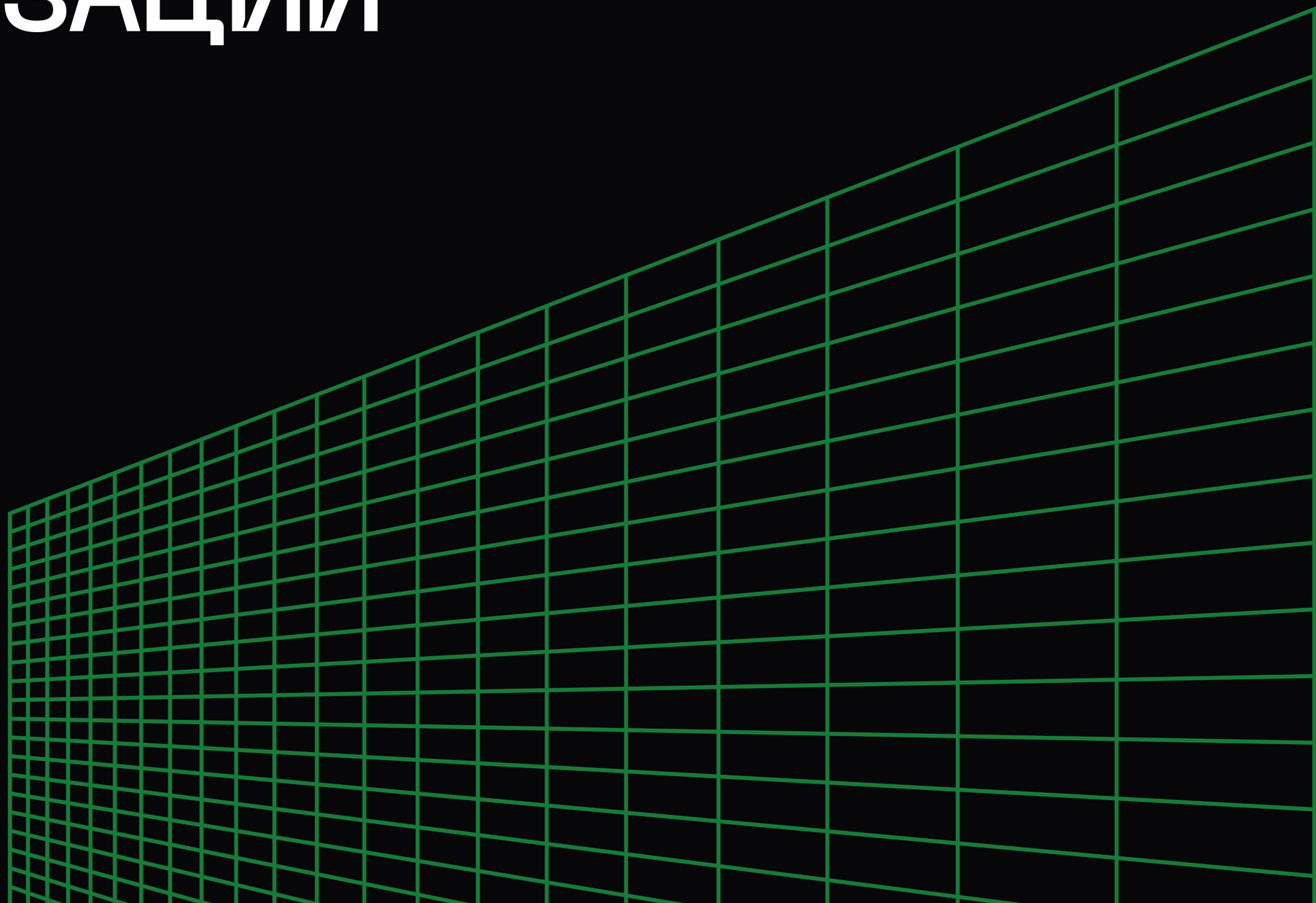
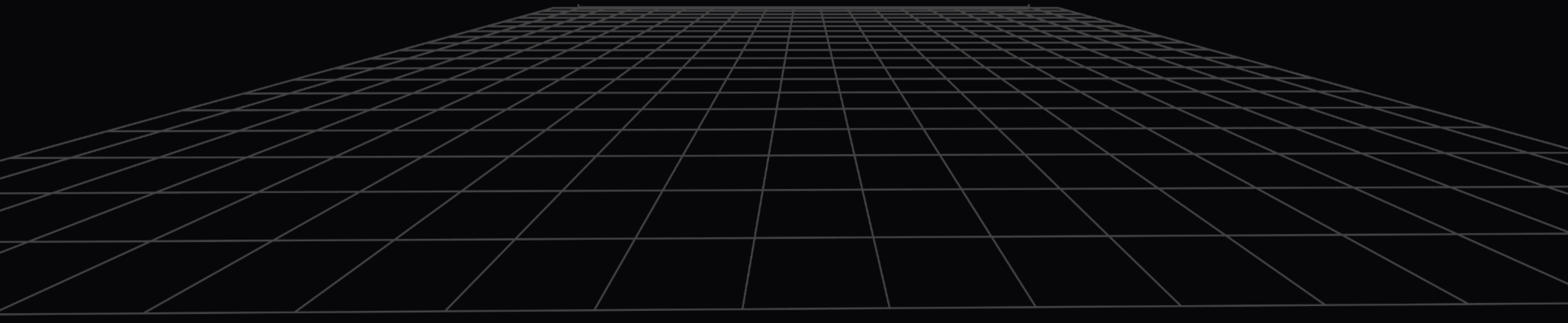
ОПТИМИЗАЦИИ В GO



Balun.Courses

ПРОВЕРЬ ЗАПИСЬ

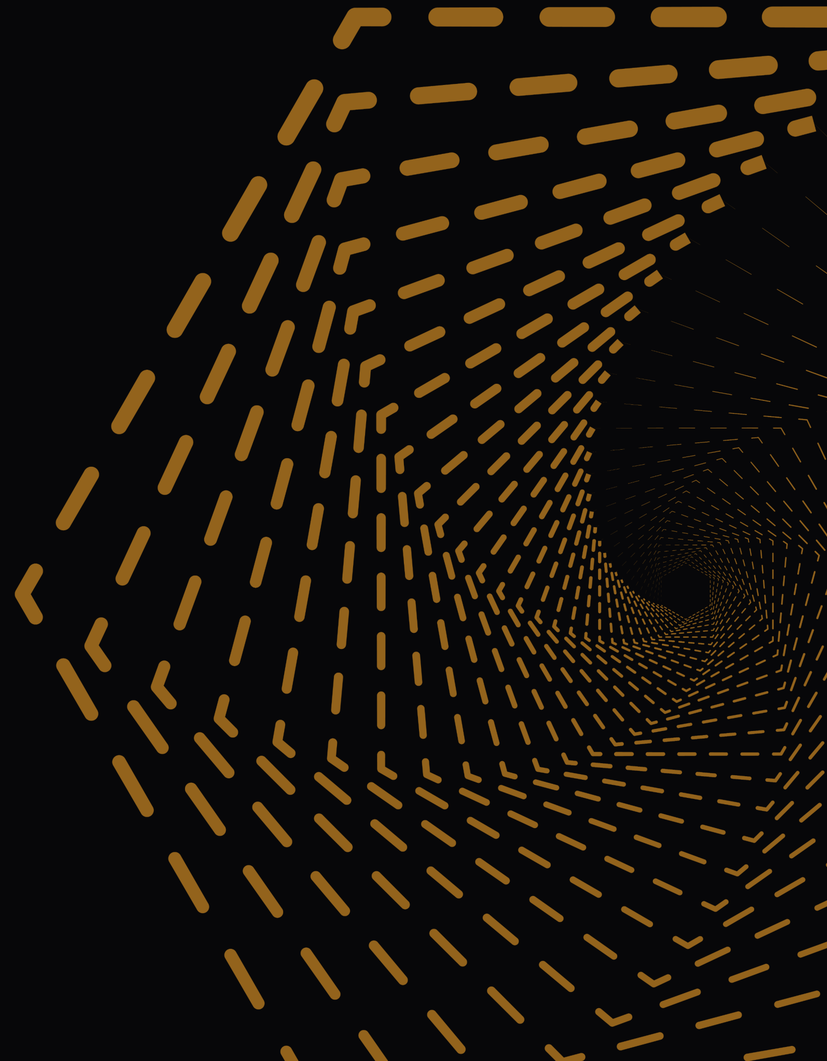


ПРАВИЛА ЗАНЯТИЯ

1. вопросы в чате можно задавать **в любое время**

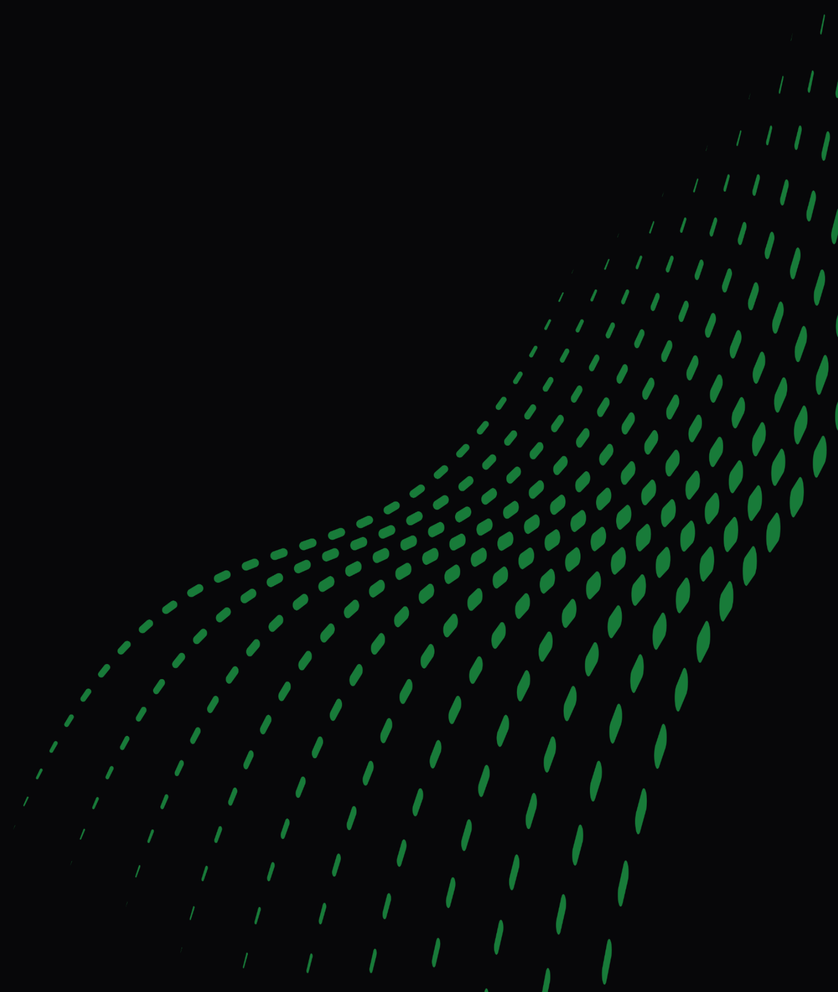
2. вопросы голосом задаем **по поднятой руке в Zoom**

3. ответы на вопросы будут **в запланированных местах**



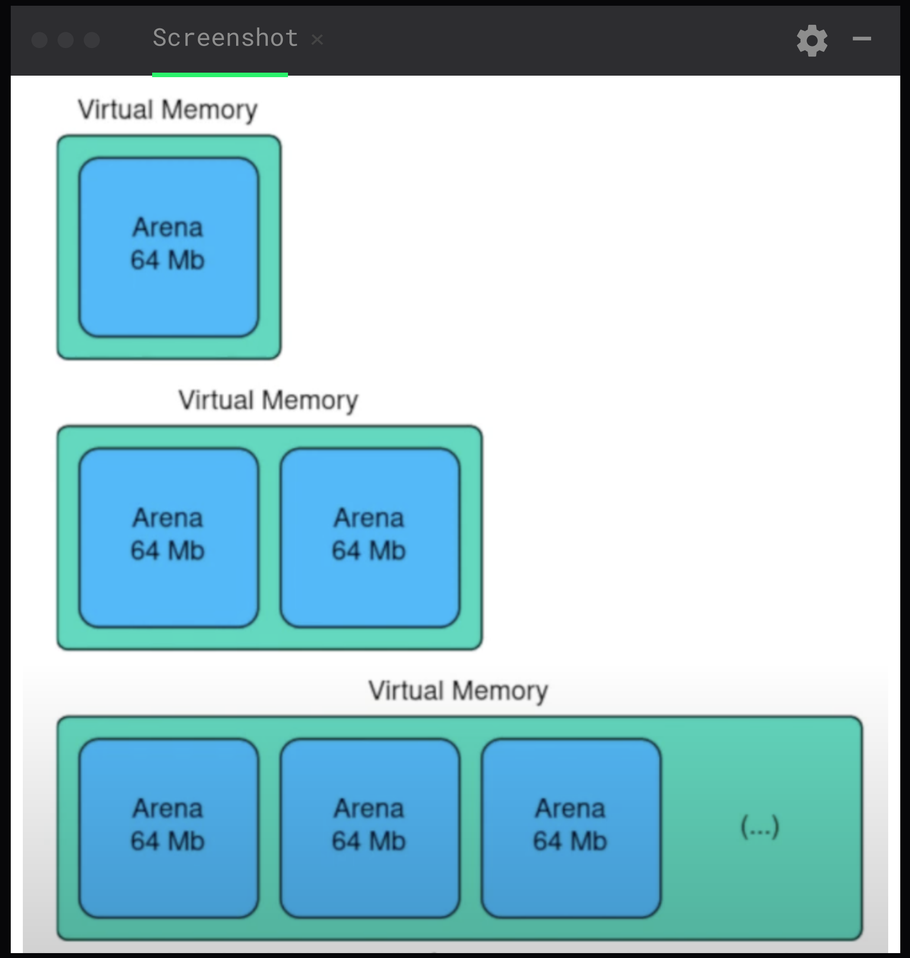
ПЛАН ЛЕКЦИИ

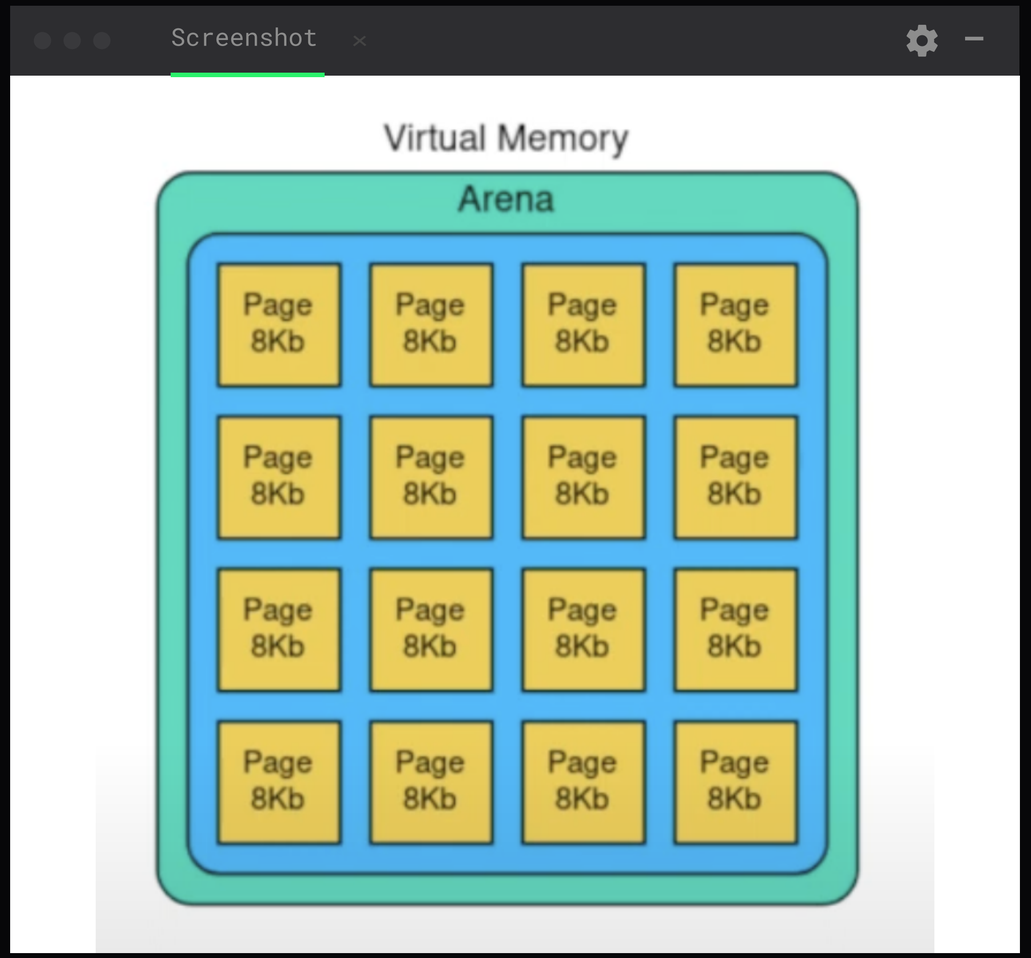
1. Внутреннее устройство памяти в Go
2. Бенчмарки в Go
3. Внутреннее представление типов данных в Go
4. Пакет unsafe
5. Шаблоны оптимизации



УСТРОЙСТВО ПАМЯТИ

Runtime Go имеет свою собственную преаллоцированную память
для ускорения выделения памяти внутри программы





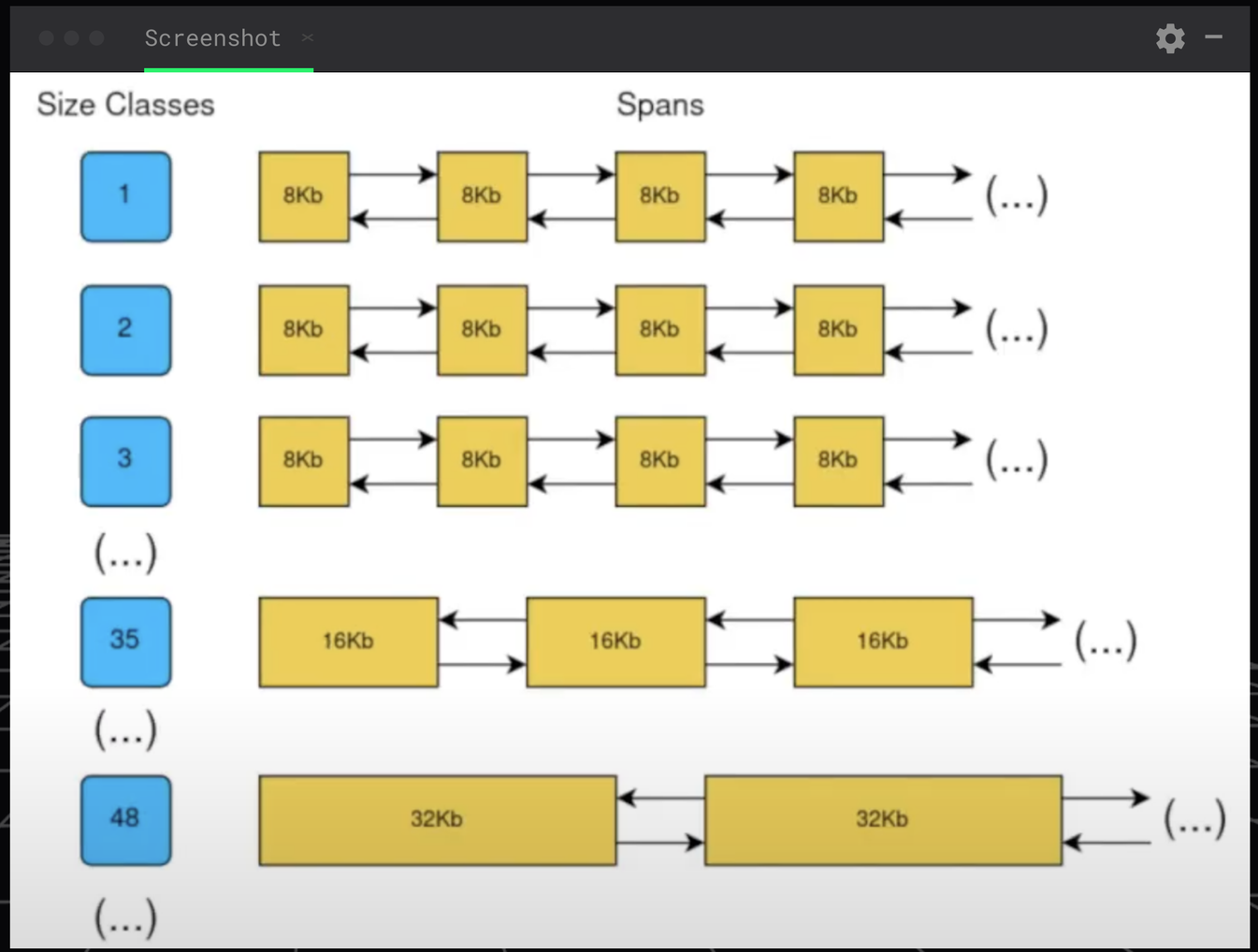
УСТРОЙСТВО ПАМЯТИ

Почему на Windows выделяют арены меньшего размера?

Screenshot

```
// heapArenaBytes is the size of a heap arena. The heap
// consists of mappings of size heapArenaBytes, aligned to
// heapArenaBytes. The initial heap mapping is one arena.
//
// This is currently 64MB on 64-bit non-Windows and 4MB on
// 32-bit and on Windows. We use smaller arenas on Windows
// because all committed memory is charged to the process,
// even if it's not touched. Hence, for processes with small
// heaps, the mapped arena space needs to be commensurate.
// This is particularly important with the race detector,
// since it significantly amplifies the cost of committed
// memory.
heapArenaBytes = 1 << logHeapArenaBytes // 67108864 ~ 64 Mb
logHeapArenaBytes = (6+20)*(_64bit*(1-goos.IsWindows)*(1-goarch.IsWasm)*(1-goos.IsIos*goarch.IsArm64)) +
    (2+20)*(_64bit*goos.IsWindows) + (2+20)*(1-_64bit) + (2+20)*goarch.IsWasm + (2+20)*goos.IsIos*goarch.IsArm64
```

УСТРОЙСТВО ПАМЯТИ




```go
// Central list of free objects of a given size.
type mcentral struct {
    _         sys.NotInHeap
    spanclass spanClass

    // partial and full contain two mspan sets: one of swept in-use
    // spans, and one of unswept in-use spans. These two trade
    // roles on each GC cycle. The unswept set is drained either by
    // allocation or by the background sweeper in every GC cycle,
    // so only two roles are necessary.
    //
    // sweepgen is increased by 2 on each GC cycle, so the swept
    // spans are in partial[sweepgen/2%2] and the unswept spans are in
    // partial[1-sweepgen/2%2]. Sweeping pops spans from the
    // unswept set and pushes spans that are still in-use on the
    // swept set. Likewise, allocating an in-use span pushes it
    // on the swept set.
    //
    // Some parts of the sweeper can sweep arbitrary spans, and hence
    // can't remove them from the unswept set, but will add the span
    // to the appropriate swept list. As a result, the parts of the
    // sweeper and mcentral that do consume from the unswept list may
    // encounter swept spans, and these should be ignored.
    partial [2]spanSet // list of spans with a free object
    full    [2]spanSet // list of spans with no free objects
}
```

УСТРОЙСТВО ПАМЯТИ

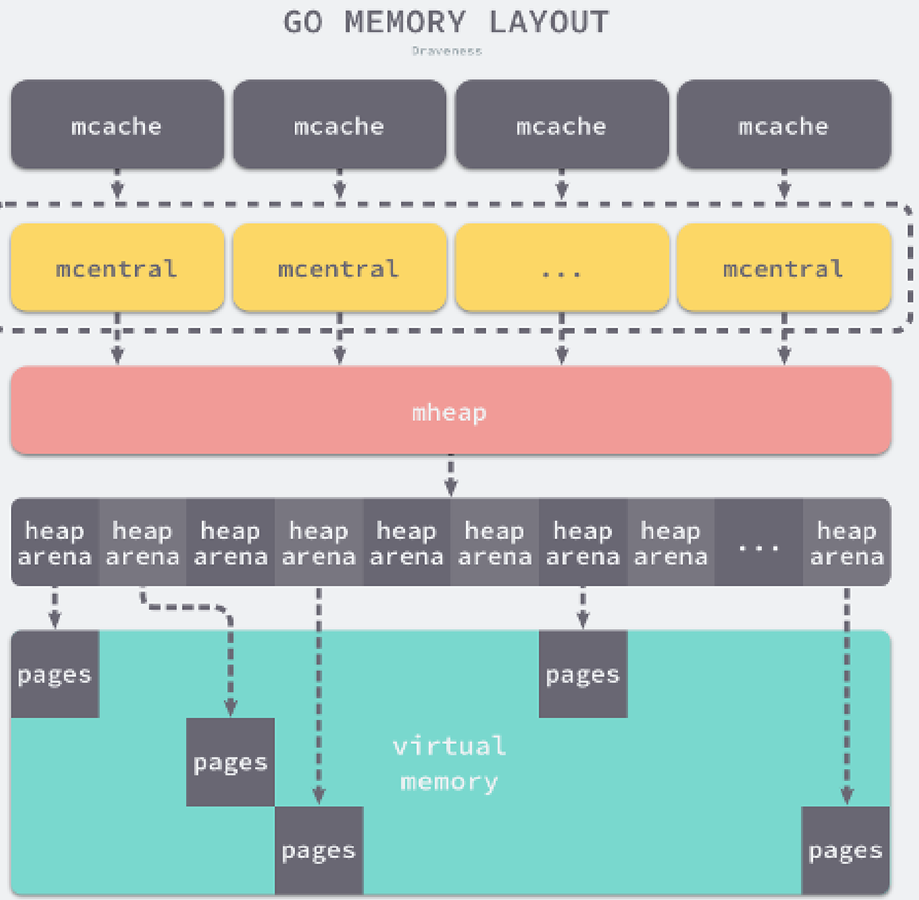
```
// class  bytes/obj  bytes/span  objects  tail waste  max waste  min align
//     1          8        8192     1024           0     87.50%          8
//     2         16        8192      512           0     43.75%         16
//     3         24        8192      341           8     29.24%          8
//     4         32        8192      256           0     21.88%         32
//     5         48        8192      170          32     31.52%         16
//     6         64        8192      128           0     23.44%         64
//     7         80        8192      102          32     19.07%         16
//     8         96        8192       85          32     15.95%         32
//     9        112        8192       73          16     13.56%         16
//    10        128        8192       64           0     11.72%        128
//    11        144        8192       56         128     11.82%         16
//    12        160        8192       51          32      9.73%         32
//    13        176        8192       46          96      9.59%         16
//    14        192        8192       42         128      9.25%         64
//    15        208        8192       39          80      8.12%         16
//    16        224        8192       36         128      8.15%         32
//    17        240        8192       34          32      6.62%         16
//    18        256        8192       32           0      5.86%        256
//    19        288        8192       28         128     12.16%         32
//    20        320        8192       25         192     11.80%         64
//    21        352        8192       23          96      9.88%         32
//    22        384        8192       21         128      9.51%        128
//    23        416        8192       19         288     10.71%         32
//    24        448        8192       18         128      8.37%         64
```

```
//    62       20480       40960       2          0     6.87%       4096
//    63       21760       65536       3        256     6.25%        256
//    64       24576       24576       1          0    11.45%       8192
//    65       27264       81920       3        128    10.00%        128
//    66       28672       57344       2          0     4.91%       4096
//    67       32768       32768       1          0    12.50%       8192
```

```
type mspan struct {
    _    sys.NotInHeap
    next *mspan     // next span in list, or nil if none
    prev *mspan     // previous span in list, or nil if none
    list *mSpanList // For debugging.

    startAddr uintptr // address of first byte of span aka s.base()
    npages    uintptr // number of pages in span

    manualFreeList gclinkptr // list of free objects in mSpanManual spans

    // freeindex is the slot index between 0 and nelems at which to begin
    // for the next free object in this span.
    // Each allocation scans allocBits starting at freeindex until it enc
    // indicating a free object. freeindex is then adjusted so that subse
    // just past the newly discovered free object.
    //
    // If freeindex == nelem, this span has no free objects.
    //
    // allocBits is a bitmap of objects in this span.
    // If n >= freeindex and allocBits[n/8] & (1<<(n%8)) is 0
    // then object n is free;
    // otherwise, object n is allocated. Bits starting at nelem are
    // undefined and should never be referenced.
    //
    // Object n starts at address n*elemsize + (start << pageShift).
    freeindex uint16
```

УСТРОЙСТВО ПАМЯТИ

Screenshot




```go
// Per-thread (in Go, per-P) cache for small objects.
// This includes a small object cache and local allocation stats.
// No locking needed because it is per-thread (per-P).
//
// mcaches are allocated from non-GC'd memory, so any heap pointers
// must be specially handled.
type mcache struct {
    _ sys.NotInHeap

    // The following members are accessed on every malloc,
    // so they are grouped here for better caching.
    nextSample uintptr // trigger heap sample after allocating this many bytes
    scanAlloc  uintptr // bytes of scannable heap allocated

    // Allocator cache for tiny objects w/o pointers.
    // See "Tiny allocator" comment in malloc.go.

    // tiny points to the beginning of the current tiny block, or
    // nil if there is no current tiny block.
    //
    // tiny is a heap pointer. Since mcache is in non-GC'd memory,
    // we handle it by clearing it in releaseAll during mark
    // termination.
    //
    // tinyAllocs is the number of tiny allocations performed
    // by the P that owns this mcache.
    tiny       uintptr
    tinyoffset uintptr
    tinyAllocs uintptr

    // The rest is not accessed on every malloc.

    alloc [numSpanClasses]*mspan // spans to allocate from, indexed by spanClass
```

УСТРОЙСТВО ПАМЯТИ SCAVENGER

```
func sysUnusedOS(v unsafe.Pointer, n uintptr) {
    madvise(v, n, _MADV_DONTNEED)
}
```

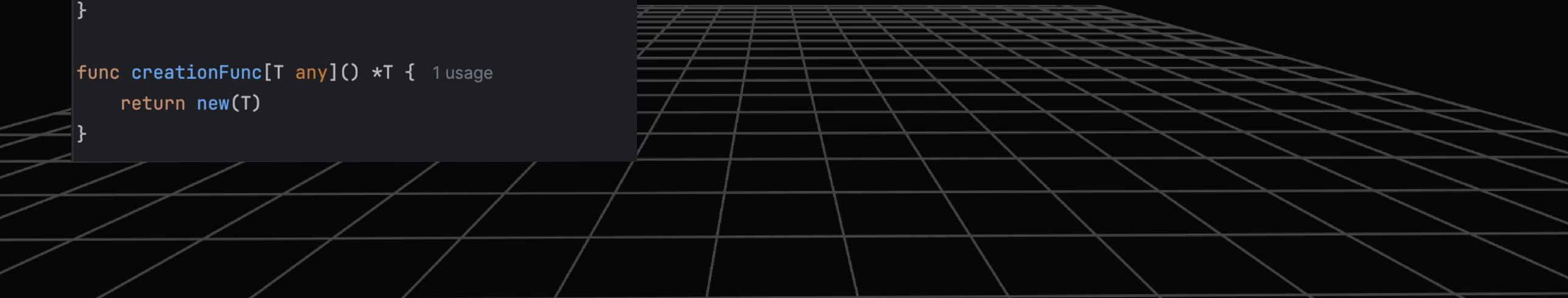
After a successful **MADV_DONTNEED** operation, the semantics of memory access in the specified region are changed: subsequent accesses of pages in the range will succeed, but will result in either repopulating the memory contents from the up-to-date contents of the underlying mapped file (for shared file mappings, shared anonymous mappings, and shmem-based techniques such as System V shared memory segments) or zero-fill-on-demand pages for anonymous private mappings.

БЕНЧМАРКИ

```go
type data struct {  1 usage
    a [10224323]int64
}

// go test -bench=. -benchmem
func BenchmarkSimple(b *testing.B) {
    b.ReportAllocs()

    for i := 0; i < b.N; i++ {
        creationFunc[data]()
    }
}

func creationFunc[T any]() *T {  1 usage
    return new(T)
}
```

```
goos: darwin
goarch: arm64
pkg: lection7
BenchmarkSimple
BenchmarkSimple-12          1569        726315 ns/op    81797126 B/op          1 allocs/op
PASS
```

```go
// The benchmark function must run the target code b.N times.
// During benchmark execution, b.N is adjusted until the benchmark function lasts
// long enough to be timed reliably. The output
//
//  BenchmarkRandInt-8      68453040            17.8 ns/op
```



БЕНЧМАРКИ

```go
// go test -bench=. -benchmem -benchtime=100x
func BenchmarkWithTimer(b *testing.B) {
    b.ReportAllocs()

    for i := 0; i < b.N; i++ {
        b.StopTimer()
        var a int
        s := make([]int64, 1000)
        b.StartTimer()
        // or ResetTimer

        a += len(s)
    }
}
```

```go
// go test -bench=. -benchmem -cpu=x
func BenchmarkParallel(b *testing.B) {
    // -cpu=x
    b.SetParallelism(runtime.NumCPU())
    b.ReportAllocs()

    b.RunParallel(func(pb *testing.PB) {
        // no timer settings here

        for pb.Next() {
            // action
        }
    })
}
```

ЧАСТЫЕ ОШИБКИ

```go
func BenchmarkWrongRecursive(b *testing.B) {
    for i := 0; i < b.N; i++ {
        recursiveFunc(b.N)
    }
}
```

```go
func BenchmarkWrongSingleAction(b *testing.B) {
    recursiveFunc( a: 1_000_000)
}
```

```go
func TestWrongComparison(t *testing.T) {
    res := testing.Benchmark(func(b *testing.B) {
        for i := 0; i < b.N; i++ {
            recursiveFunc(i)
        }
    })
    if res.NsPerOp() <= 1000 {}
}
```

ВНУТРЕННЕЕ ПРЕДСТАВЛЕНИЕ ТИПОВ

```
// SliceHeader is the runtime representation of a slice.
// It cannot be used safely or portably and its representation may
// change in a later release.
// Moreover, the Data field is not sufficient to guarantee the data
// it references will not be garbage collected, so programs must keep
// a separate, correctly typed pointer to the underlying data.
//
// Deprecated: Use unsafe.Slice or unsafe.SliceData instead.
type SliceHeader struct {
    Data uintptr
    Len  int
    Cap  int
}
```

ВНУТРЕННЕЕ ПРЕДСТАВЛЕНИЕ ТИПОВ

```
// layout of Itab known to compilers
// allocated in non-garbage-collected memory
// Needs to be in sync with
// ../cmd/compile/internal/reflectdata/reflect.go:/^func.WritePluginTable.
type itab struct {
    inter *interfacetype
    _type *_type
    hash  uint32 // copy of _type.hash. Used for type switches.
    _     [4]byte
    fun   [1]uintptr // variable sized. fun[0]==0 means _type does not implement inter.
}
```

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

ПАКЕТ UNSAFE

unsafe.Slice, unsafe.String
(Go 1.20+)

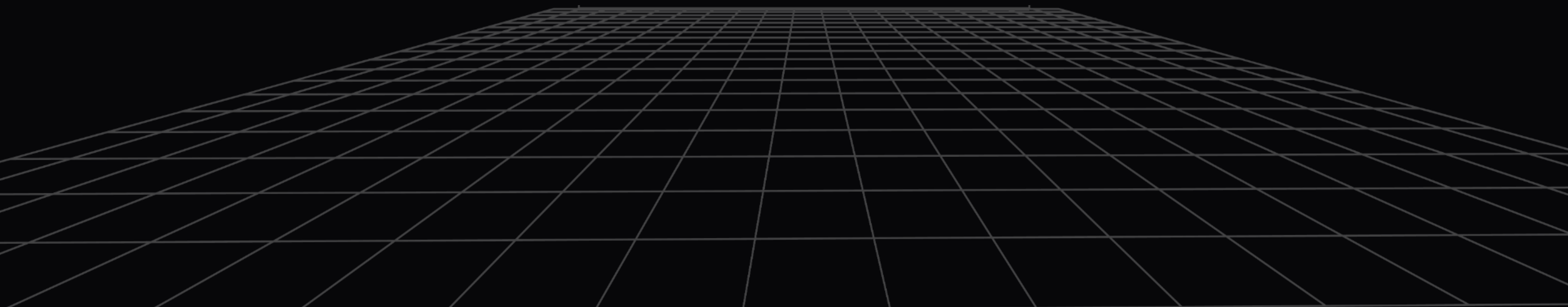
Как жили до этого?

```go
func BenchmarkZeroCopyConverter(b *testing.B) {
    b.StopTimer()
    s := make([]byte, 10_000_000)
    b.StartTimer()

    b.ReportAllocs()

    for i := 0; i < b.N; i++ {
        zeroCopyConverter(s)
    }
}

func zeroCopyConverter(bigSlice []byte) []byte {  1 usage
    str := unsafe.String(unsafe.SliceData(bigSlice), len(bigSlice))
    s := unsafe.Slice(unsafe.StringData(str), len(str))

    return s
}
```

```
goos: darwin
goarch: arm64
pkg: lection7
BenchmarkZeroCopyConverter
BenchmarkZeroCopyConverter-12       1000000000          0.3955 ns/op          0 B/op          0 allocs/op
PASS
```

DEMO

byte slice <-> string conversion



ПАКЕТ UNSAFE

```go
type T1 struct {  4 usages
    a int8
    b int64
    c int16
}

type T2 struct {  1 usage
    a int8
    c int16
    b int64
}

func main() {
    fmt.Println(unsafe.Sizeof(T1{})) // 24
    fmt.Println(unsafe.Sizeof(T2{})) // 16

    fmt.Println(unsafe.Alignof(T1{}.a)) // 1
    fmt.Println(unsafe.Alignof(T1{}.b)) // 8
    fmt.Println(unsafe.Alignof(T1{}.c)) // 2
}
```

1. **For a variable x of any type:** unsafe.Alignof(x) is at least 1

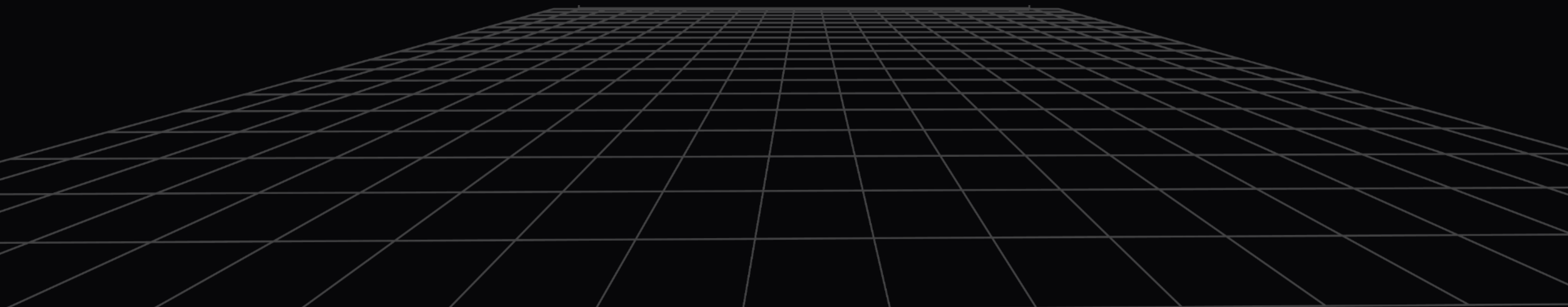
2. **For a variable x of struct type:** unsafe.Alignof(x) is the largest of all the values unsafe.Alignof(x.f) for each field f of x, but at least 1

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

DEMO

own unsafe slice

https://pkg.go.dev/unsafe



SLICE AND MAP PREALLOC

```go
func fastMatch(exampleData []string) [][]byte {  2 usages  Igor Walther
    result := make([][]byte, 0, len(exampleData))

    for i := 0; i < len(exampleData); i++ {
        data := unsafe.Slice(unsafe.StringData(exampleData[i]), len(exampleData[i]))

        if externalMatchFunction(data) {
            result = append(result, data)
        }
    }

    return result
}

func slowMatch(exampleData []string) [][]byte {  2 usages  Igor Walther
    result := make([][]byte, 0)

    for i := 0; i < len(exampleData); i++ {
        data := []byte(exampleData[i])

        if externalMatchFunction(data) {
            result = append(result, data)
        }
    }

    return result
}
```

STRING CONVERSION

```go
b.Run( name: "sprinf", func(b *testing.B) {
    b.ReportAllocs()

    for i := 0; i < b.N; i++ {
        fmt.Sprintf( format: "%d",  a...: 42)
    }
})
```

```go
b.Run( name: "strconv", func(b *testing.B) {
    b.ReportAllocs()

    for i := 0; i < b.N; i++ {
        strconv.FormatInt( i: 42,  base: 10)
    }
})
```

```
chmarkStrconv
chmarkStrconv/sprinf
chmarkStrconv/sprinf-12          38273593          30.28 ns/op        2 B/op          1 allocs/op
chmarkStrconv/strconv
chmarkStrconv/strconv-12         886768310          1.387 ns/op           0 B/op          0 allocs/op
S
```

STRING BUILDER

```go
b.Run( name: "stringBuilder", func(b *testing.B) {
    b.ReportAllocs()

    sb := &strings.Builder{}

    for i := 0; i < b.N; i++ {
        sb.WriteByte(byte(i))
    }

    result := sb.String()
    _ = result
})
```

```go
b.Run( name: "simple addition", func(b *testing.B) {
    b.ReportAllocs()

    var result string

    for i := 0; i < b.N; i++ {
        result += string(byte(i))
    }
})
```

```
goos: darwin
goarch: arm64
pkg: lection7
BenchmarkStringBuilder
BenchmarkStringBuilder/stringBuilder
BenchmarkStringBuilder/stringBuilder-12              933566685            1.154 ns/op           6 B/op          0 allocs/op
BenchmarkStringBuilder/simple_addition
BenchmarkStringBuilder/simple_addition-12             1000000         34085 ns/op      754003 B/op          2 allocs/op
PASS
```

LOOP ALLOCATION

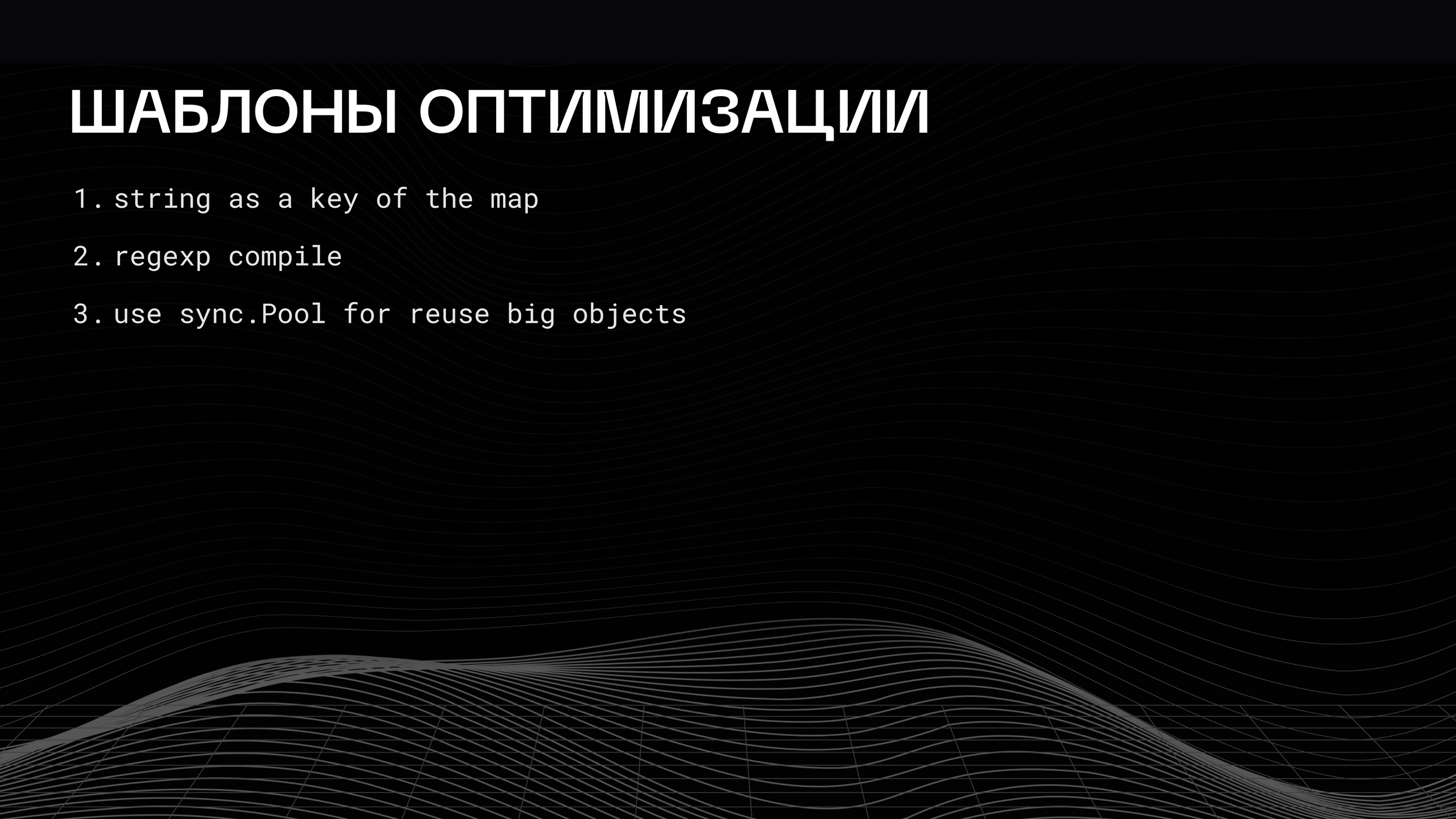
```go
b.Run( name: "loop allocation", func(b *testing.B) {
    b.ReportAllocs()

    b.StopTimer()
    sd, err := json.Marshal(TestData{
        a: "213132",
        b: 123,
        c: 123124,
    })
    require.NoError(b, err)
    b.StartTimer()

    for i := 0; i < b.N; i++ {
        d := Data{}

        err = json.Unmarshal(sd, &d)
        require.NoError(b, err)
    }
})
```

```go
b.Run( name: "outer loop allocation", func(b *testing.B) {
    b.ReportAllocs()

    b.StopTimer()
    sd, err := json.Marshal(TestData{
        a: "213132",
        b: 123,
        c: 123124,
    })
    require.NoError(b, err)
    b.StartTimer()

    d := Data{}
    for i := 0; i < b.N; i++ {
        err = json.Unmarshal(sd, &d)
        require.NoError(b, err)
    }
})
```

```
goos: darwin
goarch: arm64
pkg: lection7
BenchmarkLoopAllocation
BenchmarkLoopAllocation/loop_allocation
BenchmarkLoopAllocation/loop_allocation-12          5035930          220.2 ns/op       160 B/op          3 allocs/op
BenchmarkLoopAllocation/outer_loop_allocation
BenchmarkLoopAllocation/outer_loop_allocation-12    5920603          205.5 ns/op       152 B/op          2 allocs/op
PASS
```

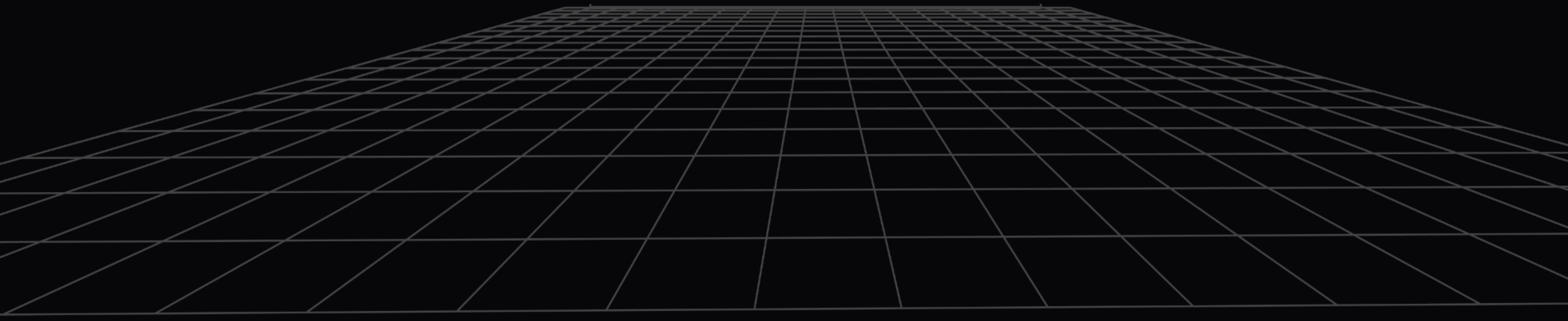
ШАБЛОНЫ ОПТИМИЗАЦИИ

1. string as a key of the map
2. regexp compile
3. use sync.Pool for reuse big objects



DEMO

Homework



ИТОГИ И ВЫВОДЫ

ЧТО БУДЕТ ДАЛЬШЕ