Go Worst Practices

李俱顺(Kevin Lee)

EKEYNOW Pte Ltd.

$ whoami

About Me

- Gopher & Pythonista
- CTO @ EKEYNOW Pte Ltd.
- Community Contributor @ GoCN

About ekeynow

- A IoT SaaS startup company. All system powered by Go.

“

DAVE SNOWDEN

Best practice is useless, we only learn from worst practice. Trying to copy the best others have done amounts to ignoring context of both place and time, and reduces you to copying which in its turn is the death of innovation.

https://www.zylstra.org/blog/2003/11/keynote_dave_sn/

#1 NOT CLOSING THINGS

NOT CLOSING THINGS

```go
func HTTPGet() {
  ...
  resp, err := http.Get(target)
  if err ≠ nil {
    return
  }
  b, err := io.ReadAll(resp.Body)
  if err ≠ nil {
    return
  }
  ...
}
```

```go
func HTTPGet() {
  ...
  resp, err := http.Get(target)
  if err ≠ nil {
    return
  }
  defer resp.Body.Close()
  b, err := io.ReadAll(resp.Body)
  if err ≠ nil {
    return
  }
  ...
}
```

https://zhuanlan.zhihu.com/p/48039838

NOT CLOSING THINGS

- Database–related
- File/socket/connection
- Something should be close.

#2 NOT SETTING TIMEOUT

NOT SETTING TIMEOUT

```go
func HTTPGet() {
  ...
  resp, err := http.Get(target)
  if err ≠ nil {
    return
  }
  b, err := io.ReadAll(resp.Body)
  if err ≠ nil {
    return
  }
  ...
}
```

```go
func HTTPGet() {
  ...
  client := &http.Client{
    Timeout: 6 * time.Second,
  }
  resp, err := client.Get(target)
  if err ≠ nil {
    return
  }
  defer resp.Body.Close()
  b, err := io.ReadAll(resp.Body)
  if err ≠ nil {
    return
  }
  ...
}
```

https://zhuanlan.zhihu.com/p/48039838

NOT SETTING TIMEOUT

```
srv := &http.
    Addr:
    Handler:
    ReadTimeo
    WriteTime
    MaxHeader
}
```

```
sql.Open(

sql.Open(
```

```
ctx, cancel := context.WithTimeout(c, time.Second * 5)
defer cancel()

select {
case sampleItem := <-chanSample:
    ...
case <-ctx.Done():
    ...
}
```

https://hackernoon.com/avoiding-memory-leak-in-golang-api-1843ef45fca8

NOT SETTING TIMEOUT

Go Concurrency Patterns: Context

https://blog.golang.org/context

#3 OVERREACHING FOR CONCURRENCY

OVERREACHING FOR CONCURRENCY

```go
func finishReq(timeout time.Duration) r ob {
    ch := make(chan ob)
    go func() {
        result := fn()
        ch ← result // block
    } ()
    select {
    case result = ← ch:
        return result
    case ← time.After(timeout):
        return nil
    }
}
```

```go
func finishReq(timeout time.Duration) r ob {
    ch := make(chan ob, 1)
    go func() {
        result := fn()
        ch ← result // block
    } ()
    select {
    case result = ← ch:
        return result
    case ← time.After(timeout):
        return nil
    }
}
```

https://github.com/system-pclub/go-concurrency-bugs

OVERREACHING FOR CONCURRENCY

Concurrency is Hard.

- CSP does not change that.

https://medium.com/oreillymedia/why-concurrency-is-hard-f93104cad54b

OVERREACHING FOR CONCURRENCY

Application	Stars	Commits	Contributors	LOC	Dev History
Docker	48975	35149	1767	786K	4.2 Years
Kubernetes	36581	65684	1679	2297K	3.9 Years
etcd	18417	14101	436	441K	4.9 Years
CockroachDB	13461	29485	197	520k	4.2 Years
gRPC-go	5594	2528	148	53K	3.3 Years
BoltDB	8530	816	98	9K	4.4 Years

OVERREACHING FOR CONCURRENCY

Application	Behavior		Root Cause	
	Blocking	Non-Blocking	Shared Memory	Message Passing
Docker	21	23	28	16
Kubernetes	17	17	20	14
etcd	21	16	18	19
CockroachDB	12	16	23	5
gRPC-Go	11	12	12	11
BoltDB	3	2	4	1
Total	85	86	105	66

OVERREACHING FOR CONCURRENCY

Single thread is still powerful & relaxed.

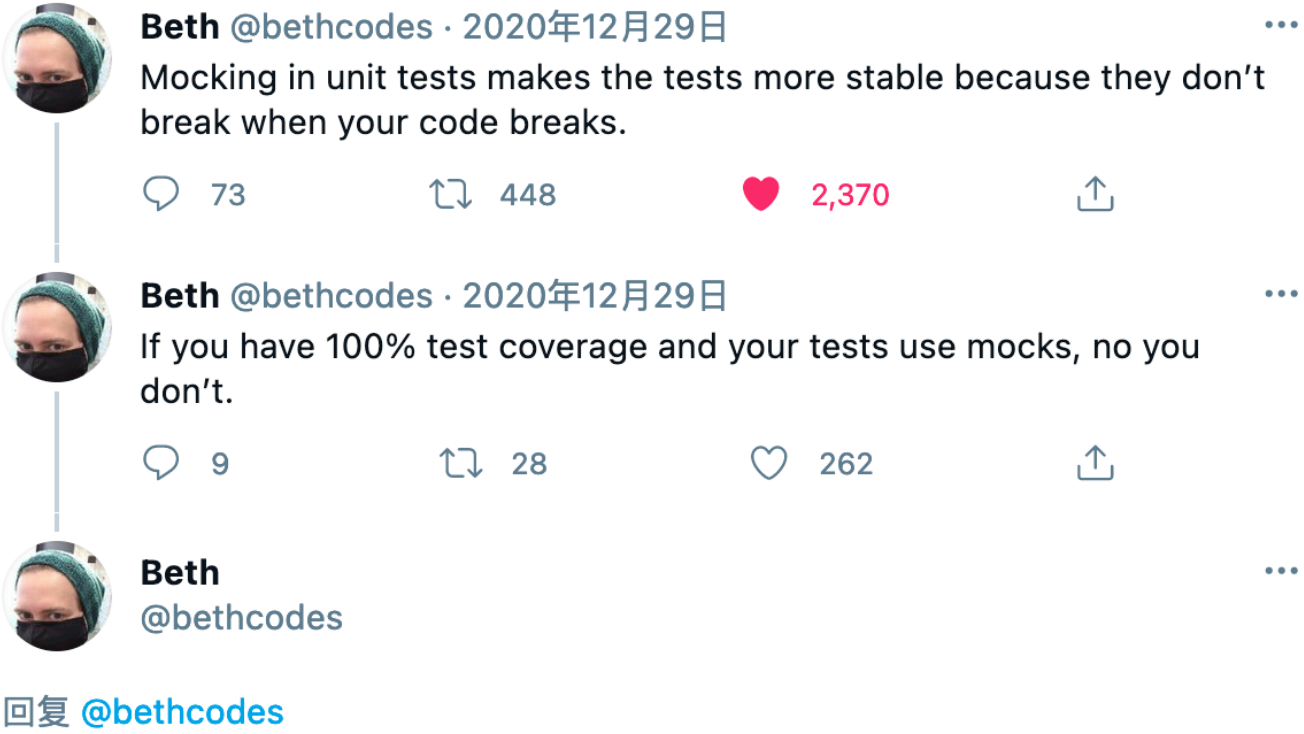
OVERREACHING FOR CONCURRENCY

Scalability! But at what COST?

https://www.usenix.org/system/files/conference/hotos15/hotos15-paper-mcsherry.pdf

#4 EXCESSIVE MOCKING

EXCESSIVE MOCKING



Business logic loves to hide in the quiet spaces between the classes, in the function tables, amongst the matchers. It wants us to miss it, to break at the most inopportune time, but we can outfox it by clearly stating our intentions for the system's behaviour.

EXCESSIVE MOCKING

Wait… don’t I need interface to mock?

It depends.

EXCESSIVE MOCKING

Be careful with interface mocking

- Mocking is not real

EXCESSIVE MOCKING

Testing your application in real world.

#5 LARGE INTERFACE

LARGE INTERFACE

Domain Driven Design microservice

LARGE INTERFACE

```go
type FoodAppInterface interface {
    GetAllFood() ([]entity.Food, error)
    GetFood(uint64) (*entity.Food, error)
    UpdateFood(*entity.Food) (*entity.Food, map[string]string)
    SaveFood(*entity.Food) (*entity.Food, map[string]string)
    DeleteFood(uint64) error
    DeleteAllFood([]uint64) error
}
```

“

ROB PIKE

The bigger the interface, the weaker the abstraction.

https://www.youtube.com/watch?v=PAAkCSZUG1c&t=317s

“

JOHN OUSTERHOUT

It is more important for a module to have a simple interface than a simple implementation.

https://www.youtube.com/watch?v=bmSAYlu0NcY

LARGE INTERFACE

- Abstractions that have simple interfaces (deep modules) but hide complex functionality help reduce the complexity of programs.
- Deep modules more better than shallow modules do – modules that have a simple implementation, but complex interfaces.

LARGE INTERFACE

```go
type IO interface {
    Read(p []byte) (n int, err error)
    Write(p []byte) (n int, err error)
    Close() error
}
```

```go
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Writer interface {
    Write(p []byte) (n int, err error)
}

type Closer interface {
    Close() error
}

type ReadCloser interface {
    Reader
    Closer
}

type WriteCloser interface {
    Writer
    Closer
}
```

#6 INTERFACE POLLUTION

INTERFACE POLLUTION

```go
type FoodAppInterface interface {
    SaveFood(*entity.Food) (*entity.Food, map[string]string)
    GetAllFood() ([]entity.Food, error)
    GetFood(uint64) (*entity.Food, error)
    UpdateFood(*entity.Food) (*entity.Food, map[string]string)
    DeleteFood(uint64) error
}

type foodApp struct {
    fr repository.FoodRepository
}

func (f *foodApp) SaveFood(food *entity.Food) (*entity.Food, map[string]string) {
    return f.fr.SaveFood(food)
}

func (f *foodApp) GetAllFood() ([]entity.Food, error) {
    return f.fr.GetAllFood()
}
...
```

INTERFACE POLLUTION

Why is this bad?

- Good abstraction is hard.
- Force interface onto the implementor.
- Dependency injection?

INTERFACE POLLUTION

defer abstraction()

#7 DRY TRAP

DRY TRAP

```go
func uploadFile(path string) (string, error) {
    // blahblah
}

func handler1() {
    ...
    uri, err := uploadFile(path)
    ...
}

func handler2() {
    ...
    uri, err := uploadFile(path)
    ...
}
```

DRY TRAP

```go
func uploadFile(f os.File, path string) (string, error) {
    // blahblah
}

func handler1() {
    ...
    uri, err := uploadFile(f, "")
    ...
}

func handler2() {
    ...
    uri, err := uploadFile(f, "blah")
    ...
}
```

DRY TRAP

defer DRY()

Q&A

Thanks for your patience