

# Опанування основами Go: Практичний посібник з освоєння мови Go

## Розділ 8: Паралельність з Goroutines та Channels



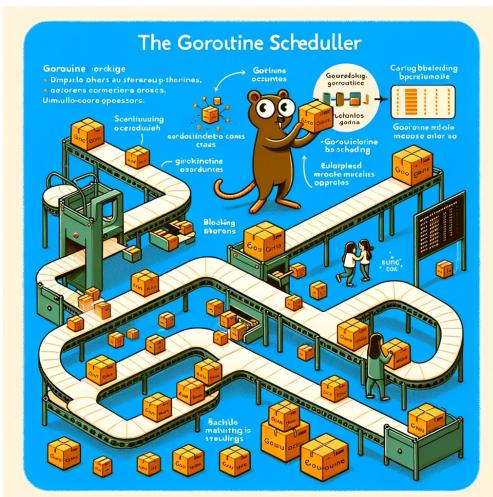
### Введення в Конкурентність у Go

Конкурентність - це концепція, яка стала ключовою в програмуванні на Go, і це одна з основних причин, чому ця мова здобула широку популярність. Вона відрізняється від традиційної багатопоточності, зосереджуючись на структуруванні програм таким чином, щоб вони могли ефективно виконувати кілька задач одночасно, незалежно від архітектури процесора.

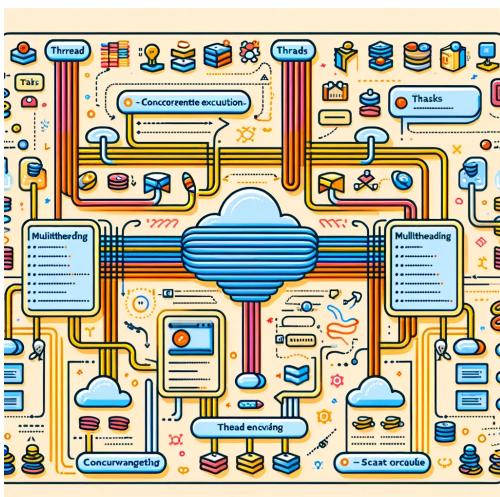
### Конкурентність vs Багатопоточність

Хоча конкурентність та багатопоточність часто використовуються як взаємозамінні терміни, важливо розуміти різницю:

## Description



- **Конкурентність** означає структурування програм для незалежного виконання різних частин коду. Це не обов'язково означає, що код виконується одночасно; скоріше, це про дизайн, де задачі розділені та можуть працювати незалежно.



- **Багатопоточність** - це про технічне виконання кількох потоків одночасно, що може бути на одному або декількох процесорних ядрах.

## Goroutines: Легковажні Потоки

Goroutines у Go - це один з основних механізмів для впровадження конкурентності. Вони схожі на потоки, але набагато легші за вагою та ефективніші з точки зору використання пам'яті та контекстного перемикання.

- **Низькі Витрати:** Goroutines займають значно менше пам'яті, ніж стандартні потоки, і можуть бути створені в тисячі разів швидше.
  - **Динамічне Управління:** Планувальник Go автоматично управлює Goroutines, розподіляючи їх між доступними процесорними ядрами для ефективного виконання.

## Планувальник Goroutines

Планувальник в Go динамічно керує виконанням Goroutines, розподіляючи їх між доступними потоками виконання (зазвичай відповідає кількості процесорних ядер). Він реалізує модель M:N, де M Goroutines виконуються на N потоках ОС. Це забезпечує ефективне використання ресурсів, оскільки планувальник може перемикати Goroutines між потоками в залежності від їхнього стану та доступності ресурсів.

## Планувальник Goroutines: Перемикання між Goroutines

Планувальник Goroutines в Go відповідає за розподіл Goroutines між потоками операційної системи та визначає, коли і як відбувається перемикання між Goroutines. Це перемикання відбувається за певних умов:

- Блокування Операцій:** Якщо Goroutine виконує операцію, яка блокується, наприклад, операції введення/виведення, виклики системних функцій або функцій блокування (`sync.Mutex.Lock()`), планувальник віддає управління іншій Goroutine.
- Довготривалі Операції:** Якщо Goroutine виконує тривалу операцію без блокувань, планувальник може перервати її виконання для того, щоб надати час процесора іншим Goroutines. Це запобігає "загарбанню" процесорного часу однією Goroutine. Але це відбувається лише у випадку, якщо доступне більше ніж одне процесорне ядро. "Основний потік" також є Goroutine (функція `main`).
- Синхронізаційні Примітиви:** Використання примітивів синхронізації, таких як м'ютекси та канали, також може спричинити перемикання контексту. Коли Goroutine чекає на розблокування м'ютекса або отримання даних з каналу, планувальник може перемкнути управління на іншу Goroutine.
- Функції `runtime.Gosched()`:** Розробники можуть вручну вказати планувальніку переключитися на іншу Goroutine, використовуючи функцію `runtime.Gosched()`. Це може бути корисно у ситуаціях, де Goroutine виконує дуже тривалу операцію без природних точок блокування.
- Завершення Goroutine:** Природно, коли Goroutine завершує виконання, управління повертається до планувальника, який вибирає наступну Goroutine для виконання.

## Важливість Планувальника

Планувальник Goroutines відіграє ключову роль у конкурентності Go, забезпечуючи, що всі Goroutines отримують час процесора та виконуються ефективно. Він розроблений таким чином, щоб максимально використовувати доступні процесорні ресурси, при цьому забезпечуючи справедливе та ефективне планування виконання Goroutines.

## Резюме

Конкурентність у Go, реалізована через Goroutines та Channels, відкриває широкі можливості для створення швидких, ефективних та **легко підтримуваних** програм. Використання цих інструментів дозволяє розробникам писати код, який може ефективно використовувати багатоядерні процесори, а також легко масштабуватися та адаптуватися до різних умов виконання.

### 8.1 Goroutines: Легковісна Паралельність

Простота використання: Створення Goroutines в Go значно спрощується завдяки використанню ключового слова `go`.

## Створення Goroutines

Створення нової Goroutine в Go є дуже простим. Для цього використовується ключове слово `go`, за яким слідує виклик функції:

```
package main

import (
    "fmt"
    "time"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
    go say("world")
    say("hello")
}
```

У цьому прикладі, `say("world")` виконується як Goroutine. Це означає, що програма не чекає завершення цієї функції та одразу ж продовжує виконання наступного рядка коду, тобто виклику `say("hello")`. Так як основна функція також є Goroutine перемикання відбувається дуже "випадково", що можна легко побачити виконавши цей код.

Функція `time.Sleep` також є дуже гарним "тригером" для перемикання між Goroutine

## Паралельне Виконання

Коли ви створюєте Goroutine, ви не маєте прямого контролю над його виконанням. Планувальник Go автоматично управлює Goroutines, виконуючи їх паралельно, якщо це можливо (наприклад, на багатоядерних системах).

## Синхронізація Goroutines

Важливою частиною роботи з Goroutines є синхронізація їх виконання. Через те, що Goroutines виконуються асинхронно, іноді потрібно синхронізувати їх для обміну даними або впорядкування виконання. Для цього використовуються канали (`channels`) та інші примітиви синхронізації, які будуть розглянуті в наступних секціях.

## Застереження

Під час роботи з Goroutines важливо пам'ятати про потенційні проблеми з конкурентним доступом до спільних ресурсів. Необхідно використовувати синхронізацію або інші методи для запобігання гонкам за дані (`race conditions`).

## 8.2 Channels: Комунікація між Goroutines

Channels у Go - це потужний механізм для безпечної та ефективної комунікації між Goroutines. Вони дозволяють Goroutines обмінюватися даними без необхідності використання примітивів синхронізації, таких як м'ютекси.

## Створення Channel

Для створення channel використовується вбудована функція `make`. Channel має бути типізований, що означає, що він може передавати дані лише певного типу. Але це не заважає використати як тип інтерфейси або універсальноті (коли канал є частиною структури яка використовує `generic`).

```
ch := make(chan int) // Створення channel, який може передавати цілі числа
```

## Відправлення та Отримання Даних

Для відправлення даних у channel використовується оператор `<-`, а для отримання - той же оператор, але з іншого боку.

```
ch <- v // Відправлення v у channel ch  
v := <-ch // Отримання даних з channel ch і присвоєння їх змінній v
```

## Закриття Channel

Channel можна закрити, використовуючи функцію `close`. Після закриття channel неможливо відправити в нього дані, але отримати дані ще можна.

```
close(ch)
```

## Використання Channels для Синхронізації

Channels часто використовуються для синхронізації Goroutines, наприклад, для повідомлення про завершення задачі.

```
func worker(done chan bool) {  
    fmt.Println("Working...")  
    time.Sleep(time.Second)  
    fmt.Println("Done")  
  
    done <- true  
}  
  
func main() {  
    done := make(chan bool, 1)  
    go worker(done)
```

```
<-done // Чекаємо на сигнал від worker
}
```

У цьому прикладі, `main` запускає Goroutine `worker` і чекає, поки не отримає сигнал про завершення через channel `done`.

## Буферизовані Channels

Channels можуть бути буферизовані, що означає, що вони можуть зберігати певну кількість значень без необхідності негайногого отримання цих значень.

```
ch := make(chan int, 2)
ch <- 1
ch <- 2
fmt.Println(<-ch)
fmt.Println(<-ch)
```

У цьому прикладі, `ch` є буферизованим channel, який може зберігати до двох цілих чисел. Абсолютно вірно, канали в Go можуть бути спеціалізовані для операцій тільки на читання або тільки на запис. Це надає додатковий рівень безпеки та чіткості в коді, оскільки ви можете точно визначити, як має використовуватися канал у вашій програмі.

## Канали Тільки для Читання

Канал тільки для читання визначається за допомогою `chan<- T`, де `T` - це тип даних, які передаються через канал. Цей тип каналу дозволяє відправляти дані до каналу, але не дозволяє їх отримувати.

```
func writeToChannel(ch chan<- int, value int) {
    ch <- value
}
```

У цьому прикладі, `ch` - це канал, який призначений тільки для відправлення цілих чисел.

## Канали Тільки для Запису

Навпаки, канал тільки для запису визначається як `<-chan T`, де `T` - тип даних, які отримуються через канал. Цей тип каналу дозволяє отримувати дані з каналу, але не дозволяє відправляти дані до каналу.

```
const channelClosed = -100
func readFromChannel(ch <-chan int) int {
    v, ok := <-ch // читаємо з перевіркою на то що канал не закритий
    if !ok {
        return channelClosed
    }
}
```

```
    return v  
}
```

У цьому прикладі, `ch` - це канал, який призначений тільки для отримання цілих чисел.

## Практичне Використання

Використання каналів тільки для читання або тільки для запису допомагає запобігти помилкам у рантаймі, пов'язаним з неправильним використанням каналів. Це також робить наміри вашого коду яснішими для інших розробників, які читають або рефакторять ваш код. Наприклад, якщо функція приймає канал тільки для читання, вона явно не може відправляти дані до цього каналу, і навпаки.

## Резюме

Channels у Go надають потужний спосіб для безпечної комунікації між Goroutines. Вони дозволяють розробникам уникати складностей, пов'язаних з прямим використанням примітивів синхронізації, і надають простий, але ефективний спосіб обміну даними і синхронізації задач у конкурентних програмах.

## 8.3 Select та Timeouts: Розширене Використання Channels

Конструкція `select` у Go дозволяє одночасно очікувати на кілька операцій каналу. Це може бути особливо корисно для реалізації таймаутів, неблокуючих читань або записів, а також для вибору між кількома каналами для виконання різних операцій.

### Використання Select

`select` дозволяє вам чекати на кілька каналів одночасно і виконати блок коду, заснований на тому, який канал готовий до комунікації.

```
select {  
    case msg1 := <-ch1:  
        fmt.Println("Отримано з ch1:", msg1)  
    case msg2 := <-ch2:  
        fmt.Println("Отримано з ch2:", msg2)  
}
```

У цьому прикладі, `select` чекає на дані з двох каналів (`ch1` та `ch2`). Якщо один з каналів отримує дані, виконується відповідний блок коду. Якщо жоден з каналів не отримує дані протягом однієї секунди, виконується блок таймауту.

### Реалізація Таймаутів

Таймаути корисні для запобігання безкінечного очікування відповіді. Ви можете використовувати `time.After` для створення таймауту:

```
select {
    case res := <-ch:
        fmt.Println(res)
    case <-time.After(1 * time.Second):
        fmt.Println("Таймаут операції")
}
```

У цьому випадку, якщо канал `ch` не надсилає відповідь протягом однієї секунди, програма приступає до виконання коду таймауту.

## Неблокуючі Операції

За допомогою `Select` можна реалізувати неблокуючі операції читання або запису в канал за допомогою `default`:

```
select {
    case msg := <-ch:
        fmt.Println("Отримано:", msg)
    default:
        fmt.Println("Немає даних")
}
```

Тут, якщо немає даних у каналі `ch`, відразу виконується `default` блок, і програма не застрягає у очікуванні даних з каналу.

## Резюме

Конструкція `Select` у Go є потужним інструментом для роботи з каналами, особливо коли мова йде про обробку кількох каналів одночасно. Це ключовий елемент для створення ефективних конкурентних програм, дозволяючи розробникам ефективно управляти часом очікування операцій, уникати блокування та обробляти кілька сценаріїв комунікації в одному контексті.

## 8.4 sync.Mutex та sync.WaitGroup: Синхронізація Goroutines

У багатьох випадках при роботі з конкурентними програмами в Go, виникає потреба у синхронізації доступу до спільних ресурсів або у координації роботи між різними Goroutines. Для цих цілей часто використовуються `sync.Mutex` та `sync.WaitGroup`.

### sync.Mutex для Взаємного Виключення

`sync.Mutex` використовується для забезпечення безпечної доступу до спільних ресурсів у конкурентному середовищі. `Mutex` стоїть за "взаємне виключення" і дозволяє блокувати доступ до ресурсу, поки одна Goroutine його використовує.

### Використання Mutex

```
var mu sync.Mutex
var sharedResource int

func worker() {
    mu.Lock() // Блокування доступу до спільного ресурсу
    sharedResource++
    mu.Unlock() // Розблокування доступу
}
```

У цьому прикладі, `mu.Lock()` та `mu.Unlock()` використовуються для контролю доступу до змінної `sharedResource`.

## **sync.Mutex для Координації Goroutines**

`sync.WaitGroup` використовується для очікування завершення групи Goroutines. Він дозволяє одній Goroutine чекати, поки інші Goroutines не завершать свою роботу.

### **Використання WaitGroup**

```
var wg sync.WaitGroup

for i := 0; i < 5; i++ {
    wg.Add(1) // Додавання лічильника для Goroutine
    go func(i int) {
        defer wg.Done() // Вказує на завершення Goroutine
        fmt.Println("Робота", i)
    }(i)
}

wg.Wait() // Чекати на завершення всіх Goroutines
```

У цьому прикладі, `wg.Add(1)` використовується для додавання лічильника для кожної Goroutine. `wg.Done()` викликається в кінціожної Goroutine для зниження лічильника. `wg.Wait()` блокує виконання до того моменту, поки всі Goroutines не завершать роботу.

## **Резюме**

`sync.Mutex` та `sync.WaitGroup` в Go - це важливі інструменти для синхронізації та координації Goroutines. Використання цих інструментів дозволяє уникнути проблем з конкурентним доступом до спільних ресурсів та гарантувати, що всі задачі будуть виконані у відповідному порядку. Вони є ключовими для написання надійних та ефективних конкурентних програм на Go.