Map Go

#map #hashmap #go

Простая реализация типа Map

Тип map по своей сути представляет таблицу из ключ значения. Посмотрим на простую реализацию:

```
type entry struct {
        key string
        value int
}

type simpleMap []entry

func lookUp(m simpleMap, key string) int {
        for _, e := range m {
                if e.key == key {
                        return e.value
                }
        }
        return 0
}
```

А теперь посчитаем сложность: $O(n)$, что не соответсвует стандарту hashMap

Идея: Деление на bucket

Разделим все наши данные на группы, назовем их бакетами. Каждый бакет будет содержать не более 8 элементов. Чтобы элементы в бакетах равномерно распределялись (не было ситуации, где все элементы начинаются с одного и того же символа), введем хэш функции.

Хэш функция

Хэш функция имеет следующий вид:

$$bucket = hash(key)$$

Данная функция должна отвечать следующим требованиям:

- Равномерность
- Быстрота
- Детерминированность
- Криптоустойчивость
  Теперь мы имеем хэш таблицу.

Хэш таблицы

`v=m[k]` скомпилируется в: `v = runtime.lookup(m,k)`
Сигнатура lookup:
`func<K,V> lookup(m map[K]V, k K) V` - то есть у нас используются дженерики, т.к. значения в мапе могут быть универсальные. Но в go они появились в 1.19.
Но как обойтись без них?

- Все операции выполняются с помощью `unsafe.Pointers`.
- Мета-информация о типах хранится в **type descriptor**.
- Type descriptor предоставляет операции hash, equal, copy.
  Например:

```
type _type struct {
        size uintptr
        equal func(unsafe.Pointer, unsafe.Pointer) bool
        hash func(unsafe.Pointer, uintptr) uintptr
        ...
}
```

```
type mapType struct {
        key *_type
        value *_type
        ...
}
```

Теперь посмотрим исходный код Go

```
// A header for a Go map.
type hmap struct {
        // Note: the format of the hmap is also encoded in
cmd/compile/internal/reflectdata/reflect.go.
        // Make sure this stays in sync with the compiler's definition.
        count     int // # live cells == size of map.  Must be first (used
by len() builtin)
        flags     uint8
        B         uint8  // log_2 of # of buckets (can hold up to loadFactor
* 2^B items)
        noverflow uint16 // approximate number of overflow buckets; see
incrnoverflow for details
        hash0     uint32 // hash seed

        buckets    unsafe.Pointer // array of 2^B Buckets. may be nil if
count==0.
        oldbuckets unsafe.Pointer // previous bucket array of half the size,
non-nil only when growing
        nevacuate  uintptr        // progress counter for evacuation
(buckets less than this have been evacuated)

        extra *mapextra // optional fields
}
```

Теперь посмотрим во что же скомпилируется `v = m[k]`:

```
pk := unsafe.Pointer(&k)
func lookup(t *mapType,
                    m *mapHeader,
                    k unsafe.Pointer) unsafe.Pointer
pv := runtime.lookup(typeOf(m),m,pk)
v = *(*V)pv
```

Посмотрим список преобразований:

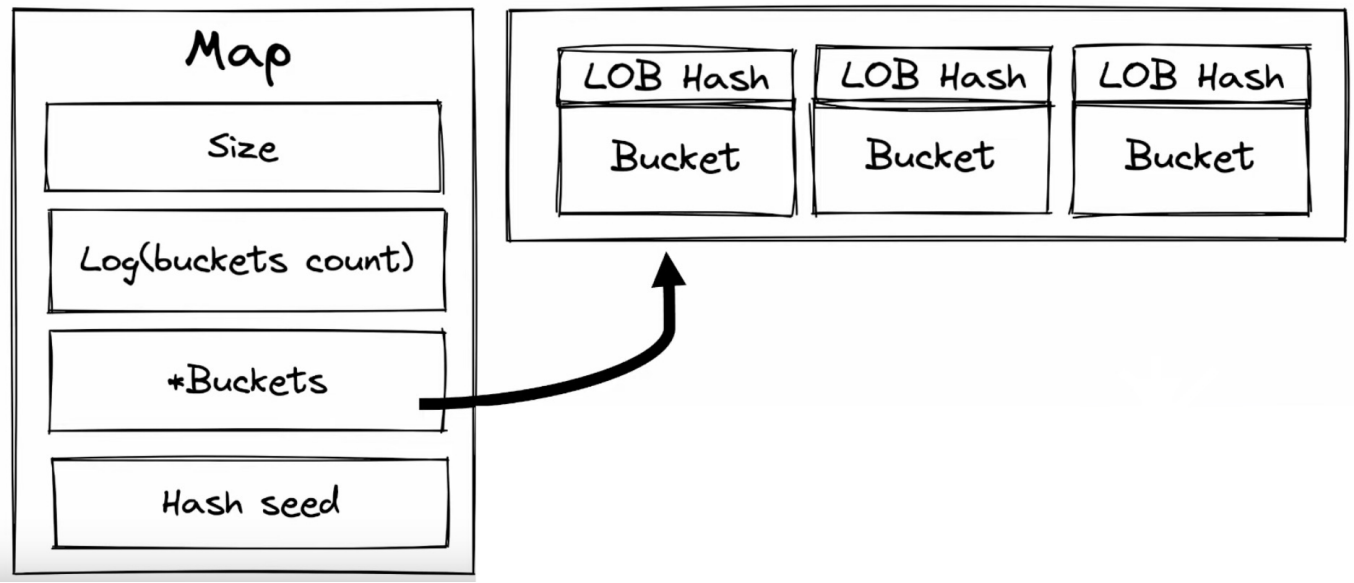
```
v:= m["k"] → func mapaccess1(t *maptype,
                                                    h *hmap,
                                                    k unsafe.Pointer)
unsafe.Pointer

v, ok:=m|"k"] → func mapaccess2(t *maptype,
                                                          h *hmap,
                                                          k
unsafe.Pointer) (unsafe.Pointer, bool)
```

```
m["k"] = 9001 → func mapassign(t *maptype,
                                                    h *hmap,
                                                    k unsafe.Pointer)
unsafe.Pointer

delete(m, "k") → func mapdelete(t *maptype,
                                                        h *hmap,
                                                        k
unsafe.Pointer)
```

Смотрим структуру

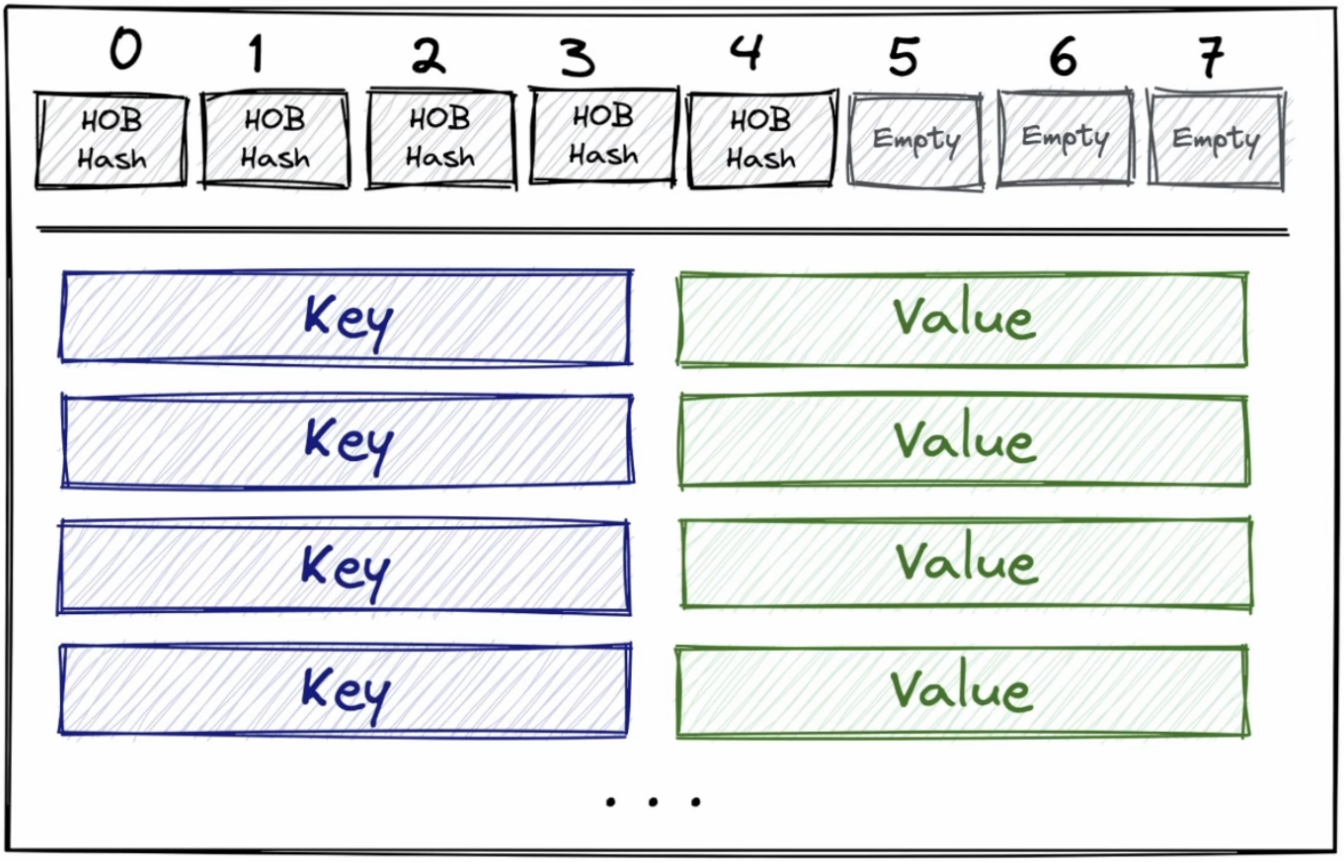


- Размер мапы.
- Логарифм от кол-ва бакетов, для ускорения побитовых операций.
- Указатель на бакеты.
- Зерно.

LOB (Low Order Bits)

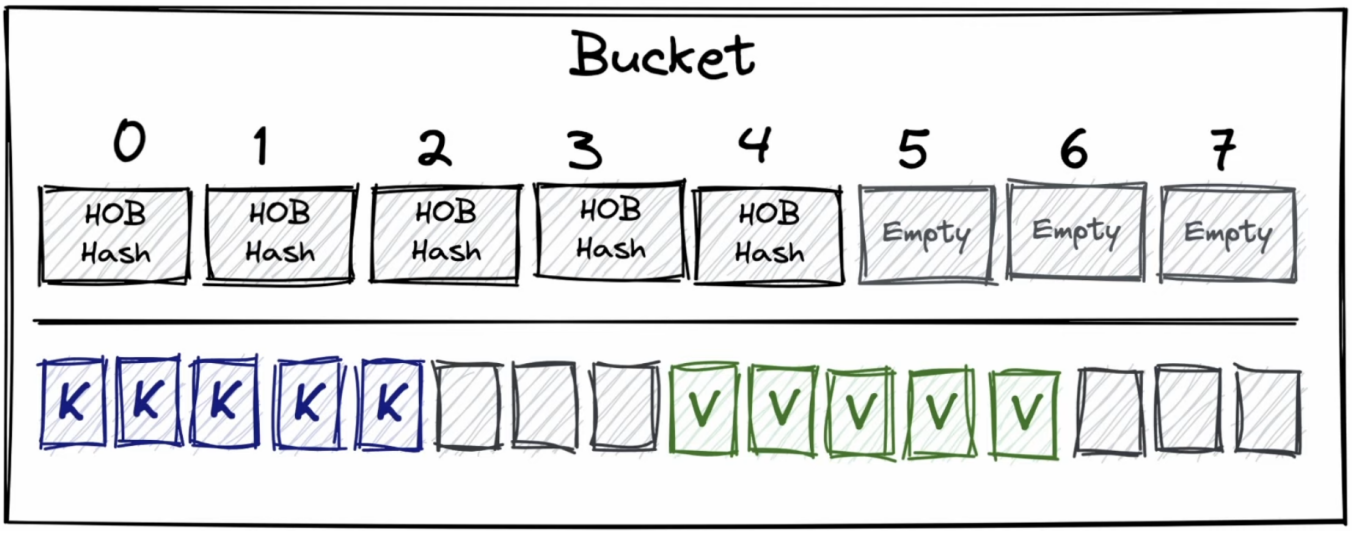
$Hash(key) = 5431$
Мы получили большое число, хотя бакетов мало. Как найти нужный?
Возьмем остаток от деление на кол-во бакетов: $5461\%4 = 1$. Однако остаток от деления считается побитово. Но там нам и нужен логарифм, т.к. в двоичном представлении log последних бит будет соответствовать номеру нашего бакета.
Таким образом LOB(Hash)=01

Структура бакета



В каждом бакете хранится не более 8 значений.

В памяти это выглядит так:

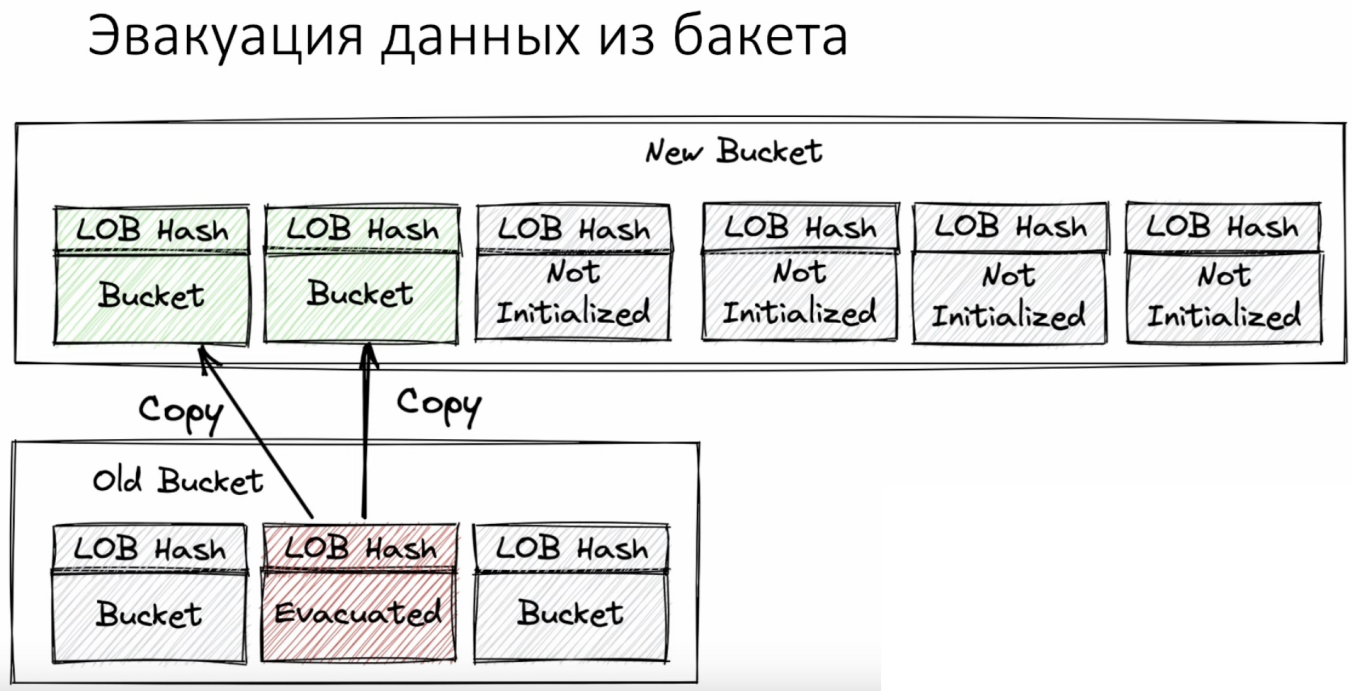


Так как нам нужно выравнивать память для экономии места.

Также, во всех функциях используются побитовые маски, 1<<log(n) -1 - универсальная операция для нахождения маски и при умножении любого числа на маску, получаем только последние n бит.

Переполнение бакета

Новая память выделяется тогда, когда в каждом бакете лежит в среднем 6.5 значений. Как только бакеты достигают этого значения , начинается эвакуация бакетов. Этот процесс достаточно медленный. Память также увеличивается в два раза.



Именно поэтому мы не можем брать ссылку на элементы в мапе, т.к. они находятся в каком-то бакете, а после эвакуации будут в другом, таким образом ссылка устареет.

Однако в mapaccess1 возвращается указатель, но он всегда разыменовывается.

Почему обход случайный?

Обход итератора зависит слишком от многих факторов, старые бакеты, хэш функция и многое другое. Поэтому в итераторе был сделан `fastrand` чтоб обход был случайный.
P.S. в пакете fmt идет сортировка элементов по ключам.