Concurrent HTTP Servers

HTTP Server Concurrency Model

Go's `http.Server` handles each request in a separate goroutine automatically.

```go
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Goroutine ID: %d\n", getGoroutineID())
}

http.HandleFunc("/", handler)
http.ListenAndServe(":8080", nil)

// Every request runs in its own goroutine automatically!
```

Key characteristics:

- **One goroutine per connection** (keep-alive reuses goroutine)
- **No limit by default** (can spawn unlimited goroutines)
- **Concurrent request handling** (N requests = N goroutines)

Basic Concurrent Server

```go
package main

import (
    "encoding/json"
    "fmt"
    "log"
    "net/http"
    "sync"
    "time"
)

type Server struct {
    mu      sync.RWMutex
    counter int
}

func (s *Server) handleRequest(w http.ResponseWriter, r *http.Request) {
    s.mu.Lock()
    s.counter++
    count := s.counter
    s.mu.Unlock()

    // Simulate work
    time.Sleep(100 * time.Millisecond)

    response := map[string]interface{}{
        "request_num": count,
        "method":      r.Method,
        "path":        r.URL.Path,
    }

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

func main() {
    server := &Server{}

    http.HandleFunc("/api", server.handleRequest)

    log.Println("Server starting on :8080")
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Test concurrency:

```bash
# Send 10 concurrent requests
for i in {1..10}; do
    curl localhost:8080/api &
done
wait

# All responses arrive ~simultaneously (100ms total, not 1 second)
```

Rate-Limited HTTP Server

Prevent overload by limiting concurrent requests.

```go
type RateLimitedServer struct {
    sem        chan struct{}  // Semaphore for max concurrent requests
    maxWorkers int
}

func NewRateLimitedServer(maxWorkers int) *RateLimitedServer {
    return &RateLimitedServer{
        sem:        make(chan struct{}, maxWorkers),
        maxWorkers: maxWorkers,
    }
}

func (s *RateLimitedServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    select {
    case s.sem <- struct{}{}:
        defer func() { <-s.sem }()

        // Process request
        s.handleRequest(w, r)
    default:
        // Rate limit exceeded
        http.Error(w, "Server overloaded", http.StatusServiceUnavailable)
        w.Header().Set("Retry-After", "1")
    }
}

func (s *RateLimitedServer) handleRequest(w http.ResponseWriter, r *http.Request) {
    // Actual request handling
    time.Sleep(100 * time.Millisecond)
    w.WriteHeader(http.StatusOK)
    w.Write([]byte("Success"))
}

// Usage:
server := NewRateLimitedServer(100)  // Max 100 concurrent requests
http.ListenAndServe(":8080", server)
```

Middleware Pattern

Chain middleware for authentication, logging, rate limiting, etc.

```go
type Middleware func(http.Handler) http.Handler

func loggingMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        start := time.Now()
        next.ServeHTTP(w, r)
        log.Printf("%s %s %v", r.Method, r.URL.Path, time.Since(start))
    })
}

func authMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        token := r.Header.Get("Authorization")
        if token == "" {
            http.Error(w, "Unauthorized", http.StatusUnauthorized)
            return
        }

        next.ServeHTTP(w, r)
    })
}

func rateLimitMiddleware(limiter *rate.Limiter) Middleware {
    return func(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            if !limiter.Allow() {
                http.Error(w, "Rate limit exceeded", http.StatusTooManyRequests)
                return
            }
            next.ServeHTTP(w, r)
        })
    }
}

// Chain middleware:
handler := http.HandlerFunc(apiHandler)
handler = loggingMiddleware(handler)
handler = authMiddleware(handler)
handler = rateLimitMiddleware(rate.NewLimiter(100, 200))(handler)

http.ListenAndServe(":8080", handler)
```

Connection Context (Request Cancellation)

Detect when client disconnects.

```go
func longRunningHandler(w http.ResponseWriter, r *http.Request) {
    ctx := r.Context()

    // Simulate long computation
    result := make(chan string, 1)

    go func() {
        // Expensive computation
        time.Sleep(10 * time.Second)
        result <- "Done"
    }()

    select {
    case <-ctx.Done():
        // Client disconnected
        log.Println("Client disconnected before completion")
        return

    case res := <-result:
        // Send response
        w.Write([]byte(res))
    }
}
```

Streaming Response

Send data incrementally as it's ready.

```go
func streamHandler(w http.ResponseWriter, r *http.Request) {
    // Enable streaming
    w.Header().Set("Content-Type", "text/event-stream")
    w.Header().Set("Cache-Control", "no-cache")
    w.Header().Set("Connection", "keep-alive")
    flusher, ok := w.(http.Flusher)
    if !ok {
        http.Error(w, "Streaming unsupported", http.StatusInternalServerError)
        return
    }

    ctx := r.Context()

    for i := 0; i < 10; i++ {
        select {
        case <-ctx.Done():
            return
        default:
        }

        // Send chunk
        fmt.Fprintf(w, "data: Event %d\n\n", i)
        flusher.Flush()

        time.Sleep(time.Second)
    }
}
```

Server with Worker Pool

Process expensive operations in fixed worker pool.

```go
type WorkerPoolServer struct {
    tasks   chan *Task
    workers int
}

type Task struct {
    Request  *http.Request
    Response chan *Result
}

type Result struct {
    Data []byte
    Err  error
}

func NewWorkerPoolServer(workers int) *WorkerPoolServer {
    s := &WorkerPoolServer{
        tasks:   make(chan *Task, workers*2),
        workers: workers,
    }

    // Start workers
    for i := 0; i < workers; i++ {
        go s.worker()
    }

    return s
}

func (s *WorkerPoolServer) worker() {
    for task := range s.tasks {
        // Process task
        result := process(task.Request)
        task.Response <- result
    }
}

func (s *WorkerPoolServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    task := &Task{
        Request:  r,
        Response: make(chan *Result, 1),
    }

    select {
    case s.tasks <- task:
        // Queued
    default:
        http.Error(w, "Server overloaded", http.StatusServiceUnavailable)
        return
    }

    // Wait for result
    select {
    case result := <-task.Response:
        if result.Err != nil {
            http.Error(w, result.Err.Error(), http.StatusInternalServerError)
            return
        }
        w.Write(result.Data)

    case <-r.Context().Done():
        // Client disconnected
        return
    }
}
```

Connection Pooling (http.Client)

Reuse connections for outbound HTTP requests.

```
// Default client reuses connections automatically
client := &http.Client{
    Timeout: 10 * time.Second,
    Transport: &http.Transport{
```

```go
        MaxIdleConns:        100,              // Total idle connections
        MaxIdleConnsPerHost: 10,               // Idle per host
        MaxConnsPerHost:     100,              // Max per host
        IdleConnTimeout:     90 * time.Second,

        // Connection timeouts
        DialContext: (&net.Dialer{
            Timeout:   5 * time.Second,
            KeepAlive: 30 * time.Second,
        }).DialContext,

        // TLS config
        TLSHandshakeTimeout: 5 * time.Second,
    },
}

// Reuse client across requests (don't create per-request!)
func fetchURL(url string) ([]byte, error) {
    resp, err := client.Get(url)
    if err != nil {
        return nil, err
    }
    defer resp.Body.Close()

    return io.ReadAll(resp.Body)
}
```

Fan-Out HTTP Requests

Make multiple API calls concurrently.

```go
func aggregateAPIs(ctx context.Context, urls []string) ([]Response, error) {
    type result struct {
        url  string
        data []byte
        err  error
    }

    results := make(chan result, len(urls))

    // Fan-out
    for _, url := range urls {
        go func(u string) {
            req, _ := http.NewRequestWithContext(ctx, "GET", u, nil)
            resp, err := http.DefaultClient.Do(req)

            if err != nil {
                results <- result{url: u, err: err}
                return
            }
            defer resp.Body.Close()
            data, err := io.ReadAll(resp.Body)
            results <- result{url: u, data: data, err: err}
        }(url)
    }

    // Fan-in
    var responses []Response
    for range urls {
        r := <-results
        if r.err != nil {
            return nil, r.err  // Fail fast
        }
        responses = append(responses, Response{URL: r.url, Data: r.data})
    }

    return responses, nil
}
```

Timeout Per Request

```go
func handlerWithTimeout(w http.ResponseWriter, r *http.Request) {
    ctx, cancel := context.WithTimeout(r.Context(), 5*time.Second)
    defer cancel()

    done := make(chan struct{})
    var result string

    go func() {
        result = expensiveOperation(ctx)
        close(done)
    }()

    select {
    case <-done:
        w.Write([]byte(result))

    case <-ctx.Done():
        http.Error(w, "Request timeout", http.StatusRequestTimeout)
    }
}
```

Graceful Shutdown

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

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

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

    // Wait for interrupt
    <-stop
    log.Println("Shutting down gracefully...")

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

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

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

Common Mistakes

Mistake 1: Shared State Without Mutex

```
// WRONG: Counter race
var counter int

func handler(w http.ResponseWriter, r *http.Request) {
    counter++  // RACE! Many goroutines
    fmt.Fprintf(w, "Request %d", counter)
}

// Fix:
var (
    counter int
    mu      sync.Mutex
)

func handler(w http.ResponseWriter, r *http.Request) {
    mu.Lock()
```

```go
    counter++
    count := counter
    mu.Unlock()

    fmt.Fprintf(w, "Request %d", count)
}
```

Mistake 2: Creating http.Client Per Request

```go
// WRONG: No connection reuse, resource leak
func fetchURL(url string) {
    client := &http.Client{}  // New client!
    resp, _ := client.Get(url)
    // ...
}

// Fix: Reuse client
var sharedClient = &http.Client{
    Timeout: 10 * time.Second,
}

func fetchURL(url string) {
    resp, _ := sharedClient.Get(url)
    // ...
}
```

Mistake 3: Not Using Request Context

```go
// WRONG: Ignores client disconnect
func handler(w http.ResponseWriter, r *http.Request) {
    time.Sleep(10 * time.Second)  // Computes even if client gone
    w.Write([]byte("Done"))
}

// Fix: Check context
func handler(w http.ResponseWriter, r *http.Request) {
    ctx := r.Context()

    select {
    case <-time.After(10 * time.Second):
        w.Write([]byte("Done"))
    case <-ctx.Done():
        return  // Client disconnected
    }
}
```

Performance Tips

1. **Buffer writes:** Use `bufio.Writer` for large responses

2. **Pool connections:** Set http.Transport `MaxIdleConns`
3. **Limit concurrency:** Use semaphore to prevent overload
4. **Use keep-alive:** Enable HTTP/1.1 persistent connections
5. **Read body fully:** Always `io.ReadAll()` or drain, even on error

Interview Questions

Q: "How does Go's HTTP server handle concurrency?"

"Each connection runs in separate goroutine. Server spawns new goroutine for each request, calls handler, goroutine exits when handler returns. Keep-alive connections reuse goroutine. No built-in limit—can spawn millions of goroutines if traffic spikes. Should add semaphore or rate limiter for production."

Q: "How do you prevent HTTP server from being overwhelmed?"

"Add concurrency limit with semaphore. Pattern: semaphore in middleware, acquire before handler, release after. Return 503 Service Unavailable if limit exceeded. Also use rate limiting per IP/user, circuit breaker for downstream failures, connection limits in Transport."

Q: "What happens when client disconnects during request?"

"Request's Context.Done() channel closes. Handler should check context periodically, especially in long operations. Pattern: `select { case <-ctx.Done(): return; default: }` in loops. Prevents wasted work on cancelled requests."

Key Takeaways

1. **Go HTTP server spawns goroutine per request automatically**
2. **Protect shared state with mutex**
3. **Reuse http.Client for connection pooling**
4. **Check request context for cancellation**
5. **Add semaphore to limit concurrent requests**
6. **Use middleware for cross-cutting concerns**
7. **Graceful shutdown with server.Shutdown()**
8. **Fan-out for parallel external API calls**
9. **Stream responses for long operations**
10. **Set reasonable timeouts everywhere**

Exercises

1. Build HTTP server with per-IP rate limiting (100 req/min).

2. Implement health check endpoint that reports goroutine count.

3. Create aggregator that fetches 5 APIs concurrently with 2-second timeout.

4. Add graceful shutdown that waits for in-flight requests up to 30 seconds.

5. Benchmark: Measure throughput with 1, 10, 100, 1000 concurrent workers.

Next: db-concurrency.md - Database connection pooling and concurrent queries.