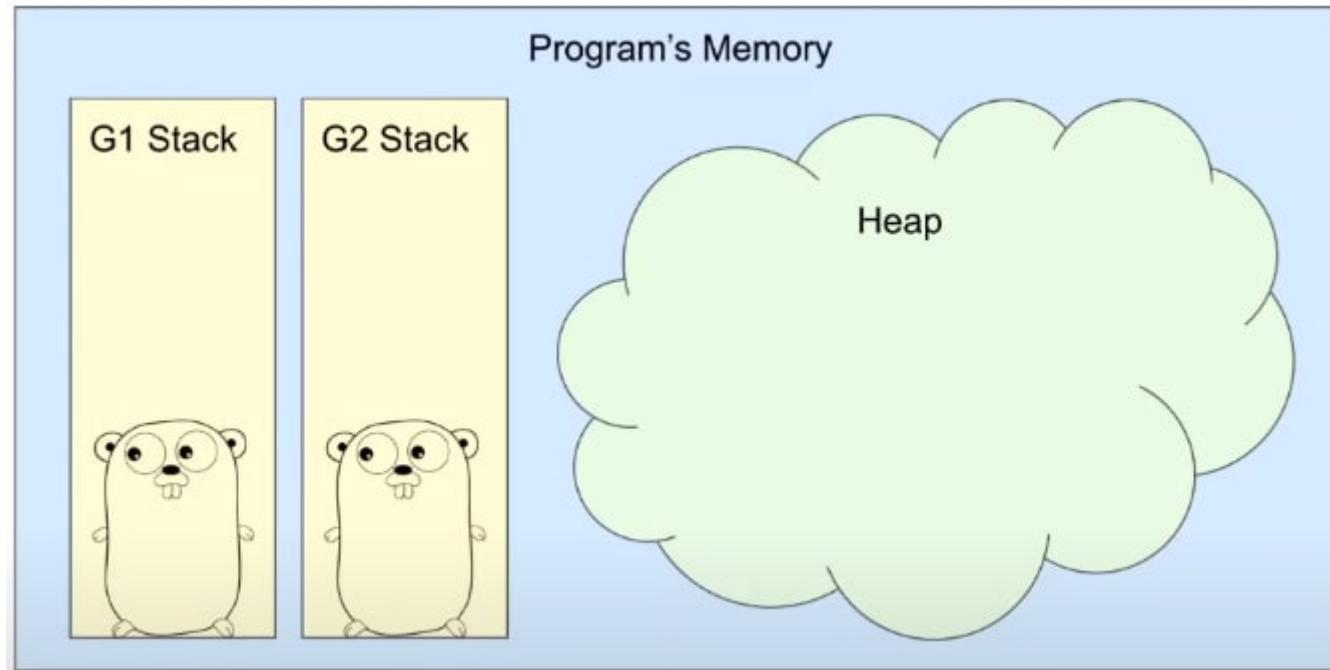


Escape Analysis in Go:

Understanding and Optimizing Memory Allocations

Harsh Gupta
Software Engineer, Money Forward Inc.

A program's memory



Note:

- In Go, we have multiple goroutines and each has its own **stack**.

Go では、複数のゴルーチンがあり、それぞれに独自の **スタック** があります。

Why allocations on the stack are preferred generally? #1

- Allocations are **faster** on the stack (although the benefit isn't that great).

スタック上の割り当てが**高速**になります(ただし、それほど大きなメリットはありません)。

Why allocations on the stack are preferred generally? #2

- Stack-allocated variables are **automatically deallocated** when the function returns or when the block where they are defined exits.

スタックに割り当てられた変数は、関数が返されるとき、または変数が定義されているブロックが終了するときに、**自動的に割り当てが解除されます。**

- On the other hand, variables allocated on the heap require **manual deallocation** or by the garbage collector when they are no longer needed.

一方、ヒープ上に割り当てられた変数は、不要になったときに**手動で割り当て解除**するか、ガベージコレクターによって割り当てを解除する必要があります。

- And GC causes latency (for the whole program).

そして、GCは(プログラム全体で)待ち時間を引き起こします。

Why allocations on the stack are preferred generally? #3

- More **efficient memory utilization** because stack allocations do not contribute to memory fragmentation.

スタック割り当てがメモリの断片化に寄与しないため、**より効率的なメモリ利用**が可能になります。

Let's look at some examples

Example 1

Stack without pointers

ポインターを使用せずにスタックする

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     a := 10
7     a1 := nextInt(a)
8
9     fmt.Println(a1)
10 }
11
12 func nextInt(i int) int {
13     return i + 1
14 }
```

Run

Example 2

Stack with pointers

スタック上のポインター

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     a := 10
7     a1 := nextIntWithPtr(&a)
8
9     fmt.Println(a1)
10 }
11
12 func nextIntWithPtr(i *int) int {
13     return *i + 1
14 }
```

Run

Example 3

Returning pointers

ポインタを返す

```
1 package main
2
3 import "fmt"
4
5 func getOne() *int {
6     i := 1
7     return &i
8 }
9
10 func main() {
11     a := getOne()
12
13     fmt.Println(*a)
14 }
```

Run

But how does that happen?

Example 4

Returning pointers in C

Cでのポインタの返し

```
#include <stdio.h>

int * getOne() {
    int i = 1;
    return &i;
}

int main() {
    int *a = getOne();    // 1

    printf("%d\n", *a);
    return 0;
}
```

What is happening here?

- The Go code works fine in *Example 3* because of an optimization made by the compiler during its compilation process called **Escape Analysis**.

Go コードは、**エスケープ分析** と呼ばれるコンパイル プロセス中にコンパイラーによって行われた最適化により、例3では正常に動作します。

Note:

The same code in C (*Example 4*) will throw random output because there is no concept of escape analysis built in and it allocates on the stack.

C の同じコード (例4) は、エスケープ解析の概念が組み込まれておらず、スタック上に割り当てられるため、ランダムな出力をスローします。

What is it?

- Escape analysis in Go is a compiler optimization technique that determines whether a variable's lifetime extends beyond its allocation site.

Go のエスケープ分析は、変数の有効期間が割り当てサイトを超えて延長されるかどうかを判断するコンパイラ最適化手法です。

- It helps the compiler decide whether to allocate variables on the stack or the heap, based on their usage within the program.

これは、プログラム内での変数の使用状況に基づいて、コンパイラが変数をスタックに割り当てるかヒープに割り当てるかを決定するのに役立ちます。

- The primary goal of escape analysis is to minimize heap allocations which results in improved performance and reduced memory usage. **Happy GC! :)**

エスケープ分析の主な目的は、ヒープ割り当てを最小限に抑えることで、パフォーマンスが向上し、メモリ使用量が削減されることです。 **ハッピーGC! :)**

How does it work? #1

- The algorithm ensures two invariants:

このアルゴリズムでは、次の 2 つの不变条件が保証されます:

- 1. pointers to stack objects cannot be stored in the heap

スタック オブジェクトへのポインタはヒープに格納できません

- 2. pointers to a stack object cannot outlive that object

スタック オブジェクトへのポインタはそのオブジェクトを超えて存続することはできません

How does it work? #2

- This is done by constructing a directed weighted graph where vertices represent variables and edges represent assignments (with weights representing addressing/dereference counts).

これは、頂点が変数を表し、エッジが割り当てを表す有向重み付きグラフを構築することによって行われます(重みはアドレス指定/逆参照カウントを表します)。

- Some examples:

いくつかの例 :

```
p = &q      // -1
p = q      //  0
p = *q     //  1
p = **q    //  2
p = **&**&q //  2
```

How does it work? #3

- It then iteratively walks through the graph looking for the assignments that might violate the invariants and based on that marks some variables as requiring heap allocation.

次に、グラフを繰り返し調べて不变条件に違反する可能性のある割り当てを探し、それに基づいて一部の変数をヒープ割り当てが必要であるとマークします。

How does it work? #4

- If the compiler cannot definitively prove that a variable will not escape, it assumes it will escape and allocates it on the heap.

コンパイラは、変数がエスケープしないことを明確に証明できない場合、変数はエスケープすると想定し、ヒープ上に変数を割り当てます。

- In addition to the algorithm, there are some additional rules as well.

アルゴリズムに加えて、追加のルールもいくつかあります。

Special cases for escape analysis #1

- Variables with **large size** escapes

サイズが大きい変数はエスケープされます

```
func main() {  
    s := make([]int64, 8*1024)    // 64KB  
  
    _ := s  
}
```

```
func main() {  
  
    // escapes to heap: too large for stack  
    s := make([]int64, 8*1024 + 1)    // just above 64KB  
  
    _ := s  
}
```

Special cases for escape analysis #2

- Slice variable with non-constant capacity size escapes

非定数容量サイズのスライス変数がエスケープされる

```
func main() {
    const c = 10

    s := make([]int, c)
    _ = s
}
```

```
func main() {
    var v = 10

    // escapes to heap: non-constant size
    s := make([]int, v)
    _ = s
}
```

Special cases for escape analysis #3

- Source variable captured by a closure function escapes

クロージャによってキャプチャされたソース変数関数がエスケープします

```
func main() {
    var x *int

    func(i *int) {
        j := 2
        i = &j
    }(x)
}
```

```
func main() {
    var x *int

    func() {
        j := 2 // j escapes to heap
        // escapes to heap: captured by a closure
        x = &j
    }()
    _ = x
}
```

One more example

io.Reader

1. Read() function signature for Go's built-in io.Buffer:

Go の組み込み io.Buffer の Read() 関数シグネチャ:

```
func (b *Buffer) Read(p []byte) (n int, err error)
```

- The byte slice should be given as an argument to the Read function.

バイト スライスは Read 関数の引数として指定する必要があります。

- The function returns an integer which is the number of bytes read or an error.

この関数は、読み取られたバイト数を示す整数、またはエラーを返します。

My custom reader

2. Read() method declaration in **MyBuffer** struct for the experiment:

実験用の **MyBuffer** 構造体の Read() メソッド宣言:

```
func (mb *MyBuffer) Read() ([]byte, error)
```

- The function does not take any arguments.

この関数は引数を取りません。

- The function returns with a byte slice containing the bytes read or an error.

この関数は、読み取られたバイトを含むバイトスライスまたはエラーを返します。

The struct

```
1 package main
2
3 import "bytes"
4
5 const SIZE = 8
6
7 type MyBuffer struct {
8     buffer *bytes.Buffer
9 }
10
11 func (mb *MyBuffer) Read() ([]byte, error) {
12
13     b := make([]byte, mb.buffer.Len(), SIZE)
14
15     copy(b, mb.buffer.Bytes())
16     return b, nil
17 }
```

The code

```
33 //go:noinline
34 func read1(buf *bytes.Buffer, b []byte) int {
35     read, err := buf.Read(b)
36     if err != nil {
37         return 0
38     }
39     return read
40 }
41
42 //go:noinline
43 func read2(buf *MyBuffer) []byte {
44     myRead, err := buf.Read()
45     if err != nil {
46         return nil
47     }
48     return myRead
49 }
```

Compiler reporting #1

Command:

```
go build -gcflags "-m -l" buffreader/*.go
```

Output:

```
buffreader/mybuffer.go:9:7: mb does not escape
buffreader/mybuffer.go:12:11: make([]byte, (*bytes.Buffer).Len(mb.buffer), SIZE) escapes to heap
buffreader/reader.go:34:12: buf does not escape
buffreader/reader.go:34:31: b does not escape
buffreader/reader.go:43:12: buf does not escape
buffreader/reader.go:15:5: func literal escapes to heap
buffreader/reader.go:16:14: ... argument does not escape
buffreader/reader.go:19:11: make([]byte, 10) does not escape
buffreader/reader.go:22:10: &MyBuffer{...} does not escape
```

Compiler reporting #2

Command:

```
go build -gcflags "-m=2 -l" buffreader/*.go
```

Output:

```
buffreader/mybuffer.go:12:11: make([]byte, (*bytes.Buffer).Len(mb.buffer), SIZE) escapes to heap:  
buffreader/mybuffer.go:12:11:    flow: b = &{storage for make([]byte,  
                                (*bytes.Buffer).Len(mb.buffer), SIZE)}:  
buffreader/mybuffer.go:12:11:        from make([]byte, (*bytes.Buffer).Len(mb.buffer), SIZE)  
                                (spill) at buffreader/mybuffer.go:12:11  
buffreader/mybuffer.go:12:11:        from b := make([]byte, (*bytes.Buffer).Len(mb.buffer), SIZE)  
                                (assign) at buffreader/mybuffer.go:12:4  
buffreader/mybuffer.go:12:11:    flow: ~r0 = b:  
buffreader/mybuffer.go:12:11:        from return b, nil (return) at buffreader/mybuffer.go:15:2  
buffreader/mybuffer.go:9:7: mb does not escape  
buffreader/mybuffer.go:12:11: make([]byte, (*bytes.Buffer).Len(mb.buffer), SIZE) escapes to heap  
buffreader/reader.go:34:12: buf does not escape  
buffreader/reader.go:34:31: b does not escape  
buffreader/reader.go:43:12: buf does not escape  
.....
```

The benchmark code

```
func BenchmarkRead1(b *testing.B) {
    buf := bytes.NewBufferString("gocon23")
    bt := make([]byte, SIZE)

    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        read1(buf, bt)
    }
}

func BenchmarkRead2(b *testing.B) {
    mbuf := &MyBuffer{buffer: bytes.NewBufferString("gocon23")}

    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        read2(mbuf)
    }
}
```

Benchmark results

Command:

```
go test -run none -bench Read -benctime 3s -benchmem
```

Output:

```
goos: darwin
goarch: arm64
pkg: example.com/goconf23/buffreader
BenchmarkRead1-8      1000000000          2.078 ns/op        0 B/op      0 allocs/op
BenchmarkRead2-8      325888257           11.05 ns/op       8 B/op      1 allocs/op
PASS
ok   example.com/goconf23/buffreader 7.286s
```

Better visualization with pprof

The screenshot shows the pprof web interface with the following navigation bar: pprof, VIEW ▾, SAMPLE ▾, REFINE ▾, CONFIG ▾, DOWNLOAD, and a search bar labeled "Search regexp".

example.com/goconf23/buffreader.(*MyBuffer).Read

/Users/harsh0240/Projects/go_playground/goconf23/buffreader/mybuffer.go

```
Total: 3.33GB 3.33GB (flat, cum) 99.87%
 8      .      .      buffer *bytes.Buffer
 9      .      .      }
10     .      .
11     .      .      func (mb *MyBuffer) Read() ([]byte, error) { // HL
12     .      .
13 3.33GB 3.33GB      b := make([]byte, mb.buffer.Len(), SIZE)
14     .      .
15     .      .      copy(b, mb.buffer.Bytes())
16     .      .      return b, nil
17     .      .      }
```

example.com/goconf23/buffreader.read2

/Users/harsh0240/Projects/go_playground/goconf23/buffreader/reader.go

```
Total: 3.33GB 3.33GB (flat, cum) 99.87%
39      .      .      return read
40      .      .      }
41      .
42      .      .      //go:noinline
43      .      .      func read2(buf *MyBuffer) []byte { // HL
44 3.33GB 3.33GB      myRead, err := buf.Read()
45      .      .      if err != nil {
46      .      .          return nil
47      .      .
48      .      .      }
49      .      .      return myRead
      .      .      }
```

Summary and takeaways #1

- Optimize for correctness first.

最初に正確性を最適化します。

- Sharing pointers down usually stays on the stack.

共有ポインターは通常、スタック上に残ります。

- Sharing pointers up usually escapes to the heap.

ポインターの共有は通常、ヒープにエスケープされます。

- Try passing variables as arguments instead of returning.

変数を返す代わりに引数として渡してみてください。

Summary and takeaways #2

- Variables escape when addresses are captured by closures or other variables.

アドレスがクロージャまたは他の変数によってキャプチャされると、変数はエスケープされます。

 - Pass variables as arguments to closures.

変数を引数としてクロージャに渡します。
- Variables escape when their size is not known at compile time.

コンパイル時にサイズが不明な場合、変数はエスケープされます。

 - Initializing slices with constant size (not large) is better.

スライスを一定のサイズ(大きくない)で初期化する方が良いでしょう。

Summary and takeaways #3

- Go will only allocate on the stack when it is sure that the variable is not used after the function returns.

Go は、関数が戻った後に変数が使用されていないことが確実な場合にのみスタックに割り当てます。

- There are also additional cases (besides the examples) where variables escape to the heap.

(例以外にも) 変数がヒープにエスケープされる追加のケースもあります。

- Use the tooling, don't guess.

推測しないで、ツールを使用してください。

FAQs

Does it matter where a variable is allocated?

変数がどこに割り当てられるかは重要ですか？

- It does not matter from a correctness standpoint, Go will take care of keeping it alive as long it is going to be used.

正確性の観点からは問題ではありません。使用される限り、Go がそれを存続させるよう処理します。

- However, it does matter for performance considerations because stack allocations are faster and do not cause GC overheads.

ただし、スタック割り当ての方が高速で GC オーバーヘッドが発生しないため、パフォーマンスを考慮すると重要になります。

Should I care?

気にした方がいいでしょうか？

- No, if your program is already fast enough.

いいえ、プログラムがすでに十分に高速である場合には可能です。

- Yes, if you can prove that memory allocations are costly in your program.

はい、プログラムでのメモリ割り当てにコストがかかることが証明できれば可能です。₃₆

How do I enable/disable escape analysis?

エスケープ分析を有効/無効にするにはどうすればよいですか？

- It's already there in the compilation process (and cannot be disabled).
これはコンパイル プロセスにすでに存在しています (無効にすることはできません)。
- There is a command which does not enable it but just logs the optimization steps.
これを有効にせず、最適化ステップをログに記録するだけのコマンドがあります。

What is the cost of it?

費用はいくらですか？

- It is a compile-time optimization that can sometimes lead to longer compilation times.
これはコンパイル時の最適化であり、コンパイル時間が長くなる場合があります。
- However, these downsides are generally outweighed by the benefits of escape analysis in most cases.

ただし、ほとんどの場合、これらのマイナス面はエスケープ分析のメリットによって補われます。

Resources

FAQ about stack and heap (https://go.dev/doc/faq#stack_or_heap)

Escape Analysis in Go code (<https://tip.golang.org/src/cmd/compile/internal/escape/escape.go>)

Talk about escape analysis (<https://youtu.be/ZMZpH4yT7M0>)

Go Escape Analysis Flaws (<https://docs.google.com/document/d/1CxgUBPlx9ijzkz9jWkb6tlpTe5q32QDmz8l0BouG0Cw/edit>)

Series about mechanics of memory allocations and more (<https://www.ardanlabs.com/blog/2017/05/language-mechanics-on-stacks-and-pointers.html>)

Informative slides on escape analysis (<https://slides.com/jalex-chang/go-esc>)

Thank you

Harsh Gupta

Software Engineer, Money Forward Inc.

harshgupta0240@gmail.com (<mailto:harshgupta0240@gmail.com>)

<https://github.com/harsh0240/goconf23> (<https://github.com/harsh0240/goconf23>)

[@dhrsh24](http://twitter.com/dhrsh24) (<http://twitter.com/dhrsh24>)

