Урок Спринт 2.11

TTL

Мы уже отмечали, что одна из проблем кеширования — сохранение актуальности данных. Эту проблему призван решить механизм TTL. Кеш TTL (Time-To-Live) — это тип механизма кеширования, который связывает продолжительность времени с каждым кешированным элементом, и элементы автоматически признаются недействительными и удаляются из кеша по истечении определенного периода, известного как `time-to-live`. Давайте реализуем простой TTL-кеш.

До этого мы хранили в кеше строки, представляя их как `interface{}`. Предлагается ввести структуру, которая будет олицетворять одну запись в кеше, которая будет хранить значение и время жизни записи.

```
// создание записи кеша, представляющей запись в TTLCache со значением и временем истечения срок
type CacheEntry struct {
    value      interface{} // значение, связанное с записью в кеше
    expiration time.Time   // время истечения срока действия записи в кеше
}
```

Структура кеша теперь будет выглядеть так:

```
// создание TTLCache в качестве потокобезопасного кеша с функциональностью time-to-live
type TTLCache struct {
    data  map[string]CacheEntry // мапа для хранения пар ключ-значение со временем истечения сро
    mutex sync.RWMutex          // мьютекс для синхронизации конкурентного доступа к кешу
}
```

При получении значения по ключу теперь будем учитывать время жизни записи:

```
entry, ok := c.data[key]
if !ok || time.Now().After(entry.expiration) {
    // если ключ не найден или срок действия истёк, возвращаем nil и false
    return nil, false
}
```

Посмотрим на полный код реализации TTLCache:

```
package main

import (
```

```
    "fmt"
    "sync"
    "time"
)

// создание записи кеша, представляющей запись в TTLCache со значением и временем истечения срок
type CacheEntry struct {
    value      interface{} // значение, связанное с записью в кеше
    expiration time.Time   // время истечения срока действия записи в кеше
}

// создание TTLCache в качестве потокобезопасного кеша с функциональностью time-to-live
type TTLCache struct {
    data  map[string]CacheEntry // мапа для хранения пар ключ-значение со временем истечения сро
    mutex sync.RWMutex          // мьютекс для синхронизации конкурентного доступа к кешу
}

// создание NewTTLCache нового экземпляра TTLCache с инициализированной картой данных
func NewTTLCache() *TTLCache {
    return &TTLCache{
        data: make(map[string]CacheEntry),
    }
}

// извлечение значения, связанного с данным ключом, из кеша
// Get() возвращает значение и признак, указывающий, был ли ключ найден и не истёк ли срок его д
func (c *TTLCache) Get(key string) (interface{}, bool) {
    c.mutex.RLock()         // получение блокировки чтения, чтобы разрешить одновременное чтение
    defer c.mutex.RUnlock() // снятие блокировки чтения, когда функция завершит работу
    entry, ok := c.data[key]
    if !ok || time.Now().After(entry.expiration) {
        // если ключ не найден или срок действия истёк, возвращаем nil и false
        return nil, false
    }
    return entry.value, true
}

// установка значения, связанного с данным ключом в кеше
// Set() получает блокировку на запись для обеспечения эксклюзивного доступа во время обновления
func (c *TTLCache) Set(key string, value interface{}, ttl time.Duration) {
    c.mutex.Lock()         // получение блокировки на запись для эксклюзивного доступа
    defer c.mutex.Unlock() // снятие блокировки записи при завершении работы функции
    c.data[key] = CacheEntry{
        value:      value,
        expiration: time.Now().Add(ttl),
    }
}

func main() {
    ttlCache := NewTTLCache()

    // установка пар ключ-значение в кеше TTL с разным временем
    ttlCache.Set("company", "yandex", 5*time.Second) // ttl 5 seconds
```

```
    ttlCache.Set("year", 2024, 10*time.Second)       //ttl 10 seconds

    // извлечение значений из кеша TTL
    if value, ok := ttlCache.Get("company"); ok {
        fmt.Println("Value for company:", value)
    } else {
        fmt.Println("company not found in the TTL cache.")
    }

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

    // создание ожидания с помощью функции Sleep() для имитации использования кеша
    // изменяем время ожидания на другое количество секунд, чтобы увидеть эффект
    time.Sleep(7 * time.Second)

    // повторное извлечение значений через некоторое время
    if value, ok := ttlCache.Get("company"); ok {
        fmt.Println("Value for company (after some time):", value)
    } else {
        fmt.Println("company not found in the TTL cache after some time.")
    }

    if value, ok := ttlCache.Get("year"); ok {
        fmt.Println("Value for year (after some time):", value)
    } else {
        fmt.Println("year not found in the TTL cache after some time.")
    }
}
```

Это хорошее простое решение, которое имеет естественные ограничения. Записи с истёкшим временем жизни не удаляются и висят в оперативной памяти. Если данные периодически устанавливаются с новым TTL, то проблемы нет. Но что делать, если данные с таким ключом больше никогда не будут установлены? Кеш будет неограниченно расти и займёт весь доступный объём оперативной памяти.

Давайте улучшим это решение. Добавим к кешу горутину, которая будет раз в определённый период удалять значения, которые больше не нужны.

Для начала немного улучшим структуру, которая хранит значение кеша. Проассоциируем с ней метод, который удаляет значение:

```
type CacheEntry struct {
    value    interface{}
    expireAt int64
}

func NewCacheEntry(value interface{}, expireAt int64) CacheEntry {
    return CacheEntry{
```

```
        value:    value,
        expireAt: expireAt,
    }
}

func (ce CacheEntry) IsExpired() bool {
    return ce.expireAt < time.Now().UnixNano()
}
```

Сам кеш теперь будет выглядеть следующим образом:

```
type Cache struct {
    kvstore  map[string]CacheEntry
    locker   sync.RWMutex
    interval time.Duration
    stop     chan struct{}
}
```

`interval` — период, в который мы будем добавлять запускать очистку кеша.

Проассоциируем с кешем функцию удаления данных:

```
func (c *Cache) purge() {
    c.locker.Lock()
    defer c.locker.Unlock()
    for key, data := range c.kvstore {
        if data.IsExpired() {
            delete(c.kvstore, key)
        }
    }
}
```

Проассоциируем функцию очистки данных с кешем:

```
fmt.Println("cleaner starting...")
ticker := time.NewTicker(c.interval)
fmt.Println("cleaner was started")
for {
    select {
    case <-ticker.C:
        c.purge()
    case <-c.stop:
        ticker.Stop()
        fmt.Println("cleaner was stopped")
        return
    }
}
```

Возможно, с тикерами вы познакомитесь в следующих уроках. На текущем этапе достаточно понимать, что `ticker.C` — это канал, куда будет приходить значение времени раз в секунду. И соответственно, раз в секунду будет запускаться функция `purge()`. При закрытии канала `stop` мы выйдем из функции.

Полный код кеша с периодической очисткой значаний с истекшим TTL представлен ниже:

```
package main

import (
    "fmt"
    "sync"
    "time"
)

type CacheEntry struct {
    value    interface{}
    expireAt int64
}

func NewCacheEntry(value interface{}, expireAt int64) CacheEntry {
    return CacheEntry{
        value:    value,
        expireAt: expireAt,
    }
}

func (ce CacheEntry) IsExpired() bool {
    return ce.expireAt < time.Now().UnixNano()
}

type Cache struct {
    kvstore  map[string]CacheEntry
    locker   sync.RWMutex
    interval time.Duration
    stop     chan struct{}
}

func NewCache(cleanUpInterval time.Duration) *Cache {
    cache := &Cache{
        kvstore:  make(map[string]CacheEntry),
        interval: cleanUpInterval,
        stop:     make(chan struct{}),
    }

    if cleanUpInterval > 0 {
        go cache.cleaning()
    }
    return cache
}

func (c *Cache) cleaning() {
    fmt.Println("cleaner starting...")
    ticker := time.NewTicker(c.interval)
    fmt.Println("cleaner was started")
    for {
        select {
        case <-ticker.C:
```

```
            c.purge()
        case <-c.stop:
            ticker.Stop()
            fmt.Println("cleaner was stopped")
            return
        }
    }
}

func (c *Cache) purge() {
    c.locker.Lock()
    defer c.locker.Unlock()
    for key, data := range c.kvstore {
        if data.IsExpired() {
            delete(c.kvstore, key)
        }
    }
}

func (c *Cache) Set(key string, value interface{}, expiryDuration time.Duration) {
    expireAt := time.Now().Add(expiryDuration).UnixNano()
    c.locker.Lock()
    defer c.locker.Unlock()
    c.kvstore[key] = NewCacheEntry(value, expireAt)

}

func (c *Cache) Get(key string) (interface{}, bool) {
    c.locker.RLock()
    defer c.locker.RUnlock()
    data, found := c.kvstore[key]
    if !found || data.IsExpired() {
        return nil, false
    }

    return data.value, true
}

func (c *Cache) Close() {
    close(c.stop)
}

func main() {
    cache := NewCache(time.Second)
    defer cache.Close()
    cache.Set("foo", "bar", 2*time.Second)
    for i := 0; i < 3; i++ {
        value, found := cache.Get("foo")
        if found {
            fmt.Println("value for key foo is ", value)
        } else {
            fmt.Println("value for key foo is not found")
            break
```

```
        }

        fmt.Println("waiting for 1 second...")
        time.Sleep(time.Second)
    }
}
```

Вы можете заметить, что для очистки кеша нужно пройтись по всем элементам мапы. Это не очень эффективно. С более эффективными структурами данных для кеширования вы познакомитесь в следующих уроках.