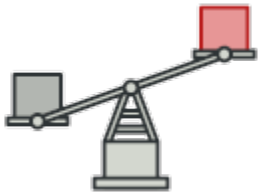
refactoring.guru

Flyweight em Go / Padrões de Projeto

6–8 minutes



O **O Flyweight** é um padrão de projeto estrutural que permite que os programas suportem grandes quantidades de objetos, mantendo baixo o consumo de memória.

O padrão consegue isso compartilhando partes do estado do objeto entre vários objetos. Em outras palavras, o Flyweight economiza RAM armazenando em cache os mesmos dados usados por objetos diferentes.

Exemplo conceitual

Em um jogo de Counter-Strike, os Terroristas e Antiterroristas têm um tipo de uniforme diferente. Para simplificar, vamos supor que tanto os Terroristas quanto os Antiterroristas tenham um tipo de uniforme cada. O objeto uniforme (dress) é embutido no objeto jogador (player) como abaixo.

Abaixo está a struct de um player. Podemos ver que o objeto dress está incorporado na struct do player:

type player struct {

```
    dress      dress
    playerType string
    lat        int
    long       int
}
```

Digamos que haja 5 Terroristas e 5 Antiterroristas, então um total de 10 players. Agora, existem duas opções de dress.

1. Cada um dos 10 objetos player cria um objeto dress diferente e os incorpora. Um total de 10 objetos dress serão criados.
2. Criamos dois objetos dress:

- Objeto Dress Terrorista Único: Isso será compartilhado por 5 terroristas.
- Objeto Dress Antiterrorista Único: Isso será compartilhado por 5 antiterroristas.

Como você pode ver na abordagem 1, um total de 10 objetos dress são criados, enquanto na abordagem 2 apenas 2 objetos dress são criados. A segunda abordagem é a que seguimos no padrão de design Flyweight. Os dois objetos dress que criamos são chamados de objetos flyweight.

O padrão Flyweight remove as partes comuns e cria objetos flyweight. Esses objetos flyweight (dress) podem então ser compartilhados entre vários objetos (player). Isso reduz drasticamente o número de dresses, e a parte boa é que mesmo se você criar mais players, apenas dois dresses serão suficientes.

No padrão flyweight, armazenamos os objetos flyweight no campo do mapa. Sempre que os outros objetos que compartilham os objetos flyweight são criados, então os objetos flyweight são

buscados no mapa.

Vamos ver quais partes desse arranjo serão estados intrínsecos e extrínsecos:

- *Estado Intrínseco*: O uniforme está no estado intrínseco, pois pode ser compartilhado entre vários objetos Terroristas e Antiterroristas.
- *Estado Extrínseco*: A localização do jogador e a arma do jogador são um estado extrínseco, pois são diferentes para cada objeto.

dressFactory.go: Flyweight factory

```
package main

import "fmt"

const (

    TerroristDressType = "tDress"

    CounterTerrroristDressType = "ctDress"
)

var (
    dressFactorySingleInstance = &DressFactory{
        dressMap: make(map[string]Dress),
    }
)

type DressFactory struct {
```

```
    dressMap map[string]Dress
}

func (d *DressFactory) getDressByType(dressType string) (Dress,
error) {
    if d.dressMap[dressType] != nil {
        return d.dressMap[dressType], nil
    }

    if dressType == TerroristDressType {
        d.dressMap[dressType] = newTerroristDress()
        return d.dressMap[dressType], nil
    }
    if dressType == CounterTerrroristDressType {
        d.dressMap[dressType] = newCounterTerroristDress()
        return d.dressMap[dressType], nil
    }

    return nil, fmt.Errorf("Wrong dress type passed")
}

func getDressFactorySingleInstance() *DressFactory {
    return dressFactorySingleInstance
}
```

dress.go: Interface do flyweight

```
package main

type Dress interface {
```

```
    getColor() string
}
```

terroristDress.go: Objeto flyweight concreto

```
package main

type TerroristDress struct {
    color string
}

func (t *TerroristDress) getColor() string {
    return t.color
}

func newTerroristDress() *TerroristDress {
    return &TerroristDress{color: "red"}
}
```

counterTerroristDress.go: Objeto flyweight concreto

```
package main

type CounterTerroristDress struct {
    color string
}

func (c *CounterTerroristDress) getColor() string {
    return c.color
}
```

```
func newCounterTerroristDress() *CounterTerroristDress {
    return &CounterTerroristDress{color: "green"}
}
```

player.go: Contexto

```
package main

type Player struct {
    dress      Dress
    playerType string
    lat        int
    long       int
}

func newPlayer(playerType, dressType string) *Player {
    dress, _ :=
getDressFactorySingleInstance().getDressByType(dressType)
    return &Player{
        playerType: playerType,
        dress:      dress,
    }
}

func (p *Player) newLocation(lat, long int) {
    p.lat = lat
    p.long = long
}
```

game.go: Código cliente

```
package main

type game struct {
    terrorists       []*Player
    counterTerrorists []*Player
}

func newGame() *game {
    return &game{
        terrorists:       make([]*Player, 1),
        counterTerrorists: make([]*Player, 1),
    }
}

func (c *game) addTerrorist(dressType string) {
    player := newPlayer("T", dressType)
    c.terrorists = append(c.terrorists, player)
    return
}

func (c *game) addCounterTerrorist(dressType string) {
    player := newPlayer("CT", dressType)
    c.counterTerrorists = append(c.counterTerrorists, player)
    return
}
```

main.go: Código cliente

```
package main

import "fmt"

func main() {
    game := newGame()


    game.addTerrorist(TerroristDressType)
    game.addTerrorist(TerroristDressType)
    game.addTerrorist(TerroristDressType)
    game.addTerrorist(TerroristDressType)


    game.addCounterTerrorist(CounterTerrroristDressType)
    game.addCounterTerrorist(CounterTerrroristDressType)
    game.addCounterTerrorist(CounterTerrroristDressType)

    dressFactoryInstance := getDressFactorySingleInstance()

    for dressType, dress := range dressFactoryInstance.dressMap {
        fmt.Printf("DressColorType: %s\nDressColor: %s\n",
dressType, dress.getColor())
    }
}
```

output.txt: Resultados da execução

```
DressColorType: ctDress
DressColor: green
```

DressColorType: tDress

DressColor: red