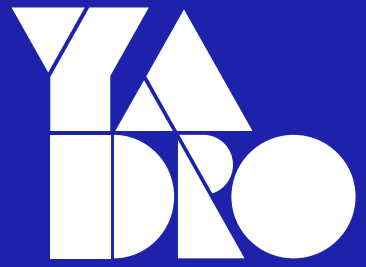


Типы данных под капотом – слайсы и как их готовить.

Влад Белогрудов, YADRO

Влад Белогрудов

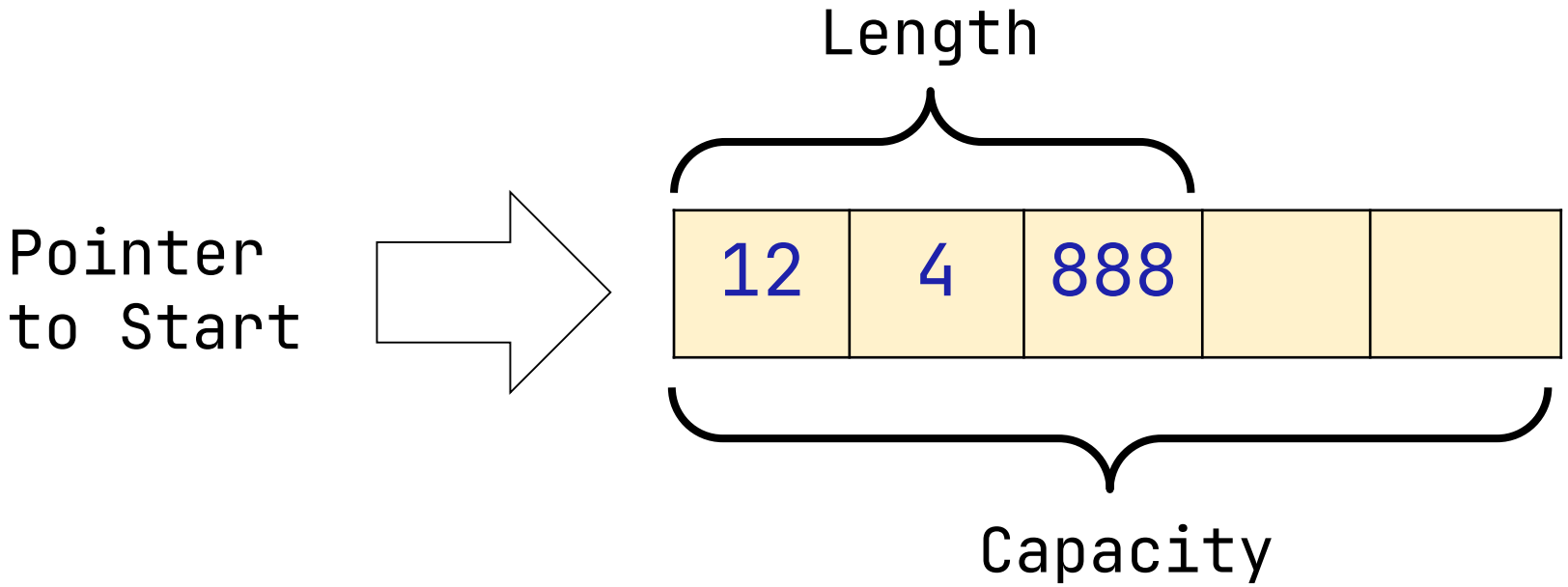
Инженер, YADRO

1998 Первая программа на C в UNIX
1999 Роботы, телеком, поисковики
~
2006 Инфраструктура, облака
2022 YADRO, разработка «спутников» и ИС

О чем?

1. Что представляют из себя
   - slice
   - string
   - sync.Pool
2. Создание, копирование, изменение
3. Полезные трюки и подводные камни
4. Вопросы производительности, альтернативы

Slice: строение



Slice: код

https://go.dev/src/runtime/slice.go

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

Slice: пустой

```
// var myslice []int -> nil
myslice := []int{}
```

```
array = nil
len   = 0
cap   = 0
```

Slice: первый append()

```
myslice = append(myslice, 11)
```

```
array -> [11]
len   = 1
cap   = 1
```

Slice: второй append()

```
myslice = append(myslice, 22)
```

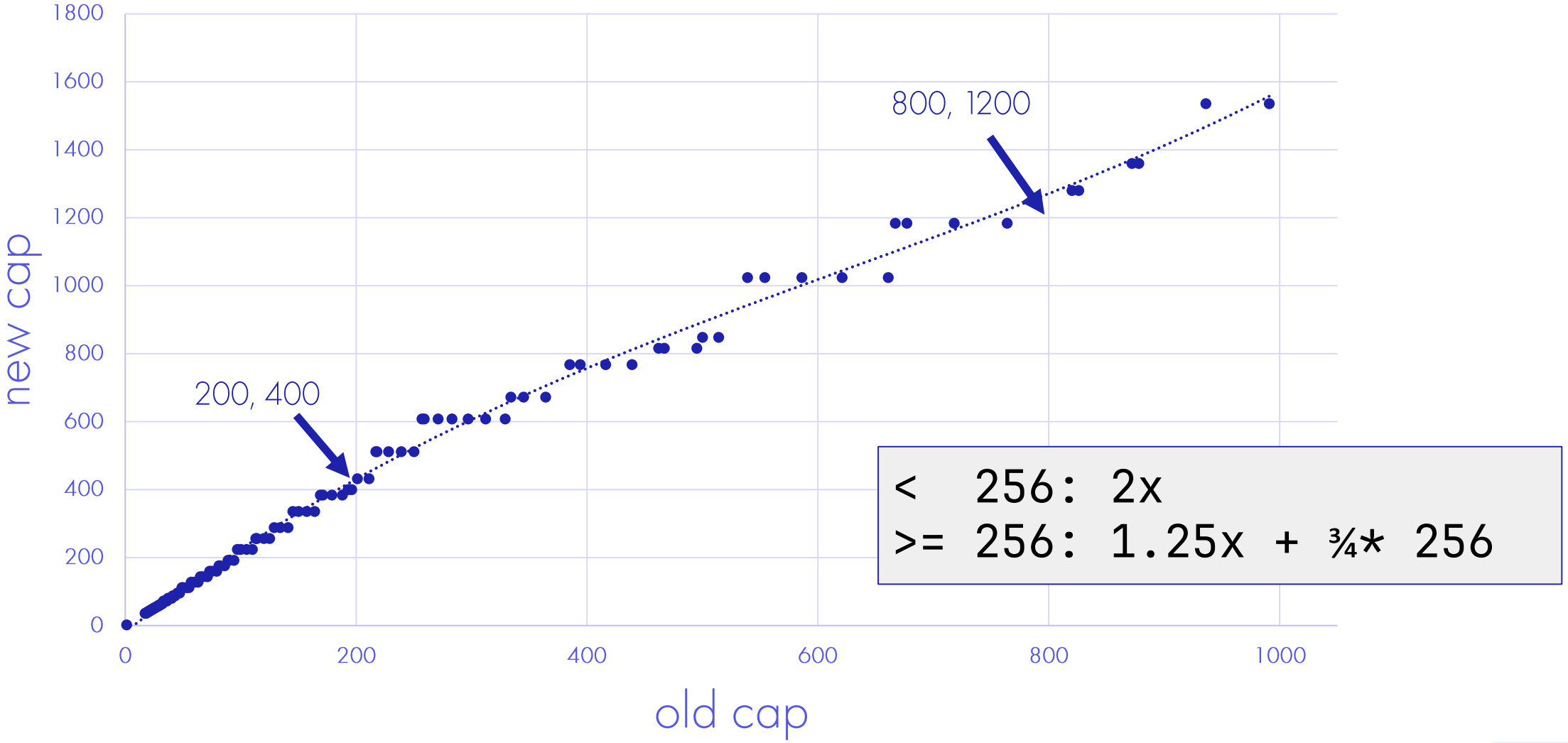
```
array -> [11]
  копирование
       -> [11, 22]
len   = 2
cap   = 2
```

Slice: третий append()

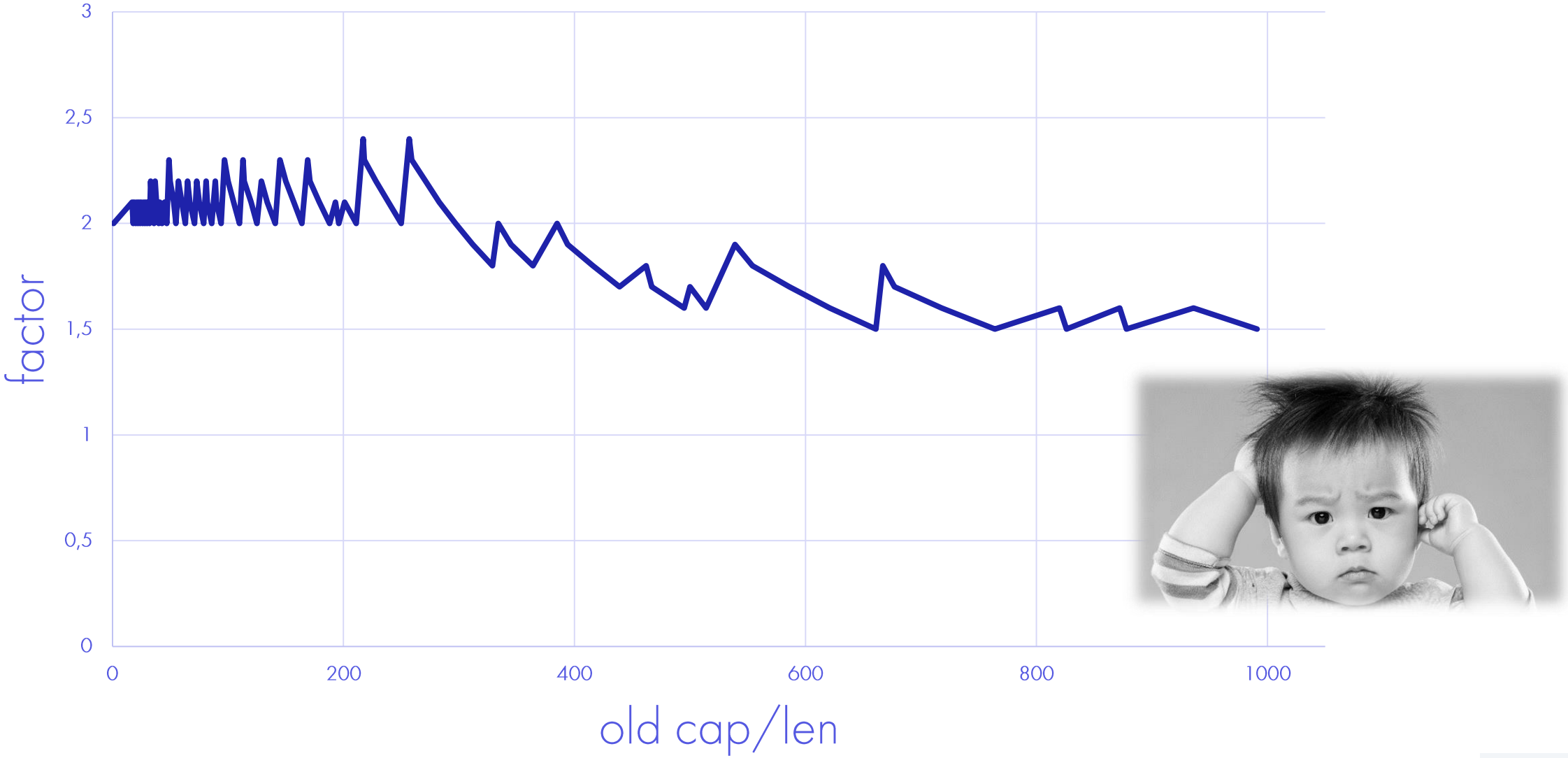
```
myslice = append(myslice, 33)
```

```
array -> [11, 22]
  копирование
       -> [11, 22, 33,  _]
len    = 3
cap    = 4
```

Slice: как быстро растет capacity?



Slice: как быстро растет capacity?



Slice: формула не верна?

```
a := make([]int, 17)
a = append(a, 1)
fmt.Println(len(a), cap(a))   // 18, 36
```

Slice: формула не верна?

```
a := make([]int, 17)
a = append(a, 1)
fmt.Println(len(a), cap(a))   // 18, 36

b := make([]int, 18)
b = append(b, 1)
fmt.Println(len(b), cap(b))   // 19, 36
```

Slice: рост слайса в заданных рамках (object round up)

https://go.dev/src/runtime/sizeclasses.go

```
// class  bytes/obj  ...  objects  ...
    1           8              1024
    2          16               512
    3          24               341
    4          32               256
   ...
   18         256                32
   19         288                28
   20         320                25
```

```
8 * 34 = 272
288 / 8 = 36
```

Slice: можно ли использовать настоящий capacity?

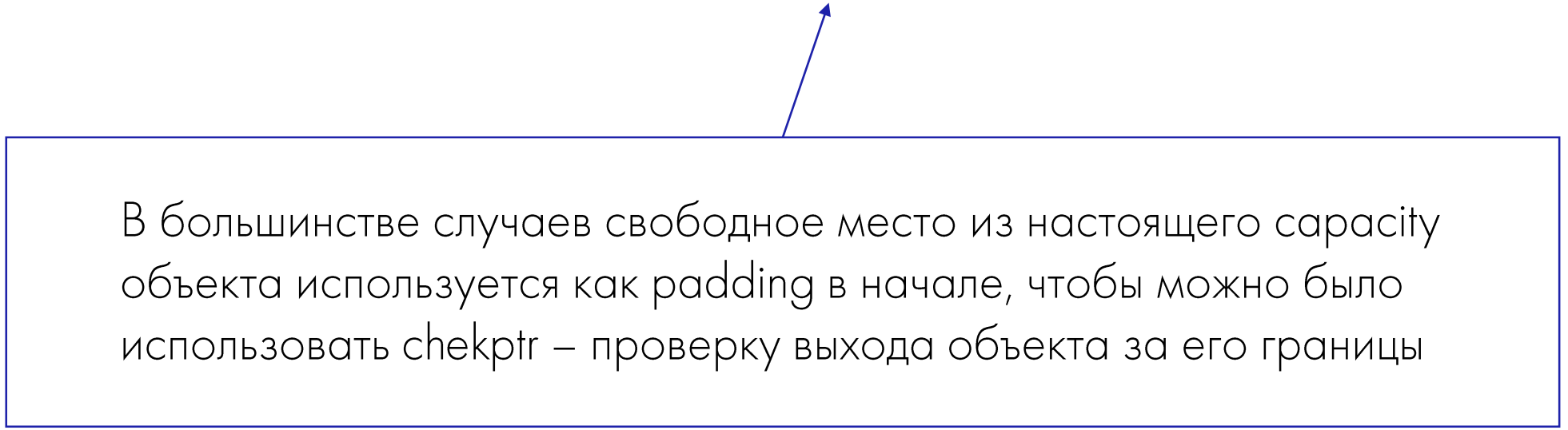
1. После append() - иногда
   - результирующий int слайс из17 + append 1 будет 36

Slice: можно ли использовать настоящий capacity?

1. После append() - иногда
   - результирующий int слайс из17 + append 1 будет 36
   - изначальный int слайс из 34 + append 1 – нет (new cap 72)

Slice: можно ли использовать настоящий capacity?

1. После append() - иногда
   - результирующий int слайс из17 + append 1 будет 36
   - изначальный int слайс из 34 + append 1 – нет (new cap 72)



Slice: можно ли использовать настоящий capacity?

1. После append() - иногда
   - результирующий int слайс из17 + append 1 будет 36
   - изначальный int слайс из 34 + append 1 – нет (new cap 72)
2. make() & co - нет, для определенности ☺

Slice: конкатенация (меньшего к большему)

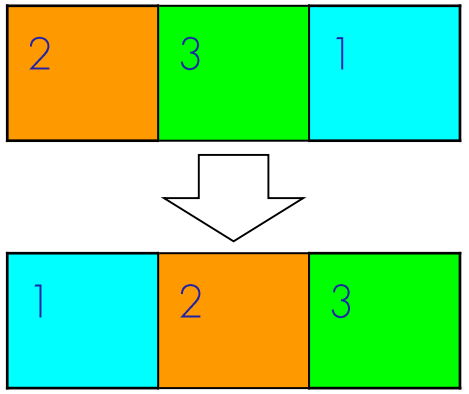
```
a := make([]int, 17)
b := make([]int, 5)

// new len < 2 * old cap
c = append(a, b...)
fmt.Println(len(c), cap(c))   // 22, 36
```

Slice: конкатенация (большего к меньшему)

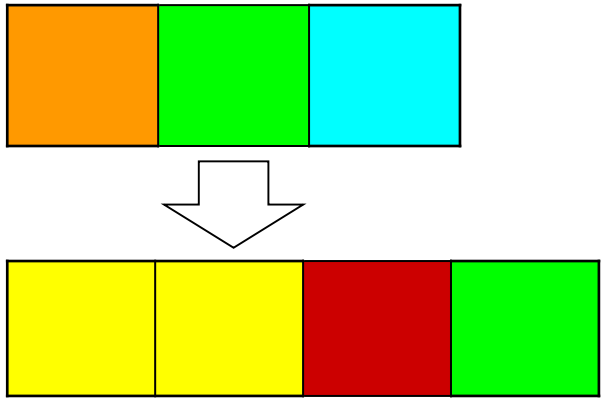
```
a := make([]int, 17)
b := make([]int, 5)

// else new cap = new len
d := append(b, a...)
fmt.Println(len(d), cap(d))    // 22, 22
```

Slice: передача в функции

- Передача по значению:
  - Можем менять значения src
  - Не можем len & cap
  - Подходит для сортировки



- Передача по указателю:
  - nanosecs быстрее
  - Можем полностью поменять все



Slice: передача в функцию по значению

```
var myslice []int
...
modify(myslice)
```

```
func modify(s []int) {
```

```
a1   -> [2, 1, 3, _]
len1 = 3
cap1 = 4
```

```
<-  a2
    len2 = 3
    cap2 = 4
```

Slice: передача в функцию по значению, sort()

```
var myslice []int
...
modify(myslice)
```

```
func modify(s []int) {
    sort.Ints(s)
```

```
a1   -> [1, 2, 3, _]
len1 = 3
cap1 = 4
```

```
<-  a2
    len2 = 3
    cap2 = 4
```

Slice: передача в функцию по значению, append()

```
var myslice []int
...
modify(myslice)
```

```
func modify(s []int) {
  ...
  s = append(s, 4)
```

```
a1   -> [1, 2, 3, 4]
len1 = 3
cap1 = 4
```

```
<-  a2
    len2 = 4
    cap2 = 4
```

Slice: передача в функцию по значению, append()

```
var myslice []int
...
modify(myslice)
```

```
func modify(s []int) {
  ...
  s = append(s, 5)
```

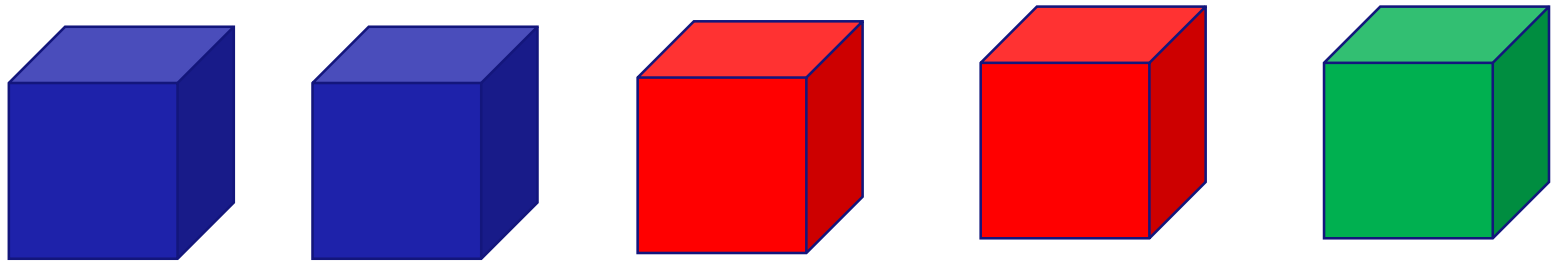
```
a1   -> [1, 2, 3, 4]
len1 = 3
cap1 = 4
```

~~<- a2~~

копирование в новый массив

```
a2 -> [1, 2, 3, 4, 5, _, _, _]
len2 = 5
cap2 = 8
```

Slice: колонка юмора – какая сортировка в Go?



Slice: колонка юмора – какая сортировка в Go?

```
func pdqsort(data Interface, a, b, limit int) {
    const maxInsertion = 12
...
    for {
        length := b - a

        if length <= maxInsertion {
            insertionSort(data, a, b)
            return
        }
```

https://github.com/golang/go/blob/master/src/sort/zsortinterface.go

Slice: колонка юмора – какая сортировка в Go?

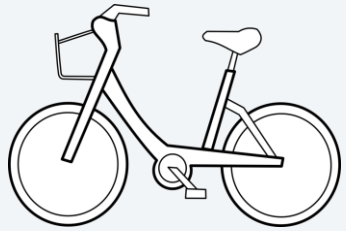
```
func pdqsort(data Interface, a, b, limit int) {
    const maxInsertion = 12
...
    for {
        length := b - a

        if length <= maxInsertion {
            insertionSort(data, a, b)
            return
        }
```

Используются несколько алгоритмов – quick sort, heap sort, insertion sort

Slice: велосипеды и Go 1.21

Min(), Max(), Sort(), Reverse(), Contains(), Delete(), Clone()... на дженериках!

```
for i := range myslice {
    if myslice[i] == searchTerm {
        return true
    }
}
return false
...

// one-liner!
if slices.Contains(myslice, searchTerm) {
```

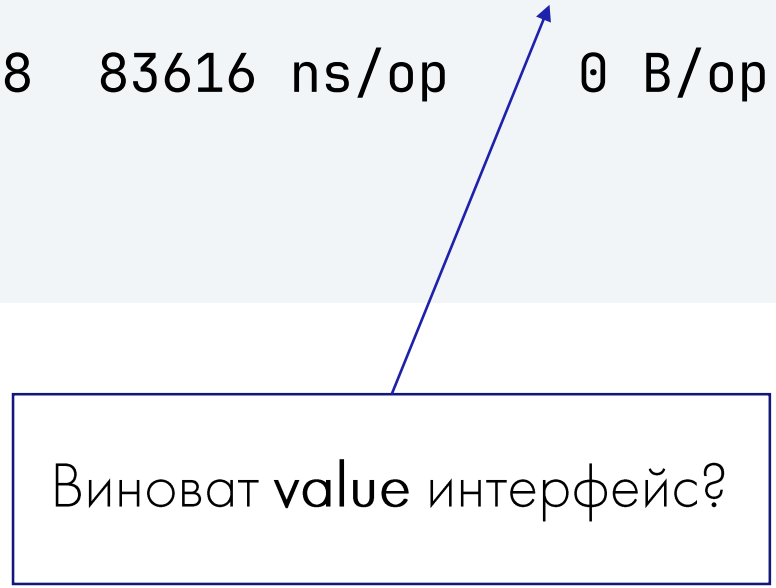


Slice: interface sort vs generics sort

```
words := []string{
      "hello", "double", "bye", "triple",
      "yes", "double", "triple", "triple“,
}

words2 := slices.Clone(words)

sort.Strings(words)
fmt.Println(words)

slices.Sort(words2)
fmt.Println(words2)
```

Slice: sorting champions

```
Uuid strings, len(words) = 1000, 10000 unique slices per experiment


BenchmarkSortInterface-8 92410 ns/op   24 B/op         1 allocs/op
BenchmarkSortGenerics-8  83616 ns/op    0 B/op         0 allocs/op
```

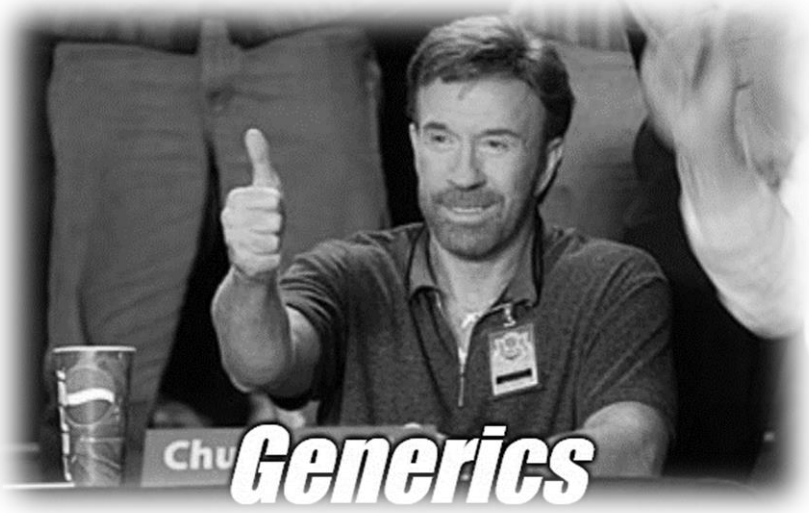


Slice: sort interface без аллокаций (old school)

```
type StrSlice []string

func (x *StrSlice) Len() int { return len(*x) }

func (x *StrSlice) Less(i, j int) bool { ... }

func (x *StrSlice) Swap(i, j int) { ... }


sort.Strings((*StrSlice)(&words))
```

Slice: sorting champions, round 2

```
BenchmarkSortInterface-8     92410 ns/op   24 B/op    1 allocs/op
BenchmarkSortGenerics-8      83616 ns/op    0 B/op    0 allocs/op
BenchmarkSortInterface2-8    91850 ns/op    0 B/op    0 allocs/op
```



Slice: Еще ускорение?

- Добавляем capacity, если знаем порядок
- container/list для быстрых вставок и удалений
- Используем *element, если element большой и много манипуляций
- sync.Pool
- Удаление лишнего кода ;)
- ?

String: отличия от slice

- read-only
- только len(), нет cap()
- string copy != slice copy
- len(“привет”) == len(internal byte buffer) == 12
- range / utf8.RuneCount() для обхода и подсчета символов

String: конвертации в байты и обратно

- Одна из самых частых операций в Go

```
bytes := []byte(myString)
str := string(myBytes)
```

String: конвертации в байты и обратно

- Одна из самых частых операций в Go

```
bytes := []byte(myString)
str := string(myBytes)
```

- Приводит к копии, если не использовать `unsafe.Pointer()`, пример k8s:
  - вредный совет, если не следить за памятью под капотом
  - https://pkg.go.dev/unsafe#String & Co для хаков

String: конвертации в байты и обратно

- Одна из самых частых операций в Go

```
bytes := []byte(myString)
str := string(myBytes)
```

- Приводит к копии, если не использовать `unsafe.Pointer()`, пример k8s:
  - вредный совет, если не следить за памятью под капотом
  - https://pkg.go.dev/unsafe#String & Co для хаков
- Компилятор сам оптимизирует в особых (безопасных) случаях

String: оптимизация перевода в байты и обратно

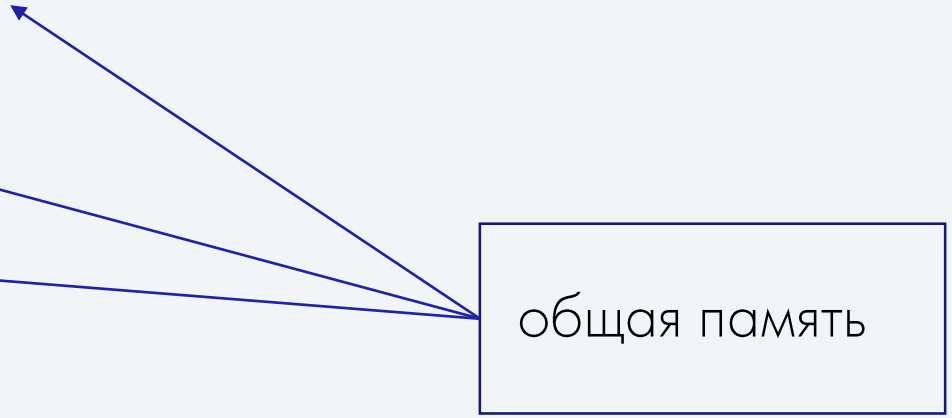
```
str := “hello”

for i, b := range([]byte(str)) {
   ...
}
```

String: оптимизация перевода в байты и обратно

```
str := “hello”
...

key = []byte{0x41, 0x42}

if str < string(key) {
    ...
}
```

String: оптимизации копирования и слайсинга

```
source = “hello world”

dest1 = source

dest2 = source[2:5]
```



String: конкатенации

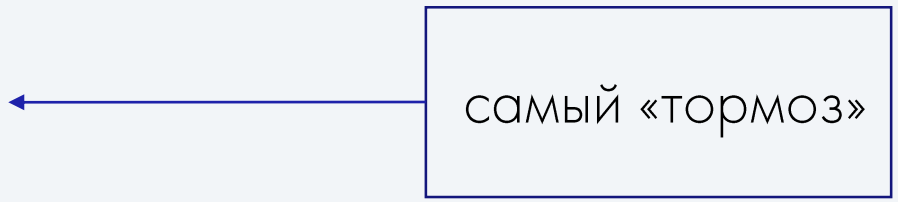
```
var words = []string{
    "hello", "double", "bye", "triple"}

var sum string
for i := range words {
    sum += words[i]
}
```

String: конкатенации

```
var words = []string{
    "hello", "double", "bye", "tri

var sum string
for i := range words {
    sum += words[i]
}
```

String: конкатенации

```
var words = []string{
    "hello", "double", "bye", "triple"}

var sum string
for i := range words {
    sum += words[i]
}
```



String: конкатенации

```
var words = []string{
    "hello", "double", "bye", "triple"}

var sum string
for i := range words {
    sum += words[i]
}

sum = words[0] + words[1] + words[2] + words[3]
```

String: конкатенации

```
var words = []string{
    "hello", "double", "bye", "triple"}

var sum string
for i := range words {
    sum += words[i]
}

sum = words[0] + words[1] + words[2] + words[3]

sum = strings.Join(words, “”) // clever?!
```

String: оптимизация конкатенации, замена кода

```
func concatstrings(buf *tmpBuf, a []string) string ..
```

https://go.dev/src/runtime/string.go

String: оптимизация конкатенации, замена кода

```
func concatstrings(buf *tmpBuf, a []string) string ..

func concatstring2(buf *tmpBuf, a0, a1 string) string {
    return concatstrings(buf, []string{a0, a1})
}
```

https://go.dev/src/runtime/string.go

String: оптимизация конкатенации, замена кода

```
func concatstrings(buf *tmpBuf, a []string) string ..

func concatstring2(buf *tmpBuf, a0, a1 string) string {
    return concatstrings(buf, []string{a0, a1})
}

func concatstring3(...

func concatstring4(...

func concatstring5(...
```

https://go.dev/src/runtime/string.go

String: оптимизация конкатенации, парсер

```
func walkAddString(n *ir.AddStringExpr, init *ir.Nodes) ir.Node {
...
  if c <= 5 {
    // small num of strings use direct runtime helpers.
     fn = fmt.Sprintf("concatstring%d", c)
   } else {
     // large num of strings go to runtime as a slice.
     fn = "concatstrings"
     t := types.NewSlice(types.Types[types.TSTRING])
```

https://github.com/golang/go/../walk/expr.go

String: оптимизация конкатенации, парсер

```
func walkAddString(n *ir.AddStringExpr, init *ir.Nodes) ir.Node {
...
  if c <= 5 {
    // small num of strings use direct runtime helpers.
     fn = fmt.Sprintf("concatstring%d", c)
   } else {
     // large num of strings go to runtime as a slice.
     fn = "concatstrings"
     t := types.NewSlice(types.Types[types.TSTRING])
```

https://github.com/golang/go/../walk/expr.go

Конкатенация строк с “+” в Го очень эффективна на любых размерах

String: bytes.Buffer

```
type Buffer struct {
    buf      []byte
    off      int
    lastRead readOp
}
```

- Buffer растет динамически, есть аллокации и копирования.
- Пустой стартует с 64 байт, скорость роста 2x.

String: bytes.Buffer

```
type Buffer struct {
    buf      []byte
    off      int
    lastRead readOp
}

var buf bytes.Buffer ...
func NewBuffer(b []byte) ...
```

- Buffer растет динамически, есть аллокации и копирования.
- Пустой стартует с 64 байт, скорость роста 2x.

String: bytes.Buffer

```
type Buffer struct {
    buf      []byte
    off      int
    lastRead readOp
}

var buf bytes.Buffer ...
func NewBuffer(b []byte) ...

func (b *Buffer) Write*(...)
func (b *Buffer) Read*(...)
func (b *Buffer) Reset()
```

- Buffer растет динамически, есть аллокации и копирования.
- Пустой стартует с 64 байт, скорость роста 2x.

String: конкатенация с bytes.Buffer

```
func ConcatBuffer(words []string) string {

    b := bytes.Buffer{}

    for i := range words {
        b.WriteString(words[i])
    }
    return b.String()
}
```

String: конкатенация с strings.Builder

```
func ConcatBuilder(words []string) string {

    var b strings.Builder

    for i := range words {
        b.WriteString(words[i])
    }
    return b.String()
}
```

String: Builder - альтернатива bytes.Buffer?

- Почти ... `[]byte` под капотом и там и тут.

String: Builder - альтернатива bytes.Buffer?

- Почти ... `[]byte` под капотом и там и тут.
- Buffer растет чуть быстрее Builder - меньше аллокаций.

String: Builder - альтернатива bytes.Buffer?

- Почти ... `[]byte` под капотом и там и тут.
- Buffer растет чуть быстрее Builder - меньше аллокаций.
- Buffer на небольших строчках и количестве быстрее (64 -> 2x).

String: Builder - альтернатива bytes.Buffer?

- Почти ... `[]byte` под капотом и там и тут.
- Buffer растет чуть быстрее Builder - меньше аллокаций.
- Buffer на небольших строчках и количестве быстрее (64 -> 2x).
- В Builder нельзя подсунуть начальный буфер, чтобы не начинать внутренний слайс с нуля.

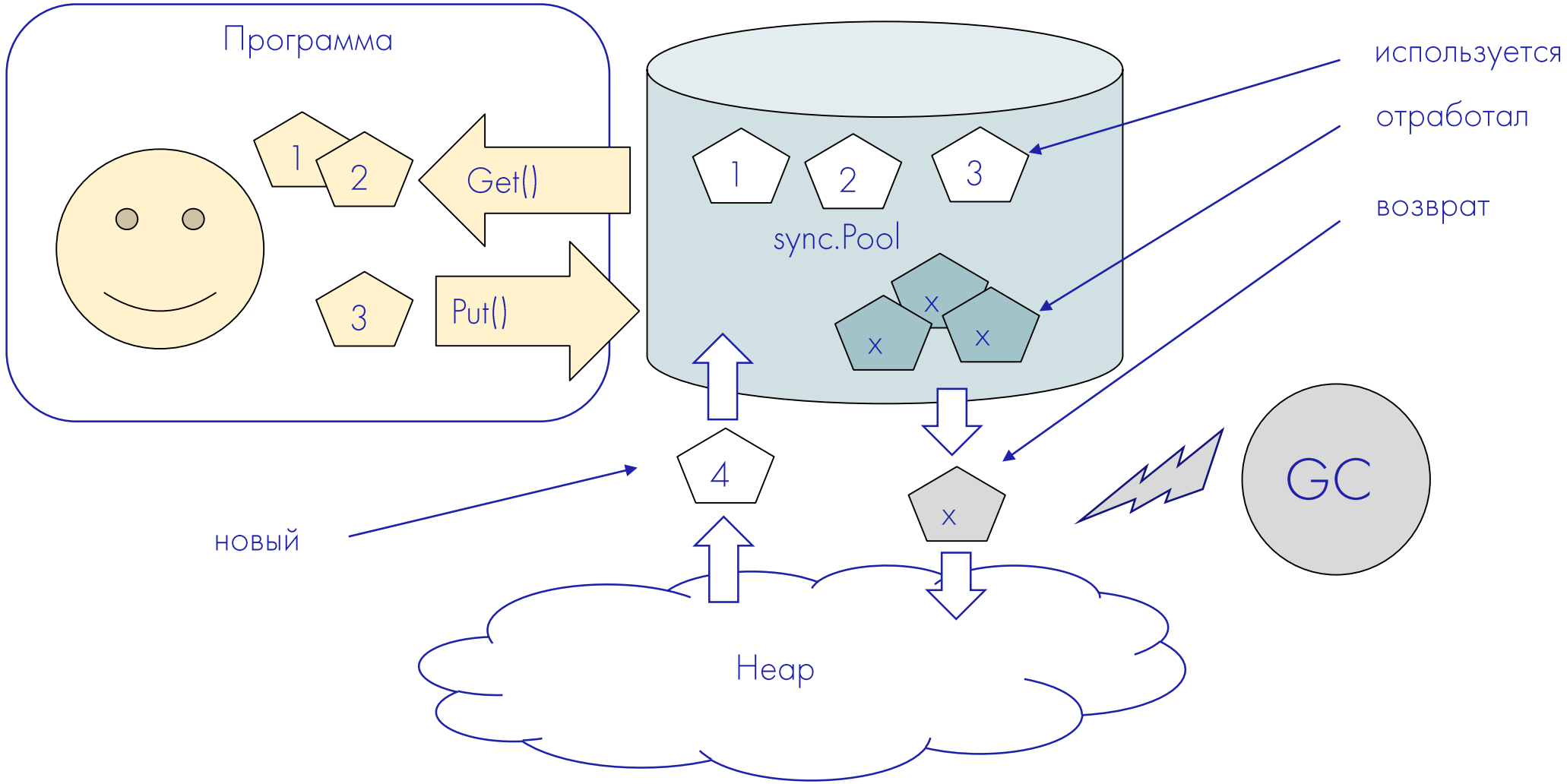
String: Builder - альтернатива bytes.Buffer?

- Почти ... `[]byte` под капотом и там и тут.
- Buffer растет чуть быстрее Builder - меньше аллокаций.
- Buffer на небольших строчках и количестве быстрее (64 -> 2x).
- В Builder нельзя подсунуть начальный буфер, чтобы не начинать внутренний слайс с нуля.
- Builder эффективнее отдает результирующую строчку
  - нет лишнего копирования (да, unsafe.Ptr)

String: Builder - альтернатива bytes.Buffer?

- Почти ... `[]byte` под капотом и там и тут.
- Buffer растет чуть быстрее Builder - меньше аллокаций.
- Buffer на небольших строчках и количестве быстрее (64 -> 2x).
- В Builder нельзя подсунуть начальный буфер, чтобы не начинать внутренний слайс с нуля.
- Builder эффективнее отдает результирующую строчку
  - нет лишнего копирования (да, unsafe.String)
- Builder.Reset не сохраняет внутренний слайс
  - Безопасно для построенных строк
  - при использовании с sync.Pool не дает никакого выигрыша

sync.Pool: устройство



sync.Pool: что можно ускорить?

```
func ConcatSimple(words []string) string {

    var accumulator string
    for i := range words {
        accumulator += words[i]
    }
    return accumulator
}
```

sync.Pool: делай раз!

```
var pool = sync.Pool{
    New: func() any {
        return &bytes.Buffer{}
    }
}
```

sync.Pool: делай два!

```
func ConcatPool(words []string) string {

    // достаем объект, новый или переиспользуемый
    b := pool.Get().(*bytes.Buffer)
...
```

sync.Pool: делай два!

```
func ConcatPool(words []string) string {

    // достаем объект, новый или переиспользуемый
    b := pool.Get().(*bytes.Buffer)

    // не забываем положить обратно после
    defer pool.Put(b)
...
```

sync.Pool: делай два!

```
func ConcatPool(words []string) string {

    // достаем объект, новый или переиспользуемый
    b := pool.Get().(*bytes.Buffer)

    // не забываем положить обратно после
    defer pool.Put(b)

    // на всякий случай чистим (если переиспользуемый)
    b.Reset()
...
```

sync.Pool: делай три!

```
func ConcatPool(words []string) string {
    ...

    // кладем слова в буфер
    for i := range words {
        b.WriteString(words[i])
    }
    //создаем результирующую строчку
    return b.String()
}
```

sync.Pool: где взял, там и отдай

Правила хорошего тона:

- Очищай объект из пула
- Не забывай отдавать память в пул
- Не используй память после отдачи

Альтернативно:

- Передаем во вне весь объект, полученный из pool.Get() и потом кладем его оттуда обратно в пул

sync.Pool: гонки без правил

```
func Read(input []string) []byte {

    buffer := pool.Get().(*bytes.Buffer)
    defer pool.Put(buffer)
    buffer.Reset()

    for i := range input {
        buffer.WriteString(input[i])
    }

    return buffer.Bytes()   /// не-е-ет!
}
```

https://go.dev/play/p/9f2CM3V51hZ

String: конкатенация – 8 слов, суммарно 65B

```
BenchmarkConcatSimple-8          222.3 ns/op   256 B/op    7 allocs/op
BenchmarkConcatBuilder-8         160.6 ns/op   248 B/op    5 allocs/op
BenchmarkConcatBuffer-8          140.3 ns/op   272 B/op    3 allocs/op
BenchmarkConcatSyncPool-8         66.65 ns/op   80 B/op    1 allocs/op
BenchmarkConcatBufInitSl1k-8      65.42 ns/op   80 B/op    1 allocs/op
BenchmarkConcatStringsJoin-8      64.82 ns/op   80 B/op    1 allocs/op
BenchmarkConcatLiteral-8          59.54 ns/op   80 B/op    1 allocs/op
```

String: конкатенация – “8” слов, суммарно 6500B

```
BenchmarkConcatBufInitSl1k-8   2395 ns/op    20864 B/op     4 allocs/op
BenchmarkConcatBuffer-8        2264 ns/op    25088 B/op     6 allocs/op
BenchmarkConcatBuilder-8       2214 ns/op    22528 B/op     7 allocs/op
BenchmarkConcatSyncPool-8       553.0 ns/op   6533 B/op     1 allocs/op
BenchmarkConcatStringsJoin-8    507.3 ns/op   6528 B/op     1 allocs/op
BenchmarkConcatLiteral-8        496.5 ns/op   6528 B/op     1 allocs/op
```

Большие строчки в малом количестве лучше не соединять через Builder/Buffer - слишком много аллокаций и копирования

String: конкатенация – 800 слов, суммарно 6500B

```
BenchmarkConcatBuffer-8          6481 ns/op    22848 B/op     9 allocs/op
BenchmarkConcatBufInitSl1k-8     5929 ns/op    20864 B/op     4 allocs/op
BenchmarkConcatBuilder-8         5747 ns/op    24824 B/op    14 allocs/op
BenchmarkConcatStringsJoin-8     5703 ns/op     6528 B/op     1 allocs/op
BenchmarkConcatSyncPool-8        4190 ns/op     6529 B/op     1 allocs/op
```

Большое количество элементов уравнивает шансы Builder/Buffer с остальными решениями. Builder более экономен с ростом элементов.

String: конкатенация – 8000 слов, суммарно 65000B

```
BenchmarkConcatBuilder-8       58583 ns/op  285434 B/op  22 allocs/op
BenchmarkConcatBuffer-8        58457 ns/op 196544 B/op   12 allocs/op
BenchmarkConcatBufInitSl1k-8   56694 ns/op  194560 B/op   7 allocs/op
BenchmarkConcatStringsJoin-8   55969 ns/op   65536 B/op   1 allocs/op
BenchmarkConcatSyncPool-8      44972 ns/op   65566 B/op   1 allocs/op
```

Две вещи, на которые можно смотреть вечно

Две вещи, на которые можно смотреть вечно

Программисты, использующие слайсы в своих программах...

Москва,
ул. Рочдельская, 15, стр. 13
+7 800 777-06-11

yadro.com

@VLAD_BELOGRUDOV

Дополнительный материал

Slice: создание

```
// construct        and          print out
var a1 []int                     // []
a2 := []int{1, 2, 3}             // [1 2 3]
a3 := make([]int, 4)             // [0 0 0 0]
a4 := make([]int, 0, 4)          // []
```

Slice: заполняем на перегонки. Просто append()

```
var s1 []int

for i := 0; i < 1024; i++ {
    s1 = append(s1, i)
}
```

Slice: заполняем на перегонки. Capacity + append()

```
s2 := make([]int, 0, 1024)

for i := 0; i < 1024; i++ {
    s2 = append(s2, i)
}
```

Slice: заполняем на перегонки. По индексу

```
s3 := make([]int, 1024)

for i := 0; i < 1024; i++ {
    s3[i] = i
}
```

Slice: заполняем на перегонки. Результаты

```
cpu: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz

BenchmarkNewIntSliceSimple-8   3039 ns/op   25208 B/op   12 allocs/op

BenchmarkNewSlicePrealloc-8    1408 ns/op    8192 B/op    1 allocs/op

BenchmarkNewSliceZeroed-8      1371 ns/op    8192 B/op    1 allocs/op
```

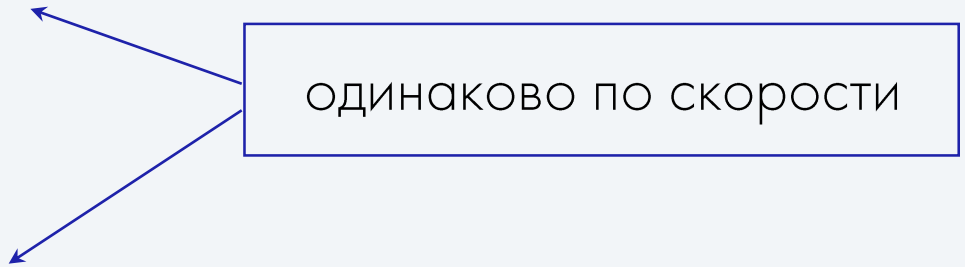
Slice: заполняем на перегонки. Результаты

```
cpu: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz

BenchmarkNewIntSliceSimple-8  3039 ns/op   25208 B/op   12 allocs/op

BenchmarkNewSlicePrealloc-8   1408 ns/op   8192 B/op     1 allocs/op

BenchmarkNewSliceZeroed-8     1371 ns/op   8192 B/op     1 allocs/op
```

Добавление элементов в слайс через индекс или append()
– дело вкуса, контекста, ...

Slice: копирование

```
dst := src          // нет!
dst := src[:]       // нет-нет!

dst := make([]int, len(src))
copy(dst, src)

var dst []uint64
dst = append(dst, src...)
```



Slice: специальный велосипед для unique list

```
numbers := []int{1, 2, 3, 2, 3, 3}

nset := make(map[int]void)
for _, n := range numbers {
    nset[n] = void{}
}
i := 0
for n := range nset {
      numbers[i] = n
      i++
}
numbers = numbers[:i]  // unsorted, inplace …
```

Не любите `struct{}{}`?

```
type void struct{}
make(map[int]void)
```

Slice: замена велосипеда

```
// UNIX Shell:
// > cat words.txt | sort | uniq


numbers := []int{1, 2, 3, 2, 3, 3}

slices.Sort(numbers)
numbers = slices.Compact(numbers)
```

Если сортировка не нужна, старый способ быстрее

Slice interview questions: вставка

```
a := []int{1, 2, 3, 4}

a = append(a[:3], a[2:]...)
a[2] = 99

fmt.Println(a)                    // [1 2 99 3 4]
```

Slice interview questions: удаление

```
a := []int{1, 2, 3, 4}

a = append(a[:2], a[3:]...)

fmt.Println(a)                    // [1 2 4]

// есть более разумный вариант, если не важна очередность
```

Slice interview questions: сдвиги

```
a := []int{1, 2, 3, 4}

copy(a[1:], a[:3])              // сдвиг влево
fmt.Println(a)                  // [1 1 2 3]

copy(a[:3], a[1:])              // сдвиг вправо
fmt.Println(a)                  // [1 2 3 3]
```

тест на внимательность ;)

Slice: проверка усвоенного

Почему присваивается результат append(myslice, 1) обратно в myslice?

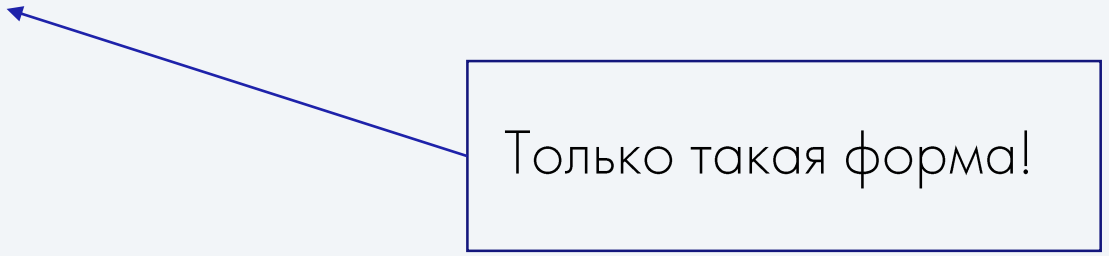
```
myslice = append(myslice, 1)
```

Варианты ответов:

1. Так принято, синтаксис Go
2. Может меняться размер внутреннего массива (length)
3. Меняются ячейки внутреннего массива (capacity)
4. Не увидим изменений снаружи функции append после вызова

String: оптимизация перевода в байты и обратно

```
m := map[string]int{
    “one”: 1,
    “two”: 2,
}

key = []byte{0x41, 0x42}
...
_, ok := m[string(key)]
```



UTF-8: слайсинг по-простому

Проблемы

- Не знаем, сколько символов и их “ширину”
- [n:m] работает с байтами

```
mir := string( ( []rune("привет мир!") )[7:] )

fmt.Println(mir) // мир!
```

UTF-8: слайсинг из почти стандартной библиотеки

```
import "golang.org/x/exp/utf8string"

...

    s := utf8string.NewString("привет мир!")

    mir = s.Slice(7, s.RuneCount())
```

`utf8string` работает с байтами без лишних (де)-кодирований, аллокаций и копирования

utf8string: производительность

```
BenchmarkSliceUtf8Simple-8        92.25 ns/op    16 B/op    1 allocs/op
BenchmarkSliceUtf8Lib-8           64.61 ns/op     0 B/op    0 allocs/op
BenchmarkSliceUtf8Simple10x-8   1132.00 ns/op   656 B/op    2 allocs/op
BenchmarkSliceUtf8Lib10x-8       368.80 ns/op     0 B/op    0 allocs/op
```

Библиотека дает существенный прирост производительности даже на небольших строках