Memory allocations and alignment. Testing in Go - Part 1.

Session 11

Golang course by Exadel

07 Nov 2022

Sergio Kovtunenko
Lead backend developer, Exadel

Agenda

▶️ Revisit question from the past session

▶️ Memory allocations

▶️ Memory layout

▶️ Testing in Go - Part 1

▶️ Next time...

Revisit question from the past session

Interface pollution problem in Go

▶️ Explanation from the Official Go Wiki: "interfaces" (https://github.com/golang/go/wiki/CodeReviewComments#interfaces) section.

▶️ Explanation for the "Avoid Interface Pollution" (https://www.ardanlabs.com/blog/2016/10/avoid-interface-pollution.html) problem by William Kennedy.

- Code example: `code/interfacepollution/01_interface_pollution_test.go`

▶️ Comparison with C++ or Java from Jaana Dogan: "Interface pollution in Go" (https://rakyll.org/interface-pollution/).

▶️ "How To Use Go Interfaces" by Chewxy (https://blog.chewxy.com/2018/03/18/golang-interfaces/)

▶️ "Golang and interfaces misuse" by Hector Yeomans (https://hyeomans.com/golang-and-interfaces-misuse/)

Memory allocations

Memory allocations

- ✓ To allocate memory Go has two primitives, `new()` and `make()`. They do different things and apply to different types, which can be confusing, but the rules are simple.
  - `new(T)` returns *T pointing to a zeroed T
  - `make(T)` returns an initialized T

Allocations with function new()

- **new(T)** allocates *zeroed storage* for a new item of type **T** and returns its address, a value of type ***T**

- Or in other words, it returns a pointer to a *newly allocated zero value* of type **T**.

- Example:

```go
type S struct { a int; b float64 }
// Allocates storage for a variable of type S, initializes it (a=0, b=0.0), and returns a value of type *S containing the address of the location.
something := new(S)
```

- It is better to use *pointer-to-struct-literals* in most cases, except special ones:

```go
// NewPointerToString will return pointer to blank string.
func NewPointerToString() (*string) {
    //return &"" // Won't compile, because unable to take address from constant!

    //var s = ""
    //return &s // OK, but 2 lines instead of one

    return new(string) // OK, one liner to initialize pointer
}
```

Allocations with function make()

✓ The built-in function make() creates slices, maps, and channels *only*, and it returns an initialized (not zero!) value of type T, and **not a pointer: *T**.

- The memory is initialized as described in the section on initial values.

Call	Type T	Result
make(T, n)	slice	*slice of type T with length n and capacity n*
make(T, n, m)	slice	*slice of type T with length n and capacity m*
make(T)	map	*map of type T*
make(T, n)	map	*map of type T with initial space for approximately n elements*
make(T)	channel	*unbuffered channel of type T*
make(T, n)	channel	*buffered channel of type T, buffer size n*

✓ Each of the size arguments **n** and **m** must be of integer type or an untyped constant.

- A constant size argument must be non-negative and representable by a value of type **int** (if it is an untyped constant it is given type **int**).
- If both **n** and **m** are provided and *are constant*, then **n** must be **<= m**.
- If **n** *is negative* or *larger than* ***m*** at run time, a run-time panic occurs.

✓ Calling **make** with a **map** type and **size** hint **n** will create a **map** with initial space to hold n map elements.

- The precise behavior is implementation-dependent.

How do I know whether a variable is allocated on the heap or the stack? (1/2)

- ✓ From a correctness standpoint, *you don't need to know*.
- ✓ Each variable in Go exists as long as there are references to it.
  - The storage location chosen by the implementation is irrelevant to the semantics of the language.

How do I know whether a variable is allocated on the heap or the stack? (2/2)

How do I know whether a variable is allocated on the heap or the stack?

From a correctness standpoint, you don't need to know. Each variable in Go exists as long as there are references to it. The storage location chosen by the implementation is irrelevant to the semantics of the language.

The storage location does have an effect on writing efficient programs. When possible, the Go compilers will allocate variables that are local to a function in that function's stack frame. However, if the compiler cannot prove that the variable is not referenced after the function returns, then the compiler must allocate the variable on the garbage-collected heap to avoid dangling pointer errors. Also, if a local variable is very large, it might make more sense to store it on the heap rather than the stack.

In the current compilers, if a variable has its address taken, that variable is a candidate for allocation on the heap. However, a basic *escape analysis* recognizes some cases when such variables will not live past the return from the function and can reside on the stack.

Reference: "Go FAQ" (https://golang.org/doc/faq#stack_or_heap)

Memory layout

Problem statement

- ▶️ Take a look at example: `code/memorylayout/01_stacktrace.go`
- ▶️ Very important: "Reading stack traces in Go" by Michele Caci (https://dev.to/mcaci/reading-stack-traces-in-go-3ah5)

Memory layout: array

▶️ **Visualization:** Array internals (https://go.dev/blog/slices-intro)

▶️ **Even more Visualization:** Go Data Structures (https://research.swtch.com/godata)

Memory layout: slice

▶ **Visualization:** Slice Internals (https://go.dev/blog/slices-intro)

▶ **Runtime definition:** `<goSource>.../src/runtime/slice.go:15`

```
type slice struct {
    array unsafe.Pointer
    len   int
    cap   int
}
```

▶ **Required memory for variable of slice type:** 3 words

Memory layout: strings

▶️ **Visualization:** "String type" by Brad Fitzpatrick (https://docs.google.com/presentation/d/1lL7Wlh9GBtTSieqHGJ5AUd1XVYR48UPhEloVem-79mA/preview?slide=id.gc5ec805d9_0_140)

▶️ **Runtime definition:** `<goSource>.../src/runtime/string.go:238`

```
type stringStruct struct {
    str unsafe.Pointer
    len int
}
```

▶️ **Required memory for variable of string type:** 2 words

Memory layout: interface

▶️ **Visualization:** "Interface type" by Brad Fitzpatrick

(https://docs.google.com/presentation/d/1lL7Wlh9GBtTSieqHGJ5AUd1XVYR48UPhEloVem-79mA/view#slide=id.gc5ec805d9_0_464)

▶️ **Runtime definition:** `<goSource>.../src/runtime/runtime2.go:202`

```
type iface struct {
    tab  *itab
    data unsafe.Pointer
}
```

▶️ **Required memory for variable of interface type:** 2 words

Memory layout: map

Visualization: ["Map type" by Brad Fitzpatrick](https://docs.google.com/presentation/d/1lL7Wlh9GBtTSieqHGJ5AUd1XVYR48UPhEloVem-79mA/view#slide=id.gc5ec805d9_0_590) (https://docs.google.com/presentation/d/1lL7Wlh9GBtTSieqHGJ5AUd1XVYR48UPhEloVem-79mA/view#slide=id.gc5ec805d9_0_590)

Runtime definition: `<goSource>.../src/runtime/map.go:116`

- Semantic: uses pointer to hmap internally for all private functions!

```
type hmap struct {
    // Note: the format of the hmap is also encoded in cmd/compile/internal/reflectdata/reflect.go.
    // Make sure this stays in sync with the compiler's definition.
    count     int // # live cells == size of map.  Must be first (used by len() builtin)
    flags     uint8
    B         uint8  // log_2 of # of buckets (can hold up to loadFactor * 2^B items)
    noverflow uint16 // approximate number of overflow buckets; see incrnoverflow for details
    hash0     uint32 // hash seed

    buckets    unsafe.Pointer // array of 2^B Buckets. may be nil if count==0.
    oldbuckets unsafe.Pointer // previous bucket array of half the size, non-nil only when growing
    nevacuate  uintptr        // progress counter for evacuation (buckets less than this have been evacuated)

    extra *mapextra // optional fields
}
```

Required memory for variable of map type: 1 word

Memory layout: function

▶️ **Visualization:** "Function type" by Brad Fitzpatrick

(https://docs.google.com/presentation/d/1lL7Wlh9GBtTSieqHGJ5AUd1XVYR48UPhEloVem-79mA/view#slide=id.gc5ec805d9_0_545)

▶️ **Runtime definition:** `<goSource>.../src/runtime/runtime2.go:197`

- Semantic: uses pointer to `funcval` internally for all private functions!

```
type funcval struct {
    fn uintptr
    // variable-size, fn-specific data here
}
```

▶️ **Required memory for variable of function type:** 1 word

Memory layout: channel

▶️ Visualization: "Channel type" by Brad Fitzpatrick (https://docs.google.com/presentation/d/1lL7Wlh9GBtTSieqHGJ5AUd1XVYR48UPhEloVem-79mA/view#slide=id.gc5ec805d9_0_590)

▶️ Runtime definition: <goSource>`.../src/runtime/chan.go:33`

- Semantic: uses pointer to hchan internally for all private functions!

```
type hchan struct {
    qcount   uint           // total data in the queue
    dataqsiz uint           // size of the circular queue
    buf      unsafe.Pointer // points to an array of dataqsiz elements
    elemsize uint16
    closed   uint32
    elemtype *_type // element type
    sendx    uint   // send index
    recvx    uint   // receive index
    recvq    waitq  // list of recv waiters
    sendq    waitq  // list of send waiters

    lock mutex
}
```

▶️ **Required memory for variable of channel type:** 1 word

Testing in Go - Part 1

Testing in Go

▶️ All information is located in the testing package documentation (https://pkg.go.dev/testing).

▶️ Typical test anatomy:

- Setup (optional)
- Execute the code under testing
- Check (assert) the results
- Tear-down (optional)

Tests

```
func TestXxx(*testing.T)
```

where Xxx does not start with a lowercase letter. The function name serves to identify the test routine.

Within these functions, use the Error, Fail or related methods to signal failure.

To write a new test suite, create a file whose name ends _test.go that contains the TestXxx functions as described here. Put the file in the same package as the one being tested. The file will be excluded from regular package builds but will be included when the "go test" command is run. For more detail, run "go help test" and "go help testflag".

A simple test function looks like this:

```
func TestAbs(t *testing.T) {
    got := Abs(-1)
    if got != 1 {
        t.Errorf("Abs(-1) = %d; want 1", got)
    }
}
```

Homework

- ▶️ Read: "Memory Layouts" by Tapir Liu (https://go101.org/article/memory-layout.html)
- ▶️ Read: "Go internals: invariance and memory layout of slices" by Eli Bendersky (https://eli.thegreenplace.net/2021/go-internals-invariance-and-memory-layout-of-slices/)
- ▶️ Read (again): "Go Data Structures" by Russ Cox (https://research.swtch.com/godata)
- ▶️ Watch: "Allocator Wrestling" by Eben Freeman (https://speakerdeck.com/emfree/allocator-wrestling)
- ▶️ Be excited: "Visualizing memory management in Golang" (https://deepu.tech/memory-management-in-golang/)
- ▶️ Very important: "Reading stack traces in Go" by Michele Caci (https://dev.to/mcaci/reading-stack-traces-in-go-3ah5)

Next time...

▶️ Session12:

Testing - part 2. Packages in Go

- Package clause
- Import declarations
- Package paths
- Package names
- Internal directories
- Vendor directories
- Circular dependencies
- Best practices to design packages
- Program initialization flow

Thank you

Golang course by Exadel

07 Nov 2022

Sergio Kovtunenko
Lead backend developer, Exadel
skovtunenko@exadel.com (mailto:skovtunenko@exadel.com)
https://github.com/skovtunenko (https://github.com/skovtunenko)
@realSKovtunenko (http://twitter.com/realSKovtunenko)