Stress Testing Concurrent Code

Why Stress Testing?

Regular tests: Test happy paths, basic edge cases
Stress tests: Expose races, deadlocks, resource exhaustion under load

What stress tests reveal:

- Race conditions (timing-dependent bugs)
- Deadlocks (only appear under contention)
- Resource leaks (accumulate over time)
- Performance degradation (under sustained load)
- Edge cases (rare but catastrophic)

Basic Stress Test Pattern

```
func TestStressCounter(t *testing.T) {
    var counter atomic.Int64
    const (
        goroutines = 1000
        iterations = 10000
    )

    var wg sync.WaitGroup
    wg.Add(goroutines)

    for i := 0; i < goroutines; i++ {
        go func() {
            defer wg.Done()
            for j := 0; j < iterations; j++ {
                counter.Add(1)
            }
        }()
    }

    wg.Wait()

    expected := int64(goroutines * iterations)
    if counter.Load() != expected {
        t.Errorf("Expected %d, got %d", expected, counter.Load())
    }
}
```

Stress Testing HTTP Server

```
func TestHTTPServerStress(t *testing.T) {
    server := httptest.NewServer(http.HandlerFunc(handler))
    defer server.Close()
```

```
    const (
        concurrentClients = 100
        requestsPerClient = 100
    )

    var wg sync.WaitGroup
    errors := make(chan error, concurrentClients*requestsPerClient)

    wg.Add(concurrentClients)
    for i := 0; i < concurrentClients; i++ {
        go func(clientID int) {
            defer wg.Done()

            client := &http.Client{
                Timeout: 5 * time.Second,
            }

            for j := 0; j < requestsPerClient; j++ {
                resp, err := client.Get(server.URL + "/api")
                if err != nil {
                    errors <- fmt.Errorf("client %d request %d: %w",
                        clientID, j, err)
                    continue
                }

                if resp.StatusCode != 200 {
                    errors <- fmt.Errorf("client %d got status %d",
                        clientID, resp.StatusCode)
                }

                resp.Body.Close()
            }
        }(i)
    }

    wg.Wait()
    close(errors)

    // Check for errors
    errorCount := 0
    for err := range errors {
        t.Error(err)
        errorCount++
        if errorCount > 10 {
            t.Fatal("Too many errors, stopping")
        }
    }
}
```

Stress Testing Worker Pool

```
func TestWorkerPoolStress(t *testing.T) {
    pool := NewWorkerPool(10)
    defer pool.Shutdown()

    const tasks = 10000
    results := make(chan int, tasks)
    var submitted atomic.Int32

    // Submit tasks rapidly
    for i := 0; i < tasks; i++ {
        i := i
        if pool.Submit(func() {
            // Simulate work
            time.Sleep(time.Millisecond)
            results <- i
            submitted.Add(1)
        }) {
            // Submitted successfully
        } else {
            t.Error("Failed to submit task")
        }
    }

    // Wait for completion
    pool.Wait()
    close(results)

    // Verify all tasks completed
    completed := 0
    for range results {
        completed++
    }

    if completed != tasks {
        t.Errorf("Expected %d tasks, completed %d", tasks, completed)
    }
}
```

Stress Testing Cache

```
func TestCacheStress(t *testing.T) {
    cache := NewCache()

    const (
        goroutines = 50
        operations = 1000
    )
```

```
    var wg sync.WaitGroup
    wg.Add(goroutines)

    for i := 0; i < goroutines; i++ {
        go func(id int) {
            defer wg.Done()

            for j := 0; j < operations; j++ {
                key := fmt.Sprintf("key-%d-%d", id, j)
                value := fmt.Sprintf("value-%d", j)

                // Interleave operations
                switch j % 3 {
                case 0:
                    cache.Set(key, value)
                case 1:
                    cache.Get(key)
                case 2:
                    cache.Delete(key)
                }
            }
        }(i)
    }

    wg.Wait()

    // Verify cache integrity
    if err := cache.Verify(); err != nil {
        t.Error(err)
    }
}
```

Continuous Stress Test

Run indefinitely until failure or timeout.

```
func TestContinuousStress(t *testing.T) {
    if testing.Short() {
        t.Skip("Skipping continuous stress test in short mode")
    }

    timeout := time.Minute
    if deadline, ok := t.Deadline(); ok {
        timeout = time.Until(deadline) - 5*time.Second
    }

    ctx, cancel := context.WithTimeout(context.Background(), timeout)
    defer cancel()

    var (
```

```
        operations atomic.Int64
        errors     atomic.Int64
    )

    const workers = 20
    var wg sync.WaitGroup
    wg.Add(workers)

    for i := 0; i < workers; i++ {
        go func() {
            defer wg.Done()

            for {
                select {
                case <-ctx.Done():
                    return
                default:
                }

                if err := performOperation(); err != nil {
                    errors.Add(1)
                } else {
                    operations.Add(1)
                }

                // Small delay to prevent CPU spinning
                time.Sleep(time.Millisecond)
            }
        }()
    }

    wg.Wait()

    t.Logf("Operations: %d, Errors: %d",
        operations.Load(), errors.Load())

    if errors.Load() > 0 {
        t.Errorf("Encountered %d errors", errors.Load())
    }
}
```

Deadlock Detection Test

```
func TestNoDeadlock(t *testing.T) {
    done := make(chan bool, 1)

    go func() {
        // Code that might deadlock
        mu1 := &sync.Mutex{}
        mu2 := &sync.Mutex{}
```

```
        var wg sync.WaitGroup
        wg.Add(2)

        go func() {
            defer wg.Done()
            mu1.Lock()
            time.Sleep(10 * time.Millisecond)
            mu2.Lock()
            mu2.Unlock()
            mu1.Unlock()
        }()

        go func() {
            defer wg.Done()
            // Different lock order - risk of deadlock
            mu2.Lock()
            time.Sleep(10 * time.Millisecond)
            mu1.Lock()
            mu1.Unlock()
            mu2.Unlock()
        }()

        wg.Wait()
        done <- true
    }()

    select {
    case <-done:
        // Success
    case <-time.After(5 * time.Second):
        t.Fatal("Deadlock detected")
    }
}
```

Memory Leak Detection Test

```
func TestNoMemoryLeak(t *testing.T) {
    var m runtime.MemStats
    runtime.ReadMemStats(&m)
    before := m.Alloc

    const iterations = 1000
    for i := 0; i < iterations; i++ {
        runOperation()
    }

    runtime.GC()
    runtime.ReadMemStats(&m)
    after := m.Alloc
```

```
    growth := after - before
    perIteration := growth / iterations

    // Allow reasonable growth (e.g., 1KB per iteration)
    if perIteration > 1024 {
        t.Errorf("Memory leak detected: %d bytes/iteration", perIteration)
    }
}
```

Goroutine Leak Detection Test

```
func TestNoGoroutineLeak(t *testing.T) {
    before := runtime.NumGoroutine()

    const iterations = 100
    for i := 0; i < iterations; i++ {
        ctx, cancel := context.WithTimeout(context.Background(), time.Second)
        runWithContext(ctx)
        cancel()
    }

    // Give goroutines time to exit
    time.Sleep(100 * time.Millisecond)
    runtime.GC()

    after := runtime.NumGoroutine()

    if after > before+2 {  // Allow small variance
        t.Errorf("Goroutine leak: before=%d, after=%d", before, after)
    }
}
```

Race Condition Stress Test

```
func TestRaceStress(t *testing.T) {
    type Counters struct {
        mu sync.Mutex
        m  map[string]int
    }

    counters := &Counters{
        m: make(map[string]int),
    }

    const (
        goroutines = 100
        operations = 1000
```

```
    )

    var wg sync.WaitGroup
    wg.Add(goroutines)

    for i := 0; i < goroutines; i++ {
        go func(id int) {
            defer wg.Done()

            for j := 0; j < operations; j++ {
                key := fmt.Sprint("counter", id%10)

                counters.mu.Lock()
                counters.m[key]++
                counters.mu.Unlock()
            }
        }(i)
    }

    wg.Wait()

    // Verify total
    total := 0
    for _, count := range counters.m {
        total += count
    }

    expected := goroutines * operations
    if total != expected {
        t.Errorf("Expected %d, got %d", expected, total)
    }
}
```

Load Pattern: Ramp Up

```
func TestRampUpLoad(t *testing.T) {
    server := httptest.NewServer(http.HandlerFunc(handler))
    defer server.Close()

    maxClients := 100
    rampUpDuration := 10 * time.Second

    start := time.Now()
    var current atomic.Int32

    for i := 0; i < maxClients; i++ {
        // Stagger client starts
        sleepDuration := rampUpDuration * time.Duration(i) /
time.Duration(maxClients)
        time.Sleep(sleepDuration / time.Duration(maxClients))
```

```
        go func() {
            current.Add(1)
            defer current.Add(-1)

            // Run for remaining duration
            deadline := start.Add(rampUpDuration)
            for time.Now().Before(deadline) {
                resp, err := http.Get(server.URL)
                if err != nil {
                    t.Error(err)
                    return
                }
                resp.Body.Close()

                time.Sleep(100 * time.Millisecond)
            }
        }()
    }

    // Monitor peak concurrency
    peak := int32(0)
    ticker := time.NewTicker(100 * time.Millisecond)
    defer ticker.Stop()

    timeout := time.After(rampUpDuration + 2*time.Second)
    for {
        select {
        case <-ticker.C:
            c := current.Load()
            if c > peak {
                peak = c
            }
            if c == 0 {
                goto done
            }
        case <-timeout:
            goto done
        }
    }

done:
    t.Logf("Peak concurrency: %d", peak)
}
```

Soak Test (Long-Running)

```
func TestSoak(t *testing.T) {
    if testing.Short() {
        t.Skip("Skipping soak test in short mode")
```

```
}

duration := time.Hour
if d := os.Getenv("SOAK_DURATION"); d != "" {
    var err error
    duration, err = time.ParseDuration(d)
    if err != nil {
        t.Fatal(err)
    }
}

ctx, cancel := context.WithTimeout(context.Background(), duration)
defer cancel()

app := NewApplication()
go app.Run(ctx)

var (
    requests atomic.Int64
    errors   atomic.Int64
)

// Continuous load
const workers = 10
var wg sync.WaitGroup
wg.Add(workers)

for i := 0; i < workers; i++ {
    go func() {
        defer wg.Done()

        for {
            select {
            case <-ctx.Done():
                return
            default:
            }

            requests.Add(1)
            if err := app.Request(); err != nil {
                errors.Add(1)
            }

            time.Sleep(100 * time.Millisecond)
        }
    }()
}

// Monitor metrics
ticker := time.NewTicker(time.Minute)
defer ticker.Stop()
```

```
    for {
        select {
        case <-ticker.C:
            r := requests.Load()
            e := errors.Load()
            t.Logf("Requests: %d, Errors: %d (%.2f%%)",
                r, e, float64(e)/float64(r)*100)

        case <-ctx.Done():
            wg.Wait()

            r := requests.Load()
            e := errors.Load()
            t.Logf("Final: Requests: %d, Errors: %d", r, e)

            if e > r/100 {  // > 1% error rate
                t.Errorf("Error rate too high: %.2f%%",
                    float64(e)/float64(r)*100)
            }

            return
        }
    }
}
```

Fuzzing Concurrent Code (Go 1.18+)

```
func FuzzConcurrentMap(f *testing.F) {
    f.Add("key1", "value1")
    f.Add("key2", "value2")

    f.Fuzz(func(t *testing.T, key, value string) {
        m := &sync.Map{}
        const goroutines = 10

        var wg sync.WaitGroup
        wg.Add(goroutines * 2)

        // Writers
        for i := 0; i < goroutines; i++ {
            go func() {
                defer wg.Done()
                m.Store(key, value)
            }()
        }

        // Readers
        for i := 0; i < goroutines; i++ {
            go func() {
                defer wg.Done()
```

```
                m.Load(key)
            }()
        }

        wg.Wait()
    })
}
```

Chaos Testing

Inject failures randomly.

```
func TestChaos(t *testing.T) {
    const (
        duration         = 30 * time.Second
        workers          = 20
        failureRate      = 0.1  // 10% operations fail
        networkDelayRate = 0.2  // 20% have delay
    )

    ctx, cancel := context.WithTimeout(context.Background(), duration)
    defer cancel()

    var wg sync.WaitGroup
    wg.Add(workers)

    for i := 0; i < workers; i++ {
        go func() {
            defer wg.Done()

            for {
                select {
                case <-ctx.Done():
                    return
                default:
                }

                // Random failure
                if rand.Float64() < failureRate {
                    continue
                }

                // Random network delay
                if rand.Float64() < networkDelayRate {
                    time.Sleep(time.Duration(rand.Intn(100)) * time.Millisecond)
                }

                performOperation()
            }
        }()
    }
```

```
    wg.Wait()
}
```

Stress Test Metrics Collection

```
type StressMetrics struct {
    mu              sync.Mutex
    requests        int64
    errors          int64
    latencies       []time.Duration
    startTime       time.Time
    peakConcurrency int32
}

func (sm *StressMetrics) RecordRequest(latency time.Duration, err error) {
    sm.mu.Lock()
    defer sm.mu.Unlock()

    sm.requests++
    if err != nil {
        sm.errors++
    }
    sm.latencies = append(sm.latencies, latency)
}

func (sm *StressMetrics) Report() string {
    sm.mu.Lock()
    defer sm.mu.Unlock()

    if len(sm.latencies) == 0 {
        return "No data"
    }

    // Calculate percentiles
    sort.Slice(sm.latencies, func(i, j int) bool {
        return sm.latencies[i] < sm.latencies[j]
    })

    p50 := sm.latencies[len(sm.latencies)/2]
    p95 := sm.latencies[len(sm.latencies)*95/100]
    p99 := sm.latencies[len(sm.latencies)*99/100]

    duration := time.Since(sm.startTime)
    rps := float64(sm.requests) / duration.Seconds()
    errorRate := float64(sm.errors) / float64(sm.requests) * 100

    return fmt.Sprintf(`
Stress Test Results:
  Duration:     %v
```

```
  Requests:     %d
  Errors:       %d (%.2f%%)
  RPS:          %.2f
  Latency P50:  %v
  Latency P95:  %v
  Latency P99:  %v
  Peak Concurrency: %d
`, duration, sm.requests, sm.errors, errorRate, rps, p50, p95, p99,
sm.peakConcurrency)
}
```

Interview Questions

Q: "How do stress tests differ from regular tests?"

"Regular tests verify correctness with small inputs and single-threaded execution. Stress tests verify correctness under high concurrency, sustained load, and resource pressure. They expose timing-dependent bugs (races), resource leaks (memory, goroutines), performance degradation, and rare edge cases that only appear under load. Pattern: High concurrency (100+ goroutines), many iterations (1000+), long duration (minutes to hours)."

Q: "How do you determine appropriate stress test parameters?"

"Based on production load + margin. Example: Production sees 1000 req/sec peak, test at 2000 req/sec. Goroutines: 2-10x production concurrency. Duration: Long enough for leaks to accumulate (1-60 minutes). Iterations: Enough to trigger race conditions (10,000+). Adjust based on failures—if tests pass easily, increase stress."

Q: "What's the difference between stress testing and fuzzing?"

"Stress testing: Fixed operations, high concurrency/load, checks for races/deadlocks/leaks. Fuzzing: Random inputs, discovers unexpected edge cases, finds input-triggered bugs. Both valuable: Stress tests find concurrency bugs, fuzzing finds input validation bugs. Can combine: Fuzz test with concurrent execution."

Key Takeaways

1. **Stress tests reveal timing-dependent bugs**
2. **Use high concurrency (100+ goroutines)**
3. **Run many iterations (1000+ operations)**
4. **Check for races, leaks, deadlocks**
5. **Measure latency, throughput, error rate**
6. **Continuous stress tests run until timeout**
7. **Soak tests run hours to find slow leaks**
8. **Always use -race flag**
9. **Ramp up load gradually**
10. **Chaos testing injects random failures**

Exercises

1. Write stress test: 1000 goroutines incrementing counter, verify correctness.
2. Build HTTP server stress test with metrics (P50, P95, P99 latency).
3. Create soak test that runs 1 hour, detects memory leaks.
4. Implement chaos test that randomly injects 10% failure rate.
5. Stress test worker pool: Submit 100,000 tasks, verify all complete.

Next: benchmarks.md - Benchmarking concurrent performance.