Pipeline Pattern

What is a Pipeline?

Pipeline: A series of stages connected by channels, where each stage:

1. Receives values from upstream via input channel
2. Performs transformation/processing
3. Sends results downstream via output channel

Purpose:

- **Decomposition:** Break complex processing into simple stages
- **Parallelism:** Each stage runs concurrently
- **Streaming:** Process data as it arrives (no buffering entire dataset)

Analogy: Assembly line where each station does one thing, passes to next.

Basic Pipeline

```
// Stage 1: Generate numbers
func generator(nums ...int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for _, n := range nums {
            out <- n
        }
    }()
    return out
}

// Stage 2: Square numbers
func square(in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for n := range in {
            out <- n * n
        }
    }()
    return out
}

// Stage 3: Filter even numbers
func filterEven(in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for n := range in {
            if n%2 == 0 {
```

```
                out <- n
            }
        }
    }()
    return out
}

// Build pipeline
func main() {
    // Generate → Square → Filter
    numbers := generator(1, 2, 3, 4, 5)
    squared := square(numbers)
    evens := filterEven(squared)

    // Consume
    for n := range evens {
        fmt.Println(n)  // Prints: 4, 16
    }
}
```

Key properties:

- Each stage is independent goroutine
- Channels connect stages
- Closing channel signals "no more data"
- Range loop naturally handles closed channels

Advanced Pipeline with Context

```
// Generic stage with cancellation
func stage(
    ctx context.Context,
    in <-chan int,
    fn func(int) int,
) <-chan int {
    out := make(chan int)

    go func() {
        defer close(out)
        for {
            select {
            case n, ok := <-in:
                if !ok {
                    return  // Input closed
                }

                result := fn(n)

                select {
                case out <- result:
                case <-ctx.Done():
```

```
                    return
                }

            case <-ctx.Done():
                return
            }
        }
    }()

    return out
}

// Usage with cancellation
func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()

    numbers := generator(ctx, 1, 2, 3, 4, 5)
    squared := stage(ctx, numbers, func(n int) int { return n * n })
    doubled := stage(ctx, squared, func(n int) int { return n * 2 })

    for n := range doubled {
        fmt.Println(n)
    }
}
```

Pipeline with Multiple Workers per Stage

```
// Fan-out stage: Multiple workers process in parallel
func fanOutStage(in <-chan int, numWorkers int) <-chan int {
    outputs := make([]<-chan int, numWorkers)

    // Start workers
    for i := 0; i < numWorkers; i++ {
        out := make(chan int)
        outputs[i] = out

        go func(ch chan<- int) {
            defer close(ch)
            for n := range in {
                result := expensiveComputation(n)
                ch <- result
            }
        }(out)
    }

    // Fan-in: Merge worker outputs
    return fanIn(outputs...)
}
```

```
// Complete pipeline with parallel stage
func main() {
    input := generator(1, 2, 3, 4, 5)

    // Fast stage
    preprocessed := preprocess(input)

    // Slow stage (fan-out to parallelize)
    processed := fanOutStage(preprocessed, 10)

    // Fast stage
    postprocessed := postprocess(processed)

    for result := range postprocessed {
        fmt.Println(result)
    }
}
```

Real-World Example: Data Processing Pipeline

```
package main

import (
    "context"
    "encoding/csv"
    "encoding/json"
    "fmt"
    "os"
    "strings"
)

type Record struct {
    ID    string
    Name  string
    Value float64
}

// Stage 1: Read CSV file
func readCSV(ctx context.Context, filename string) <-chan []string {
    out := make(chan []string)

    go func() {
        defer close(out)

        file, err := os.Open(filename)
        if err != nil {
            return
        }
        defer file.Close()
```

```
        reader := csv.NewReader(file)
        records, _ := reader.ReadAll()

        for _, record := range records {
            select {
            case out <- record:
            case <-ctx.Done():
                return
            }
        }
    }()

    return out
}

// Stage 2: Parse records
func parseRecords(ctx context.Context, in <-chan []string) <-chan Record {
    out := make(chan Record)

    go func() {
        defer close(out)

        for fields := range in {
            if len(fields) < 3 {
                continue
            }

            record := Record{
                ID:    fields[0],
                Name:  fields[1],
                Value: parseFloat(fields[2]),
            }

            select {
            case out <- record:
            case <-ctx.Done():
                return
            }
        }
    }()

    return out
}

// Stage 3: Filter invalid records
func filterValid(ctx context.Context, in <-chan Record) <-chan Record {
    out := make(chan Record)

    go func() {
        defer close(out)

        for record := range in {
```

```
            if record.Value > 0 {
                select {
                case out <- record:
                case <-ctx.Done():
                    return
                }
            }
        }
    }()

    return out
}

// Stage 4: Transform (normalize)
func normalize(ctx context.Context, in <-chan Record) <-chan Record {
    out := make(chan Record)

    go func() {
        defer close(out)

        for record := range in {
            record.Name = strings.ToUpper(record.Name)
            record.Value = record.Value / 100.0

            select {
            case out <- record:
            case <-ctx.Done():
                return
            }
        }
    }()

    return out
}

// Stage 5: Write JSON output
func writeJSON(ctx context.Context, in <-chan Record, filename string) <-chan error
{
    errs := make(chan error, 1)

    go func() {
        defer close(errs)

        file, err := os.Create(filename)
        if err != nil {
            errs <- err
            return
        }
        defer file.Close()

        encoder := json.NewEncoder(file)
```

```
        count := 0
        for record := range in {
            if err := encoder.Encode(record); err != nil {
                errs <- err
                return
            }
            count++
        }

        fmt.Printf("Wrote %d records\n", count)
    }()

    return errs
}

func main() {
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()

    // Build pipeline
    csvRecords := readCSV(ctx, "input.csv")
    parsed := parseRecords(ctx, csvRecords)
    valid := filterValid(ctx, parsed)
    normalized := normalize(ctx, valid)
    errs := writeJSON(ctx, normalized, "output.json")

    // Wait for completion or error
    if err := <-errs; err != nil {
        fmt.Fprintf(os.Stderr, "Error: %v\n", err)
    }
}
```

Pipeline with Buffering

```
// Buffered stages for throughput
func bufferedPipeline() {
    numbers := generator(1, 2, 3, 4, 5)

    // Buffer between stages
    buffered := make(chan int, 100)
    go func() {
        defer close(buffered)
        for n := range numbers {
            buffered <- n
        }
    }()

    squared := square(buffered)

    // Heavy consumer
```

```
    for n := range squared {
        time.Sleep(100 * time.Millisecond)  // Slow
        fmt.Println(n)
    }
}
```

When to buffer:

- Stage speeds vary significantly
- Want to decouple fast producer from slow consumer
- Smooth out bursty traffic

Trade-off: Memory usage vs. throughput

Pipeline Cancellation Patterns

Pattern 1: Early Termination

```
func pipeline(ctx context.Context) <-chan Result {
    stage1 := generateData(ctx)
    stage2 := process(ctx, stage1)
    stage3 := filter(ctx, stage2)

    return stage3
}

func main() {
    ctx, cancel := context.WithCancel(context.Background())

    results := pipeline(ctx)

    // Cancel after 5 results
    count := 0
    for result := range results {
        fmt.Println(result)
        count++
        if count >= 5 {
            cancel()  // Stops all stages
            break
        }
    }
}
```

Pattern 2: Drain on Cancel

```
func drainablePipeline(ctx context.Context) <-chan Result {
    out := make(chan Result)

    go func() {
        defer close(out)
```

```
        // Process data
        for {
            result := processNext()

            select {
            case out <- result:
            case <-ctx.Done():
                // Drain remaining work
                for hasMore() {
                    processNext()
                }
                return
            }
        }
    }()

    return out
}
```

Real-World Example: Image Processing Pipeline

```
type ImageData struct {
    ID       string
    Data     []byte
    Width    int
    Height   int
}

// Pipeline: Load → Decode → Resize → Compress → Save
func imageProcessingPipeline(ctx context.Context, filenames []string) {
    // Stage 1: Load files
    files := loadFiles(ctx, filenames)

    // Stage 2: Decode images (CPU-intensive, parallelize)
    decoded := parallelDecode(ctx, files, 4)

    // Stage 3: Resize
    resized := resize(ctx, decoded, 800, 600)

    // Stage 4: Compress
    compressed := compress(ctx, resized)

    // Stage 5: Save
    saved := save(ctx, compressed, "output/")

    // Wait for completion
    for s := range saved {
        fmt.Printf("Saved: %s\n", s)
    }
```

```
}

func parallelDecode(ctx context.Context, in <-chan string, workers int) <-chan
ImageData {
    out := make(chan ImageData)

    var wg sync.WaitGroup
    for i := 0; i < workers; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            for filename := range in {
                img := decodeImage(filename)
                select {
                case out <- img:
                case <-ctx.Done():
                    return
                }
            }
        }()
    }

    go func() {
        wg.Wait()
        close(out)
    }()

    return out
}
```

Common Mistakes

Mistake 1: Not Closing Channels

```
// WRONG: Channel never closed
func generator(nums ...int) <-chan int {
    out := make(chan int)
    go func() {
        for _, n := range nums {
            out <- n
        }
        // Missing: close(out)
    }()
    return out
}

// Consumer blocks forever after last value
for n := range generator(1, 2, 3) {
    fmt.Println(n)
```

```
}
// Deadlock: waits for more values
```

Mistake 2: Goroutine Leaks on Early Exit

```
// WRONG: Goroutines leak if consumer exits early
func pipeline() {
    stage1 := generate()
    stage2 := process(stage1)

    // Consumer exits early
    for n := range stage2 {
        if n > 10 {
            break  // stage1 and stage2 goroutines still running!
        }
    }
}
```

Fix: Use context

```
func pipeline(ctx context.Context) {
    stage1 := generate(ctx)
    stage2 := process(ctx, stage1)

    for n := range stage2 {
        if n > 10 {
            cancel()  // Signals all stages to stop
            break
        }
    }
}
```

Mistake 3: Blocking on Output Send

```
// WRONG: Goroutine blocks if no consumer
func stage(in <-chan int) <-chan int {
    out := make(chan int)  // Unbuffered
    go func() {
        defer close(out)
        for n := range in {
            out <- compute(n)  // Blocks if no consumer
        }
    }()
    return out
}
```

Fix: Check context

```go
func stage(ctx context.Context, in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for n := range in {
            result := compute(n)
            select {
            case out <- result:
            case <-ctx.Done():
                return
            }
        }
    }()
    return out
}
```

Performance Optimization

Benchmark Pipeline Stages

```go
func BenchmarkPipeline(b *testing.B) {
    for _, numWorkers := range []int{1, 2, 4, 8} {
        b.Run(fmt.Sprintf("Workers-%d", numWorkers), func(b *testing.B) {
            ctx := context.Background()
            in := generator(ctx, b.N)
            out := parallelStage(ctx, in, numWorkers)

            for range out {
                // Drain
            }
        })
    }
}
```

Profile Bottlenecks

```
go test -cpuprofile=cpu.out -bench=.
go tool pprof cpu.out
```

Look for:

- Hot functions (CPU bottlenecks)
- Channel operations (contention)
- Memory allocations

Interview Questions

Q: "What's difference between pipeline and fan-out?"

"Pipeline chains stages sequentially—output of one is input to next. Each stage transforms data. Fan-out distributes same input to multiple workers for parallel processing. Pipeline is about decomposition; fan-out is about parallelism. Can combine: parallelize slow stages within pipeline."

Q: "How do you handle errors in pipeline?"

"Include error channel alongside data channel: return both `<-chan Data` and `<-chan error`. Or wrap in result type: `type Result struct { Data T; Err error }`. Consumer checks for errors. For early termination, use context cancellation—cancel on first error."

Q: "When should stages be buffered?"

"Buffer when stage speeds differ significantly. Example: fast producer, slow consumer—buffer smooths bursts. Buffer size: 10-100 items typically. Trade-off: memory for throughput. Unbuffered provides natural backpressure."

Key Takeaways

1. **Pipeline = chain of stages connected by channels**
2. **Each stage: goroutine reading input, writing output**
3. **Always close output channel** when done
4. **Use context for cancellation** across all stages
5. **Parallelize slow stages** with fan-out
6. **Buffer between stages** to smooth throughput
7. **Check ctx.Done() when sending** to prevent leaks
8. **Fan-out slow stages, sequential fast stages**
9. **Profile to find bottlenecks** (don't guess)
10. **Error handling: separate error channel or Result type**

Exercises

1. Build CSV → Parse → Filter → JSON pipeline.
2. Add parallel decode stage (4 workers) to image pipeline.
3. Implement pipeline with timeout: cancel if not done in 10s.
4. Profile pipeline, identify bottleneck, add parallelism.
5. Build log processing pipeline: Read → Parse → Aggregate → Store.

Next: semaphore.md - Limiting resource access with semaphores.