Testing with Concurrency in Go

Oct. 2017

@brownylin

Outline

- When testing concurrent code, there's a temptation to use `sleep`
- Review some common concurrency patterns, and see how to stay away from `sleep`
- Samples: goo.gl/nvtoDZ

Thread-Safe Operation

```go
var (
	chDelete = make(chan string)
	chSet    = make(chan string)
	chQuit   = make(chan bool)

	set      = make(map[string]bool)
)
```

```go
func opSet1() {
	for {
		select {
		case v := <-chSet:
			fmt.Printf("set %s\n", v)
			set[v] = true
		case v := <-chDelete:
			fmt.Printf("delete %s\n", v)
			delete(set, v)
		case <-chQuit:
			return
		}
	}
}
```

Assert too early

```go
func Test_opSet1(t *testing.T) {
	go opSet1()
	chSet <- "foo"
	assert.True(t, set["foo"])
}
```

```
>$ go test -run opSet1
set foo
--- FAIL: Test_opSet1 (0.00s)
        Error Trace:    main_test.go:34
        Error:          Should be true
FAIL
exit status 1
FAIL 0.015s
```

Sleep(1 * time.Second)

```go
func Test_opSet1(t *testing.T) {
	go opSet1()
	chSet <- "foo"
	time.Sleep(1 * time.Second)
	assert.True(t, set["foo"])
}
```

```
>$ go test -run opSet1
set foo
PASS
ok 1.016s
```


NO
SLEEP.

Arrange a sentinel func `done()` to sync

```go
func opSet1() {
	for {
		select {
		case v := <-chSet:
			fmt.Printf("set %s\n", v)
			set[v] = true
		case v := <-chDelete:
			fmt.Printf("delete %s\n", v)
			delete(set, v)
		case <-chQuit:
			return
		}
	}
}
```

```go
var done = func() {}

func opSet2() {
	for {
		select {
		case v := <-chSet:
			fmt.Printf("set %s\n", v)
			set[v] = true
			done()
		case v := <-chDelete:
			fmt.Printf("delete %s\n", v)
			delete(set, v)
			done()
		case <-chQuit:
			return
		}
	}
}
```

Sentinel act as a syncer

```go
func Test_opSet2(t *testing.T) {
	chDone := make(chan struct{}, 1)
	done = func() {
		chDone <- struct{}{}
	}

	go opSet2()
	chSet <- "foo"

	<-chDone
	assert.True(t, set["foo"])
}
```

```
>$ go test -run opSet2
set foo
PASS
ok 0.014s
```

```go
var done = func() {}

func opSet2() {
	for {
		select {
		case v := <-chSet:
			fmt.Printf("set %s\n", v)
			set[v] = true
			done()
		case v := <-chDelete:
			fmt.Printf("delete %s\n", v)
			delete(set, v)
			done()
		case <-chQuit:
			return
		}
	}
}
```

Worker Pools

https://gobyexample.com/worker-pools

Dispatch goroutine

```go
func dispatch1(nw, nj int) {
	jobs := make(chan int, 100)

	for w := 1; w <= nw; w++ {
		go worker1(w, jobs)
	}

	for j := 1; j <= nj; j++ {
		jobs <- j
	}
	close(jobs)
}
```

Worker goroutine (with different efficiency)

```go
func worker1(id int, jobs <-chan int) {
	for j := range jobs {
		r := rand.Intn(100)
		time.Sleep(time.Duration(r) * time.Millisecond)
		fmt.Printf("finished: worker[%d], job[%d]\n", id, j)
	}
}
```

No worker run

```go
func Test_dispatch1(t *testing.T) {
	nw, nj := 3, 10
	dispatch1(nw, nj)
}
```

```
>$ go test -run dispatch1
PASS
ok 0.008s
```

Sleep(3 * time.Second)

```go
func Test_dispatch1(t *testing.T) {
	nw, nj := 3, 10
	dispatch1(nw, nj)
	time.Sleep(3 * time.Second)
}
```

```
>$ go test -run dispatch1
finished: worker[3], job[3]
finished: worker[1], job[1]
finished: worker[2], job[2]
finished: worker[2], job[6]
...
PASS
ok 3.012s
```


NO
SLEEP.

Again, sentinel func

```go
func dispatch2(nw, nj int) {
	jobs := make(chan int, 100)

	for w := 1; w <= nw; w++ {
		go worker2(w, jobs, done)
	}

	for j := 1; j <= nj; j++ {
		jobs <- j
	}
	close(jobs)
}
```

```go
var done = func() {}

func worker2(id int, jobs <-chan int, done func()) {
	for j := range jobs {
		r := rand.Intn(100)
		time.Sleep(time.Duration(r) * time.Millisecond)
		done()
		fmt.Printf("finished: worker[%d], job[%d]\n", id, j)
	}
}
```

Sentinel act as syncer and counter (with `sync.WaitGroup`)

```go
func Test_dispatch2(t *testing.T) {
	nw, nj := 3, 10

	var wg sync.WaitGroup
	wg.Add(nj)
	done = func() {
		wg.Done()
	}
	dispatch2(nw, nj)

	wg.Wait()
}
```

```
>$ go test -run dispatch2
finished: worker[2], job[3]
finished: worker[1], job[1]
finished: worker[3], job[2]
finished: worker[3], job[6]
finished: worker[2], job[4]
finished: worker[3], job[7]
finished: worker[2], job[8]
finished: worker[2], job[10]
finished: worker[1], job[5]
finished: worker[3], job[9]
PASS
ok  0.198s
```

Polling

```go
var (
	timeout   = 5 * time.Second
	interval  = 1 * time.Second
	numOfTick = 0
	pollFn    = func() error { return nil }
)
```

```go
func polling1() error {
	chTo := time.NewTimer(timeout).C
	chTk := time.NewTicker(interval).C
	for {
		select {
		case <-chTo:
			fmt.Println("timeout")
			return fmt.Errorf("timeout")
		case <-chTk:
			numOfTick++
			fmt.Printf("tick %d\n", numOfTick)

			err := pollFn()
			if err != nil {
				continue
			}
			return nil
		}
	}
}
```

Sleep(6 * time.Second)

```go
func Test_polling1Timeout(t *testing.T) {
	pollFn = func() error {
		return fmt.Errorf("err")
	}

	var err error
	go func() {
		err = polling1()
	}()

	time.Sleep(6 * time.Second)

	assert.Equal(t, 4, numOfTick)
	assert.Error(t, err)
}
```

```
>$ go test -run polling1Timeout
tick 1
tick 2
tick 3
tick 4
timeout
PASS
ok 6.019s
```

Sleep(3 * time.Second)

```go
func Test_polling1Success(t *testing.T) {
	pollFn = func() error {
		if numOfTick == 1 {
			return fmt.Errorf("err")
		}
		return nil
	}

	var err error
	go func() {
		err = polling1()
	}()

	time.Sleep(3 * time.Second)

	assert.Equal(t, 2, numOfTick)
	assert.NoError(t, err)
}
```

```
>$ go test -run polling1Success
tick 1
tick 2
PASS
ok 3.018s
```


NO
SLEEP.

Don’t depend on `real` time, on fake time

```go
import (
	"code.cloudfoundry.org/clock/fakeclock"
)

var fc = fakeclock.NewFakeClock(time.Now())
var tickDone = func() {}
```

```go
func polling2() error {
	chTo := fc.NewTimer(timeout).C()
	chTk := fc.NewTicker(interval).C()

	for {
		select {
		case <-chTo:
			fmt.Println("timeout")
			return fmt.Errorf("timeout")
		case <-chTk:
			numOfTick++
			fmt.Printf("tick %d\n", numOfTick)
			tickDone()

			err := pollFn()
			if err != nil {
				continue
			}
			return nil
		}
	}
}
```

Fast-forward `timeout` seconds

```go
func Test_polling2Timeout(t *testing.T) {
	pollFn = func() error {
		return fmt.Errorf("err")
	}

	chDone := make(chan struct{})
	var err error
	go func() {
		err = polling2()
		close(chDone)
	}()

	fc.WaitForNWatchersAndIncrement(timeout, 2)
	<-chDone

	assert.Error(t, err)
}
```

```
>$ go test -run polling2Timeout
timeout
PASS
ok 0.014s
```

Step by step forward `interval` seconds

```go
func Test_polling2Success(t *testing.T) {
	chTickDone := make(chan struct{}, 1)
	tickDone = func() {
		chTickDone <- struct{}{}
	}

	pollFn = func() error {
		if numOfTick == 1 {
			return fmt.Errorf("err")
		}
		return nil
	}

	chDone := make(chan struct{})
	var err error
	go func() {
		err = polling2()
		close(chDone)
	}()

	fc.WaitForNWatchersAndIncrement(interval, 2)
	<-chTickDone
	assert.Equal(t, 1, numOfTick)
	assert.Nil(t, err)

	fc.WaitForNWatchersAndIncrement(interval, 2)
	<-chTickDone
	assert.Equal(t, 2, numOfTick)

	<-chDone
	assert.NoError(t, err)
}
```

```
>$ go test -run polling2Success
tick 1
tick 2
PASS
ok 0.015s
```

Summary

- **Challenges**
  - Finishment uncertainty
  - Real `time` dependent
- **Tricks**
  - Sentinel func to sync (with *WaitGroup* to count)
  - Depend on fake `time`
- **Notes**
  - Testing with `Sleep` usually has data race issue
  - go test -race

References

- Testing Techniques - https://youtu.be/ndmB0bj7eyw
- Fake clock - https://github.com/cloudfoundry/clock
- Samples - https://github.com/browny/testing-with-concurrency