Урок Спринт 2.6

Mutex/RWMutex

Аннотация

В этой главе мы познакомимся с очень важным пакетом стандартой библиотеки языка GO sync и паттернами его использования в конкурентных программах.
Все примеры программ из этой главы нужно запускать с `go run -race main.go` Флаг `-race` активирует инструмент анализа гонок во время компиляции и выполнения программы. Если конкурентный код будет написан с проблемами — вы увидите надписи об этом в терминале программы.

Ранее вы уже знакомились с примитивом синхронизации мьютекс. Следуя тезису "повторение — мать учения", вспомним о его использовании. Если в нашей задаче мы хотим гарантировать, что только одна горутина имеет доступ к ресурсу в один момент времени, мьютекс может быть очень удобным и выразительным инструментом.

Давайте представим, что у нас есть функция, которая возвращает случайное число, и мы хотим получить 10 таких случайных чисел. Для получения случайного числа напишем функцию.

```
func random() int {
    const max int = 100
    return rand.Intn(max)
}
```

Давайте заведем слайс из 10 элементов и вызовем в цикле такую функцию.

Получим следующую простую программу.

```
package main

import (
    "fmt"
    "math/rand"
)
```

```
// Функция генерирует случайное число в интервале [0, 100)
func random() int {
    const max int = 100
    return rand.Intn(max)
}

func main() {
    const size int = 10
    results := []int{}
    // Заполняем слайс случайными числами
    for i := 0; i < size; i++ {
        results = append(results, random())
    }

    // Поэлементно печатаем слайс на экран
    for i := 0; i < size; i++ {
        fmt.Println(results[i])
    }
}
```

Давайте теперь представим, что `random()` - это очень сложная для вычислений. У нас появляется вполне естественное желание сделать вычисления конкурентыми, а не делать все последовательно.

Вы уже знакомились с горутинами. Давайте вызовем каждую функцию `random()` в отдельной горутине следующим образом

```
go func() {
    results = append(results, random())
}()
```

После заполнения слайса поставьте задержку `time.Sleep(time.Second)` в секунду, чтобы все горутины успели завершиться (это некрасивое решение, но давайте не обращать сейчас на это внимание)

Запустите получившуюся программу

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

// Функция генерирует случайное число в интервале [0, 100)
func random() int {
    const max int = 100
    return rand.Intn(max)
}

func main() {
```

```
    const size int = 10
    results := []int{}
    // Заполняем слайс случайными числами
    for i := 0; i < size; i++ {
        go func() {
            results = append(results, random())
        }()
    }
    time.Sleep(time.Second)

    // Поэлементно печатаем слайс на экран
    for i := 0; i < size; i++ {
        fmt.Println(results[i])
    }
}
```

с флагом `-race (go run -race main.go)` Программа выдаст на экран сообщение **WARNING: DATA RACE** и сообщение вида **panic: runtime error: index out of range [3] with length 3**. Тут мы должны вспомнить, что делает функция `append` и как работает слайс. Данный код меняет массив, ссылка на который хранится в слайсе `results`, потому что вместимости слайса недостаточно, и необходимо выделить новый массив и скопировать туда данные из старого. Таким образом, из нескольких горутин программа пытается поменять этот массив.

Решить данную проблему можно, использовав мьютекс, как вы уже знаете. Мьютекс создается так: `mx := &sync.Mutex{}` Окружаем нужный нам блок кода методами Lock и Unlock, и гарантируем тем самым, что доступ к слайсу получает только одна горутина в один момент времени.

```
go func() {
    // Вызван Lock, поэтому только одна горутина за раз может получить доступ к слайсу
    mx.Lock()
    defer mx.Unlock()
    results = append(results, random())
}()
```

На этом этапе программа выглядит следующим образом

```
package main

import (
    "fmt"
    "math/rand"
    "sync"
    "time"
)

// Функция генерирует случайное число в интервале [0, 100)
func random() int {
    const max int = 100
    return rand.Intn(max)
}
```

```
func main() {
    const size int = 10
    mx := &sync.Mutex{}
    results := []int{}
    // Заполняем слайс случайными числами
    for i := 0; i < size; i++ {
        go func() {
            // Вызван Lock, поэтому только одна горутина за раз может получить доступ к слайсу
            mx.Lock()
            defer mx.Unlock()
            results = append(results, random())
        }()
    }
    time.Sleep(time.Second)

    // Вызван Lock, потому что здесь тоже обращаемся к results
    mx.Lock()
    defer mx.Unlock()
    // Поэлементно печатаем слайс на экран
    for i := 0; i < size; i++ {
        fmt.Println(results[i])
    }
}
```

Данная программа конкурентна — но ее сложно читать и модифицировать. Давайте воспользуемся структурами, чтобы создать безопасный слайс и использовать его в функции `main()`. Структура будет выглядеть следующим образом:

```
type SafeSlice struct {
    results []int
    mx      *sync.Mutex
}
```

Обратите внимание — мы не хотим давать пользователю доступ к внутренним полям нашей структуры. Проассоциируйте со структурой две функции:

```
// Добавляем к слайсу элемент item
func (s *SafeSlice) Append(item int) {
    // Вызван Lock, поэтому только одна горутина за раз может получить доступ к слайсу
    s.mx.Lock()
    defer s.mx.Unlock()
    s.results = append(s.results, random())
}

// Получаем элемент слайсу по индексу
func (s *SafeSlice) Get(index int) int {
    // Вызван Lock, поэтому только одна горутина за раз может получить доступ к слайсу
    s.mx.Lock()
    defer s.mx.Unlock()
    return s.results[index]
```

```
}
```

Мы получили структуру, которая инкапсулирует в себя слайс и даёт пользователи две возможности:

– Добавить элемент

– Получить элемент по индексу. Все как в обычном слайсе, только можно вызвать в разных горутинах.

Добавьте функцию создания экземлряра нашей структуры для удобства пользователя.

```
func New() *SafeSlice {
    return &SafeSlice{
        mx:      &sync.Mutex{},
        results: []int{},
    }
}
```

Получившаяся программа удобна для чтения и модификации. Пользователь может безопасно создавать `slice`, с которым можно работать из разных горутин и не думать о безопасности доступа к ним.

```
package main

import (
    "fmt"
    "math/rand"
    "sync"
    "time"
)

type SafeSlice struct {
    results []int
    mx      *sync.Mutex
}

func New() *SafeSlice {
    return &SafeSlice{
        mx:      &sync.Mutex{},
        results: []int{},
    }
}

// Добавляем к слайсу элемент item
func (s *SafeSlice) Append(item int) {
    // Вызван Lock, поэтому только одна горутина за раз может получить доступ к слайсу
    s.mx.Lock()
    defer s.mx.Unlock()
    s.results = append(s.results, random())
}

// Получаем элемент слайсу по индексу
func (s *SafeSlice) Get(index int) int {
    // Вызван Lock, поэтому только одна горутина за раз может получить доступ к слайсу
```

```
    s.mx.Lock()
    defer s.mx.Unlock()
    return s.results[index]
}

// Функция генерирует случайное число в интервале [0, 100)
func random() int {
    const max int = 100
    return rand.Intn(max)
}

func main() {
    safeSlice := New()
    const size int = 10
    // Заполняем слайс случайными числами
    for i := 0; i < size; i++ {
        go func() {
            safeSlice.Append(random())
        }()
    }
    time.Sleep(time.Second)

    // Поэлементно печатаем слайс на экран
    for i := 0; i < size; i++ {
        fmt.Println(safeSlice.Get(i))
    }
}
```

Если есть переменная (обычно это слайс или карта), с которой хочется работать из разных горутин, бывает удобно инкапуслировать ее в структуру в виде поля и добавить мьютекс, как в приведенном примере. Давайте теперь придумаем способ избавиться от `time.Sleep(time.Second)`