Как устроен map в go

января 7, 2025 · 11 минут · German Gorelkin

Перевод/заметки Go Maps Explained: How Key-Value Pairs Are Actually Stored

---

Quick Start

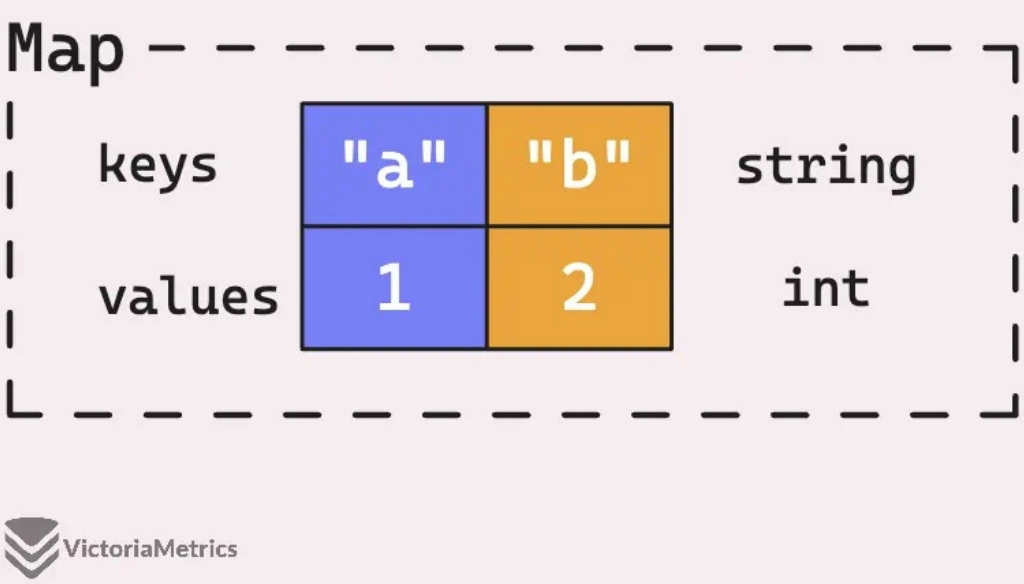
Итак, давайте обсудим map в Go. Это встроенный тип данных, который служит хранилищем пар ключ-значение. В отличие от массивов, где ключи представляют собой возрастающие индексы, такие как 0, 1, 2 и так далее, в map ключ может быть любого *сопоставимого(comparable)* типа.

```
m := make(map[string]int)
```

```
m["a"] = 1
m["b"] = 2

m // map[a:1 b:2]
```

В этом примере мы создали пустую map с помощью `make()`, где ключи - строки, а значения - целые числа.



Вместо того чтобы назначать каждый ключ вручную, вы можете сэкономить время, используя литерал для map. Это позволяет вам задать пары ключ-значение сразу, непосредственно при создании map:

```
m := map[string]int{
    "a": 1,
    "b": 2,
}
```

Всё, что вам нужно сделать, — это указать ключи и их значения в фигурных скобках при первом создании map.

Если позже вы решите, что определённая пара ключ-значение вам больше не нужна, вы можете воспользоваться удобной функцией `delete`, которая удалит этот ключ. Например, чтобы удалить ключ `a` из `m`, вы можете использовать следующий код: `delete(m, "a")`.

Нулевое значение map - `nil`. Вы можете попытаться найти в ней ключи, и Go не

станет пниковать.

Если вы ищете ключ, которого там нет, Go просто спокойно выдает вам *«нулевое значение» для типа значения* этой map:

```
var m map[string]int

println(m["a"]) // 0
m["a"] = 1      // panic: assignment to entry in nil map
```

Но вы не можете добавлять новые пары ключ-значение в `nil` *map*.

На самом деле, Go работает с map примерно так же, как и со слайсами. И map, и слайсы изначально равны nil, и Go не вызывает панику, если вы делаете с ними что-то «безобидное». Например, вы можете выполнить цикл над nil-слайсом без каких-либо проблем.

Что же произойдет, если вы попытаетесь выполнить цикл по nil map?

```
var m map[string]int

for k, v := range m {
    println(k, v)
}
```

Ничего не происходит, никаких ошибок, никаких сюрпризов.

Подход Go заключается в том, чтобы рассматривать значение по умолчанию любого типа как нечто полезное, а не как нечто, из-за чего ваша программа сломается. Единственный случай, когда Go начинает возмущаться, - это когда вы делаете что-то действительно незаконное, например, пытаетесь добавить новую пару ключ-значение в nil map или обращаетесь к выходящему за границы индексу в слайсе.

Есть еще несколько вещей, которые вы должны знать о map в Go:

- Цикл `for-range` по map не будет возвращать ключи в каком-либо определенном порядке.
- map не являются потокобезопасными, go runtime вызовет панику, если вы попытаетесь одновременно читать (или выполнять итерацию с помощью `for-`

`range`) и записывать в одну и ту же map.

- Вы можете проверить, находится ли ключ в map, выполнив простую ok-проверку: `_, ok := m[key]`.
- Тип ключа для map должен быть сравнимым(comparable).

> *Что именно является сопоставимым типом? А что нет?*

Все очень просто: если вы можете использовать == для сравнения двух значений одного и того же типа, то этот тип считается сравнимым.

```
func main() {
    var s map[int]string

    if s == s {
        println("comparable")
    }
}

// compile error: invalid operation: s == s (map can only be compared to nil)
```

Это же правило относится и к другим несопоставимым типам, таким как слайсы, функции, структуры, содержащие слайсы или map, и т.д. Поэтому, если вы пытаетесь использовать любой из этих типов в качестве ключа, вам не повезло.

```
func main() {
  var s map[[]int]string
}

// compile error: invalid map key type []intcompilerIncomparableMapKey
```

Небольшой секрет: **интерфейсы могут быть как сопоставимыми, так и несопоставимыми**.

Можно определить map с пустым интерфейсом в качестве ключа без каких-либо ошибок компиляции. Но будьте осторожны, велика вероятность, что вы столкнетесь с ошибкой во время выполнения.

```
func main() {
    m := map[interface{}]int{
        1: 1,
        "a": 2,
```

```
    }

    m[[]int{1, 2, 3}] = 3
    m[func() {}] = 4
}

// panic: runtime error: hash of unhashable type []int
// panic: runtime error: hash of unhashable type func()
```

Все выглядит хорошо, пока вы не попытаетесь присвоить несравнимый тип в качестве ключа.

Тогда вы столкнетесь с ошибкой времени выполнения, с которой справиться сложнее, чем с ошибкой времени компиляции. Поэтому обычно лучше не использовать `interface{}` в качестве ключа, если у вас нет веских причин и ограничений.

Однако сообщение об ошибке `hash of unhashable type []int` может показаться немного непонятным. Что это за хэш такой?

Map Anatomy

map в go состоит из множества более мелких элементов, которые называются «*buckets*».

```
type hmap struct {
  ...
  buckets unsafe.Pointer
  ...
}
```

map содержит указатель, который указывает на массив бакетов.

Поэтому, когда вы присваиваете map переменной или передаете ее в функцию, и переменная, и аргумент функции используют один и тот же указатель map.

```
func changeMap(m2 map[string]int) {
  m2["hello"] = 2
}

func main() {
```

```
  m1 := map[string]int{"hello": 1}
  changeMap(m1)
  println(m1["hello"]) // 2
}
```

map - это указатели на `hmap` под капотом, но это не ссылочные типы, и они не передаются по ссылке, как `ref` аргумент в C#, если вы измените всю карту `m2`, это не отразится на оригинальной карте `m1` в вызывающей программе.

```
func changeMap(m2 map[string]int) {
  m2 = map[string]int{"hello": 2}
}

func main() {
  m1 := map[string]int{"hello": 1}
  changeMap(m1)
  println(m1["hello"]) // 1
}
```

В Go все передается по значению. На самом деле все происходит немного иначе: когда вы передаете `m1` в функцию `changeMap`, Go создает копию структуры `*hmap`. Таким образом, `m1` в функции `main()` и `m2` в функции `changeMap()` технически являются разными указателями на одну и ту же `hmap`.

Каждый из бакетов может содержать не более 8 пар ключ-значение.

В приведенной выше map 2 бакета, а `len(map)` равно 6.

Поэтому, когда вы добавляете пару ключ-значение в map, Go не просто помещает её туда случайным или последовательным образом. Вместо этого он отправляет пару в один из этих бакетов, основываясь на хэш-значении ключа, которое вычисляется с помощью функции `hash(key, seed)`..

Процесс начинается с преобразования слова `"hello"` в число с помощью функции хэширования. Затем это число модифицируется с учётом количества бакетов.

Поскольку у нас есть только один бакет, любое число по модулю 1 становится 0, поэтому оно сразу помещается в бакет 0. То же самое происходит, когда вы

добавляете новую пару ключ-значение. Система попытается разместить её в бакет 0. Если первый слот занят или уже существует ключ с таким значением, то новая пара будет перемещена в следующий свободный слот в этом бакете.

Если вы посмотрите на `hash(key, seed)`, то заметите, что когда вы используете цикл `for-range` над двумя map, у которых одинаковые ключи, то ключи выводятся в разных порядках:

```
func main() {
    a := map[string]int{"a": 1, "b": 2, "c": 3, "d": 4, "e": 5, "f": 6}
    b := map[string]int{"a": 1, "b": 2, "c": 3, "d": 4, "e": 5, "f": 6}

    for i := range a {
        print(i, " ")
    }
    println()

    for i := range b {
        print(i, " ")
    }
}

// Output:
// a b c d e f
// c d e f a b
```

> *Как такое возможно? Разве ключ "a" в `a` и ключ "a" в `b` не хешируются одинаково?*

Хотя хэш-функция, используемая для map в Go, является последовательной для всех map с одним и тем же типом ключа, `seed`, используемый этой хэш-функцией, уникален для каждого отдельного экземпляра map.

Это означает, что при *создании новой map Go генерирует случайный `seed`, который будет использоваться исключительно для этой конкретной map*.

В приведенном выше примере map `a` и `b` используют одну и ту же хэш-функцию, поскольку их ключи относятся к строковым типам, но каждая map имеет свое уникальное seed.

> *Подождите, в бакете всего восемь мест? А что будет, если он полностью*

> *заполнится? Увеличится, как слайс?*

Когда бакеты начинают заполняться или почти заполняются, в зависимости от алгоритма, map активирует процесс роста, который может привести к удвоению количества основных бакетов(main buckets).

У map в go существуют две концепции бакетов: **main buckets** и **overflow buckets**.

Overflow buckets вступают в игру, когда возникает ситуация с большим количеством коллизий. Представьте, что у вас есть 4 бакета, но один из них полностью заполнен 8 парами ключ-значение из-за сильных коллизий.

Стоит ли увеличивать количество бакетов до восьми только для того, чтобы добавить новую запись, которая, к сожалению, также попадает в первый полный бакет?

Конечно, нет. В этой ситуации было бы нерационально увеличивать количество бакетов вдвое.

Вместо этого Go решает эту проблему более эффективно, создавая *overflow buckets*, которые связаны с первым бакетом.

map в go растет, когда выполняется одно из двух условий:

- есть слишком много **переполненных(overflow) бакетов**
- **map перегружена(overloaded)**, то есть **коэффициент загрузки(load factor)** слишком высок.

Из-за этих двух условий существует также два типа роста:

- при перегрузке кол-во бакетов удваивается.
- кол-во бакетов остаётся прежним, но записи перераспределяются, если слишком много переполненных бакетов.

Если число переполненных бакетов слишком велико, то лучше перераспределить записи, чем просто увеличивать объём памяти.

*В Go в настоящее время установлен **коэффициент загрузки (load factor)**, равный `6.5`. Это означает, что map рассчитана на то, **чтобы в среднем хранить 6.5 элементов в каждом бакете**, что составляет примерно 80% от её ёмкости.*

*Когда коэффициент загрузки превышает этот порог, map считается **перегруженной(overloaded)**. В этом случае она будет расти, выделяя новый массив бакетов, вдвое больший, чем текущий, и повторно хэшируя элементы в эти новые бакеты.*

Причина, по которой нам нужно увеличивать map, даже если бакеты заполнены не полностью, сводится к производительности. Обычно мы думаем, что доступ и присвоение значений в map - это O(1), но это не совсем так.

Чем больше мест в бакете занято, тем медленнее он работает.

Когда вы хотите добавить новую пару ключ-значение в бакет, недостаточно просто проверить, есть ли свободное место. Вам необходимо сравнить новый ключ с каждым существующим ключом в этом бакете, чтобы определить, нужно ли добавить новую запись или обновить существующую.

Ситуация становится ещё хуже, когда у вас есть переполненные бакеты, потому что тогда вам придется проверять каждый слот еще и в них. Это замедление также влияет на операции доступа и удаления.

К счастью, в Go есть оптимизация для сравнения ключей.

```
// A bucket for a Go map.
type bmap struct {
    // tophash generally contains the top byte of the hash value
    // for each key in this bucket. If tophash[0] < minTopHash,
    // tophash[0] is a bucket evacuation state instead.
    tophash [abi.OldMapBucketCount]uint8
    // Followed by bucketCnt keys and then bucketCnt elems.
}
```

Вместе со значениями в бакете еще хранятся первые 8 бит от их хэша. Для сравнения не нужно получать хэш каждого ключа или хранить его. Эффективней хранить `uint8` в `tophash` и сравнивать по 8 бит.

После сравнения `tophash`, если они совпадают, это значит, что ключи *могут быть одинаковыми*. Затем Go переходит к более медленному процессу проверки, чтобы убедиться, что ключи действительно идентичны.

> *Почему создание новой map с помощью `make(map, hint)` не дает точного размера, а только подсказку?*

Параметр `hint` в функции `make(map, hint)` сообщает Go начальное количество элементов.

Эта подсказка позволяет минимизировать количество раз, когда map будет увеличиваться по мере добавления новых элементов.

Поскольку каждая операция роста требует выделения нового массива бакетов и копирования существующих элементов, это не самый эффективный процесс. Если же начинать работу с большим начальным объемом, можно избежать некоторых из этих дорогостоящих операций.

> *Почему при hint 14 получается 4 бакета? Нам нужно всего 2 бакета, чтобы покрыть 14 записей*

Именно в этот момент в игру вступает *коэффициент загрузки(load factor)*. Помните, что порог коэффициента загрузки составляет *6,5*? Этот показатель напрямую влияет на то, когда map должна расти.

- При hint равным 13 у нас есть 2 бакета, и это дает коэффициент загрузки 13/2 = 6,5, что соответствует порогу, но не превышает его. А когда hint равным 14, коэффициент загрузки превысит 6,5, что вызовет необходимость роста.
- То же самое относится и к 26. При 4 бакетах коэффициент загрузки составляет 26/4 = 6,5, что опять же соответствует порогу. Когда вы выходите за пределы 26, map должна увеличиться, чтобы эффективно разместить больше элементов.

Evacuation

Эвакуация не всегда означает, что количество бакетов удвоится. Если количество

переполненных бакетов слишком велико, эвакуация всё равно состоится, но новый массив бакетов будет иметь тот же размер, что и предыдущий.

Более интересный сценарий - когда размер бакетов удваивается, поэтому остановимся на нем.

Рост map отвечает на два распространенных вопроса: "*Почему нельзя получить адрес элемента map?*" и "*Почему порядок for-range не гарантируется в разное время?*".

```
func main() {
    a := map[string]int{"a": 1}
    ptr := &a["a"]
}

// compiler error: invalid operation: cannot
// take address of a["a"] (map index expression of type int)
```

Когда map увеличивается, она выделяет новый массив бакетов, который в два раза превышает размер старого. Все позиции записей в старых бакетах становятся недействительными, и их нужно переместить в новые, где они получат новые адреса памяти.

Если в вашей map содержится, например, 1000 пар ключ-значение, то перемещение всех этих ключей за один раз может оказаться довольно дорогостоящей операцией. Это может привести к блокировке вашей программы на значительное время. Чтобы избежать такой ситуации, Go использует механизм *инкрементного роста*, который позволяет перемещать только часть элементов за один раз.

Процесс немного усложняется, потому что нам нужно поддерживать целостность map при чтении, записи, удалении или итерации, управляя при этом старыми и новыми бакетами.

> *Когда происходит инкрементный рост?*

Есть два сценария, при которых происходит инкрементный рост: когда вы добавляете пару ключ-значение или когда удаляете ключ. Любое из этих действий

запускает процесс эвакуации, и как минимум один бакет перемещается в новый массив.

Когда вы делаете что-то вроде `m[«Hello»] = 2`, если map находится в процессе роста, первое, что она делает, - это эвакуирует старый бакет, содержащий ключ “Hello“.

Например, если вы увеличиваете количество бакетов с 4 до 8, элементы из старого бакета 1 переместятся либо в новый бакет 1, либо в новый 5.

Если `hash % 4 == 1`, это означает, что `hash % 8 == 1` или `hash % 8 == 5`. Потому что для старого бакета, где `H % 4 == 1`, два последних бита `H` равны `01`. Когда мы рассматриваем последние три бита для нового массива:

- Если третий бит (справа) равен `0`, то последние три бита равны `001`, что означает `H % 8 == 1`.
- Если третий бит (справа) равен `1`, то последние три бита равны `101`, что означает `H % 8 == 5`.

Если у старого бакета есть overflow бакеты, map также перемещает элементы из них. После того как все элементы перемещены, map помечает старый бакет как “эвакуированное” через поле `tophash`.

Комментарии в Telegram-группе!

go

« ПРЕДЫДУЩАЯ
Быстрый альтернативный способ выполнения операции взятия по модулю(остаток от деления)

СЛЕДУЮЩАЯ »
Слабые Указатели в Go