DWARVES FOUNDATION

Concurrency and Goroutines

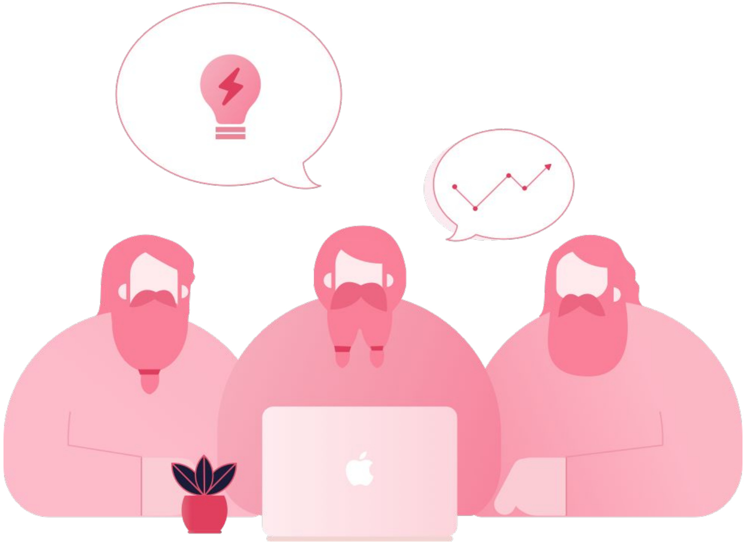
Goroutines, channels and concurrency patterns

Author Name

Hieu Phan

@hieupq

andy@dwarvesv.com



DWARVES FOUNDATION

Agenda

Topic summary

1. Understanding Goroutines
2. Communicating through channels
3. Wait groups and mutexes
4. Concurrent Patterns
5. Demo

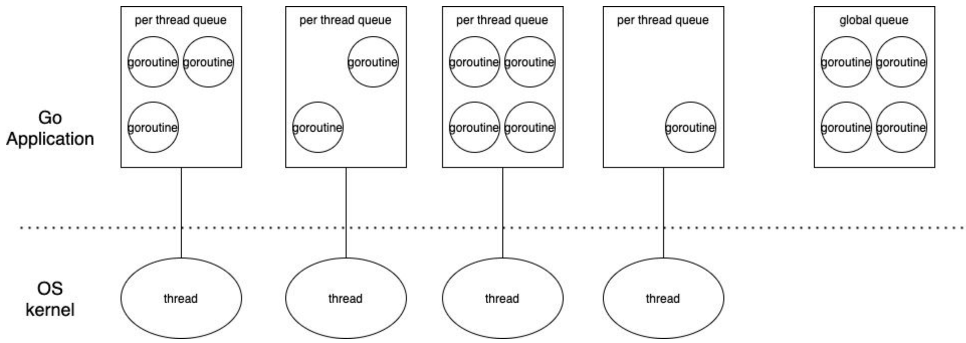
Introduction

Concurrency & Go's Concurrency model

Understanding Goroutines

Goroutines

- Goroutines are similar to threads in other programming languages but are designed to be more lightweight and efficient
- You can think of them as independent workers executing tasks concurrently



DWARVES FOUNDATION

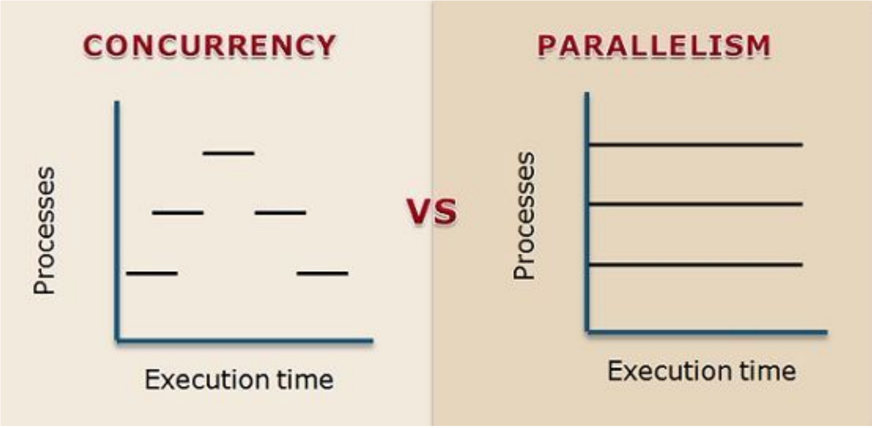
Goroutines

- You can create a Goroutine simply by prefixing a function or method call with the keyword “go”

```go
package main

import (
  "fmt"
  "time"
)

func numbers() {
  for i := 1; i <= 5; i++ {
    time.Sleep(250 * time.Millisecond)
    fmt.Printf("%d ", i)
  }
}

func alphabets() {
  for i := 'a'; i <= 'e'; i++ {
    time.Sleep(400 * time.Millisecond)
    fmt.Printf("%c ", i)
  }
}

func main() {
  go numbers()
  go alphabets()
  time.Sleep(3000 * time.Millisecond)
  fmt.Println("main terminated")
}

// Output: 1 a 2 b 3 c 4 d 5 e main terminated
```

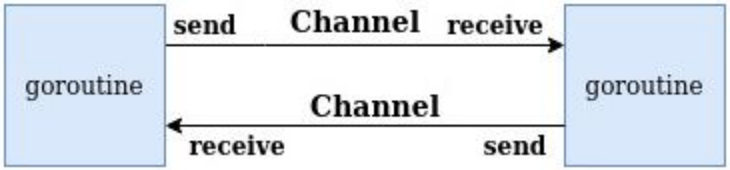
Concurrency & Parallelism

- Goroutines are concurrent, which means they may not execute at the exact same time
- Go's scheduler determines when a goroutine gets CPU time
- If you want parallelism, you can use multiple goroutines to take advantage of multi-core CPUs



Communication

- Go provides channels for communication between goroutines
- Channels are used to pass data from one goroutine to another in a safe and synchronized manner
- They help prevent race conditions and facilitate inter-goroutine communication



Channels

Channels

- A channel is a typed conduit through which you can send and receive values with the <- operator
- Channels can be thought of as pipes that connect goroutines, allowing them to communicate and synchronize their execution.

```go
package main

import (
    "fmt"
    "time"
)

func hello(done chan bool) {
    fmt.Println("hello go routine is going to sleep")
    time.Sleep(4 * time.Second)
    fmt.Println("hello go routine awake and going to write to done")
    done <- true
}
func main() {
    done := make(chan bool)
    fmt.Println("Main going to call hello go goroutine")
    go hello(done)
    <-done
    fmt.Println("Main received data")
}

/*
    Main going to call hello go goroutine
    hello go routine is going to sleep
    hello go routine awake and going to write to done
    Main received data
*/
```

DWARVES FOUNDATION

Channels

- Channels are created using the make() function with the chan keyword followed by the type of values that will be passed through the channel
- You can send values into a channel using the <- operator and receive values from a channel in a similar way

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

```
ch <- value            // Send value into the channel
receivedValue := <-ch    // Receive value from the channel
```

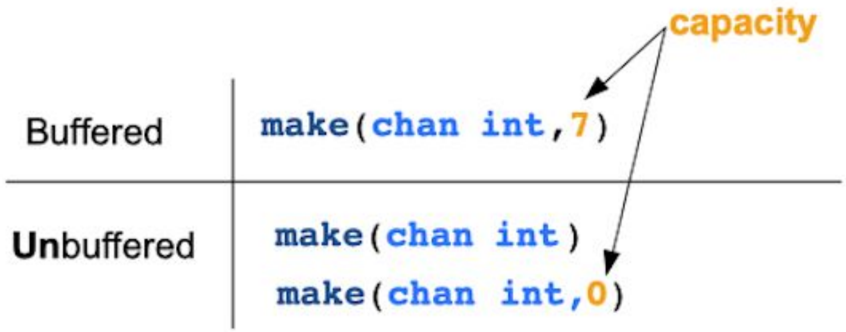
Channels

- Blocking nature of channels: Channel operations block by default until both the sender and receiver are ready
- If the sending or receiving operation is not ready, the goroutine will pause its execution until it becomes available
- This behavior enables synchronization and prevents race conditions

	NIL	Open	Closed
Send	Blocked	Allowed	Panic
Receive	Blocked	Allowed	Allowed

Buffered channels

- By specifying a buffer size when creating a channel, you can create buffered channels
- Buffered channels allow a certain number of values to be sent into the channel without blocking
- Once the buffer is full, subsequent sends will block until there is space in the buffer, or until a receiver is available



DWARVES FOUNDATION

Closing Channels

- Channels can be closed to indicate that no more values will be sent
- Receivers can check if a channel is closed using the multiple assignment form of the receive operation
- Closed channels will not cause panics if values are received from them

```go
package main

import (
	"fmt"
)

func producer(chnl chan int) {
	for i := 0; i < 10; i++ {
		chnl <- i
	}
	close(chnl)
}
func main() {
	ch := make(chan int)
	go producer(ch)
	for {
		v, ok := <-ch
		if ok == false {
			break
		}
		fmt.Println("Received ", v, ok)
	}
}
```

DWARVES FOUNDATION

Closing Channels - Range

- Channels can also be closed and looped through ranges
- Ranges will stop once the for range reaches the maximum range of the goroutine loop

```
// Received  0
// Received  1
// Received  2
// Received  3
// Received  4
// Received  5
// Received  6
// Received  7
// Received  8
// Received  9
```

```go
package main

import (
	"fmt"
)

func producer(chnl chan int) {
	for i := 0; i < 10; i++ {
		chnl <- i
	}
	close(chnl)
}
func main() {
	ch := make(chan int)
	go producer(ch)
	for v := range ch {
		fmt.Println("Received ",v)
	}
}
```

DWARVES FOUNDATION

Channels - Select

- The select statement allows you to wait on multiple channel operations simultaneously
- It is useful when you want to perform different actions based on which channel is ready to send or receive data
- Channels can be specified as either send-only (chan<-) or receive-only (<-chan)
- This feature helps enforce the correct usage of channels by restricting certain operations

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    sendOnlyChan := make(chan<- string) // channel with send-only direction
    receiveOnlyChan := make(<-chan string) // channel with receive-only direction

    // Send-only goroutine
    go func(ch chan<- string) {
        time.Sleep(2 * time.Second)
        ch <- "Hello from send-only goroutine!"
    }(sendOnlyChan)

    // Receive-only goroutine
    go func(ch <-chan string) {
        msg := <-ch
        fmt.Println(msg)
    }(receiveOnlyChan)

    select {
    case sendOnlyChan <- "Hello from main goroutine!": // Sending to send-only channel
        fmt.Println("Sent message to send-only channel")
    case msg := <-receiveOnlyChan: // Receiving from receive-only channel
        fmt.Println("Received message from receive-only channel:", msg)
    case <-time.After(3 * time.Second): // Timeout after 3 seconds
        fmt.Println("Timeout!")
    }
}
```

Synchronization

Wait Groups and Mutexes

DWARVES FOUNDATION

Synchronization

- Synchronization ensures that only one goroutine accesses a shared resource at a time or that certain tasks are completed before others proceed
- It helps maintain consistency and avoid conflicts between concurrent operations
- Example about race condition when multiple goroutines access the counter variable without and sync mechanism

```go
package main

import (
    "fmt"
    "sync"
)

var counter int

func incrementCounter(wg *sync.WaitGroup) {
    defer wg.Done()
    counter++
}

func main() {
    var wg sync.WaitGroup
    wg.Add(10)

    for i := 0; i < 10; i++ {
        go incrementCounter(&wg)
    }

    wg.Wait()

    fmt.Println("Final Counter Value:", counter)
}
```

Wait Groups

- Wait groups in Go provide a simple way to synchronize goroutines
- A Wait group allows you to wait for the completion of all goroutines before proceeding further in the program
- It provides a mechanism to block the execution of the main goroutine until the specified number of goroutines has finished their work

```go
package main

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

func worker(id int, wg *sync.WaitGroup) {
    defer wg.Done()

    fmt.Printf("Worker %d starting\n", id)
    time.Sleep(time.Second)
    fmt.Printf("Worker %d done\n", id)
}

func main() {
    var wg sync.WaitGroup

    for i := 1; i <= 5; i++ {
        wg.Add(1)
        go worker(i, &wg)
    }

    wg.Wait()

    fmt.Println("All workers have completed their work")
}
```

Mutexes

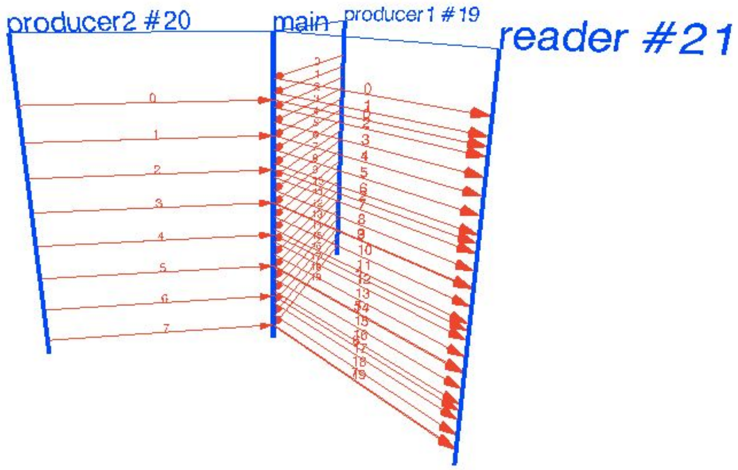
- They allow you to protect shared resources from being accessed simultaneously by multiple goroutines
- A mutex has two states, locked and unlocked
- When a goroutine requests the lock on a mutex, it becomes the owner and gains exclusive access to the protected resource
- Other goroutines trying to acquire the lock will be blocked until the current lock is released

```go
package main

import (
    "fmt"
    "sync"
)

var counter int
var mutex sync.Mutex

func incrementCounter(wg *sync.WaitGroup) {
    defer wg.Done()

    mutex.Lock()
    counter++
    mutex.Unlock()
}

func main() {
    var wg sync.WaitGroup
    wg.Add(10)

    for i := 0; i < 10; i++ {
        go incrementCounter(&wg)
    }

    wg.Wait()

    fmt.Println("Final Counter Value:", counter)
}
```

Concurrent Patterns

Producer-Consumer

- In this pattern, one or more goroutines (producers) generate data or tasks, which are then consumed by one or more goroutines (consumers)
- Channels are commonly used to establish communication between producers and consumers
- This pattern enables efficient coordination and sharing of work among multiple goroutines



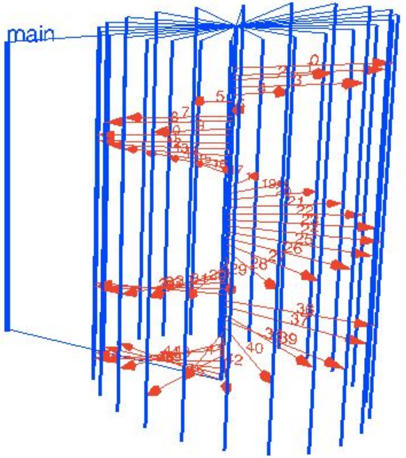
DWARVES FOUNDATION

Producer-Consumer

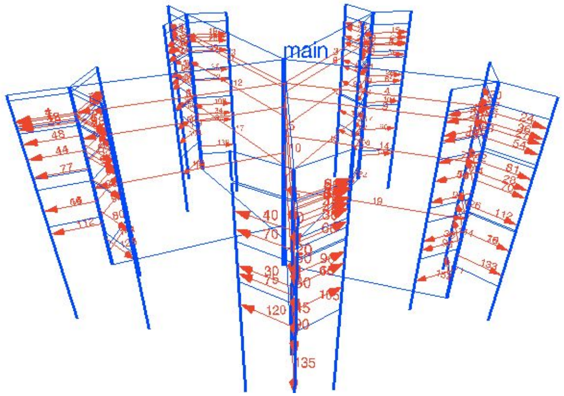
- The producer function sends integers 1 to 5 to the channel ch, while the consumer function receives the data from the channel and processes it by printing
- When you run this program, you will see the consumer processing the data produced by the producer
- The order of processing may vary as per the goroutine scheduler

```go
package main

import "fmt"

func producer(ch chan<- int) {
    defer close(ch)
    for i := 1; i <= 5; i++ {
        ch <- i // Send data to the channel
    }
}

func consumer(ch <-chan int) {
    for item := range ch {
        fmt.Println("Processed:", item)
    }
}

func main() {
    ch := make(chan int)

    go producer(ch) // Start the producer goroutine
    go consumer(ch) // Start the consumer goroutine

    // Wait for the producer and consumer to finish
    var input string
    fmt.Scanln(&input)
}
```

Worker Pools



- The worker pool pattern helps manage resource usage, prevents overwhelming the system with a large number of goroutines, and allows for controlled parallel processing of tasks
- It is commonly used in scenarios where there's a need to process a large amount of work concurrently while limiting the number of concurrent operations



DWARVES FOUNDATION

Worker Pools - Implementation

- Task Queue: The first step is to create a task queue, which is a data structure that holds the tasks that need to be processed. This can be implemented using a Go channel or a custom data structure
- Worker Creation: Next, a fixed number of worker goroutines are created. These workers are responsible for fetching tasks from the task queue and processing them. The number of workers in the pool is usually determined based on factors like the available resources and the desired level of parallelism.
- Task Distribution: As tasks arrive, they are added to the task queue. The workers continuously check the task queue for new tasks. When a worker fetches a task from the queue, it processes it
- Synchronization: To ensure that all tasks are completed before the program exits, some form of synchronization is needed. This can be achieved using a sync.WaitGroup or other synchronization primitives provided by the Go standard library.

Worker Pools

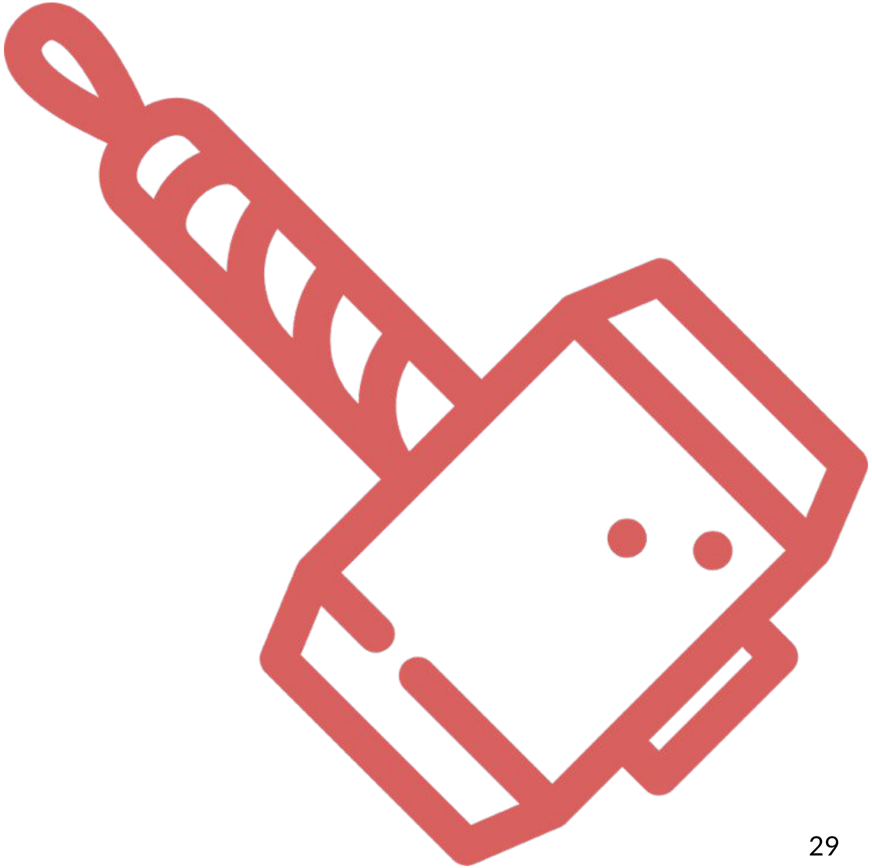
- In this example, numWorkers represents the number of worker goroutines we want in our pool, and numTasks represents the total number of tasks we want to process
- The worker function is responsible for processing each task, and the main function creates the workers, adds tasks to the task queue, and waits for all the workers to finish

```go
package main

import (
    "fmt"
    "sync"
)

type Task struct {
    ID  int
    Msg string
}

func worker(id int, tasks <-chan Task, wg *sync.WaitGroup) {
    defer wg.Done()

    for task := range tasks {
        fmt.Printf("Worker %d processing task %d: %s\n", id, task.ID, task.Msg)
        // Process the task here
    }
}

func main() {
    const numWorkers = 3
    const numTasks = 10

    var wg sync.WaitGroup
    wg.Add(numWorkers)

    tasks := make(chan Task)

    // Create worker goroutines
    for i := 1; i <= numWorkers; i++ {
        go worker(i, tasks, &wg)
    }

    // Add tasks to the task queue
    for i := 1; i <= numTasks; i++ {
        tasks <- Task{ID: i, Msg: fmt.Sprintf("Task %d", i)}
    }

    close(tasks) // Close the task channel to signal that no more tasks will be added

    wg.Wait() // Wait for all workers to finish

    fmt.Println("All tasks processed")
}
```

Demo

Demo - Zer0 to Hero

- Step 1
- Step 2



Reference

Resources & Reference links

- https://divan.dev/posts/go_concurrency_visualize/
- https://golangbot.com/goroutines/
- https://golangbot.com/channels/

DWARVES
FOUNDATION

Thank You

DWARVES
FOUNDATION

Q&A