Go Concurrency Complete Guide 🚀

1. Goroutines

What it is:

Lightweight threads managed by Go runtime. Just add `go` keyword before any function call!

Problem it solves:

- Run multiple tasks at the same time
- Don't wait for slow operations to complete
- Better performance and responsiveness

Use cases:

- Web servers handling multiple requests
- Background tasks
- Parallel processing
- Non-blocking operations

Simple Example:

```
package main

import (
	"fmt"
	"time"
)

func sayHello(name string) {
	fmt.Println("hello from ", name)
}

func main() {
	sayHello("main call")		// Normal call - waits

	go sayHello("gorutine 1")	// Goroutine - runs in background
	go sayHello("gorutine 2")	// Goroutine - runs in background
	go sayHello("gorutine 3")	// Goroutine - runs in background

	fmt.Println("main is running")
	time.Sleep(1 * time.Second)	// Give goroutines time to finish
	fmt.Println("Main function ending")
}
```

Key Points:

- `go function()` creates a goroutine
- Main goroutine controls program life
- If main exits, all goroutines die
- Order of execution is unpredictable

---

2. Channels

What it is:

Pipes that allow goroutines to communicate with each other safely.

Problem it solves:

- How to send data between goroutines?
- How to coordinate goroutines?
- How to avoid race conditions?

Use cases:

- Passing data between goroutines
- Signaling when work is done
- Producer-consumer patterns
- Event notifications

Simple Example:

```
package main

import (
	"fmt"
	"time"
)

// Baker goroutine
func baker(ch chan string) {
	fmt.Println("Baker: Making bread...")
	time.Sleep(2 * time.Second)
	ch <- "Fresh bread ready!" // Send to customer
}

// Customer goroutine
func customer(ch chan string) {
	fmt.Println("Customer: Waiting for bread...")
	bread := <-ch // Wait for baker
	fmt.Println("Customer got:", bread)
}

func main() {
	ch := make(chan string)    // Create channel

	go baker(ch)               // Goroutine 1
	go customer(ch)            // Goroutine 2

	time.Sleep(3 * time.Second)
	fmt.Println("Shop closed!")
}
```

Key Points:

- `ch <- data` sends data to channel
- `data := <-ch` receives data from channel
- Receiving blocks until data is available
- Channels synchronize goroutines

---

3. Buffered vs Unbuffered Channels

Unbuffered Channels:

```
ch := make(chan string)        // Buffer size = 0
```

- **Synchronous**: Sender waits until receiver is ready
- Direct handoff between goroutines
- Like passing a ball hand-to-hand

Buffered Channels:

```
ch := make(chan string, 3)     // Buffer size = 3
```

- **Asynchronous**: Sender can continue until buffer is full
- Can store multiple values
- Like a mailbox that holds messages

What happens when buffer is full?

```
ch := make(chan string, 2)     // Buffer size 2

ch <- "msg1"                   // ✅ Goes in buffer
ch <- "msg2"                   // ✅ Goes in buffer
ch <- "msg3"                   // 🔴 BLOCKS! Waits for space
```

- **No error occurs**
- Sender **blocks** (waits) until receiver takes a message
- Then sender can continue

Use cases:

- **Unbuffered**: When you need tight synchronization
- **Buffered**: When you want to smooth out timing differences
- **Small buffer (1-10)**: For signaling and coordination
- **Large buffer**: For high-throughput producer-consumer scenarios

---

4. WaitGroup

What it is:

A smart counter that tracks how many goroutines are still running.

Problem it solves:

- "I started 5 workers, how do I know when ALL are done?"
- No more guessing with `time.Sleep()`
- Precise coordination of multiple goroutines

How it works:

- `wg.Add(n)`: "I'm expecting n more workers to finish" (counter += n)
- `wg.Done()`: "One worker finished" (counter -= 1)
- `wg.Wait()`: "Wait until counter reaches 0"

Use cases:

- Wait for all background tasks to complete
- Coordinating multiple workers
- Batch processing
- Parallel computations

Simple Example:

```
package main

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

func worker(id int, wg *sync.WaitGroup) {
	defer wg.Done()                  // Mark done when function exits

	fmt.Printf("Worker %d working...\n", id)
	time.Sleep(2 * time.Second)      // Simulate work
	fmt.Printf("Worker %d finished!\n", id)
}

func main() {
	var wg sync.WaitGroup

	// Start 4 workers
	for i := 1; i <= 4; i++ {
		wg.Add(1)                    // "Expecting 1 more worker"
		go worker(i, &wg)            // Start worker
	}

	fmt.Println("Waiting for all workers...")
	wg.Wait()                        // Wait until all call Done()
	fmt.Println("All workers finished!")
}
```

Key Points:

- Always use `defer wg.Done()` to ensure it's called
- `Add()` and `Done()` calls must match
- `Wait()` blocks until all goroutines finish
- Pass WaitGroup by pointer (`*sync.WaitGroup`)

---

5. Additional Concepts

Mutex (sync.Mutex)

Problem: Multiple goroutines accessing shared data (race condition) **Solution**: Mutex provides exclusive access

```
var mu sync.Mutex
var counter int

func increment(wg *sync.WaitGroup) {
    defer wg.Done()
    mu.Lock()         // Lock before accessing shared data
    counter++         // Safe access
    mu.Unlock()       // Unlock when done
}
```

Select Statement

Problem: Working with multiple channels **Solution**: Select lets you handle multiple channel operations

```
select {
case msg := <-ch1:
    fmt.Println("Got from ch1:", msg)
case msg := <-ch2:
    fmt.Println("Got from ch2:", msg)
case <-time.After(1 * time.Second):
    fmt.Println("Timeout!")
}
```

Context (context.Context)

Problem: How to cancel long-running operations? **Solution**: Context provides cancellation and timeouts

```
ctx, cancel := context.WithTimeout(context.Background(), 2*time.Second)
defer cancel()

// Use ctx in goroutines to handle cancellation
```

Real-World Example Pattern

```
package main

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

// Typical pattern: Producer-Consumer with WaitGroup
func producer(jobs chan<- int, wg *sync.WaitGroup) {
    defer wg.Done()
    for i := 1; i <= 5; i++ {
        fmt.Printf("Producing job %d\n", i)
        jobs <- i
        time.Sleep(100 * time.Millisecond)
    }
    close(jobs)  // Signal no more jobs
}

func consumer(id int, jobs <-chan int, wg *sync.WaitGroup) {
    defer wg.Done()
    for job := range jobs {  // Receive until channel closed
        fmt.Printf("Consumer %d processing job %d\n", id, job)
        time.Sleep(200 * time.Millisecond)
    }
}

func main() {
    jobs := make(chan int, 3)  // Buffered channel
    var wg sync.WaitGroup

    // Start producer
    wg.Add(1)
    go producer(jobs, &wg)

    // Start consumers
    for i := 1; i <= 2; i++ {
        wg.Add(1)
        go consumer(i, jobs, &wg)
    }

    wg.Wait()  // Wait for all to finish
    fmt.Println("All done!")
}
```