Go Race Detector

What is the Race Detector?

Race detector: Built-in tool that detects data races at runtime.

How it works:

- Instruments memory accesses
- Tracks synchronization operations
- Detects unsynchronized conflicting accesses
- Reports races with stack traces

Not a static analyzer: Only finds races in executed code paths.

Enabling the Race Detector

```
# Run tests with race detector
go test -race

# Run application with race detector
go run -race main.go

# Build binary with race detection
go build -race

# Benchmark with race detection
go test -race -bench=.
```

Example Race Detection

```
package main

import (
    "fmt"
    "time"
)

var counter int

func increment() {
    counter++  // RACE!
}

func main() {
    go increment()
    go increment()

    time.Sleep(time.Second)
```

```go
    fmt.Println(counter)
}
```

Output with `-race`:

```
==================
WARNING: DATA RACE
Write at 0x00c000014098 by goroutine 7:
  main.increment()
      /app/main.go:10 +0x3c

Previous write at 0x00c000014098 by goroutine 6:
  main.increment()
      /app/main.go:10 +0x3c

Goroutine 7 (running) created at:
  main.main()
      /app/main.go:14 +0x64

Goroutine 6 (running) created at:
  main.main()
      /app/main.go:14 +0x64
==================
Found 1 data race(s)
```

Race Detection in Tests

```go
func TestIncrement(t *testing.T) {
    counter := 0

    // This will fail with -race
    go func() { counter++ }()
    go func() { counter++ }()

    time.Sleep(100 * time.Millisecond)

    if counter != 2 {
        t.Errorf("Expected 2, got %d", counter)
    }
}

// Run: go test -race
```

Common Races Detected

Race #1: Unsynchronized Variable Access

```go
// RACE: Multiple goroutines write to shared variable
type Server struct {
```

```go
    requestCount int  // RACE!
}

func (s *Server) handleRequest(w http.ResponseWriter, r *http.Request) {
    s.requestCount++  // Multiple goroutines
    // ...
}

// Fix: Use atomic or mutex
type Server struct {
    requestCount atomic.Int64
}

func (s *Server) handleRequest(w http.ResponseWriter, r *http.Request) {
    s.requestCount.Add(1)
    // ...
}
```

Race #2: Slice/Map Concurrent Access

```go
// RACE: Map not thread-safe
m := make(map[string]int)

go func() {
    m["key"] = 1  // RACE!
}()

go func() {
    _ = m["key"]  // RACE!
}()

// Fix: Use sync.Map or mutex
var (
    m  = make(map[string]int)
    mu sync.RWMutex
)

go func() {
    mu.Lock()
    m["key"] = 1
    mu.Unlock()
}()

go func() {
    mu.RLock()
    _ = m["key"]
    mu.RUnlock()
}()
```

Race #3: Closure Variable Capture

```go
// RACE: Loop variable captured by closure
for i := 0; i < 10; i++ {
    go func() {
        fmt.Println(i)  // RACE! Reads shared variable
    }()
}

// Fix: Pass as parameter
for i := 0; i < 10; i++ {
    go func(n int) {
        fmt.Println(n)
    }(i)
}
```

Race #4: Struct Field Update

```go
// RACE: Concurrent struct writes
type Config struct {
    Host string
    Port int
}

var config Config

go func() {
    config.Host = "localhost"  // RACE!
}()

go func() {
    config.Port = 8080  // RACE!
}()

// Fix: Use atomic.Value for entire struct
var config atomic.Value  // stores *Config

go func() {
    newConfig := Config{Host: "localhost", Port: 8080}
    config.Store(&newConfig)
}()

go func() {
    cfg := config.Load().(*Config)
    fmt.Println(cfg.Host, cfg.Port)
}()
```

Race Detector Limitations

Limitation #1: Only Detects Executed Races

```go
func TestRace(t *testing.T) {
    counter := 0

    if false {  // Never executed
        // Race detector won't find this
        go func() { counter++ }()
        go func() { counter++ }()
    }
}
```

Solution: Test with high concurrency, multiple runs, stress tests.

Limitation #2: Performance Impact

- **10x slower** execution
- **5-10x more memory**
- Not suitable for production

```bash
# Benchmark shows slowdown
go test -bench=. -race

# vs normal run
go test -bench=.
```

Limitation #3: False Negatives

Race detector can miss races that don't execute during test.

```go
// Rare race, might not trigger
if time.Now().Unix()%100 == 0 {
    // Race here might not execute during test
    sharedVar++
}
```

Solution: Use stress tests, run multiple times, long-running tests.

Using Race Detector Effectively

Strategy 1: Run All Tests with -race

```bash
# Add to CI/CD pipeline
go test -race ./...

# With coverage
go test -race -coverprofile=coverage.out ./...
```

Strategy 2: Stress Tests

```go
func TestConcurrentAccess(t *testing.T) {
    var counter atomic.Int32

    // High concurrency to increase race probability
    const goroutines = 100
    const iterations = 1000

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

Strategy 3: Integration Tests with -race

```go
func TestHTTPServerRace(t *testing.T) {
    server := NewServer()
    go server.Start()
    defer server.Stop()

    // Make many concurrent requests
    const clients = 50
    var wg sync.WaitGroup
    wg.Add(clients)

    for i := 0; i < clients; i++ {
        go func() {
            defer wg.Done()
            resp, _ := http.Get("http://localhost:8080/api")
            resp.Body.Close()
        }()
    }

    wg.Wait()
}
```

Real-World Example: Finding Production Race

```go
// Production code with subtle race
type Cache struct {
    data map[string]string
    mu   sync.RWMutex
}

func (c *Cache) Get(key string) (string, bool) {
    c.mu.RLock()
    defer c.mu.RUnlock()

    val, ok := c.data[key]
    return val, ok
}

func (c *Cache) Set(key, value string) {
    c.mu.Lock()
    defer c.mu.Unlock()

    c.data[key] = value
}

func (c *Cache) Clear() {
    // RACE! Recreating map without lock!
    c.data = make(map[string]string)
}

// Test finds race:
func TestCache(t *testing.T) {
    cache := &Cache{data: make(map[string]string)}

    go func() {
        for i := 0; i < 1000; i++ {
            cache.Set(fmt.Sprintf("key%d", i), "value")
        }
    }()

    go func() {
        for i := 0; i < 1000; i++ {
            cache.Get(fmt.Sprintf("key%d", i))
        }
    }()

    go func() {
        for i := 0; i < 10; i++ {
            cache.Clear()  // RACE detected here!
            time.Sleep(10 * time.Millisecond)
        }
    }()
```

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

// Fix: Lock in Clear
func (c *Cache) Clear() {
    c.mu.Lock()
    defer c.mu.Unlock()

    c.data = make(map[string]string)
}
```

Interpreting Race Detector Output

```
==================
WARNING: DATA RACE
Write at 0x00c000094000 by goroutine 7:
  main.(*Server).increment()
      /app/server.go:45 +0x3c

Previous read at 0x00c000094000 by goroutine 6:
  main.(*Server).getCount()
      /app/server.go:50 +0x2a

Goroutine 7 (running) created at:
  main.main()
      /app/main.go:20 +0x64

Goroutine 6 (running) created at:
  main.main()
      /app/main.go:19 +0x50
==================
```

Key information:

1. **Memory address:** `0x00c000094000` (same address = same variable)
2. **Operation type:** "Write" vs "Previous read"
3. **Location:** File and line number
4. **Goroutine:** Which goroutines involved
5. **Creation site:** Where goroutines were spawned

Race Detector Environment Variables

```bash
# Control race detector behavior
export GORACE="log_path=/tmp/race halt_on_error=1 strip_path_prefix=/app/"

# Options:
# log_path=file        - Write to file instead of stderr
# halt_on_error=0/1    - Exit on first race
# strip_path_prefix=   - Remove prefix from paths
```

```
# history_size=N       - Number of memory accesses to track (default 1)
# exitcode=N           - Exit code on race (default 66)

go test -race
```

Combining Race Detector with Other Tools

With Benchmarks

```go
func BenchmarkCounter(b *testing.B) {
    counter := atomic.Int32{}

    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            counter.Add(1)
        }
    })
}

// Run: go test -bench=. -race
```

With Coverage

```
# Race detection + coverage
go test -race -coverprofile=coverage.out ./...
go tool cover -html=coverage.out
```

With Fuzzing (Go 1.18+)

```go
func FuzzConcurrentMap(f *testing.F) {
    f.Add("key", "value")

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

        go func() {
            m.Store(key, value)
        }()

        go func() {
            m.Load(key)
        }()
    })
}

// Run: go test -fuzz=. -race
```

Common Mistakes with Race Detector

Mistake #1: Not Running in CI

```
# WRONG: Only testing locally
go test ./...

# Right: Always include -race in CI
go test -race ./...
```

Mistake #2: Ignoring Warnings

```
// "It's just one race, and it sometimes passes..."
// NO! Fix ALL races, they compound!
```

Mistake #3: Using in Production

```
# WRONG: Deploying -race build
go build -race -o server
./server  # 10x slower, not production-ready

# Right: Only for testing
go test -race
```

Performance Benchmarks

```
func BenchmarkWithoutRace(b *testing.B) {
    counter := 0
    for i := 0; i < b.N; i++ {
        counter++
    }
}

func BenchmarkWithRace(b *testing.B) {
    counter := 0
    for i := 0; i < b.N; i++ {
        counter++
    }
}

// Run:
// go test -bench=. -run=^$           →  100 ns/op
// go test -bench=. -run=^$ -race     → 1200 ns/op (12x slower)
```

Interview Questions

Q: "How does Go's race detector work?"

"Instruments all memory accesses and synchronization operations at compile time. Tracks happens-before relationships between goroutines. Detects when two goroutines access same memory location without synchronization, and at least one is a write. Reports all detected races with stack traces. Limitation: Only finds races in executed code paths."

Q: "Why can't you use -race in production?"

"Race detector adds significant overhead: 10x slower execution, 5-10x more memory usage due to tracking metadata for all memory accesses. Designed for testing only. Solution: Run comprehensive tests with -race in CI/CD, deploy normal builds to production."

Q: "What's a false negative in race detection?"

"Race that exists in code but detector doesn't find because code path never executed during test run. Example: race in error handling branch that doesn't trigger, or rare timing-dependent race. Solution: Write comprehensive tests, use stress tests with high concurrency, run tests multiple times."

Q: "How do you ensure all races are found?"

"1) Run all tests with -race in CI, 2) Write stress tests with high concurrency, 3) Integration tests with real load patterns, 4) Run tests multiple times, 5) Use fuzzing for input variations, 6) Long-running soak tests. Can't guarantee 100% detection, but these maximize coverage."

Key Takeaways

1. **Always run tests with -race in CI/CD**
2. **Race detector only finds executed races**
3. **10x slower, not for production**
4. **Fix ALL races, even "harmless" ones**
5. **Write stress tests with high concurrency**
6. **Use with benchmarks to verify thread safety**
7. **Interpret output: address, operation, goroutines**
8. **Environment variables customize behavior**
9. **Combine with coverage and fuzzing**
10. **False negatives exist—comprehensive testing essential**

Exercises

1. Create program with intentional race, detect with -race.

2. Write stress test that runs 1000 goroutines accessing shared map.

3. Fix all races in this code (hint: 3 races):

```go
var counter int
var users []string

func add(name string) {
    users = append(users, name)
    counter++
}
```

```go
func main() {
    for i := 0; i < 100; i++ {
        go add(fmt.Sprintf("user%d", i))
    }
    time.Sleep(time.Second)
    fmt.Println(counter, len(users))
}
```

4. Build HTTP server, test with -race under concurrent load.
5. Use GORACE environment variables to customize output.

Next: stress-testing.md - Testing concurrent systems under load.