Введение в Golang. Лекция 2

Методы обработки запросов

Методы обработки запросов и плюсы неблокирующего подхода

В этой лекции мы рассмотрим асинхронное программирование и те подходы, которые реализованы в Go. Что такое асинхронное программирование? Асинхронное программирование — это парадигма, при которой операции вашей программы выполняются не строго последовательно, а могут быть прерваны какими-то другими операциями вашей же программы. Самым известным примером асинхронного программирования является технология Ajax — асинхронный JavaScript и XML, когда во время запроса на сервер ваша страница не замораживается, а продолжает работать.

Почему асинхронное программирование появилось и почему оно эффективно? Давайте обсудим, каким образом реализована обработка запроса в современном веб-сервере. Скорость современного процессора гораздо больше, чем скорость оперативной памяти. Для того чтобы как-то компенсировать это, был введен кэш процессора, который располагается вместе с ним на одном кристалле. Это очень быстрая память, но она очень маленькая. Сначала был введен кэш первого уровня, потом второго, потом третьего, когда-нибудь, скорее всего, введут кэш и четвертого уровня. После кэша идет оперативная память. То в процессоре есть кэш и он быстрый. Есть плашки памяти, которые являются оперативной памятью. Там память медленнее. Например, для того чтобы получить доступ из ядра процессора в основную оперативную память, может потребоваться до 100 наносекунд. С учетом количества операций, которые выполняет современный процессор, это достаточно много.

Почему память важна? Память важна при рассмотрении такого понятия, как переключение контекста. Что такое переключение контекста? Процессор выполняет просто последовательно какие-то операции, он ничего не знает про какие-то другие программы. Переключением из одной программы в другую программу занимается планировщик задач. Для этого он берёт один процесс, один тред, сохраняет его состояние. Потом берет состояние другого треда, загружает его в процессор, и начинает выполняться этот процесс. Причем здесь память? Дело в том, что при переключении задач, переключении процессов или системных тредов может возникнуть потребность в обращении к основной оперативной памяти, потому что в кэше данных для этого процесса нет. И нужно будет их подгрузить, то есть инвалидировать вообще весь процессорный кэш. Поэтому переключение контекста — операция достаточно дорогая. Мы можем затратить на нее до одной микросекунды в современных процессорах, что довольно много. Именно из-за переключения контекста, когда вы увеличиваете количество программ, которые выполняются на вашем компьютере, всё начинает работать медленнее. Этот подход к вытеснению одной программы другой, это называется вытесняющей многозадачностью. Теперь, помня и зная про переключение контекста, можно начать рассматривать уже методы обработки запросов в веб-сервере.

Итак, начнем мы cgi-bin. — технологии, в которой на каждый запрос поднимается новая программа, создается новый процесс, это тяжелая операция, нужно подключать довольно много памяти. Запрос выполняется, и после этого программа убивается. Это может быть очень не эффективно, потому что при увеличении количества запросов мы начнем тратить много времени на создание и завершение процессов и можем банально упереться в количество оперативной памяти.

Эволюцией этого подхода является worker pool, когда у нас есть некое количество процессов, которые не убиваются после завершения работы, а остаются в ожидании следующего запроса. Также есть подход, который называется мультитрединг. Это значит, что мы уже создаем не целый процесс на один запрос или одно соединение, а всего лишь тред. Тред — это более легкая сущность, чем процесс. Тред имеет доступ к памяти своего процесса, То есть вы можете переиспользовать какие-то соединения, например, с базой данных. Тред занимает меньше памяти, но для процессора это тоже системный тред, он тоже выполняется процессором, его тоже нужно переключать context switch. Но все же за счет экономии памяти мы можем обработать большее количество запросов. Также можно создать worker pool и обрабатывать запросы, не плодя бесконечно новые треды, а распределять запросы по фиксированному количеству тредов.

Может быть, можно как-то еще ускорится? Основное время на современном web api уходит на ожидание запроса от какой-то удаленной базы данных или веб-сервиса. И получается так, что тред блокируется на ожидание этого ответа и не выполняет никакой другой полезной работы. Из понимания этого родился событийный подход к обработке запросов, который реализован, например, в JS, в JavaScript. Это значит, что реализован неблокирующий ввод-вывод. Когда мы отправиляем запрос в базу данных, мы на этом не блокируемся, а продолжаем выполнять какие-то другие запросы, потому что иначе процессор бездействует. Таким образом, мы можем получить очень хорошую производительность. Мы можем обрабатывать много запросов внутри одного треда. Но тут есть нюансы. Дело в том, что, поскольку у нас тред один, то мы никак не можем выполнять запросы параллельно. В случае с вытесняющей многозадачностью, когда тред блокируется, какой-то другой тред работает. В случае с кооперативной многозадачностью в этом случае, мы должны дождаться окончания работы запроса № 1 для того, чтобы выполнять запрос № 2, и это может быть плохо, если у нас много операций на ЦПУ. Например, мы считаем какие-то хеши, занимаемся шифрованием либо архивированием, это тяжелая процессорная операция, и нам будет не хватать времени, которое мы проводим в ожидании ответов от базы данных. И соответственно будет не хватать времени для обработки всех запросов. Соответственно, возникает желание как-то разнести все на несколько ядер, для того чтобы, пока один тред занят, мы могли выполнять что-то в другом треде. То есть размасштабироваться.

Именно такой подход реализован в Go. Основан он на модели, которая называется communicating sequential processes от Тони Хоара, и оперируем мы в этом подходе такой сущностью, как горутина. Горутина — это аналог сопрограммы, когда в одном системном треде может выполняться несколько горутин, несколько сопрограмм. При этом особенностью является то, что наша горутина может начать выполняться на одном треде, потом уйти в ожидание данных из базы и продолжить выполняться в другом системном треде, потому что первый системный тред занят уже какой-то другой горутиной. Этот подход позволяет получить очень хорошую производительность — хорошую пропускную способность. Горутины являются одной из ключевых особенностей языка Go и одной из самых сильных его сторон. А теперь давайте рассмотрим, как это выглядит в коде.

Горутины и каналы

Горутины - легковесные процессы

Прежде всего подробнее проговорим про горутины — одну из тех приятных вещей, которые сделают вашу жизнь при разработке многопоточных программ в Go гораздо проще. Итак, рассмотрим для начала следующую программу.

```
const (
        iterationsNum = 7
        goroutinesNum = 5
)

func doSomeWork(in int) {
        for j := 0; j < iterationsNum; j++ {
                fmt.Printf(formatWork(in, j))
        }

}

func main() {
        for i := 0; i < goroutinesNum; i++ {
                doSomeWork(i)
```

```
        }
}

func formatWork(in, j int) string {
        return fmt.Sprintln(strings.Repeat("  ", in), "*",
                strings.Repeat("  ", goroutinesNum-in),
                "th", in,
                "iter", j, strings.Repeat("*", j))
}
```

в ней запускается цикл, который запускает worker'ов, которые делают какую-то работу. В данном случае работа — это вывод на экран. Как отработает эта программа? Сначала запустится нулевой worker и что-то напечатает, затем первый worker что-то напечатает и так далее. Пусть теперь мы хотим, чтобы эти worker'ы работали не последовательно, один за другим, а параллельно. Для этого можно запустить их в разных горутинах. Для того чтобы запустить какую-либо функцию внутри отдельной горутины, нужно написать просто слово go.

```
func main() {
        for i := 0; i < goroutinesNum; i++ {
                go doSomeWork(i)
        }
}
//fmt.Scanln ()
```

Теперь при вызове функции doSomeWork программа в цикле не заблокируется, пока worker не отработает, а сразу же перейдёт к следующему циклу, и в нём запустит следующую горутину. Но есть нюанс. В данном случае моя программа может завершиться сразу же после окончания цикла, не дав отработать никаким горутинам. Поэтому пока поставим fmt.Scanln (), чтобы она дожидалась ввода с экрана. Теперь вывод будет не последовательным. При этом каждый раз он будет различным. Что бы убедиться в этом, несколько раз запустите программу. Подробнее поговорим о том, как дожидаться выполнения всех горутин, в следующих главах.

Так же в go есть возможность передать управление другой горутине. В пакете runtime находится функция Goshed(), вызывающая планировщик задач. Добавим её вызов в функцию воркера.

```
func doSomeWork(in int) {
        for j := 0; j < iterationsNum; j++ {
                fmt.Printf(formatWork(in, j))
                runtime.Goshed()
        }

}
```

Можно заметить, что теперь вывод программы принципиально изменится. Если раньше наши горутины выполнялись в какой-то непонятной последовательности, при этом одна могла успеть выполнить либо много итераций, либо мало итераций, то сейчас она выполняет всего одну итерацию и передаёт управление другой горутине. На самом деле, внутри функции Printf() тоже может вызваться планировщик задач, и он тоже может передать работу другой горутине. При этом одна работа может начать выполняться на одном системном треде, а продолжить выполняться на другом системном треде. То есть go не просто блокируется на каких-то вызовах и передаёт управление дальше, планировщик может решить, что эта горутина уже много поработала, и передать управление другой, то есть ваша программа не заблокируется просто в цикле. Точнее она может заблокироваться в цикле, если там выполняются простые математические операции без вызова каких-либо функций.

Также стоит отметить, что из горутины нельзя вернуть никакое значение в основную программу. Для этих целей используются каналы, про которые мы поговорим в следующем разделе.

Каналы - передаем данные между горутинами

Основная трудность при многопроцессорной разработке, когда программа использует несколько ядер процессоров или даже несколько процессоров, состоит в том, что нужно как-то их синхронизировать между собой и каким-то образом передавать данные между разными потоками. В Go для этих целей есть каналы. Каналы нужны, для того чтобы передавать владения данными в другую горутину. Как это работает? Для

начала нужно создать какой-то канал, канал создается конструкцией make, после идет ключевое слово chan и тип канала. Каналы тоже типизированы, в них можно класть только определенный тип данных.

```
func main() {
        ch1 := make(chan int)

        go func(in chan int) {
                val := <-in
                fmt.Println("GO: get from chan", val)
                fmt.Println("GO: after read from chan")
        }(ch1)

        ch1 <- 42

        fmt.Println("MAIN: after put to chan")
        fmt.Scanln()
}
```

После создания канала запустим горутину, которая принимает на вход канал (in — это имя канала, chan int — это тип канала), вычитывает оттуда какое-то значение и печатает его. Вычитывание из канала происходит посредством стрелки слева от канала. В основной горутине мы кладет значение в этот канал. Кладется значение в канал, используя стрелку справа от канала. Теперь давайте запустим это и посмотрим, как оно работает.

```
GO: get from chan 42
MAIN: after put to chan
GO: after read from chan
```

Смотрите: сначала мы положили значение в канал, после этого мы вывели текст из главной горутины, после этого основная программа ожидает ввода строки, а управление переключилось в горутину. Горутина получила значение из канала, и после значения что-то вывела. Но есть нюанс. Конструкция make (chan int) создает небуферизированный канал. Это значит, что, когда мы кладем в канал какое-то значение, кто-то с другой стороны этого канала должен его прочитать. И если никто не читает, то горутина блокируется, до тех пор пока на той стороне не появится тот, кто вычтет это значение. Это может к тому, что мы залочимся. Такая ситуация называется deadlock — когда горутина ждет, но не может дождаться. Чтобы воспроизвести такую ситуацию, достаточно в основной программе положить еще что-нибудь в канал. Теперь программа не завершится или ругнется в рантайме. В случае, если в deadlock попадут какие-то горутины внутри программы, а другие горутины продолжат работать, то может прийти ситуация, которая называется утечка горутин. Это приводит к утечке памяти.

Каким образом этого можно избежать? Каналы в Go могут быть буферизированными. Это значит, что канал может принимать не одно значение, до того как он залочится, а несколько. Осуществляется это если мы укажем значение буфера для этого канала.

```
ch1 := make(chan int, 1)
```

После этого программа успешно отработает.

Если после запятой в конструкции make указать 0, то это по-прежнему будет небуферизированный канал. Если указать значение n, это будет значит, что можно положить n значений в буфер, и только при попытке положить n+1-ое значение программа залочится и будет ждать пока там освободится место.

А теперь давайте рассмотрим, как можно работать в цикле канала. Перед нами программа, которая запускается, принимает небуферизированный канал. И начинает печатать туда значения. После закрывает канал, а в главной горутине у есть цикл, который, используя range, итерируется по значениям, которые приходят в канал.

```
func main() {
        in := make(chan int, 0)

        go func(out chan<- int) {
                for i := 0; i <= 10; i++ {
                        fmt.Println("before", i)
                        out <- i
```

```
                fmt.Println("after", i)
        }
        close(out)
        fmt.Println("generator finish")
    }(in)

    for i := range in {
        fmt.Println("\tget", i)
    }

    // fmt.Scanln()
}
```

После 10-й итерации наш генератор, завершит работу, управление вернется в главную горутину и она завершится. Всегда ли иттерация по каналу завершается? Важно знать, что именно закрытие канала ведет к завершению иттерации. Если убрать закрытие канала, генератор закончил работу, мы получим последнее значение, но впадем в deadlock, потому что в канал in уже больше никто не пишет, но мы все еще ждем оттуда значений. То есть конструкция close, она позволяет завершить цикл, которые пытается итерироваться по значениям, идущим в канал.

При передаче канала в функцию можно указать, какого типа будет этот канал. Например, если я укажу просто chan int, это будет означать, что в канал можно и писать, и читать. Если я поставить стрелку после канала, как в примере, это будет значить, что в этот канал можно только писать. Если поставить стрелку слева, это будет значит, что в канал можно только писать. Это удобно, чтобы запретить делать с каналом те операции, которые не ожидаются от этой горутины. Соответствие этому будет проверяться на уровне компиляции.

Мультеплексирование каналов через оператор select

Теперь поговорим о мультиплексировании каналов при помощи инструкции select. Часто бывают случаи, когда нам нужно опрашивать несколько каналов, но мы не можем делать это последовательно или хотя бы в цикле, потому что тогда они заблокируются, если еще не готовы. Для того, чтобы опрашивать несколько каналов, есть специальная инструкция select.

```
func main() {
    ch1 := make(chan int, 1)
    ch2 := make(chan int)
    //ch1 <- 1
    select {
    case val := <-ch1:
        fmt.Println("ch1 val", val)
    case ch2 <- 1:
        fmt.Println("put val to ch2")
    default:
        fmt.Println("default case")
    }
}
```

Итак, у нас есть два канала и инструкция select. Select очень похожа на switch, однако в ней в качестве случаев для case выступают не логические выражения, а операции над каналами. Вы можете как читать из канала какое-то значение, так и писать в канал. И есть оператор default, который выполняется в том случае, если никакой из других случаев не ответил, то есть все каналы заблокированы. Давайте запустим эту программу. Выполнится default case. это произошло потому, что в первом канале, откуда мы пытались прочитать, ничего не было, а второй канал, куда мы что-то попытались записать, был заблокирован, потому что из него никто не пытался читать. Теперь, если раскоментировать третью строчку и положить единицу в первый канал, то выполнится случай, когда мы пытались прочитать что-то из первого канала. Если случая default не будет, и при этом никакой канал не будет готов, то наступит ситуация deadlock. За раз внутри select выполняется только одна операция, то есть не происходит так, что select сам опросил несколько каналов и каждый из них выполнил, нет. Если мы хотим, чтобы попытки опроса каналов происходили еще, то нам приходится делать цикл. Давайте посмотрим, как это работает.

```
func main() {
    ch1 := make(chan int, 2)
```

```
        ch1 <- 1
        ch1 <- 2
        ch2 := make(chan int, 2)
        ch2 <- 3
LOOP:
        for {
                select {
                case v1 := <-ch1:
                        fmt.Println("chan1 val", v1)
                case v2 := <-ch2:
                        fmt.Println("chan2 val", v2)
                default:
                        break LOOP
                }
        }
}
```

Создадим два канала. В каждый из них положим по несколько значений. Теперь внутри цикла мы будем опрашивать эти каналы, если там что-то есть, мы будем выводить это на экран. А если ничего нам не вернется, то мы завершим цикл, используя break и метку цикла. Мы не можем написать просто break, потому что в этом случае мы завершим select, поэтому нужна метка цикла. Таким образом мы прочитаем все значения из каналов и завершимся. Попробуйте запустить программу несколько раз и обратите внимание, что каждый раз чтение происходит не в том порядке, в котором мы их указали внутри select. Какой из каналов выбрать для чтения, планировщик Go выбирает случайным образом. Контролировать это невозможно.

В рамках одного select мы можем не только читать из разных каналов, но и писать. И это открывает нам возможность для того, чтобы завершать какие-то внешние каналы, внешние функции, используя так называемый канал отмены. Давайте посмотрим, как это работает.

```
func main() {
        cancelCh := make(chan struct{})
        dataCh := make(chan int)

        go func(cancelCh chan struct{}, dataCh chan int) {
                val := 0
                for {
                        select {
                        case <-cancelCh:
                                return
                        case dataCh <- val:
                                val++
                        }
                }
        }(cancelCh, dataCh)

        for curVal := range dataCh {
                fmt.Println("read", curVal)
                if curVal > 3 {
                        fmt.Println("send cancel")
                        cancelCh <- struct{}{}
                        break
                }
        }

}
```

Итак, у нас есть два канала. Первый канал — канал отмены, в него мы можем класть пустую структуру. Пустая структура — это структура без полей, она не занимает места, то есть это как будто только сам признак, что там что-то есть. И канал с данными. Теперь вызовем функцию, внутри которой будет бесконечный цикл с select-ом с несколькими case-ами внутри. Первый case выполняется только если мы что-то прочитали из канала отмены, В этом случае мы завершим работу всей функции. В противном

случае мы будем писать в канал с данными увеличивающиеся значения. Ну и теперь давайте попробуем итерироваться по этому каналу со значениями — data Channel, выводить на экран прочитаное значение и отправлять в канал отмены пустую структуру, если значение больше 3. После отправки пустой структуры, бесконечный цикл должен будет завершиться. Программа выведет пять значений.

```
read 0
read 1
read 2
read 3
read 4
send cancel
```

В этом разделе мы обсудили как можно мультиплексировать каналы, как можно читать из нескольких каналов, как использовать мультиплексор select для того, чтобы завершить работу канала. В следующем разделе мы посмотрим, какие другие функции Go предоставляет для удобной работы с таймингами и завершением каналов.

Инструменты для многопроцессорного программирования

Таймеры и таймауты

Первым в этой главе рассмотрим, как работать с таймаутами и организовывать операции, которые нужно выполнять через какие-то промежутки времени. Начнем с таймаутов. Бывает, что нам нужно ждать выполнения операции фиксированное количество времени, например, пользователь может ждать несколько секунд, пока мы чтото считаем, но не будет ожидать бесконечно. Рассмотрим пример.

```
package main

import (
        "fmt"
        "time"
)

func longSQLQuery() chan bool {
        ch := make(chan bool, 1)
        go func() {
                time.Sleep(2 * time.Second)
                ch <- true
        }()
 package main

import (
        "fmt"
        "time"
)

func longSQLQuery() chan bool {
        ch := make(chan bool, 1)
        go func() {
                time.Sleep(2 * time.Second)
                ch <- true
        }()
        return ch
}

func main() {
        // при 1 выполнится таймаут, при 3 - выполнится операция
        timer := time.NewTimer(2 * time.Second)
        select {
        case <-timer.C:
                fmt.Println("timer.C timeout happend")
        case <-time.After(time.Minute):
```

```
                // пока не выстрелит - не соберётся сборщиком мусора
                fmt.Println("time.After timeout happend")
        case result := <-longSQLQuery():
                // освобождет ресурс
                if !timer.Stop() {
                        <-timer.C
                }
                fmt.Println("operation result:", result)
        }
}       return ch
        }

        func main() {
                // при 1 выполнится таймаут, при 3 - выполнится операция
                timer := time.NewTimer(1 * time.Second)
                select {
                case <-timer.C:
                        fmt.Println("timer.C timeout happend")
                case <-time.After(time.Minute):
                        // пока не выстрелит - не соберётся сборщиком мусора
                        fmt.Println("time.After timeout happend")
                case result := <-longSQLQuery():
                        // освобождет ресурс
                        if !timer.Stop() {
                                <-timer.C
                        }
                        fmt.Println("operation result:", result)
                }
        }
```

Мы объявили долгую функцию, которая спит две секунды, а потом пишет в канал, что она выполнилась. Объявим timer, на одну секунду, который не даст нам дождаться выполнения функции. Таймеры находится в пакете time. У таймера есть канал, и в случае если мы будем использовать мультиплексор select, то мы можем поставить case на чтение с этого канала, и, как только наступит нужное время, появится событие, и он сработает. У NewTimer есть короткое объявление, когда нам сразу же возвращается канал без промежуточной переменной, time.After, использованный во втором case. Но у него есть небольшие особенности. Мы можем прервать выполнение timer , а time.After мы остановить не можем, и пока он не выполнится, даже если мы завершили функцию, он не освободит ресурсы. Поэтому, если нужно лучше контролировать расход ресурсов, расход памяти, пользуйтесь простым таймером и останавливайте его, когда вам это требуется. Если запустим нашу программу, случится таймаут, который мы указывали в первом case. Теперь, если поменять время таймера на тройку, например, то сработает наша длинная операция.

Организация таймаута на какую-то операцию, делается она через мультиплексор select. Теперь рассмотрим как работать с периодическими событиями, то есть повторяющимися с определенным интервалом, пока вы что-то делаете в это время. Для их обработки используется конструкция ticker, она тоже находится в пакете time, ее можно создать через NewTicker.

```
        func main() {
                ticker := time.NewTicker(time.Second)
                i := 0
                for tickTime := range ticker.C {
                        i++
                        fmt.Println("step", i, "time", tickTime)
                        if i >= 5 {
                                // надо останавливать, иначе потечет
                                ticker.Stop()
                                break
                        }
                }
                fmt.Println("total", i)

                // не может быть остановлен и собран сборщиком мусора
```

```
        // используйте если должен работать вечено
        c := time.Tick(time.Second)
        i = 0
        for tickTime := range c {
                i++
                fmt.Println("step", i, "time", tickTime)
                if i >= 5 {
                        break
                }
        }

}
```

Если вы читаете из канала, который предоставляет ticker, то вам возвращается время, когда было срабатывание. Поэтому вы можете использовать это время, если вам нужно зафиксировать, в какой момент было произведено событие. В примере мы итерируемся по каналу тикера и выводим номер шага и время срабатывания. Повторяем всего лишь пять секунд и останавливаем ticker, чтобы он больше не посылал мне время, не посылал мне тики, и прерываем цикл.

Опять-таки у тикера есть короткий alias, time.Tick, который возвращает сразу канал, с которого вы можете читать, но у него есть особенность: он работает бесконечно, вы никак не можете его освободить. В данном случае, если завершить цикл но не выйти из программы, ticker будет продолжать работать. Его надо использовать тогда, когда вы точно не планируете его останавливать. Например, для сбора какого-то мониторинга с вашей программы, который запускается, например, каждую минуту, собирает метрики и куда-то их отсылает.

Еще одна конструкция, которую рассмотрим, это AfterFunc.

```
package main

import (
        "fmt"
        "time"
)

func sayHello() {
        fmt.Println("Hello World")
}

func main() {
        timer := time.AfterFunc(5*time.Second, sayHello)

        fmt.Scanln()
        timer.Stop()

        fmt.Scanln()
}
```

AfterFunc, как это видно из названия, запускает поданную на вход функцию через определенный промежуток времени. Как результат выполнения AfterFunc, возвращается таймер, который можно остановить самостоятельно, прервав выполнение отложенной функции. Если запустить этот код, то будет видно, что мы сначала ждем пять секунд, потом функция отрабатывает и выводит приветствие. Но в этом же примере можно и очень быстро прервать функцию, нажав любую клавишу во время ожидания, так как вызовется timer.Stop. В результате функция никогда не будет вызвана.

Пакет context и отмена выполнения

Теперь поговорим про пакет Context. Context очень широко используется в Go, вы будете встречать его практически повсеместно, и одно из его основных назначений — это отмена выполнения асинхронных операции. Давайте рассмотрим пример. Допустим, мы отправляем запросы на несколько разных серверов, при этом нам нужен результат от только от того, который ответит первым из них, всех остальных мы не хотим дожидаться. Можно, конечно, попробовать реализовать ожидания через какой-то канал или ещё что-то, но обычно это делается через Context. Рассмотрим пример.

```
func main() {
        ctx, finish := context.WithCancel(context.Background())
        result := make(chan int, 1)

        for i := 0; i <= 10; i++ {
                go worker(ctx, i, result)
        }

        foundBy := <-result
        fmt.Println("result found by", foundBy)
        finish()

        time.Sleep(time.Second)
}
```

Мы вызываем context.WithCancel(context.Background()) и нам возвращается сам контекст и функция отмены finish. ContextBackground — это базовый контекст, от которого всё наследуется. После этого мы запускаем десять воркеров; первым параметром мы им передаём контекст, номер воркера и канал для записи результата. После этого мы дожидаемся первого результата, выводим его на экран и вызываем функцию finish. Finish обрабатывается посредством чтения канала, который возвращает метод контекста ctx.Done, безопасный для использования между несколькими горутинами. Что происходит в воркере?

```
func worker(ctx context.Context, workerNum int, out chan<- int) {
        waitTime := time.Duration(rand.Intn(100)+10) * time.Millisecond
        fmt.Println(workerNum, "sleep", waitTime)
        select {
        case <-ctx.Done():
                return
        case <-time.After(waitTime):
                fmt.Println("worker", workerNum, "done")
                out <- workerNum
        }
}
```

Мы эмулируем работу засыпанием на случайное время, Мы окажемся или в ситуации, когда подошло к концу время сна воркера, мы запишем результат в канал и напишем, что воркер отработал. Или получим что-то из канала ctx.Done, в котором что-то появилась после запуска функции finish, тогда мы завершим работы.

Теперь рассмотрим, как не вручную завершить выполнение контекста, а по таймауту. На этот случай в пакете Context есть функция WithTimeout, которая также возвращает контекст и функцию finish Установим время ожидания на 50 миллисекунд. Функцию воркера оставим прежнемей, а в main фактически повторим то же самое.

```
func main() {
        workTime := 50 * time.Millisecond
        ctx, _ := context.WithTimeout(context.Background(), workTime)
        result := make(chan int, 1)

        for i := 0; i <= 10; i++ {
                go worker(ctx, i, result)
        }

        totalFound := 0
LOOP:
        for {
                select {
                case <-ctx.Done():
                        break LOOP
                case foundBy := <-result:
                        totalFound++
                        fmt.Println("result found by", foundBy)
                }
```

```
        }
        fmt.Println("totalFound", totalFound)
        time.Sleep(time.Second)
}
```

Только теперь завершения контекста мы будем ждать и в мэйне (это произойдет, когда истечет время, тогда мы выйдем из цикла), а результаты получать в цикле и подсчитывать, сколько воркеров успело отработать за отведенное время.

Асинхронное получение данных

Теперь поговорим про прикладное применение каналов и горутин. Рассмотри ускорение работы запросов путём распараллеливания. Например, представьте, что есть страницы с и комментарии и связанными статьям. При этом комментарии и статьи между собой никак не связаны, В go вы можете распараллелить получение статей и комментариев, внеся тяжёлую операцию в отдельную горутину, которая будет общаться с основной программой через канал. Давайте посмотрим, как это работает на практике.

```
func getComments() chan string {
        // надо использовать буферизированный канал
        result := make(chan string, 1)
        go func(out chan<- string) {
                time.Sleep(2 * time.Second)
                fmt.Println("async operation ready, return comments")
                out <- "32 комментария"
        }(result)
        return result
}

func getPage() {
        resultCh := getComments()

        time.Sleep(1 * time.Second)
        fmt.Println("get related articles")

        return

        commentsData := <-resultCh
        fmt.Println("main goroutine:", commentsData)
}
```

Внутри функции, которая будет обрабатывать страницу, запускаем getComments, которая вернет канал. После этого мы не начинаем сразу ждать ответ из этого канала, а выполняем ещё какую-то работу. Например, получаем статьи. (Для простоты мы, вместо того, чтоб получать статьи, засыпаем.) И только потом дожидаемся результата из канала. Давайте посмотрим, как работает функции getComments(): мы создаём канал, после этого запускаем горутину, куда передаём этот канал, и возвращаем его. При этом сама полезная работа, будет выполняться уже в горутине. (И опять мы вместо полезной работы для простоты поспим) Когда операция закончится, в канал запишется результат и мы получим его в главной горутине. Получается, что в главной горутине мы получаем связанные статьи и параллельно комментарии.

Однако есть один нюанс, который стоит знать. Если в случае, когда вы используете небуферизированный канал, по какой-то причине произойдет ошибка и вы вернётесь из функции, не дождавшись результата из канала, получится, что горутина попытается записать в канал синхронно, то есть ожидая с другой стороны чтение, но никто читать не будет. То есть получится утечка горутин и, как следствие, утечка памяти. Поэтому канал должен быть буферизированным. Это даёт нам возможность записать нужное количество значений в канал, не блокируясь. Функция корректно выполнится, завершится, а сам канал уберётся сборщиком мусора, потому что из него никто не пытается ни читать, никто туда не пишет.

Распараллеливание очень полезный инструмент в работе, и позволяет неплохо оптимизировать некоторые операции.

Пул воркеров

Теперь рассмотрим еще один инструмент распараллеливания — пул воркеров. Довольно часто встречаются случаи, когда работа осуществляется через какие-то очереди, с которыми работают какие-то офлайновые «разгребаторы». Часто для этих целей есть «форкающиеся демоны», но в Go можно поступить гораздо проще. Можно создать несколько горутин, которые будут читать из какого-то канала задачи и выполнять их, а в главной горутине мы будем получать задачи и помещать их в канал.

Рассмотрим пример.

```
const goroutinesNum = 3

func startWorker(workerNum int, in <-chan string) {
        for input := range in {
                fmt.Printf(formatWork(workerNum, input))
                runtime.Gosched() // попробуйте закомментировать
        }
        printFinishWork(workerNum)
}

func main() {
        worketInput := make(chan string, 2) // попробуйте увеличить размер канала
        for i := 0; i < goroutinesNum; i++ {
                go startWorker(i, worketInput)
        }

        months := []string{"Январь", "Февраль", "Март",
                "Апрель", "Май", "Июнь",
                "Июль", "Август", "Сентябрь",
                "Октябрь", "Ноябрь", "Декабрь",
        }

        for _, monthName := range months {
                worketInput <- monthName
        }
        close(worketInput) // попробуйте закомментировать

        time.Sleep(time.Millisecond)
}
```

Создадим три воркера — три горутины, которые будут выводить на экран свой номер и то, что считали из канала. Как они работают внутри? Внутри функции-воркеры читают из канала и выполняют работу. И все! При этом, если нужно завершить пул воркеров, тодостаточно просто закрыть канал, и тогда цикл, в котором крутится внутри воркер, закончится. При этом, если вдруг не закрыть канал, то главная горутина не дождется окончания работы воркера это может привести к утечкам горутин либо же к дедлоку.

sync.Waitgroup - ожидание завершения работы

В предыдущих примерах, когда нам надо было дождаться завершения всех горутин, мы часто использовали функцию Scanln(). Это удобно для демонстрации во время курса, но совсем не подходит для применения в реальной работе. В Go в пакете sync есть специальная структура WaitGroup(). Она работает довольно просто. Перед запуском worker'ов либо ещё чего-то, чего нужно дождаться, вы увеличиваете счетчик, а потом ждете пока этот счётчик не уменьшится в ноль. Давайте посмотрим ещё детальнее.

```
const (
        iterationsNum = 7
        goroutinesNum = 5
)

func startWorker(in int, wg *sync.WaitGroup) {
        defer wg.Done() // уменьшаем счетчик на 1
        for j := 0; j < iterationsNum; j++ {
                fmt.Printf(formatWork(in, j))
```

```
                runtime.Gosched()
        }
}

func main() {
        wg := &sync.WaitGroup{} // wait_2.go инициализируем группу
        for i := 0; i < goroutinesNum; i++ {
                wg.Add(1) // добавляем воркер
                go startWorker(i, wg)
        }
        time.Sleep(time.Millisecond)

        wg.Wait() // wait_2.go ожидаем, пока waiter.Done() не приведёт счетчик к 0
}
```

Первым делом создаем ссылку на wait group (чтобы структуру не надо было копировать после создания всегда создавайте ее по указателю). Перед запуском горутины кладем её в wait-группу, то есть выполняем wg.Add(1). Это нужно сделать обязательно перед запуском горутины, а не в самой горутине, что вроде бы было бы логично? потому что горутина может не запуститься до того, как завершится цикл. Тогда программа успеет дойти до wg.Wait(), увидеть, что там ещё никого нет, и завершить работу, не дождавшись выполнения. Внутри worker'а сразу же пишем defer, отложенный вызов, метода Done у структуры WaitGroup. Метод Done уменьшает счётчик внутри WaitGroup. Поскольку мы используем defer, сброс счетчика произойдет даже при аварийном заершении горутины. Таким образом, когда все workerы отрабатывают своё, срабатывает wg.Wait(), и мы завершаем программу корректно.

Ограничение по ресурсам

Напоследок поговорим про rate-лимиты. Довольно часто бывает так, что нам нужно каким-либо образом затормозить программу, например, в зависимости от утилизации процессора, либо дисковой подсистемы, нужно уменьшить нагрузки на эти части. В Go мы можем реализовать это, используя буферизированные каналы. Давайте посмотрим на пример кода.

```
func startWorker(in int, wg *sync.WaitGroup, quotaCh chan struct{}) {
        quotaCh <- struct{}{} // ratelim.go, берём свободный слот
        defer wg.Done()
        for j := 0; j < iterationsNum; j++ {
                fmt.Printf(formatWork(in, j))
        }
        <-quotaCh // ratelim.go, возвращаем слот
}

func main() {
        wg := &sync.WaitGroup{}
        quotaCh := make(chan struct{}, quotaLimit) // ratelim.go
        for i := 0; i < goroutinesNum; i++ {
                wg.Add(1)
                go startWorker(i, wg, quotaCh)
        }
        time.Sleep(time.Millisecond)
        wg.Wait()
}
```

В maine, так как мы уже научились правильно дожидаться завершения работы горутин, создаем WaitGroup. После этого создаем буферезированный канал — канал с квотой. Он будет основным инструментом для решения поставленной задачи. Размер буфера равен нашему ограничению. Теперь мы будем передавать канал с квотой в Workerы, Посмотрим, что происходит в Worker'е. Прежде чем начать выполнять какую-то полезную работу, Worker пытается взять слот на эту работу. В данном случае мы пытаемся положить пустую структуру в канал. Если буфер канала уже заполнен, значит, два Worker'а уже работают, и мы блокируемся, до тех пор пока не освободится место. Таким образом мы не позволяем числе работающих горутин расти сверх квоты. Когда воркер дождался очереди, он выполняет работу и возвращает слот на работу, то есть читаем из канала, для того чтобы какая-то другая горутина смогла туда написать и захватить работу.

Теперь давайте немножко изменим воркер, так чтобы он возвращал квоту обратно на каждой второй итерации.

```
func startWorker(in int, wg *sync.WaitGroup, quotaCh chan struct{}) {
        quotaCh <- struct{}{} // ratelim.go, берём свободный слот
        defer wg.Done()
        for j := 0; j < iterationsNum; j++ {
                fmt.Printf(formatWork(in, j))

                if j%2 == 0 {
                        <-quotaCh             // ratelim.go, возвращаем слот
                        quotaCh <- struct{}{} // ratelim.go, берём слот
                }

                runtime.Gosched() // даём поработать другим горутинам
        }
        <-quotaCh // ratelim.go, возвращаем слот
}
```

Проверьте, как теперь изменился вывод программы. Должно быть видно, что теперь Worker не полностью захватывает квоту и выполняет всю свою работу до конца, а дает возможность выполнять ее каким-то другим Worker'ам. При этом по-прежнему работает одновременно не более двух рутин.

Это очень мощный инструмент. Пользуйтесь им!

Состояние гонки

Ситуация гонки на примере конкурентной записи в map

В этой главе поговорим про состояние гонки, оно же race condition, или data race. Вам оно может быть знакомо, если вы занимались базами данных и изучали транзакции, которые нужны, для того чтобы атомарно обновить какое-то значение в базе данных. Рассмотрим пример, когда можно встретить состояние гонки в Go.

```
func main() {
        var counters = map[int]int{}
        for i := 0; i < 5; i++ {
                go func(counters map[int]int, th int) {
                        for j := 0; j < 5; j++ {
                                counters[th*10+j]++
                        }
                }(counters, i)
        }
        fmt.Scanln()
        fmt.Println("counters result", counters)
}
```

В этой программе обновляются значения в мапе со счетчиками из разных горутин. Результат этой работы этой программы нас может не порадовать, потому что map в Go — это конкурентно небезопасный тип данных. Дело в том, что мапа внутри себя — это ссылка на структуру данных, как хеш-таблица. И она может перестраиваться и копироваться в другое место, или еще что-то с ней может происходить. При этом в кэше одного процессора может лежать одно значение, в кэше другого — другое значение. И когда разные горутины попытаются обновить значение в основной памяти, они могут попасть в состояние гонки. Если запустить программу, она почти в любом случае вылетит с ошибкой fatal error: concurrent map read and map write. Но так же эта ошибка может и не всегда воспроизвестись. Ошибки, связанные с состоянием гонки бывает сложно отловить как раз из-за этого. Для того чтобы ловить похожие ошибки (их еще называют плавающими багами, либо гейзенбагами), в Go есть специальная директива при запуске программы или компиляции — race. Попробуйте запустить вашу программу с ней. Вы увидите, что счетчики были выведены на экран, то есть программа хотя бы не упала. Но написала нам, что нашлось два состояния гонки. Как раз на строчках. где мы пытаемся менять мапу и читать из нее. Давайте теперь посмотрим, как с этим бороться.

sync.Mutex для синхронизации данных

Каким образом нам обезопасить наши данные от состояния гонки? Одним из инструментов, которые предлагает стандартная библиотека go для этих целей, является Mutex. Давайте посмотрим, как с ним работать.

```
import (
        "fmt"
        "sync"
)

func main() {
        var counters = map[int]int{}
        mu := &sync.Mutex{}
        for i := 0; i < 5; i++ {
                go func(counters map[int]int, th int, mu *sync.Mutex) {
                        for j := 0; j < 5; j++ {
                                mu.Lock()
                                counters[th*10+j]++
                                mu.Unlock()
                        }
                }(counters, i, mu)
        }
        fmt.Scanln()
        mu.Lock()
        fmt.Println("counters result", counters)
        mu.Unlock()
}
```

Мы создаем Mutex. Обратите внимание, что мы создаем его, как ссылку на объект, в связи со своей внутренней реализацией мьютекс не должен быть копирован после первого использования, как значение, поэтому его всегда надо создавать и передавать, как ссылку на объект, После создания мы передаем мьтекс в go-рутину, и внутри go-рутины Mutex выполняет основную свою работу. Перед началом работы с данными мы блокируем мьютекс. Пока он заблокирован, все другие, кто попробует взять lock будут ожидать. Таким образом с данными будет работать одновременно только одна горутина. После завершения работы lock обязательно надо снять. Кроме того мы защищаемся мьютексом в главной горутине при чтении данных, чтобы не возникла такая ситуация, когда мы пытаемся вычитать то, что другая горутина сейчас меняет. (Попробуйте запустить программу с локом на чтение и без него с дериктивой race и посмотреть. что она выведет.) Итак, мы защитились от состоятия гонки в данном примере с помощью мьютекса, но при работе с Mutex-ами есть некоторые нюансы. Рассмотрим их в следующей части.

sync.Atomic

В Go mutex — это такой большой, «жирный» примитив для синхронизации мощных кусков. Но иногда нам бывает нужно всего лишь атомарно инкрементировать просто переменную. Для этих целей есть примитив под названием atomic. Давайте начнем с примера.

```
package main

import (
        "fmt"
        "time"
)

var totalOperations int32 = 0

func inc() {
        totalOperations++
}

func main() {
        for i := 0; i < 1000; i++ {
                go inc()
```

```
        }
        time.Sleep(2 * time.Millisecond)
        // ождается 1000, но по факту будет меньше
        fmt.Println("total operation = ", totalOperations)
}
```

Программа запускает тысячу горутин, каждая из которых пытается увеличить глобальный счетчик. Мы ожидаем, что по результатам работы программы в нашем счетчике будет 1000. Однако из-за того, что счетчики инкрементируются из разных горутин, это будет не так. При одном запуске может получиться, что счетчик равен 950, а при следующем запуске вывестись и вовсе другое значение. Дело в том, что мы попадаем в состояние гонки. Для того, чтобы сделать атомарный счетчик, нужно воспользоваться атомиком из пакета sync/atomic.

```
package main

import (
        "fmt"
        "sync/atomic" // atomic_2.go
        "time"
)

var totalOperations int32

func inc() {
        atomic.AddInt32(&totalOperations, 1) // автомарно
}

func main() {
        for i := 0; i < 1000; i++ {
                go inc()
        }
        time.Sleep(2 * time.Millisecond)
        fmt.Println("total operation = ", totalOperations)
}
```

В примере мы используем функцию из пакета atomic AddInt32, куда передаем адрес нашей переменной и то значение, на которое мы хотим увеличить. Больше ничем программа не отличается, но теперь в ней не возникает состояния гонки и счетчик всегда становится равным 1000.

Atomic быстрее мьютекса. На самом деле внутри мьютекса встроен atomic как более низкоуровневый примитив. В пакете atomic довольно много разных функций для атомарных операций со счетчиками либо для сравнения и замены. Если вы активно занимаетесь многопроцессорной разработкой, то очень рекомендуется знать, что он может.