 * 1024) bytes, then `T` values allocated by `new(T)` and `&T{}` will be allocated on heap.
- if the size of the backing array `[N]T` of the result of the `make([]T, N)` call is larger than 64K (64 * 1024) bytes, then the backing array will

be allocated on heap. Here, `N` is a constant so that the compile could make decisions at compile time. If it is not a constant and larger than zero, then the backing array will be always allocated on heap.

- in a variable declaration, if the size of the direct part of the variable is larger than 10M (10 * 1024 * 1024) bytes, then the direct part will be allocated on heap.

The following are some examples to show the effects of these thresholds.

Example 1:

```
package main

import "testing"

const t = "1234567890ABCDEF" // len(t) == 16
const S = t + t + t + t + t  // len(S) == 80
var bs33 = make([]byte, 33)

func f(w []byte) func() {
   return func() {
      _ = string(w) // string(w) does not escape
   }
}

func g(v string) func() {
   return func() {
      _ = []byte(v) // ([]byte)(v) does not escape
   }
}

func h() {
```

```
    _ = []byte(S) // ([]byte)(S) does not escape
}

func main() {
    stat := func( f func() ) int {
        allocs := testing.AllocsPerRun(1, f)
        return int(allocs)
    }
    println(stat(f(bs33)))      // 1 (heap allocation)
    println(stat(f(bs33[:32]))) // 0 (heap allocations)

    println(stat(g(S[:33])))    // 1 (heap allocation)
    println(stat(g(S[:32])))    // 0 (heap allocations)
    println(stat(h))            // 0 (heap allocations)
}
```

Please note: the `"XXX does not escape"` analysis message means the compiler knows that `XXX` is used only by one goroutine, but if the corresponding threshold is surpassed, then `XXX` will be allocated on heap (special thanks to Keith Randall for the explanations).

From the outputs of example 1, we could affirm the byte elements of the result strings or byte slices of conversions between string and byte slice will be always allocated on heap if the length of the result is larger than 32.

Note, however, the threshold of converting a constant string is relaxed to 64K bytes, which is why these are no escapes happening in the `h` function.

Example 2:

```
package main

import "testing"

const N = 64 * 1024 // 65536
var n byte = 123

func new65537() byte {
   // new([65537]byte) escapes to heap
   var a = new([N+1]byte)
   for i := range a { a[i] = n }
   return a[n]
}

func new65535() byte {
   // new([65535]byte) does not escape
   var a = new([N-1]byte)
   for i := range a { a[i] = n }
   return a[n]
}

func comptr65537() byte {
   // &[65537]byte{} escapes to heap
   a := &[N+1]byte{}
   for i := range a { a[i] = n }
   return a[n]
}

func comptr65535() byte {
   // &[65535]byte{} does not escape
   a := &[N-1]byte{}
   for i := range a { a[i] = n }
   return a[n]
```

```
}

func main() {
   stat := func( f func() byte ) int {
      allocs := testing.AllocsPerRun(1, func() {
         f()
      })
      return int(allocs)
   }
   println(stat(new65537))    // 1 (heap allocation)
   println(stat(new65535))    // 0 (heap allocations)
   println(stat(comptr65537)) // 1 (heap allocation)
   println(stat(comptr65535)) // 0 (heap allocations)
}
```

From the outputs of example 2, we could affirm that the `T` value created in `new(T)` and `&T{}` will be always allocated on heap if the size of type `T` is larger than 64K (65536) bytes.

(Please note that, we deliberately ignore the case of size 65536. Before Go toolchain 1.17, `T` values will be allocated on heap if the size of `T` is 65536. Go toolchain v1.17 changed this.)

Example 3:

```
package main

import "testing"

const N = 64 * 1024 // 65536
var n = 1
```

```
func makeSlice65537() bool {
   // make([]bool, N + 1) escapes to heap
   s := make([]bool, N+1)
   for i := range s { s[i] = n>0 }
   return s[len(s)-1]
}

func makeSlice65535() bool {
   // make([]bool, N - 1) does not escape
   s := make([]bool, N-1)
   for i := range s { s[i] = n>0 }
   return s[len(s)-1]
}

func makeSliceVarSize() bool {
   // make([]bool, n) escapes to heap
   s := make([]bool, n)
   for i := range s { s[i] = n>0 }
   return s[len(s)-1]
}

func main() {
   stat := func( f func() bool) int {
      allocs := testing.AllocsPerRun(1, func() {
         f()
      })
      return int(allocs)
   }
   println(stat(makeSlice65537))   // 1 (heap allocation)
   println(stat(makeSlice65535))   // 0 (heap allocations)
   println(stat(makeSliceVarSize)) // 1 (heap allocation)
}
```

From the outputs of example 3, we could affirm that:

- if `N` is a constant and the size of the backing array `[N]T` of the result of the `make([]T, N)` call is larger than 64K (64 * 1024) bytes, then the backing array will be allocated on heap.
- if `n` is not a constant and `n` is larger than zero, then the backing array will be always allocated on heap, for compilers couldn't determine the backing array size of the result slice at compile time.

(Again, please also note that, if the constant `N` equals to 65536, the elements of `make([]T, N)` will be allocated on heap before Go toolchain v1.17, but will be allocated on stack since Go toolchain v1.17.)

Example 4:

```
package main

import "testing"

const N = 10 * 1024 * 1024 // 10M
var n = N - 1

func declare10M() byte {
   var a [N]byte
   for i := range a { a[i] = byte(i)}
   return a[n]
}

func declare10Mplus1() byte {
   var a [N+1]byte // moved to heap: a
   for i := range a { a[i] = byte(i)}
```

```
   return a[n]
}

func redeclare10M() byte {
   a := [N]byte{}
   for i := range a { a[i] = byte(i)}
   return a[n]
}

func redeclare10Mplus1() byte {
   a := [N+1]byte{} // moved to heap: a
   for i := range a { a[i] = byte(i)}
   return a[n]
}

func main() {
   stat := func( f func() byte) int {
      allocs := testing.AllocsPerRun(1, func() {
         f()
      })
      return int(allocs)
   }
   println(stat(declare10M))         // 0
   println(stat(declare10Mplus1))    // 1
   println(stat(redeclare10M))       // 0
   println(stat(redeclare10Mplus1)) // 1
}
```

From the execution result of example 4, we could affirm that, in a variable declaration, if the size of the direct part of the variable is larger than 10M (10 * 1024 * 1024) bytes, then the direct part will be allocated on heap.

Use smaller thresholds

Please note, the above mentioned thresholds 64K and 10M will become much smaller (16K and 128K respectively) if the `-smallframes` compiler option is specified. For example, if we use the command `go run -gcflags='-smallframes' main.go` to run the following program, then all the checked functions will make one allocation on heap.

```
// main.go
package main

import "testing"

const N = 16 * 1024  // 16384
const M = 128 * 1024 // 131072
var x byte = 123

func new16384() byte {
   // new([65535]byte) escapes to heap
   var a = new([N+1]byte)
   for i := range a { a[i] = x }
   return a[x]
}

func comptr16384() byte {
   // &[65535]byte{} escapes to heap
   a := &[N+1]byte{}
   for i := range a { a[i] = x }
   return a[x]
}
```

```
func make16384() byte {
   // make([]byte, N + 1) escapes to heap
   a := make([]byte, N+1)
   for i := range a { a[i] = x }
   return a[x]
}

func declare131073() byte {
   var a [M+1]byte
   for i := range a { a[i] = x }
   return a[x]
}

func main() {
   stat := func( f func() byte ) int {
      allocs := testing.AllocsPerRun(1, func() {
         f()
      })
      return int(allocs)
   }

   // All print 1 if "-smallframes" is specified,
   // and print 0 if "-smallframes" is unspecified,
   println(stat(new16384))
   println(stat(comptr16384))
   println(stat(make16384))
   println(stat(declare131073))
}
```

If we use the command `go run main.go` to run the program, then none of the checked functions will make allocations on heap.

Allocate the backing array of a slice on stack even if its size is larger than or equal to 64K (but not larger than 10M)

We have learned that the largest slice backing array which could be allocated on stack is 65536 (or 65535 before Go toolchain v1.17). But there is a tip to raise the limit to 10M: derive slices from a stack allocated array. For example, the elements of the slice `s` created in the following program are allocated on stack. The length of `s` is 10M, which is far larger than 65536.

```
package main

import "testing"

const N = 10 * 1024 * 1024 // 10M
var n = N/2

func f() byte {
   var a [N]byte // allocated on stack
   for i := range a {
      a[i] = byte(i)
   }
   var s = a[:] // a slice with 10M elements
   var p = &a
   return s[n] + p[n]
}

func main() {
   stat := func( f func() byte ) int {
      allocs := testing.AllocsPerRun(1, func() {
         f()
```

```
        })
        return int(allocs)
    }
    println(stat(f)) // 0
}
```

As of Go toolchain v1.19, this tip still works.

Allocate the backing array of a slice with an arbitrary length on stack

It looks the escape analysis module in the current stanard Go compiler implementation (v1.19) misses the case of using composite literals to create slices. When using the composite literal way to create a slice, the elements of the slice will be always allocated on stack (if the compiler makes sure that the elements of the slice will be used by only one goroutine), regardless of the length of the slice and the size of the elements. For example, the following example program will allocate the 500M elements of a byte slice on stack.

```
package main

import "testing"

const N = 500 * 1024 * 1024 // 500M
var v byte = 123

func createSlice() byte {
    // []byte{...} does not escape
    var s = []byte{N: 0}
```

```
    for i := range s { s[i] = v }
    return s[v]
}

func main() {
    stat := func( f func() byte ) int {
        allocs := testing.AllocsPerRun(1, func() {
            f()
        })
        return int(allocs)
    }
    println(stat(createSlice)) // 0 (heap allocations)
}
```

Future Go toolchain versions might change the implementation so that this tip might not work later. But up to now (v1.19), this tip still works.

More tricks to allocate arbitrary-size values on stack

There are more corner cases in which values with arbitrary sizes will be allocated on stack. Similarly, future Go toolchain versions might (or might not) change the implementation details so that the allocation behaviors for these cases might change later.

All of the following `[N]byte` values are allocated on stack.

```
const N = 1024 * 1024 * 100

func boxLargeSizeValue() {
    type _ int // avoid being inlined
    var x interface{} = [N]byte{} // 1
    println(x != nil)
```

```
}

func largeSizeParameter(x [N]byte) { // 2
   type _ int // avoid being inlined
}

func largeSizeElement() {
   type _ int // avoid being inlined
   var m map[int][N]byte
   m[0] = [N]byte{} // 3
}
```

However, as what has been mentioned in the chapter before the last, to avoid large value copy costs, generally, we should not

- boxing large-size values into interfaces.
- use large-size types as function parameter types.
- use large-size types are map key and element types.

That is why the three cases shown in the above code snippet are corner cases. Generally, we should not write such code in practice.

Grow stack in less times

A stack growth includes two steps. The first step is to find a larger continuous memory segment as the new stack zone. The second step is to copy the used part on the old stack to the new one and adjust some pointer values if necessary. Comparatively, the second step is more costly than the first one.

Sometimes, during the execution of a goroutine, its stack will grow to a large size eventually, in several times. If we could confidently predict that this will happen and know the peek stack size, then we could use the trick shown in the following code to initialize the stack size to the peek size, so that multiple stack growths will be avoided during the execution of the goroutine (if).

The trick is shown in the following example.

```
// init-stack-size.go
package main

import "fmt"
import "time"

func demo(n int) byte {
   var a [8192]byte
   var b byte
   if n--; n > 0 {
      b = demo(n)
   }
   return a[n] + b
}

func foo(c chan time.Duration) {
   start := time.Now()
   demo(8192)
   c <- time.Since(start)
}

func bar(c chan time.Duration) {
   start := time.Now()
```

```
    // Use a dummy anonymous function to enlarge stack.
    func(x *interface{}) {
        type _ int // avoid being inlined
        if x != nil {
            *x = [1024 * 1024 * 64]byte{}
        }
    }(nil)
    demo(8192)
    c <- time.Since(start)
}

func main() {
    var c = make(chan time.Duration, 1)
    go foo(c)
    fmt.Println("foo:", <-c)
    go bar(c)
    fmt.Println("bar:", <-c)
}
```

Run it:

```
$ go run init-stack-size.go
foo: 42.051369ms
bar: 4.740523ms
```

From the outputs, we could get that the efficiency of the `bar` goroutine is higher than the `foo` goroutine. The reason is simple, only one stack growth happens in the lifetime of the `bar` goroutine, whereas more than 10 stack growths happen in the lifetime of the `foo` goroutine.

The official standard Go compiler decides to let a `[1024 * 1024 * 64]byte` value allocated on the stack of the `bar` goroutine (in fact, this will

not happen) and calculate the frame size of the dummy anonymous function as 67108888 bytes (larger than 64 MiB), so the invocation of the anonymous function makes the stack of the `bar` goroutine grow to the peak size (128 MiB) before calling the `demo` function. Assuming no garbage collections happen during the lifetime of the goroutine, the stack doesn't need to grow any more.

At the beginning of the goroutine lifetime, the size of the used part of the stack is still small, so the stack copy cost is small. Along with the used part becomes larger and larger, a stack growth will copy more and more memory. So the sum of the stack copy costs in the `foo` goroutine is much larger than the one single stack copy cost in the `bar` goroutine.

Garbage Collection

This chapter doesn't plan to explain the Garbage Collection (GC) implementation made by the official standard Go runtime in detail. Only several facts in the GC implementation will be touched.

Within a GC cycle, there is a scan+mark phase and a sweep phase.

During the scan+mark phase of a GC cycle, the garbage collector will scan all pointers in the already-known alive value parts to find more alive value parts (referenced by those already-known alive ones), until no more alive value parts need to be scanned. In the process, all heap memory blocks hosting alive value parts are marked as non-garbage memory blocks.

During the sweep phase of a GC cycle, the heap memory blocks which are not marked as non-garbage will be viewed as garbage and collected.

So, generally speaking, the more pointers are used, the more pressure is made for GC (for the more scan work to do).

GC pacer

At run time, a new GC cycle will start automatically when certain conditions are reached. The current automatic GC pacer design includes the following scheduling strategies:

- When the scan+mark phase of a GC cycle is just done, Go runtime will calculate a target heap size by using the configured `GOGC` percentage value. After the GC cycle, when the size of the heap (approximately) exceeds the target heap size later, the next GC cycle will automatically start. This strategy (called new heap memory percentage strategy below) will be described with more details in a later section.
- A new GC cycle will also automatically start if the last GC cycle has ended for at least two minutes. This is important for some value finalizers to get run and some goroutine stacks get shrunk in time.
- Go toolchain 1.19 introduced a new scheduling strategy: the memory limit strategy. When the total amount of memory Go runtime uses (approximately) surpasses the (soft) limit, a new GC cycle will start.

(Note: The above descriptions are rough. The official standard Go runtime also considers some other factors in the GC pacer implementation, which is why the above descriptions use the "approximately" wording.)

The three strategies may take effects at the same time.

The second strategy is an auxiliary strategy. This article will not talk more about it. With the other two strategies in play, the more frequently memory blocks are allocated on heap, the more frequently GC cycles start.

Please note that, the current GC pacer design is not promised to be perfect for every use cases. It will be improved constantly in future official standard Go compiler/runtime versions.

Automatic GC might affect Go program execution performance

GC cycles will consume some CPU resources, so they might affect Go program execution performance to some extend, depending on how program code is written. Generally, if the code of a program is not written very badly, we don't need to worry about the performance impact caused by GC, for GC cycles only consume a small fragment of the CPU resources consumed by the main program logic. Comparing to the conveniences brought by automatic GC, the small performance loss is not a big deal.

However, if some code of a program is written in a bad way which makes GC cycles busy (run frequently and scan a large quantity of pointers), then GC might affect program execution performance seriously.

How to reduce GC pressure?

No matter how the GC pacer is (and will be) designed, the basic principles to reduce GC pressure include:

- allocate less short-lived memory blocks on heap (garbage production speed control).
- create less pointers (scan work load control).

When a local value part is allocated on heap, an implicit pointer is created on stack to reference the local value part allocated on heap so that the value part is traceable (from stacks). This means reducing heap allocations is helpful to reduce pointers.

Heap memory allocations and pointer creations are inevitable in programming. If we need, just do it. We don't need to deliberately avoid using pointers with a lot of efforts.

In fact, pointers could be often used to reduce copy costs. So whether or not some pointers should be used depends on specific situations. We should only avoid using pointers in the situations where their drawbacks are obvious.

We should try to avoid unnecessary memory allocations, especially unnecessary heap memory allocations. However, admittedly, for many reasons, such as language safety and API design restrictions, unnecessary memory allocations are hard to be avoided totally. Go is not C, we should acknowledge and accept the fact. After all, Go provides much more conveniences than C.

Previous chapters have introduced some tips to avoid allocations. The coming chapters will introduce more.

Memory fragments

With memory blocks being allocated and collected from time to time, memory will become fragmented. The current official standard Go runtime implementation manages memory in an effective way (based on tcmalloc), so generally memory fragmenting is not a problem in Go programming.

This book doesn't provide suggestions to avoid memory fragments.

Memory wasting caused by sharing memory blocks

Sometimes, multiple value parts are carried on the same memory block (allocated on heap). The memory block could be only collected when all these value parts are become unused. In other words, a tiny alive value part carried on a memory block will prevent the memory block from being collected, even if the tiny value part only occupies a small portion of the memory block.

Generally, such situations happen in the following cases:

- several elements of a slice (or array or string) are still alive but others become unused.
- several fields of a struct value are still alive but others become unused.
- a hybrid of the above two.

The following is an example which shows one such situation. In the example, the 1000 elements of the slice `s` are all carried on the same memory block.

```
package main

import (
   "log"
   "runtime"
   "time"
)

func main() {
```

```
    log.SetFlags(0)

    var s = make([]int, 1000)
    for i := range s {
        s[i] = i
        runtime.SetFinalizer(&s[i], func(v interface{}) {
            log.Println("element", *v.(*int), "is collected")
        })
    }

    var p = &s[999]

    runtime.GC()

    // log.Println(*p) // 999
    _ = p
    time.Sleep(time.Second)
}
```

Run this program, it will print:

```
element 999 is collected
element 998 is collected
element 997 is collected
...
element 1 is collected
element 0 is collected
```

The output indicates that the a new GC cycle (triggered by the manual `runtime.GC` call) makes the memory block carrying the slice elements collected, otherwise the fanalizers of these elements will not get executed.

Let's turn on the `log.Println(*p)` line and run the program again, then it will merely print `999`, which indicates the memory block carrying the slice elements is still not collected when the manual GC cycle ends. Yes, the fact that the last element in the slice will still be used prevents the whole memory block from being collected.

We may copy the long-lived tiny value part (so that it will be carried on a small memory block) to let the old larger memory block collected. For example, we may replace the following line in the above program (with the `log.Println(*p)` line turned on):

```
var p = &a[999]
```

with

```
var v = a[999] // make a copy
var p = &v
```

, then the manual GC will make the memory block carrying the slice elements collected.

The return results of some functions in the standard `strings` (or `bytes`) packages are substrings (or subslices) of some arguments passed to these functions. Such functions include `Fields`, `Split` and `Trim` functions. Similarly, we should duplicate such a result substring (or subslice) if it is long lived but its length is small, and the corresponding argument is short lived but has a large length; otherwise, the memory block carrying the underlying bytes of (the result and the argument) will not get collected in time.

Try to generate less short-lived memory blocks to lower automatic GC frequency

In practice, most garbage comes from short-lived temporary value parts. Such short-lived value parts are often allocated in the following scenarios:

- string concatenations. Many concatenated result strings are passed as arguments of some function calls, then they will be discarded immediately. For some cases, to produce the final argument strings, many intermediate temporary strings are created.
- conversions between string and byte slice. The conversion results will be used then discarded soon. For safety, such a conversion needs to duplicate the underlying bytes. A memory block allocation is often needed for the duplication.
- box non-interface values into interfaces. The interfaces will be used then discarded soon. Most value boxing operations need one allocation.

There are also some other scenarios in which short-lived value parts will be generated.

In programming, we should try to generate less short-lived value parts to lower automatic GC frequency.

Use new heap memory percentage strategy to control automatic GC frequency

The value parts which contain pointers and are known clearly alive by Go runtime are called (scannable) root value parts (call roots below). Roots mainly come from the value parts allocated on stack and the direct parts of global (package-level) variables. To simplify the implementation, the official Go runtime views all the value parts allocated on stack as roots, including the value parts which don't contain pointers.

The memory blocks hosting roots are called root memory blocks. For the official standard Go runtime, before version 1.18, roots have no impact on the new heap memory percentage strategy; since version 1.18, they have.

The new heap memory percentage strategy is configured through a `GOGC` value, which may be set via the `GOGC` environment variable or modified by calling the `runtime/debug.SetGCPercent` function. The default value of the `GOGC` value is `100`.

We may set the `GOGC` environment variable as `off` or call the `runtime/debug.SetGCPercent` function with a negative argument to disable the new heap memory percentage strategy. Note: doing these will also disable the two-minute auto GC strategy. To disable the new heap memory percentage strategy solely, we may set the `GOGC` value as `math.MaxInt64`.

If the new heap memory percentage strategy is enabled (the `GOGC` value is non-negative), when the scan+mark phase of a GC cycle is just done, the official standard Go runtime (version 1.18+) will calculate the target heap size for the next garbage collection cycle, from the non-garbage heap

memory total size (call live heap here) and the root memory block total size (called GC roots here), according to the following formula:

```
Target heap size = Live heap + (Live heap + GC roots) * GOGC / 100
```

For versions before 1.18, the formula is:

```
Target heap size = Live heap + (Live heap) * GOGC / 100
```

When heap memory total size (approximately) exceeds the calculated target heap size, the next GC cycle will start automatically.

Note: the minimum target heap size is `(GOGC * 4 / 100)MB`, which is also the target heap size for the first GC cycle.

We could use the `gctrace=1` GODEBUG environment variable option to output a summary log line for each GC cycle. Each GC cycle summary log line is formatted like the following text (in which each `#` presents a number and `...` means ignored messages by the current article).

```
gc # @#s #%: ..., #->#->#  MB, # MB goal,# MB stacks,# MB globals,...
```

The meaning of each field:

```
gc #          the GC number, incremented at each GC
@#s           escaped time in seconds since program start
#%            percentage of time spent in GC since program start
#->#-># MB    heap sizes at GC start, scan end, and sweep end
# MB goal     target heap size
```

```
# MB stacks  estimated scannable stack size
# MB globals scannable global size
```

Note: in the following outputs of examples, to keep the each output line short, not all of these fields will be shown.

Let's use an unreal program as an example to show how to use this option.

```
// gctrace.go
package main

import (
   "math/rand"
   "time"
)

var x [512][]*int

func garbageProducer() {
   rand.Seed(time.Now().UnixNano())

   for i := 0; ; i++ {
      n := 6 + rand.Intn(6)
      for j := range x {
         x[j] = make([]*int, 1<<n)
         for k := range x[j] {
            x[j][k] = new(int)
         }
      }
      time.Sleep(time.Second / 1000)
   }
}
```

```
func main() {
   garbageProducer() // never exit
}
```

Run it with the `gctrace=1` GODEBUG environment variable option (several unrelated starting lines are omitted in the outputs).

```
$ GODEBUG=gctrace=1 go run gctrace.go
...
gc 1 @0.017s 8%: ..., 3->4->4 MB, 4 MB goal, ...
gc 2 @0.037s 8%: ..., 7->8->4 MB, 8 MB goal, ...
gc 3 @0.064s 13%: ..., 8->9->4 MB, 9 MB goal, ...
gc 4 @0.108s 10%: ..., 9->9->0 MB, 10 MB goal, ...
gc 5 @0.127s 10%: ..., 3->4->3 MB, 4 MB goal, ...
gc 6 @0.155s 10%: ..., 6->7->2 MB, 8 MB goal, ...
gc 7 @0.175s 10%: ..., 5->5->4 MB, 5 MB goal, ...
gc 8 @0.206s 10%: ..., 8->8->3 MB, 9 MB goal, ...
gc 9 @0.232s 10%: ..., 5->6->4 MB, 6 MB goal, ...
gc 10 @0.269s 12%: ..., 8->10->10 MB, 9 MB goal, ...
...
```

(On Windows, the command should be `set "GODEBUG=gctrace=1" & "go run gctrace.go"`.)

Here, the `#->#-># MB` and `# MB goal` fields are what we have most interests in. In a `#->#-># MB` field,

- the last number is non-garbage heap memory total size (a.k.a. live heap).
- the first number is the heap size at a GC cycle start, which should be approximately equal to the target heap size (the number in the `# MB`

`goal` field of the same line).

From the outputs, we could find that

- the live heap sizes stagger much (yes, the above program is deliberately designed as such), so the GC cycle intervals also stagger much.
- the GC cycle frequency is so high that the percentage of time spent on GC is too high. The above outputs show the percentage varies from 8% to 12%.

The reason of the findings is that the live heap size is small (staying roughly under 5MiB) and staggers much, but the root memory block total size is almost zero.

One way to reduce the time spent on GC is to increase the `GOGC` value:

```
$ GOGC=1000 GODEBUG=gctrace=1 go run gctrace.go
...
gc 1 @0.074s 2%: ..., 38->43->15 MB, 40 MB goal, ...
gc 2 @0.810s 1%: ..., 160->163->9 MB, 167 MB goal, ...
gc 3 @1.285s 1%: ..., 105->107->11 MB, 109 MB goal, ...
gc 4 @1.835s 1%: ..., 125->128->10 MB, 129 MB goal, ...
gc 5 @2.331s 1%: ..., 114->117->18 MB, 118 MB goal, ...
gc 6 @3.250s 1%: ..., 199->201->8 MB, 204 MB goal, ...
gc 7 @3.703s 1%: ..., 96->98->18 MB, 100 MB goal, ...
gc 8 @4.580s 1%: ..., 201->204->10 MB, 207 MB goal, ...
gc 9 @5.111s 1%: ..., 118->119->3 MB, 122 MB goal, ...
gc 10 @5.306s 1%: ..., 43->43->4 MB, 44 MB goal, ....
...
```

(On Windows, the command should be `set "GOGC=1000" & set "GODEBUG=gctrace=1" & "go run gctrace.go"`.)

From the above outputs, we could find that, after increase the `GOGC` value to 1000, GC cycle intervals and the heap sizes at GC cycle beginning both become much larger now. But GC cycle intervals and live heap sizes still stagger much, which might be a problem for some programs. The following sections will introduce some ways to solve the problems.

Since Go toolchain 1.18, the larger GC roots, the larger GC cycle intervals

The following program will create a goroutine with a 150MiB real size at run time. The size of the stack memory segment is 256MiB.

```
// bigstacks.go
package main

import (
   "math/rand"
   "time"
)

var x [512][]*int

func garbageProducer() {
   rand.Seed(time.Now().UnixNano())

   for i := 0; ; i++ {
      n := 6 + rand.Intn(6)
      for j := range x {
```

```
            x[j] = make([]*int, 1<<n)
            for k := range x[j] {
               x[j][k] = new(int)
            }
         }
         time.Sleep(time.Second / 1000)
      }
}

func bigStack(c chan int, v byte) byte {
   defer func() {
      c <- 1
   }()
   var s = []byte{150 << 20: 0} // on stack
   for i := range s { s[i] = v }

   return s[v]
}

func main() {
   go bigStack(nil, 123)
   garbageProducer() // never exit
}
```

Run it, get the following outputs:

```
$ go version
go version go1.19 linux/amd64

$ GODEBUG=gctrace=1 go run bigstacks.go
...
gc 1 @0.018s 22%: ..., 3->3->1 MB, 4 MB goal, 256 MB stacks,
...
```

```
gc 2 @0.565s 6%: ..., 148->150->16 MB, 152 MB goal, 256 MB
stacks, ...
gc 3 @1.360s 3%: ..., 178->180->17 MB, 183 MB goal, 256 MB
stacks, ...
gc 4 @2.136s 2%: ..., 180->181->4 MB, 184 MB goal, 256 MB
stacks, ...
gc 5 @2.812s 1%: ..., 155->155->2 MB, 159 MB goal, 256 MB
stacks, ...
gc 6 @3.511s 1%: ..., 150->152->14 MB, 154 MB goal, 256 MB
stacks, ...
gc 7 @4.280s 1%: ..., 174->176->8 MB, 178 MB goal, 256 MB
stacks, ...
gc 8 @5.007s 1%: ..., 163->163->1 MB, 167 MB goal, 256 MB
stacks, ...
gc 9 @5.665s 1%: ..., 150->155->21 MB, 154 MB goal, 256 MB
stacks, ...
gc 10 @6.459s 1%: ..., 182->183->4 MB, 193 MB goal, 256 MB
stacks, ...
...
```

The outputs show that GC cycle intervals become larger and much less staggering (with Go toolchain version 1.18+).

In calculating the root memory block total size, the official standard runtime version 1.18 uses stack memory segment sizes, whereas version 1.19+ use the real sizes of stacks (however, the numbers shown in the `# MB stacks` elements are still stack memory segment sizes). So, when using Go toolchain 1.18 to run the above example program, the target heap sizes will be some larger:

```
$ go version
go version go1.18.5 linux/amd64

$ GODEBUG=gctrace=1 go run bigstacks.go
...
gc 1 @0.015s ..., 4 MB goal, 256 MB stacks, ...
gc 2 @1.597s ..., 263 MB goal, 256 MB stacks, ...
gc 3 @2.772s ..., 258 MB goal, 256 MB stacks, ...
gc 4 @3.945s ..., 258 MB goal, 256 MB stacks, ...
gc 5 @5.239s ..., 288 MB goal, 256 MB stacks, ...
gc 6 @6.122s ..., 264 MB goal, 256 MB stacks, ...
gc 7 @7.384s ..., 279 MB goal, 256 MB stacks, ...
gc 8 @8.796s ..., 288 MB goal, 256 MB stacks, ...
gc 9 @10.102s ..., 291 MB goal, 256 MB stacks, ...
gc 10 @10.951s ..., 262 MB goal, 256 MB stacks, ...
...
```

The following program uses a huge-size package-level variable which contains a pointer, so that the root memory block total size of the program is also huge.

```
// bigglobals.go
package main

import (
   "math/rand"
   "time"
)

var x [512][]*int

func garbageProducer() {

```

```
   rand.Seed(time.Now().UnixNano())

   for i := 0; ; i++ {
      n := 6 + rand.Intn(6)
      for j := range x {
         x[j] = make([]*int, 1<<n)
         for k := range x[j] {
            x[j][k] = new(int)
         }
      }
      time.Sleep(time.Second / 1000)
   }
}

var bigGlobal struct {
   p *int
   _ [150 << 20]byte
}

func main() {
   garbageProducer()    // never exit
   println(bigGlobal.p) // unreachable
}
```

Run it, get the following outputs:

```
$ GODEBUG=gctrace=1 go run bigglobals.go
...
gc 1 @0.266s 0%: ..., 150 MB goal, ..., 150 MB globals, ...
gc 2 @0.885s 0%: ..., 154 MB goal, ..., 150 MB globals, ...
gc 3 @1.616s 0%: ..., 168 MB goal, ..., 150 MB globals, ...
gc 4 @2.327s 0%: ..., 165 MB goal, ..., 150 MB globals, ...
gc 5 @3.011s 0%: ..., 168 MB goal, ..., 150 MB globals, ...
```

```
gc 6 @3.493s 0%: ..., 155 MB goal, ..., 150 MB globals, ...
gc 7 @4.208s 0%: ..., 167 MB goal, ..., 150 MB globals, ...
gc 8 @4.897s 0%: ..., 162 MB goal, ..., 150 MB globals, ...
gc 9 @5.618s 0%: ..., 165 MB goal, ..., 150 MB globals, ...
gc 10 @6.338s 0%: ..., 169 MB goal, ..., 150 MB globals, ...
...
```

The same, the outputs show that GC cycle intervals become larger and much less staggering (with Go toolchain version 1.18+).

The examples in the current section show that root memory blocks objectively act as memory ballasts.

The next section will introduce a memory ballast trick which works with Go toolchain version 1.18-.

Use memory ballasts to avoid frequent GC cycles

The article Go memory ballast introduces a trick to control GC cycles in a much less staggering pace. The trick is like:

```
import "runtime"

func main() {
   // ballastSize is value much larger than the
   // maximum possible live heap size of the program.
   ballast := make([]byte, ballastSize)

   programRun()

   runtime.KeepAlive(&ballast)
}
```

The trick allocate a slice with a big element sum size. The size contributes to non-garbage heap size.

Let's modify the `gctrace` example shown above as:

```
// gcballast.go
package main

import (
   "math/rand"
   "runtime"
   "time"
)

var x [512][]*int

func garbageProducer() {
   rand.Seed(time.Now().UnixNano())

   for i := 0; ; i++ {
      n := 6 + rand.Intn(6)
      for j := range x {
         x[j] = make([]*int, 1<<n)
         for k := range x[j] {
            x[j][k] = new(int)
         }
      }
      time.Sleep(time.Second / 1000)
   }
}

func main() {
   const ballastSize = 150 << 20 // 150 MiB
```

```
    ballast := make([]byte, ballastSize)

    garbageProducer()

    runtime.KeepAlive(&ballast)
}
```

This program uses a 150MiB memory ballast, so that the non-garbage heap size (live heap) of the program keeps at about 150-160MiB. Consequently, the target heap size of the program keeps at a bit over 300MiB (assume the `GOGC` value is `100`). This makes GC cycle intervals become larger and much less staggering.

Run it to verify the effect:

```
$ GODEBUG=gctrace=1 go run gcballast.go
...
gc 1 @0.005s 0%: ..., 150->150->150 MB, 4 MB goal, ...
gc 2 @0.333s 5%: ..., 255->261->171 MB, 300 MB goal, ...
gc 3 @0.989s 2%: ..., 293->296->161 MB, 344 MB goal, ...
gc 4 @1.554s 1%: ..., 276->276->152 MB, 324 MB goal, ...
gc 5 @2.065s 1%: ..., 260->261->159 MB, 305 MB goal, ...
gc 6 @2.595s 1%: ..., 271->271->152 MB, 318 MB goal, ...
gc 7 @3.082s 1%: ..., 259->267->173 MB, 305 MB goal, ...
gc 8 @3.708s 1%: ..., 295->296->155 MB, 347 MB goal, ...
gc 9 @4.214s 1%: ..., 265->266->153 MB, 311 MB goal, ...
gc 10 @4.731s 1%: ..., 262->262->156 MB, 307 MB goal, ...
...
```

Note: the elements of the local slice is never used, so the allocated memory block for the elements is only allocated virtually, not physically (at least on

Linux). This means the elements of the slice don't consume physical memory, which is an advantage over using root memory blocks as memory ballasts.

Use Go toolchain 1.19 introduced memory limit strategy to avoid frequent GC cycles

Go official standard compiler 1.19 introduced a new scheduling strategy: the memory limit strategy.. The strategy may be configured either via the `GOMEMLIMIT` environment variable or through the runtime/debug.SetMemoryLimit function. This memory limit sets a maximum on the total amount of memory that the Go runtime should use. In other words, if the total amount of memory Go runtime uses (approximately) surpasses the limit, a new garbage collection process will start. The limit is soft, a Go program will not exit when this limit is exceeded. The default value of the memory limit is `math.MaxInt64`, which effectively disables this strategy.

The value of the `GOMEMLIMIT` environment variable may have an optional unit suffix. The supported suffixes include B, KiB, MiB, GiB, and TiB. A value without a unit means B (bytes).

The memory limit strategy and the new heap memory percentage strategy may take effect together. For demonstration purpose, let's disable the new heap memory percentage strategy and enable the memory limit strategy to run the `gctrace` example program shown above again. Please make sure to run the program with Go toolchain 1.19+; otherwise, the `GOMEMLIMIT`

environment variable will not get recognized so that automatic garbage collection will be turned off totally.

```
$ go version
go version go1.19 linux/amd64

$ GOMEMLIMIT=175MiB GOGC=off GODEBUG=gctrace=1 go run
gctrace.go
gc 1 @0.283s 1%: ..., 153->155->18 MB, 157 MB goal, ...
gc 2 @0.784s 1%: ..., 151->152->6 MB, 155 MB goal, ...
gc 3 @1.445s 1%: ..., 149->151->7 MB, 152 MB goal, ...
gc 4 @2.113s 1%: ..., 148->150->8 MB, 152 MB goal, ...
gc 5 @2.765s 1%: ..., 148->150->18 MB, 152 MB goal, ...
gc 6 @3.394s 1%: ..., 147->150->18 MB, 152 MB goal, ...
gc 7 @4.006s 1%: ..., 147->151->19 MB, 152 MB goal, ...
gc 8 @4.653s 1%: ..., 147->147->1 MB, 152 MB goal, ...
gc 9 @5.326s 1%: ..., 148->152->14 MB, 152 MB goal, ...
gc 10 @6.007s 1%: ..., 148->149->9 MB, 152 MB goal, ...
...
```

From the outputs, we could find that, the memory limit strategy can also make GC cycle intervals become much larger and less staggering.

One goal of the memory limit strategy to provide an official way to replace the memory ballast trick. For some Go programs, if the memory limit values are set properly, the goal will be achieved. However, sometimes, it might be difficult to choose a proper value for the limit. If the value is set smaller than the amount memory of the Go program really needs, then GC cycles will start frequently. This is a disadvantage compared to the memory ballast solutions shown above. Please read the suggestions for using the

memory limit strategy (that containing article is an excellent article to understand garbage collection in Go).

Pointers

Avoid unnecessary nil array pointer checks in a loop

There are some flaws in the current official standard Go compiler implementation (v1.19). One of them is some nil array pointer checks are not moved out of loops. Here is an example to show this flaw.

```
// unnecessary-checks.go
package pointers

import "testing"

const N = 1000
var a [N]int

//go:noinline
func g0(a *[N]int) {
   for i := range a {
      a[i] = i // line 12
   }
}

//go:noinline
func g1(a *[N]int) {
   _ = *a // line 18
   for i := range a {
      a[i] = i // line 20
   }
```

```
}

func Benchmark_g0(b *testing.B) {
   for i := 0; i < b.N; i++ { g0(&a) }
}

func Benchmark_g1(b *testing.B) {
   for i := 0; i < b.N; i++ { g1(&a) }
}
```

Let's run the benchmarks with the `-S` compiler option, the following outputs are got (uninterested texts are omitted):

```
$ go test -bench=. -gcflags=-S unnecessary-checks.go
...
0x0004 00004 (unnecessary-checks.go:12) TESTB     AL, (AX)
0x0006 00006 (unnecessary-checks.go:12) MOVQ CX, (AX)(CX*8)
...
0x0000 00000 (unnecessary-checks.go:18) TESTB     AL, (AX)
0x0002 00002 (unnecessary-checks.go:18) XORL CX, CX
0x0004 00004 (unnecessary-checks.go:19) JMP  13
0x0006 00006 (unnecessary-checks.go:20) MOVQ CX, (AX)(CX*8)
...
Benchmark_g0-4  517.6 ns/op
Benchmark_g1-4  398.1 ns/op
```

From the outputs, we could find that the `g1` implementation is more performant than the `g0` implementation, even if the `g1` implementation contains one more code line (line 18). Why? The question is answered by the outputted assembly instructions.

In the `g0` implementation, the `TESTB` instruction is generated within the loop, whereas in the `g1` implementation, the `TESTB` instruction is generated out of the loop. The `TESTB` instruction is used to check whether or not the argument `a` is a nil pointer. For this specified case, checking once is enough. The one more code line avoids the flaw in the compiler implementation.

There is a third implementation which is as performant as the `g1` implementation. The third implementation uses a slice derived from the array pointer argument.

```
//go:noinline
func g2(x *[N]int) {
   a := x[:]
   for i := range a {
      a[i] = i
   }
}
```

Please note that the flaw might be fixed in future compiler versions.

And please note that, if the three implementation functions are inline-able, the benchmark results will change much. That is the reason why the `//go:noinline` compiler directives are used here. (Before Go toolchain v1.18, the `//go:noinline` compiler directives are actually unnecessary here. Because Go toolchain v1.18- never inlines a function containing a `for-range` loop.)

The case in which an array pointer is a struct field

For the cases in which an array pointer is a struct field, things are a little complex. The `_ = *t.a` line in the following code is useless to avoid the compiler flaw. For example, in the following code, the performance difference between the `f1` function and the `f0` function is small. (In fact, the `f1` function might be even slower, if a `NOP` instruction is generated within its loop.)

```
type T struct {
   a *[N]int
}

//go:noinline
func f0(t *T) {
   for i := range t.a {
      t.a[i] = i
   }
}

//go:noinline
func f1(t *T) {
   _ = *t.a
   for i := range t.a {
      t.a[i] = i
   }
}
```

To move the nil array pointer checks out of the loop, we should copy the `t.a` field to a local variable, then adopt the trick introduced above:

```
//go:noinline
func f3(t *T) {
```

```
   a := t.a
   _ = *a
   for i := range a {
      a[i] = i
   }
}
```

Or simply derive a slice from the array pointer field:

```
//go:noinline
func f4(t *T) {
   a := t.a[:]
   for i := range a {
      a[i] = i
   }
}
```

The benchmark results:

```
Benchmark_f0-4  622.9 ns/op
Benchmark_f1-4  637.4 ns/op
Benchmark_f2-4  511.3 ns/op
Benchmark_f3-4  390.1 ns/op
Benchmark_f4-4  387.6 ns/op
```

The results verify our previous conclusions.

Note, the `f2` function mentioned in the benchmark results is declared as

```
//go:noinline
func f2(t *T) {
   a := t.a
   for i := range a {
```

```
        a[i] = i
    }
}
```

The `f2` implementation is not fast as the `f3` and `f4` implementations, but it is faster than the `f0` and `f1` implementations. However, that is another story.

If the elements of an array pointer field are not modified (only read) in the loop, then the `f1` way is as performant as the `f3` and `f4` way.

Personally, for most cases, I think we should try to use the slice way (the `f4` way) to get the best performance, because generally slices are optimized better than arrays by the official standard Go compiler.

Avoid unnecessary pointer dereferences in a loop

Sometimes, the current official standard Go compiler (v1.19) is not smart enough to generate assembly instructions in the most optimized way. We have to write the code in another way to get the best performance. For example, in the following code, the `f` function is much less performant than the `g` function.

```
// avoid-indirects_test.go
package pointers

import "testing"

//go:noinline
func f(sum *int, s []int) {
```

```
   for _, v := range s { // line 8
      *sum += v // line 9
   }
}

//go:noinline
func g(sum *int, s []int) {
   var n = *sum
   for _, v := range s { // line 16
      n += v // line 17
   }
   *sum = n
}

var s = make([]int, 1024)
var r int

func Benchmark_f(b *testing.B) {
   for i := 0; i < b.N; i++ {
      f(&r, s)
   }
}

func Benchmark_g(b *testing.B) {
   for i := 0; i < b.N; i++ {
      g(&r, s)
   }
}
```

The benchmark results (uninterested texts are omitted):

```
$ go test -bench=. -gcflags=-S avoid-indirects_test.go
...
```

```
0x0009 00009 (avoid-indirects_test.go:9)     MOVQ (AX), SI
0x000c 00012 (avoid-indirects_test.go:9)     ADDQ (BX)(DX*8),
SI
0x0010 00016 (avoid-indirects_test.go:9)     MOVQ SI, (AX)
0x0013 00019 (avoid-indirects_test.go:8)     INCQ DX
0x0016 00022 (avoid-indirects_test.go:8)     CMPQ CX, DX
0x0019 00025 (avoid-indirects_test.go:8)     JGT  9
...
0x000b 00011 (avoid-indirects_test.go:16)    MOVQ (BX)(DX*8),
DI
0x000f 00015 (avoid-indirects_test.go:16)    INCQ DX
0x0012 00018 (avoid-indirects_test.go:17)    ADDQ DI, SI
0x0015 00021 (avoid-indirects_test.go:16)    CMPQ CX, DX
0x0018 00024 (avoid-indirects_test.go:16)    JGT  11
...
Benchmark_f-4  3024 ns/op
Benchmark_g-4   566.6 ns/op
```

The outputted assembly instructions show the pointer `sum` is dereferenced within the loop in the `f` function. A dereference operation is a memory operation. For the `g` function, the dereference operations happen out of the loop, and the instructions generated for the loop only process registers. It is much faster to let CPU instructions process registers than process memory, which is why the `g` function is much more performant than the `f` function.

This is not a compiler flaw. In fact, the `f` and `g` functions are not equivalent (though for most use cases in practice, their results are the same). For example, if they are called like the following code shows, then they return different results (thanks to skeeto@reddit for making this correction).

```
{
   var s = []int{1, 1, 1}
   var sum = &s[2]
   f(sum, s)
   println(*sum) // 6
}
{
   var s = []int{1, 1, 1}
   var sum = &s[2]
   g(sum, s)
   println(*sum) // 4
}
```

Another performant implementation for this specified case is to move the pointer parameter out of the function body (again, it is not totally equivalent to either `f` or `g` function):

```
//go:noinline
func h(s []int) int {
   var n = 0
   for _, v := range s {
      n += v
   }
   return n
}

func use_h(s []int) {
   var sum = new(int)
   *sum += h(s)
   ...
}
```

Structs

Avoid accessing fields of a struct in a loop though pointers to the struct

It is much faster to let CPU instructions process registers than process memory. So, to let the compiler generate less memory-processing assembly instructions for a loop, we should avoid accessing fields of a struct though pointers to the struct in the loop.

For example, in the following code, the function `g` is much performant than the function `f`.

```
package structs

import "testing"

const N = 1000

type T struct {
   x int
}

//go:noinline
func f(t *T) {
   t.x = 0
   for i := 0; i < N; i++ {
      t.x += i
   }
```

```
}

//go:noinline
func g(t *T) {
   var x = 0
   for i := 0; i < N; i++ {
      x += i
   }
   t.x = x
}

var t = &T{}

func Benchmark_f(b *testing.B) {
   for i := 0; i < b.N; i++ {
      f(t)
   }
}

func Benchmark_g(b *testing.B) {
   for i := 0; i < b.N; i++ {
      g(t)
   }
}
```

The benchmark results:

```
Benchmark_f-4  2402 ns/op
Benchmark_g-4   461.3 ns/op
```

The function `g` uses a local variable `x` to store the sum value and assigns the sum value to the struct field in the end. The official standard Go

compiler is smart enough to only generate register-processing assembly instructions for the loop of the function `g`.

The function `f` is actually equivalent to the function `h` declared below.

```
//go:noinline
func h(t *T) {
   x := &t.x
   for i := 0; i < N; i++ {
      *x += i
   }
}
```

So the suggestion introduced here is actually equivalent to the one introduced in the last chapter (avoid unnecessary pointer dereferences in a loop).

Small-size structs are optimized specially

This has been talked about in value copy costs and small-size types/values.

Make struct size smaller by adjusting field orders

Struct field orders might affect struct sizes, which has been mentioned in a previous chapter.

Arrays and Slices

Avoid using literals of large-size array types as comparison operands

For example, in the following code, the function `CompareWithGlobalVar` is more performant than the function `CompareWithLiteral` (for the official standard Go compiler v1.19).

```
package arrays

import "testing"

type T [1000]byte

var zero = T{}

func CompareWithLiteral(t *T) bool {
   return *t == T{}
}

func CompareWithGlobalVar(t *T) bool {
   return *t == zero
}

var x T
var r bool

func Benchmark_CompareWithLiteral(b *testing.B) {
   for i := 0; i < b.N; i++ {
```

```
        r = CompareWithLiteral(&x)
    }
}

func Benchmark_CompareWithGlobalVar(b *testing.B) {
    for i := 0; i < b.N; i++ {
        r = CompareWithGlobalVar(&x)
    }
}
```

The benchmark results:

```
Benchmark_CompareWithLiteral-4    21214032  52.18 ns/op
Benchmark_CompareWithGlobalVar-4  36417091  31.03 ns/op
```

By using the `-S` compiler option, we could find that the compile generates less instructions for the function `CompareWithGlobalVar` than the function `CompareWithLiteral`. That is why the function `CompareWithGlobalVar` is more performant.

For small-size arrays, the performance difference between the two functions is small.

Please note that future compiler versions might be improved to remove the performance difference between the two functions.

Using slice-to-array-pointer conversions introduced in Go 1.17 to copy slices

Go 1.17 introduces a slice to array pointer conversion. We can make use of such conversions to copy slices. It is found that, on any platform, the

conversion way is more performant than the builtin `copy` function if the sum element sizes of the copied slices are not larger than 64.

```
package arrays

import "testing"

const N = 64

var r []byte
var s = make([]byte, N)

func init() {
   println("============= N =", N)
   r = make([]byte, N)
}

func copy2(d, s []byte) {
   *(*[N]byte)(d) = *(*[N]byte)(s)
}

func Benchmark_Copy(b *testing.B) {
   for i := 0; i < b.N; i++ {
      copy(r, s)
   }
}

func Benchmark_Copy2(b *testing.B) {
   for i := 0; i < b.N; i++ {
      copy2(r, s)
   }
}
```

The benchmark results:

```
============ N = 32
Benchmark_Copy-4   4.580 ns/op
Benchmark_Copy2-4  1.403 ns/op
============ N = 64
Benchmark_Copy-4   5.187 ns/op
Benchmark_Copy2-4  2.000 ns/op
```

On my machine (Linux/amd64 Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz), The `copy2` way is less performant only if the sizes of the slice backing array are in the range [250, 2250] or larger than 1MB. However, some people reported that the `copy2` way is always less performant for larger-size arrays. So if we want to use the `copy2` way to copy slices for performance reason, try to use it for slices which sum element sizes are not larger than 64.

The `make` and `append` builtin function implementations

Go is not C, so generally, the slice elements allocated by a `make` call will be all zeroed in the `make` call.

Since Go toolchain v1.15, the official standard Go compiler makes a special optimization: in the following alike code, the elements within `y[:m]`, where `m` is the return result of the `copy` call, will not be zeros in the `make` call, for they will be overwritten by the subsequent `copy` call.

```
    y = make([]T, n)
  copy(y, x) // assume the return value is m
```

The optimization is often used to clone slices:

```
    y = make([]T, len(x))
copy(y, x)
```

Up to now (Go toolchain v1.19), the optimization has not be implemented perfectly yet. In the following code, the elements within `s[len(x):]` will still get zeroed in the `make` call, which is actually unnecessarily.

```
    s = make([]T, len(x) + len(y))
copy(s, x)
copy(s[len(x):], y)
```

Another imperfection of this optimization is that several requirements must be satisified to make it work:

- the cloned slice must present as a pure or qualified identifier.
- the `make` call must only take two arguments.
- the `copy` call must not present as an expression in another statement.

In other words, the optimization only works for the first case in the following code:

```
    var s = make{[]T, 10000)

// case 1:
y = make([]T, len(s)) // works
copy(y, s)

// case 2:
y = make([]T, len(s)) // not work
```

```
_ = copy(y, s)

// case 3:
y = make([]T, len(s)) // not work
f(copy(y, s))

// case 4:
y = make([]T, len(s), len(s)) // not work
copy(y, s)

// case 5:
var a = [1][]T{s}
y = make([]T, len(a[0])) // not work
copy(y, a[0])

// case 6:
type SS struct {x []T}
var ss = SS{x: s}
y = make([]T, len(ss.x)) // not work
copy(y, ss.x)
```

The capacity of the result of a `make` call is exactly the argument passed to the `make` call. For example, `cap(make([]T, n) == n` and `cap(make([]T, n, m) == m`. This means there might be some bytes are wasted in the memory block hosting the elements of the result.

If an `append` call needs to allocate, then the capacity of the result slice of the `append` call is unspecified. The capacity is often larger than length of the result slice. Assume the result of the `append` call is assigned to a slice `s`, then the elements within `s[:len(s):cap(s)]` will get zeroed in the `append` call. The other elements will be overwritten by the elements of the

argument slices. For example, in the following code, the elements within `s[len(x)+len(y):]` will get reset in the `append` call.

```
    s = append(x, y...)
```

If the `append` call in the `new = append(old, values...)` statement allocates, then the capacity of the result slice `new` is determined by the following shown algorithm (assume `elementSize` is not zero) in Go 1.17:

```
    var newcap int
  var required = old.len + values.len
  if required > old.cap * 2 {
     newcap = required
  } else {
     if old.cap < 1024 {
        newcap = old.cap * 2
     } else {
        newcap = old.cap
        for 0 < newcap && newcap < required {
           newcap += newcap / 4
        }
        // Avoid overflowing.
        if newcap <= 0 {
           newcap = required
        }
     }
  }

  var memsize = newcap * elementSize

  ... // Adjust memsize upwards to a nearby memory block class
size.
```

```
   newcap = memsize / elementSize
```

The v1.17 algorithm has a drawback: the capacity of the result slice doesn't increase monotonically with the length of the first parameter of the `append` function. An example:

```
package main

func main() {
   x1 := make([]int, 897)
   x2 := make([]int, 1024)
   y := make([]int, 100)
   println(cap(append(x1, y...)))
   println(cap(append(x2, y...)))
}
```

With Go toolchain v1.17, the above example prints:

```
2048
1280
```

With Go toolchain v1.18+, the above example prints:

```
1360
1536
```

That is because Go 1.18 removes this drawback by tweaking the algorithm a bit:

```
     var newcap int
   var required = old.len + values.len
   if required > old.cap * 2 {
```

```
    newcap = required
} else {
    const threshold = 256
    if old.cap < threshold {
        newcap = old.cap * 2
    } else {
        newcap = old.cap
        for 0 < newcap && newcap < required {
            newcap += (newcap + 3*threshold) / 4
        }
        // Avoid overflowing.
        if newcap <= 0 {
            newcap = required
        }
    }
}
```

The new algorithm in Go 1.18+ often allocates less memory than the old one in Go 1.17.

Please note, each slice growth needs one memory allocation. So we should try to grow slices with less times in programming.

Try to clip the first argument of an `append` call if we know the call will allocate

If an `append` call allocates and there are no more elements to be appended to the result slice, then it is best to clip the first argument of the `append` call, to try to save some memory (and consume less CPU resources).

An example:

```
package main

func main() {
   x := make([]byte, 100, 500)
   y := make([]byte, 500)
   a := append(x, y...)
   b := append(x[:len(x):len(x)], y...)
   println(cap(a)) // 1024
   println(cap(b)) // 640
}
```

The outputs shown as comments are for Go 1.17. For Go 1.18, instead, the above program prints:

```
896
640
```

Surely, if we confidently know that the free capacity of the first argument slice of an `append` call is enough to hold all appended elements, then we should not clip the first argument.

Grow slices (enlarge slice capacities)

There are two ways to grow the capacity of a slice `x` to `c` if the backing array of the slice is needed to be re-allocated in the growth.

```
// way 1
func Grow_MakeCopy(x []T, c int) []T {
   r := make([]T, c)
   copy(r, x)
   return r[:len(x)]
}
```

```
// way 2
func Grow_Oneline(x []T, c int) []T {
   return append(x, make([]T, c - len(x))...)[:len(x)]
}
```

Both of the two ways are specially optimized by the official standard Go compiler. As mentioned above, the `make` call in way 1 doesn't reset elements within `r[:len(x)]`. In way 2, the `make` call doesn't make allocations at all.

In theory, with the two optimizations, the two ways have comparable performance. But benchmark results often show way 1 is a little more performant.

Please note, up to now (verison 1.19), in the current official standard Go compiler implementation, the optimization for way 2 doesn't work if the two argument slice types of the `make` call don't match. For example, the following `Grow_Oneline_Named` function is much slower than the above `Grow_Oneline` function.

```
func Grow_Oneline_Named(x []T, c int) []T {
   type S []T
   return append(x, make(S, c - len(x))...)[:len(x)]
}
```

This flaw will be fixed in Go toolchain v1.20.

Try to grow a slice in one step

As mentioned above, how slices grow in `append` calls is implementation specific, which means the result capacity of a slice growth is unspecified by Go specification.

If we could predict the max length of a slice at coding time, we should allocate the slice with the max length as its capacity, to avoid some possible future allocations caused by more slice growths.

If a slice is short-lived, then we could allocate it with an estimated large enough capacity. There might be some memory wasted temporarily, but the element memory will be released soon. Even if the estimated capacity is proved to be not large enough, there might still be several allocations saved.

Clone slices

Since Go toolchain version 1.15, the most efficient way to clone a slice is the `make+copy` way:

```
    sCloned = make([]T, len(s))
  copy(sCloned, s)
```

For many cases, the `make+copy` way is a little faster than the following `append` way, because as mentioned above, an `append` call might allocate and zero some extra elements.

```
    sCloned = append([]T(nil), s...)
```

For example, in the following code, 8191 extra elements are allocated and zeroed.

```
    x := make([]byte, 1<<15+1)
  y := append([]byte(nil), x...)
  println(cap(y) - len(x)) // 8191
```

Merge two slices

There is not a universally perfect way to merge two slices into a new slice with the current official standard Go compiler (up to Go v1.19).

If the element orders of the merged slice are important, we could use the following two ways to merge the slice `x` and `y` (assume the length of `y` is not zero).

```
    // The make+copy way
  merged = make([]T, len(x) + len(y))
  copy(merged, x)
  copy(merged[len(x):], y...)

  // The append way
  merged = append(x[:len(x):len(x)], y...)
```

The `append` way is clean but it is often a little slower, because the `append` function often allocates and zeroes some extra elements. But if the length of `y` is much larger than the length of `x`, then the `append` way is probably faster, because the elements within `merged[len(x):]` are (unnecessarily) zeroed in the `make+copy` way (then overridden by the elements of `y`). So, which way is more performant depends on specific situations.

If the element orders of the merged slice are not important and the `append` way is chosen, then try to pass the shorter slice as the first argument, so that

some memory might be saved. An example to show the fact:

```
package main

func main() {
   x := make([]int, 98)
   y := make([]int, 666)
   a := append(x, y...)
   b := append(y, x...)
   println(cap(a)) // 768
   println(cap(b)) // 1360
}
```

The outputs shown as comments are for Go 1.17. For Go 1.18, instead, the above program prints:

```
768
1024
```

If the free element slots in slice `x` are enough to hold all elements of slice `y` and it is allowed to let the result slice and `x` share elements, then `append(x, y...)` is the most performant way, for it doesn't allocate.

Merge more than two slices (into a new slice)

Up to now, there is not an extreme performant way to merge 2+ slices into a new slice. The reason is there will be always some elements (unnecessarily) get zeroed in the `make` call.

If the element orders of the merged slice are important, we could use the following implementations to do the merging. Their performances are

almost the same.

```
func MergeN_MakeCopy(ss ...[]byte) []byte {
   var n = 0
   for _, s := range ss {
      n += len(s)
   }
   i, r := 0, make([]byte, n)
   for _, s := range ss {
      copy(r[i:], s)
      i += len(s)
   }
   return r
}

func MergeN_MakeAppend(ss ...[]byte) []byte {
   var n = 0
   for _, s := range ss {
      n += len(s)
   }
   r := make([]byte, 0, n)
   for _, s := range ss {
      r = append(r, s...)
   }
   return r
}
```

If the element orders of the merged slice are not important, we could use the following way instead. This implementation tries to make use of the `make+copy` optimization to zero less elements in the `make` call.

```
func MergeN_Orderless(ss ...[]byte) []byte {
   if len(ss) == 0 {
      return []byte{}
   }
   var k, max = 0, len(ss[0])
   var n = max
   for i := 1; i < len(ss); i++ {
      li := len(ss[i])
      n += li
      if li > max {
         k = i
         max = li
      }
   }

   if k > 0 {
      ss2 := make([][]byte, len(ss))
      copy(ss2, ss)
      ss2[k], ss2[0] = ss2[0], ss2[k]
      ss = ss2
   }
   var s = ss[0]
   var r = make([]byte, n)
   copy(r, s)
   r = r[:max]

   for _, s := range ss[1:] {
      r =append(r, s...)
   }
   return r
}
```

Although the `MergeN_Orderless` implementation is more performant (often only a bit), it is quite verbose and not graceful. So please only adopt it if a high performance is desired strongly.

Insert a slice into another one

In theory, between the following two implementations of inserting one slice into another one, the `Insert2` implementation is always more performant than the `Insert1` implementation, because the `Insert2` implementation makes use of the `make+copy` optimization mentioned above, so that the elements within `s2[:k]` is not zeroed within the `make` call.

However, for a runtime implementation imperfection (in Go 1.15/1.16/1.17), if the length of the result slice is not larger than 32768, then the `Insert2` implementation might be actually slower. This imperfection has been fixed since Go 1.18.

```
func Insert1(s []byte, k int, vs []byte) []byte {
   s2 := make([]byte, len(s) + len(vs))
   copy(s2, s[:k])
   copy(s2[k:], vs)
   copy(s2[k+len(vs):], s[k:])
   return s2
}

func Insert2(s []byte, k int, vs []byte) []byte {
   a := s[:k]
   s2 := make([]byte, len(s) + len(vs))
   copy(s2, a)
   copy(s2[len(a):], vs)
```

```
    copy(s2[len(a)+len(vs):], s[k:])
    return s2
}
```

The one-line trick to insert a slice, `append(x1[:k:k], append(vs, x1[k:]...)...)`, is often very inefficient. It copies the elements within `x1[k:]` twice and often needs two allocations. This trick might be optimized specially later but has not yet.

If the free capacity of the base slice is large enough to hold all the inserted elements, and it is allowed to let the result slice and the base slice share elements, then the following way is the most efficient, for this way doesn't allocate.

```
    s = s[:len(s)+len(vs)]
  copy(s[i+len(vs):], s[i:])
  copy(s[i:], vs)
```

If the insertion operations are performed frequently, please consider using insertion-friendly data structure (such as linked list) instead.

Don't use the second iteration variable in a `for-range` loop if high performance is demanded

In the value parts and value sizes article, we have learned that we should avoid

- ranging over an array with two iteration variables if the size of the array is large, because the array is copied. (Note, if only one iteration variable is used, then the copy will not be made.)

- ranging over a slice or array with two iteration variables if the element size is large, because each element will be copied to the second iteration variable once in the iteration process.

We could get an intuitive understanding from the following equivalence relation:

```
    for i, v := range anArray {
     ...
  }
```

is equivalent to

```
    {
     aCopy := anArray  // all elements are copied
     var v ElementType // element type of anArray
     for i := range aCopy {
        v = anArray[i] // each element is copied
        ...
     }
  }
```

That means, if the second iteration variable in a `for-range` loop is effectively used when ranging an array, then each element of the array will be copied twice. (If the ranged container is a slice, then each element will be copied once.)

In fact, even if the element size of a slice is small, it is still comparatively less performant to use the second iteration variable in a `for-range` loop. This could be proved from the following benchmark code:

```
package arrays

import "testing"

//go:noinline
func sum_forrange1(s []int) int {
   var n = 0
   for i := range s {
      n += s[i]
   }
   return n
}

//go:noinline
func sum_forrange2(s []int) int {
   var n = 0
   for _, v := range s {
      n += v
   }
   return n
}

//go:noinline
func sum_plainfor(s []int) int {
   var n = 0
   for i := 0; i < len(s); i++ {
      n += s[i]
   }
   return n
}

var s = make([]int, 1<<16)
var r int
```

```
func Benchmark_forrange1(b *testing.B) {
   for i := 0; i < b.N; i++ {
      r = sum_forrange1(s)
   }
}

func Benchmark_forrange2(b *testing.B) {
   for i := 0; i < b.N; i++ {
      r = sum_forrange2(s)
   }
}

func Benchmark_plainfor(b *testing.B) {
   for i := 0; i < b.N; i++ {
      r = sum_plainfor(s)
   }
}
```

The benchmark results:

```
Benchmark_forrange1-4  34044  33793 ns/op
Benchmark_forrange2-4  31474  37819 ns/op
Benchmark_plainfor-4   34646  33704 ns/op
```

From the benchmark results, we could find that the plain `for` loop and the `for-range` loop with only one iteration variable are both more performant than the `for-range` loop which uses the second iteration variable.

Reset all elements of an array or slice

Since Go toolchain v1.5, a special optimization has been introduced, to efficiently zero the elements in an array or slice. Specifically, the following one-iteration-variable `for-range` loop will be the official standard Go compiler optimized as an internal `memclr` operation:

```
// v0 is the zero value literal (might be nil) of
// the element type of the container aSliceOrArray.
for i := range aSliceOrArray {
   aSliceOrArray[i] = v0
}
```

This optimization also works if two iteration variables present but the second one is the blank identifier `_`.

For most cases, the above code is more performant than the following code:

```
for i := 0; i < len(aSliceOrArray); i++ {
   aSliceOrArray[i] = v0
}
```

On my machine, the `memclr` way is slower only if the length of the array of slice is smaller than 6 (element type is `byte`).

Before Go toolchain v1.19, the ranged container must be an array or slice to make this optimization work. Since Go toolchain v1.19, it may be also a pointer to an array.

In fact, this optimization is more meaningful for slices than for arrays and array pointers, as there is a more simple (and sometimes more performant) way to reset array elements:

```
anArray = ArrayType{}
*anArrayPointer = ArrayType{}
```

Specify capacity explicitly in subslice expression

In Go, at compile time, if it is not obviously (for the compiler) to deduce the capacity of a subslice expression `s[a:b:c]` is non-zero, then the compiler will generate several extra CPU instructions, to prevent the underlying element pointer of the result subslice from pointing to the end boundary of the memory block carrying the elements of `s`. Because the end boundary might be the (start) address of another memory block, and pointing to it might prevent the other memory block from being garbage collected.

For an implementation imperfection, the current official standard Go compiler (v1.19) succeeds to deduce the capacity of `s[i:i+4:i+4]` is non-zero, but fails to deduce the capacity of `s[i:i+4]` is also non-zero. So the compiler will generate more CPU instructions for the latter. That means the `g` function is more performant than the `f` function in the following code.

```
package arrays

import "testing"

var N = 1 << 6
var s = make([]byte, N)
var r = make([]byte, N/4)

func f(rs []byte, bs []byte) {
   for i, j := 0, 0; i < len(bs) - 3; i += 4 {
      s2 := bs[i:i+4]
```

```
      rs[j] = s2[3] ^ s2[2] ^ s2[1] ^ s2[0]
      j++
   }
}

func g(rs []byte, bs []byte) {
   for i, j := 0, 0; i < len(bs) - 3; i += 4 {
      s2 := bs[i:i+4:i+4]
      rs[j] = s2[3] ^ s2[2] ^ s2[1] ^ s2[0]
      j++
   }
}

func Benchmark_f(b *testing.B) {
   for i := 0; i < b.N; i++ {
      f(r, s)
   }
}

func Benchmark_g(b *testing.B) {
   for i := 0; i < b.N; i++ {
      g(r, s)
   }
}
```

The benchmark results:

```
Benchmark_f-4  32603492  32.90 ns/op
Benchmark_g-4  42112129  25.68 ns/op
```

Please note that, this implementation imperfection might be improved in future versions of the official standard Go compiler.

Use index tables to save some comparisons

In the following code, the `bar` function is more performant than the `foo` function.

```
func foo(n int) {
   switch n % 10 {
   case 1, 2, 6, 7, 9:
      // do something 1
   default:
      // do something 2
   }
}

var indexTable = [10]bool {
   1: true, 2: true, 6: true, 7: true, 9: true,
}

func bar(n int) {
   switch {
   case indexTable[n % 10]:
      // do something 1
   default:
      // do something 2
   }
}
```

The `foo` function needs to make one to five comparisons before entering a branch code block, whereas the `bar` function always needs only one index operation. This is why the `bar` function is more performant.

String and Byte Slices

Conversions between strings and byte slices

Strings could be viewed as element-immutable byte slices. With the guarantee that the content (the bytes stored in) of a string will never be modified, any function can safely use the content of the string without worrying about the content of the string being modified elsewhere.

To make the guarantee (the bytes stored in a string will never be modified), when converting a string to byte slice, the string and the result byte slice should not share byte elements. This means that the string content (the bytes) will be duplicated and stored into the result byte slice (one memory allocation is needed). Similarly, for the same reason, when converting a byte slice to string, the byte slice content (the byte elements of the the slice) will be duplicated and stored into the result string too (one memory allocation is needed).

In fact, under some situations, the duplications are not necessary. The current standard Go compiler makes several special optimizations to avoid duplications for some simple cases. Such optimizations will be listed below.

A string-to-byte-slice conversion following the `range` keyword doesn't allocate

If we use the official standard Go compiler to build the following code, the performance difference between the function `f` and the function `g` is neglectable. The reason is the `[]byte(s)` conversion used in the function `f` doesn't make allocations.

```
func f(s string) {
   for i, b := range []byte(s) {
      // do something with i and b
      _, _ = i, b
   }
}


func g(s string) {
   for i := 0; i < len(s); i++ {
      b := s[i]
      // do something with i and b
      _, _ = i, b
   }
}
```

A byte-slice-to-string conversion appearing as a comparison operand doesn't allocate

The following is the implementation of the `Equal` function in the `bytes` standard package. In the implementation, the two conversions both don't duplicate the byte elements of the corresponding arguments.

```
func Equal(a, b []byte) bool {
   return string(a) == string(b)
}
```

This optimization leads to the `verbose` function is more efficient than the `clean` function shown in the following code (as of the official standard Go compiler v1.19):

```
package main

import t "testing"

func verbose(x, y, z []byte){
   switch {
   case string(x) == string(y):
      // do something
   case string(x) == string(z):
      // do something
   }
}

func clean(x, y, z []byte){
   switch string(x) {
   case string(y):
      // do something
   case string(z):
      // do something
   }
}

func main() {
   x := []byte{1023: 'x'}
   y := []byte{1023: 'y'}
   z := []byte{1023: 'z'}
   stat := func(f func(x, y, z []byte)) int {
      allocs := t.AllocsPerRun(1, func() {
```

```
            f(x, y, z)
        })
        return int(allocs)
    }
    println(stat(verbose)) // 0
    println(stat(clean))   // 3
}
```

From the outputs, we could get that the the `verbose` function doesn't make allocations but the `simple` function makes three ones, which is just the reason why the former one is more performant.

The performance difference between the two functions might be removed since a future Go toolchain version.

We could also use the `bytes.Compare` function to compare two byte slices. The `bytes.Compare` function way is often more performant for the cases in which three-way comparisons (like the following code shows) are needed.

```
// Note, two branches are enough
// to form a three-way comparison.
func doSomething(x, y []byte) {
    switch bytes.Compare(x, y) {
    case -1:
        // ... do something 1
    case 1:
        // ... do something 2
    default:
        // ... do something 3
    }
}
```

Don't use the `bytes.Compare` function in simple (one-way) byte slice comparisons, as the following code shows. It is slower for such cases.

```
func doSomething(x, y []byte) {
   if bytes.Compare(x, y) == 0 {
      ... // do something
   }
}
```

A byte-slice-to-string conversion appearing as the index key of a map element retrieval expression doesn't allocate

Note: this optimization is not made for map element modification statements. This could be proved by the following example, in which the conversion `string(key)` in the `get` function doesn't allocate, but the two conversions in the other two functions do allocate.

```
package main

import t "testing"

var m = map[string]int{}
var key = []byte{'k', 'e', 'y'}
var n int

func get() {
   n = m[string(key)]
}

func inc() {
   m[string(key)]++
```

```
}

func set() {
   m[string(key)] = 123
}

func main() {
   stat := func(f func()) int {
      allocs := t.AllocsPerRun(1, f)
      return int(allocs)
   }
   println(stat(get)) // 0
   println(stat(set)) // 1
   println(stat(inc)) // 1
}
```

This optimization also works if the key presents as a struct or array composite literal form: `T1{... Tn{..., string(key), ...} ...}`, where `Tx` is either a struct type or an array type. For example, the conversion `string(key)` in the following code doesn't do duplications, too.

```
package main

import t "testing"

type T struct {
   a int
   b bool
   k [2]string
}

var m = map[T]int{}
```

```
var key = []byte{'k', 'e', 'y', 99: 'z'}

var n int
func get() {
   n = m[T{k: [2]string{1: string(key)}}]
}

func main() {
   print(int(t.AllocsPerRun(1, get))) // 0
}
```

This optimization leads to an interesting case. In the following code snippet, the function `modify1` makes one allocation but the function `modify2` makes none, so the function `modify2` is more performant than the function `modify1`. The reason could be easily found out from their respective equivalent forms. The `string(key)` used in the function `modify2` only appears in a map element retrieval expression, whereas the `string(key)` used in the function `modify1` should be thought as appearing in a map element modification statement.

```
package main

import t "testing"

var key = []byte{'k', 'e', 'y'}

var m1 = map[string]int{"key": 0}
func modify1() {
   m1[string(key)]++
   // (logically) equivalent to:
   // m1[string(key)] = m1[string(key)] + 1
```

```
}

var m2 = map[string]*int{"key": new(int)}
func modify2() {
   *m2[string(key)]++
   // equivalent to:
   // p := m2[string(key)]; *p = *p + 1
}

func main() {
   stat := func(f func()) int {
      allocs := t.AllocsPerRun(1, f)
      return int(allocs)
   }
   println(stat(modify1)) // 1
   println(stat(modify2)) // 0
}
```

So if the entries of a map are seldom deleted but the elements of the map are modified frequently, it is best to use a pointer type as the map element type.

A byte-slice-to-string conversion appearing as an operand in a string concatenation expression doesn't allocate if at least one of concatenated operands is a non-blank string constant

In the following example, the function `f` (the a bit verbose one) is much more efficient than the function `g` for most cases (as of the Go toolchain v1.19).

```
package main

import "testing"

var s = []byte{0: '$', 32: 'x'} // len(s) == 33

func f() string {
   return (" " + string(s) + string(s))[1:]
}

func g() string {
   return string(s) + string(s)
}

var x string
func stat(add func() string) int {
   c := func() {
      x = add()
   }
   allocs := testing.AllocsPerRun(1, c)
   return int(allocs)
}

func main() {
   println(stat(f)) // 1
   println(stat(g)) // 3
}
```

Please note that, currently (Go toolchain v1.19), this optimization is only useful for byte slices with lengths larger than 32. If we change the length of the string `s` to 32 (by declaring it with `var s = []byte{31: 'x'}`), then

the performance difference between the functions `f` and `g` will become neglectable. Please read the next section for the reason.

The a bit verbose way actually has a drawback: it wastes at least one byte more memory. If, at coding time, we know the byte value at a specified index of one operand, the this drawback could be avoided. For example, assume we know the first byte of the first operand is always `$`, then we could modify the a bit verbose way as the following code shows, to avoid wasting more memory.

```
func f() string {
   return "$" + string(s[1:]) + string(s)
}
```

Please note that, this optimization is some unintended. It might be not supported any more since a future Go toolchain version.

If the result of an operation is a stirng or byte slice, and the length of the result is larger than 32, then the byte elements of the result will be always allocated on heap

In fact, recall that the example in the last section uses a byte slice with 33 bytes, the reason is to avoid allocating the byte elements of the string concatenation operands on stack.

In the following program, the function `g` needs 3 heap allocations, but the function `f` needs none. The only differences between the two functions are the lengths of the involved byte slice and strings. The function `f` actually

makes 3 stack allocations, but the function `testing.AllocsPerRun` only counts heap allocations.

```
package main

import "testing"

var str = "1234567890abcdef" // len(str) == 16

func f() {
   x := str + str // does not escape
   y := []byte(x) // does not escape
   println(len(y), cap(y)) // 32 32
   z := string(y) // does not escape
   println(len(x), len(z)) // 32 32
}

func g() {
   x := str + str + "x" // does not escape
   y := []byte(x)       // does not escape
   println(len(y), cap(y)) // 33 48
   z := string(y)       // does not escape
   println(len(x), len(z)) // 33 33
}

func stat(fn func()) int {
   allocs := testing.AllocsPerRun(1, fn)
   return int(allocs)
}

func main() {
   println(stat(f)) // 0
```

```
   println(stat(g)) // 3
}
```

In the following benchmark code, the `concat_splited` way is more performant than the normal `concat` way, because the conversions in the former way don't make heap allocations.

```
package bytes

import "testing"

var s37 = []byte{36: 'x'} // len(s37) == 37
var str string

func Benchmark_concat(b *testing.B) {
   for i := 0; i < b.N; i++ {
      str = string(s37) + string(s37)
   }
}

func Benchmark_concat_splited(b *testing.B) {
   for i := 0; i < b.N; i++ {
      str = string(s37[:32]) +
         string(s37[32:]) +
         string(s37[:32]) +
         string(s37[32:])
   }
}
```

The benchmark results:

```
Benchmark_concat-4          420.6 ns/op  176 B/op  3 allocs/op
Benchmark_concat_splited-4  359.7 ns/op   80 B/op  1 allocs/op
```

Efficient ways to concatenate strings

There are two principal ways to effectively concatenate strings in Go: use the `+` operator and use `strings.Builder`. Each way has its own advantages and drawbacks.

The `+` operator is mainly used to effectively concatenate multiple strings in one statement. If it is impossible to use the `+` operator to concatenate some strings in one statement, then it is more efficient to use `strings.Builder` to concatenate them.

The byte elements of the result string of using `strings.Builder` will be always allocated on heap. On the other hand, by using the `+` operator, the byte elements of the result string will be allocated on stack if the length of the result string is small (not larger than 32) and the result string doesn't escape to heap.

The `strings.Builder` way calls the `make` function to allocate enough bytes for concatenations and the `make` call zeros those bytes, which is unnecessarily.

For the above two reasons, in theory, the `+` operator way should be always more performant than the `strings.Builder` way. However, for a runtime implementation imperfection (in Go 1.15/1.16/1.17), sometimes the `+` operator way might be a little slower than the `strings.Builder` way. This imperfection has been fixed since Go 1.18.

Let's compare the efficiencies of the two ways.

```
package bytes

import "testing"
import "strings"

const M, N, K = 12, 16, 32

var s1 = strings.Repeat("a", M)
var s2 = strings.Repeat("a", N)
var s3 = strings.Repeat("a", K)
var r1, r2 string

func init() {
   println("======", M, N, K)
}

//go:noinline
func Concat_WithPlus(a, b, c string) string {
   return a + b + c
}

//go:noinline
func Concat_WithBuilder(ss ...string) string {
   var b strings.Builder
   var n = 0
   for _, s := range ss {
      n += len(s)
   }
   b.Grow(n)
   for _, s := range ss {
      b.WriteString(s)
   }
   return b.String()
```

```
}

func Benchmark_Concat_WithPlus(b *testing.B) {
   for i := 0; i < b.N; i++ {
      r1 = Concat_WithPlus(s1, s2, s3)
      r2 = Concat_WithPlus(s3, s3, s3)
   }
}

func Benchmark_Concat_WithBuilder(b *testing.B) {
   for i := 0; i < b.N; i++ {
      r1 = Concat_WithBuilder(s1, s2, s3)
      r2 = Concat_WithBuilder(s3, s3, s3)
   }
}
```

The benchmark results:

```
====== 12 16 32
Benchmark_Concat_WithPlus-4     192.2 ns/op
Benchmark_Concat_WithBuilder-4  196.8 ns/op
```

So if it is possible to do the whole concatenation in one statement, generally, we should use the `+` operator to concatenate strings for its simplicity and great performance.

The `strings.Builder` way might waste some memory sometimes

The `strings.Builder` way is great to build a string from unknown number (at coding time) of strings and byte slices. When using this way to build a

string, if it is possible, we should determine the length of the final built string in advance and call the `Grow` method of a `strings.Builder` value to grow the underlying byte slice to that length in one step.

However, for some cases, it might be impossible or inconvenient to calculate the length of the final built string in advance. In such cases, the underlying byte slice used by a `strings.Builder` value will grow when needed. It might grow to a capacity which is larger than the length of the final built string. As the final built string will reuse the memory block of the underlying byte slice, there will be some memory wasted in the end.

Use byte slice to concatenate strings

There is actually a third way to concatenate strings: build a byte slice, then copy the bytes from the concatenated strings to the byte slice, finally convert the byte slice to the result string.

The byte slice way is almost the same as the implementation of the `strings.Builder` way, except that the final conversion (from byte slice to string) needs a memory allocation, whereas the `strings.Builder` way uses unsafe mechanism to avoid the allocation. So in theory, the byte slice way should be alwys slower than the the `strings.Builder` way. However, if the byte slice used in the byte slice way is created with a constant capacity, then its elements might be allocated on stack, which is an advantage over the `strings.Builder` way. The advantage might make the byte slice way faster than the above introduced two ways for some use cases.

In the following benchmark code, if the length of the package-level string `s` is not larger than 16, then the `Concat_WithBytes` way is more performant than the `Concat_WithPlus` way; otherwise, the `Concat_WithBytes` way is slower.

```
package bytes

import "testing"

var s = "1234567890abcdef" // len(s) == 16
var r string

//go:noinline
func Concat_WithPlus(a, b, c, d string) string {
   return a + b + c + d
}

//go:noinline
func Concat_WithBytes(ss ...string) string {
   var n = 0
   for _, s := range ss {
      n += len(s)
   }
   var bs []byte
   if n > 64 {
      bs = make([]byte, 0, n) // escapes to heap
   } else {
      bs = make([]byte, 0, 64) // does not escape
   }
   for _, s := range ss {
      bs = append(bs, s...)
   }
```

```
    return string(bs)
}

func Benchmark_Concat_WithPlus(b *testing.B) {
    for i := 0; i < b.N; i++ {
        r = Concat_WithPlus(s, s, s, s)
    }
}

func Benchmark_Concat_WithBytes(b *testing.B) {
    for i := 0; i < b.N; i++ {
        r = Concat_WithBytes(s, s, s, s)
    }
}
```

The benchmark results (when the length of `s` is 16):

```
Benchmark_Concat_WithPlus-4   235.1 ns/op
Benchmark_Concat_WithBytes-4  208.3 ns/op
```

The benchmark results (when the length of `s` is 17):

```
Benchmark_Concat_WithPlus-4   236.3 ns/op
Benchmark_Concat_WithBytes-4  357.5 ns/op
```

We could also modify the byte elements of the slice as needed before converting the slice to the final string. This is another benefit comparing to the other two ways.

Merge a string and a byte slice into a new byte slice

Sometimes, we need to merge a string (`str`) a byte slice (`bs`) into a new byte slice. There are two ways to achieve this goal.

Way 1 (the one-line way):

```
    var newByteSlice = append([]byte(str), bs...)
```

Way 2 (the verbose way):

```
    var newByteSlice = make([]byte, len(str) + len(bs))
  copy(newByteSlice, str)
  copy(newByteSlice[len(str):], bs)
```

Generlaly, if the length of the string is much larger than the byte slice, then the verbose way is more performant. On the contrary, if the length of the byte slice is much larger than the string, then the one-line way is more performant.

Sadly, currently (Go toolchain v1.19), there is not an extreme performant way to do the merge.

- In the one-line way, the conversion `[]byte(str)` will duplicate the underlying of the string, which is unnecessarily.
- In the verbose way, the elements within `newByteSlice[len(str):]` will be unnecessarily zeroed in the `make` call.

The `strings.Compare` function is not very performant now

Currently (Go v1.19), the implementation of the `strings.Compare` function is deliberately implemented with a not extreme high performant. In other words, it is not recommended to be used in practice now. Instead, it is recommended by Go core developers to use comparison operators to compare strings. In my honest opinion, it is an infeasible idea for all scenarios. After all, the two ways both have their respective best use scenarios.

The following code shows the current implementation of the `strings.Compare` function. The implementation is inefficient for the cases that the two argument strings have the same length and share a not-short common prefix. For such cases, two raw comparisons are performed.

```
package strings

func Compare(a, b string) int {
   if a == b {
      return 0
   }
   if a < b {
      return -1
   }
   return +1
}
```

As a contrast, the implementation of the `bytes.Compare` function is highly optimized by calling an internal function `bytealg.Compare`.

```
package bytes
```

```
func Compare(a, b []byte) int {
   return bytealg.Compare(a, b)
}
```

Hope future Go versions could optimize this function.

The `strings.Compare` function is expected to be used in three-way string comparisons, as the following code shows:

```
func foo(x, y string) {
   switch strings.Compare(x, y) {
   case -1:
      // do something 1
   case 0:
      // do something 2
   case 1:
      // do something 3
   }
}
```

As the `strings.Compare` function is not very efficient now, some people would use comparison operators to do the job instead. But please note that, if the lengths of the comparison operands are often not equal, then we should not handle the "equal" cases in the `default` branch. For example, in the following code, the function `f3` is often less performant than the other two functions. The reason is the comparison `x == y` is much faster than `x < y` and `x > y` if the lengths of the comparison operands are not equal.

```
func f1(x, y string) {
   switch {
   case x == y: // ... handle 1
```

```
    case x < y:  // ... handle 2
    default:     // ... handle 3
    }
}

func f2(x, y string) {
    switch {
    case x < y:  // ... handle 2
    case x == y: // ... handle 1
    default:     // ... handle 3
    }
}

func f3(x, y string) {
    switch {
    case x < y: // ... handle 2
    case x > y: // ... handle 3
    default:    // ... handle 1
    }
}
```

Avoid allocations if possible

String and byte slice operations often involve memory allocations. Many allocated memory blocks in such operations are short-lived. To make program run efficiently, we should try to allocate less short-lived memory blocks.

In this section, some examples will be provided to show how to save allocations under some situations.

In the following example, the function `f` is more efficient than the function `g`. The reason is the former makes one allocation, whereas the latter makes two.

```
func f(a, b, c string) {
   abc := a + b + c
   ab := abc[:len(abc)-len(c)]
   ...
}

func g(a, b, c string) {
   ab := a + b
   abc := ab + c
   ...
}
```

In the following code, using array values as the map keys is more performant than using strings as the map keys, because the former way doesn't allocate in building entry keys.

```
package bytes

import "testing"

var ma = make(map[[2]string]struct{})
var ms = make(map[string]struct{})

var keyparts = []string {
   "docs", "aaa",
   "pictures", "bbb",
   "downloads", "ccc",
}
```

```
func fa(a, b string) {
   ma[[2]string{a, b}] = struct{}{}
}

func fs(a, b string) {
   ms[a + "/" + b] = struct{}{}
}

func Benchmark_array_key(b1 *testing.B) {
   for i := 0; i < b1.N; i++ {
      for i := 0; i < len(keyparts); i+= 2 {
         fa(keyparts[i], keyparts[i+1])
      }
      for key := range ma {delete(ma, key)}
   }
}

func Benchmark_string_key(b1 *testing.B) {
   for i := 0; i < b1.N; i++ {
      for i := 0; i < len(keyparts); i+= 2 {
         fs(keyparts[i], keyparts[i+1])
      }
      for key := range ms {delete(ms, key)}
   }
}
```

The benchmark result:

```
Benchmark_array_key-4   147.0 ns/op   0 B/op  0 allocs/op
Benchmark_string_key-4  507.9 ns/op  40 B/op  3 allocs/op
```

We could also use struct values as the map keys, which should be as performant as using array keys.

The third example, which shows the performance difference between two ways of string comparisons by ignoring cases.

```
package bytes

import "testing"
import "strings"

var ss = []string {
   "AbcDefghijklmnOpQrStUvwxYz1234567890",
   "abcDefghijklmnopQRSTuvwXYZ1234567890",
   "aBcDefgHIjklMNOPQRSTuvwxyz1234567890",
}

func Benchmark_EqualFold(b1 *testing.B) {
   for i := 0; i < b1.N; i++ {
      for _, a := range ss {
         for _, b := range ss {
            r := strings.EqualFold(a, b)
            if !r {panic("not equal!")}
         }
      }
   }
}

func Benchmark_CompareToLower(b1 *testing.B) {
   for i := 0; i < b1.N; i++ {
      for _, a := range ss {
         for _, b := range ss {
```

```
            r := strings.ToLower(a) ==
               strings.ToLower(b)
            if !r {panic("not equal!")}
         }
      }
   }
}
```

From the following benchmark results, we know the `EqualFold` way is much more performant the `ToLower` comparison way, because the former way doesn't allocate.

```
Benchmark_EqualFold-4        1271 ns/op    0 B/op   0 allocs/op
Benchmark_CompareToLower-4   7157 ns/op  864 B/op  18 allocs/op
```

The `io.Writer` type in the standard library only has one method: `Write([]byte) (int, error)`. When a string needs to be written, it must be converted to a byte slice before being passed to the `Write` method. This is quite inefficient. In the following code, a `BytesWriter` type is implemented to support writing strings without converting the strings to byte slices.

```
package bytes

import "testing"
import "io"

type BytesWriter struct {
   io.Writer
   buf []byte
}
```

```
func NewBytesWriter(w io.Writer, bufLen int) *BytesWriter {
   return &BytesWriter {
      Writer: w,
      buf: make([]byte, bufLen),
   }
}

func (sw *BytesWriter) WriteString(s string) (int, error) {
   var sum int
   var err error
   for len(s) > 0 {
      n := copy(sw.buf, s)
      n, err = sw.Write(sw.buf[:n])
      sum += n
      s = s[n:]
      if err != nil || n == 0 {
         break
      }
   }
   if err == nil && len(s) > 0 {
      err = io.ErrShortWrite
   }
   return sum, err
}

type DummyWriter struct{}

func (dw DummyWriter) Write(bs []byte) (int, error) {
   return len(bs), nil
}

var s = string(make([]byte, 500))
```

```
var w io.Writer = DummyWriter{}
var bytesw = NewBytesWriter(DummyWriter{}, 512)

func Benchmark_BytesWriter(b *testing.B) {
   for i := 0; i < b.N; i++ {
      bytesw.WriteString(s)
   }
}

func Benchmark_GeneralWriter(b *testing.B) {
   for i := 0; i < b.N; i++ {
      w.Write([]byte(s))
   }
}
```

The benchmark results:

```
Benchmark_BytesWriter-4     21.03 ns/op   0 B/op  0 allocs/op
Benchmark_GeneralWriter-4  390.3 ns/op  512 B/op  1 allocs/op
```

From the benchmark results, we could find that the `BytesWriter` way is much more performant than the general `io.Writer` way, because the former way doesn't allocate (except the single buffer allocation).

Please note, there is a type in the standard package, `bufio.Writer`, which acts like the `BytesWriter` type. Generally, we should use that type instead.

BCE (Bound Check Eliminate)

Go is a memory safe language. In array/slice/string 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 checking.

Bounds checks make our code run safely, on the other hand, they also make our code run a little slower. This is a trade-off a safe language must made.

Since Go toolchain v1.7, the standard Go compiler has started to support BCE (bounds check elimination). BCE can avoid some unnecessary bounds checks, so that the standard Go compiler could generate more efficient programs.

The following will list some examples to show in which cases BCE works and in which cases BCE doesn't work.

We could use the `-d=ssa/check_bce` compiler option to show which code lines need bound checks.

Example 1

A simple example:

```
// example1.go
package main
```

```
func f1a(s []struct{}, index int) {
   _ = s[index] // line 5: Found IsInBounds
   _ = s[index]
   _ = s[index:]
   _ = s[:index+1]
}

func f1b(s []byte, index int) {
   s[index-1] = 'a' // line 12: Found IsInBounds
   _ = s[:index]
}

func f1c(a [5]int) {
   _ = a[0]
   _ = a[4]
}

func f1d(s []int) {
   if len(s) > 2 {
       _, _, _ = s[0], s[1], s[2]
   }
}

func f1e(s []byte) {
   if len(s) >= 2 {
      r := len(s) % 2
      _ = s[r]
   }
}

func main() {}
```

Let's run it with the `-d=ssa/check_bce` compiler option:

```
$ go run -gcflags="-d=ssa/check_bce" example1.go
./example1.go:5:7: Found IsInBounds
./example1.go:12:3: Found IsInBounds
```

The outputs show that only two code lines needs bound checks in the above example code.

Note that, Go toolchains with version smaller than 1.19 failt to remove the unnecessary bound check in the `f1e` function.

Example 2

All the bound checks in the slice element indexing and subslice operations shown in the following example are eliminated.

```
// example2.go
package main

func f2a(s []int) {
   for i := range s {
      _ = s[i]
      _ = s[i:len(s)]
      _ = s[:i+1]
   }
}

func f2b(s []int) {
   for i := 0; i < len(s); i++ {
      _ = s[i]
      _ = s[i:len(s)]
```

```
        _ = s[:i+1]
    }
}

func f2c(s []int) {
    for i := len(s) - 1; i >= 0; i-- {
        _ = s[i]
        _ = s[i:len(s)]
        _ = s[:i+1]
    }
}

func f2d(s []int) {
    for i := len(s); i > 0; {
        i--
        _ = s[i]
        _ = s[i:len(s)]
        _ = s[:i+1]
    }
}

func f2e(s []int) {
    for i := 0; i < len(s) - 1; i += 2 {
        _ = s[i]
        _ = s[i:len(s)]
        _ = s[:i+1]
    }
}

func main() {}
```

Run it, we will find that nothing is outputted. Yes, the official standard Go compiler is so clever that it finds all bound checks may be removed in the

above example code.

```
$ go run -gcflags="-d=ssa/check_bce" example2.go
```

There are still some small imperfections. If we modify the `f2c` and `f2e` functions as that shown in the following code, then the compiler (v1.19) fails to remove the bound checks for the two subslice operations.

```
// example2b.go
package main

func f2c(s []int) {
   for i := len(s) - 1; i >= 0; i-- {
      //_ = s[i]
      _ = s[:i+1] // line 7: Found IsSliceInBounds
   }
}

func f2e(s []int) {
   for i := 0; i < len(s) - 1; i += 2 {
      //_ = s[i]
      _ = s[:i+1] // line 14: Found IsSliceInBounds
   }
}

func main() {}
```

Run it, we will get the following output:

```
$ go run -gcflags="-d=ssa/check_bce" example2b.go
./example2b.go:7:8: Found IsSliceInBounds
./example2b.go:14:8: Found IsSliceInBounds
```

We may give the compiler some hints by turning on the comment lines to remove these bound checks.

Example 3

We should try to evaluate the element indexing or subslice operation with the largest index as earlier as possible to reduce the number of bound checks.

In the following example, if the expression `s[3]` is evaluated without panicking, then the bound checks for `s[0]`, `s[1]` and `s[2]` could be eliminated.

```
// example3.go
package main

func f3a(s []int32) int32 {
   return s[0] | // Found IsInBounds (line 5)
      s[1] | // Found IsInBounds
      s[2] | // Found IsInBounds
      s[3]   // Found IsInBounds
}

func f3b(s []int32) int32 {
   return s[3] | // Found IsInBounds (line 12)
      s[0] |
      s[1] |
      s[2]
}
```

```
func main() {
}
```

Run it, we get:

```
./example3.go:5:10: Found IsInBounds
./example3.go:6:4: Found IsInBounds
./example3.go:7:4: Found IsInBounds
./example3.go:8:4: Found IsInBounds
./example3.go:12:10: Found IsInBounds
```

From the output, we could learn that there are 4 bound checks in the `f3a` function, but only one in the `f3b` function.

Example 4

Since Go toolchain v1.19, the bould check in the `f5a` function is successfully removed,

```
func f5a(isa []int, isb []int) {
   if len(isa) > 0xFFF {
      for _, n := range isb {
         _ = isa[n & 0xFFF]
      }
   }
}
```

However, before Go toolchain v1.19, the check is not removed. The compiler (v1.18-) needs a hint to be removed, as shown in the `f5b` function:

```
func f5b(isa []int, isb []int) {
   if len(isa) > 0xFFF {
```

```
      // A successful hint (for v1.18- compilers)
      isa = isa[:0xFFF+1]
      for _, n := range isb {
         _ = isa[n & 0xFFF] // BCEed!
      }
   }
}

func f5c(isa []int, isb []int) {
   if len(isa) > 0xFFF {
      // A not-workable hint (for v1.18- compilers)
      _ = isa[:0xFFF+1]
      for _, n := range isb {
         _ = isa[n & 0xFFF] // Found IsInBounds
      }
   }
}

func f5d(isa []int, isb []int) {
   if len(isa) > 0xFFF {
      // A not-workable hint (for v1.18- compilers)
      _ = isa[0xFFF]
      for _, n := range isb {
         _ = isa[n & 0xFFF] // Found IsInBounds
      }
   }
}
```

The next section shows more cases which need compiler hints to avoid some unnecessary bound checks.

Sometimes, the compiler needs some hints to remove some bound checks

The official standard Go compiler is still not smart enough to remove all unnecessary bound checks. Sometimes, the compiler needs to be given some hints to remove some bound checks.

In the following example, by adding one hint line in the function `f4b`, all bound checks in the loop are eliminated.

```
func f4a(is []int, bs []byte) {
   if len(is) >= 256 {
      for _, n := range bs {
         _ = is[n] // Found IsInBounds
      }
   }
}

func f4b(is []int, bs []byte) {
   if len(is) >= 256 {
      is = is[:256] // a successful hint
      for _, n := range bs {
         _ = is[n] // BCEed!
      }
   }
}

func f4c(is []int, bs []byte) {
   if len(is) >= 256 {
      _ = is[:256] // a non-workable hint
      for _, n := range bs {
```

```
            _ = is[n] // Found IsInBounds
        }
    }
}

func f4d(is []int, bs []byte) {
    if len(is) >= 256 {
        _ = is[255] // a non-workable hint
        for _, n := range bs {
            _ = is[n] // Found IsInBounds
        }
    }
}
```

Please note that, as of Go toolchain v1.19, the two hints used in the `f4c` and `f4d` functions are not workable (but they should).

In the following example, by adding a redundant `if` code block in the function `NumSameBytes_2`, all bound checks in the loop are eliminated.

```
type T = string

func NumSameBytes_1(x, y T) int {
    if len(x) > len(y) {
        x, y = y, x
    }
    for i := 0; i < len(x); i++ {
        if x[i] !=
            y[i] { // Found IsInBounds
            return i
        }
    }
```

```
    return len(x)
}

func NumSameBytes_2(x, y T) int {
    if len(x) > len(y) {
        x, y = y, x
    }

    // a successful hint
    if len(x) > len(y) {
        panic("unreachable")
    }

    for i := 0; i < len(x); i++ {
        if x[i] != y[i] { // BCEed!
            return i
        }
    }
    return len(x)
}
```

The above hint works for `T` is either a string type or a slice type. But each of the following two hints only works for one case (as of Go toolchain v1.19).

```
func NumSameBytes_3(x, y T) int {
    if len(x) > len(y) {
        x, y = y, x
    }

    y = y[:len(x)] // a hint, only works if T is slice
    for i := 0; i < len(x); i++ {
```

```
      if x[i] != y[i] {
         return i
      }
   }
   return len(x)
}

func NumSameBytes_4(x, y T) int {
   if len(x) > len(y) {
      x, y = y, x
   }

   _ = y[:len(x)] // a hint, only works if T is string
   for i := 0; i < len(x); i++ {
      if x[i] != y[i] {
         return i
      }
   }
   return len(x)
}
```

Please note that, the future versions of the standard official Go compiler will become smarter so that the above hints will become unnecessary later.

Write code in BCE-friendly ways

In the following example, the `f7b` and `f7c` functions makes 3 less bound checks than `f7a`.

```
func f7a(s []byte, i int) {
   _ = s[i+3] // Found IsInBounds
   _ = s[i+2] // Found IsInBounds
```

```
    _ = s[i+1] // Found IsInBounds
    _ = s[i]   // Found IsInBounds
}

func f7b(s []byte, i int) {
    s = s[i:i+4] // Found IsSliceInBounds
    _ = s[3]
    _ = s[2]
    _ = s[1]
    _ = s[0]
}

func f7c(s []byte, i int) {
    s = s[i:i+4:i+4] // Found IsSliceInBounds
    _ = s[3]
    _ = s[2]
    _ = s[1]
    _ = s[0]
}
```

However, please note that, there might be some other factors which will affect program performances. On my machine (Intel i5-4210U CPU @ 1.70GHz, Linux/amd64), among the above 3 functions, the function `f7b` is actually the least performant one.

```
Benchmark_f7a-4  3861 ns/op
Benchmark_f7b-4  4223 ns/op
Benchmark_f7c-4  3477 ns/op
```

In practice, it is encouraged to use the three-index subslice form (`f7c`).

In the following example, benchmark results show that

- the `f8z` function is the most performant one (in line with expectation)
- but the `f8y` function is as performant as the `f8x` function (unexpected).

```
func f8x(s []byte) {
   for i := 0; i < len(s) - 3; i += 4 {
      _ = s[i+3] // Found IsInBounds
      _ = s[i+2] // Found IsInBounds
      _ = s[i+1] // Found IsInBounds
      _ = s[i]
   }
}

func f8y(s []byte) {
   for i := 0; i < len(s) - 3; i += 4 {
      s2 := s[i:]
      _ = s2[3] // Found IsInBounds
      _ = s2[2]
      _ = s2[1]
      _ = s2[0]
   }
}

func f8z(s []byte) {
   for i := 0; len(s) >= 4; i += 4 {
      _ = s[3]
      _ = s[2]
      _ = s[1]
      _ = s[0]
      s = s[4:]
   }
}
```

In fact, benchmark results also show the following `f8y3` function is as performant as the `f8z` function and the performance of the `f8y2` function is on par with the `f8y` function. So it is encouraged to use three-index subslice forms for such situations in practice.

```
func f8y2(s []byte) {
   for i := 0; i < len(s) - 3; i += 4 {
      s2 := s[i:i+4] // Found IsInBounds
      _ = s2[3]
      _ = s2[2]
      _ = s2[1]
      _ = s2[0]
   }
}

func f8y3(s []byte) {
   for i := 0; i < len(s) - 3; i += 4 {
      s2 := s[i:i+4:i+4] // Found IsInBounds
      _ = s2[3]
      _ = s2[2]
      _ = s2[1]
      _ = s2[0]
   }
}
```

In the following example, there are no bound checks in the `f9b` function, but there is one in the `f9a` function.

```
func f9a(n int) []int {
   buf := make([]int, n+1)
   k := 0
   for i := 0; i <= n; i++ {
```

```
      buf[i] = k // Found IsInBounds
      k++
   }
   return buf
}


func f9b(n int) []int {
   buf := make([]int, n+1)
   k := 0
   for i := 0; i < len(buf); i++ {
      buf[i] = k
      k++
   }
   return buf
}
```

In the following code, the function `f0b` is much more performant than `f0a`.

```
func f0a(x [16]byte) (r [4]byte){
   for i := 0; i < 4; i++ {
      r[i] =
         x[i*4+3] ^ // Found IsInBounds
         x[i*4+2] ^ // Found IsInBounds
         x[i*4+1] ^ // Found IsInBounds
         x[i*4]     // Found IsInBounds
   }
   return
}

func f0b(x [16]byte) (r [4]byte){
   r[0] = x[3] ^ x[2] ^ x[1] ^ x[0]
   r[1] = x[7] ^ x[6] ^ x[5] ^ x[4]
```

```
    r[2] = x[11] ^ x[10] ^ x[9] ^ x[8]
    r[3] = x[15] ^ x[14] ^ x[13] ^ x[12]
    return
}
```

Global (package-level) slices are often unfriendly to BCE, so we should try to assign them to local ones to eliminate some unnecessary bound checks. For example, in the following code, the `fa0` function does one more bound check than the `fa1` and `fa2` functions, so the function calls `fa1()` and `fa2(s)` are both more performant than `fa0()`.

```
var s = make([]int, 5)

func fa0() {
    for i := range s {
        s[i] = i // Found IsInBounds
    }
}

func fa1() {
    s := s
    for i := range s {
        s[i] = i
    }
}

func fa2(x []int) {
    for i := range x {
        x[i] = i
    }
}
```

Arrays are often more BCE-friendly than slices. In the following code, the array version functions (`fb2` and `fc2`) don't need bound checks.

```
var s = make([]int, 256)
var a = [256]int{}

func fb1() int {
    return s[100] // Found IsInBounds
}

func fb2() int {
    return a[100]
}

func fc1(n byte) int {
    return s[n] // Found IsInBounds
}

func fc2(n byte) int {
    return a[n]
}
```

Please note that, the future versions of the standard official Go compiler will become smarter so that some BCE-unfriendly code might become BCE-friendly later.

The current official standard Go compiler fails to eliminate some unnecessary bound checks

As of Go toolchain v1.19, the official standard Go compiler doesn't eliminate the following unnecessary bound checks.

```
func fb(s, x, y []byte) {
   n := copy(s, x)
   copy(s[n:], y) // Found IsSliceInBounds
   _ = x[n:]      // Found IsSliceInBounds
}

func fc(s []byte) {
   const N = 6
   for i := 0; i < len(s)-(N-1); i += N {
      _ = s[i+N-1] // Found IsInBounds
   }
}

func fd(data []int, check func(int) bool) []int {
   var k = 0
   for _, v := range data {
      if check(v) {
         data[k] = v // Found IsInBounds
         k++
      }
   }
   return data[:k] // Found IsSliceInBounds
}
```

The future versions of the standard official Go compiler will become smarter so that the above unnecessary bound checks will be eliminated later.

Maps

In Go, the capacity of a map is unlimited in theory, it is only limited by available memory. That is why the builtin `cap` function doesn't apply to maps.

In the official standard Go runtime implementation, maps are implemented as hashtables internally. Each map/hashtable maintains a backing array to store map entries (key-value pairs). Along with more and more entries are put into a map, the size of the backing array might be thought as too small to store more entries, thus a new larger backing array will be allocated and the current entries (in the old backing array) will be moved to it, then the old backing array will be discarded.

In the official standard Go runtime implementation, the backing array of a map will never shrink, even if all entries are deleted from the map. This is a form of memory wasting.

Clear map entries

We could use the following loop to clear all entries in a map:

```
    for key := range aMap {
     delete(aMap, key)
  }
```

The loop is specially optimized so that its execution is very fast. However, please note that, as mentioned above, the backing array of the cleared map

doesn't shrink after the loop. Then how to release the backing array of the map? There are two ways:

```
    aMap = nil
  // or
  aMap = map(map[K]V)
```

If the backing array of the map is not referenced elsewhere, then the backing array will be collected eventually after being released.

If there will be many new entries to be put in the map after it is cleared, then the former way is preferred; otherwise, the latter (release) ways are preferred.

`aMap[key]++` is more efficient than `aMap[key] = aMap[key] + 1`

In the statement `aMap[key] = aMap[key] + 1`, the `key` are hashed twice, but in the statement `aMap[key]++`, it is only hashed once.

Similarly, `aMap[key] += value` is more efficient than `aMap[key] = aMap[key] + value`.

These could be proved by the following benchmark code:

```
package maps

import "testing"

var m = map[int]int{}
```

```
func Benchmark_increment(b *testing.B) {
   for i := 0; i < b.N; i++ {
      m[99]++
   }
}

func Benchmark_plusone(b *testing.B) {
   for i := 0; i < b.N; i++ {
      m[99] += 1
   }
}

func Benchmark_addition(b *testing.B) {
   for i := 0; i < b.N; i++ {
      m[99] = m[99] + 1
   }
}
```

The benchmark results:

```
Benchmark_increment-4  11.31 ns/op
Benchmark_plusone-4    11.21 ns/op
Benchmark_addition-4   16.10 ns/op
```

Pointers in maps

If the key type and element type of a map both don't contain pointers, then in the scan phase of a GC cycle, the garbage collector will not scan the entries of the map. This could save much time.

This tip is also valid for other kinds of container in Go, such as slices, arrays and channels.

Using byte arrays instead of short strings as keys

Internally, each string contains a pointer, which points to the underlying bytes of that string. So if the key or element type of a map is a string type, then all the entries of the map needs to be scanned in GC cycles.

If we can make sure that the string values used in the entries of a map have a max length and the max length is small, then we could use the array type `[N]byte` to replace the string types (where `N` is the max string length). Doing this will save much garbage collection scanning time if the number of the entries in the map is very large.

For example, in the following code, the entries of `mapB` contain no pointers, but the (string) keys of `mapA` contain pointers. So garbage collector will skip `mapB` during the scan phase of a GC cycle.

```
    var mapA = make(map[string]int, 1 << 16)
  var mapB = make(map[[32]byte]int, 1 << 16)
```

Lower map element modification frequency

In the previous "strings and byte slices" chapter, it has been mentioned that a byte-slice-to-string conversion appearing as the index key in a map element retrieval expression doesn't allocate, but such conversions in L-value map element index expressions will allocate.

So sometimes, we could lower the frequency of using such conversions in L-value map element index expressions to improve program performance.

In the following example, the B way (pointer element way) is more performant than the A way. The reason is the B way modifies element values rarely. The elements in the B way are pointers, once they are created, they are never changed.

```
package maps

import "testing"

var wordCounterA = make(map[string]int)
var wordCounterB = make(map[string]*int)
var key = make([]byte, 64)

func IncA(w []byte) {
   wordCounterA[string(w)]++
}

func IncB(w []byte) {
   p := wordCounterB[string(w)]
   if p == nil {
      p = new(int)
      wordCounterB[string(w)] = p
   }
   *p++
}

func Benchmark_A(b *testing.B) {
   for i := 0; i < b.N; i++ {
      for i := range key {
         IncA(key[:i])
      }
   }
```

```
}

func Benchmark_B(b *testing.B) {
   for i := 0; i < b.N; i++ {
      for i := range key {
         IncB(key[:i])
      }
   }
}
```

The benchmark results:

```
Benchmark_A-4  11600 ns/op  2336 B/op  62 allocs/op
Benchmark_B-4   1543 ns/op     0 B/op   0 allocs/op
```

Although the B way (pointer element way) is less CPU consuming, it creates many pointers, which increases the burden of pointer scanning in a GC cycle. But generally, the B way is more efficient.

We could use an extra counter table (a slice) and let the map record indexes to the table, to avoid making many allocations and creating many pointers, as the following code shows:

```
var wordIndexes = make(map[string]int)
var wordCounters []int

func IncC(w []byte) {
   if i, ok := wordIndexes[string(w)]; ok {
      wordCounters[i]++
   } else {
      wordIndexes[string(w)] = len(wordCounters)
      wordCounters = append(wordCounters, 1)
```

```
    }
}

func Benchmark_C(b *testing.B) {
    for i := 0; i < b.N; i++ {
        for i := range key {
            IncC(key[:i])
        }
    }
}
```

The benchmark results:

```
Benchmark_A-4  11600 ns/op  2336 B/op  62 allocs/op
Benchmark_B-4   1543 ns/op     0 B/op   0 allocs/op
Benchmark_C-4   1609 ns/op     0 B/op   0 allocs/op
```

From a short-period view, the C way is as almost performant as the B way, But as it uses much less pointers, it is actually more efficient than the B way in a long-period view.

Please note that the above benchmark results show the latter two ways both make zero allocations. This is actually not true. It is just that each of latter two benchmark runs makes less than one allocation averagely, which is truncated to zero. This is a deliberate design of the benchmark reports in the standard packages.

Try to grow a map in one step

If we could predict the max number of entries will be put into a map at coding time, we should create the map with the `make` function and pass the

max number as the `size` argument of the `make` call, to avoid growing the map in multiple steps later.

Use index tables instead of maps which key types have only a small set of possible values

Some programmers like to use a map with bool key to reduce verbose `if-else` code block uses. For example, the following code

```
    // Within a function ...
  var condition bool
  condition = evaluateCondition()
  ...
  if condition {
     counter++
  } else {
     counter--
  }
  ...
  if condition {
     f()
  } else {
     g()
  }
  ...
```

could be replaced with

```
// Package-level maps.
var boolToInt = map[bool]int{true: 1, false: 0}
var boolToFunc = map[bool]func(){true: f, false: g}
```

```
   // Within a function ...
   var condition bool
   condition = evaluateCondition()
   ...
   counter += boolToInt[condition]
   ...
   boolToFunc[condition]()
   ...
```

If there are many such identical `if-else` blocks used in code, using maps with bool keys will reduce many boilerplates and make code look much cleaner. For most use cases, this is generally good. However, as of Go toolchain v1.19, the map way is not very efficient from the code execution performance view. The following benchmarks show the performance differences.

```
package maps

import "testing"

//go:noiline
func f() {}

//go:noiline
func g() {}

func IfElse(x bool) func() {
   if x {
      return f
   } else {
      return g
   }
```

```
}

var m = map[bool]func() {true: f, false: g}
func MapSwitch(x bool) func() {
   return m[x]
}

func Benchmark_IfElse(b *testing.B) {
   for i := 0; i < b.N; i++ {
      IfElse(true)()
      IfElse(false)()
   }
}

func Benchmark_MapSwitch(b *testing.B) {
   for i := 0; i < b.N; i++ {
      MapSwitch(true)()
      MapSwitch(false)()
   }
}
```

The benchmark results:

```
Benchmark_IfElse-4      4.155 ns/op
Benchmark_MapSwitch-4  47.46 ns/op
```

From the benchmark results, we could get that the `if-else` block way is much more performant than the map-switch way.

For the use cases which require high code performance, we can use index tables to reduce `if-else` boilerplates but still keep the simplicity of the

map switch way, with the help of a bool-to-int function. The following benchmarks show how to use the index table way.

```
func b2i(b bool) (r int) {
   if b {r = 1}
   return
}
var a = [2]func() {g, f}
func IndexTable(x bool) func() {
   return a[b2i(x)]
}

func Benchmark_IndexTable(b *testing.B) {
   for i := 0; i < b.N; i++ {
      IndexTable(true)()
      IndexTable(false)()
   }
}
```

From the above code, we could find that the uses of the index table way is as clean as the map-switch way, though it needs to define an extra tiny `b2i` function. And from the following benchmark results, we know that the index table way is as performant as the `if-else` block way.

```
Benchmark_IfElse-4      4.155 ns/op
Benchmark_MapSwitch-4  47.46 ns/op
Benchmark_IndexTable-4  4.135 ns/op
```

Channels

Programming with channels is fun but channels are not the most performant way for some use cases

The channel way might be fun to use, but it is be not the most efficient way for some scenarios. In the current official standard Go compiler implementation (version 1.19), channels are slower than the other synchronization ways. This could be proved by the following benchmark code.

```
package channels

import (
   "sync"
   "sync/atomic"
   "testing"
)

var g int32

func Benchmark_NoSync(b *testing.B) {
   for i := 0; i < b.N; i++ {
      g++
   }
}

func Benchmark_Atomic(b *testing.B) {
```

```
   for i := 0; i < b.N; i++ {
      atomic.AddInt32(&g, 1)
   }
}

var m sync.Mutex
func Benchmark_Mutex(b *testing.B) {
   for i := 0; i < b.N; i++ {
      m.Lock()
      g++
      m.Unlock()
   }
}

var ch = make(chan struct{}, 1)
func Benchmark_Channel(b *testing.B) {
   for i := 0; i < b.N; i++ {
      ch <- struct{}{}
      g++
      <-ch
   }
}
```

The benchmark results:

```
Benchmark_NoSync-4    2.246 ns/op
Benchmark_Atomic-4    7.112 ns/op
Benchmark_Mutex-4    14.25 ns/op
Benchmark_Channel-4  61.44 ns/op
```

From the results, we could find that using channels to concurrently increase a value is much slower that the other synchronization ways. The atomic

way is the best.

If it is possible, we should try to not share a value between multiple goroutines, so that we don't need do synchronizations at all for the value.

Use one channel instead of several ones to avoid using `select` blocks

For a `select` code block, the more `case` branches are in it, the more CPU consuming the code block is. This could be proved by the following benchmark code.

```
package channels

import "testing"

var ch1 = make(chan struct{}, 1)
var ch2 = make(chan struct{}, 1)

func Benchmark_Select_OneCase(b *testing.B) {
   for i := 0; i < b.N; i++ {
      select {
      case ch1 <- struct{}{}:
         <-ch1
      }
   }
}

func Benchmark_Select_TwoCases(b *testing.B) {
   for i := 0; i < b.N; i++ {
      select {
```

```
      case ch1 <- struct{}{}:
         <-ch1
      case ch2 <- struct{}{}:
         <-ch2
      }
   }
}
```

The benchmark results:

```
Benchmark_Select_OneCase-4    58.90 ns/op
Benchmark_Select_TwoCases-4  115.3 ns/op
```

So we should try to limit the number of case branches within a `select` code block.

The official standard Go compiler treats a `select` code block with only one `case` branch (and without `default` branch) as a simple general channel operation.

For some cases, we could merge multiple channels as one, to avoid the performance loss on executing multi-case `select` code blocks. We could use an interface type or a struct type as the channel element type to achieve this goal. If the channel element type is interface, then we can use a type switch to distinguish message kinds. If the channel element type is struct, then we can check which field is set to distinguish message kinds. The following benchmark code shows the performance differences between these ways.

```
package channels

```

```
import "testing"

var vx int
var vy string

func Benchmark_TwoChannels(b *testing.B) {
   var x = make(chan int)
   var y = make(chan string)
   go func() { for {x <- 1} }()
   go func() { for {y <- "hello"} }()
   b.ResetTimer()
   for i := 0; i < b.N; i++ {
      select {
      case vx = <-x:
      case vy = <-y:
      }
   }
}

func Benchmark_OneChannel_Interface(b *testing.B) {
   var x = make(chan interface{})
   go func() { for {x <- 1} }()
   go func() { for {x <- "hello"} }()
   b.ResetTimer()
   for i := 0; i < b.N; i++ {
      select {
      case v := <-x:
         switch v := v.(type) {
         case int: vx = v
         case string: vy = v
         }
      }
   }
```

```
}

func Benchmark_OneChannel_Struct(b *testing.B) {
   type T struct { x int; y string }
   var x = make(chan T)
   go func() { for {x <- T{x: 1}} }()
   go func() { for {x <- T{y: "hello"}} }()
   b.ResetTimer()
   for i := 0; i < b.N; i++ {
      v := <-x
      if v.y != "" {
         vy = v.y
      } else {
         vx = v.x
      }
   }
}
```

The benchmark results:

```
Benchmark_TwoChannels-4          1295 ns/op
Benchmark_OneChannel_Interface-4  940.9 ns/op
Benchmark_OneChannel_Struct-4     851.0 ns/op
```

From the results, we could get that two-case `select` code blocks are slower than one-case `select` code blocks. In other words, for some cases, merging several channels into one is a good way to improve program performance.

Try-send and try-receive `select` code blocks are specially optimized

A try-send or try-receive `select` code block contains one `default` branch and exact one `case` branch. Such code blocks are specially optimized by the official standard Go compiler, so their executions are very fast. This could be proved by the following benchmark code:

```
package channels

import "testing"

var c = make(chan struct{})

func Benchmark_TryReceive(b *testing.B) {
   for i := 0; i < b.N; i++ {
      select {
      case <-c:
      default:
      }
   }
}

func Benchmark_TrySend(b *testing.B) {
   for i := 0; i < b.N; i++ {
      select {
      case c <- struct{}{}:
      default:
      }
   }
}
```

The benchmark results:

```
Benchmark_TryReceive-4  5.646 ns/op
Benchmark_TrySend-4     5.293 ns/op
```

From the above results and the results shown in the first section of the current chapter, we could get that a try-send or try-receive code block is much less CPU consuming than a normal channel receive or send channel operation.

Functions

Function inlining

The official standard Go compiler will automatically inline some small functions to improve code execution speed.

Let's look at an example:

```
// inline.go
package inline

func bar(a, b int) int {
   return a*a - b*b + 2 * (a - b)
}

func foo(x, y int) int {
   var a = bar(x, y)
   var b = bar(y, x)
   var c = bar(a, b)
   var d = bar(b, a)
   return c*c + d*d
}
```

Build it with single `-m` compiler option:

```
$ go build -gcflags="-m" inline.go
# command-line-arguments
./inline.go:4:6: can inline bar
./inline.go:9:13: inlining call to bar
./inline.go:10:13: inlining call to bar
```

```
./inline.go:11:13: inlining call to bar
./inline.go:12:13: inlining call to bar
```

From the output, we know that the compiler thinks the `bar` function is inline-able, so the `bar` function calls within the `foo` function will be automatically flattened as:

```
func foo(x, y int) int {
   var a = x*y - y*y + 2 * (x - y)
   var b = y*y - x*x + 2 * (y - x)
   var c = a*a - b*b + 2 * (a - b)
   var d = b*b - a*a + 2 * (b - a)
   return c*c + d*d
}
```

After the flattening, some stack operations originally happening when calling the `bar` functions are saved so that code execution performance gets improved.

Inlining will make generated Go binaries larger, so compilers only inline calls to small functions.

Which functions are inline-able?

How small for a function is enough to be capable of being inlined? Each statement within a function has an inline cost. If the sum inline cost of all the statements within a function doesn't exceed the threshold set by the compiler, then the compiler thinks calls to the function could be inlined.

We can use double `-m` compiler option to show why some functions are inline-able but others aren't. Still use the above example:

```
$ go build -gcflags="-m -m" inline.go
# command-line-arguments
./inline.go:4:6: can inline bar with cost 14 as: ...
./inline.go:8:6: cannot inline foo: ... cost 96 exceeds budget
80
...
```

From the output, we could learn that the `foo` function is not inline-able, for its inline cost is 96, which exceeds the inline threshold (80).

Recursive functions will never get inlined. For example, the `sumSquares` function shown in the following code is not inline-able.

```
package main

func sumSquares(n uint) uint {
   if n == 0 {
      return 0
   }
   return sumSquares(n-1) + n*n
}

func main() {
   println(sumSquares(5))
}
```

Besides the above rules, for various reasons, currently (v1.19), the official standard Go compiler never inlines functions containing:

- builtin `recover` function calls
- type declarations
- `defer` calls and `go` calls

For example, in the following code, the official standard Go compiler (v1.19) thinks all of the `fN` functions are inline-able but none of the `gN` functions are.

```
func f1(s []int) int {
   return cap(s) - len(s)
}

func g1(s []int) int {
   recover()
   return cap(s) - len(s)
}

func f2(b bool) string {
   if b {
      return "y"
   }
   return "N"
}

func g2(b bool) string {
   type  _ int
   if b {
      return "y"
   }
   return "N"
}
```

```
func f3(c chan int) int {
   return <-c
}

func f4(a, b int) int {
   return a*a - b*b
}

func g4(a, b int) int {
   defer func(){}()
   return a*a - b*b
}

func f5(a, b int) int {
   return a*a - b*b
}

func g5(a, b int) int {
   go func(){}()
   return a*a - b*b
}
```

The official standard Go compiler might change inline strategies, the inline cost threshold and statement inline costs from version to version. For example, in the above example, * before v1.16, the compiler thought the function `f6` in the following code was not inline-able (but since v1.16, it has become inline-able). * before v1.18, the compiler thought the function `f7` in the following code was not inline-able (but since v1.18, it has become inline-able). * before v1.19, the compiler thought the function `f8` in the following code was not inline-able (but since v1.19, it has become inline-able).

```
func f6(s []int) {
   for i := 0; i < len(s); i++ {
      s[i] = i
   }
}

func f7(s []int) {
   for i := range s {
      s[i] = i
   }
}

func f8(c chan int) int {
   select {
   case r := <-c:
      return r
   }
}
```

As another example, the `plusSquare` function shown in the following code in inline-able only since Go v1.17. Before v1.17, a function containing closures was thought as not inline-able.

```
package main

func plusSquare(n int) int {
   f := func(x int) int { return x*x }
   return f(n) + 2*n + 1
}

func main() {
```

```
    println(plusSquare(5))
}
```

A call to a function value is not inline-able if the value is hard to be determined at compile time

For example, in the following code, the call to the package-level `addFunc` function variable is not inline-able, because the compiler doesn't confirm the values of package-level variables at compile time (for compilation speed consideration). However, it does try to confirm the values of local variables at compile time, if it is possible. So the call to local `addFunc` function (variable) will get inlined.

```
package main

func add(a, b int) int {
    return a + b
}

var addFunc = add
func main() {
    println(addFunc(11, 22)) // not inlined
    var addFunc = add
    println(addFunc(11, 22)) // inlined
}
```

The `go:noinline` comment directive

Sometimes, we might want calls to a function to never get inlined, for study and testing purposes, or to make a caller function of the function inline-able

(see below for an example), etc. Besides the several ways introduced above, we could also use the `go:noinline` comment directive to achieve this goal. For example, the compiler will not inline the call to the `add` function in the following code, even if the `add` function is very simple.

```
package main

//go:noinline
func add(x, y int) int {
   return x + y
}

func main() {
   println(add(1, 2))
}
```

However, please note that this is not a formal way to avoid inlining. It is mainly intended to be used in standard package and Go toolchain developments. But personally, I think this directive will be supported in a long term.

Write code in the ways which are less inline costly

Generally, we should try to make more functions inline-able, to get better program execution performances.

Besides the rules introduce above, we should know that different code implementation ways might have different inline costs, even if the code differences are subtle. We could make use of this fact to try different implementation ways to find out which way has the lowest inline cost.

Let's use the first example shown above again.

```
// inline2.go
package inline

func bar(a, b int) int {
   return a*a - b*b + 2 * (a - b)
}

func foo(x, y int) int { // line 8
   var a = bar(x, y)
   var b = bar(y, x)
   var c = bar(a, b)
   var d = bar(b, a)
   return c*c + d*d
}

func foo2(x, y int) int { // line 16
   var a = x*y - y*y + 2 * (x - y)
   var b = y*y - x*x + 2 * (y - x)
   var c = a*a - b*b + 2 * (a - b)
   var d = b*b - a*a + 2 * (b - a)
   return c*c + d*d
}
```

Build it with double `-m` options:

```
$ go build -gcflags="-m -m" inline2.go
...
./inline2.go:8:6: cannot inline foo: function too complex: cost
96 exceeds budget 80
...
./inline2.go:16:6: can inline foo2 with cost 76 as: ...
```

From the outputs, we could learn that although the compiler thinks the `foo` function is not inline-able, but it thinks its manual-flattened version (the `foo2` function) is inline-able, because the inline cost of the `foo2` function calculated by the compiler is 76, which doesn't exceed the inline threshold (80). Yes, manual-inlining is often less costly than compiler auto-inlining. And, in practice, manual inlined code is indeed often comparatively more performant (see below for an example).

Another example:

```
// sum.go
package inline

func sum1(s []int) int { // line 4
   var r = 0
   for i := 0; i < len(s); i++ {
      r += s[i]
   }
   return r
}

func sum2(s []int) (r int) { // line 12
   for i := 0; i < len(s); i++ {
      r += s[i]
   }
   return r
}

func sum3(s []int) (r int) { // line 19
   for i := 0; i < len(s); i++ {
      r += s[i]
```

```
    }
    return
}
```

The compiler (v1.19) thinks the inline costs of the `sumN` functions are different, which could be verified from the following outputs:

```
$ go build -gcflags="-m -m" sum.go
# command-line-arguments
./sum.go:4:6: can inline sum1 with cost 25 as: ...
./sum.go:12:6: can inline sum2 with cost 20 as: ...
./sum.go:19:6: can inline sum3 with cost 19 as: ...
...
```

The calculated inline costs of the three functions are 25, 20 and 19, respectively. From the above example, we could get:

- local variable declarations contributes to inline costs.
- bare `return` statements are less inline costly than non-bare `return` statements.

(Please note that code inline costs don't mean code execution costs. In fact, the official standard Go compiler generate identical assembly instructions for the above `sumN` functions.)

The third example:

```
// branches.go
package inline

func foo(a, b, c, d bool) int { // line 4
    if a { return 0 }
```

```
   if b { return 0 }
   if c { return 0 }
   if d { return 0 }
   return 1
}

func foo2(a, b, c, d bool) (r int) { // line 12
   if a { return }
   if b { return }
   if c { return }
   if d { return }
   return 1
}

func bar(a, b, c, d bool) int { // line 20
   if a || b || c || d {
      return 0
   }
   return 1
}

func bar2(a, b, c, d bool) (r int) { // line 27
   if a || b || c || d {
      return
   }
   return 1
}
```

The following outputs show less branches lead to lower inline costs.

```
$ go build -gcflags="-m -m" branches.go
# command-line-arguments
./branches.go:4:6: can inline foo with cost 18 as: ...
```

```
./branches.go:12:6: can inline foo2 with cost 14 as: ...
./branches.go:20:6: can inline bar with cost 12 as: ...
./branches.go:27:6: can inline bar2 with cost 11 as: ...
```

The 4th example:

```
// funcvalue.go
package inline

func add(x, y int) int {
   return x+y
}

func sub(x, y int) int {
   return x-y
}

func op1(op string, a, b int) int { // line 12
   switch op {
   default: panic("unknown op " + op)
   case "+": return add(a, b) // inlining call to add
   case "-": return sub(a, b) // inlining call to sub
   }
}

func op2(op string, a, b int) int { // line 20
   var f func(int, int) int
   switch op {
   default: panic("unknown op " + op)
   case "+": f = add
   case "-": f = sub
   }
```

```
    return f(a, b)
}
```

The following outputs show the inline cost of the function `op2` is much higher than the function `op1`:

```
$ go build -gcflags="-m -m" funcvalue.go
...
./aaa.go:12:6: can inline op1 with cost 30 as: ...
...
./aaa.go:20:6: cannot inline op2: ...: cost 84 exceeds budget
80
...
```

Since v1.18, the official standard Go compiler thinks the inline cost of `for-range` loop is smaller than a plain `for` loop.

```
func f6(s []int) { // inline code is 18
    for i := 0; i < len(s); i++ {
        s[i] = i
    }
}

func f7(s []int) { // inline code is 10
    for i := range s {
        s[i] = i
    }
}
```

The official standard Go compiler might be improved in future versions so that the calculated costs in the examples shown in this section will become more consistent.

Make hot paths inline-able

If a function is not inline-able but contains a starting hot path, to make the hot path inline-able, we could wrap the cold parth part into another function.

For example, the `concat` function in the following code is not inline-able, for its inline cost is 85 (larger than the threshold 80).

```
func concat(bss ...[]byte) []byte {
   n := len(bss)
   if n == 0 {
      return nil
   } else if n == 1 {
      return bss[0]
   } else if n == 2 {
      return append(bss[0], bss[1]...)
   }

   var m = 0
   for i := 0; i < len(bss); i++ {
      m += len(bss[i])
   }
   var r = make([]byte, 0, m)
   for i := 0; i < len(bss); i++ {
      r = append(r, bss[i]...)
   }
   return r
}
```

If, in practice, most cases are concatenating two byte slices, then we could rewrite the above code as the following shown. Now the inline cost of the `concat` function becomes 74 so that it is inline-able now. That means the hot path will be always inlined.

```
func concat(bss ...[]byte) []byte {
   if len(bss) == 2 {
      return append(bss[0], bss[1]...)
   }

   return concatSlow(bss...)
}

//go:noinline
func concatSlow(bss ...[]byte) []byte {
   if len(bss) == 0 {
      return nil
   } else if len(bss) == 1 {
      return bss[0]
   }

   var m = 0
   for i := 0; i < len(bss); i++ {
      m += len(bss[i])
   }
   var r = make([]byte, 0, m)
   for i := 0; i < len(bss); i++ {
      r = append(r, bss[i]...)
   }
   return r
}
```

If the inline cost of the function wrapping the code path part doesn't exceed the inline threshold, then we should use the above introduced avoid-being-inlined ways to prevent the function from being inline-able. Otherwise, the rewritten `concat` function is still not inline-able, for the wrapped part be automatically flattened into the rewritten function. That is why the `go:noinline` comment directive is put before the rewritten `concatSlow` function.

Please note that, currently (Go toolchain v1.19), the inline cost of a function call is 59. That means a function is not inline-able if it contains 2+ calls which are not inlined.

And please note that, since Go toolchain v1.18, if we replace the two plain `for` loops within the original `concat` function with two `for-range` loops, then the original function will become inline-able already. Here, for demo purpose, we use two plain `for` loops.

Manual inlining is often more performance than auto-inlining

An example:

```
package functions

import "testing"

const N = 100

// This function is inline-able.
func Slice2Array(b []byte) [N]byte {
   return *(*[N]byte)(b)
```

```
}

//====================

var buf = make([]byte, 8192)
var r [128][N]byte

func Benchmark_ManualInline(b *testing.B) {
   for i := 0; i < b.N; i++ {
      r[i&127] = *(*[N]byte)(buf)
   }
}

func Benchmark_AutoInline(b *testing.B) {
   for i := 0; i < b.N; i++ {
      r[i&127] = Slice2Array(buf) // inlined
   }
}
```

The benchmark results:

```
Benchmark_ManualInline-4  5.134 ns/op
Benchmark_AutoInline-4   11.49 ns/op
```

From the results, we could find that manual-inlining is more performance than auto-inlining, at least for this specified case.

The official standard Go compiler might be improved in future versions so that automatic inlining will become more smart.

Inlining might do negative impact on performance

For some compiler implementation flaws, sometimes, inlined calls might perform worse than not-inlined ones. Here is an example:

```
package functions

import "testing"

type T [1<<8]byte

var r, s T

//go:noinline
func not_inline_able(x1, y1 *T) {
   x, y := x1[:], y1[:]
   for k := 0; k < len(T{}); k++ {
      x[k] = y[k]
   }
}

func inline_able(x1, y1 *T) {
   x, y := x1[:], y1[:]
   for k := 0; k < len(T{}); k++ {
      x[k] = y[k]
   }
}

func Benchmark_not_inlined(b *testing.B) {
   for i := 0; i < b.N; i++ {
      not_inline_able(&r, &s)
   }
}

func Benchmark_auto_inlined(b *testing.B) {
```

```
    for i := 0; i < b.N; i++ {
        inline_able(&r, &s)
    }
}

func Benchmark_manual_inlined(b *testing.B) {
    for i := 0; i < b.N; i++ {
        for k := 0; k < len(T{}); k++ {
            r[k] = s[k]
        }
    }
}
```

The benchmark results:

```
Benchmark_not_inlined-4     127.9 ns/op
Benchmark_auto_inlined-4    196.4 ns/op
Benchmark_manual_inlined-4  196.4 ns/op
```

The implementation flaw (in the official standard Go compiler v1.19) presents when the manipulated values are global (package-level) arrays.

Future official standard compiler versions might fix the flaw.

Pointer parameters/results vs. non-pointer parameters/results

Arguments and return values of functions calls are passed by copy in Go. So using large-size types as the parameter/result types of a function causes large value copy costs when invoking calls to the function.

Large parameter/result types also increase the possibility of stack growing.

To avoid the high argument copy costs caused by a large-size parameter type `T`, we could use the pointer type `*T` as the parameter type instead. However, please note that pointer parameters/results have their own drawbacks. For some scenarios, they might cause more heap allocations.

The following code shows the effect of value copy costs.

```
package functions

import "testing"

type T5 struct{a, b, c, d, e float32}
var t5 T5

//go:noinline
func Add5_TT_T(x, y T5) (z T5) {
   z.a = x.a + y.a
   z.b = x.b + y.b
   z.c = x.c + y.c
   z.d = x.d + y.d
   z.e = x.e + y.e
   return
}

//go:noinline
func Add5_PPP(z, x, y *T5) {
   z.a = x.a + y.a
   z.b = x.b + y.b
   z.c = x.c + y.c
   z.d = x.d + y.d
```

```
    z.e = x.e + y.e
}

func Benchmark_Add5_TT_T(b *testing.B) {
    for i := 0; i < b.N; i++ {
        var x, y, z T5
        z = Add5_TT_T(x, y)
        t5 = z
    }
}

func Benchmark_Add5_PPP(b *testing.B) {
    for i := 0; i < b.N; i++ {
        var x, y, z T5
        Add5_PPP(&z, &x, &y)
        t5 = z
    }
}
```

The benchmark results:

```
Benchmark_Add5_TT_T-4  17.73 ns/op
Benchmark_Add5_PPP-4   11.95 ns/op
```

From the above results, we get that the function `Add5_PPP` is more efficient than the function `Add5_TT_T`.

For small-size types, the benchmarks results will invert. The reason is the official standard Go compiler specially optimizes some operations on small-size values.

```
package functions

import "testing"

type T4 struct{a, b, c, d float32}
var t4 T4

//go:noinline
func Add4_TT_T(x, y T4) (z T4) {
   z.a = x.a + y.a
   z.b = x.b + y.b
   z.c = x.c + y.c
   z.d = x.d + y.d
   return
}

//go:noinline
func Add4_PPP(z, x, y *T4) {
   z.a = x.a + y.a
   z.b = x.b + y.b
   z.c = x.c + y.c
   z.d = x.d + y.d
}

func Benchmark_Add4_TT_T(b *testing.B) {
   for i := 0; i < b.N; i++ {
      var x, y, z T4
      z = Add4_TT_T(x, y)
      t4 = z
   }
}

func Benchmark_Add4_PPP(b *testing.B) {
```

```
   for i := 0; i < b.N; i++ {
      var x, y, z T4
      Add4_PPP(&z, &x, &y)
      t4 = z
   }
}
```

The new benchmark results:

```
Benchmark_Add4_TT_T-4  2.716 ns/op
Benchmark_Add4_PPP-4   9.006 ns/op
```

Named results vs. anonymous results

It is often said that generally named results make function more performant. This is true for most cases, but not for some ones. For example, in the following two `ConvertToArray` implementations, the one with named results is slower than the one with anonymous results.

```
package functions

import "testing"

const N = 1<<12
var buf = make([]byte, N)
var r [128][N]byte

func Benchmark_ConvertToArray_Named(b *testing.B) {
   for i := 0; i < b.N; i++ {
      r[i&127] = ConvertToArray_Named(buf)
   }
}
```

```
func Benchmark_ConvertToArray_Unnamed(b *testing.B) {
   for i := 0; i < b.N; i++ {
      r[i&127] = ConvertToArray_Unnamed(buf)
   }
}

func ConvertToArray_Named(b []byte) (ret [N]byte) {
   // type _ int
   ret = *(*[N]byte)(b)
   return
}

func ConvertToArray_Unnamed(b []byte) [N]byte {
   // type _ int
   return *(*[N]byte)(b)
}
```

The benchmark result:

```
Benchmark_ConvertToArray_Named-4    472.2 ns/op
Benchmark_ConvertToArray_Unnamed-4  332.9 ns/op
```

From the results, we could find that the function with a named result performs slower. It looks this is a problem related to code inlining. If the two `type _ int` lines are enabled (to prevent the calls to the two functions from being inlined), then there is no performance difference between the two functions. The future compiler versions might remove the performance difference when the two functions are both inline-able.

The following two `CopyToArray` implementations shows the opposite result. the one with anonymous results is slower than the one with named results, whether or not the two functions are inlined.

```
package functions

import "testing"

const N = 1<<12
var buf = make([]byte, N)
var r [128][N]byte

func Benchmark_CopyToArray_Named(b *testing.B) {
   for i := 0; i < b.N; i++ {
      r[i&127] = CopyToArray_Named(buf)
   }
}

func Benchmark_CopyToArray_Unnamed(b *testing.B) {
   for i := 0; i < b.N; i++ {
      r[i&127] = CopyToArray_Unnamed(buf)
   }
}

func CopyToArray_Named(b []byte) (ret [N]byte) {
   // type _ int
   copy(ret[:], b)
   return
}

func CopyToArray_Unnamed(b []byte) [N]byte {
   // type _ int
```

```
   var ret [N]byte
   copy(ret[:], b)
   return ret
}
```

The benchmark results:

```
Benchmark_CopyToArray_Named-4    408.3 ns/op
Benchmark_CopyToArray_Unnamed-4  547.5 ns/op
```

Try to store intermediate calculation results in local variables

Using local variables to store intermediate calculation result will make code get a chance to be better optimized by the compiler so that a better performance will be got. An example:

```
package functions

import "testing"

var sum int

func f(s []int) {
   for _, v := range s {
      sum += v
   }
}

func g(s []int) {
   var n = 0
   for _, v := range s {
```

```
        n += v
    }
    sum = n
}

var s = make([]int, 1024)

func Benchmark_f(b *testing.B) {
    for i := 0; i < b.N; i++ {
        f(s)
    }
}
func Benchmark_g(b *testing.B) {
    for i := 0; i < b.N; i++ {
        g(s)
    }
}
```

The benchmark results:

```
Benchmark_f-4  3293 ns/op
Benchmark_g-4   654.1 ns/op
```

Avoid deferred calls in loops

Since v1.14, the official standard Go compiler has specially optimized open-ended deferred calls (the deferred calls which are not within loops). But the costs of deferred calls within loops are still high. This could be proved from the following example.

```
package functions
```

```
import "testing"

var n int
func inc() {
   n++
}

func f(n int) {
   for i := 0; i < n; i++ {
      defer inc()
      inc()
   }
}

func g(n int) {
   for i := 0; i < n; i++ {
      func() {
         defer inc()
         inc()
      }()
   }
}

func Benchmark_f(b *testing.B) {
   n = 0
   for i := 0; i < b.N; i++ {
      f(100)
   }
}

func Benchmark_g(b *testing.B) {
   n = 0
   for i := 0; i < b.N; i++ {
```

```
        g(100)
    }
}
```

The benchmark results:

```
Benchmark_f-4  61797 ns/op  4 B/op  0 allocs/op
Benchmark_g-4   5990 ns/op  0 B/op  0 allocs/op
```

The reason why the function `g` is much more performant than the function `f` is that deferred calls which are not directly in loops are specially optimized by the official standard Go compiler. The function `g` wraps the code in the loop into an anonymous function call so that the deferred call is not directly enclosed in the loop.

Please note that, the two functions are not equivalent to each other in logic. If this is a problem, then the anonymous function call trick should not be used.

Avoid using deferred calls if extreme high performance is demanded

Even if the official standard Go compiler has specially optimized open-ended deferred calls, such a deferred call still has a small extra overhead comparing non-deferred calls. And as above mentioned, currently (Go toolchain v1.19), a function containing deferred calls is not inline-able. So please try to avoid using deferred calls in a piece of code if extreme high performance is demanded for the piece of code.

The arguments of a function call will be always evaluated when the call is invoked

For example, the following program prints `1`, which means the string concatenation expression `h + w` is evaluated, even if the parameter `s` is actually not used in the `debugPrint` function at run time.

```
package main

import (
   "log"
   "testing"
)

var debugOn = false

func debugPrint(s string) {
   if debugOn {
      log.Println(s)
   }
}

func main() {
   stat := func(f func()) int {
      allocs := testing.AllocsPerRun(1, f)
      return int(allocs)
   }

   var h, w = "hello ", "world!"

   var n = stat(func(){
      debugPrint(h + w)
```

```
    })

    println(n) // 1
}
```

One way to avoid the unnecessary argument evaluation is to change the `debugOn` value to a constant. But some programs might need to change the value on the fly, so this way is not always feasible. We could let the `debugPrint` function return a `bool` result and call the function in a boolean-and operation, like the following code shows:

```
package main

import (
    "log"
    "testing"
)

var debugOn = false

func debugPrint(s string) bool {
    log.Println(s)
    return true // or false
}

func main() {
    stat := func(f func()) int {
        allocs := testing.AllocsPerRun(1, f)
        return int(allocs)
    }

    var h, w = "hello ", "world!"
```

```
    var n = stat(func(){
        _ = debugOn && debugPrint(h + w)
    })

    println(n) // 0
}
```

In the above code, the string concatenation expression `h + w` is not evaluated, because the `debugPrint` function call is not invoked at all.

Try to make less values escape to heap in the hot paths

Assume most calls to the function `f` shown in the following code return from the `if` code block (most arguments are in the range `[0, 9]`), then the implementation of the function `f` is not very efficient, because the argument `x` will escape to heap.

```
package main

import "strconv"

func f(x int) string { // x escapes to heap
    if x >= 0 && x < 10 {
        return "0123456789"[x:x+1]
    }

    return g(&x)
}
```

```
func g(x *int) string {
   escape(x) // for demo purpose
   return strconv.Itoa(*x)
}

var sink interface{}

//go:noinline
func escape(x interface{}) {
   sink = x
   sink = nil
}

func main() {
   var a = f(100)
   println(a)
}
```

By making use of the trick introduced in [the stack and escape analysis] article, we could rewrite the function `f` as the following shows, to prevent the argument `x` from escaping to heap.

```
func f(x int) string {
   if x >= 0 && x < 10 {
      return "0123456789"[x:x+1]
   }

   x2 := x // x2 escapes to heap
   return g(&x2)
}
```

Interfaces

Box values into and unbox values from interfaces

An interface value could be viewed as a box to hold at most one non-interface value. A nil interface value holds nothing. On the contrary, a type assertion could be viewed as a value unboxing operation.

When a non-interface value is assigned to an interface value, generally, a copy of the non-interface value will be boxed in the interface value. In the official standard Go compiler implementation, generally, the copy of the non-interface value is allocated somewhere and its address is stored in the interface value.

So generally, the cost of boxing a value is approximately proportional to the size of the value.

```
package interfaces

import "testing"

var r interface{}

var n16 int16 = 12345
func Benchmark_BoxInt16(b *testing.B) {
   for i := 0; i < b.N; i++ { r = n16 }
}

var n32 int32 = 12345
```

```
func Benchmark_BoxInt32(b *testing.B) {
   for i := 0; i < b.N; i++ { r = n32 }
}

var n64 int64 = 12345
func Benchmark_BoxInt64(b *testing.B) {
   for i := 0; i < b.N; i++ { r = n64 }
}

var f64 float64 = 1.2345
func Benchmark_BoxFloat64(b *testing.B) {
   for i := 0; i < b.N; i++ { r = f64 }
}

var s = "Go"
func Benchmark_BoxString(b *testing.B) {
   for i := 0; i < b.N; i++ { r = s }
}

var x = []int{1, 2, 3}
func Benchmark_BoxSlice(b *testing.B) {
   for i := 0; i < b.N; i++ { r = x }
}

var a = [100]int{}
func Benchmark_BoxArray(b *testing.B) {
   for i := 0; i < b.N; i++ { r = a }
}
```

The benchmark results:

```
Benchmark_BoxInt16-4    18.44 ns/op   2 B/op  1 allocs/op
Benchmark_BoxInt32-4    43.53 ns/op   4 B/op  1 allocs/op
```

```
Benchmark_BoxInt64-4    55.92 ns/op   8 B/op  1 allocs/op
Benchmark_BoxFloat64-4  62.63 ns/op   8 B/op  1 allocs/op
Benchmark_BoxString-4  104.1 ns/op   16 B/op  1 allocs/op
Benchmark_BoxSlice-4   114.6 ns/op   24 B/op  1 allocs/op
Benchmark_BoxArray-4   591.9 ns/op  896 B/op  1 allocs/op
```

From the above benchmark results, we could get that each value boxing operation generally needs one allocation, and the size of the allocated memory block is the same as the size of the boxed value.

The official standard Go compiler makes some optimizations so that the general rule mentioned above is not always obeyed. One optimization made by the official standard Go compiler is that no allocations are made when boxing zero-size values, boolean values and 8-bit integer values.

```
package interfaces

import "testing"

var r interface{}

var v0 struct{}
func Benchmark_BoxZeroSize1(b *testing.B) {
   for i := 0; i < b.N; i++ { r = v0 }
}

var a0 [0]int64
func Benchmark_BoxZeroSize2(b *testing.B) {
   for i := 0; i < b.N; i++ { r = a0 }
}

var b bool
```

```
func Benchmark_BoxBool(b *testing.B) {
   for i := 0; i < b.N; i++ { r = b }
}

var n int8 = -100
func Benchmark_BoxInt8(b *testing.B) {
   for i := 0; i < b.N; i++ { r = n }
}
```

The benchmark results:

```
Benchmark_BoxZeroSize1-4  1.133 ns/op  0 B/op  0 allocs/op
Benchmark_BoxZeroSize2-4  1.180 ns/op  0 B/op  0 allocs/op
Benchmark_BoxBool-4       1.180 ns/op  0 B/op  0 allocs/op
Benchmark_BoxInt8-4       1.822 ns/op  0 B/op  0 allocs/op
```

From the results, we could get that boxing zero-size values, boolean values and 8-bit integer values doesn't make memory allocations, which is one reason why such boxing operations are much faster.

Another optimization made by the official standard Go compiler is that no allocations are made when boxing pointer values into interfaces. Thus, boxing pointer values is often much faster than boxing non-pointer values.

The official standard Go compiler represents (the direct parts of) maps, channels and functions as pointers internally, so boxing such values is also as faster as boxing pointers.

This could be proved from the following code:

```
package interfaces
```

```
import "testing"

var r interface{}

var p = new([100]int)
func Benchmark_BoxPointer(b *testing.B) {
   for i := 0; i < b.N; i++ { r = p }
}

var m = map[string]int{"Go": 2009}
func Benchmark_BoxMap(b *testing.B) {
   for i := 0; i < b.N; i++ { r = m }
}

var c = make(chan int, 100)
func Benchmark_BoxChannel(b *testing.B) {
   for i := 0; i < b.N; i++ { r = c }
}

var f = func(a, b int) int {return a + b}
func Benchmark_BoxFunction(b *testing.B) {
   for i := 0; i < b.N; i++ { r = f }
}
```

The benchmark results:

```
Benchmark_BoxPointer-4   1.192 ns/op  0 B/op  0 allocs/op
Benchmark_BoxMap-4       1.182 ns/op  0 B/op  0 allocs/op
Benchmark_BoxChannel-4   1.216 ns/op  0 B/op  0 allocs/op
Benchmark_BoxFunction-4  1.126 ns/op  0 B/op  0 allocs/op
```

From the above results, we could get that boxing pointer values is very fast and doesn't make memory allocations. This explains the reason why

declaring a method for `*T` is often more efficient than for `T` if we intend to let the method implement an interface method.

By making use of this optimization, for some use cases, we could use a loop-up table to convert some non-pointer values in a small set into pointer values. For example, in the following code, we use an array to convert `uint16` values into pointers to get much lower value boxing costs.

```
package interfaces

import "testing"

var values [65536]uint16

func init() {
   for i := range values {
      values[i] = uint16(i)
   }
}

var r interface{}

func Benchmark_Box_Normal(b *testing.B) {
   for i := 0; i < b.N; i++ {
      r = uint16(i)
   }
}

func Benchmark_Box_Lookup(b *testing.B) {
   for i := 0; i < b.N; i++ {
      r = &values[uint16(i)]
   }
```

```
}

var x interface{} = uint16(10000)byte
var n uint16

func Benchmark_Unbox_Normal(b *testing.B) {
   for i := 0; i < b.N; i++ {
      n = x.(uint16)
   }
}

func Benchmark_Unbox_Lookup(b *testing.B) {
   for i := 0; i < b.N; i++ {
      n = *y.(*uint16)
   }
}
```

The benchmark results:

```
Benchmark_Box_Normal-4   22.69 ns/op    1 B/op  0 allocs/op
Benchmark_Box_Lookup-4    1.137 ns/op   0 B/op  0 allocs/op
Benchmark_Unbox_Normal-4  0.7535 ns/op  0 B/op  0 allocs/op
Benchmark_Unbox_Lookup-4  0.7510 ns/op  0 B/op  0 allocs/op
```

(Please note that, the results show the `Box_Normal` function makes zero allocations, which is not true. The value is about `0.99`, which is truncated to zero. The reason is boxing values within [0, 255] doesn't allocate, which will be mentioned below.)

The same optimization is also applied for boxing constant values:

```
package interfaces

import "testing"

var r interface{}

const N int64 = 12345
func Benchmark_BoxInt64(b *testing.B) {
   for i := 0; i < b.N; i++ { r = N }
}

const F float64 = 1.2345
func Benchmark_BoxFloat64(b *testing.B) {
   for i := 0; i < b.N; i++ { r = F }
}

const S = "Go"
func Benchmark_BoxConstString(b *testing.B) {
   for i := 0; i < b.N; i++ { r = S }
}
```

The benchmark results:

```
Benchmark_BoxInt64-4         1.136 ns/op  0 B/op  0 allocs/op
Benchmark_BoxFloat64-4       1.196 ns/op  0 B/op  0 allocs/op
Benchmark_BoxConstString-4   1.211 ns/op  0 B/op  0 allocs/op
```

In fact, the official standard Go compiler also makes similar optimizations for boxing the following values:

- non-constant small integer values (in range `[0, 255]`) of any integer types (expect for 8-bit ones, which have been covered in the first

optimization mentioned above).

- non-constant zero values of floating-point/string/slice types.

Box non-constant small integer values:

```
package interfaces

import "testing"

var r interface{}

func Benchmark_BoxSmallInt16(b *testing.B) {
   for i := 0; i < b.N; i++ { r = int16(i&255) }
}

func Benchmark_BoxSmallInt32(b *testing.B) {
   for i := 0; i < b.N; i++ { r = int32(i&255) }
}

func Benchmark_BoxSmallInt64(b *testing.B) {
   for i := 0; i < b.N; i++ { r = int64(i&255) }
}
```

The benchmark results:

```
Benchmark_BoxSmallInt16-4  3.409 ns/op  0 B/op  0 allocs/op
Benchmark_BoxSmallInt32-4  3.379 ns/op  0 B/op  0 allocs/op
Benchmark_BoxSmallInt64-4  3.404 ns/op  0 B/op  0 allocs/op
```

Box zero floating-point, string and slices values:

```
package interfaces
```

```
import "testing"

var r interface{}

var f32 float32 = 0
func Benchmark_BoxZeroFloat32(b *testing.B) {
   for i := 0; i < b.N; i++ { r = f32 }
}

var f64 float64 = 0
func Benchmark_BoxZeroFloat64(b *testing.B) {
   for i := 0; i < b.N; i++ { r = f64 }
}

var str = ""
func Benchmark_BoxBlankString(b *testing.B) {
   for i := 0; i < b.N; i++ { r = str }
}

var slc []int
func Benchmark_BoxNilSlice(b *testing.B) {
   for i := 0; i < b.N; i++ {
      r = slc
   }
}
```

The benchmark results:

```
Benchmark_BoxZeroFloat32-4  3.386 ns/op  0 B/op  0 allocs/op
Benchmark_BoxZeroFloat64-4  3.385 ns/op  0 B/op  0 allocs/op
Benchmark_BoxBlankString-4  3.427 ns/op  0 B/op  0 allocs/op
Benchmark_BoxNilSlice-4     3.930 ns/op  0 B/op  0 allocs/op
```

The cost of boxing a struct (array) value with only one field (element) which is a small integer or a zero bool/numeric/string/slice/pointer/map/channel/function value is the same as boxing that field (element) (before Go toolchain v1.19, except the only field/element is a a 8-bit integer).

```
package interfaces

import "testing"

var r interface{}

func Benchmark_SmallInt32Field(b *testing.B) {
   for i := 0; i < b.N; i++ {
      r = struct{ n int32 }{255}
   }
}

func Benchmark_ZeroFloat64Element(b *testing.B) {
   for i := 0; i < b.N; i++ {
      r = [1]float64{0}
   }
}

func Benchmark_BlankStringField(b *testing.B) {
   for i := 0; i < b.N; i++ {
      r = struct{ n string }{""}
   }
}

func Benchmark_NilSliceField(b *testing.B) {
   for i := 0; i < b.N; i++ {
```

```
      r = struct{ n []int }{nil}
   }
}

func Benchmark_NilFunctionElement(b *testing.B) {
   for i := 0; i < b.N; i++ {
      r = [1]func(){func(){}}
   }
}
```

Benchmark results:

```
Benchmark_SmallInt32Field-4     3.387 ns/op  0 B/op  0
allocs/op
Benchmark_ZeroFloat64Element-4  3.884 ns/op  0 B/op  0
allocs/op
Benchmark_BlankStringField-4    3.528 ns/op  0 B/op  0
allocs/op
Benchmark_NilSliceField-4       3.931 ns/op  0 B/op  0
allocs/op
Benchmark_NilFunctionElement-4  1.134 ns/op  0 B/op  0
allocs/op
```

A summary based on the above benchmark results (as of Go toolchain v1.19):

- Boxing pointer values is much faster than boxing non-pointer values.
- Boxing maps, channels and functions is as fast as boxing pointers.
- Boxing constant values is as fast as boxing pointers.
- Boxing zero-size values is as fast as boxing pointers.
- Boxing boolean and 8-bit integer values is as fast as boxing pointers.

- Boxing non-constant small values (in range `[0, 255]`) of any integer types (expect for 8-bit ones) is about 3 times slower than boxing a pointer value.
- Boxing floating-point/string/slice zero values is about 3 times slower than boxing a pointer value.
- Boxing a non-constant not-small integer value (out of range `[0, 255]`) or a non-zero floating-point value is about (or more than) 20 times slower than boxing a pointer value.
- Boxing non-nil slices or non-blank non-constant string values is about (or more than) 50 times slower than boxing a pointer values.
- Boxing a struct (array) value with only one field (element) which is a small integer or a zero bool/numeric/string/slice/point value is as faster as boxing that field (element).

So, if value boxing operations are made frequently on the hot paths of code execution, it is recommended to box values with small boxing costs.

Try to void memory allocations by assigning interface to interface

Sometimes, we need to box a non-interface value in two interface values. There are two ways to achieve the goal:

1. box the non-interface value in the first interface value then box the non-interface value in the second one.
2. box the non-interface value in the first interface value then assign the first interface to the second one.

If boxing the value needs an allocation, then which way is more performant? No doubly, the second way, which could be proved by the following benchmark code.

```
package main

import "testing"

var v = 9999999
var x, y interface{}

func Benchmark_BoxBox(b *testing.B) {
   for i := 0; i < b.N; i++ {
      x = v // needs one allocation
      y = v // needs one allocation
   }
}

func Benchmark_BoxAssign(b *testing.B) {
   for i := 0; i < b.N; i++ {
      x = v // needs one allocation
      y = x // no allocations
   }
}
```

The benchmark results:

```
Benchmark_BoxBox-4     130.5 ns/op  16 B/op  2 allocs/op
Benchmark_BoxAssign-4  68.31 ns/op   8 B/op  1 allocs/op
```

The second way saves one allocation so it is more performant.

In practice, the tip could be used when the same (non-interface) value is passed to print functions (which parameters are mostly of type `interface{}`) provided in the `fmt` standard package as multiple arguments. For example, in the following code, the second `fmt.Fprint` call in the following code is more performant than the first one, because it saves two allocations.

```
package main

import (
   "fmt"
   "io"
   "testing"
)

func main() {
   stat := func(f func()) int {
      allocs := testing.AllocsPerRun(100, f)
      return int(allocs)
   }

   var x = "aaa"

   var n = stat(func(){
      // 3 allocations
      fmt.Fprint(io.Discard, x, x, x)
   })
   println(n) // 3

   var m = stat(func(){
      var i interface{} = x // 1 allocation
      // No allocations
```

```
      fmt.Fprint(io.Discard, i, i, i)
   })
   println(m) // 1
}
```

Calling interface methods needs a little extra cost

Calling an interface method needs to look up a virtual table to find the called concrete method. And, calling a concrete method through an interface value prevents the method from being inlined.

```
package interfaces

import "testing"

type BinaryOp interface {
   Do(x, y float64) float64
}

type Add struct {}

func (a Add) Do(x, y float64) float64 {
   return x+y
}

//go:noinline
func (a Add) Do_NotInlined(x, y float64) float64 {
   return x+y
}

var x1, y1, r1 float64
var add Add
```

```
func Benchmark_Add_Inline(b *testing.B) {
   for i := 0; i < b.N; i++ {
      r1 = add.Do(x1, y1)
   }
}

var x2, y2, r2 float64
var add_NotInlined Add
func Benchmark_Add_NotInlined(b *testing.B) {
   for i := 0; i < b.N; i++ {
      r2 = add.Do_NotInlined(x2, y2)
   }
}

var x3, y3, r3 float64
var add_Interface BinaryOp = Add{}
func Benchmark_Add_Interface(b *testing.B) {
   for i := 0; i < b.N; i++ {
      r3 = add_Interface.Do(x3, y3)
   }
}
```

The benchmark results:

```
Benchmark_Add_Inline-4      0.6254 ns/op
Benchmark_Add_NotInlined-4  2.340 ns/op
Benchmark_Add_Interface-4   4.935 ns/op
```

From the benchmark results, we could get the cost of virtual table looking-up is about 2.5ns, and (for this specified case) the performance loss caused by not inlined is about 1.7 ns. The losses are small for most cases, but might be non-ignorable if the code runs frequently on the hot execution paths.

This has been mentioned in the stacks and escape analysis chapter.

However, no need to be too resistant to interface methods. For most cases, a clean design is more important than a bit better performance. And the official standard Go compiler is able to de-virtualize some interface method calls at compile time.

Avoid using interface parameters and results in small functions which are called frequently

Because most value boxing operations need memory allocations, we should try to avoid declaring small functions (including methods) with interface parameters and results if the functions are called massively and frequently in code execution.

For example, in the standard `image` package, there are many `At(x, y int) color.Color` and `Set(x, y int, c color.Color)` methods, which are declared to implement the `image/draw.Image` interface. The type `color.Color` is an interface type:

```
type Color interface {
    RGBA() (r, g, b, a uint32)
}
```

Calling these `At` and `Set` methods causes short-lived memory allocations (for boxing non-interface values) and calling the `Color.RGBA` interface method consumes a bit extra CPU resources (for looking up a virtual table and not inline-able). These methods are very likely called massively in image processing applications, which leads to a very bad code execution

efficiency. To alleviate the problem, Go 1.17 introduced a new interface type, `image/draw.RGBA64Image`:

```
type RGBA64Image interface {
   image.Image
   RGBA64At(x, y int) color.RGBA64
   SetRGBA64(x, y int, c color.RGBA64)
}
```

The new added method `RGBA64At` returns a non-interface type `color.RGBA64`, and the new added method `SetRGBA64` accept a `color.RGBA64` argument.

```
type RGBA64 struct {
   R, G, B, A uint16
}
```

By using the `RGBA64At` and `SetRGBA64` methods, many memory allocations and CPU resources are saved, so that code execution efficiency is improved much.