Golang Tutorial #3

- unit test
- benchmark
- go-grpc
- context

unit test tool

- assert
- dockertest
- gock

unit test flow control

```go
// init_test.go
func TestMain(m *testing.M) {
        log.SetOutput(os.Stdout)
        log.SetFlags(log.LstdFlags)
        var p *int
        retCode := 0
        p = &retCode
        BeforeTest()
        defer AfterTest(p)
        *p = m.Run()
}
```

Assert

```go
func TestGetMongoDBInfo(t *testing.T) {
	mongoConfig := getMongoDBInfo()
	assert.Equal(t, "testt", mongoConfig.Name)
}
```

```
--- FAIL: TestGetMongoDBInfo (0.00s)
    .../main_test.go:54:
            Error Trace:    main_test.go:54
            Error:          Not equal:
                            expected: "testt"
                            actual  : "test"
            Test:           TestGetMongoDBInfo
FAIL
```

HTTP mock

```go
defer gock.Off() // Flush pending mocks after test executi
gock.InterceptClient(httpClient)
defer gock.RestoreClient(httpClient)
apDomain := "http://test.com"
path := "/test"
gock.New(apDomain).
    Get(path).
    Reply(200).
    JSON(map[string]string{
        "id": "123",
    })
```

dockertest run mongo

```go
var (
        dockerPool     *dockertest.Pool
        dockerResource *dockertest.Resource
)

dockerPool, err = dockertest.NewPool("")
dockerResource, err = dockerPool.Run("mongo", "3.4", nil)
dockerResource.GetPort("27017/tcp")
```

dockertest teardown

```go
func AfterTest(ret *int) {
	if e := recover(); e != nil {
		dockerPool.Purge(dockerResource)
		os.Exit(1)
	}
	dockerPool.Purge(dockerResource)
	os.Exit(*ret)
}
```

- sometimes teardown fail, please use `docker system prune -a`

go benchmark #1

- `go test -benchmem -run=xxx` (test cpu time and memory alloc)
- used when compared two or more syntax/function

```go
func BenchmarkIfLt1(b *testing.B) {
        count := 0
        test := ""
        for n := 0; n < b.N; n++ {
                if len(test) < 1 {
                        count++
                }
        }
        fmt.Println("lt1:", count)
}
```

go benchmark result

```
BenchmarkIfLt1-4        lt1: 100
lt1: 10000
lt1: 1000000
lt1: 100000000
lt1: 2000000000
2000000000               0.64 ns/op           0 B/op
}
```

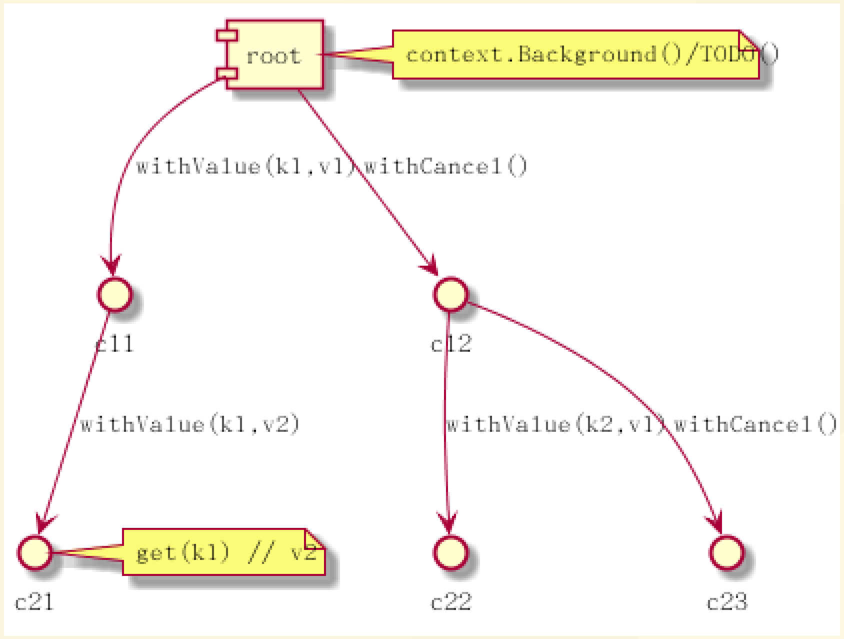
go gRPC implementaion

- grpc intro
- go-grpc
- `protoc --go_out=plugins=grpc:. *.proto`
- go grpc example
- server and client struct implement interface
- `RegisgerXXXServiceServer` `NewXXXServiceClient`
- example in the example3/

go context

- built-in library [context]
- bi-directional tree structure.
- tricky but flexible design.
- default empty ctx implement all methods
- always start with context.Background()/TODO()
- feature context focus on its method

go context tree example



go cancel context struct

- mixin a Context, use chain-map to find all children

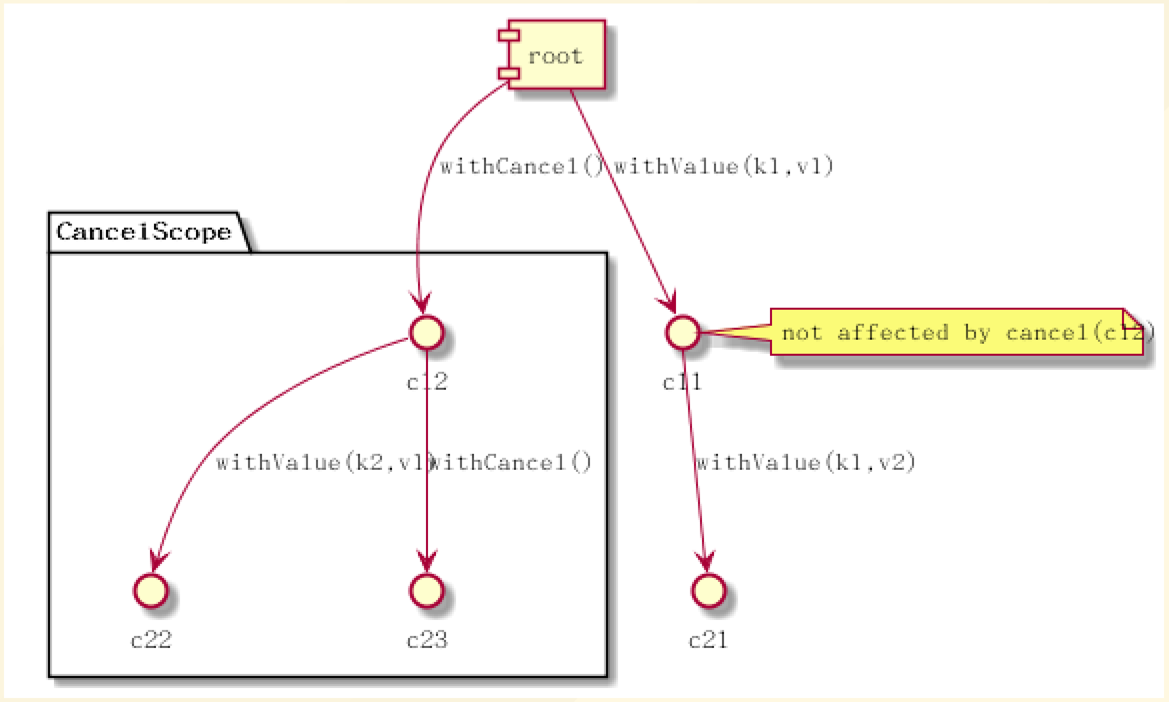
```
type cancelCtx struct {
        Context

        mu       sync.Mutex            // protects followi
        done     chan struct{}         // created lazily, c
        children map[canceler]struct{} // set to nil by the
        err      error                 // set to non-nil by
}
```

go cancel

- use chain-map to find all children

```
for child := range c.children {
        // NOTE: acquiring the child's lock while holding p
        child.cancel(false, err)
}
```

go cancel tree example



go timer context

- mixin a cancelCtx
- only focus on Deadline() and Timer

```go
// A timerCtx carries a timer and a deadline. It embeds a c
// implement Done and Err. It implements cancel by stopping
// delegating to cancelCtx.cancel.
type timerCtx struct {
	cancelCtx
	timer *time.Timer // Under cancelCtx.mu.

	deadline time.Time
}
```

go timer.Timer

- timer is a channel waiting event at given time

```
// from time.sleep.go
// The Timer type represents a single event.
// When the Timer expires, the current time will be sent o
// unless the Timer was created by AfterFunc.
// A Timer must be created with NewTimer or AfterFunc.
type Timer struct {
        C <-chan Time
        r runtimeTimer
}
```

go timer context cancel

- stop timer and cancel its related cancelCtx

```
func (c *timerCtx) cancel(removeFromParent bool, err error)
c.cancelCtx.cancel(false, err)
if removeFromParent {
        // Remove this timerCtx from its parent cancelCtx's
        removeChild(c.cancelCtx.Context, c)
}
c.timer.Stop()
c.timer = nil
}
```

go value context

- find parent if not found

```
type valueCtx struct {
        Context
        key, val interface{}
}

func (c *valueCtx) Value(key interface{}) interface{} {
        if c.key == key {
                return c.val
        }
        return c.Context.Value(key)
}
```

go context deadline/timeout example

- when timer event trigger, it send cancel signal to its cancelCtx, then Done() received signal

```
ctx, cancel := context.WithDeadline(context.Background(), 
// Even though ctx will be expired, it is good practice to
// cancelation function in any case.
defer cancel()
select {
case <-time.After(1 * time.Second):
    fmt.Println("overslept")
case <-ctx.Done():
    fmt.Println(ctx.Err())
}
```