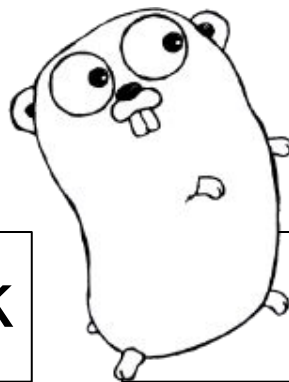




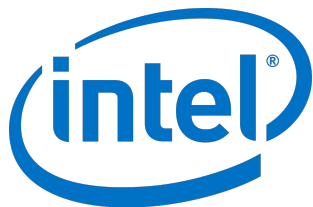
gocritic



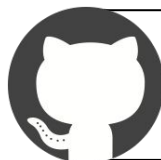
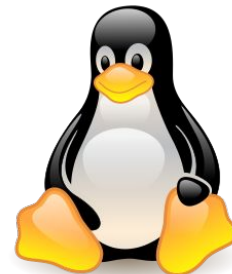
lintpack



golang



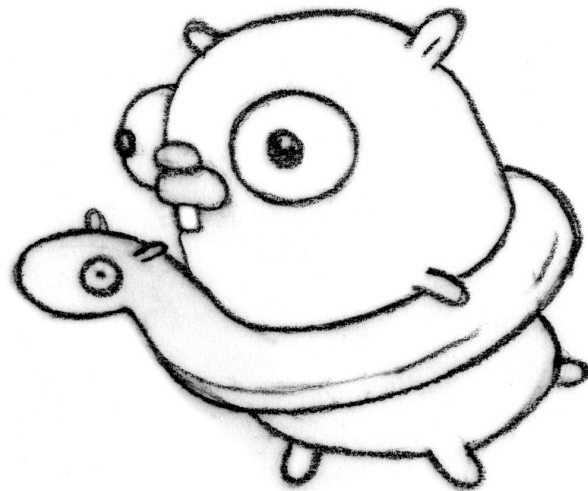
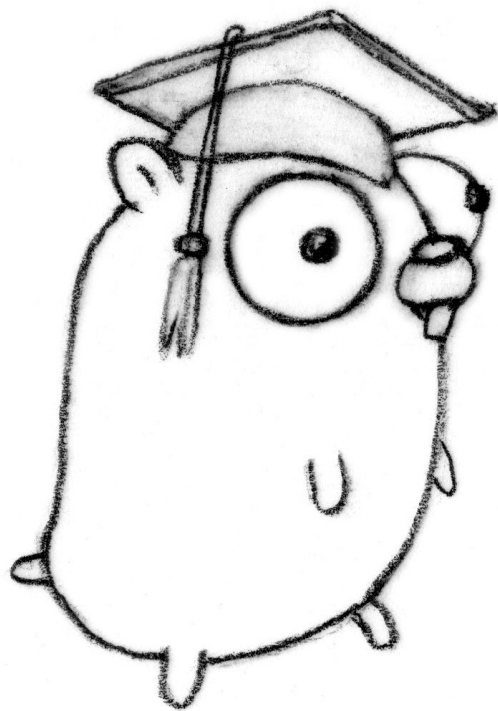
open source



@quasilyte

Некоторые другие мои проекты

- <https://github.com/go-toolsmith>
- <https://github.com/Quasilyte/go-consistent>
- <https://github.com/Quasilyte/go-namecheck>
- <https://speaking-clubs-nizhny.github.io>
- <https://speaking-clubs-kazan.github.io>



Into Go from the inside



“From the inside” подход

Вы узнаете некоторые особенности внутреннего устройства Go, далеко не все из которых являются чем-то, за что вы могли бы его полюбить.

Чем лучше вы знаете какую-то технологию, тем отчётливее видны её недостатки.

The background is a dark, almost black, 3D-rendered scene. It features a grid of rectangular blocks or platforms of varying heights. Each block is outlined with a bright blue, glowing light that creates a sense of depth and perspective. The lines of light are sharp and vibrant, contrasting sharply with the dark environment. The overall effect is futuristic and technological, suggesting a digital or data-related theme.

Часть 1

Эффективное использование структур
данных



{Интерфейсные значения}

Особенности полиморфизма в Go

Интерфейсы и реализации

```
type Stringer interface {  
    String() string  
}  
  
type point struct { x, y float64 }  
  
func (p point) String() string {  
    return fmt.Sprintf("<%d, %d>", p.x, p.y)  
}
```

$I \{ \underbrace{itab}_{\substack{T \text{ itab} \\ \text{for } I}} ; \underbrace{data}_T \}$

itab хранит данные о динамическом типе

I - статический тип интерфейсного значения

T - обёрнутый конкретный тип

itab

```
type itab struct {  
    itype *interfaceType  
    dtype *typeInfo  
    hash  uint32 // Для type switch  
}  
  
type interfaceType {  
    pkg string  
    typ  *typeInfo  
}
```

```
type T struct {}  
type I interface { ... }
```

```
var a *T = new(T)  
var b I = new(T)
```

static
type

same
init
expr

```
type T struct {}  
type I interface {...}
```

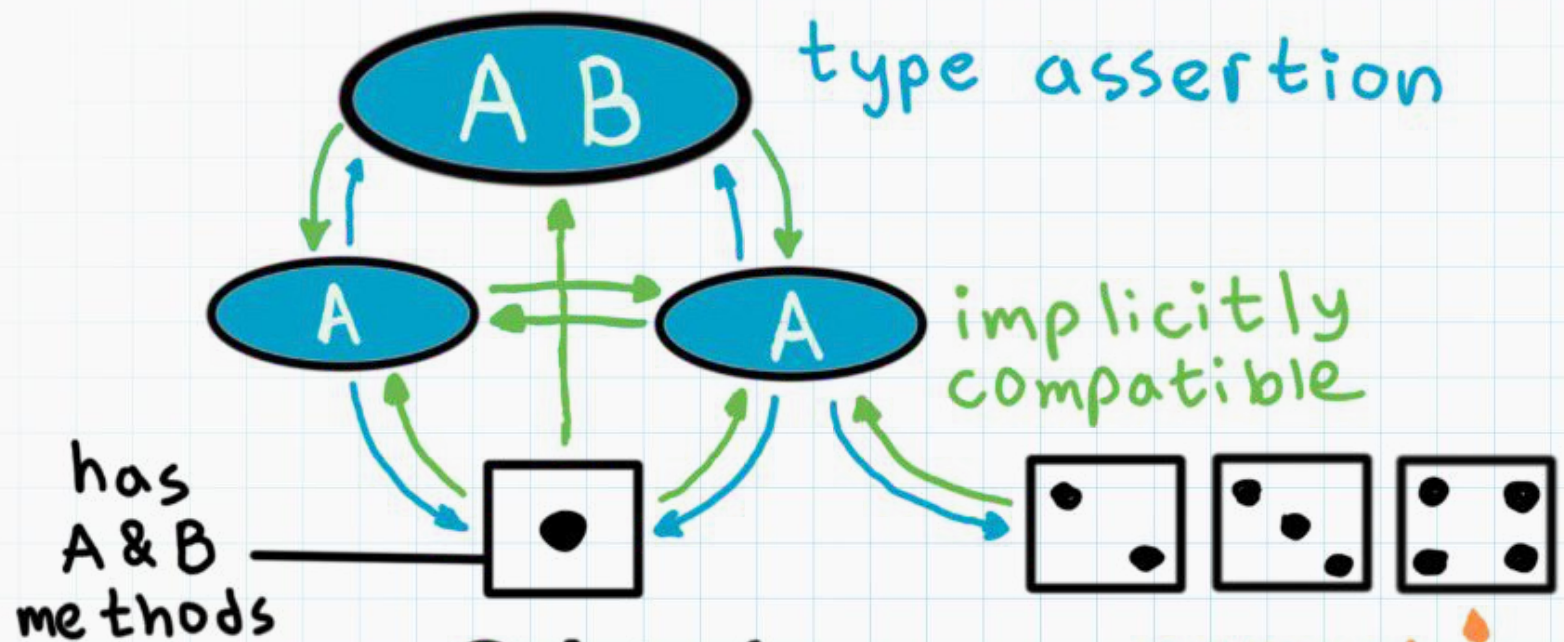
```
var a *T = new(T)  
var b I = new(T)
```

carries no itab

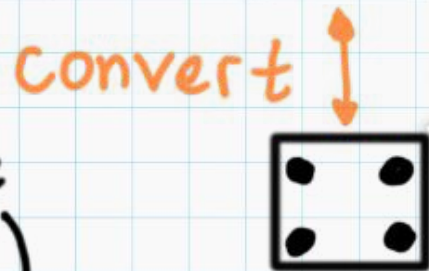
polymorphic

will be boxed

b - интерфейсное значение



- interface
- concrete type (e.g. struct)



Пустой интерфейс

Особое значение в Go имеет пустой интерфейс, `interface{}`.

Любое значение может быть неявно преобразовано к `interface{}`, но не наоборот. Для обратного преобразования используется `type assertion`.

Неявный (implicit) boxing

```
var globalX interface{}  
  
func int2boxed(x int) {  
    // x преобразуется в интерфейсное  
    // значение при присваивании.  
    // Каждый раз выделяется память в куче.  
    globalX = x  
}
```

Выделение памяти

До Go 1.4, интерфейсные значения оптимизировались при `sizeof(T) <= sizeof(word)`. Теперь внутри всегда указатель.

См. <https://golang.org/issue/17725> для подробностей об оптимизациях операции выделения интерфейсного значения.

Type assertion: I->T

```
var iface interface{} = int32(10)

a, isInt := iface.(int)
fmt.Println(a, isInt) // => 0, false

b, isInt32 := iface.(int32)
fmt.Println(b, isInt32) // => 10, true

fmt.Println(iface.(int32)) // => 10
```

Type assertion: I->I

```
type I1 interface {  
    Method1()  
    Method2()  
}  
type I2 interface { Method1() }  
type I3 interface { Method1() }  
  
// I1 замещает I2 и I3.  
// I2 может замещать I3, и наоборот.
```

Interface -> Interface

Преобразование из интерфейса в интерфейс может требовать дополнительных вычислений при первом преобразовании так как Go не хранит все `{itype, dtype}` комбинации.

Таблицы для этих типов достраиваются во время выполнения, по мере необходимости.

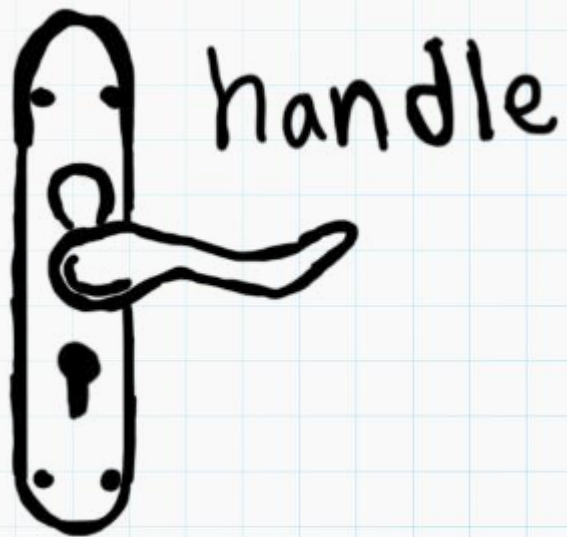


{Структуры данных}

Высокоуровневый обзор встроенных в
Go структур данных

Структуры данных в Go

- Массив (array)
- Слайс / динамический массив (slice)
- Строка (string)
- Словарь / ассоциативный массив (map)
- Канал (chan)



Handle - структура,
которая содержит
указатель на данные,
но не сами данные.

Структуры данных в Go

<code>array</code>	Value тип
<code>slice</code>	Handle к значению
<code>string</code>	Immutable handle к значению
<code>map</code>	Ссылочный тип
<code>chan</code>	Ссылочный тип

Структуры данных в Go

<code>sizeof([N]T)</code>	<code>sizeof(T) * N</code>
<code>sizeof([]T)</code>	24 (on AMD64), 3 words
<code>sizeof(string)</code>	16 (on AMD64), 2 words
<code>sizeof(map[K]V)</code>	8 (on AMD64), 1 word
<code>sizeof(chan T)</code>	8 (on AMD64), 1 word


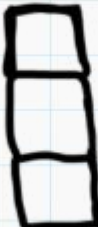

string != immutable []byte

```
type string struct {  
    data *byte  
    len int  
}  
type byteSlice struct {  
    data *byte  
    len int  
    cap int  
}
```

Слайс (slice)

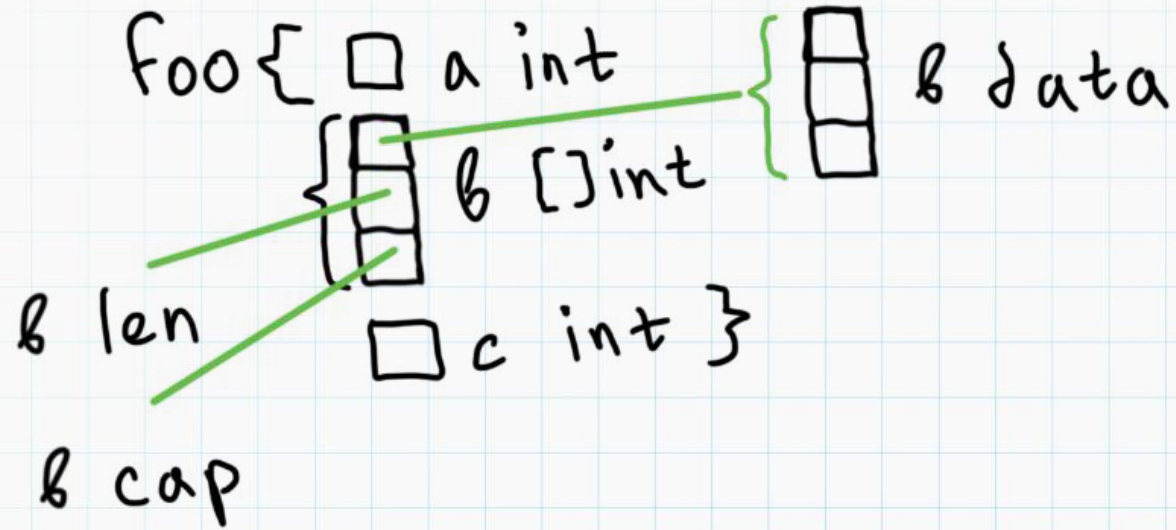
Динамически растущий,
мутабельный массив.

C#: List, Java: ArrayList, C++: std::vector

foo {  a int
 b [3] int
 c int }

$\text{sizeof(foo): } \text{sizeof(int)} * 5$

Memory used: $\text{sizeof(int)} * 5$



$\text{sizeof(foo): } \text{sizeof(int)} * 5$

Memory used: $\text{sizeof(int)} * 5 + \text{sizeof(int)} * 3$

range по массиву

```
var ops[256]operation

// Копирует массив дважды.
for _, op := range ops {
    if op.enabled {
        op.fn()
    }
}
```

range по массиву

```
var ops[256]operation

// Копирует массив только один раз.
for _, op := range &ops {
    if op.enabled {
        op.fn()
    }
}
```



{Классы памяти}

Где могут располагаться локальные
данные программы

Статическая память (static)

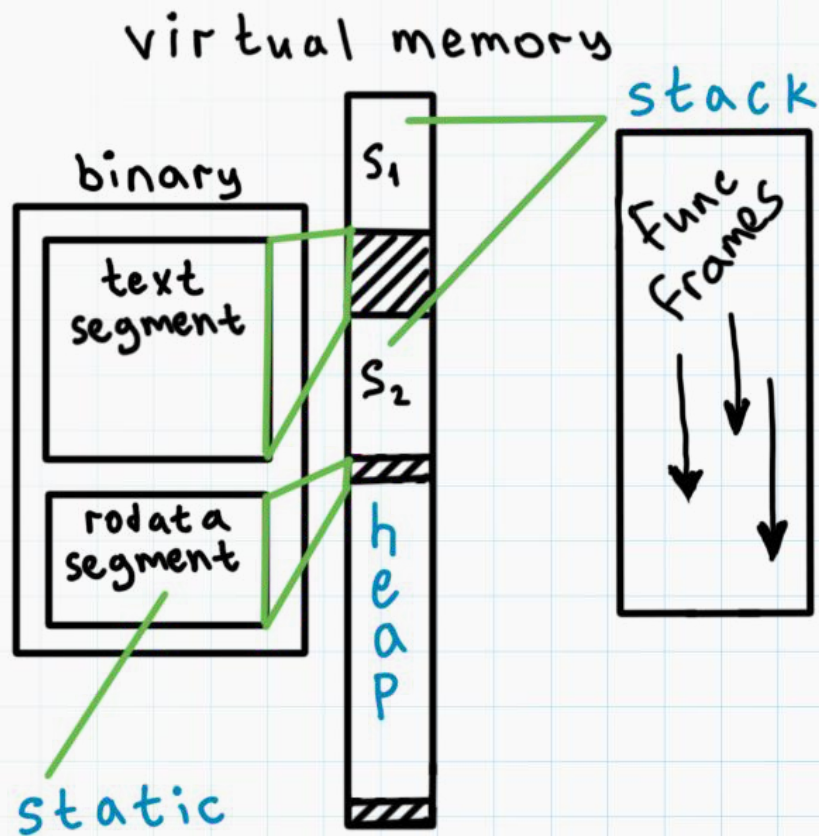
- Никогда не очищается
- Не требует выделения (выделена заранее)
- Глобальные переменные
 - Инициализация внутри package init
- Так называемые static tmp значения
 - Не требуют инициализации

Стековая память (stack)

- Очищается при возврате из функции
- Выделяется смещением SP регистра
- Локальные переменные
- Входные и выходные параметры функции
- Память под non-escaping данные

Куча (heap)

- Очищается сборщиком мусора
- Выделяется менеджером памяти
- Память под escaping данные



Каждая горутина
имеет отдельный стек

Иллюстрация классов памяти

Стековый кадр (фрейм)

Фрейм является хранилищем для локальных данных, параметров и результатов функции.

System stack

Горутины в Go не используют “системный стек”. Исключение - runtime (см. функцию `systemstack`).

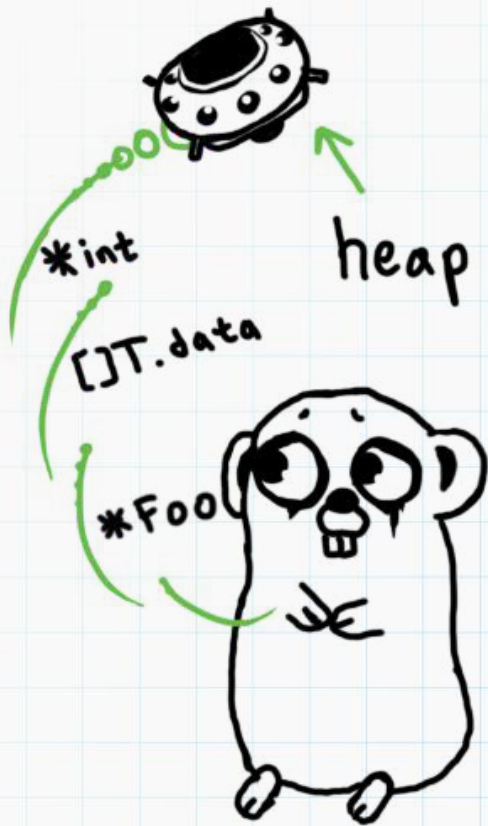
Stack vs Heap

Аллокация на стеке всегда предпочтительна и не добавляет работы сборщику мусора.

The background is a dark, abstract 3D scene. It features several rectangular blocks or cubes of varying heights and positions. These blocks are interconnected by a network of glowing blue lines that appear to be light trails or data paths. The lines and blocks are arranged in a way that suggests a complex, interconnected system, possibly representing a memory structure or a data flow. The overall aesthetic is futuristic and technological.

Выделение в куче

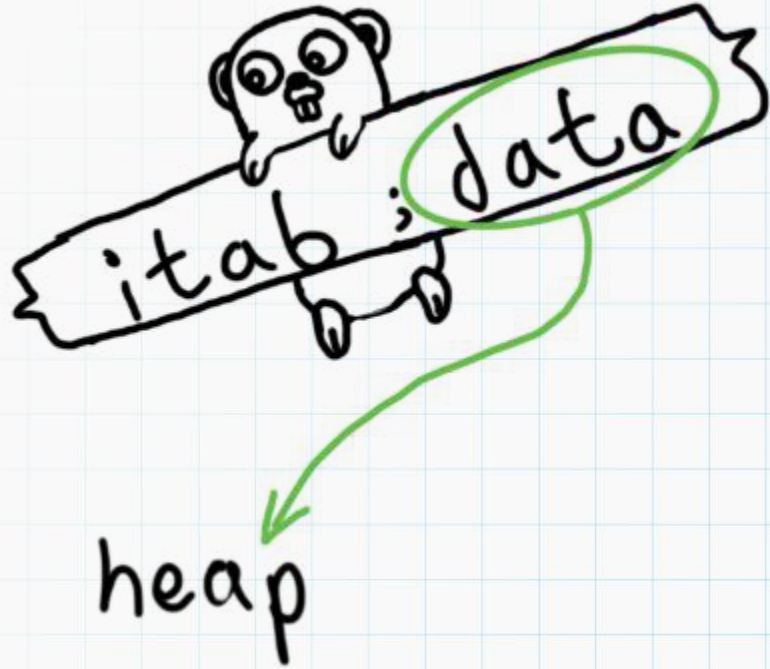
Когда объекты не размещаются на стеке и нуждаются в присмотре со стороны сборщика мусора



Данные, хранимые по
нелокальным
указателям,
размещаются в куче



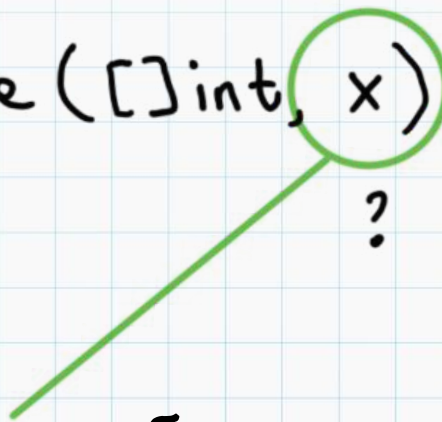
Крупные (~64 KB)
объекты всегда
размещаются в куче



Данные интерфейсного значения, как правило, выделяются в куче


```
slice := make([]int, x)
```

?



Если размер создаваемого объекта неизвестен на этапе компиляции, аллокация на стеке невозможна.

The background is a dark, abstract 3D scene. It features several rectangular blocks or cubes of varying sizes, some of which are illuminated from within, creating a glowing effect. Bright blue lines, resembling light trails or data paths, crisscross the scene, connecting different points and creating a sense of movement and connectivity. The overall aesthetic is futuristic and technological.

{Escape analysis}

Обзор анализа указателей

Явное выделение памяти

```
func noAlloc() int {  
    x := new(int) // Указатель на x локален  
    return *x  
}
```

```
func alloc() *int {  
    x := new(int)  
    return x // x "убегает"  
}
```

“Убегающие” указатели

Основной компонент компилятора, который определяет является ли указатель локальным или нет, называется `escape analysis`.

Данный анализ пытается найти те объекты, указатели на которые не покидают фрейма, на котором они определены.

Escape analysis (почти) работает

Спецификация языка не регламентирует как и где выделяется память ни для одной из языковых конструкций.

В идеальном мире в “кучу” попадает только то, что должно туда попадать для корректности программы.

Просмотр результатов escape analysis

```
$ go tool compile -m e.go
```

```
e.go:3:6: can inline NoAlloc
```

```
e.go:8:6: can inline Alloc
```

```
e.go:4:10: NoAlloc new(int) does not escape
```

```
e.go:9:10: new(int) escapes to heap
```

Does not escape 👍

Данные могут быть размещены в стеке. “Время жизни” привязано к фрейму вызова.

Escapes to heap 👉

Данные будут размещены в куче.
“Время жизни” определяется во время выполнения, сборщиком мусора.

Leaking param 🖐️

Объект-параметр будет выделен в куче в месте вызова.

Leaking param content 🖐

Данные внутри объекта-параметра
будут выделены в куче в месте
вызова.

```
type pair struct {  
    x *int  
    y *int  
}
```

content


object

leaking param

leaking param content

Только 2 уровня гранулярности

```
type pair struct{ x, y *int }
var sink *int
func tooDumb() {
    p := pair{x: new(int), y: new(int)}
    sink = p.x // Только одно поле "убегает"
}
// new(int) escapes to heap (2 раза)
```



Самоприсваивание

```
type box struct{ data *byte }  
func dumb(b *box) {  
    *b = *b  
}  
// leaking param content: s
```



Escaped

Самоприсваивание слайса

```
type object struct {  
    slice1 []int  
    slice2 []int  
}  
func selfAssignSlicing(o *object) {  
    o.slice1 = o.slice2[:]  
}  
// ignoring self-assignment
```


Самоприсваивание слайса

```
type object struct {  
    slice []int  
    arr    [10]int  
}  
func selfAssignSlicing(o *object) {  
    o.slice = o.arr[:]  
}  
// leaking param: o  
// o.arr escapes to heap
```



Escaped

Присваивание поля через указатель

```
type box struct{ data *byte }  
  
func dumber() {  
    b := new(box)  
    b.data = new(byte)  
}  
// new(byte) escapes to heap
```



Присваивание поля через указатель

```
type box struct{ data *byte }  
  
func worksFine() {  
    var b box  
    b.data = new(byte)  
}
```

Присваивание поля через указатель

```
type box struct{ data *byte }  
  
func stillDumb() {  
    var b box  
    b2 := &b  
    b2.data = new(byte)  
}  
// new(byte) escapes to heap
```



Присваивание поля через указатель

```
type stack struct { elems []int }  
  
func (st *stack) push(v int) {  
    st.elems = append(st.elems, v)  
}  
// leaking param content: st
```

Escaped
Escaped

Повторные присваивания

Escaped

```
var sink *int
func duuuubm() {
    a := new(int) // Убегает в кучу
    a = new(int)  // Тоже размещается в куче
    a = new(int)  // Заслуженно идёт в heap
    sink = a
}
// new(int) escapes to heap (3 раза!)
```

Indirect call

```
type fooer interface{ Foo(xs []int) }  
type myFooer struct{ _ int }  
func (f *myFooer) Foo(xs []int) {}  
func alsoDumb(f fooer, xs []int) {  
    f.Foo(xs)  
}  
// leaking param: f  
// leaking param: xs
```




```
if indirect {  
    // We know nothing!  
    // Leak all the parameters  
    for _, arg := range args {  
        e.escassignSinkWhy(call, arg,  
        if Debug['m'] > 3 {  
            fmt.Printf("%v::esccai  
        }  
    }  
}
```

cmd/compile/internal/gc/esc.go



Иногда данные
размещаются в куче
даже тогда, когда
стек был бы более
оптимальным выбором

```
go build -gcflags="-m=2" .
```

Флаг -m может иметь значение от 1 до 4, определяя количество информации об оптимизациях.

Пример с -m=2

```
func leakInt(x *int) *int {  
    alias := x  
    return alias  
}  
// leaking param: x to result ~r1 level=0  
//     from alias (assigned) at e.go:4:8  
//     from ~r1 (return) at e.go:5:2
```



`{string}`

Оптимизация коротких строк в Go

strings.Index vs bytes.Index

```
var haystack = bytes.Repeat([]byte("a"), 100)
var needle = "aaa"
```

```
// (A)
```

```
strings.Index(string(haystack), needle)
```

```
// (B)
```

```
bytes.Index(haystack, []byte(needle))
```

strings.Index vs bytes.Index

StringsIndex-8	73.3	ns/op	112	B/op
BytesIndex-8	15.1	ns/op	0	B/op

Даже если `string(b)` не покидает фрейма, выделение будет происходить в куче, если длина слайса превышает длину "малого буфера" (32 байта).

Short (local) string concatenation

```
func concatLen(x, y string) int {  
    result := x + y  
    return len(result)  
}
```

// Когда $\text{len}(x) + \text{len}(y) \leq 32$, динамической
// аллокации под `result` происходить не будет.
// Go выделяет маленький буфер на стеке
// для результата локальных конкатенаций.



`{map}`

Особенности встроенного типа `map` в
Go

map[K]V trivia

- Хеш-таблица
- Нельзя предоставить свою хеш-функцию
- Недетерменированный порядок обхода
- Не синхронизирована (NTS)
- Нельзя брать адрес элемента внутри map
- `delete(m, k)` не освобождает память

Fast версии map

Существуют “_fast” версии операций над map, когда удовлетворены определённые условия.

Fast map

- Размер ключей не превышает 128 байт
- Размер значений не превышает 128 байт
- Специализация для ключа `uint32/int32`,
`uint64/int64`, `string`

Хеширование ключа

Иногда выгоднее использовать `uint32/uint64` ключ и хешировать объекты своим алгоритмом.

$\text{len}(m) < 10$

Для очень маленьких наборов
эффективнее использовать слайс и
линейный поиск.

Ключи типа `byte/int8/etc`

Массив вида `[256]T` будет намного эффективнее, чем `map[byte]T`.

map[T]bool vs map[T]struct{}

Для set-подобного map немного эффективнее иметь struct{} в качестве типа-значения.

Optimized: очистка map

```
// Очистить map, переиспользовать память.  
// https://golang.org/cl/110055.  
for k := range m {  
    delete(m, k)  
}  
  
// Очистить map без переиспользования памяти.  
m = make(map[K]V)
```

Optimized: map append

```
// Вместо 2 mapaccess к m[k] будет только 1.  
// https://golang.org/cl/100838.  
m[k] = append(m[k], v)
```

```
// Код ниже не оптимизируется.  
s := m[k]  
m[k] = append(s, v)
```

Optimized: map append

```
// Не оптимизируется, если в выражении,  
// которое вычисляет ключ, есть побочные  
// эффекты (или вызов любой функции).  
m[getKey(x)] = append(m[getKey(x)], v)
```

```
// Вынесите вычисление ключа во  
// временную переменную.  
k := getKey(x)  
m[k] = append(m[k], v)
```