Опановування основами Go: Практичний посібник з освоєння мови Go

Розділ 8.5 Примітиви сінхронізації

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
}
```

sync.Mutex для Взаємного Виключення

`sync.Mutex` використовується для забезпечення безпечного 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.WaitGroup 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 кінці кожної Goroutine для зниження лічильника. `wg.Wait()` блокує виконання до того моменту, поки всі Goroutines не завершать роботу. `sync.Map` - це спеціалізована реалізація карти (map) у пакеті `sync` Go, яка оптимізована для випадків, коли ключі та значення карти часто змінюються різними goroutines. Вона забезпечує безпечність при конкурентному доступі без потреби вручну керувати блокуванням за допомогою м'ютексів.

Використання `sync.Map`

`sync.Map` має декілька ключових особливостей:

1. **Безпечність при Конкурентному Доступі:** Вона безпечно підтримує доступ до своїх елементів від декількох goroutines без додаткового зовнішнього блокування.
2. **Ефективна Підтримка Частого Змінювання:** Оптимізована для використання, де елементи часто додаються або видаляються goroutines.
3. **Load, Store, Delete:** Пропонує методи, які дозволяють безпечно виконувати операції читання, запису та видалення.

Використання `sync.Map`

```
package main

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

```
func main() {
    var m sync.Map

    // Збереження значень
    m.Store("hello", "world")
    m.Store("number", 42)

    // Отримання значення
    if value, ok := m.Load("hello"); ok {
        fmt.Println("hello:", value)
    }

    // Оновлення значення
    m.Store("hello", "Go")

    // Видалення значення
    m.Delete("number")

    // Перебір елементів карти
    m.Range(func(key, value interface{}) bool {
        fmt.Println(key, value)
        return true
    })
}
```

Застосування `sync.Map`

`sync.Map` є ідеальним вибором у наступних сценаріях:

- Коли кілька goroutines часто читають, пишуть або видаляють ключі картки.
- Коли стандартні карти Go вимагають складного управління блокуванням для забезпечення безпечності при конкурентному доступі.

`sync.Once` у пакеті `sync` Go - це спеціальна структура, яка забезпечує, що дія буде виконана лише один раз, незалежно від того, скільки разів вона викликається і скільки goroutines намагаються її виконати. Це корисно для ініціалізації ресурсів, яка повинна відбутися лише один раз у програмі.

sync.Once

1. **Гарантія Єдиного Виклику:** `sync.Once` гарантує, що навіть при конкурентному доступі дія буде виконана лише один раз.
2. **Безпека при Конкурентності:** Це забезпечує безпечне виконання в багатопоточному середовищі без необхідності вручну керувати блокуванням.

Використання `sync.Once`

```
package main

import (
```

```
    "fmt"
    "sync"
)

var once sync.Once
var resource string

func initResource() {
    resource = "Initialized"
}

func main() {
    for i := 0; i < 10; i++ {
        go func() {
            once.Do(initResource) // initResource буде викликано лише один
раз
        }()
    }

    // Чекаємо достатньо часу, щоб всі goroutines встигли запуститись
    // У реальному коді слід використовувати sync.WaitGroup або інший
механізм синхронізації
    fmt.Scanln()
    fmt.Println("Resource state:", resource)
}
```

Застосування `sync.Once`

`sync.Once` використовується в таких сценаріях:

- **Ініціалізація Спільного Ресурсу:** Для безпечного створення або ініціалізації спільного ресурсу, що використовується декількома goroutines.
- **Лінива Ініціалізація:** Коли ініціалізація ресурсу є ресурсоємною, і ви хочете відкласти її до моменту, коли вона дійсно потрібна.

Резюме

`sync.Once` є важливим інструментом у багатопоточному програмуванні на Go, особливо коли потрібно забезпечити, що деяка операція виконується лише один раз. Це дозволяє уникнути зайвих витрат на ініціалізацію та забезпечує безпечність та ефективність у конкурентному середовищі.

`sync.Cond` в Go - це синхронізаційний примітив, який використовується для управління очікуванням між goroutines. Він дозволяє одній або більше goroutines чекати на певну умову (condition) і повідомляє про її зміну іншими goroutines.

sync.Cond

`sync.Cond` складається з трьох основних компонентів:

1. **Locker:** Зазвичай це `sync.Mutex` або `sync.RWMutex`, який використовується для блокування доступу до частини коду.

2. **Wait:** Метод, який блокує викликаючу goroutine до тих пір, поки не буде викликано `Signal` або `Broadcast` на тому ж `sync.Cond`.

3. **Signal і Broadcast:**

    - `Signal` пробуджує одну чекаючу goroutine.
    - `Broadcast` пробуджує всі чекаючі goroutines.

Використання `sync.Cond`

Розглянемо простий приклад, де `sync.Cond` використовується для координації роботи між goroutines:

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
    var wg sync.WaitGroup
    var mu sync.Mutex
    cond := sync.NewCond(&mu)

    for i := 0; i < 5; i++ {
        wg.Add(1)
        go func(i int) {
            defer wg.Done()

            mu.Lock()
            cond.Wait() // Чекаємо на умову
            mu.Unlock()

            fmt.Println("Горутина", i, "пробуджена")
        }(i)
    }

    // Даємо час горутинам блокуватися на умові
    time.Sleep(1 * time.Second)

    // Пробуджуємо одну горутину
    cond.Signal()

    // Даємо час на пробудження
    time.Sleep(1 * time.Second)

    // Пробуджуємо всі решту горутин
    cond.Broadcast()
```

```
    wg.Wait()
}
```

Пояснення

1. Створюється `sync.Cond` з м'ютексом.
2. П'ять горутин очікують (`Wait`) на умову.
3. `Signal` пробуджує одну горутину, в той час як `Broadcast` пробуджує всі інші.

Застосування `sync.Cond`

`sync.Cond` корисний у таких сценаріях:

- Коли потрібно, щоб одна або декілька горутин чекали на деякі події чи умови, перш ніж продовжити виконання.
- Для реалізації бар'єрів синхронізації або подій оповіщення у конкурентних програмах.

Резюме

`sync.Cond` надає спосіб для горутин чекати на визначені умови та координувати їх виконання у відповідь на зміни цих умов. Це корисний інструмент у ситуаціях, де потрібно забезпечити синхронізацію між декільком

`sync.Pool` у Go - це спеціалізована структура, що використовується для тимчасового зберігання та повторного використання об'єктів. Це допомагає оптимізувати продуктивність шляхом зменшення кількості операцій виділення пам'яті, особливо в програмах з високим рівнем конкурентності.

sync.Pool

1. **Зменшення Виділення Пам'яті:** `sync.Pool` повторно використовує вже виділені об'єкти, зменшуючи тим самим навантаження на сміттєзбірник (garbage collector) та систему виділення пам'яті.

2. **Автоматичне Очищення:** Об'єкти, збережені в `sync.Pool`, можуть бути автоматично видалені при кожному проході сміттєзбірника, тому вони підходять лише для тимчасового зберігання.

3. **Конкурентно Безпечний:** `sync.Pool` безпечний для використання у багатопоточних середовищах.

Використання `sync.Pool`

Дуже спрощений приклад використання:

```
package main

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

```
func main() {
    var pool sync.Pool

    // Встановлення функції для створення нового об'єкта, коли Pool
порожній
    pool.New = func() interface{} {
        return "Новий Об'єкт"
    }

    // Отримання об'єкта з Pool
    obj := pool.Get()
    fmt.Println(obj) // Виведе "Новий Об'єкт"

    // Повернення об'єкта назад у Pool
    pool.Put(obj)

    // Повторне отримання того ж об'єкта
    obj2 := pool.Get()
    fmt.Println(obj2) // Виведе "Новий Об'єкт" знову
}
```

Застосування `sync.Pool`

`sync.Pool` особливо корисний у таких сценаріях:

- **Висока Частота Короткотривалих Об'єктів:** У ситуаціях, де програма часто створює та знищує короткотривалі об'єкти.
- **Кешування Ресурсів:** Для кешування великих або ресурсоємних об'єктів, що необхідні регулярно.

Резюме

`sync.Pool` в Go - це потужний інструмент для оптимізації продуктивності програм, який забезпечує ефективне управління пам'яттю шляхом повторного використання об'єктів. Він знижує навантаження на сміттєзбірник і допомагає уникнути частих операцій виділення та звільнення пам'яті, особливо у програмах з високим рівнем конкурентності.