Урок Спринт 2.11

Стратегии вытеснения

Random — стратегия вытеснения, при которой удаляются случайные записи.

Эта стратегия вытеснения применена здесь для полноты картины. При решении задачи всегда нужно в первую очередь рассматривать ответ "не будем решать". Такой вариант может привести в некоторых случаях к простым поддерживаемым решениям.

LRU (Least Recently Used) — стратегия вытеснения, которая опирается на время последнего использования записи.

Кеш LRU (Наименее недавно использованный) — это тип кеширования, который поддерживает ограниченный набор элементов и сначала удаляет наименее недавно использованные элементы, когда кеш достигает своего максимального размера. Идея кеширования LRU заключается в том, чтобы отслеживать порядок, в котором осуществляется доступ к элементам или их изменение. Когда кеш заполняется, будем удалять наименее недавно использованный элемент, чтобы освободить место для новых записей. Мы можем реализовать LRU, отбрасывая наименее недавно использованные элементы, когда кеш достигает своего максимального размера, используя комбинацию двусвязного списка и мапы.

Теперь наш кеш будет выглядеть следующим образом:

```
type LRUCache struct {
    capacity int
    cache    map[string]*list.Element
    list     *list.List
    mutex    sync.Mutex
}
```

`capacity` — ёмкость кеша.

`list` — связный список, который будет хранить записи кеша.

При установке значения по ключу есть 2 варианта:

– значение уже есть в кеше. В этом случае перемещаем запись в начало списка:

```
// обновляем существующую запись и перемещаем ее в начало (использовалась в последний раз)
element.Value.(*CacheEntry).value = value
lru.list.MoveToFront(element)
```

– значения нет в кеше. Сохраняем запись в мапу и добавляем её в начало списка:

```
// добавление новой записи в кеш
entry := &CacheEntry{key: key, value: value}
element := lru.list.PushFront(entry)
lru.cache[key] = element
```

– если при этом места в кеше не осталось, то удаляем самую старую запись из мапы и списка:

```
oldest := lru.list.Back()
if oldest != nil {
    delete(lru.cache, oldest.Value.(*CacheEntry).key)
    lru.list.Remove(oldest)
}
```

При получении значения по ключу каждый раз будем перемещать элемент в начало списка (говоря тем самым, что это самое недавно использованное значение)

```
if element, ok := lru.cache[key]; ok {
    // перемещение доступного элемента в начало списка (последний раз использованный)
    lru.list.MoveToFront(element)
    return element.Value.(*CacheEntry).value, true
}
```

Полный код кеша приведен ниже:

```
package main

import (
    "container/list"
    "fmt"
    "sync"
)

// создание LruCache, представляющего кеш LRU
type LRUCache struct {
    capacity int
    cache    map[string]*list.Element
    list     *list.List
    mutex    sync.Mutex
}

// создание записи кеша, представляющей запись в кеше LRU
type CacheEntry struct {
    key   string
    value interface{}
```

```
}

// NewLRUCache создает новый экземпляр LruCache с указанной ёмкостью
func NewLRUCache(capacity int) *LRUCache {
    return &LRUCache{
        capacity: capacity,
        cache:    make(map[string]*list.Element),
        list:     list.New(),
    }
}

// извлечение значения, связанного с данным ключом, из кеша
func (lru *LRUCache) Get(key string) (interface{}, bool) {
    lru.mutex.Lock()
    defer lru.mutex.Unlock()

    if element, ok := lru.cache[key]; ok {
        // перемещение доступного элемента в начало списка (последний раз использованный)
        lru.list.MoveToFront(element)
        return element.Value.(*CacheEntry).value, true
    }

    return nil, false
}

// добавление или обновление пары ключ-значение в кеше
func (lru *LRUCache) Set(key string, value interface{}) {
    lru.mutex.Lock()
    defer lru.mutex.Unlock()

    // проверка того, существует ли ключ уже в кэше
    if element, ok := lru.cache[key]; ok {
        // обновляем существующую запись и перемещаем ее в начало (использовалась в последний ра
        element.Value.(*CacheEntry).value = value
        lru.list.MoveToFront(element)
    } else {
        // добавление новой записи в кеш
        entry := &CacheEntry{key: key, value: value}
        element := lru.list.PushFront(entry)
        lru.cache[key] = element

        // проверяем, заполнен ли кеш (есть ли место), при необходимости удаляем наименее недавн
        if lru.list.Len() > lru.capacity {
            oldest := lru.list.Back()
            if oldest != nil {
                delete(lru.cache, oldest.Value.(*CacheEntry).key)
                lru.list.Remove(oldest)
            }
        }
    }
}

// PrintCache, который печатает текущее содержимое кеша
```

```
func (lru *LRUCache) PrintCache() {
    lru.mutex.Lock()
    defer lru.mutex.Unlock()

    fmt.Printf("LRU Cache (Capacity: %d, Size: %d): [", lru.capacity, lru.list.Len())
    for element := lru.list.Front(); element != nil; element = element.Next() {
        entry := element.Value.(*CacheEntry)
        fmt.Printf("(%s: %v) ", entry.key, entry.value)
    }
    fmt.Println("]")
}

func main() {
    // создание кэша LRU ёмкостью 3
    lruCache := NewLRUCache(3)

    // задаём пары ключ-занчение
    lruCache.Set("company", "Yandex")
    lruCache.Set("division", "Yandex Lyceum")
    lruCache.Set("course", "Golang")

    lruCache.PrintCache()

    if value, ok := lruCache.Get("company"); ok {
        fmt.Println("Value for company:", value)
    } else {
        fmt.Println("company not found in the cache.")
    }

    // установка дополнительных пар ключ-значение для инициирования вытеснения
    lruCache.Set("year", 2024)
    lruCache.Set("age", "13-17yrs")

    lruCache.PrintCache()
}
```

Обратим внимание на то, что, инвертировав условие, мы можем хранить в кеше самые редко запрашиваемые значения. Этот подход называется MRU — most recently used.

LFU (Least Frequently Used) — стратегия вытеснения, опирающаяся на частоту использования записи.

Реализация алгоритма будет вынесена как задача к данному уроку. Давайте разберёмся, что такое LFU. Суть этого алгоритма состоит в том, чтобы записывать частоту доступа к каждому элементу. В этом случае когда кеш заполняется — будем удалять данные с наименьшей частотой. Для того, чтобы найти данные для удаления заведём отдельную карту частот использования элементов.

Справка