
che1nov
5 мая в 21:05

Карты (maps) в Go

6 мин 4.3K

Go*, Программирование*

Карты (maps) в Go — это отличный инструмент для хранения данных в виде пар «ключ — значение». Они широко используются в разработке благодаря своей гибкости и удобству. Например, карты часто применяются для кэширования данных, хранения конфигураций или обработки больших объемов информации. Однако эффективная работа с картами требует понимания их внутреннего устройства и особенностей управления памятью. Под капотом карты реализованы на основе хеш-таблиц, что обеспечивает быстрый доступ к данным, но также создает потенциальные проблемы, такие как неэффективное использование памяти или утечки. В этой статье мы разберем устройство карт в Go, рассмотрим, как они растут и работают, а также обсудим способы оптимизации их использования. Особое внимание уделено проблемам, связанным с инициализацией карт и управлением памяти, чтобы помочь вам писать более эффективный и надежный код.

Устройство хеш-таблицы

В Go 1.24 и более поздних версиях карты реализованы с использованием Swiss Tables, что значительно повышает производительность и эффективность использования памяти по сравнению с предыдущими версиями.

1. **Группы (Groups):** Swiss Tables организованы в виде массива групп. Каждая группа состоит из 8 слотов (slots), в которых хранятся пары «ключ — значение».
2. **Контрольные байты (control bytes)**: по 1 байту на слот. Используются для быстрой фильтрации — позволяют понять, находится ли нужный ключ в группе, прежде чем сравнивать значения напрямую.
3. **Хеш-функция:** Хеш-функция играет ключевую роль в работе карт. Она должна быть детерминированной (один и тот же ключ всегда отображается на один и тот же индекс массива) и равномерно распределять ключи по группам, чтобы минимизировать коллизии.
4. **Обработка коллизий:** Коллизии обрабатываются с помощью битовых масок и быстрых пробингов внутри группы. Если в группе нет свободного места, происходит переход к

Для выполнения операций чтения, обновления и удаления Go сначала вычисляет хеш ключа,

чтобы найти соответствующую группу. Затем происходит поиск нужного слота в группе с использованием контрольных слов.

```
// Пример операций с картой
m := map[string]int{"one": 1, "two": 2}
fmt.Println(m["one"]) // Чтение значения
m["two"] = 20         // Обновление значения
delete(m, "one")      // Удаление значения
```

В среднем операции выполняются за O(1) благодаря эффективной хеш-функции и структуре Swiss Tables. Даже в худшем случае производительность остается высокой благодаря оптимизированным алгоритмам пробинга.

Почему важна эффективная инициализация карт?

Инициализация карты играет ключевую роль в оптимизации производительности и использования памяти, так как неправильная настройка может привести к частым перераспределениям памяти. Карта по умолчанию начинается с небольшого количества групп, и при добавлении элементов Go может потребоваться расширение хеш-таблицы. Это требует дополнительных временных и ресурсных затрат, а также может привести к увеличению времени доступа. Большое количество групп и связей между ними может замедлить выполнение операций чтения, обновления и удаления. Наконец, некорректная работа с картами может привести к сохранению ссылок на ненужные данные, усложняя работу сборщика мусора и вызывая потенциальные утечки памяти.

Инициализация карт в Go

Карты (maps) в Go дают много возможностей для хранения данных в виде пар «ключ — значение». Однако их производительность и использование памяти во многом зависят от того, как они инициализируются. Чтобы понять, почему так важна правильная настройка инициализации карт, давайте разберем основы их работы и особенности роста.

Проблемы неэффективной инициализации

Допустим, мы создаем карту с тремя элементами:

```
m := map[string]int{
    "1": 1,
    "2": 2,
    "3": 3,
}
```

Под капотом эта карта использует хеш-таблицу с небольшим количеством групп, которые

содержат эти три элемента. Однако что произойдет, если добавить 1 миллион элементов? Чтобы избежать таких проблем, карта автоматически увеличивается в размере, добавляя новые группы при необходимости.

1. **Коэффициент загрузки** : Среднее количество элементов в сегментах превышает пороговое значение (в текущей реализации Go это 6.5). Этот параметр может измениться в будущих версиях, так как является внутренним.
2. **Переполнение сегментов** : Слишком много сегментов содержат более 8 элементов.

При увеличении карты все ключи перераспределяются по новым группам. В худшем случае вставка нового ключа может занять O(n) времени, где n — общее количество элементов в карте.

Оптимизация инициализации карт

Если заранее известно количество элементов, которые будут добавлены в карту, можно использовать функцию `make` с указанием начального размера:

```
m := make(map[string]int, 1_000_000)
```

Здесь мы подсказываем среде выполнения Go, что карта должна быть готова вместить как минимум 1 миллион элементов. Это позволяет:

1. **Сэкономить процессорное время** : Карте не нужно динамически создавать новые сегменты и перебалансировать их.
2. **Улучшить производительность** : Избегаются дорогостоящие операции по увеличению размера карты.

Важно отметить, что указание начального размера не ограничивает карту этим количеством элементов. Если потребуется, можно добавить больше элементов, чем указано. Однако заданный размер помогает оптимизировать выделение памяти и уменьшить нагрузку на систему.

Сравнение производительности

Рассмотрим два бенчмарка:

1.Без указания начального размера

```
m := make(map[string]int)
for i := 0; i < 1_000_000; i++ {
```

```
    m[fmt.Sprintf("key%d", i)] = i
}
```

```
BenchmarkMapWithoutSize-4   6   227413490 ns/op
```

2.С указанием начального размера

```
m := make(map[string]int, 1_000_000)
for i := 0; i < 1_000_000; i++ {
    m[fmt.Sprintf("key%d", i)] = i
}

// Результат: 91174193 нс/оп
```

```
BenchmarkMapWithSize-4   13   91174193 ns/op
```

Второй вариант работает примерно на **60 % быстрее**, так как предотвращает множественные операции по увеличению размера карты.

Почему важно указывать начальный размер?

Если заранее известно количество элементов, всегда следует указывать начальный размер карты, так как это позволяет избежать частых перераспределений памяти, которые возникают при динамическом увеличении размера хеш-таблицы, сократить время выполнения операций вставки за счет минимизации необходимости создания новых сегментов и перебалансировки данных, а также уменьшить нагрузку на сборщик мусора, который должен очищать временные массивы, образующиеся при расширении карты.

Проблема утечек памяти при работе с картами

Рассмотрим пример, который демонстрирует проблему:

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
```

```
    var m map[int][128]byte
    printMemStats()

    m = make(map[int][128]byte)
    for i := 0; i < 1000000; i++ {
        m[i] = [128]byte{}
    }
    printMemStats()

    for i := 0; i < 1000000; i++ {
        delete(m, i)
    }
    runtime.GC()
    printMemStats()
}

func printMemStats() {
    var m runtime.MemStats
    runtime.ReadMemStats(&m)
    fmt.Printf("Alloc = %v MiB\n", m.Alloc/1024/1024)
}
```

При выполнении этого кода мы наблюдаем следующее:

- **Начальный размер кучи** : ~0 МБ.
- **После добавления 1 миллиона элементов** : ~461 МБ.
- **После удаления всех элементов и вызова GC** : ~293 МБ.

Почему память не освободилась полностью? Причина кроется в том, как устроены карты в Go.

Внутреннее устройство карт

Карта в Go основана на структуре данных хеш-таблицы, которая состоит из групп (groups). После добавления большого количества элементов количество групп остается неизменным даже после удаления всех элементов. Это ключевая особенность: карты в Go могут только увеличиваться в размере, но никогда не уменьшаются.

Рассмотрим реальный сценарий. Предположим, мы создаем кэш с помощью карты `map[int][128]byte` , где каждый клиент представлен ID (ключом), а его данные — массивом из 128 байтов. Если система работает стабильно, например, сохраняя последние 1000 клиентов, проблем с памятью не возникает. Однако во время пиковых нагрузок (например, во время Черной пятницы) карта может вырасти до миллионов элементов. После завершения пиковой нагрузки количество групп останется прежним, что приведет к высокому потреблению памяти даже после удаления элементов.

Возможные решения

Пересоздание карты: регулярно пересоздавать карту, копируя активные элементы в новую карту:

```
newMap := make(map[int][128]byte)
for k, v := range oldMap {
    newMap[k] = v
}
oldMap = nil
runtime.GC()
```

Недостаток в том что, временно требуется в два раза больше памяти.

Использование указателей: Вместо хранения массивов напрямую можно хранить указатели на них:

```
map[int]*[128]byte
```

Это значительно уменьшает потребление памяти, так как каждый сегмент хранит только указатель (8 байт на 64-битных системах вместо 128 байт).

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    printMemStats()
    m := make(map[int]*[128]byte)
    for i := 0; i < 1000000; i++ {
        arr := [128]byte{}
        m[i] = &arr
    }
    printMemStats()

    for i := 0; i < 1000000; i++ {
        delete(m, i)
    }
    runtime.GC()
    printMemStats()
```

```
}

func printMemStats() {
    var m runtime.MemStats
    runtime.ReadMemStats(&m)
    fmt.Printf("Alloc = %v MiB\n", m.Alloc/1024/1024)
}
```

`Шаг`	`map[int][128]byte`	`map[int]*[128]byte`
Создание пустой карты	0 MB	0 MB
Добавление 1 миллиона элементов	461 MB	182 MB
Удаление всех элементов и вызов GC	293 MB	38 MB

Как видно, использование указателей позволяет значительно снизить потребление памяти.

Если ключ или значение превышает 128 байт, Go автоматически хранит указатель на данные, а не сами данные. Это помогает оптимизировать использование памяти.

Подводя итоги

Эффективное использование карт в Go требует четкого понимания их внутреннего устройства и особенностей работы с памятью. Карты, основанные на хеш-таблицах, обеспечивают быстрый доступ к данным, но их неправильная инициализация или управление могут привести к неоптимальному использованию ресурсов. Мы рассмотрели, как правильно задавать начальный размер карты, чтобы избежать частых перераспределений памяти и минимизировать нагрузку на сборщик мусора. Также обсудили проблему утечек памяти, связанную с тем, что карты в Go могут только увеличиваться в размере, и предложили решения, такие как использование указателей или регулярное пересоздание карт. Понимание этих аспектов поможет вам создавать высокопроизводительные приложения, которые эффективно управляют памятью и избегают распространенных ошибок.

Теги: go, golang, map, maps, мапы, мапа

Хабы: Go, Программирование

Электропочта

Оставляя свою почту, я принимаю Политику конфиденциальности и даю согласие на получение рассылок

15 Карма 26 Рейтинг

@che1nov

Разработчик

Подписаться

GitHub Telegram

Комментарии 13

Публикации

ЛУЧШИЕ ЗА СУТКИ ПОХОЖИЕ

HydrAttack
8 часов назад

Как я вскрыл «умный» замок пятью способами за пять минут

7 мин 12K

+95 36 85

Liisign
9 часов назад

CLI в 2025: отголоски динозавров или реальная потребность

Простой 9 мин 2.4K

+33 7 7

is113
5 часов назад

Обзор UserGate WAF: тесты и особенности эксплуатации

Простой 8 мин 922

Обзор

+25 5 0

nnovichOK
10 часов назад

Заблокировать нельзя использовать: кратчайшая история противостояния трекинговых кук и современных браузеров

9 мин 1.9K

+24 17 10

A_Tsymbalyuk
10 часов назад

Что искать на крыше осенью, чтобы вам не пришлось потом платить за ремонт 17 млн. Показываю

Простой 6 мин 4.4K

Кейс

+21 17 3

beget_com

10 часов назад

10 любопытных картографических сервисов для тех, кто не поехал в отпуск

4 мин 2.5K

Дайджест

+19 30 5

AlexeyNadezhin
6 часов назад

Тест настольных светильников dpDUPI серии PRO

Простой 4 мин 715

+18 9 3

popski_ruvds
2 часа назад

Как австралийский хакер сломал канализацию в шире: первая в истории кибератака на физическую инфраструктуру

Простой 7 мин 864

Ретроспектива

+17 1 3

CyberPaul
9 часов назад

NABU. Феномен «канадского интернета 80-х», родившегося задолго до появления интернета

Простой 8 мин 1.4K

Ретроспектива

+17 7 4

ru_vds
8 часов назад

VPS-сервер как платформа для ИИ-агентов

8 мин 990

+16 11 0

Ловкость рук — и приз ваш! Играйте в пинбол, побеждайте, забирайте награды!

Турбо

Показать еще

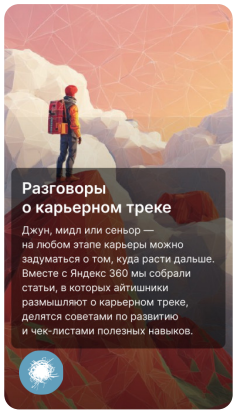
ИСТОРИИ

Чего хотят лиды в бигтехе?

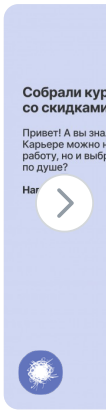
В поисках пятого элемента ИТ

Закрыть вкладки — открыть окно

Годнота от компаний



Как расти в ИТ: советы, гайды и опыт сеньоров



Курсы со до 60%

КУРСЫ

Python-разработчик

По мере набора группы

Фронтенд-разработчик

По мере набора группы

Профессия «Системный аналитик PRO»

По мере набора группы

Аналитик данных

По мере набора группы

Data scientist

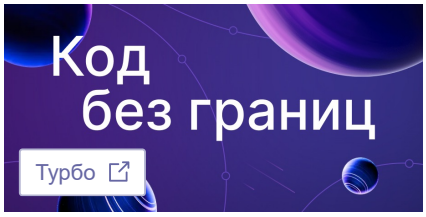
По мере набора группы

Больше курсов на Хабр Карьере

МИНУТОЧКУ ВНИМАНИЯ

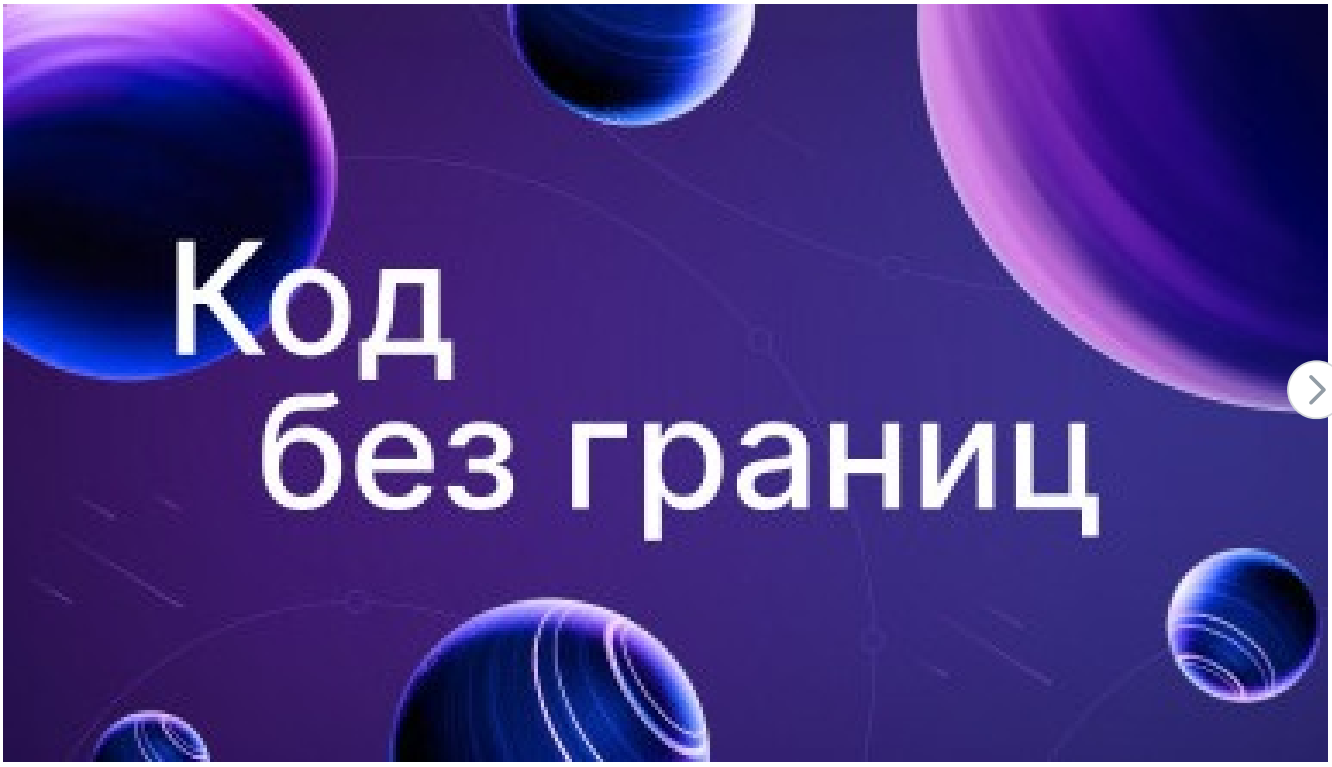
Да начнётся битва: выбираем лучший IT-бренд работодателя

Что ни день – то ивент, если следить за Календарём



Отправь свой open source проект на конкурс и выиграй грант

БЛИЖАЙШИЕ СОБЫТИЯ



3 сентября – 31 октября

Программа грантов для развития open source проектов «Код без границ»

Онлайн

Разработка

Больше событий в календаре

Настройка языка

Техническая поддержка