GO-02 04: Основы языка. Map'ы в Go

Описание:

К этому моменту мы рассмотрели подавляющее большинство реализованных в Go типов данных, и у вас уже мог возникнуть вполне логичный вопрос: «А где же ассоциативные массивы?». В этом разделе мы продолжим изучать типы данных и рассмотрим мапы или, как их еще называют, отображения или хеш-таблицы.
Мапа в Go является удобной и функциональной структурой данных и представляет собой неупорядоченный набор элементов одного типа, называемых значениями, которые связываются с элементами другого типа — ключами. Как и ассоциативные массивы в других языках, мапы в Go используют для хранения связанных данных. Это очень удобно, так как концепция хеш-таблицы, которую реализуют мапы, позволяет очень быстро получить, изменить или удалить значение, зная его ключ.
Объявление и инициализация
Объявление мапы осуществляется при помощи ключевого слова map. В выражении объявления необходимо указать типы для ключей и значений. Для мап существует несколько способов объявлений, как и в случае с другими типами данных.

```
var book map[string]int


var author map[string]string = map[string]string{
  "name":       "Stephen",
  "lastName":    "King"}

reader := map[string]string{
  "name": "John",
  "lastName": "Smith",
  "city": "NY"}
```

У отображения есть размер, который показывает текущее количество элементов. Узнать размер можно при помощи встроенной функции len().

```
months := map[string]int{
  "Jan": 31,
  "Feb": 28,
  "Mar": 31,
}
fmt.Println(len(months))   // 3
```

В качестве ключей могут быть значения любых типов данных, которые можно сравнивать. В противном случае встроенная хеш-функция не смогла бы проверить ключи на равенство. Значения, в отличие от ключей, могут иметь любой тип данных, включая слайсы, мапы и структуры. Например, создадим мапу, в которой будем хранить тип издания и количество единиц с разбивкой по году выпуска.

```
items := map[int]map[string]int{
  2020: {
     "books": 10,
     "periodical": 8,
  },
  2019: {
     "books": 12,
     "periodical": 10,
  },
}
```

Работа с элементами

Доступ к элементам мапы производится по ключу.

```
fmt.Println(items[2020])    // map[books:10 periodical:8]

items[2018] = map[string]int{
  "books": 5,
}
fmt.Println(items)       // map[2018:map[books:5] 2019:map[books:12
periodical:10] 2020:map[books:10 periodical:8]]
```

При попытке доступа к несуществующим элементам в мапе Go не вернет ошибку, и программа продолжит корректно работать. Вместо этого будет возвращено значение по умолчанию для указанного типа значений.

```
book := map[string]int{}
fmt.Println(book["Book1"])    // 0
```

Но часто бывает необходимо проверить, существует ли в отображении элемент с нужным ключом. Как вы уже догадались, указанный в примере выше способ не подходит. Потому что у разных типов данных разные значения по умолчанию, и невозможно определить, например, 0 — это значение по умолчанию для отсутствующего ключа или вполне корректное значение. Это решается при помощи второго возвращаемого значения.

```
book := map[string]int{}
a, ok := book["Book1"]
fmt.Println(a, ok)    // 0 false
```

При попытке присвоить переменной значение из мапы по ключу возвращается два значения: первое — значение ключа или значение по умолчанию, второе — логическое значение, которое показывает, присутствует такой ключ в отображении или нет.
Удалить элемент из мапы можно при помощи встроенной функции delete(). Функция принимает два параметра: первый — отображение, из которого необходимо удалить элемент, второй — ключ.

```
delete(items, 2020)
fmt.Println(items)        // map[2018:map[books:5] 2019:map[books:12
periodical:10]]
```

Отображение является ссылочным типом данных, соответственно, значением по умолчанию является nil.

```
var book map[string]int
fmt.Println(book == nil)   // true
```

При объявлении мапы Go еще не выделяет память для хранения элементов, по этой причине при попытке добавить новые значения мы получим ошибку.

```
var book map[string]int
book["Book1"] = 2     // panic: assignment to entry in nil map
fmt.Println(book)
```

Перед началом работы с мапой ее надо инициализировать — выделить память для элементов.

```
var book map[string]int = map[string]int{}
book["Book1"] = 2
fmt.Println(book)        // map[Book1:2]
```

В отличие от добавления новых элементов, выполнение других операций над неинициализированными мапами не приводит к ошибкам, и они выполняются также, как с пустыми мапами. Пустая мапа — это инициализированная мапа, содержащая в себе 0 элементов.

```
var book map[string]int
reader := map[string]string{}
fmt.Println(book == nil, len(book))        // true 0
fmt.Println(reader == nil, len(reader))    // false 0
```

В предыдущих разделах мы узнали о встроенной функции make(), которая позволяет инициализировать и выделять память для объектов типов слайс, мапа и каналы. Со слайсами и мапами вы уже знакомы, а о каналах мы расскажем в следующих разделах. Функция make() особенно полезна в тех случаях, когда мы изначально знаем количество элементов, которые будут храниться в мапе, и поэтому можем указать необходимый

размер новой мапы. В этом случае мы исключаем дополнительные операции по выделению памяти и копированию всех элементов в нее.

```
book := make(map[string]int, 100)
book["Book1"] = 10
book["Book2"] = 6
```

В отличие от слайса, мапа - более сложная структура, но в ней тоже реализован механизм эвакуации данных в случае аллокации новой памяти при увеличении количества элементов в мапе. Поэтому, если вы заранее знаете число элементов в мапе, то лучше сразу выделить достаточное количество памяти. Это особенно критично при работе с данными больших объемов.

Перебор элементов

Стоит уделить внимание перебору элементов отображения. При переборе мапы не гарантируется, что порядок элементов будет одинаковым. Это сделано для того, чтобы разработчики не полагались на порядок, заложенный в реализацию, так как она может измениться, и разрабатывали устойчивые алгоритмы. Если есть необходимость в постоянном порядке перебора мапы, то можно выделить ключи в отдельный слайс, отсортировать их, например, при помощи функций пакета sort, и затем проитерировать получившийся слайс. Это хороший пример, когда мы знаем максимальное количество элементов и можем заранее задать размер слайса, в котором будут храниться ключи. На примере ниже показан описанный алгоритм перебора мапы с постоянным порядком элементов. Обход элементов реализован через цикл по диапазону, при котором возвращается два значения: первый — ключ, второй — значение. Более подробно с циклами вы познакомитесь в следующем разделе.

```
book := map[string]int{
  "Book2": 2,
  "Book4": 4,
  "Book1": 1,
  "Book3": 3}
```

```
keys := make([]string, 0, len(book))
for k := range book{
  keys = append(keys, k)
}
sort.Strings(keys)

for _, v := range keys {
  fmt.Println(book[v])      // 1 2 3 4
}

fmt.Println(len(keys), cap(keys))  // 4
```

Конкурентный доступ
Об асинхронности и конкурентном доступе вы узнаете в последующих разделах. Сейчас стоит упомянуть о том, что отображение не является безопасной структурой для конкурентного доступа. Это объясняется внутренней реализацией мапы и желанием разработчиков языка сделать Go максимально производительным. Синхронизация операций при конкурентном доступе к мапе может быть свободно реализована при помощи мьютексов, функционал которых представлен в пакете sync, входящем в стандартную поставку языка.

Полезные ссылки:

- Хеш таблицы в Go. Детали реализации
- Go maps in action
- Go: Concurrency Access with Maps

Задание:

1. Создайте в проекте module02 новую ветку 04_task.
2. Создайте в отдельной директории файл main.go. Напишите код, в котором:
- Создайте map, в которой необходимо хранить информацию о выданных читателю печатных изданиях: книгах и периодических изданиях.
- Тип ключей отображения является строкой, тип значений — отображением с ключами типа строка и значениями с типом слайс-строк.
3. Добавьте несколько произвольных значений, моделирующих наличие изданий на руках у читателей.
4. Выведите на экран количество читателей с изданиями на руках.
5. Выведите на экран общее количество изданий на руках у каждого читателя. Для перебора мап используйте приведенный в разделе цикл по диапазону for … range {}
6. Создайте новый коммит с вашим решением и отправьте изменения в удаленный репозиторий проекта.
7. В качестве ответа пришлите ссылку на merge request в ветку master своего проекта ветки 04_task.