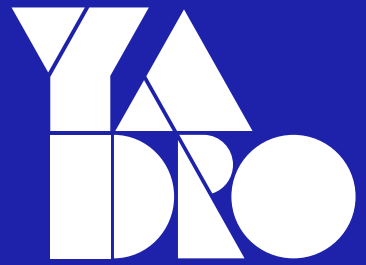


Типы данных под капотом – быстро ориентируемся в Go maps!

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 ИАС

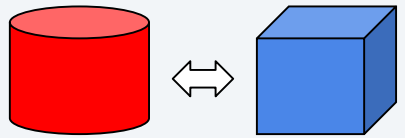
О чем?

1. Типы хеш-таблиц
2. Тип map в Go:
   - Внутреннее устройство
   - Вычисление хеша
3. Создание, копирование, изменение
4. Полезные трюки и подводные камни
5. Вопросы производительности

Небольшая проблема

- Имеем N структур - диски, ... с UUID
- N не очень большое, скажем < 20

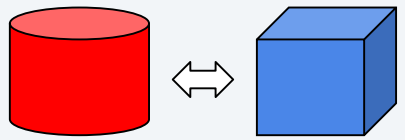
```
type Disk struct {
     UUID       string
     Type       string
     Model      string
     Vendor     string
     Size       uint64
}
```



Небольшая проблема

- Можем хранить их в
  - ✓ slice размера N

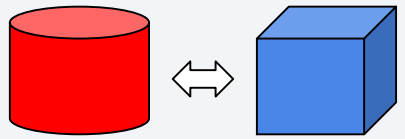
```
// somewhere: disks := make([]Disk, 0, N)

// find disk by uuid
for _, d := range disks {
    if d.UUID == uuid {
        return d, true
    }
}
```



Небольшая проблема

- Можем хранить их в
  - ✓ map размера N

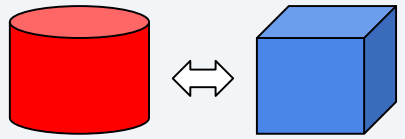
```
// somewhere: disks := make(map[string]Disk, N)

// find disk by uuid
d, ok := disks[uuid]
```



Небольшая проблема

- Можем хранить их в
  - ✓ map размера 128

```
// somewhere: disks := make(map[string]Disk, 128)

// find disk by uuid
d, ok := disks[uuid]
```

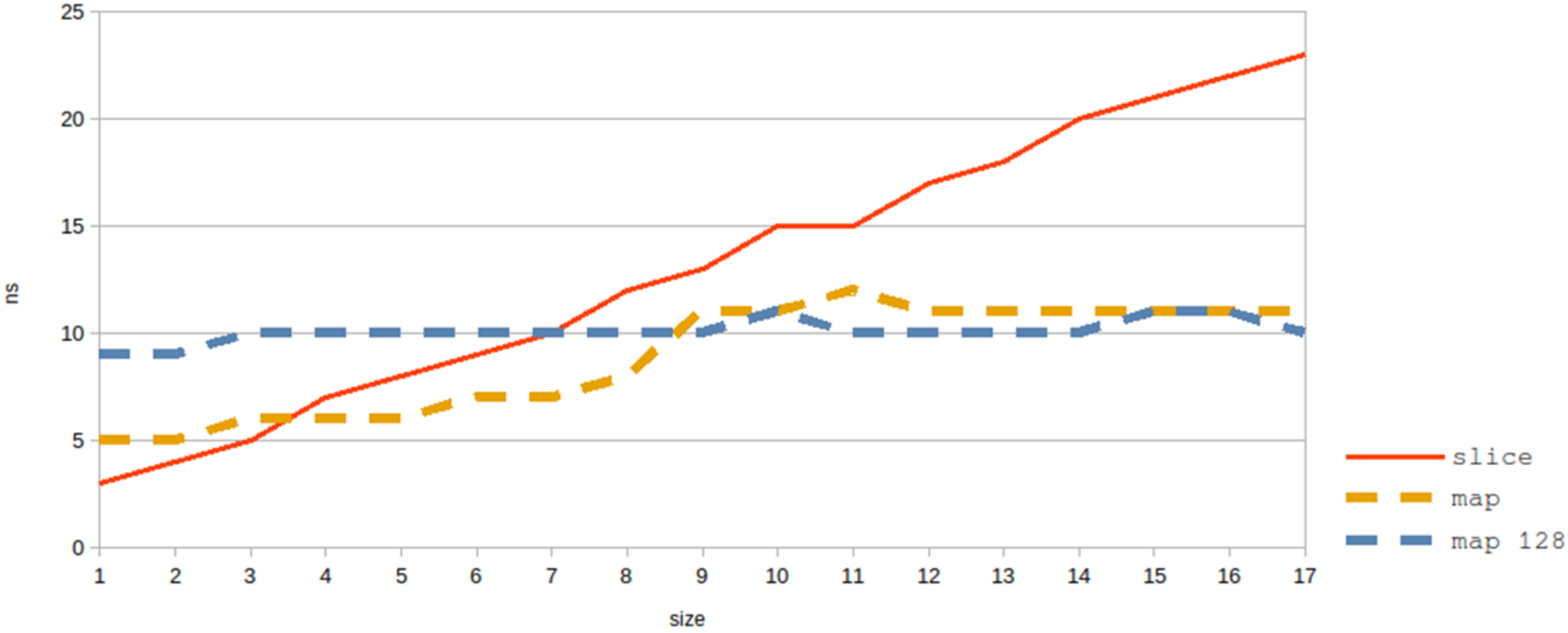


Небольшая проблема

- ✓ slice размера N
- ✓ map размера N
- ✓ map размера 128

Что быстрее для получения элемента по ключу?

Слайсы наиболее предсказуемы в O(n)



Выводы

1. slice очень даже неплох на небольших данных
    - а если еще и отсортировать ...

Выводы

1. slice очень даже неплох на небольших данных
   - а если еще и отсортировать ...
2. При росте количества значений в коллекции map лучше и к тому же дает постоянное время поиска

Выводы

1. slice очень даже неплох на небольших данных
   - а если еще и отсортировать ...
2. При росте количества значений в коллекции map лучше и к тому же дает постоянное время поиска
3. Большой map не лучше маленького, а иногда даже наоборот?

Что такое хэш-таблица?

Greetings:

key = “greeting”

value = “some description, use cases”

Что такое хэш-таблица?

Greetings:

key = “greeting”

value = “some description, use cases”

```
Insert() -> O(1)
Delete() -> O(1)
Get()    -> O(1)
```

Что такое хэш-таблица?

Greetings:

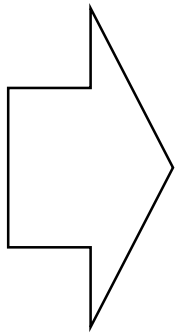
key = “greeting”

value = “some description, use cases”

key	value
“hello”	“Very friendly .. “
“bye”	“Englishmen never say ..”

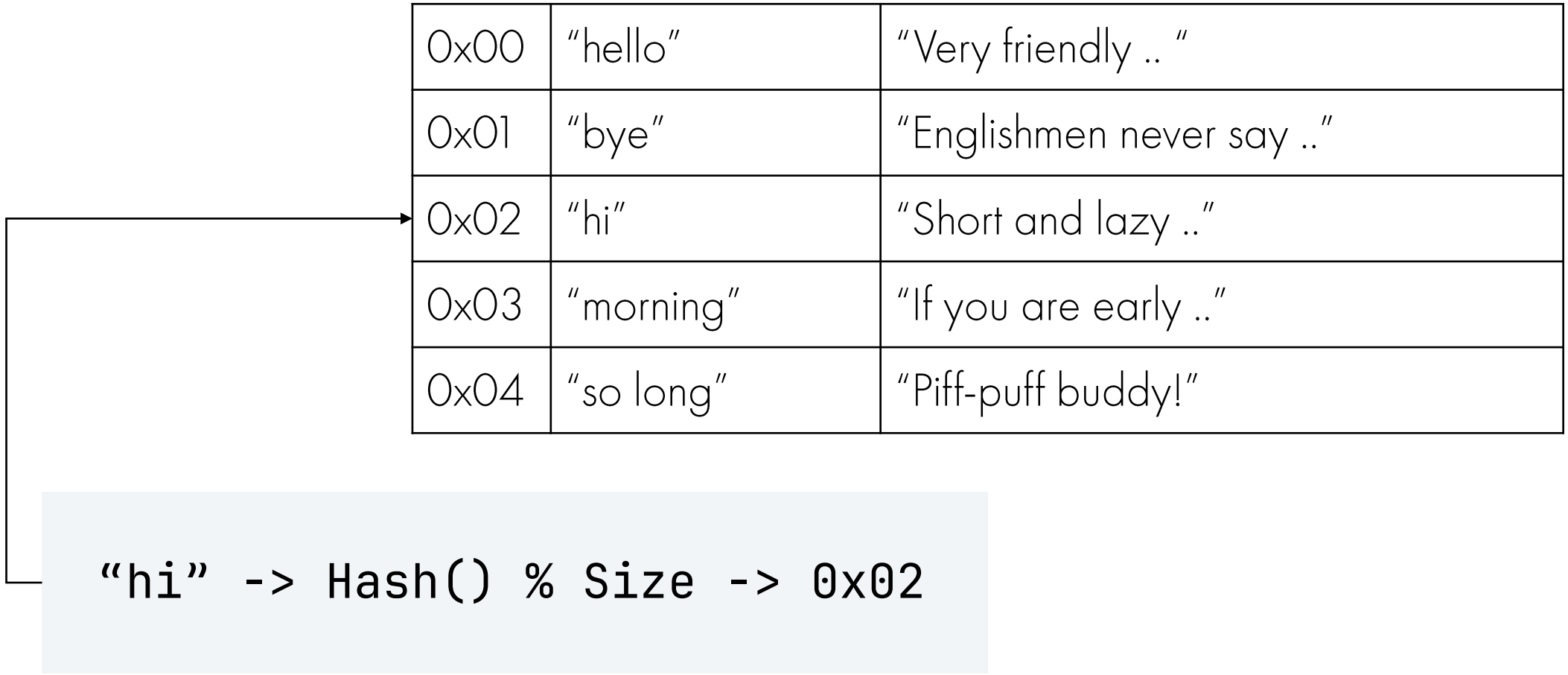
Что такое хэш-таблица?

Массив



0x00	“hello”	“Very friendly .. “
0x01	“bye”	“Englishmen never say ..”
0x02	“hi”	“Short and lazy ..”
0x03	“morning”	“If you are early ..”
0x04	“so long”	“Piff-puff buddy!”

Что такое хэш-таблица?



0x00	“hello”	“Very friendly .. ”
0x01	“bye”	“Englishmen never say ..”
0x02	“hi”	“Short and lazy ..”
0x03	“morning”	“If you are early ..”
0x04	“so long”	“Piff-puff buddy!”

```
“hi” -> Hash() % Size -> 0x02
```

Hash() дает одинаковую ячейку для двух ключей?

0x00	“hello”	“Very friendly .. ”
0x01	“bye”	“Englishmen never say ..”
0x02	“hi”	“Short and lazy ..”
0x03	“morning”	“If you are early ..”
0x04	“so long”	“Piff-puff buddy!”

```
“hi” -> Hash() % Size -> 0x02

“yo” -> Hash() % Size -> 0x02
```

Основные типы адресации

С открытой адресацией / двойной хеш:

- Если ячейка занята, хешируем еще раз

Основные типы адресации

С открытой адресацией / двойной хеш:

- Если ячейка занята, хешируем еще раз
- Большая скорость при малой загрузке таблицы

Основные типы адресации

С открытой адресацией / двойной хеш:

- Если ячейка занята, хешируем еще раз
- Большая скорость при малой загрузке таблицы
- Долго (очень) ищем свободную ячейку или нужный ключ при почти полной загрузке

Основные типы адресации

С открытой адресацией / двойной хеш:

- Если ячейка занята, хешируем еще раз
- Большая скорость при малой загрузке таблицы
- Долго (очень) ищем свободную ячейку или нужный ключ при почти полной загрузке

```
Hash(“hi”)          -> 0x02
Hash(Hash(“yo”))-> 0x03
```

0x02	“hi”	“Short and lazy ..”
0x03	“yo”	“If you are early ..”

Основные типы адресации

С открытой адресацией / двойной хеш:

- Если ячейка занята, хешируем еще раз
- Большая скорость при малой загрузке таблицы
- Долго (очень) ищем свободную ячейку или нужный ключ при почти полной загрузке

```
Hash(“hi”)          -> 0x02
Hash(Hash(“yo”))-> 0x03
```

0x02	“hi”	“Short and lazy ..”
0x03	“yo”	“If you are early ..”

Python dicts explained

Основные типы адресации

С открытой адресацией / двойной хеш:

- Если ячейка занята, хешируем еще раз
- Большая скорость при малой загрузке таблицы
- Долго (очень) ищем свободную ячейку или нужный ключ при почти полной загрузке

```
Hash(“hi”)          -> 0x02
Hash(Hash(“yo”))-> 0x03
```

0x02	“hi”	“Short and lazy ..”
0x03	“yo”	“If you are early ..”

Списки (цепочка коллизий):

- Помещаем дубликаты в списки

Основные типы адресации

С открытой адресацией / двойной хеш:

- Если ячейка занята, хешируем еще раз
- Большая скорость при малой загрузке таблицы
- Долго (очень) ищем свободную ячейку или нужный ключ при почти полной загрузке

```
Hash(“hi”)          -> 0x02
Hash(Hash(“yo”))-> 0x03
```

0x02	“hi”	“Short and lazy ..”
0x03	“yo”	“If you are early ..”

Списки (цепочка коллизий):

- Помещаем дубликаты в списки
- Простота реализации

Основные типы адресации

С открытой адресацией / двойной хеш:

- Если ячейка занята, хешируем еще раз
- Большая скорость при малой загрузке таблицы
- Долго (очень) ищем свободную ячейку или нужный ключ при почти полной загрузке

```
Hash(“hi”)       -> 0x02
Hash(Hash(“yo”))-> 0x03
```

0x02	“hi”	“Short and lazy ..”
0x03	“yo”	“If you are early ..”

Списки (цепочка коллизий):

- Помещаем дубликаты в списки
- Простота реализации
- Большая скорость при большой загрузке таблицы

Основные типы адресации

С открытой адресацией / двойной хеш:

- Если ячейка занята, хешируем еще раз
- Большая скорость при малой загрузке таблицы
- Долго (очень) ищем свободную ячейку или нужный ключ при почти полной загрузке

```
Hash("hi")       -> 0x02
Hash(Hash("yo"))-> 0x03
```

0x02	"hi"	"Short and lazy .."
0x03	"yo"	"If you are early .."

Списки (цепочка коллизий):

- Помещаем дубликаты в списки
- Простота реализации
- Большая скорость при большой загрузке таблицы

```
Hash("hi")  -> 0x02
Hash("yo")  -> 0x02
```

0x02	"hi": ... -> "yo": ... ->

Основные типы адресации

С открытой адресацией / двойной хеш:

- Если ячейка занята, хешируем еще раз
- Большая скорость при малой загрузке таблицы
- Долго (очень) ищем свободную ячейку или нужный ключ при почти полной загрузке

```
Hash("hi")       -> 0x02
Hash(Hash("yo"))-> 0x03
```

0x02	"hi"	"Short and lazy .."
0x03	"yo"	"If you are early .."

Списки (цепочка коллизий):

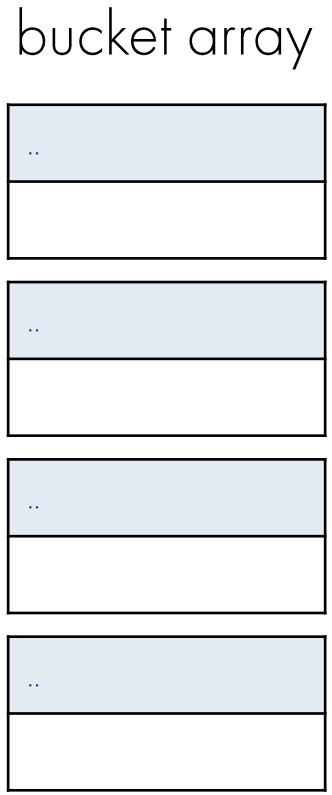
- Помещаем дубликаты в списки
- Простота реализации
- Большая скорость при большой загрузке таблицы

```
Hash("hi")  -> 0x02
Hash("yo")  -> 0x02
```

0x02	"hi": ... -> "yo": ... ->

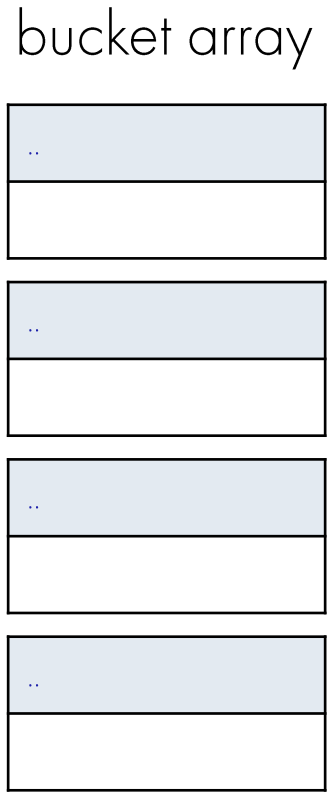
Go way!

Go map с высоты птичьего полета



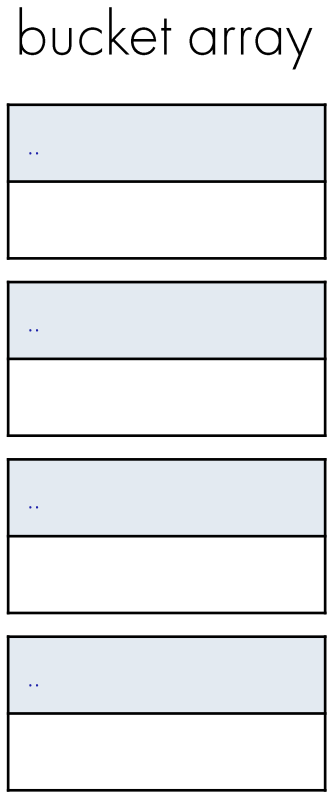
- Бакеты - списки коллизий по 8 элементов

Go map с высоты птичьего полета



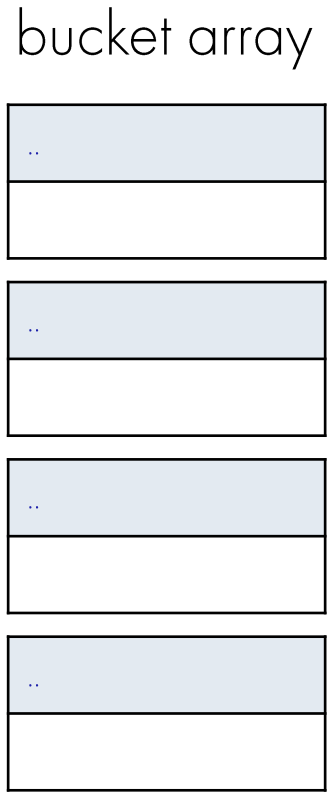
- Бакеты - списки коллизий по 8 элементов
- Map указывает на array бакетов

Go map с высоты птичьего полета



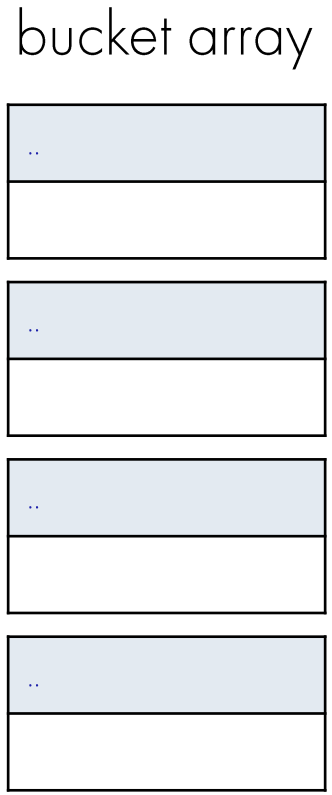
- Бакеты - списки коллизий по 8 элементов
- Map указывает на array бакетов
- Начинаем с пустоты

Go map с высоты птичьего полета



- Бакеты - списки коллизий по 8 элементов
- Map указывает на array бакетов
- Начинаем с пустоты
- Количество бакетов – степень двойки

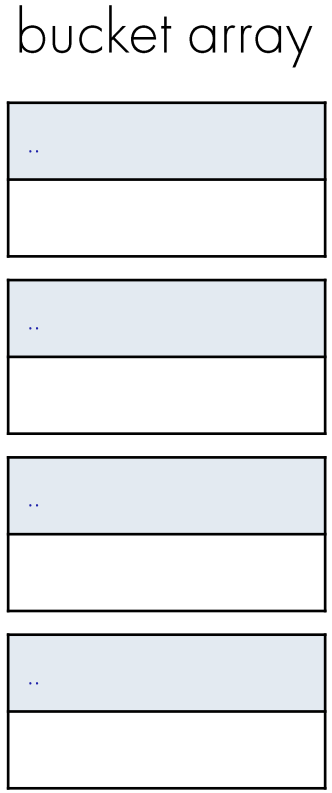
Go map с высоты птичьего полета



- Бакеты - списки коллизий по 8 элементов
- Map указывает на array бакетов
- Начинаем с пустоты
- Количество бакетов – степень двойки
- Растим и ребалансируем по мере заполнения

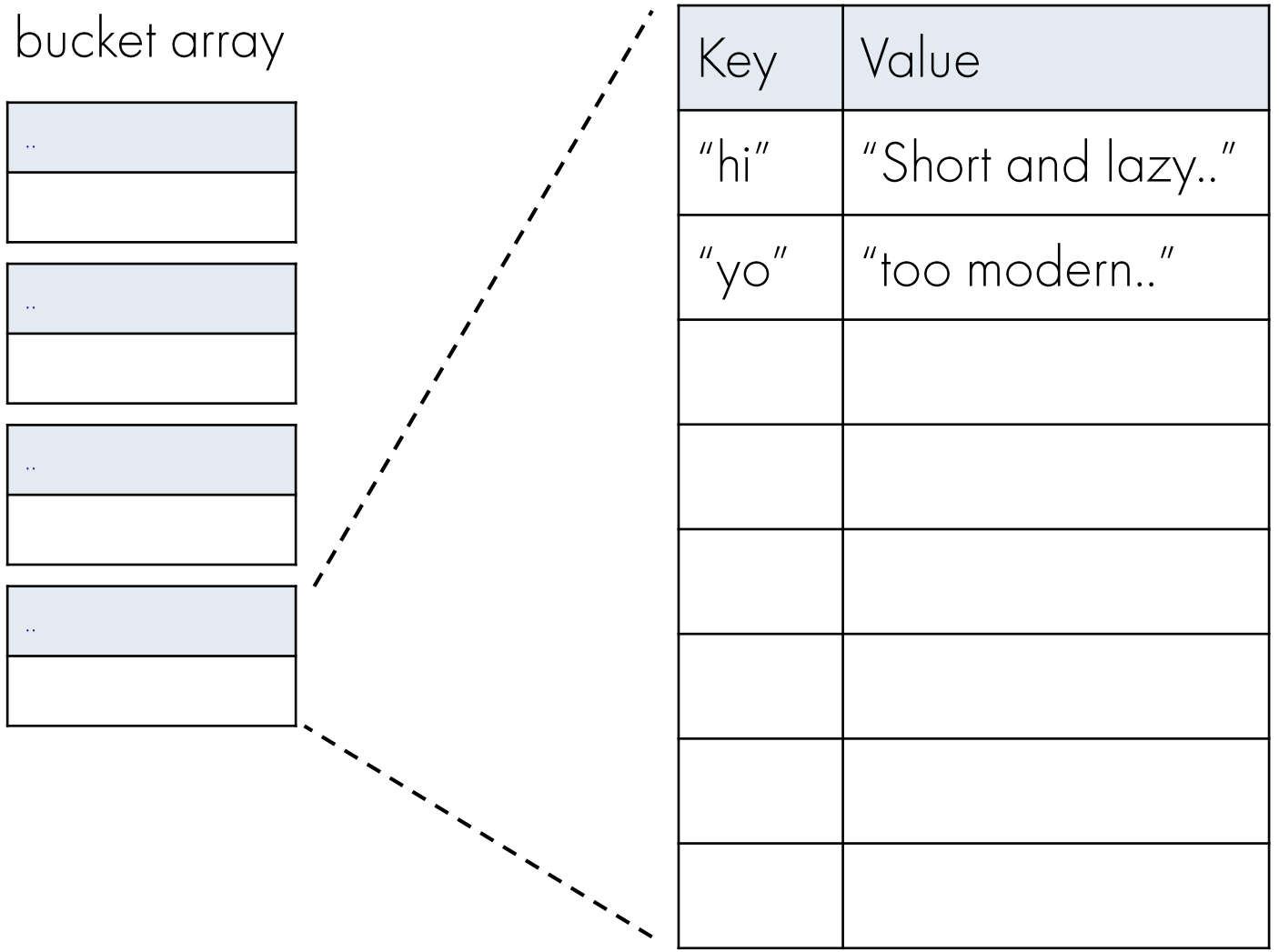
Go map с высоты птичьего полета

1. “hi”
2. hash()
3. 0x12345603
4. mask (0b11)
5. bucket 0x03 →



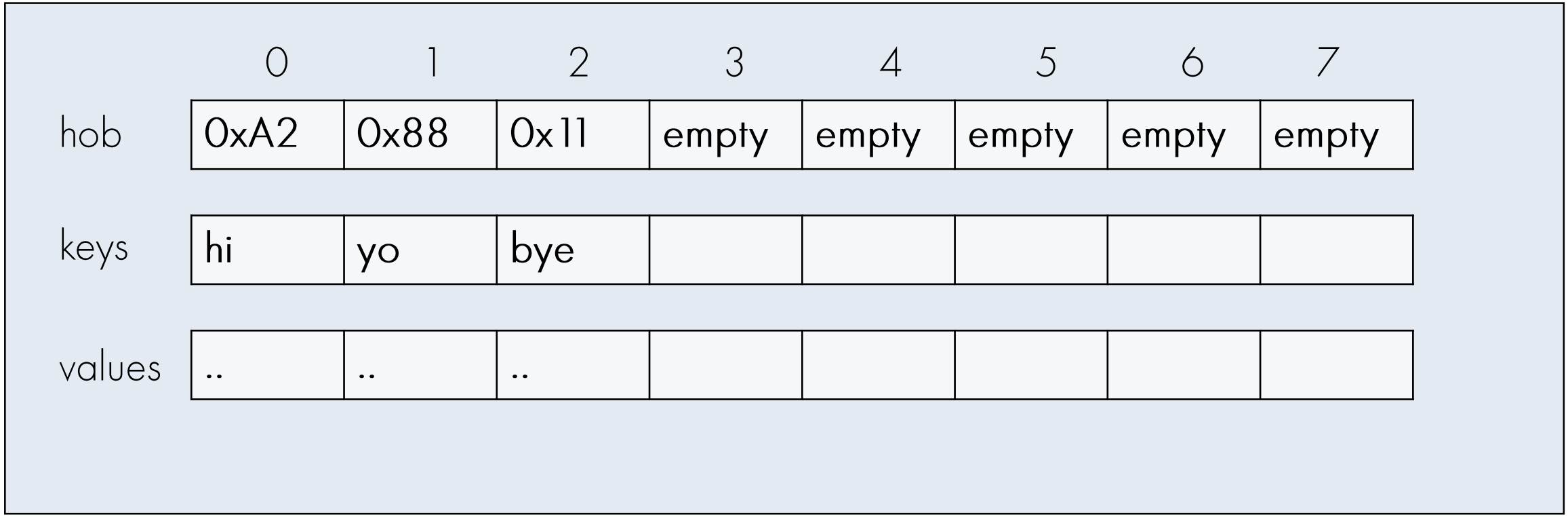
Go map с высоты птичьего полета

1. “hi”
2. hash()
3. 0x12345603
4. mask (0b11)
5. bucket 0x03 →



Go map с высоты птичьего полета

bucket




```
tophash(“yo”) -> 0x88
```

Как растут “мапы”?

Bucket Load Factor

Key	Value
adf	..
s	..
df	..
sfff	..
f	..
ssss	..
fsfs	..

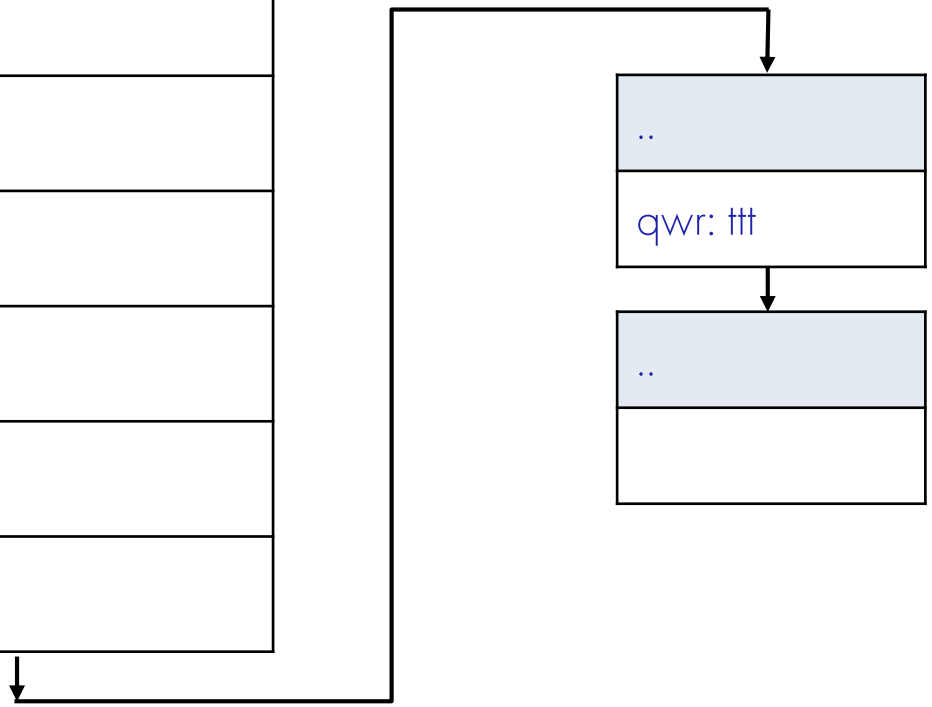
Key	Value
n4hh	..
ndg	..
dhjd	..
bdj	..
mdg	..
4ggs	..
dfaa	..
ggg	..

```
2 buckets:
(8 + 7) / 2 = 7.5
```

Overflow buckets

Key	Value

Key	Value
n4hh	..
ndg	..
dhjd	..
bdj	..
mdg	..
4ggs	..
dfaa	..
ggg	..



Map rebuild – когда?

- Bucket load factor
  - В среднем 6.5 из 8 ячеек заполнены

Map rebuild – когда?

- Bucket load factor
  - В среднем 6.5 из 8 ячеек заполнены
- Много overflow buckets
  - N overflow > N normal

Map rebuild – как?

- Делаем новый bucket array
  - x2, если большой load factor
  - “same size”, если много overflow

Map rebuild – как?

- Делаем новый bucket array
  - x2, если большой load factor
  - “same size”, если много overflow
- Текущий bucket array -> old buckets

Map rebuild – как?

- Делаем новый bucket array
- Текущий bucket array -> old buckets
- Копируем лениво:
  - во время записи новых значений
  - бакет, который “подошел”
  - +1 бакет “to make progress”

https://en.wikipedia.org/wiki/Hash_table#Dynamic_resizing

Выводы

Go map:

- основан на списках коллизий (8+ конфликтов)

Выводы

Go map:

- основан на списках коллизий (8+ конфликтов)
- компактный

Выводы

Go map:

- основан на списках коллизий (8+ конфликтов)
- компактный
- динамический рост и ленивая ребалансировка

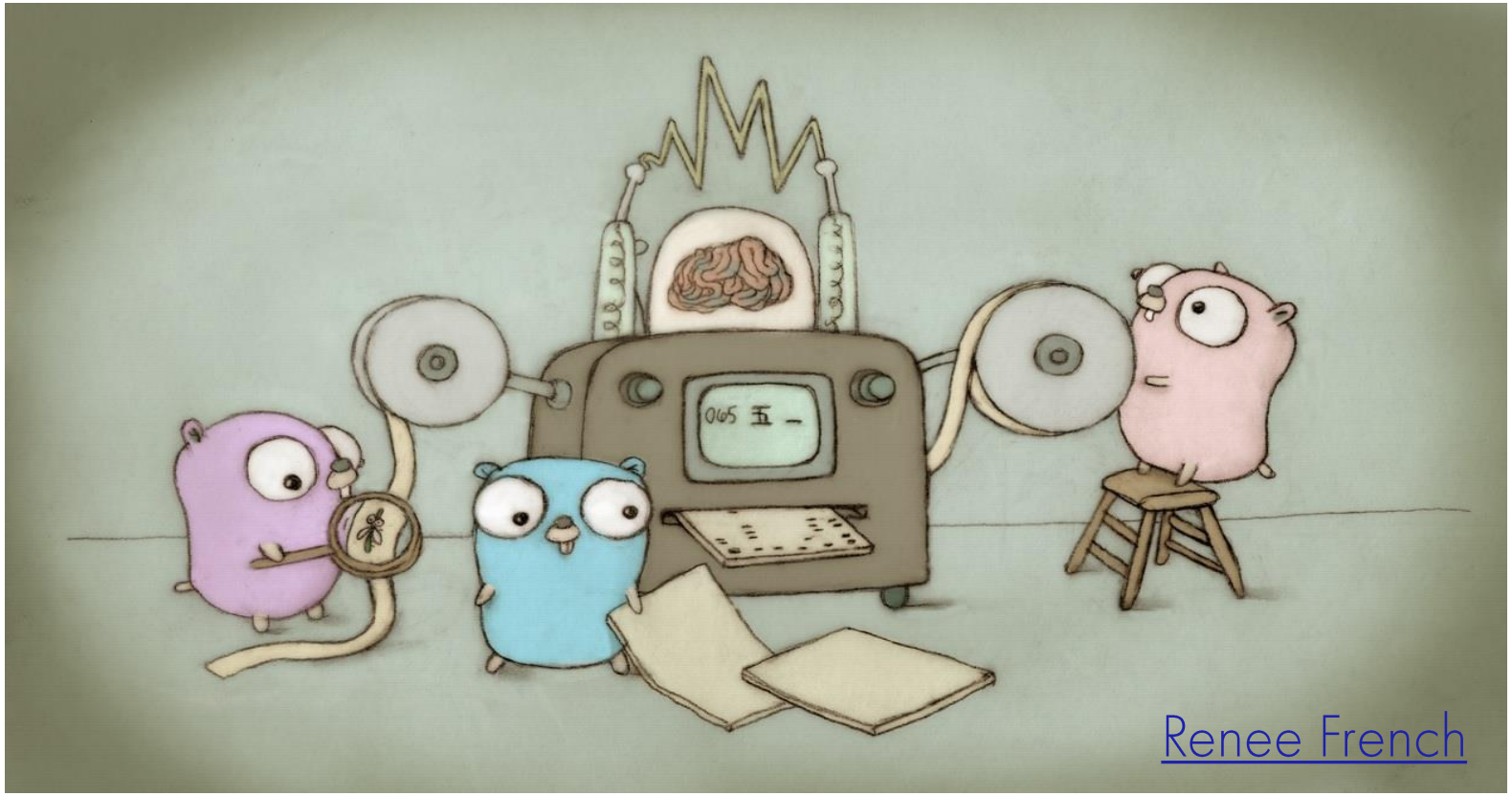
Выводы

Go map:

- основан на списках коллизий (8+ конфликтов)
- компактный
- динамический рост и ленивая ребалансировка
- map из менее 7 элементов = список

“Покажите мне код!”

- неизвестный, из неопубликованного



map – тип-указатель

```
type hmap struct {
      count      int                // num of entries
      hash0      uint32             // seed
      B          uint8              // mask (log2 of N)
      buckets    unsafe.Pointer  // array of buckets
      oldbuckets unsafe.Pointer
      …
}

// make(map[k]v)
func makemap(t *maptype, hint int, h *hmap) *hmap {..}
```

map: передача в функции и возврат

- Передача по указателю (указателя):
  - Можем полностью поменять все
- Возврат по указателю (указателя):
  - nil как признак пустого map

Нет семантики передача или возврат по значению

Map “magic”

```
v := m["key"]      → runtime.mapaccess1(m, "key")
v, ok := m["key"] → runtime.mapaccess2(m, "key")
m["key"] = 777     → runtime.mapinsert(m, "key", 777)
delete(m, "key")   → runtime.mapdelete(m, "key")
```

Все на кодогенерации и указателях, в первом приближении

Map “magic”

```
type maptype struct {
     key       *_type
     …

type _type struct {
     alg       *typeAlg
     …
```

Map access “simplified”

```
func mapaccess1(t *maptype, h *hmap,
    key unsafe.Pointer) unsafe.Pointer {

    alg    := t.key.alg
```

Map access “simplified”

```
func mapaccess1(t *maptype, h *hmap,
    key unsafe.Pointer) unsafe.Pointer {

    alg    := t.key.alg

    hash   := alg.hash(key, uintptr(h.hash0))
```

Map access “simplified”

```
func mapaccess1(t *maptype, h *hmap,
    key unsafe.Pointer) unsafe.Pointer {

    alg    := t.key.alg

    hash   := alg.hash(key, uintptr(h.hash0))

    mask   := bucketMask(h.B)
```

Map access “simplified”

```
func mapaccess1(t *maptype, h *hmap,
    key unsafe.Pointer) unsafe.Pointer {

    alg    := t.key.alg

    hash   := alg.hash(key, uintptr(h.hash0))

    mask   := bucketMask(h.B)

    bucket := (*bmap)(add(
        h.buckets, (hash&m)*uintptr(t.bucketsize)))
```

Хорошо, а что с хеш-функциями?

Если есть поддержка AES ENC в CPU:

- AES hash от слайса байт - int, float, ptr, string

Хорошо, а что с хеш-функциями?

Если есть поддержка AES ENC в CPU:

- AES hash от слайса байт - int, float, ptr, string
- Хеш структуры - хеши полей

Хорошо, а что с хеш-функциями?

Если есть поддержка AES ENC в CPU:

- AES hash от слайса байт - int, float, ptr, string
- Хеш структуры - хеши полей
- Хеш функция структуры - генерируемый код

Хорошо, а что с хеш-функциями?

Если есть поддержка AES ENC в CPU:

- AES hash от слайса байт - int, float, ptr, string
- Хеш структуры - хеши полей
- Хеш функция структуры - генерируемый код
- Реализация в ASM, оптимизации под конкретные типы

Хорошо, а что с хеш-функциями?

Если есть поддержка AES ENC в CPU:

- AES hash от слайса байт - int, float, ptr, string
- Хеш структуры - хеши полей
- Хеш функция структуры - генерируемый код
- Реализация в ASM, оптимизации под конкретные типы

Если нет:

- Используется wyhash: https://github.com/wangyi-fudan/wyhash

Давайте выбираться наружу ...

Организованный беспорядок

```
m := map[int]string {
  1: "one", 2: "two",
  3: "three", 4: "four"}

fmt.Println(m) // always sorted

for k, v := range m {
      fmt.Printf("%s:%d ", k, v) // always random
}
```

Нужен ли hint для 1k ключей?

```
checkin := make(map[string]bool)
// fastCheckin := make(map[string]bool, 1024)

for _, p := range passengers {
    checkin[p] = true
}
```

```
BenchmarkMap/hint:_0-8       69552 ns/op    94250 B/op      22 allocs/op
BenchmarkMap/hint:_512-8     47943 ns/op    49882 B/op       5 allocs/op
BenchmarkMap/hint:_1024-8    27711 ns/op        0 B/op       0 allocs/op
BenchmarkMap/hint:_2048-8    26472 ns/op        0 B/op       0 allocs/op
```

Нужен ли hint для 1k ключей?

```
// создание мапы включено в тест
checkin := make(map[string]bool, size)

for _, p := range passengers {
    checkin[p] = true
}
```

```
BenchmarkMap/hint:_0-8        71171 ns/op    94285 B/op      23 allocs/op
BenchmarkMap/hint:_512-8      55714 ns/op    71710 B/op       8 allocs/op
BenchmarkMap/hint:_1024-8     32499 ns/op    49224 B/op       3 allocs/op
BenchmarkMap/hint:_2048-8     42092 ns/op    90184 B/op       3 allocs/op
BenchmarkMap/hint:_4096-8     40567 ns/op   172104 B/op       3 allocs/op
```

Эмулируем set

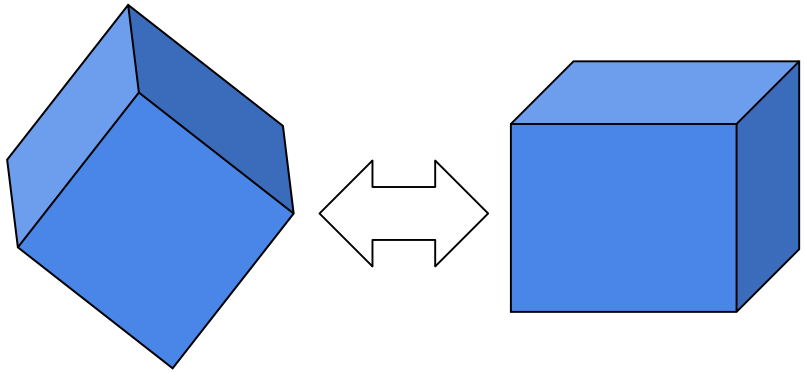
```
type void struct{}
seen := make(map[string]void)

for i, w := range words {
    seen[w] = void{}
}

// check word is in
if _, ok := seen[“hello]; ok {
    …
}
```

Эмулируем set – способ 2

```
seen := make(map[string]bool)

for i, w := range words {
    seen[w] = true
}

// check word is in
if seen[“hello”] {
    …
}
```

Что выбрать?

- Одинаковая скорость поиска
- bool проще
- Go source code, употребление
  - map[string]bool 471
  - map[string]struct{} 27



Вставить, нельзя поменять?

```
m["Misha"] = Contact{
      Email: "misha@home.org",
      Phone: "22333322",
}
…
m["Misha"].Phone = "77777777"
                  // Error: “Cannot assign struct field”
```

Неадресуемые значения - почему?

- Слишком много коллизий - ленивая ребалансировка:
  - При добавлении элементов происходит перенос в новую таблицу.

Неадресуемые значения - почему?

- Слишком много коллизий - ленивая ребалансировка:
  - При добавлении элементов происходит перенос в новую таблицу.
- Go не дает поменять value - может возникнуть ситуация гонки:
  - Вернувшееся значение может переехать на новый адрес

Неадресуемые значения - решение 1

```
…
contact := m["Misha"]
contact.Phone = "77777777"
m["Misha"] = contact
```

Неадресуемые значения - решение 1

```
m := make(map[string]*Contact)

…
m["Misha"].Phone = "77777777"
```

Какое решение быстрее? Создание мапы


```
m[id] = Contact{
    ID:    id,
    Name:  "my name",
    Phone: "my phone",
}
```

```
m[id] = &Contact{
    ID:    id,
    Name:  "my name",
    Phone: "my phone",
}
```

Какое решение быстрее? Создание мапы


```
m[id] = Contact{
    ID:    id,
    Name:  "my name",
    Phone: "my phone",
}
```

```
m[id] = &Contact{
    ID:    id,
    Name:  "my name",
    Phone: "my phone",
}
```

```
BenchmarkCreateValueMap-8     44135 ns/op  147528 B/op     3 allocs/op
BenchmarkCreatePointerMap-8 101578 ns/op  106568 B/op  1027 allocs/op
```

Какое решение быстрее? Получение элемента

```
contact := m[id]
```

```
contact := m[id]
```

```
BenchmarkAccessPointerMap-8    8262 ns/op     0 B/op    0 allocs/op
BenchmarkAccessValueMap-8      8694 ns/op     0 B/op    0 allocs/op
```

Какое решение быстрее? Замена

```
tmp := m[id]
tmp.Phone = "123-456"
m[id] = tmp
```

```
m[id].Phone = "123-456"
```

```
BenchmarkChangeValueMap-8      26828 ns/op   0 B/op   0 allocs/op
BenchmarkChangePointerMap-8     9602 ns/op   0 B/op   0 allocs/op
```

Велосипеды

```
// I like bicycles!
for k := range myMap {
    keys = append(keys, k)
}

...
// one-liner!
keys := maps.Keys(myMap)
```

Clone(), Copy(), Equal(), **Keys(), Values()** на дженериках и итераторах!

Конкуренция

```
type LockMap struct {
    lock *sync.RWMutex
    m    map[string]int
}
func NewLockMap(…)
…
m := NewLockMap()
m.Save(“score”, 777)
```

```
import “sync”




var m sync.Map
m.Store(“score”, 777)
```

sync.Map vs map + mutex

- map + sync.RWMutex не очень быстр когда много ядер и горутин
  - cache coherence - соответствие кешей на разных ядрах
  - блокировка портит кеши
  - чем больше ядер, тем больше синхронизации

sync.Map vs map + mutex

- map + sync.RWMutex не очень быстр когда много ядер и горутин
  - cache coherence - соответствие кешей на разных ядрах
  - блокировка портит кеши
  - чем больше ядер, тем больше синхронизации
- sync.Mutex работает быстрее sync.RWMutex

sync.Map vs map + mutex

- map + sync.RWMutex не очень быстр когда много ядер и горутин
  - cache coherence - соответствие кешей на разных ядрах
  - блокировка портит кеши
  - чем больше ядер, тем больше синхронизации
- sync.Mutex работает быстрее sync.RWMutex
- И дело не только в кешах

sync.Map vs map + mutex

- sync.Map - специализированная имплементация

sync.Map vs map + mutex

- sync.Map - специализированная имплементация
  - ✓ write once read many

sync.Map vs map + mutex

- sync.Map - специализированная имплементация
  - ✓ write once read many
  - ✓ множество горутин пишут-читают свое подмножество ключей

sync.Map vs map + mutex

- sync.Map - специализированная имплементация
  - ✓ write once read many
  - ✓ множество горутин пишут-читают свое подмножество ключей
  - ✓ позволяет избежать издержек мьютексов

sync.Map vs map + mutex

- sync.Map - специализированная имплементация
  - ✓ write once read many
  - ✓ множество горутин пишут-читают свое подмножество ключей
  - ✓ позволяет избежать издержек мьютексов
  - ✓ Не дает напортачить новичкам с конкурентным доступом

sync.Map vs map + mutex

- sync.Map - специализированная имплементация
  - ✓ write once read many
  - ✓ множество горутин пишут-читают свое подмножество ключей
  - ✓ позволяет избежать издержек мьютексов
  - ✓ Не дает напортачить новичкам с конкурентным доступом
    - ➢ проще код ревью, меньше шансов на непоправимую панику

sync.Map vs map + mutex

- sync.Map - специализированная имплементация
  - ✓ write once read many
  - ✓ множество горутин пишут-читают свое подмножество ключей
  - ✓ позволяет избежать издержек мьютексов
  - ✓ Не дает напортачить новичкам с конкурентным доступом
    - ➢ проще код ревью, меньше шансов на непоправимую панику
  - Type safety - any

sync.Map vs map + mutex

“Используй map + mutex почти везде”, - говорили они

50% random read + 50% random write, 100k элементов

```
BenchmarkLockMap/2-8     41_271042 ns/op         178 B/op          5 allocs/op
BenchmarkLockMap/4-8     75_334550 ns/op         764 B/op         11 allocs/op
BenchmarkLockMap/6-8     83_730845 ns/op        1065 B/op         18 allocs/op
BenchmarkLockMap/8-8     99_367751 ns/op        1250 B/op         20 allocs/op

BenchmarkSyncMap/2-8     10_496833 ns/op     4094089 B/op     306949 allocs/op
BenchmarkSyncMap/4-8     14_816287 ns/op     8188220 B/op     613897 allocs/op
BenchmarkSyncMap/6-8     18_310827 ns/op    12282887 B/op     920846 allocs/op
BenchmarkSyncMap/8-8     21_380772 ns/op    16376559 B/op    1227793 allocs/op
```

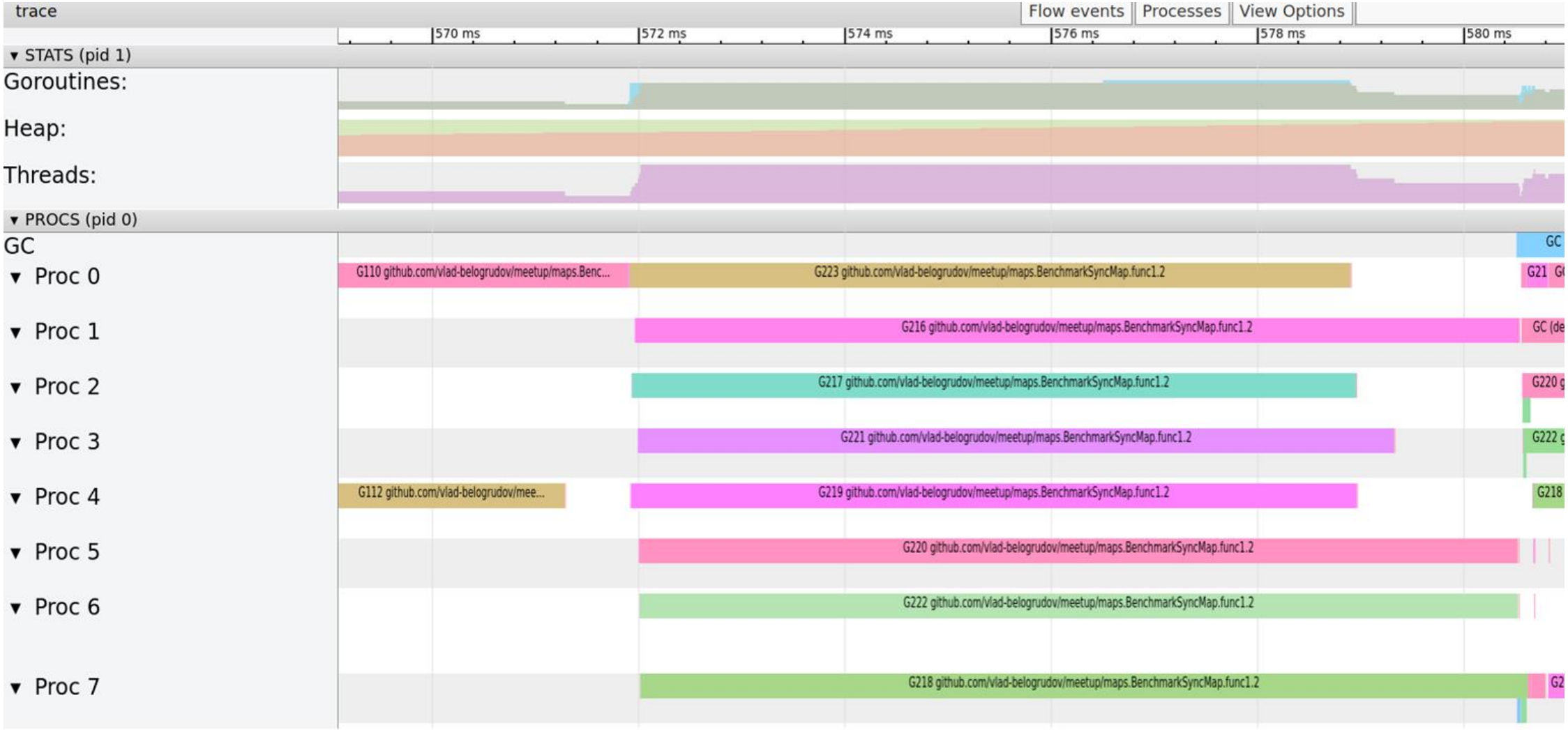
sync.Map vs map + mutex

И еще немножко веселья

```
> go test -bench='^(BenchmarkLockMap)$' -run='^$'

BenchmarkLockMap/2-8         164       7_322615 ns/op
BenchmarkLockMap/4-8          55      20_928206 ns/op
BenchmarkLockMap/6-8          28      39_958776 ns/op
BenchmarkLockMap/8-8          19      60_600790 ns/op

> GOMAXPROCS=1 go test -bench='^(BenchmarkLockMap)$' -run='^$'

BenchmarkLockMap/2           218       5_385667 ns/op
BenchmarkLockMap/4           100      11_223965 ns/op
BenchmarkLockMap/6            76      15_947113 ns/op
BenchmarkLockMap/8            52      22_159325 ns/op
```

map + mutex: trace view

sync.Map: trace view



Подведем итоги

1. Тип-указатель - передача всегда “by pointer”

Подведем итоги

1. Тип-указатель - передача всегда “by pointer”
2. Принудительный беспорядок итераций

Подведем итоги

1. Тип-указатель - передача всегда “by pointer”
2. Принудительный беспорядок итераций
3. Размер в make(..) имеет значение

Подведем итоги

1. Тип-указатель - передача всегда “by pointer”
2. Принудительный беспорядок итераций
3. Размер в make(..) имеет значение
4. “Неадресуемость” значений и 2 пути решения

Подведем итоги

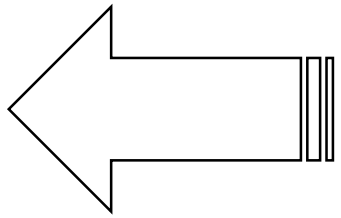
1. Тип-указатель - передача всегда “by pointer”
2. Принудительный беспорядок итераций
3. Размер в make(..) имеет значение
4. “Неадресуемость” значений и 2 пути решения
5. “value map” для чтения, “pointer map” для изменений

Подведем итоги

1. Тип-указатель - передача всегда “by pointer”
2. Принудительный беспорядок итераций
3. Размер в make(..) имеет значение
4. “Неадресуемость” значений и 2 пути решения
5. “value map” для чтения, “pointer map” для изменений
6. Стандартный пакет “maps” в помощь

Подведем итоги

1. Тип-указатель - передача всегда “by pointer”
2. Принудительный беспорядок итераций
3. Размер в make(..) имеет значение
4. “Неадресуемость” значений и 2 пути решения
5. “value map” для чтения, “pointer map” для изменений
6. Стандартный пакет “maps” в помощь



7. sync.Map для “суетливых” горутин!

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

yadro.com

@VLAD_BELOGRUDOV