GO-05 04: Асинхронность. Пакеты sync и atomic

Описание:

В стандартной библиотеке golang есть пакет sync, содержащий средства синхронизации, такие как: Mutex, RWMutex и WaitGroup.
Mutex (с ним мы немного познакомились в предыдущей главе) и RWMutex реализуют интерфейс Locker (также описан в пакете sync).
RWMutex используется в случаях, когда некоторый участок памяти может использоваться только на чтение (но не на запись). Для этого у RWMutex есть методы RLock() и RUnlock().
Пример:

```
package main

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

type MutexMap struct {
	storage map[string]float64
	mu      sync.RWMutex
}

func NewStorage(initStorage map[string]float64) *MutexMap {
	if initStorage != nil {
		return &MutexMap{
			storage: initStorage,
		}
	}
	return &MutexMap{
		storage: make(map[string]float64),
	}
}

func (m *MutexMap) GetValue(key string) float64 {
	m.mu.RLock()
```

```
	defer m.mu.RUnlock()
	return m.storage[key]
}

func (m *MutexMap) SetValue(key string, value float64) {
	m.mu.Lock()
	defer m.mu.Unlock()
	m.storage[key] = value
}

func (m *MutexMap) IncreaseValue(key string, value float64) {
	m.mu.Lock()
	defer m.mu.Unlock()
	m.storage[key] += value
}

func (m *MutexMap) GetKeys() []string {
	m.mu.RLock()
	defer m.mu.RUnlock()
	keys := make([]string, 0, len(m.storage))
	for k := range m.storage {
		keys = append(keys, k)
	}
	return keys
}

func (m *MutexMap) Print() {
	m.mu.RLock()
	defer m.mu.RUnlock()
	for k, v := range m.storage {
		fmt.Printf("%s:%v\n", k, v)
	}
}

func main() {
	m := NewStorage(map[string]float64{
		"Alex":  10.0,
		"Paul":  40.0,
		"Frank": 15.0,
	})
```

```
m.Print()

for i := 0; i < 5; i++ {
      go func() {
            for _, key := range m.GetKeys() {
                  time.Sleep(time.Millisecond * 10)
                  m.IncreaseValue(key, 1)
            }
      }()
}

time.Sleep(time.Second)
      fmt.Println()
      m.Print()
}
```

Вывод:

```
Alex:10
Paul:40
Frank:15

Alex:15
Paul:45
Frank:20
```

Обратите внимание: если использовать вместо `m.IncreaseValue(key, 1)` функцию `m.SetValue(key, m.GetValue(key)+1)`, то числа в выводе могут отличаться.

В предыдущих программах мы использовали функцию time.Sleep(), чтобы все горутины успели закончить выполнение. Однако этот способ не надежен, так как завязан на время (которое, на самом деле, может зависеть от разных факторов). Поэтому лучше использовать тип sync.WaitGroup.

Перепишем функцию main, добавив WaitGroup:

```
func main() {
      m := NewStorage(map[string]float64{
            "Alex":  10.0,
            "Paul":  40.0,
```

```
		"Frank": 15.0,
	})
	m.Print()
	fmt.Println()

	var wg sync.WaitGroup

	for i := 0; i < 5; i++ {
		wg.Add(1)  // добавляем по 1 на каждую горутину
		go func() {
			defer wg.Done() // вызываем Done, когда горутина
закончит выполнение
			for _, key := range m.GetKeys() {
				time.Sleep(time.Millisecond * 10)
				m.IncreaseValue(key, 1)
			}
		}()
	}

	wg.Wait() // ждем окончания работы всех горутин
	fmt.Println()
	m.Print()
}
```

Функция Add увеличивает счетчик на указанное число, функция Done уменьшает его на 1. Функция Wait блокирует выполнение горутины до тех пор, пока счетчик WaitGroup не станет 0.
Обратите внимание, что для самой WaitGroup не надо создавать отдельный мьютекс.
В пакете sync присутствует специальный тип - Map, реализующий хранилище вида «ключ : значение» с встроенным мьютексом. Его можно использовать вместо связки map + sync.Mutex, однако у этого есть особенности:

- sync.Map используется в качестве ключей и значений интерфейсов, поэтому придется делать преобразование типов.
- sync.Map более эффективен, чем map + sync.Mutex, только начиная с некоторого количества чтений в секунду при количестве ядер не меньше 4.

Пакет atomic (sync/atomic), на котором построена большая часть пакета sync, предоставляет низкоуровневые примитивы для работы с памятью. Этот пакет оперирует в основном с целыми числами и указателями на них.
Кроме отдельных случаев низкоуровневых программ, рекомендуется использовать не пакет atomic, а компоненты пакета sync или каналы, так как они более оптимизированы и предназначены для этого.

Полезные ссылки:

- package sync
- package atomic
- Разбираемся с новым sync.Map в Go 1.9

Задание:

1. Создать у себя в проекте module05 ветку module05_04.
2. Перепишите класс Cache из предыдущего задания, используя sync.RWMutex.
3. Замените вашу функцию main() и добавьте константу из кода, который предложен далее:

```
const (
	k1   = "key1"
	step = 7
)

func main() {
	cache := Cache{storage: make(map[string]int)}

	for i := 0; i < 10; i++ {
		go func() {
			cache.Increase(k1, step)
			time.Sleep(time.Millisecond * 100)
		}()
	}

	for i := 0; i < 10; i++ {
		i := i // copy variable
		go func() {
			cache.Set(k1, step*i)
			time.Sleep(time.Millisecond * 100)
		}()
	}

	fmt.Println(cache.Get(k1))
}
```

4. Модифицируйте код, использующий класс Cache, из прошлого задания так, чтобы все горутины завершали свое выполнение после отработки (используйте для этого sync.WaitGroup).

Обратите внимание на строчку i := i // copy variable. Без нее возможно возникновение ошибок, так как горутина будет использовать переменную i, которая на тот момент уже будет иметь значение, отличающееся от того, что было в момент создания горутины. Обычно для этого используют переменные с таким же названием (i := i), для того чтобы не плодить имена. Однако область видимости позволяет таким образом создавать новые переменные и взаимодействовать с ними, соответственно.

Еще один способ избежать такой ошибки - использовать параметры функции, например:

```
for i := 0; i < 10; i++ {
        go func(i int) {
                cache.Set(k1, step*i)
                time.Sleep(time.Millisecond * 100)
        }(i)
}
```

5. В ответе пришлите ссылку на MP в ветку master своего проекта ветки module05_04.