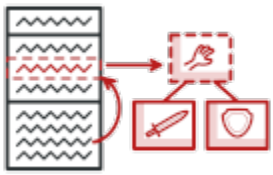
refactoring.guru

Strategy em Go / Padrões de Projeto

4–5 minutes



O **Strategy** é um padrão de projeto comportamental que transforma um conjunto de comportamentos em objetos e os torna intercambiáveis dentro do objeto de contexto original.

O objeto original, chamado contexto, mantém uma referência a um objeto strategy e o delega a execução do comportamento. Para alterar a maneira como o contexto executa seu trabalho, outros objetos podem substituir o objeto strategy atualmente vinculado por outro.

Exemplo conceitual

Suponha que você esteja criando um cache na memória. Por estar na memória, seu tamanho é limitado. Sempre que atinge seu tamanho máximo, algumas entradas precisam ser despejadas para liberar espaço. Isso pode acontecer por meio de vários algoritmos. Alguns dos algoritmos populares são:

- Least Recently Used (LRU - Menos Usada Recentemente): remove uma entrada que foi menos usada recentemente.

- First In, First Out (FIFO - Primeira a entrar, primeira a sair): remove uma entrada que foi criada primeiro.
- Least Frequently Used (LFU - Menos Usada Frequentemente): remove uma entrada que foi usada com menos frequência.

  O problema é como desacoplar nossa classe de cache desses algoritmos para que possamos alterar o algoritmo em tempo de execução. Além disso, a classe de cache não deve ser alterada quando um novo algoritmo está sendo adicionado.

  É aqui que o padrão Strategy entra em cena. Ele sugere a criação de uma família do algoritmo com cada algoritmo tendo sua própria classe. Cada uma dessas classes segue a mesma interface, e isso torna o algoritmo intercambiável dentro da família. Digamos que o nome comum da interface seja `evictionAlgo`.

  Agora nossa classe de cache principal irá incorporar a interface `evictionAlgo`. Em vez de implementar todos os tipos de algoritmos de remoção em si, nossa classe de cache irá delegar tudo para a interface `evictionAlgo`. Como `evictionAlgo` é uma interface, podemos executar a alteração do algoritmo em tempo de execução para LRU, FIFO, LFU sem qualquer alteração na classe de cache.

  evictionAlgo.go: Interface de estratégia

```
package main

type EvictionAlgo interface {
    evict(c *Cache)
}
```

fifo.go: Estratégia concreta

```
package main

import "fmt"

type Fifo struct {
}

func (l *Fifo) evict(c *Cache) {
    fmt.Println("Evicting by fifo strtegy")
}
```

lru.go: Estratégia concreta

```
package main

import "fmt"

type Lru struct {
}

func (l *Lru) evict(c *Cache) {
    fmt.Println("Evicting by lru strtegy")
}
```

lfu.go: Estratégia concreta

```
package main
```

```
import "fmt"

type Lfu struct {
}

func (l *Lfu) evict(c *Cache) {
    fmt.Println("Evicting by lfu strtegy")
}
```

cache.go: Context

```
package main

type Cache struct {
    storage      map[string]string
    evictionAlgo EvictionAlgo
    capacity     int
    maxCapacity  int
}

func initCache(e EvictionAlgo) *Cache {
    storage := make(map[string]string)
    return &Cache{
        storage:      storage,
        evictionAlgo: e,
        capacity:     0,
        maxCapacity:  2,
    }
}
```

```
func (c *Cache) setEvictionAlgo(e EvictionAlgo) {
    c.evictionAlgo = e
}

func (c *Cache) add(key, value string) {
    if c.capacity == c.maxCapacity {
        c.evict()
    }
    c.capacity++
    c.storage[key] = value
}

func (c *Cache) get(key string) {
    delete(c.storage, key)
}

func (c *Cache) evict() {
    c.evictionAlgo.evict(c)
    c.capacity--
}
```

main.go: Código cliente

```
package main

func main() {
    lfu := &Lfu{}
    cache := initCache(lfu)

    cache.add("a", "1")
```

```
    cache.add("b", "2")

    cache.add("c", "3")

    lru := &Lru{}
    cache.setEvictionAlgo(lru)

    cache.add("d", "4")

    fifo := &Fifo{}
    cache.setEvictionAlgo(fifo)

    cache.add("e", "5")

}
```

output.txt: Resultados da execução

```
Evicting by lfu strtegy
Evicting by lru strtegy
Evicting by fifo strtegy
```