Fan-In and Fan-Out Patterns

What are Fan-In and Fan-Out?

Fan-out: Distributing work from one channel to multiple goroutines (workers).

Fan-in: Combining results from multiple channels into one channel (merge).

Purpose:

- **Parallelism:** Process data concurrently across multiple workers
- **Aggregation:** Collect results from distributed computation
- **Load distribution:** Split work evenly across resources

Fan-Out Pattern

Basic Fan-Out

```
func fanOut(input <-chan Task, numWorkers int) []<-chan Result {
    outputs := make([]<-chan Result, numWorkers)

    for i := 0; i < numWorkers; i++ {
        ch := make(chan Result)
        outputs[i] = ch

        go func(out chan<- Result) {
            defer close(out)
            for task := range input {
                result := process(task)
                out <- result
            }
        }(ch)
    }

    return outputs
}

// Usage:
input := make(chan Task, 100)
outputs := fanOut(input, 5)  // 5 workers

// All workers read from same input channel
// Each task processed by exactly one worker
```

How it works:

1. Single input channel shared by all workers
2. First available worker takes task (FIFO queue behavior)
3. Each worker sends results to its own output channel

Fan-Out with Round-Robin

```go
func fanOutRoundRobin(input <-chan Task, numWorkers int) []chan<- Task {
    workers := make([]chan<- Task, numWorkers)

    for i := 0; i < numWorkers; i++ {
        ch := make(chan Task)
        workers[i] = ch

        go func(tasks <-chan Task) {
            for task := range tasks {
                process(task)
            }
        }(ch)
    }

    // Distributor goroutine
    go func() {
        i := 0
        for task := range input {
            workers[i%numWorkers] <- task
            i++
        }

        // Close all worker channels
        for _, w := range workers {
            close(w)
        }
    }()

    return workers
}
```

Difference from basic fan-out:

- Basic: Workers compete for tasks (first-come-first-served)
- Round-robin: Tasks explicitly distributed (load-balanced)

Fan-In Pattern

Basic Fan-In

```go
func fanIn(inputs ...<-chan Result) <-chan Result {
    output := make(chan Result)
    var wg sync.WaitGroup

    // Multiplex each input channel
    multiplex := func(input <-chan Result) {
        defer wg.Done()
        for result := range input {
            output <- result
        }
    }
```

```
    wg.Add(len(inputs))
    for _, input := range inputs {
        go multiplex(input)
    }

    // Close output when all inputs done
    go func() {
        wg.Wait()
        close(output)
    }()

    return output
}

// Usage:
outputs := fanOut(input, 5)  // 5 worker channels
merged := fanIn(outputs...)   // Single output channel

for result := range merged {
    handle(result)
}
```

Fan-In with Select (Deterministic)

```
func fanInSelect(inputs ...<-chan Result) <-chan Result {
    output := make(chan Result)

    go func() {
        defer close(output)

        // Create cases for select
        cases := make([]reflect.SelectCase, len(inputs))
        for i, ch := range inputs {
            cases[i] = reflect.SelectCase{
                Dir:  reflect.SelectRecv,
                Chan: reflect.ValueOf(ch),
            }
        }

        // Multiplex until all channels closed
        for len(cases) > 0 {
            chosen, value, ok := reflect.Select(cases)
            if !ok {
                // Channel closed, remove from cases
                cases = append(cases[:chosen], cases[chosen+1:]...)
                continue
            }

            output <- value.Interface().(Result)
        }
```

```
    }()

    return output
}
```

Fan-In with Priority

```
func fanInPriority(high, low <-chan Result) <-chan Result {
    output := make(chan Result)

    go func() {
        defer close(output)

        for {
            select {
            case result, ok := <-high:
                if !ok {
                    high = nil  // Disable high channel
                    continue
                }
                output <- result

            default:
                select {
                case result, ok := <-high:
                    if !ok {
                        high = nil
                        continue
                    }
                    output <- result

                case result, ok := <-low:
                    if !ok {
                        low = nil
                        continue
                    }
                    output <- result
                }
            }

            // Both channels closed
            if high == nil && low == nil {
                return
            }
        }
    }()

    return output
}
```

Combined Fan-Out/Fan-In Pattern

```
func processPipeline(input <-chan Task, numWorkers int) <-chan Result {
    // Fan-out: Distribute to workers
    workerOutputs := make([]<-chan Result, numWorkers)

    for i := 0; i < numWorkers; i++ {
        ch := make(chan Result)
        workerOutputs[i] = ch

        go func(out chan<- Result) {
            defer close(out)
            for task := range input {
                result := process(task)
                out <- result
            }
        }(ch)
    }

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

// Usage:
input := make(chan Task, 100)
results := processPipeline(input, 10)

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

Real-World Example: Image Processing

```
package main

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

type ImageTask struct {
    ID   string
    Path string
}

type ImageResult struct {
    ID        string
```

```
    Thumbnail image.Image
    Err       error
}

// Fan-out: Distribute image tasks to workers
func imageWorker(tasks <-chan ImageTask, results chan<- ImageResult, wg
*sync.WaitGroup) {
    defer wg.Done()

    for task := range tasks {
        start := time.Now()

        // Load and process image
        img := loadImage(task.Path)
        thumbnail := resize(img, 200, 200)

        results <- ImageResult{
            ID:        task.ID,
            Thumbnail: thumbnail,
        }

        fmt.Printf("Processed %s in %v\n", task.ID, time.Since(start))
    }
}

func processImages(imagePaths []string, numWorkers int) []ImageResult {
    tasks := make(chan ImageTask, len(imagePaths))
    results := make(chan ImageResult, len(imagePaths))

    var wg sync.WaitGroup

    // Fan-out: Start workers
    for i := 0; i < numWorkers; i++ {
        wg.Add(1)
        go imageWorker(tasks, results, &wg)
    }

    // Send tasks
    go func() {
        for i, path := range imagePaths {
            tasks <- ImageTask{
                ID:   fmt.Sprintf("image-%d", i),
                Path: path,
            }
        }
        close(tasks)
    }()

    // Wait for workers and close results
    go func() {
        wg.Wait()
        close(results)
```

```
    }()

    // Fan-in: Collect results
    var allResults []ImageResult
    for result := range results {
        allResults = append(allResults, result)
    }

    return allResults
}
```

Real-World Example: Log Aggregation

```
type LogEntry struct {
    Timestamp time.Time
    Level     string
    Message   string
}

// Fan-in: Merge logs from multiple sources
func aggregateLogs(sources ...<-chan LogEntry) <-chan LogEntry {
    output := make(chan LogEntry)
    var wg sync.WaitGroup

    // Start goroutine for each source
    for _, source := range sources {
        wg.Add(1)
        go func(logs <-chan LogEntry) {
            defer wg.Done()
            for entry := range logs {
                output <- entry
            }
        }(source)
    }

    // Close output when all sources exhausted
    go func() {
        wg.Wait()
        close(output)
    }()

    return output
}

// Usage: Collect from multiple services
func main() {
    serviceA := readLogs("service-a.log")
    serviceB := readLogs("service-b.log")
    serviceC := readLogs("service-c.log")
```

```
    allLogs := aggregateLogs(serviceA, serviceB, serviceC)

    // Process merged stream
    for entry := range allLogs {
        // Store in database, index, etc.
        store(entry)
    }
}
```

Pattern: Bounded Fan-Out (Semaphore)

```
func boundedFanOut(input <-chan Task, maxConcurrency int) <-chan Result {
    results := make(chan Result)
    semaphore := make(chan struct{}, maxConcurrency)

    go func() {
        defer close(results)
        var wg sync.WaitGroup

        for task := range input {
            semaphore <- struct{}{}  // Acquire
            wg.Add(1)

            go func(t Task) {
                defer wg.Done()
                defer func() { <-semaphore }()  // Release

                result := process(t)
                results <- result
            }(task)
        }

        wg.Wait()
    }()

    return results
}
```

Common Mistakes

Mistake 1: Forgetting to Close Input Channel

```
// WRONG: Workers block forever
input := make(chan Task)
outputs := fanOut(input, 5)

// Sends tasks but never closes channel
for _, task := range tasks {
    input <- task
```

```
}
// Workers wait forever on: for task := range input
```

Fix:

```
for _, task := range tasks {
    input <- task
}
close(input)  // Signal no more tasks
```

Mistake 2: Not Waiting for Workers Before Closing Output

```
// WRONG: Race condition
func fanIn(inputs ...<-chan Result) <-chan Result {
    output := make(chan Result)

    for _, input := range inputs {
        go func(in <-chan Result) {
            for r := range in {
                output <- r
            }
        }(input)
    }

    close(output)  // Closes immediately! Workers panic
    return output
}
```

Fix:

```
func fanIn(inputs ...<-chan Result) <-chan Result {
    output := make(chan Result)
    var wg sync.WaitGroup

    for _, input := range inputs {
        wg.Add(1)
        go func(in <-chan Result) {
            defer wg.Done()
            for r := range in {
                output <- r
            }
        }(input)
    }

    go func() {
        wg.Wait()
        close(output)
    }()
```

```
    return output
}
```

Mistake 3: Unbounded Goroutine Creation

```
// WRONG: Creates goroutine per task
for task := range input {
    go func(t Task) {
        result := process(t)
        results <- result
    }(task)
}

// If input has 1M tasks → 1M goroutines → OOM
```

Fix: Use bounded fan-out (semaphore)

```
semaphore := make(chan struct{}, 100)

for task := range input {
    semaphore <- struct{}{}
    go func(t Task) {
        defer func() { <-semaphore }()
        result := process(t)
        results <- result
    }(task)
}
```

Performance Considerations

When to Fan-Out

Good candidates:

- CPU-intensive tasks (image processing, encoding)
- I/O-bound tasks (HTTP requests, database queries)
- Independent operations (no dependencies between tasks)

Poor candidates:

- Memory-intensive tasks (limited by RAM, not CPU)
- Tasks with dependencies (pipeline better)
- Very fast tasks (< 1ms, goroutine overhead dominates)

Optimal Worker Count

```
// CPU-bound
workers := runtime.NumCPU()

// I/O-bound
```

```
workers := runtime.NumCPU() * 10

// Benchmark to find optimal
func BenchmarkFanOut(b *testing.B) {
    for workers := 1; workers <= 64; workers *= 2 {
        b.Run(fmt.Sprintf("Workers-%d", workers), func(b *testing.B) {
            // ... benchmark logic
        })
    }
}
```

Buffer Sizing

Input channel:

```
// No buffer: Producer blocks until worker ready (backpressure)
input := make(chan Task)

// Small buffer: Smooth out timing differences
input := make(chan Task, numWorkers)

// Large buffer: Decouple producer/consumer
input := make(chan Task, 1000)
```

Output channels:

```
// Rule of thumb: bufferSize = 1-2x numWorkers
results := make(chan Result, numWorkers*2)
```

Interview Questions

Q1: "What's the difference between fan-out and worker pool?"

Answer:
"Fan-out creates multiple goroutines reading from a single shared channel—goroutines are created per distribution point. Worker pool pre-creates a fixed number of goroutines that continuously process tasks from a queue. Fan-out is a specific pattern; worker pool is a resource management pattern. Worker pool is generally preferred for production as it bounds concurrency."

Q2: "How do you ensure fair distribution in fan-out?"

Answer:
"Basic fan-out (shared channel) provides FIFO fairness—first available worker takes next task. For explicit load balancing, use round-robin distribution where tasks are explicitly sent to workers in rotation. For priority-based distribution, use separate high/low priority channels and workers check high priority first."

Q3: "What's the complexity of fan-in with N channels?"

Answer:
"Time: O(M) where M = total messages across all channels (process each message once). Space: O(N) for goroutines (one per input channel). Channel operations are O(1), so linear scaling with total message count."

Q4: "How do you handle errors in fan-out?"

Answer:
"Include error in result type: `type Result struct { Value T; Err error }`. Workers send both success and failure. Consumer checks `result.Err` and handles accordingly. For early termination on error, use context cancellation—cancel context on first error, workers check `ctx.Done()`."

Key Takeaways

1. **Fan-out = distribute work to multiple workers**
2. **Fan-in = merge results from multiple sources**
3. **Workers compete for tasks from shared channel** (FIFO)
4. **Always close input channel** when done sending
5. **Use WaitGroup + goroutine to close output** safely
6. **Bound concurrency** with semaphore or worker pool
7. **Fan-in with select** allows priority handling
8. **Buffer size: 1-2x numWorkers** for smooth operation
9. **Error handling: include Err in Result type**
10. **Context for cancellation** across all workers

Exercises

1. Implement fan-out/fan-in for parallel HTTP requests (URL fetcher).
2. Create priority fan-in: 3 channels (critical, normal, low), prefer critical.
3. Build bounded fan-out with max 50 concurrent goroutines.
4. Add timeout to fan-out: cancel all workers if not done in 5 seconds.
5. Benchmark fan-out with 1, 2, 4, 8, 16 workers on CPU-bound tasks.

Next: pipeline.md - Chaining processing stages with channels.