Graceful Shutdown

Why Graceful Shutdown Matters

Without graceful shutdown:

- In-flight requests fail
- Data loss (incomplete writes)
- Connections dropped mid-transaction
- Corrupted state

With graceful shutdown:

- Complete in-flight work
- Close connections cleanly
- Save state
- Report completion

Signal Handling

```
import (
    "context"
    "os"
    "os/signal"
    "syscall"
)

func main() {
    // Create context that cancels on interrupt signal
    ctx, stop := signal.NotifyContext(
        context.Background(),
        os.Interrupt,    // CTRL+C
        syscall.SIGTERM, // Kubernetes sends SIGTERM
    )
    defer stop()

    // Start application
    go runApp(ctx)

    // Wait for signal
    <-ctx.Done()

    log.Println("Shutting down gracefully...")

    // Cleanup
    shutdown(ctx)
}
```

HTTP Server Graceful Shutdown

```go
func main() {
    server := &http.Server{
        Addr:    ":8080",
        Handler: http.HandlerFunc(handler),
    }

    // Channel for shutdown signal
    stop := make(chan os.Signal, 1)
    signal.Notify(stop, os.Interrupt, syscall.SIGTERM)

    // Start server in goroutine
    go func() {
        log.Println("Server starting on :8080")
        if err := server.ListenAndServe(); err != nil && err != http.ErrServerClosed
{
            log.Fatal("ListenAndServe error:", err)
        }
    }()

    // Wait for interrupt signal
    <-stop

    log.Println("Shutdown signal received")

    // Graceful shutdown with 30-second timeout
    ctx, cancel := context.WithTimeout(context.Background(), 30*time.Second)
    defer cancel()

    if err := server.Shutdown(ctx); err != nil {
        log.Fatal("Server forced to shutdown:", err)
    }

    log.Println("Server stopped gracefully")
}
```

What `server.Shutdown()` does:

1. Stops accepting new connections
2. Waits for in-flight requests to complete
3. Times out after context deadline
4. Returns error if timeout exceeded

Worker Pool Graceful Shutdown

```go
type WorkerPool struct {
    tasks   chan Task
    workers int
    wg      sync.WaitGroup
    quit    chan struct{}
}
```

```go
func NewWorkerPool(workers int) *WorkerPool {
    pool := &WorkerPool{
        tasks:   make(chan Task, workers*2),
        workers: workers,
        quit:    make(chan struct{}),
    }

    // Start workers
    for i := 0; i < workers; i++ {
        pool.wg.Add(1)
        go pool.worker()
    }

    return pool
}

func (p *WorkerPool) worker() {
    defer p.wg.Done()

    for {
        select {
        case task := <-p.tasks:
            task.Execute()

        case <-p.quit:
            // Drain remaining tasks
            for task := range p.tasks {
                task.Execute()
            }
            return
        }
    }
}

func (p *WorkerPool) Submit(task Task) bool {
    select {
    case p.tasks <- task:
        return true
    case <-p.quit:
        return false  // Pool shutting down
    }
}

func (p *WorkerPool) Shutdown() {
    close(p.quit)   // Signal workers to stop
    close(p.tasks)  // No more tasks
    p.wg.Wait()     // Wait for all workers
    log.Println("Worker pool shutdown complete")
}

func (p *WorkerPool) ShutdownWithTimeout(timeout time.Duration) error {
```

```
    done := make(chan struct{})

    go func() {
        p.Shutdown()
        close(done)
    }()

    select {
    case <-done:
        return nil
    case <-time.After(timeout):
        return fmt.Errorf("shutdown timeout after %v", timeout)
    }
}
```

Complete Application Shutdown

```
type Application struct {
    server     *http.Server
    db         *sql.DB
    workerPool *WorkerPool
    cache      *Cache
}

func (app *Application) Run() error {
    // Setup signal handling
    ctx, stop := signal.NotifyContext(
        context.Background(),
        os.Interrupt,
        syscall.SIGTERM,
    )
    defer stop()

    // Start components
    go app.startServer()
    go app.startBackgroundJobs(ctx)

    // Wait for shutdown signal
    <-ctx.Done()
    log.Println("Shutdown initiated")

    // Shutdown with timeout
    shutdownCtx, cancel := context.WithTimeout(context.Background(), 30*time.Second)
    defer cancel()

    return app.Shutdown(shutdownCtx)
}

func (app *Application) Shutdown(ctx context.Context) error {
    log.Println("Shutting down components...")
```

```
    // Shutdown order matters!

    // 1. Stop accepting new work
    if err := app.server.Shutdown(ctx); err != nil {
        log.Printf("HTTP server shutdown error: %v", err)
    }

    // 2. Stop background jobs
    app.workerPool.ShutdownWithTimeout(20 * time.Second)

    // 3. Flush caches
    if err := app.cache.Flush(); err != nil {
        log.Printf("Cache flush error: %v", err)
    }

    // 4. Close database connections
    if err := app.db.Close(); err != nil {
        log.Printf("Database close error: %v", err)
    }

    log.Println("Shutdown complete")
    return nil
}
```

Graceful Shutdown with Multiple Services

```
type Service interface {
    Start(context.Context) error
    Stop(context.Context) error
}

type ServiceManager struct {
    services []Service
}

func (sm *ServiceManager) Start(ctx context.Context) error {
    errCh := make(chan error, len(sm.services))

    // Start all services
    for _, svc := range sm.services {
        go func(s Service) {
            if err := s.Start(ctx); err != nil {
                errCh <- err
            }
        }(svc)
    }

    // Wait for context cancellation or error
    select {
```

```
    case <-ctx.Done():
        return sm.Stop(ctx)
    case err := <-errCh:
        sm.Stop(ctx)
        return err
    }
}

func (sm *ServiceManager) Stop(ctx context.Context) error {
    log.Println("Stopping all services...")

    var wg sync.WaitGroup
    errCh := make(chan error, len(sm.services))

    // Stop all services concurrently
    for i := len(sm.services) - 1; i >= 0; i-- {
        wg.Add(1)
        go func(s Service) {
            defer wg.Done()
            if err := s.Stop(ctx); err != nil {
                errCh <- err
            }
        }(sm.services[i])
    }

    // Wait with timeout
    done := make(chan struct{})
    go func() {
        wg.Wait()
        close(done)
    }()

    select {
    case <-done:
        close(errCh)
        for err := range errCh {
            if err != nil {
                return err
            }
        }
        return nil

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

Database Connection Graceful Close

```go
func shutdownDatabase(db *sql.DB, timeout time.Duration) error {
    done := make(chan error, 1)

    go func() {
        // Set short timeouts to prevent hanging
        db.SetMaxIdleConns(0)
        db.SetConnMaxLifetime(0)

        // Close all connections
        done <- db.Close()
    }()

    select {
    case err := <-done:
        return err
    case <-time.After(timeout):
        return fmt.Errorf("database shutdown timeout")
    }
}
```

Graceful Consumer (Message Queue)

```go
type Consumer struct {
    messages chan Message
    quit     chan struct{}
    wg       sync.WaitGroup
}

func (c *Consumer) Start() {
    c.wg.Add(1)
    go c.consume()
}

func (c *Consumer) consume() {
    defer c.wg.Done()

    for {
        select {
        case msg := <-c.messages:
            c.process(msg)
            msg.Ack()  // Acknowledge after processing

        case <-c.quit:
            log.Println("Consumer shutting down...")

            // Process remaining messages
            for {
                select {
                case msg := <-c.messages:
```

```
                    c.process(msg)
                    msg.Ack()
                default:
                    return
                }
            }
        }
    }
}

func (c *Consumer) Shutdown(timeout time.Duration) error {
    log.Println("Stopping message consumption...")
    close(c.quit)

    done := make(chan struct{})
    go func() {
        c.wg.Wait()
        close(done)
    }()

    select {
    case <-done:
        log.Println("Consumer shutdown complete")
        return nil
    case <-time.After(timeout):
        return fmt.Errorf("consumer shutdown timeout")
    }
}
```

Graceful Shutdown in Kubernetes

Kubernetes sends SIGTERM, waits 30 seconds (default), then sends SIGKILL.

```
func main() {
    server := &http.Server{Addr: ":8080"}

    // Start server
    go server.ListenAndServe()

    // Wait for SIGTERM (from Kubernetes)
    stop := make(chan os.Signal, 1)
    signal.Notify(stop, syscall.SIGTERM)

    <-stop
    log.Println("SIGTERM received, shutting down...")

    // Kubernetes gives 30s, use 25s to be safe
    ctx, cancel := context.WithTimeout(context.Background(), 25*time.Second)
    defer cancel()

    if err := server.Shutdown(ctx); err != nil {
```

```
        log.Printf("Forced shutdown: %v", err)
        os.Exit(1)
    }

    log.Println("Graceful shutdown complete")
}
```

Kubernetes best practices:

1. Handle SIGTERM signal
2. Stop accepting new work immediately
3. Complete in-flight work within 30 seconds
4. Use shorter timeout (25s) for safety margin
5. Update readiness probe to fail during shutdown

Health Checks During Shutdown

```
type Server struct {
    httpServer  *http.Server
    isShuttingDown atomic.Bool
}

func (s *Server) healthCheckHandler(w http.ResponseWriter, r *http.Request) {
    if s.isShuttingDown.Load() {
        w.WriteHeader(http.StatusServiceUnavailable)
        w.Write([]byte("Shutting down"))
        return
    }

    w.WriteHeader(http.StatusOK)
    w.Write([]byte("OK"))
}

func (s *Server) Shutdown(ctx context.Context) error {
    // Mark as shutting down (fail health checks)
    s.isShuttingDown.Store(true)

    // Wait a moment for load balancer to notice
    time.Sleep(2 * time.Second)

    // Now shutdown
    return s.httpServer.Shutdown(ctx)
}
```

Testing Graceful Shutdown

```
func TestGracefulShutdown(t *testing.T) {
    server := &http.Server{Addr: ":8080"}
```

```
    // Start server
    go server.ListenAndServe()
    time.Sleep(100 * time.Millisecond)

    // Make long-running request
    done := make(chan bool)
    go func() {
        resp, err := http.Get("http://localhost:8080/slow")
        if err != nil {
            t.Error("Request failed:", err)
        } else {
            resp.Body.Close()
        }
        done <- true
    }()

    // Wait for request to start
    time.Sleep(100 * time.Millisecond)

    // Shutdown (should wait for request)
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()

    if err := server.Shutdown(ctx); err != nil {
        t.Fatal("Shutdown failed:", err)
    }

    // Verify request completed
    select {
    case <-done:
        // Success
    case <-time.After(6 * time.Second):
        t.Fatal("Request didn't complete")
    }
}
```

Common Mistakes

Mistake 1: Immediate os.Exit()

```
// WRONG: Doesn't wait for cleanup
signal.Notify(stop, os.Interrupt)
<-stop
os.Exit(0)  // Immediate exit, no cleanup!

// Fix: Graceful shutdown
<-stop
server.Shutdown(ctx)
db.Close()
os.Exit(0)
```

Mistake 2: No Shutdown Timeout

```
// WRONG: Can hang forever
server.Shutdown(context.Background())

// Fix: Use timeout
ctx, cancel := context.WithTimeout(context.Background(), 30*time.Second)
defer cancel()
server.Shutdown(ctx)
```

Mistake 3: Wrong Shutdown Order

```
// WRONG: Close DB before HTTP server
db.Close()
server.Shutdown(ctx)  // Handlers fail, can't access DB!

// Fix: Stop accepting work first
server.Shutdown(ctx)  // Stop new requests
db.Close()            // Then close DB
```

Interview Questions

Q: "How does HTTP server graceful shutdown work in Go?"

"server.Shutdown() stops accepting new connections, waits for existing requests to complete, respects context timeout. Pattern: 1) Stop accepting new connections immediately, 2) Wait for in-flight handlers to return, 3) Timeout if context deadline exceeded. Returns http.ErrServerClosed when closed voluntarily."

Q: "What's the correct order for shutting down components?"

"Reverse dependency order. General pattern: 1) Stop accepting new work (HTTP listener, message consumers), 2) Complete in-flight work (ongoing requests, jobs), 3) Flush caches/buffers, 4) Close stateful connections (database, message queues). Example: HTTP server → worker pool → cache → database."

Q: "How do you test graceful shutdown?"

"Start server, make long-running request, initiate shutdown mid-request, verify request completes successfully, verify shutdown within timeout. Pattern: use goroutine for request, channel to signal completion, select with timeout to verify. Test edge cases: no requests, many requests, timeout exceeded."

Q: "What signals should you handle?"

"SIGINT (os.Interrupt, Ctrl+C) for local development. SIGTERM (syscall.SIGTERM, standard kill) for production. Kubernetes sends SIGTERM. SIGKILL can't be caught (immediate termination). Use signal.NotifyContext or signal.Notify with both SIGINT and SIGTERM."

Key Takeaways

1. **Always handle SIGTERM and SIGINT**
2. **Use server.Shutdown() for HTTP**
3. **Set shutdown timeout (30s typical)**

4. **Shut down in reverse dependency order**
5. **Drain work queues before closing**
6. **Update health checks to fail during shutdown**
7. **Test with long-running requests**
8. **Kubernetes: use 25s timeout (30s grace period)**
9. **Log shutdown progress**
10. **Don't use os.Exit() without cleanup**

Exercises

1. Implement graceful shutdown for HTTP server + worker pool + database.
2. Add health check endpoint that fails during shutdown.
3. Test shutdown with 100 concurrent long-running requests.
4. Build service manager that shuts down multiple services in correct order.
5. Simulate Kubernetes: Send SIGTERM, verify completion within 30 seconds.

Next: bounded-parallelism.md - Controlling parallelism for batch operations.