Go benchmark measures cpu & memory metrics as a part of `StartTimer` & `StopTimer` methods.

These methods are invoked from `RunN`, which runs N iterations of a benchmark.

```go
func (b *B) runN(n int) {
    benchmarkLock.Lock()
    defer benchmarkLock.Unlock()
    ctx, cancelCtx := context.WithCancel(context.Background())
    defer func() {
        b.runCleanup(normalPanic)
        b.checkRaces()
    }()
    // Try to get a comparable environment for each run
    // by clearing garbage from previous runs.
    runtime.GC()
    b.resetRaces()
    b.N = n
    b.loop.n = 0
    b.loop.i = 0
    b.loop.done = false
    b.ctx = ctx
    b.cancelCtx = cancelCtx

    b.parallelism = 1
    b.ResetTimer()
    b.StartTimer()
    b.benchFunc(b)
    b.StopTimer()
    b.previousN = n
    b.previousDuration = b.duration
```

clear state of metrics, start timer

stop timer

Please note runN is called from launch() method.

```go
func (b *B) launch() {
    /.../
    defer func() {
        b.signal <- true
    }()
    /.../
    if b.loop.n == 0 {
        // Run the benchmark for at least the specified amount of time.
        if b.benchTime.n > 0 {
            /.../
            if b.benchTime.n > 1 {
                b.runN(b.benchTime.n)
            }
        } else {
            d := b.benchTime.d
            for n := int64(1); !b.failed && b.duration < d && n < 1e9; {
                last := n
                // Predict required iterations.
                goalns := d.Nanoseconds()
                prevIters := int64(b.N)
                n = int64(predictN(goalns, prevIters, b.duration.Nanoseconds(), last))
                b.runN(int(n))
            }
        }
    }
    b.result = BenchmarkResult{N: b.N, T: b.duration, Bytes: b.bytes, MemAllocs: b.netAllocs, MemBytes: b.netBytes, Extra: b.extra}
}
```

launch

store result

Let's cover the following:

① start timer

② stop timer

③ reset timer

④ BenchmarkResult

⑤ add result. (from Run(..))

```go
func (b *B) StartTimer() {
    if !b.timerOn {
        runtime.ReadMemStats(&memStats)
        b.startAllocs = memStats.Mallocs
        b.startBytes = memStats.TotalAlloc
        b.start = highPrecisionTimeNow()
        b.timerOn = true
        b.loop.i &^= loopPoisonTimer
    }
}
```

time.Now()

(more on this here:)

StartTimer

- collect memory statistics
- capture start time

↓

time.Now()

Memory statistics
↳ TotalAlloc: cumulative bytes allocated
↳ Mallocs: cumulative count of heap allocations

```go
func (b *B) StopTimer() {
    if b.timerOn {
        b.duration += highPrecisionTimeSince(b.start)
        runtime.ReadMemStats(&memStats)
        b.netAllocs += memStats.Mallocs - b.startAllocs
        b.netBytes += memStats.TotalAlloc - b.startBytes
        b.timerOn = false
        // If we hit B.Loop with the timer stopped, fail.
        b.loop.i |= loopPoisonTimer
    }
}
```

StopTimer

→ time.Now() - b.start

- Capture duration

- Capture allocation

- Compute net allocations & netBytes

```go
func (b *B) ResetTimer() {
    if b.extra == nil {
        // Allocate the extra map before reading memory stats.
        // Pre-size it to make more allocation unlikely.
        b.extra = make(map[string]float64, 16)
    } else {
        clear(b.extra)
    }
    if b.timerOn {
        runtime.ReadMemStats(&memStats)
        b.startAllocs = memStats.Mallocs
        b.startBytes = memStats.TotalAlloc
        b.start = highPrecisionTimeNow()
    }
    b.duration = 0
    b.netAllocs = 0
    b.netBytes = 0
}
```

ResetTimer

- Capture allocation & time, if timer is on.

- clear duration, net allocation & netBytes.

Everytime runN finishes,
properties like duration,
net allocation & netBytes are
set in the pointer to B.
(represents one benchmark)

After N iterations are done
for a benchmark, an instance
of BenchmarkResult is created.

```go
func (b *B) launch() {
        b.signal <- true
    }()
    /.../
    if b.loop.n == 0 {
        // Run the benchmark for at least the specified amount of time.
        if b.benchTime.n > 0 {
            // We already ran a single iteration in run1.
            // If -benchtime=1x was requested, use that result.
            // See https://golang.org/issue/32051.
            if b.benchTime.n > 1 {
                b.runN(b.benchTime.n)
            }
        } else {
            d := b.benchTime.d
            for n := int64(1); !b.failed && b.duration < d && n < 1e9; {
                last := n
                // Predict required iterations.
                goalns := d.Nanoseconds()
                prevIters := int64(b.N)
                n = int64(predictN(goalns, prevIters, b.duration.Nanoseconds(), last))
                b.runN(int(n))
            }
        }
    }
    b.result = BenchmarkResult{N: b.N, T: b.duration, Bytes: b.bytes, MemAllocs: b.netAllocs, MemBytes: b.netBytes, Extra: b.extra}
}
```

capture BenchmarkResult after all iterations are done for a benchmark.

Effectively, b.N, b.duration, b.bytes, b.netAllocs, b.netBytes are reported for the last iteration value.

```go
type BenchmarkResult struct {
    N         int           // The number of iterations.
    T         time.Duration // The total time taken.
    Bytes     int64         // Bytes processed in one iteration.
    MemAllocs uint64        // The total number of memory allocations.
    MemBytes  uint64        // The total number of bytes allocated.

    // Extra records additional metrics reported by ReportMetric.
    Extra map[string]float64
}
```

BenchmarkResult has N, T, MemAllocs and MemBytes.

```go
func (b *B) Run(name string, f func(b *B)) bool {
    sub := &B{
        common: common{
            signal:  make(chan bool),
            name:    benchName,
            parent:  &b.common,
            level:   b.level + 1,
            creator: pc[:n],
            w:       b.w,
            chatty:  b.chatty,
            bench:   true,
        },
        importPath: b.importPath,
        benchFunc:  f,
        benchTime:  b.benchTime,
        bstate:     b.bstate,
    }
    if partial {...}

    if b.chatty != nil {...}

    if sub.run1() {
        sub.run()
    }
    b.add(sub.result)
    return !sub.failed
}
```

After Run() method is done for a benchmark, the add() method is called to add the Benchmark-Result.

```go
func (b *B) add(other BenchmarkResult) {
    r := &b.result
    // The aggregated BenchmarkResults resemble running all subbenchmarks as
    // in sequence in a single benchmark.
    r.N = 1
    r.T += time.Duration(other.NsPerOp())
    if other.Bytes == 0 {
        // Summing Bytes is meaningless in aggregate if not all subbenchmarks
        // set it.
        b.missingBytes = true
        r.Bytes = 0
    }
    if !b.missingBytes {
        r.Bytes += other.Bytes
    }
    r.MemAllocs += uint64(other.AllocsPerOp())
    r.MemBytes += uint64(other.AllocedBytesPerOp())
}
```

The add(...) method simply computes NsPerOp(), AllocsPerOp(), AllocedBytesPerOp() ...

* PerOp represents per iteration.

```go
func (r BenchmarkResult) AllocsPerOp() int64 {
    if v, ok := r.Extra["allocs/op"]; ok { return int64(v) }
    if r.N <= 0 { return 0 }
    return int64(r.MemAllocs) / int64(r.N)
}
```

The above code AllocsPerOp() is simply $\frac{\text{memAllocs}}{N}$ > $\underline{\text{iterations}}$

Other PerOp() methods are similar.

Also the diff. b/w the two is the reported time cputime or usertime?

Open:
9s