

Backend Golang

Урок 7

Шаблоны синхронизации (errgroup, semaphore, rwmutex).
Работа с HTTP: простой сервер

Вспомним базовые механизмы синхронизации

- `sync.WaitGroup` — ожидание завершения горутин
- `sync.Mutex` — взаимоисключающая блокировка ресурса
- Каналы — средство передачи данных между горутинами (и не только)
- `select` — ожидание чтения из и записи в каналы

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    var (
        counter int
        wg       sync.WaitGroup
    )
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            counter++
            wg.Done()
        }()
    }
    wg.Wait()
    fmt.Println(counter)
}
```

Пример со счетчиком

Использование sync.Mutex

https://play.golang.org/p/_GDD-yQsMiq

sync.RWMutex

Зачастую доступ к данным — неравномерный в плане чтения и записи.

Одни горутины пишут в общую память, другие — читают из нее.

<https://play.golang.org/p/M9b4BcgxedK>

sync.Map

Словарь с оптимизированным конкурентным доступом

sync.Map “из коробки” реализует API для работы с map+RWMutex:

<https://play.golang.org/p/EVP8vRVC-Sz>

```
type Map struct{ /**/ }

func (m *Map) Delete(key interface{})
func (m *Map) Load(key interface{}) (value interface{}, ok bool)
func (m *Map) LoadOrStore(key, value interface{}) (actual interface{}, loaded bool)
func (m *Map) Range(f func(key, value interface{}) bool)
func (m *Map) Store(key, value interface{})
```

Задание 1

`sync.Map` — нетипизированный словарь. Необходимо написать обертку над `sync.Map`, которая в качестве ключей использует `int`, а в качестве значений — `string`.

https://play.golang.org/p/_QvEpgqX0-z

Обработка ошибок

1. <https://play.golang.org/p/cAK1br5J6J2>
2. <https://play.golang.org/p/qe6OjcAcgwF>
3. <https://play.golang.org/p/PnpyfIGhE7O>
4. <https://play.golang.org/p/YvHVdR98HYc>

errgroup.Group

Группа горутин, которые выполняют подзадачи.

```
type Group struct {  
    cancel func()  
  
    wg sync.WaitGroup  
  
    errOnce sync.Once  
    err      error  
}  
  
func (g *Group) Go(f func() error) {}  
  
func (g *Group) Wait() error {}
```

errgroup.Group

Первая возникшая ошибка, будет возвращена при вызове метода

`Wait() error.`

<https://play.golang.org/p/lzDi7QQQim9>

Что делать при выполнении тяжеловесных задач?

<https://play.golang.org/p/JRRjP6Z30ru>

```
func do() {  
    for i := 0; i < 10; i++ {  
        heavyOperation()  
    }  
    fmt.Println("all done")  
}  
  
func heavyOperation() {  
    for i := 0; i < 1e8; i++ {  
        _ = strconv.Itoa(i)  
    }  
    fmt.Println("done")  
}
```

Пускаем горютины

<https://play.golang.org/p/nMg-oTQahmR>

```
func do() {  
    var wg sync.WaitGroup  
  
    for i := 0; i < 10; i++ {  
        wg.Add(1)  
        go func() {  
            defer wg.Done()  
            heavyOperation()  
        }()  
    }  
  
    wg.Wait()  
    fmt.Println("all done")  
}
```

Запуск неограниченного количества горутин

- Увеличили скорость работы
- Забрали все ресурсы

Semaphore

Семафор — примитив синхронизации работы процессов и потоков, в основе которого лежит счётчик, над которым можно производить две атомарные операции:

- увеличение значения на единицу
- уменьшение значения на единицу.

Вычислительные семафоры используются для контроля над ограниченными ресурсами.

Semaphore на базе каналов

<https://play.golang.org/p/7RrjvDNPGEu>

```
func do() {  
    var wg sync.WaitGroup  
    sm := make(chan struct{}, 3)  
  
    for i := 0; i < 10; i++ {  
        wg.Add(1)  
        sm <- struct{}{}  
        go func() {  
            defer func() { <-sm }()  
            defer wg.Done()  
            heavyOperation()  
        }()  
    }  
  
    wg.Wait()  
    fmt.Println("all done")  
}
```

Задание 2

Реализовать семафор в виде структуры:

1. с использованием каналов
2. с использованием счетчика + `sync.Mutex`

<https://play.golang.org/p/lxfcD1tV-w->

```
// Semaphore - TODO: implement
type Semaphore struct {
}

func NewSemaphore(size int) *Semaphore { panic("implement me") }

func (s *Semaphore) Acquire(n int) { panic("implement me") }

func (s *Semaphore) Release(n int) { panic("implement me") }
```


HTTP-сервер

<https://play.golang.org/p/632qrbbmeul>

```
package main

import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(writer http.ResponseWriter, request
    *http.Request) {
        msg := fmt.Sprintf("requested path: %s", request.RequestURI)
        _, _ = writer.Write([]byte(msg))
    })
    _ = http.ListenAndServe(":8080", nil)
}
```

Раздача статики

```
package main

import (
    "net/http"
)

func main() {
    http.HandleFunc("/", func(writer http.ResponseWriter, request *http.Request) {
        http.ServeFile(writer, request, "main.go")
    })
    _ = http.ListenAndServe(":8080", nil)
}
```

Context

Механизм, который выполняет две задачи:

1. Прерывание выполнения операций (например, запроса)
2. Передача специфичных данных в виде ключей и значений

```
// A Context carries a deadline, a cancellation signal, and other  
values across  
// API boundaries.  
//  
// Context's methods may be called by multiple goroutines  
simultaneously.  
type Context interface {  
    Deadline() (deadline time.Time, ok bool)  
  
    Done() <-chan struct{}  
  
    Err() error  
  
    Value(key interface{}) interface{}  
}
```

Применение контекста

```
package main

import (
    "fmt"
    "time"
)

func main() {
    doSomething()
}

func doSomething() {
    fmt.Println("started doing something")
    time.Sleep(time.Second * 3)
    fmt.Println("done something")
}
```

Прерывание выполнения при вводе с клавиатуры

<https://play.golang.org/p/J-DZzZh1Nma>

```
func main() {  
    ctx := context.Background()  
    ctx, cancelFunc :=  
context.WithCancel(ctx)  
  
    go func() {  
        fmt.Scanln()  
        cancelFunc()  
    }()  
  
    doSomething(ctx)  
}
```

```
func doSomething(ctx context.Context) {  
    fmt.Println("started doing something")  
    select {  
    case <-time.After(time.Second * 3):  
        fmt.Println("done something")  
    case <-ctx.Done():  
        fmt.Println("execution cancelled")  
    }  
}
```

Определить самый быстрый URL

<https://play.golang.org/p/h6DJxB1Rw2>

Heavy operation again...

<https://play.golang.org/p/SJLJdz4QKGU>

```
func main() {  
    http.HandleFunc("/", func(writer http.ResponseWriter, request  
*http.Request) {  
        do()  
        _, _ = writer.Write([]byte("all done"))  
    })  
    _ = http.ListenAndServe(":8080", nil)  
}
```

Прекращение обработки запроса

<https://play.golang.org/p/RrDHQv3DABZ>

Выводы

Каналы используйте для:

- передачи данных
- распределенных вычислений
- передачи асинхронных результатов

Мьютексы используйте для:

- реализации кэшей (но лучше `sync.Map`)
- изменения состояние общей памяти

Выводы

Семафоры используйте для:

- Ограничения количества запущенных горутин

Контекст используйте для:

- прерывания выполнения чего-либо (системных вызовов, сетевых запросов)
- передачи специфичных данных

Вопросы?

ДЗ

- почитать про каналы: <https://go101.org/article/channel.html>,
<https://go101.org/article/channel-closing.html>
- посмотреть: <https://youtu.be/rDRa23k70CU>, <https://youtu.be/5zXAHh5tJqQ>,
<https://youtu.be/f6kdp27TYZs>
- Посмотреть, как сделать graceful shutdown

Д3 — Rate-limited tcp server with graceful shutdown

Реализовать TCP-сервер, который возводит переданное ему число в квадрат и возвращает результат. Количество обрабатываемых запросов в один момент времени должно быть настраиваемым. Должен быть предусмотрен graceful shutdown (перед завершением программы необходимо обработать все открытые соединения).

Написать клиент для тестирования сервера.