FUNCTION PROGRAMMING IN GO

Very basic introduction to FP and some techniques in Go.

CONTENT

Theory	- Difference between 2 paradigm - Which paradigm is Go?
Techniques	- Higher-order functions - Closure - Currying and Partial Functions - Tail recursion
Wrap up	- Take away - Q&A

DIFFERENCE BETWEEN 2 PARADIGM

Warning: this is a very shallow overview.

OOP CONCEPTS

- **Objects** are instances of **class** (or **prototype**), contain **data**, expose and communicate via **methods**.
- **Encapsulation**: binds data and functions that manipulate it, keeps both safe from outside interference and misuse
- **Composition**: object can contains other objects.
- **Inheritance**: share code between those **class** that can be seen as *same thing*
- **Polymorphism**: objects in an inheritance hierarchy may behave differently if same method is called.

FP CONCEPTS

- Function are **first-class**
- **Higher-order** functions: take other function as input.
- **Pure** functions: no side-effect, good for optimization
  - If result of a pure expression is unused, it can be removed.
  - Result is same if argument is unchanged: cache, or substitute at compile time.
  - If there is no dependency between 2 pure functions: change execution order or parallel...
- **Referential transparency**: Expressions can be replaced by their values.

THE DIFFERENCE

FP	OOP
Emphasize on function evaluation	Emphasize on protecting data
Favor pure functions	Favor side-effect with mutable data
Declarative	Imperative
Few things with more operations	*Many things with clear operations*

WHICH PARADIGM IS GO?

Go is OK, but not great for both paradigms.

- OOP:
  - Go have **struct** instead of **class**, don't have inheritance, OK for Encapsulation, Composition.
  - Polymorphism can be done with `interface`.
- FP:
  - Has **first class function**.
  - No generic, hence function composition is limited.

HIGHER ORDER FUNCTIONS

Similar to Strategy and Visitor patterns in OOP.

STANDARD LIBRARY EXAMPLES

sorts

```
ss := []string{"a", "b"}
sort.Slice(ss, func(i, j int) bool { return ss[i] < ss[j] })
sort.Search(ss, func(i) bool {return ss[i] == "x"})
```

testing:

```
func (t *T) Run(name string, f func(t *T)) bool {...}
```

sync

```
func (m *Map) Range(f func(key, value interface{}) bool) {...}
```

INTERFACES METHOD TRICK - 1

Given following design

```
type Cache interface {
    Set(name string)
    AsyncSet(name string)
}

var p = fmt.Println

type redis struct{}
func (redis) Set(name string)      { p("Set, " + name) }
func (redis) AsyncSet(name string) { p("AsyncSet, " + name) }

type mem struct{}
func (mem) Set(name string)      { p("Set, " + name) }
func (mem) AsyncSet(name string) { p("AsyncSet, " + name) }
```

INTERFACES METHOD TRICK - 2

This code is valid

```
func TestCache_Set(t *testing.T) {
    tests := []struct {
        name string
        g    Cache
        fn   func(Cache, string)
    }{
        {name: "redis", g: redis{}, fn: Cache.Set},
        {name: "redis", g: redis{}, fn: Cache.AsyncSet},
        {name: "mem",   g: mem{},   fn: Cache.Set},
        {name: "mem",   g: mem{},   fn: Cache.AsyncSet},
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            tt.fn(tt.g, tt.name)
        })
    }
}
```

INTERFACES METHOD TRICK - TAKE AWAYS

- In go, these 2 are equivalent:
  - `<type>.<func>(args...)`
  - `<func>(<type>, args...)`.
- We can use that design to simplify many code.
- Real world examples:
  - memcached `onItem`

CLOSURE

A closure is a function value that references variables from **outside its body**.

The function may access and assign to the referenced variables; in this sense the function is **bound** to the variables.

EXAMPLES

```go
func makeJoiner(sep string) func(ss ...string) string {
    return func(ss ...string) string {
        return strings.Join(ss, sep)
    }
}

comma := makeJoiner(",")
fmt.Println(comma("a", "b")) // a,b
pipe := makeJoiner("|")
fmt.Println(pipe("a", "b")) // a|b
```

Real world example:

- go-redis

CURRYING AND PARTIAL FUNCTIONS

- **Currying**: Converting a function that takes multiple arguments into a sequence of functions that each take **fewer arguments**.
- **Partial function**: Fixing a number of arguments to a function, producing another function of smaller arguments list.

THEORY EXAMPLE

Given `x = f(a,b,c)`. `f` can be curried into 3 *partial functions* `m`, `n`, `p`, such that:

```
n = m(a)
p = n(b)
x = p(c)
```

Thus `x = f(a,b,c) = m(a)(b)(c)`.

EXAMPLE

```go
func key(prefix string, id int) string {
    return prefix + "::" + strconv.Itoa(id)
}

func makeKeyFn(prefix string) func(id int) string {
    return func(id int) string {
        return key(prefix, id)
    }
}

var itemKey = makeKeyFn("item")
var modelKey = makeKeyFn("model")

fmt.Println(itemKey(10)) // item::10
fmt.Println(modelKey(20)) // model::20
```

TAIL RECURSION

A tail call is a subroutine call performed as the **final action of a procedure**. If a tail call might lead to the same subroutine being called again later in the call chain, the subroutine is said to be tail-recursive.

This that **can be implemented without adding a new stack frame** to the call stack. Most of the frame of the current procedure is no longer needed, and can be replaced by the frame of the tail call, modified as appropriate

EXAMPLE - COMPUTE FACTORIAL

```
func iter(n int) int {
    res := 1
    for n > 0 { res *= n; n-- }
    return res
}

func recur(n int) int {
    if n == 1 { return 1 }
    return recur(n-1) * n
}

func tail(n int) int { return tailHelper(n-1, n) }

func tailHelper(i int, cur int) int {
    if i <= 0 { return cur }
    return tailHelper(i-1, i*cur)
}
```

BENCHMARK

```
Benchmark/recur,_n=10-12        50000000        27.9 ns/op
Benchmark/tail,_n=10-12         100000000       23.1 ns/op
Benchmark/iter,_n=10-12         200000000       6.09 ns/op
Benchmark/recur,_n=20-12        50000000        28.2 ns/op
Benchmark/tail,_n=20-12         100000000       23.1 ns/op
Benchmark/iter,_n=20-12         300000000       5.91 ns/op
```

Tail recursion is not as fast as pure iteration, but it's better than normal recursion, with the trade off is code become more verbose.

TAKE AWAYS

- Minimize side-effect.
- Function is data.
- Love but don't abuse closure.
- Mind the tail.

Q&A

REFERENCE

- SOLID design principles
- Wiki Functional Programming
- Functional Go, Medium article by Geison
- Tail recursion in Go
- When do you choose FP over OOP?
- Tail call