OTUS

Golang Developer. Professional

otus.ru

REC **Проверить, идет ли запись**

Меня хорошо видно && слышно?

Ставим “+”, если все хорошо
“-”, если есть проблемы

Тема вебинара

Concurrency patterns

Рубаха Юрий
Golang-разработчик

Правила вебинара

Активно
участвуем

Off-topic обсуждаем
в учебной группе

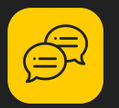
Задаем вопрос
в чат или голосом

Вопросы вижу в чате,
могу ответить не сразу

Условные обозначения

Индивидуально

Время, необходимое
на активность



Пишем в чат

Говорим голосом

Документ

Ответьте себе или
задайте вопрос

О чем будем говорить

- паттерны синхронизации данных;
- функции-генераторы и пайплайн;
- работа с многими каналами: or, fanin, fanout, etc.

Конкурентный код

Глобально мы обеспечиваем безопасность за счет:

- примитивов синхронизации (e.g. sync.Mutex, etc)
- каналы
- confinement-техники

Confinement-техники

Варианты:

- неизменяемые данные (идеально, но далеко не всегда возможно)
- ad hock
- lexical

Ad hoc

По сути, неявная договоренность, что "я - читаю, а ты пишешь", поэтому мы не используем никакие средства синхронизации.

```
data := make([]int, 4)

loopData := func(handleData chan<- int) {
    defer close(handleData)
    for i := range data {
        handleData <- data[i]
    }
}

handleData := make(chan int)
go loopData(handleData)

for num := range handleData {
    fmt.Println(num)
}
```

Lexical

Никакой договоренности нет, но она, по сути, неявно создана кодом.

```
chanOwner := func() <-chan int {
    results := make(chan int, 5)
    go func() {
        defer close(results)
        for i := 0; i <= 5; i++ {
            results <- i
        }
    }()
    return results
}

consumer := func(results <-chan int) {
    for result := range results {
        fmt.Printf("Received: %d\n", result)
    }
    fmt.Println("Done receiving!")
}

results := chanOwner()
consumer(results)
```

For-select цикл

Пример 1

```
for _, i := range []int{1, 2, 3, 4, 5} {
    select {
    case <-done:
        return
    case intStream <- i:
    }
}
```

Пример 2 (активное ожидание)

```
for {
    select {
    case <- done:
        return
    default:
    }
}
```

Как предотвратить утечку горутин

Проблема:

```
doWork := func(strings <-chan string) <-chan struct{} {
    completed := make(chan struct{})
    go func() {
        defer fmt.Println("doWork exited.")
        defer close(completed)
        for s := range strings {
            fmt.Println(s)
        }
    }()
    return completed
}

doWork(nil)

time.Sleep(time.Second * 5)
fmt.Println("Done.")
```

Невидимые ошибки Go-разработчика. Артём Картасов
https://youtu.be/TVe8pIFn2mY

Как предотвратить утечку горутин

Решение - явный индиктор того, что пора завершаться:

```
doWork := func(done <-chan struct{}, strings <-chan string)
    <-chan struct{} {
    terminated := make(chan struct{})
    go func() {
        defer fmt.Println("doWork exited.")
        defer close(terminated)
        for {
            select {
            case s := <-strings:
                fmt.Println(s)
            case <-done:
                return
            }
        }
    }()
    return terminated
}
...
```

Or-channel

А что, если источников несколько?
Можно воспользоваться идеей выше и применить ее к нескольким каналам.

And-channel

А как сделать аналогичную функцию с логикой "И"? :)

Обработка ошибок

Главный вопрос - кто ответственнен за обработку ошибок?
Варианты:

- просто логировать (имеет право на жизнь)
- падать (плохой вариант, но встречается)
- возвращать ошибку туда, где больше контекста для обработки

Обработка ошибок

Пример:

```
checkStatus := func(done <-chan struct{}, urls ...string) <-chan Result {
    results := make(chan Result)
    go func() {
        defer close(results)
        or _, url := range urls {
            var result Result
            resp, err := http.Get(url)
            result = Result{Error: err, Response: resp}
            select {
            case <-done:
                return
            case results <- result:
            }
        }
    }()
    return results
}
```

Pipeline

- Некая концепция.
- Суть - разбиваем работу, которую нужно выполнить, на некие этапы.
- Каждый этап получает какие-то данные, обрабатывает, и отсылает их дальше.
- Можно легко менять каждый этап, не задевая остальные.

https://blog.golang.org/pipelines
https://medium.com/statuscode/pipeline-patterns-in-go-a37bb3a7e61d

Pipeline

Свойства, обычно применимые к этапу (stage)

- входные и выходные данные имеют один тип
- должна быть возможность передавать этап (например, фукнции в Go - подходят)

Простой пример (batch processing)

Stage 1

```
multiply := func(values []int, multiplier int) []int {
    multipliedValues := make([]int, len(values))
    for i, v := range values {
        multipliedValues[i] = v * multiplier
    }
    return multipliedValues
}
```

Stage 2

```
add := func(values []int, additive int) []int {
    addedValues := make([]int, len(values))
    for i, v := range values {
        addedValues[i] = v + additive
    }
    return addedValues
}
```

Простой пример (batch processing)

Использование:

```
ints := []int{1, 2, 3, 4}
for _, v := range add(multiply(ints, 2), 1) {
    fmt.Println(v)
}
```

Тот же пайплайн, но с горутинами

Генератор

```
generator := func(done <-chan struct{}, integers ...int) <-chan int {
    intStream := make(chan int)
    go func() {
        defer close(intStream)
        for _, i := range integers {
            select {
            case <-done:
                return
            case intStream <- i:
            }
        }
    }()
    return intStream
}
```

Тот же пайплайн, но с горутинами

Горутина с умножением

```
multiply := func(done <-chan struct{}, intStream <-chan int, multiplier int) <-chan int {
    multipliedStream := make(chan int)
    go func() {
        defer close(multipliedStream)
        for i := range intStream {
            select {
            case <-done:
                return
            case multipliedStream <- i*multiplier:
            }
        }
    }()
    return multipliedStream
}
```

Тот же пайплайн, но с горутинами

Горутина с добавлением

```
add := func(done <-chan struct{},intStream <-chan int, additive int) <-chan int {
    addedStream := make(chan int)
    go func() {
        defer close(addedStream)
        for i := range intStream {
            select {
            case <-done:
                return
            case addedStream <- i+additive:
            }
        }
    }()
    return addedStream
}
```

Тот же пайплайн, но с горутинами

Использование:

```
done := make(chan struct{})
defer close(done)

intStream := generator(done, 1, 2, 3, 4)
pipeline := multiply(done, add(done, multiply(done, intStream, 2), 1), 2)

for v := range pipeline {
        fmt.Println(v)
}
```

Полезные генераторы - Repeat

```
repeatFn := func(done <-chan struct{}, fn func() interface{}) <-chan interface{} {
    valueStream := make(chan interface{})
    go func() {
        defer close(valueStream)
        for {
            select {
            case <-done:
                return
            case valueStream <- fn():
            }
        }
    }()
    return valueStream
}
```

Полезные генераторы - Take

```
take := func(done <-chan struct{}, valueStream <-chan interface{}, num int) <-chan interface{} {
    takeStream := make(chan interface{})
    go func() {
        defer close(takeStream)
        for i := 0; i < num; i++ {
            select {
            case <-done:
                return
            case takeStream <- <-valueStream:
            }
        }
    }()
    return takeStream
}
```

Fan-Out

Процесс запуска нескольки горутин для обработки входных данных.

Fan-In

Процесс слияния нескольких источников результатов в один канал.

Fan-Out & Fan-In

Смотрим на примере нахождения простых чисел.

Выводы

- старайтесь писать максимально простой и понятный код
- порождая горутину, задумайтесь, не нужен ли ей done-канал
- не игнорируйте ошибки, старайтесь вернуть их туда, где больше контекста
- использование пайплайнов делает код более читаемым
- использование пайплайнов позволяет легко менять отдельные этапы

Дополнительные материалы

https://blog.golang.org/pipelines
https://github.com/golang/go/wiki/LearnConcurrency
https://github.com/KeKe-Li/book/blob/master/Go/go-in-action.pdf
http://s1.phpcasts.org/Concurrency-in-Go_Tools-and-Techniques-for-Developers.pdf

---

https://github.com/uber-go/goleak

Задача из Ozon Go School

```
Необходимо в package main написать функцию
func Merge2Channels(
    f func(int) int,
    in1 <-chan int,
    in2 <-chan int,
    out chan <-int,
    n int)

Описание ее работы:
 n раз сделать следующее:
 прочитать по одному числу из каждого из двух каналов in1 и in2, назовем их x1 и x2.
 вычислить f(x1) + f(x2)
 записать полученное значение в out

Функция Merge2Channels должна быть неблокирующей, сразу возвращая управление.
Функция f может работать долгое время, ожидая чего-либо или производя вычисления.

Формат ввода
  Количество итераций передается через аргумент n.
  Целые числа подаются через аргументы-каналы in1 и in2.
  Функция для обработки чисел перед сложением передается через аргумент f.

Формат вывода
  Канал для вывода результатов передается через аргумент out.
```

Вопросы?

Ставим “+”,
если вопросы есть

Ставим “–”,
если вопросов нет

Заполните, пожалуйста, опрос о занятии по ссылке в чате

Спасибо за внимание!

Приходите на следующие вебинары

Рубаха Юрий
Golang разработчик