Context and Cancellation Pattern

What is context.Context?

Context: A value that carries deadlines, cancellation signals, and request-scoped data across API boundaries and goroutines.

Purpose:

- **Cancellation:** Stop work in multiple goroutines
- **Timeouts:** Bound operation duration
- **Deadlines:** Absolute time limits
- **Request-scoped values:** Pass metadata (trace IDs, auth tokens)

Created by: Package `context` (standard library)

Basic Usage

```
import "context"

// Create root context
ctx := context.Background()

// Create cancellable context
ctx, cancel := context.WithCancel(ctx)
defer cancel()  // Always call cancel

// Create context with timeout
ctx, cancel := context.WithTimeout(ctx, 5*time.Second)
defer cancel()

// Create context with deadline
deadline := time.Now().Add(10 * time.Second)
ctx, cancel := context.WithDeadline(ctx, deadline)
defer cancel()

// Check if cancelled
select {
case <-ctx.Done():
    return ctx.Err()  // context.Canceled or context.DeadlineExceeded
default:
    // Continue work
}
```

Context Methods

```
type Context interface {
    // Done returns channel that closes when context cancelled
    Done() <-chan struct{}
```

```
    // Err returns why context was cancelled
    Err() error  // nil, Canceled, or DeadlineExceeded

    // Deadline returns when context will be cancelled
    Deadline() (deadline time.Time, ok bool)

    // Value returns request-scoped value
    Value(key interface{}) interface{}
}
```

Cancellation Pattern

Basic Cancellation

```
func worker(ctx context.Context) error {
    for {
        select {
        case <-ctx.Done():
            return ctx.Err()  // Cancelled
        default:
            // Do work
            if err := doWork(); err != nil {
                return err
            }
        }
    }
}

// Usage:
ctx, cancel := context.WithCancel(context.Background())

go worker(ctx)

// Cancel after 5 seconds
time.Sleep(5 * time.Second)
cancel()
```

Cancelling Multiple Goroutines

```
func processData(ctx context.Context, data []Item) error {
    ctx, cancel := context.WithCancel(ctx)
    defer cancel()

    errCh := make(chan error, len(data))

    for _, item := range data {
        go func(i Item) {
            errCh <- processItem(ctx, i)
```

```
        }(item)
    }

    // Wait for first error or all success
    for range data {
        if err := <-errCh; err != nil {
            cancel()  // Cancel all other goroutines
            return err
        }
    }

    return nil
}

func processItem(ctx context.Context, item Item) error {
    for j := 0; j < 100; j++ {
        // Check cancellation periodically
        select {
        case <-ctx.Done():
            return ctx.Err()
        default:
        }

        // Do work
        compute(item, j)
    }
    return nil
}
```

Timeout Pattern

```
func callWithTimeout(fn func() error) error {
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()

    done := make(chan error, 1)

    go func() {
        done <- fn()
    }()

    select {
    case err := <-done:
        return err
    case <-ctx.Done():
        return ctx.Err()  // context.DeadlineExceeded
    }
}
```

Real-World Example: HTTP Request with Timeout

```
func fetchURL(ctx context.Context, url string) ([]byte, error) {
    req, err := http.NewRequestWithContext(ctx, "GET", url, nil)
    if err != nil {
        return nil, err
    }

    resp, err := http.DefaultClient.Do(req)
    if err != nil {
        return nil, err
    }
    defer resp.Body.Close()

    return io.ReadAll(resp.Body)
}

// Usage:
ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
defer cancel()

data, err := fetchURL(ctx, "https://example.com")
if err != nil {
    if errors.Is(err, context.DeadlineExceeded) {
        log.Println("Request timed out")
    }
    return err
}
```

Real-World Example: Database Query with Context

```
func getUser(ctx context.Context, db *sql.DB, userID int) (*User, error) {
    query := "SELECT id, name, email FROM users WHERE id = $1"

    var user User
    err := db.QueryRowContext(ctx, query, userID).Scan(
        &user.ID,
        &user.Name,
        &user.Email,
    )

    if err != nil {
        return nil, err
    }

    return &user, nil
}

// Usage with timeout:
```

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

user, err := getUser(ctx, db, 123)
if err != nil {
    if errors.Is(err, context.DeadlineExceeded) {
        log.Println("Database query timed out")
    }
    return err
}
```

Propagating Context Through Call Stack

```
// Top-level handler
func handler(w http.ResponseWriter, r *http.Request) {
    ctx := r.Context()  // Get request context

    // Process with context
    result, err := processRequest(ctx, r)
    if err != nil {
        if errors.Is(err, context.Canceled) {
            // Client disconnected
            return
        }
        http.Error(w, err.Error(), 500)
        return
    }

    json.NewEncoder(w).Encode(result)
}

// Middle layer
func processRequest(ctx context.Context, r *http.Request) (*Result, error) {
    // Parse request
    data := parseRequest(r)

    // Call service layer
    return service.Process(ctx, data)
}

// Service layer
func (s *Service) Process(ctx context.Context, data *Data) (*Result, error) {
    // Check context before expensive operation
    if err := ctx.Err(); err != nil {
        return nil, err
    }

    // Call database
    dbResult, err := s.db.Query(ctx, data)
    if err != nil {
```

```
        return nil, err
    }

    // Call external API
    apiResult, err := s.api.Fetch(ctx, dbResult)
    if err != nil {
        return nil, err
    }

    return &Result{DB: dbResult, API: apiResult}, nil
}
```

Context Values (Request-Scoped Data)

```
type contextKey string

const (
    requestIDKey contextKey = "requestID"
    userIDKey    contextKey = "userID"
)

// Set value
func middleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        requestID := generateRequestID()
        ctx := context.WithValue(r.Context(), requestIDKey, requestID)

        next.ServeHTTP(w, r.WithContext(ctx))
    })
}

// Get value
func handler(w http.ResponseWriter, r *http.Request) {
    requestID, ok := r.Context().Value(requestIDKey).(string)
    if !ok {
        requestID = "unknown"
    }

    log.Printf("[%s] Processing request", requestID)
}
```

Important: Use context values for request-scoped data only (trace IDs, auth), NOT for passing optional parameters.

Context Best Practices

✅ Do:

```go
// 1. Pass context as first parameter
func doWork(ctx context.Context, data Data) error

// 2. Always defer cancel()
ctx, cancel := context.WithCancel(parent)
defer cancel()

// 3. Check cancellation in loops
for {
    select {
    case <-ctx.Done():
        return ctx.Err()
    default:
        // Work
    }
}

// 4. Use context.Background() at top level
func main() {
    ctx := context.Background()
    // ...
}

// 5. Use context.TODO() when unsure
func oldCode() {
    ctx := context.TODO()  // Placeholder
    // ...
}
```

❌ **Don't:**

```go
// 1. DON'T store context in struct
type Server struct {
    ctx context.Context  // WRONG
}

// 2. DON'T pass nil context
doWork(nil, data)  // WRONG, use context.Background()

// 3. DON'T use context values for function parameters
type contextKey string
ctx = context.WithValue(ctx, "data", data)  // WRONG

// 4. DON'T ignore cancellation
func worker(ctx context.Context) {
    for {
        // WRONG: Never checks ctx.Done()
        doWork()
    }
}
```

```
// 5. DON'T forget to call cancel()
ctx, cancel := context.WithTimeout(ctx, time.Second)
// WRONG: Missing defer cancel()
```

Graceful Shutdown Pattern

```
func main() {
    ctx, stop := signal.NotifyContext(context.Background(),
        os.Interrupt, syscall.SIGTERM)
    defer stop()

    server := &http.Server{Addr: ":8080"}

    // Start server
    go func() {
        if err := server.ListenAndServe(); err != nil &&
           err != http.ErrServerClosed {
            log.Fatal(err)
        }
    }()

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

    // Graceful shutdown
    shutdownCtx, cancel := context.WithTimeout(context.Background(), 30*time.Second)
    defer cancel()

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

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

Worker with Context

```
type Worker struct {
    tasks chan Task
}

func (w *Worker) Start(ctx context.Context) {
    for {
        select {
        case <-ctx.Done():
            log.Println("Worker shutting down")
            return
```

```
        case task := <-w.tasks:
            if err := w.process(ctx, task); err != nil {
                if errors.Is(err, context.Canceled) {
                    return
                }
                log.Printf("Error: %v", err)
            }
        }
    }
}

func (w *Worker) process(ctx context.Context, task Task) error {
    // Check context before expensive work
    if err := ctx.Err(); err != nil {
        return err
    }

    // Process task
    return task.Execute(ctx)
}
```

Context with Explicit Error Handling

```
func robustWorker(ctx context.Context) error {
    defer func() {
        if r := recover(); r != nil {
            log.Printf("Panic: %v", r)
        }
    }()

    for {
        select {
        case <-ctx.Done():
            // Distinguish cancellation reasons
            switch ctx.Err() {
            case context.Canceled:
                log.Println("Cancelled by caller")
            case context.DeadlineExceeded:
                log.Println("Timed out")
            }
            return ctx.Err()

        default:
            if err := doWork(ctx); err != nil {
                // Check if error is due to cancellation
                if errors.Is(err, context.Canceled) {
                    return err
                }
                // Other errors, continue or return based on logic
```

```
                log.Printf("Error: %v", err)
            }
        }
    }
}
```

Testing with Context

```
func TestWorkerCancellation(t *testing.T) {
    ctx, cancel := context.WithCancel(context.Background())

    done := make(chan bool)
    go func() {
        worker(ctx)
        done <- true
    }()

    // Cancel after 100ms
    time.Sleep(100 * time.Millisecond)
    cancel()

    // Verify worker stopped
    select {
    case <-done:
        // Success
    case <-time.After(time.Second):
        t.Fatal("Worker didn't stop after cancellation")
    }
}

func TestWorkerTimeout(t *testing.T) {
    ctx, cancel := context.WithTimeout(context.Background(), 100*time.Millisecond)
    defer cancel()

    err := worker(ctx)
    if !errors.Is(err, context.DeadlineExceeded) {
        t.Fatalf("Expected DeadlineExceeded, got %v", err)
    }
}
```

Common Mistakes

Mistake 1: Not Checking Context

```
// WRONG: Never checks cancellation
func worker(ctx context.Context) {
    for i := 0; i < 1000000; i++ {
        compute(i)  // Long-running, ignores context
    }
```

```
}

// Right:
func worker(ctx context.Context) {
    for i := 0; i < 1000000; i++ {
        select {
        case <-ctx.Done():
            return
        default:
        }
        compute(i)
    }
}
```

Mistake 2: Context in Struct

```
// WRONG: Storing context
type Server struct {
    ctx context.Context
}

// Right: Pass context to methods
type Server struct {
    // No context
}

func (s *Server) Handle(ctx context.Context) error {
    // Use context parameter
}
```

Mistake 3: Ignoring Cancel Function

```
// WRONG: Goroutine leak
ctx, cancel := context.WithCancel(parent)
go worker(ctx)
// Never calls cancel() → goroutine may leak

// Right:
ctx, cancel := context.WithCancel(parent)
defer cancel()  // Ensures cleanup
go worker(ctx)
```

Interview Questions

Q: "What's the difference between WithCancel, WithTimeout, and WithDeadline?"

"WithCancel returns context + cancel function—caller controls when to cancel. WithTimeout cancels after duration—use for operations with time limit. WithDeadline cancels at absolute time—use when specific wall-

clock time matters. All three return cancel function that must be called to release resources. Internally: WithTimeout calls WithDeadline(now + duration)."

Q: "Why must you always call cancel() even if context times out?"

"Context internals maintain timer and goroutine for timeout. If cancel() not called, timer isn't freed until timeout expires, leaking resources. Even if operation completes early or times out, must call cancel() to stop timer immediately. Use `defer cancel()` right after creating context."

Q: "When should you use context.Value?"

"Only for request-scoped data that crosses API boundaries: trace IDs, auth tokens, deadlines. NOT for optional parameters, feature flags, or data that could be explicit params. Context values are untyped and easy to misuse. Prefer explicit parameters. Rule: if removing context.Value breaks functionality, it should be a parameter."

Q: "How do you handle partial results when context cancelled?"

"Depends on requirements. Option 1: Return partial results + error (e.g., fetched 8/10). Option 2: Rollback and return error (transactions). Option 3: Cache partial work for retry. Pattern: wrap results in struct with completion indicator: `type Result struct { Data []; Partial bool; Err error }`"

Key Takeaways

1. **Context carries cancellation, timeouts, deadlines**
2. **Pass context as first parameter**
3. **Always defer cancel() after creating context**
4. **Check ctx.Done() in loops and before expensive work**
5. **Use context.Background() at top level**
6. **Use context.TODO() as placeholder**
7. **Don't store context in structs**
8. **Context values for request-scoped data only**
9. **WithCancel for manual control, WithTimeout for time limit**
10. **Cancellation is cooperative** (goroutines must check)

Exercises

1. Implement worker that stops within 1 second of cancellation. Test it.
2. Build HTTP server with per-request timeout (5 seconds). Test with slow handler.
3. Create fan-out function that cancels all workers on first error.
4. Implement graceful shutdown: wait up to 30s for workers to finish.
5. Write tests for: cancellation, timeout, deadline, value propagation.

Next: backpressure.md - Handling overload and applying backpressure.