GO-05 05: Асинхронность. Каналы ч.1. Deadlocks

Описание:

Вы уже знакомы с таким типом golang, как каналы (chan). Настало время взглянуть на них более внимательно. Каналы бывают буферизированные и небуферизированные. Создаются каналы с помощью функции make (по умолчанию канал имеет значение nil). Чтение из канала производится операцией <-ch, запись - операцией ch <-. Закрыть канал можно с помощью функции close. Запись в закрытый канал вызовет панику. Чтение из закрытого канала вернет значение по умолчанию и false в качестве второго аргумента. len(ch) показывает, сколько элементов сейчас содержится в канале, а cap(ch) показывает его емкость

```
ch1 := make(chan float64) // небуферизированный, для float64
fmt.Println(len(ch1), cap(ch1)) // 0, 0
ch2 := make(chan int, 3) // буферизированный, ёмкостью 3 для int
fmt.Println(len(ch2), cap(ch2)) // 0, 3
ch2 <- 4
fmt.Println(len(ch2), cap(ch2)) // 1, 3

close(ch1)
result, ok := <- ch1
fmt.Println(result, ok) // 0, false
result, ok = <- ch2
fmt.Println(result, ok) // 4, true
```

Операции чтения и записи в канал являются блокирующими. Это значит следующее:

1. Когда горутина подходит к выполнению записи или чтения в канал, она блокируется и проверяет возможность действия.
2. Если это небуферизированный канал, то запись в него возможна, только если другая горутина его читает (и если он еще не был закрыт).
3. Если это небуферизированный канал, то чтение из него возможно, только если другая горутина в него пишет (или он уже закрыт).
4. Следовательно, неверно рассматривать канал как «трубу с сообщениями», хотя для объяснения часто приводят эту аналогию.
5. Если это буферизированный канал, то запись в него возможна, только если len(ch) < cap(ch) (количество элементов в нем меньше максимального, канал не заполнен).

6. Если это буферизированный канал, то чтение из него возможно, только если в нем содержится хотя бы один элемент.

В исходном файле go/src/runtime/chan.go описана структура канала. Канал состоит из:

- счетчика текущих элементов;
- значения емкости канала;
- указателя на массив элементов;
- значения размера элемента;
- значения «канал закрыт»;
- указателя на тип элемента;
- индекса отправки;
- индекса получения;
- списка горутин, читающих канал;
- списка горутин, пишущих в канал;
- мьютекса.

```
type hchan struct {
	qcount   uint           // total data in the queue
	dataqsiz uint           // size of the circular queue
	buf      unsafe.Pointer // points to an array of dataqsiz
elements
	elemsize uint16
	closed   uint32
	elemtype *_type // element type
	sendx    uint   // send index
	recvx    uint   // receive index
	recvq    waitq  // list of recv waiters
	sendq    waitq  // list of send waiters
	lock     mutex
}
```

При попытке выполнить следующий код:

```
package main
import "fmt"

func main()  {
	r := make(chan float64)
	r <- 1.2
	val, ok := <-r
	fmt.Println(val, ok)
}
```

Мы получим ошибку вида:

```
fatal error: all goroutines are asleep - deadlock!
```

Это ситуация «взаимной блокировки» (deadlock). Такое случается, когда горутина заблокирована и не может освободиться от блокировки. В данном случае это произошло, потому что у нас одна горутина, которая встретила небуферизированный канал и не может действовать дальше. Перепишем этот код, чтобы другая горутина читала из канала и тем самым разблокировала основную:

```
package main

import "fmt"

func main() {
      r := make(chan float64)
      go func() {
            val, ok := <-r
            fmt.Println(val, ok)
      }()

      r <- 1.2
    fmt.Println("success")
}
```

В итоге получим закономерное:

```
1.2 true
success
```

и отсутствие ошибок.

Deadlock возможен и при работе с мьютексами. Например, если мы попытаемся закрыть (lock) уже закрытый мьютекс в рамках одной горутины.

Классическим примером deadlock является «задача об обедающих философах». Формулируется она так: за круглым столом сидят пять философов, каждый из которых имеет перед собой тарелку со спагетти, слева от тарелки у каждого лежит вилка. Таким образом на столе лежат пять тарелок и пять вилок. Каждый философ может либо есть, либо размышлять. Чтобы есть, нужно две вилки. Отбирать вилки нельзя. Задача состоит в том, чтобы каждый философ мог и подумать, и поесть (то есть надо правильно распределить контроль над ресурсами).

Одним из возможных средств решения «задачи об обедающих философах» являются семафоры. В golang семафор можно реализовать с помощью буферизированных каналов.

```
package main

import (
```

```
	"fmt"
	"time"
)

func main() {
	semaphore := make(chan int, 3)

	for i := 0; i < 10; i++ {
		semaphore <- i
		go func() {
			defer func() {
				msg := <-semaphore
				fmt.Println(msg)
			}()
			time.Sleep(time.Millisecond * 1000) // some operations
		}()
	}
	for len(semaphore) > 0 {
		time.Sleep(time.Millisecond * 10)
	}
}
```

Полезные ссылки:

- Дейкстра, Эдсгер Вибе
- Задача об обедающих философах
- Семафор (программирование)

Задание:

1. Создайте у себя в проекте module05 ветку module05_05.
2. Используя код предыдущего задания, перепишите основную функцию так, чтобы одновременно использовалось не более 4 дополнительных горутин.
3. Перепишите код без использования sync.WaitGroup.
4. В ответе пришлите ссылку на MP в ветку master своего проекта ветки module05_05.