

GO-05 03: Асинхронность. Race condition

Описание:

В многопоточных программах возможно возникновение так называемого «состояния гонок» (Race condition). Это состояние возникает при одновременном доступе к какому-либо участку памяти двух и более потоков (в случае с go lang - горутин). В качестве классического примера для объяснения состояния гонок приводят банковскую ячейку. Допустим, у нас есть банковская ячейка, в которой лежит 10\$. Два человека с правом доступа одновременно делают запрос, чтобы узнать, какая сумма доступна для снятия и снять 7\$, если это возможно.

Если система недостаточно продумана, то оба человека получат по 7\$, а на счете должно оказаться -4\$ (это невозможно, то есть мы наткнулись на ошибку).

На примере в go lang:

```
package main

import (
    "fmt"
    "time"
)

type BankCell struct {
    DollarCell float64
}

func (b *BankCell) GetBalance() float64 {
    return b.DollarCell
}

func (b *BankCell) SubBalance(value float64) {
    if b.DollarCell-value >= 0 {
        time.Sleep(time.Millisecond * 1)
        b.DollarCell -= value
    }
}
```

```
func (b *BankCell) AddBalance(value float64) {
```

Куплено
благодаря

Skladchik.com

```

        b.DollarCell += value
    }

func main() {
    bc := &BankCell{
        DollarCell: 10.0,
    }

    go bc.SubBalance(7.0)
    bc.SubBalance(7.0)
    time.Sleep(time.Second)

    fmt.Println(bc.GetBalance())
}

```

Отследить data race весьма сложно, и в golang для этого есть специальный инструмент race. Попробуйте запустить код выше с флагом -race:

```
go run -race main.go
```

В консоли должно вывестись предупреждение вида:

```
WARNING: DATA RACE
```

```
Write at 0x00c00002c1fc by main goroutine
```

```
.....
```

указывающее на наличие и (место в коде) race в программе.

Для избежания состояния гонок был придуман механизм мьютексов. В golang он реализован с помощью структуры Mutex пакета sync.

Обратите внимание: стандарт golang не гарантирует отсутствие ошибок при доступе из разных горутин к одному участку памяти (переменной) без использования средств синхронизации (таких, как мьютекс).

Попробуйте переписать код выше, добавив мьютекс в класс банковской ячейки:

```

package main

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

type BankCell struct {
    DollarCell float64
    mu          sync.Mutex
}

```

```

}

func (b *BankCell) GetBalance() float64 {
    b.mu.Lock()
    defer b.mu.Unlock()
    return b.DollarCell
}

func (b *BankCell) SubBalance(value float64) {
    b.mu.Lock()
    defer b.mu.Unlock()
    if b.DollarCell-value >= 0 {
        time.Sleep(time.Millisecond * 1)
        b.DollarCell -= value
    }
}

func (b *BankCell) AddBalance(value float64) {
    b.mu.Lock()
    defer b.mu.Unlock()
    b.DollarCell += value
}

func main() {
    bc := &BankCell{
        DollarCell: 10.0,
    }

    go bc.SubBalance(7.0)
    bc.SubBalance(7.0)
    time.Sleep(time.Second)

    fmt.Println(bc.GetBalance())
}

```

Mutex имеет методы Lock() и Unlock(). Когда горютина в ходе своего выполнения доходит до команды закрытия (lock) мьютекса, она «проверяет», что мьютекс можно закрыть. Если это так, то горютина закрывает мьютекс и продолжает выполнение. Иначе горютина будет ждать, пока мьютекс не будет открыт (unlock) другой горютиной. Обратите внимание: попытка вызвать Unlock() на незакрытом мьютексе вызовет панику.

Запустите его с флагом `-race` и без него.

Подробнее о средствах синхронизации в пакетах `sync` и `atomic` будет рассказано далее.

Полезные ссылки:

- [Introducing the Go Race Detector](#)
- [Race condition](#)

Задание:

1. Создайте в проекте `module05` ветку `module05_03`.
2. Код для данного задания расположен в файле `cache/main.go` репозитория `module05`.
3. Перепишите класс `Cache` так, чтобы он стал `thread-safe` (потокбезопасным). Потокбезопасность означает, что код работает исправно при использовании как одним, так и несколькими потоками.
4. В ответе пришлите ссылку на MR в ветку `master` своего проекта ветки `module05_03`.