Go Execution Tracer

What is the Execution Tracer?

Execution tracer: Built-in tool that records program execution events over time.

What it captures:

- Goroutine creation/blocking/unblocking
- System calls
- GC events
- Network operations
- Processor utilization

Purpose:

- Understand goroutine behavior
- Find blocking operations
- Visualize concurrency
- Debug performance issues

Collecting a Trace

###Method 1: Command-Line

```
package main

import (
    "os"
    "runtime/trace"
)

func main() {
    f, err := os.Create("trace.out")
    if err != nil {
        panic(err)
    }
    defer f.Close()

    if err := trace.Start(f); err != nil {
        panic(err)
    }
    defer trace.Stop()

    // Your program
    runApplication()
}

// Run: go run main.go
// View: go tool trace trace.out
```

Method 2: Test with Trace

```
# Generate trace from test
go test -trace=trace.out

# View trace
go tool trace trace.out
```

Method 3: HTTP Handler (Production)

```
import (
    "net/http"
    _ "net/http/pprof"
    "runtime/trace"
)

func main() {
    http.HandleFunc("/debug/trace", func(w http.ResponseWriter, r *http.Request) {
        trace.Start(w)
        defer trace.Stop()

        // Trace for 5 seconds
        time.Sleep(5 * time.Second)
    })

    http.ListenAndServe(":6060", nil)
}

// Collect: curl http://localhost:6060/debug/trace > trace.out
// View: go tool trace trace.out
```

Trace Viewer UI

```
go tool trace trace.out

# Browser opens with:
# - View trace: Timeline of all goroutines
# - Goroutine analysis: Per-goroutine stats
# - Network blocking profile
# - Synchronization blocking profile
# - Syscall blocking profile
# - Scheduler latency profile
```

Example: Finding Goroutine Blocking

```go
package main

import (
    "os"
    "runtime/trace"
    "time"
)

func worker(id int, ch chan int) {
    for {
        val := <-ch  // Blocks here if no data
        time.Sleep(10 * time.Millisecond)
        process(val)
    }
}

func main() {
    f, _ := os.Create("trace.out")
    defer f.Close()

    trace.Start(f)
    defer trace.Stop()

    ch := make(chan int)

    // Start workers
    for i := 0; i < 10; i++ {
        go worker(i, ch)
    }

    // Send slowly
    for i := 0; i < 100; i++ {
        ch <- i
        time.Sleep(50 * time.Millisecond)  // Slow!
    }
}

// Trace shows:
// - Workers mostly blocked on channel receive
// - Low CPU utilization
// - High goroutine wait time
```

Trace Viewer: Key Views

View 1: Timeline

Shows all goroutines over time.

What to look for:

- **Green:** Goroutine running

- **Blue:** Goroutine runnable (waiting for CPU)
- **White:** Goroutine blocked
- **Red:** GC activity

Common patterns:

- **Too much blue:** CPU-bound, need more GOMAXPROCS
- **Too much white:** I/O-bound or blocked on sync primitives
- **Frequent red:** GC pressure, reduce allocations

View 2: Goroutine Analysis

Per-goroutine statistics.

Metrics:

- **Execution time:** Time spent running
- **Network wait time:** Blocked on network I/O
- **Sync block time:** Blocked on mutexes/channels
- **Blocking syscall time:** Blocked in syscalls
- **Scheduler wait time:** Waiting for CPU

Example output:

```
Goroutine 19:
  Execution:       12.3ms
  Network wait:     0.1ms
  Sync block:     150.2ms  ← Problem!
  Syscall:          0.0ms
  Sched wait:       1.2ms
```

View 3: Blocking Profiles

Shows where goroutines block.

Types:

- **Network blocking:** I/O operations
- **Sync blocking:** Mutexes, channels
- **Syscall blocking:** OS calls

Custom Trace Regions

Annotate code for easier analysis.

```
import "runtime/trace"

func processRequest(ctx context.Context, req Request) {
    // Create trace region
    ctx, task := trace.NewTask(ctx, "processRequest")
    defer task.End()

    // Phase 1: Database query
    trace.WithRegion(ctx, "database", func() {
        result := db.Query(ctx, req.Query)
```

```
    })

    // Phase 2: Processing
    trace.WithRegion(ctx, "process", func() {
        data := process(result)
    })

    // Phase 3: Response
    trace.WithRegion(ctx, "response", func() {
        sendResponse(data)
    })
}

// Trace shows labeled regions in timeline
```

Example: Diagnosing Worker Pool

```
func TestWorkerPoolTrace(t *testing.T) {
    // Start tracing
    f, _ := os.Create("worker_trace.out")
    defer f.Close()

    trace.Start(f)
    defer trace.Stop()

    // Create pool with 5 workers
    pool := NewWorkerPool(5)
    defer pool.Shutdown()

    // Submit 100 tasks
    for i := 0; i < 100; i++ {
        i := i
        pool.Submit(func() {
            ctx := context.Background()
            ctx, task := trace.NewTask(ctx, "task")
            defer task.End()

            trace.WithRegion(ctx, "compute", func() {
                time.Sleep(10 * time.Millisecond)
            })
        })
    }

    pool.Wait()
}

// Trace reveals:
// - How many workers active at any time
```

```
// - Task queueing behavior
// - Worker idle time
```

Finding Deadlocks with Trace

```
func TestDeadlock(t *testing.T) {
    f, _ := os.Create("deadlock_trace.out")
    defer f.Close()

    trace.Start(f)
    defer trace.Stop()

    mu1 := &sync.Mutex{}
    mu2 := &sync.Mutex{}

    done := make(chan bool)

    go func() {
        mu1.Lock()
        time.Sleep(10 * time.Millisecond)
        mu2.Lock()
        mu2.Unlock()
        mu1.Unlock()
        done <- true
    }()

    go func() {
        mu2.Lock()
        time.Sleep(10 * time.Millisecond)
        mu1.Lock()  // DEADLOCK!
        mu1.Unlock()
        mu2.Unlock()
        done <- true
    }()

    select {
    case <-done:
    case <-time.After(time.Second):
        t.Fatal("Deadlock detected")
    }
}

// Trace shows:
// - Both goroutines blocked permanently
// - Lock acquisition order
// - Exact deadlock point
```

Trace-Based Performance Optimization

Before: Inefficient Pipeline

```
func processPipeline(items []Item) {
    f, _ := os.Create("before_trace.out")
    trace.Start(f)
    defer trace.Stop()

    // Stage 1: Load (sequential!)
    loaded := make([]Item, 0)
    for _, item := range items {
        loaded = append(loaded, load(item))
    }

    // Stage 2: Process (sequential!)
    processed := make([]Item, 0)
    for _, item := range loaded {
        processed = append(processed, process(item))
    }

    // Stage 3: Save (sequential!)
    for _, item := range processed {
        save(item)
    }
}

// Trace shows: Linear execution, no concurrency
```

After: Concurrent Pipeline

```
func processPipelineOptimized(items []Item) {
    f, _ := os.Create("after_trace.out")
    trace.Start(f)
    defer trace.Stop()

    // Stage 1: Load
    loadCh := make(chan Item, 10)
    go func() {
        defer close(loadCh)
        for _, item := range items {
            loadCh <- load(item)
        }
    }()

    // Stage 2: Process (parallel)
    processCh := make(chan Item, 10)
    var wg sync.WaitGroup
    for i := 0; i < runtime.NumCPU(); i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
```

```
            for item := range loadCh {
                processCh <- process(item)
            }
        }()
    }

    go func() {
        wg.Wait()
        close(processCh)
    }()

    // Stage 3: Save
    for item := range processCh {
        save(item)
    }
}

// Trace shows: Concurrent execution, higher CPU utilization
```

Measuring GC Impact

```
func TestGCImpact(t *testing.T) {
    f, _ := os.Create("gc_trace.out")
    defer f.Close()

    trace.Start(f)
    defer trace.Stop()

    // Allocate aggressively
    items := make([][]byte, 0)
    for i := 0; i < 100000; i++ {
        items = append(items, make([]byte, 1024))
    }
}

// Trace shows:
// - GC pause frequency
// - STW (stop-the-world) duration
// - Goroutine pause due to GC
```

Trace Analysis Script

```
#!/bin/bash

# Collect trace
go test -trace=trace.out -run=TestFunction

# View in browser
```

```
go tool trace trace.out

# Extract stats
go tool trace -pprof=net trace.out > net.pprof
go tool trace -pprof=sync trace.out > sync.pprof
go tool trace -pprof=syscall trace.out > syscall.pprof

# Analyze blocking
go tool pprof sync.pprof
```

Common Patterns in Trace

Pattern 1: Goroutine Leak

Symptom: Goroutine count increases over time
Trace shows: Goroutines created but never exit
Solution: Add context cancellation

Pattern 2: Lock Contention

Symptom: Low CPU utilization, high sync block time
Trace shows: Many goroutines blocked on same mutex
Solution: Reduce critical section or shard locks

Pattern 3: Channel Bottleneck

Symptom: Goroutines blocked on channel send/receive
Trace shows: All workers waiting on channel
Solution: Increase buffer size or add workers

Pattern 4: CPU-Bound

Symptom: High CPU, goroutines runnable but waiting
Trace shows: Many blue goroutines (runnable)
Solution: Increase GOMAXPROCS or reduce goroutines

Comparing Before/After Traces

```
# Collect baseline
go test -trace=before.out -run=TestOriginal

# Make optimization

# Collect new trace
go test -trace=after.out -run=TestOptimized

# Compare visually in browser
go tool trace before.out  # Note timings
go tool trace after.out   # Compare
```

Trace Performance Overhead

Overhead: ~10-20% slowdown (less than -race)
File size: Can be large (MB per second)
Duration: Limit to 5-30 seconds for analysis

```
// Good: Short, focused trace
trace.Start(f)
runBenchmark()  // 5 seconds
trace.Stop()

// Bad: Long trace (huge file)
trace.Start(f)
runForHours()  // GB of trace data!
trace.Stop()
```

Interview Questions

Q: "What's the difference between pprof and trace?"

"pprof: Sampling profiler, shows hotspots (CPU, memory, blocking), statistical, low overhead. Trace: Records all events, shows timeline of execution, goroutine behavior over time, higher overhead. Use pprof to find 'what' is slow (CPU/memory), use trace to find 'why' slow (blocking, contention, concurrency issues)."

Q: "How do you diagnose goroutine blocking with trace?"

"Collect trace with runtime/trace, open with go tool trace. Look at Timeline view: white spaces = blocked goroutines. Click goroutine to see blocking reason (sync, network, syscall). Check Goroutine Analysis for sync block time. Use Synchronization Blocking Profile to find which mutexes/channels. Pattern: High sync block time → lock contention or channel bottleneck."

Q: "What does 'too much blue' in trace mean?"

"Blue = runnable but not running (waiting for CPU). Means more goroutines want CPU than cores available. Causes: 1) Too many CPU-bound goroutines, 2) GOMAXPROCS too low. Solutions: Reduce goroutine count (use worker pool), or increase GOMAXPROCS. I/O-bound workloads should NOT have much blue."

Q: "How do custom trace regions help?"

"trace.NewTask() and trace.WithRegion() annotate code phases. Shows labeled regions in timeline, easier to identify which phase slow. Example: Split request handler into 'database', 'process', 'response' regions. Trace shows time spent in each, identifies bottleneck. Essential for complex concurrent workflows."

Key Takeaways

1. **Trace shows goroutine timeline and blocking**
2. **Green = running, blue = runnable, white = blocked**
3. **Collect with runtime/trace, view with go tool trace**
4. **Use custom regions to annotate code**
5. **Goroutine analysis shows where time spent**
6. **Find lock contention (high sync block time)**
7. **Find goroutine leaks (count increasing)**

8. **Limit trace duration (5-30 seconds)**
9. **10-20% overhead, use for debugging**
10. **Compare before/after optimization**

Exercises

1. Create program with intentional goroutine blocking, visualize with trace.
2. Build worker pool, use trace to find optimal worker count.
3. Add custom trace regions to pipeline, measure each stage.
4. Diagnose lock contention: Find slow mutex in trace.
5. Compare traces before/after optimization, document improvements.

Next: debugging-deadlocks.md - Techniques for finding and fixing deadlocks.