МИСИС УНИВЕРСИТЕТ ITAM

Занятие 4

Тесты и бенчмарки в Golang

Занятие 4

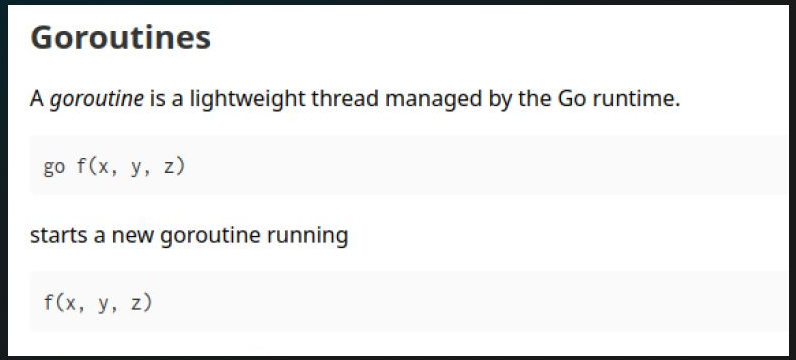
МИСИС УНИВЕРСИТЕТ ITAM

Структура

1. Рекап занятия 3
    a. Goroutines
    b. Channels
    c. Range and close
    d. Select
2. Mutex
3. Тесты
    a. Обычные тесты
    b. Немного конкурентных тестов
4. Бенчмарки
    a. Бенчмарки конкурентности
5. Библиотека testify

Рекап прошлого занятия

МИСИС УНИВЕРСИТЕТ ITAM



Goroutines

A *goroutine* is a lightweight thread managed by the Go runtime.

```
go f(x, y, z)
```

starts a new goroutine running

```
f(x, y, z)
```

- Легкий “поток” управляемый Go runtime
- Вызывается ключевым словом go
- Использует общую память, доступ к которой нужно синхронизировать

Рекап прошлого занятия

МИСИС УНИВЕРСИТЕТ ITAM

Channels

Channels are a typed conduit through which you can send and receive values with the channel operator, `<-`.

```
ch <- v    // Send v to channel ch.
v := <-ch  // Receive from ch, and
           // assign value to v.
```

(The data flows in the direction of the arrow.)

Like maps and slices, channels must be created before use:

```
ch := make(chan int)
```

By default, sends and receives block until the other side is ready. This allows goroutines to synchronize without explicit locks or condition variables.

- В каналы можно:
  писать (ch <-)
  читать из них (<- ch)
- Нужно создать, перед использованием

- Каналы блокируются до отправки или получения другой стороной

Рекап прошлого занятия

МИСИС УНИВЕРСИТЕТ ITAM

Buffered Channels

Channels can be *buffered*. Provide the buffer length as the second argument to `make` to initialize a buffered channel:

```
ch := make(chan int, 100)
```

Sends to a buffered channel block only when the buffer is full. Receives block when the buffer is empty.

Modify the example to overfill the buffer and see what happens.

- Можно создать канал с буфером
  ch := make(chan int, 100)

- Отправка ch <- в канал блокируется только при заполнении буфера
- Чтение <- ch из канала блокируется только, когда буфер пуст

Рекап прошлого занятия

МИСИС УНИВЕРСИТЕТ ITAM

Range and Close

A sender can `close` a channel to indicate that no more values will be sent. Receivers can test whether a channel has been closed by assigning a second parameter to the receive expression: after

```
v, ok := <-ch
```

`ok` is `false` if there are no more values to receive and the channel is closed.

The loop `for i := range c` receives values from the channel repeatedly until it is closed.

Note: Only the sender should close a channel, never the receiver. Sending on a closed channel will cause a panic.

Another note: Channels aren't like files; you don't usually need to close them. Closing is only necessary when the receiver must be told there are no more values coming, such as to terminate a `range` loop.

- Канал закрывают close(ch), чтобы показать конец отправки значений.
- Можно проверить, был ли закрыт канал с помощью параметра ok
  v, ok := <-ch
- Из канала можно читать в range, пока он не будет закрыт
- Канал закрывает отправитель
- Отправка в закрытый канал вызывает panic

Рекап прошлого занятия

МИСИС УНИВЕРСИТЕТ ITAM

Select

The `select` statement lets a goroutine wait on multiple communication operations.

A `select` blocks until one of its cases can run, then it executes that case. It chooses one at random if multiple are ready.

- Select блокируется пока один из его кейсов может быть выполнен
- Если готовы несколько кейсов выбирается один рандомный

Рекап прошлого занятия

МИСИС УНИВЕРСИТЕТ ITAM

Default Selection

The `default` case in a `select` is run if no other case is ready.

Use a `default` case to try a send or receive without blocking:

```
select {
case i := <-c:
    // use i
default:
    // receiving from c would block
}
```

- Если никакой из кейсов не готов: выполняется default, при наличии

sync.Mutex

МИСИС УНИВЕРСИТЕТ ITAM

sync.Mutex

We've seen how channels are great for communication among goroutines.

But what if we don't need communication? What if we just want to make sure only one goroutine can access a variable at a time to avoid conflicts?

This concept is called *mutual exclusion*, and the conventional name for the data structure that provides it is *mutex*.

Go's standard library provides mutual exclusion with `sync.Mutex` and its two methods:

```
Lock
Unlock
```

We can define a block of code to be executed in mutual exclusion by surrounding it with a call to `Lock` and `Unlock` as shown on the `Inc` method.

We can also use `defer` to ensure the mutex will be unlocked as in the `Value` method.

- Используем мьютексы для того, чтобы только одна горутина имела доступ к переменной
- sync.Mutex zero value можно использовать сразу же без инициализации
- Lock для обозначения, что манипуляции с данными начинаются
- Unlock вызывается всегда, даже при ошибках, чтобы не случился Deadlock

Зачем вообще нужны тесты?

- Неопределенное поведение выявляется быстрее
- Написание лучшего кода
- Проектировании лучшей архитектуры
- Проще проводить Bug Regression
- Рефакторинг становится безопаснее
- Деплой происходит увереннее

Как писать тесты в Golang?

- Сигнатуры функций должны быть вида func TestXxxx(t *testing.T)
- Название всегда начинается с Test
- Принимает единственный параметр *testing.T
- Файлы с тестами обычно называются так: sth_test.go
- Если происходит ошибка, тест считается зафейленным
- Фейлить можно вручную с помощью t.Error

Зачем вообще нужны бенчмарки?

- Оптимизация кода ради скорости/потребления памяти
- Поиск боттлнеков
- Сравнение эффективности различных реализаций функционала
- Победить в споре с коллегами🤓

Как писать бенчмарки в Golang?

- Сигнатуры функций должны быть вида BenchmarkXxxx(b *testing.B)
- Бенчмарки обычно хранятся в одних файлах с тестами
- Принимает единственный параметр *testing.B
- Название всегда начинается с Benchmark

Benchmarking

МИСИС УНИВЕРСИТЕТ ITAM

CPU Bound

- Слабый процессор не позволяет выполняться задачам быстрее

I/O Bound

- Медленные диски, оперативная память или сеть не позволяют задачам выполняться быстрее

Testify

МИСИС УНИВЕРСИТЕТ ITAM

QA