5 Mistakes C/C++ Devs make writing Go

A newbie's journey into Go

Aug 29 2018

Nyah Check
Software Engineer, Altitude Networks

Why am I here?

- Wrote C/C++ for close to 5 years before Go.
- Brought bad C style code in Go and had a lot of issues

What you'll learn...

- Learn from my mistakes
- Avoid some common pitfalls newbies face writing Go

Agenda

I classified the my mistakes under 3 topics:

- Heap Vs Stack
- Memory & Goroutine leaks
- Error handling

One more thing ...

This is a discussion

If you don't understand something, or think what I'm saying is incorrect, please ask at the end.

I'll leave some minutes at the end of the presentation for some Q/A

Heap Vs Stack

What is a Heap and Stack in Go?

A **Stack** is a special region in created to store **temporary** variables **bound** to a function.
It's self cleaning and expands and shrinks accordingly.

A **Heap** is a bigger region in memory in addition to the stack used for storing values,
It's more costly to maintain since the GC needs to clean the region from time to time adding extra latency.

Mistake 1: New doesn't mean heap && var doesn't mean stack

An early mistake was to minimize **escape analysis** and it's possible implications on my program's perf.

Consider the following *C++* code

```
int foo() {
    int *a = new(int);
    return *a;
}
```

Wrong assumptions..

- In C++, we know **new(int)** is allocated on the heap.
- In Go, we don't really know for sure.
- May be the **new** keyword was stolen from C++ as a result might likely be allocated on the heap?
- Given my C++ bias, I thought minimizing it's use will reduce *heap* allocation.

Let's look at some code...

```
package main

import "fmt"

func newIntStack() *int {

    vv := new(int)

    return vv
}

func main() { fmt.Println(*newIntStack()) }
```

Run

Question

Where do we think the **vv** variable will be allocated?

Stack or **Heap**?

Let's look at the compiler escape decisions output

```
➜  examples git:(master) ✗ go run -gcflags -m stack.go
# command-line-arguments
./stack.go:7:6: can inline newIntStack
./stack.go:14:39: inlining call to newIntStack
./stack.go:9:11: new(int) escapes to hea
./stack.go:14:27: *(*int)(~r0) escapes to heap
./stack.go:14:39: main new(int) does not escape <-- STACK ALLOCATED
./stack.go:14:26: main ... argument does not escape
0
➜  examples git:(master)
```

Let's take a look at another example

```
package main

import "fmt"

func main() {
    x := "GOPHERCON-2018"
    fmt.Println(x)
}
```

Run

Where will x be allocated?

Heap or **Stack**?

Let's find out...

```
➜  examples git:(master) ✗ go run -gcflags -m heap.go
# command-line-arguments
./heap.go:9:13: x escapes to heap <---- Something strange happens
./heap.go:9:13: main ... argument does not escape
GOPHERCON-2018
```

It's surprising to see **x** which not called outside may is allocated on the heap instead.

Why?

I'll pass the -m option multiple times to make the output more verbose:

```
# command-line-arguments
./heap.go:7:6: cannot inline main: non-leaf function
./heap.go:9:13: x escapes to heap
./heap.go:9:13:         from ... argument (arg to ...) at ./heap.go:9:13
./heap.go:9:13:         from *(... argument) (indirection) at ./heap.go:9:13
./heap.go:9:13:         from ... argument (passed to call[argument content escapes]) at ./heap.go:9:13
./heap.go:9:13: main ... argument does not escape
GOPHERCON-2018
➜  examples git:(master) ✗
```

What happened?

So looking at **Line 13**

- x is passed to `fmt.Println` which receives an **interface** argument
- so x is converted to an interface whose value 'x' is alloc on the heap.
- So x `escapes`

This is very confusing/counterintuitive to a C/C++ developer, yet this is how Go works.

Lessons

- Escape analysis is very important in writing more performant Go programs, yet there's no language specification on this.
- Some of the compiler's escape analysis decisions are counterintuitive, yet trial and error is the only way to know
- Do not make assumptions, rather do **escape analysis** on the code and make informed decisions.

Escape Analysis guidelines

- Functions calling other functions
- references assigned to struct members
- slices and maps
- pointers to variables

Conclusion

"Understand heap vs stack allocation in your Go program by checking the compiler's escape analysis report and making informed decisions, do not guess"

Memory Leaks

How does memory leak in Go

- I assumed since there's a garbage collector, then everything is fine

Not True!

- Memory leaks are common in any language including garbage collected languages
- It can be caused by: assigned but unused memory, dangling pointers, synchronization issues.
- Some of these errors can be hard to detect, but Go has a set of tools which could be very effective in debugging these bugs

Mistake 2: Do not defer in an infinite Loop

The **defer** statement is used to clean up resources after you open up a resource(e.g. file, connection etc)

So an idiomatic way will be:

```
fp, err := os.Open("path/to/file.text")
if err != nil {
    //handle error gracefully
}
defer fp.Close()
```

This snippet is guaranteed to work even if cases where there's a panic and it's **standard** Go practice.

So what's the problem?

In very large files where resources cannot be tracked and freed properly, this becomes a problem.

Consider a file monitoring program in **C** where:
- We check a specific directory for db file dumps
- perform some operation(logging, file versioning, etc)

Something like this might work

```
#define TIME_TO_WAIT 1 /* wait for one second */
int main() {
    FILE *fp;
    clock_t last = clock();
    char* directory[2] = {"one.txt", "two.txt"};
    for ( ; ; ) {
        clock_t current = clock();
        if (current >= (last + TIME_TO_WAIT + CLOCKS_PER_SEC)) {
            for (int i = 0; i < 2; i++) {
                fp = fopen(directory[i], "r+");
                printf("\nopening %s", directory[i]);
                if (fp == NULL) {
                        fprintf(stderr, "Invalid file %s", directory[i]);
                    exit(EXIT_FAILURE);
                }
                //some FILE processing happens
                fclose(fp);
                printf("\nclosing %s", directory[i]);
                last = current;
            }
        }//executes every second
    }
```

Run

This will be sure to open and close up the files once the operations are done.

However in Go

```
func loggingMonitorErr(files ...string) {
    for range time.Tick(time.Second) {
        for _, f := range files {
            //files coming in through the channel.
            fp := OpenFile(f)
            // The line below will never execute.
            defer fp.Close()
            //process file
        }
    }
}
```

Run

Problems:

- Deferred code never executes since the function has not returned
- So memory clean up never happens and it's use keeps piling up
- Files will never be closed, therefore causing loss of data due to lack of flush.

How do I fix this?

- Spin up a goroutine for each file monitoring
- This ensures everything is bound to the context
- Hence files are opened and closed

Solution

```
type file string

func OpenFile(s string) file {
    log.Printf("opening %s", s)
    return file(s)
}
func (f file) Close() { log.Printf("closing %s", f) }

func loggingMonitorFix(files ...string) {
    for range time.Tick(time.Second) {
        for _, f := range files {
            //files coming in through the channel.
            func() {
                fp := OpenFile(f)
                defer fp.Close()
                //process file
            }()
        }
    }
}
```

Run

Lessons learned

- Since defer is tied to the new function context, we are sure it's executed and memory is flushed when files close
- When defer executes we are certain our goroutine finished execution, so no memory leaks

Conclusion

"Do not defer in an infinite loop, since the defer statement invokes the function execution **ONLY** when the surrounding function returns"

Pointers to accessible parts of a slice

What's a slice?

A **slice** is a dynamically sized flexible view into an array.

We know arrays have fixed fizes.

There are **two** main **features** of slices to think about:

- The **length** of a slice is simply the total number of elements contained in the slice
- The **capacity** of a slice is the number of elements in the underlying array.

Their understanding can avoid some robustness issues.

How?

Mistake 3: Keeping pointers to an accessible(although not visible) part of a slice

Prior to Go 1.2 there was a memory safety issue with slices

- access to elements of the underlying array.
- This could lead to unintended memory writes.
- Cause **robustness** issues where these regions of memory are not garbage collected.

Let's use an example.

```
func main() {
    a := []*int{new(int), new(int)}
    fmt.Println(a)
    b := a[:1]
    fmt.Println(b)

    // second element is not garbage collected, because it's *still* accessible
    c := b[:2]
    fmt.Println(c)
}
```

Run

Our output will be:

```
➜  examples git:(master) ✗ go run main.go
[0xc420016090 0xc420016098]
[0xc420016090]
[0xc420016090 0xc420016098]
```

What are some of the problems?

- Write regions of memory unintentionally.
- Robustness issues: Memory is not garbage collected since there's a **reference** to it.
- It's a source for potential bugs

How do you solve this then?

Go 1.2++ added the **3-Index-Slice** operation

- This enables you to specify the capacity during slicing.
- The restricted slice capacity provides a level of protection to the underlying array
- No unintended memory writes.
- Unused areas of the underlying array are garbage collected.

How do we use it then

Rewriting our code gives

```
func main() {
    a := []*int{new(int), new(int)}
    fmt.Println(a)

    // Using the 3- index slice operation
    b := a[:1:1]
    fmt.Println(b)
    c := b[:2]
    fmt.Println(c)
}
```

Run

Our output becomes ...

```
➜  examples git:(master) ✗ go run main.go
[0xc420016090 0xc420016098]
[0xc420016090]
panic: runtime error: slice bounds out of range

goroutine 1 [running]:
main.main()
    /Users/nyahcheck/go/src/github.com/Ch3ck/5-mistakes-c-cpp-devs-make-writing-go/03-pointer-in-non-vis
exit status 2
```

Our slice cap was set to 1, we can't access regions of memory we don't have permissions to, rightly creating a panic.

Lesson

- Our slice capacity was set to 1, so can't access restricted regions in memory, rightly creating a panic
- More robust programs since unused memory is garbae collected.
- Reduce sources for potential bugs in your code.

Goroutine leaks

What's a Goroutine

It's a **lightweight** thread of execution, it consists of functions that run **concurrently** with other functions/methods.

What about channels?

A **channel** is a pipe that connects concurrent goroutines.

An understanding of these two concepts embodies concurrency in Go.

How do they leak?

There are different possible causes for goroutine leaks, some include:

- Infinite loops
- Blocks on synchronization points(channels, mutexes), deadlocks

However when these occur the program takes up more memory than it actually needs leading to **high latency** and frequent crashes.

Let's take a look at an example

Mistake 4: Error handling with channels where # channels < # goroutines

- C/C++ has libraries for multi-threaded programming.
- Concurrency in Go materializes itself in the form of goroutines and channels.

Problem:

- We'll look at the issue
- Fix it and discuss go tools available to handle these kinds of issues

main.go

```
func doSomethingTwice() error {
    // Issue occurs below
    errc := make(chan error)

    go func() {
        defer fmt.Println("done wth a")
        errc <- doSomething("a")
    }()
    go func() {
        defer fmt.Println("done with b")
        errc <- doSomething("b")
    }()
    err := <-errc
    return err
}
```

Run

What does our code do?

This program spins up two goroutines and runs some internal processes and writes the results to a channel

Our output here is:

```
➜  examples git:(master) ✗ go run main.go
2018/08/28 14:09:54 profile: trace enabled, /var/folders/s7/s4fj1d3j07b5wqy3jw_lpwj00000gn/T/profile3074
something went wrong with b
done wth a
done wth a
something went wrong with a
done with b
done with b
something went wrong with b
done wth a
done wth a
something went wrong with a
done with b
done with b
```

Let's look at the memory build up

```
Processes: 481 total, 2 running, 479 sleeping, 3067 threads
Load Avg: 2.10, 2.16, 2.20  CPU usage: 2.95% user, 2.36% sys, 94.67% idle  SharedLibs: 157M resident, 40
MemRegions: 179430 total, 5220M resident, 119M private, 2881M shared. PhysMem: 16G used (3359M wired), 1
VM: 7554G vsize, 1111M framework vsize, 3678046(0) swapins, 4327421(0) swapouts. Networks: packets: 1468

PID    COMMAND      %CPU      TIME     #TH    #WQ   #PORT MEM    PURG   CMPRS  PGRP  PPID  STATE    BOOS
39491  main         0.6       00:00.30 10     0     32    1404K+ 0B     0B     39477 39477 sleeping *0[1
39477  go           0.0       00:00.22 16     0     57    9212K  0B     0B     39477 8036  sleeping *0[1
39465  quicklookd   0.0       00:00.14 4      1     85    5060K  32K    0B     39465 1     sleeping  0[0
```

What are the problems with the code

- More goroutines than channels are present to write to send data back to main
- When one routine writes to the channel, the program exits and the other goroutine is lost, building up memory use as a results
- that region of memory is not garbage collected

How do we fix this?

We simply increase the number of channels to 2,
This makes it possible for the two goroutines to pass their results to the calling program.

```
func doSomethingTwice() error {
    // Issue occurs below
    errc := make(chan error, 2)

    go func() {
        defer fmt.Println("done wth a")
        errc <- doSomething("a")
    }()
    go func() {
        defer fmt.Println("done with b")
        errc <- doSomething("b")
    }()
    err := <-errc
    return err
}
```

Run

Our output here becomes

```
➜  examples git:(master) ✗ go run fixed.go
2018/08/29 00:40:45 profile: trace enabled, /var/folders/s7/s4fj1d3j07b5wqy3jw_lpwj00000gn/T/profile8553
------
done wth a
something went wrong with a
done with b
------
done wth a
something went wrong with a
done with b
------
done with b
something went wrong with b
done wth a
------
done wth a
something went wrong with a
done with b
------
```

Our memory build up is:

```
Processes: 480 total, 2 running, 478 sleeping, 3066 threads
Load Avg: 1.79, 2.17, 2.19  CPU usage: 2.70% user, 3.41% sys, 93.87% idle  SharedLibs: 159M resident, 40
MemRegions: 211071 total, 5727M resident, 103M private, 2878M shared. PhysMem: 16G used (3397M wired), 4
VM: 7551G vsize, 1112M framework vsize, 3678685(0) swapins, 4327421(0) swapouts. Networks: packets: 1468

PID    COMMAND      %CPU TIME     #TH   #WQ  #PORT MEM    PURG   CMPRS  PGRP  PPID  STATE    BOOSTS
39844  top          4.8  00:02.74 1/1   0    23    5880K  0B     0B     39844 15534 running  *0[1]
39835  main         0.6  00:00.30 11    0    35    1396K+ 0B     0B     39821 39821 sleeping *0[1]
39821  go           0.0  00:00.18 15    0    54    9272K  0B     0B     39821 8036  sleeping *0[1]
39813  quicklookd   0.0  00:00.08 4     1    85    5020K  32K    0B     39813 1     sleeping  0[0]
```

Performing Traces on the code

Lessons

Goroutine leaks are very common in Go development.

However there are some best practices you can follow to avoid some of these errors:

- Using the context package to terminate or timeout goroutines which may otherwise run indefinitely
- Using a done signal or timeout channel can help in terminating a running goroutine preventing leaks

Best practices (cont.)

- Profiling the code, Stack trace instrumentation and adding benchmarks can go a long way in finding these leaks
- Take advantage of the go tooling ecosystem: go tool trace, go tool profile , go-torch, gops, leaktest etc

Error handling

What are errors in Go?

Go has a built-in **error** type which uses error values to indicate an abnormal state.

Also these error type is based on an error *interface.

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

The error variable is a string variable.

A closer look at the **errors package** will provide some good insides into handling errors in Go.

Mistake 5: Errors are not just strings, but much more

Consider a **C** program with a division by zero error

```
#include <stdio.h> /* for fprintf and stderr */
#include <stdlib.h> /* for exit */
int main( void )
{
   int dividend = 50;
   int divisor = 0;
   int quotient;

   if (divisor == 0) {
       /* Example error handling.
        * Writing a message to stderr, and
        * exiting with failure.
        */
       fprintf(stderr, "Division by zero! Aborting...\n");
       exit(EXIT_FAILURE); /* indicate failure.*/
   }

   quotient = dividend / divisor;
   exit(EXIT_SUCCESS); /* indicate success.*/
}
```

Run

Handling errors in C

- Typically consists of writing error message to stderr and returning an exit code.

However, in Go errors are much more complicated than strings.

Consider this example:

```
func main() {
    conn, err := net.Dial("tcp", "gooooooooooogle.com:80")
    if err != nil {
        fmt.Println("%T", err)
        switch err := err.(type) {
        case *net.OpError:
            fmt.Println(err)
            fmt.Printf("failed to connect to %s because %v\n", err.Net, err.Err)
            return
        default:
            log.Fatal(err)
        }
    }
    defer conn.Close()
}
```

Run

Wrapping Errors in Go(Errors package)

Consider another example

```
func connect(addr string) error {
    conn, err := net.Dial("tcp", addr)
    if err != nil {
        switch err := err.(type) {
        case *net.OpError:
            // return fmt.Errorf("failed to connect to %s: %v", err.Net, err)
            return errors.Wrapf(err, "failed to connect to %s", err.Net)
        default:
            // return fmt.Errorf("unkown error: %v", err)
            return errors.Wrap(err, "unknown error")
        }
    }

    defer conn.Close()

    return nil
}
```

Run

Advantages of Wrap and Cause funcs

- You can preserve the error context and pass to the calling program
- Using the `errors.Cause()` function call we can determine what caused this error later in the program

I believe it's a feature some developers my overlook but if used properly will give a better Go development experience.

Lesson learned:

- The errors package provides a lot of powerful tools for handling errors which some devs may ignore.
- Wrap() and errors.Cause() are very useful in preserving context of an error later in the program.

Take a look at the errors package and see elegant examples.

Conclusion

- **Understand** Escape analysis by looking at the compiler decisions, do not make **reasonable guesses**.
- **Defer** executes only when the function returns. Using it in a infinite loop is a **mistake**.
- Three Index_slices adds an extra **robustiness** utility in Go, use it.

Conclusion (cont.)

- **Profile** your Go code to identify bottlenecks early on, it's a good practice.
- Errors in Go are not just strings, but much more.
- Wrap errors to preserve context and handle them gracefully.

There are many more errors C/C++ devs make

Just remember ...

- Bringing concepts from C/C++ is fine but be ready to be challenged by differences.
- "Programming in Go is like being young again (but more productive!)."

Discussion

Any questions?

Thank you

Aug 29 2018

Nyah Check
Software Engineer, Altitude Networks
nyah@altitudenetworks.com (mailto:nyah@altitudenetworks.com)
https://github.com/Ch3ck (https://github.com/Ch3ck)
@nyah_check (http://twitter.com/nyah_check)