FUNCTIONAL OPTION

A pattern to implement scalable library APIs.

DISCLAMER

- I'm not the author of the pattern introduce here. I just show a way to apply it when building package.
- Code show in this slide is faked, and the style is not nice due to spacing constraint in a slide.

Notes: I learn it from Rob Pike Self-referential functions articles And using the name Functional Option popularized by Dave Cheney because it's easier to remember.

CONTENT

THE PROBLEM

- Package/sevice/function usually starts smalls
- More and more features:
  - Its APIs become more and more complex
  - Next features become harder to implement.

EXAMPLE: ITEM LOADING FUNCTION

Its starts simple

```
func (dm *itemdm) load(shopid, itemid int64) Item {...}
```

Then, we need to find deleted items

```
func (dm *itemdm) Load(
  shopid, itemid int64,
  needDeleted bool,
) Item {...}
```

Then, we need models as well, to reduce requests

```
func (dm *itemDM) Load(
  shopid, itemid int64,
  needModels, needDeleted bool,
) Item {...}
```

Its usage be like:

```go
func main() {
	dm := &ItemDM{}
	_ = dm.Load(123, 4567, true, false)
}
```

Code readers:

Hey, is this load deleted item without models or ...? Nevermind, let's check that function again.

We're not done yet!

We need some flag to:

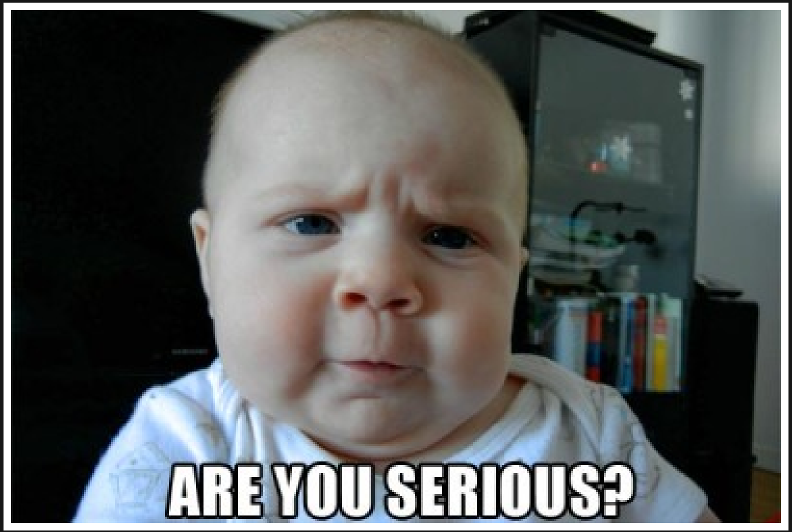
- Read from?
  - slave
  - cache
- Recalcule?
  - safe
  - unsafe

Easy, just change it to:

```
func (dm *ItemDM) Load(
  shopid, itemid int64,
  needModels bool,
  useSlaveAPI bool
  useUnsafeAPI bool,
) Item {...}
```

And use it like:

```go
func main() {
  dm := &ItemDM{}
  _ = dm.Load(123, 4567,
    true, false, false, true,
  )
}
```



Notes: ask audience if they want to skip next example

EXAMPLE: LOGGER

It starts simple

```go
type Logger interface {
  Printf(msg string, args... interface{})
}

func New(io.Writer) *Logger {...}
```

Its usage is simple:

```go
func main() {
  lg := logger.New(os.Stdout)
  lg.Printf("hello")
  // output: main.go:3 | hello
}
```

Then, new features:

- prefix
- log level

```go
type Logger interface {
  Printf(msg string, args... interface{})
  Debuf(msg string, args... interface{})
  Errorf(msg string, args... interface{})
}

func New(
  out io.Writer,
  prefix string,
  level int,
) *Logger {...}
```

Usage still quite simple, no need a config struct yet.

```go
func main() {
  lg := logger.New(os.Stdout, "IIS", logger.DEBUG)
  lg.Printf("hello")
  // output: main.go:3 | DEBUG
}
```

Then more features

- file rotation
- split file by log level
- async
- write to network
- serialization
    - protobuf
    - json
    - text
- ...

Need a config structs

```
package logger

type Config struct {
  Level     string
  LogPath        string
```

```go
  Handlers []Handler
}

type Handler struct {
  AsyncConfig    AsyncConfig
  FormatConfig   FormatConfig
  RolloverConfig RolloverConfig
}

type AsyncConfig struct {...}
```

And the usage be like 😂

```go
func main() {
  handlerConfig := logger.FileHandlerConfig{
    Type:    "FileHandler",
    Levels: []string{"debug", "trace", "info", "warn", "error"
    Sync: multilevel.LogSyncConfig{
      SyncWrite:     syncWrite,
      FlushInterval: 100,
      QueueSize:     uint32(queueSize),
    },
    File: fileFullPath,
    Message: multilevel.LogMessageConfig{
      Format:       "short",
      FieldsFormat: "text",
    MaxBytes:      10 * 1024 * 1024,
```

IDEAS

- Define a config struct
- Define sensible default
- *Define a scalable interface*
- *Provide some optional functions* help user modify the config to the state their need.

Notes: let do it step by step

DEFINE A CONFIG STRUCT

```
type Option struct {
  useSlave    bool
  useUnsafe   bool
  needModels  bool
  needDeleted bool
}
```

DEFINE SENSIBLE DEFAULT

```
func defaultOps() *Option {
  return &Option{
    useSlave:  !globalCfg.UseCache,
    useUnsafe: false, // must be set explicitly
  }
}
```

DEFINE A SCALABEL INTERFACE

```
func (dm *ItemDM) Load(
  shopid, itemid int64,
  mods... OptionMod,
) Item {...}

// OptionMod is a function that modifies the input Option
type OptionMod func(o *Option)
```

- The `OptionMod` are optional arguments.
- We can provide more `OptionMod` as we adding more features.
- Existing code won't break because our API doesn't change.

PROVIDE OPTION FUNCTIONS

```
func UseSlaveAPI(b bool) OptionMod {
  return func(o *Option) {o.useSlave = b}
}
func UseUnsafeAPI(b bool) OptionMod {
  return func(o *Option) {o.useUnsafe = b}
}
func NeedModels(b bool) OptionMod {
  return func(o *Option) {o.needModels = b}
}
func NeedDeletedItem(b bool) OptionMod {
  return func(o *Option) {o.needDeleted = b}
}
```

- Each one is short, easy to skim, clearly self-documented

API USAGE

```
func main() {
  dm := &ItemDM{}
  // defualt option
  item, := dm.Load(123, 4567)
  // use custormized config
  item, := dm.Load(123, 4567,
    NeedModels(globalCfg.LoadModels)
  )

  item, := dm.Load(123, 4567,
    UseUnsafeAPI(true), NeedModels(false),
    needDeleted(true), useSlaveAPI(false)
  )
}
```

- Clear usage intention
- Safe to refactor or reorder `OptionMod`

DOWNSIDES

- More functions to defines
- Naming those functions might be hard.
- Usage code is more verbose

APPLICATIONS

Open source

- sqlboiler: a Go ORM generator

Shopee Internal

- `ItemInfoClient.go` in Core Server
- `sps.NewAgent()` in sps lib.
- `spkit.Client()` and `spkit.Server()`

SQLBOILER

Example from their readme.

```
// Query all users
users, err := models.Users().All(ctx, db)

// complex query
users, err := models.Users(
  Where("age > ?", 30),
  Limit(5),
  Offset(6),
).All(ctx, db)
```

SPS

```go
func NewAgent(opts ...InitOption) (ag Agent, err error) {...}
// usage
sps.NewAgent(
  sps.WithInstanceID(iid),
  sps.WithConfigKey(cfg.ConfigKey),
)

// Init global agent
func Init(opts ...InitOption) error {...}
// Usage
_ = sps.Init(
  sps.WithInstanceID(iid),
  sps.WithConfigKey(configKey),
)
```

Btw, that lib has a function that drive me crazy

```go
_, _ := sps.GenerateInstanceID(
  "item.info", "", "", "", "", ""
)
```

Which of the 4 strings to put "test" as env name?