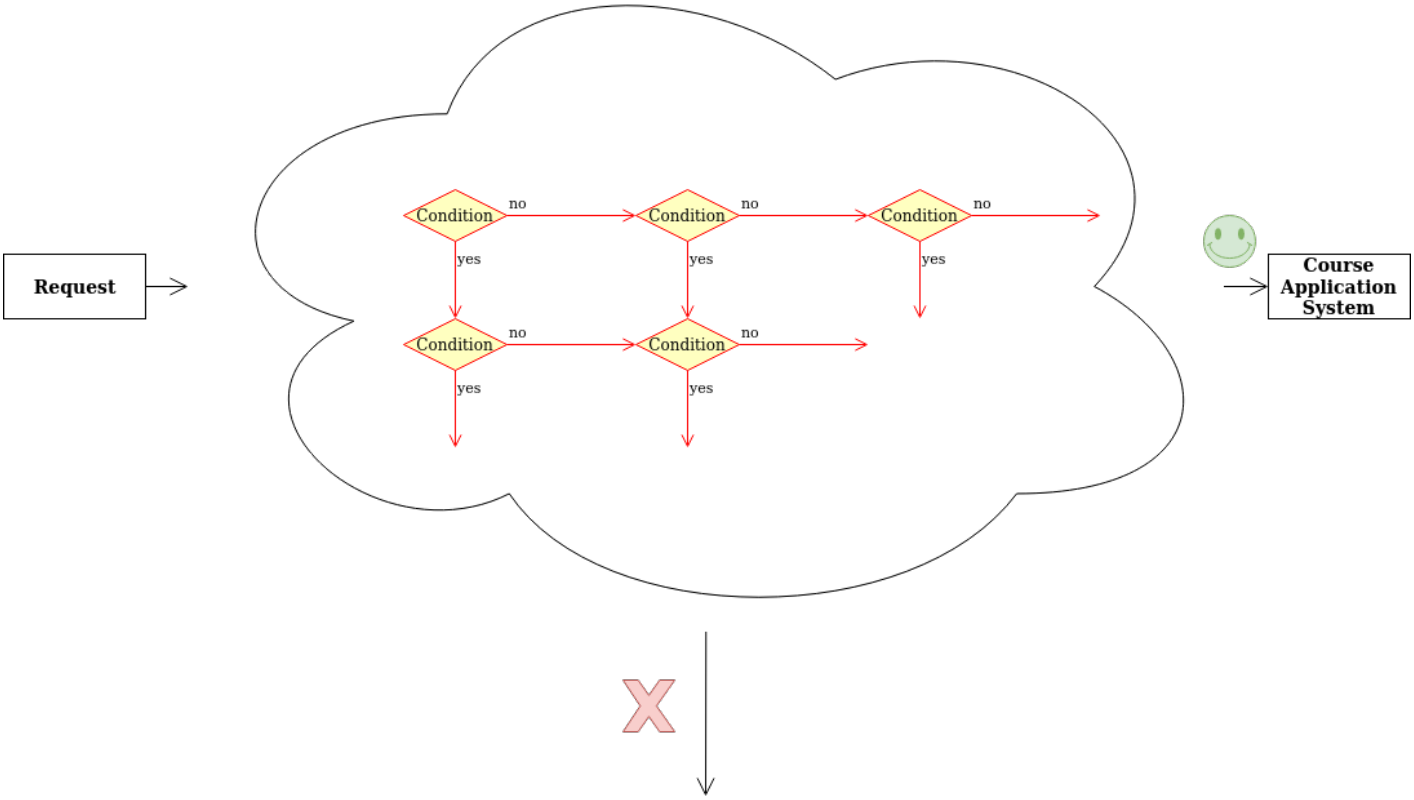
Если вы видите что-то необычное, просто сообщите мне.

Простая инструкция изучения Цепочи обязаностей в Golang

Что такое цепочка обязаностей(Chain of Responsibility)?

- Поведенческий шаблон проектирования который позволяет вам передавать запрос через цепочку обработчиков.
- При получении запроса, каждый обработчик решает сам обрабатывать запрос или передать его дальше по цепочке.

Проблема



Представим, вы рабоаете над приложениме онлаин курсов. Вот список проверок перед тем как можно использовать курс.

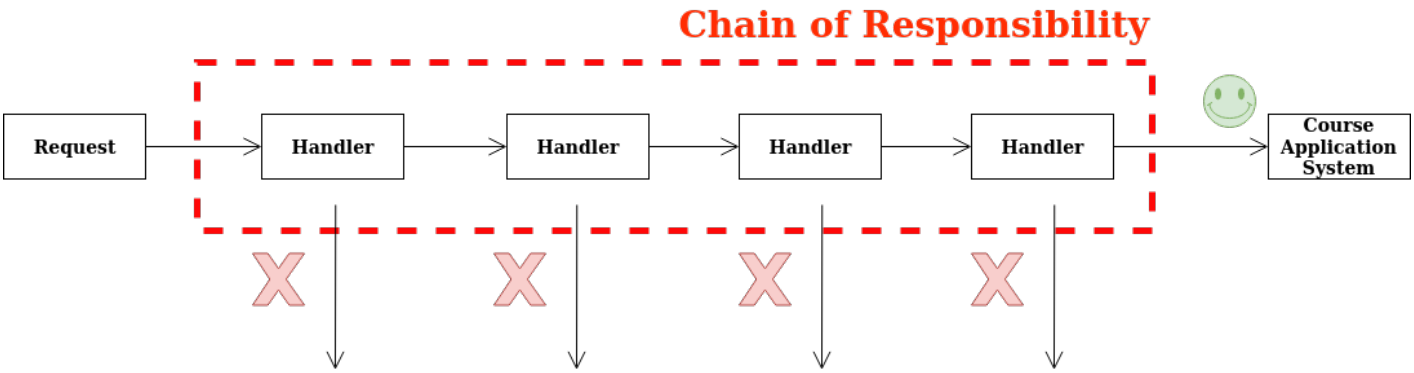
- Один из ваших коллег, Джимм, предположил, что это не безопасно передавать чистые данные в приложении. Поэтому вы добавли дополнительную проверку шага для обработки данных в запросе.
- One of your colleagues, Tommy, noticed that the system is vulnerable to brute force password cracking. To negate this, you promptly added a check that filters repeated failed requests coming from the same IP address.
- One of your colleagues, Daniel, suggested that you could speed up the system by returning cached results on repeated requests containing the same data. Hence, you added another check which lets the request pass through to the system only if there’s no suitable cached response.

Having a big headache

- The code of the checks, which had already looked like a mess, became more and more bloated as you added each new feature
- Changing one check sometimes affected the others

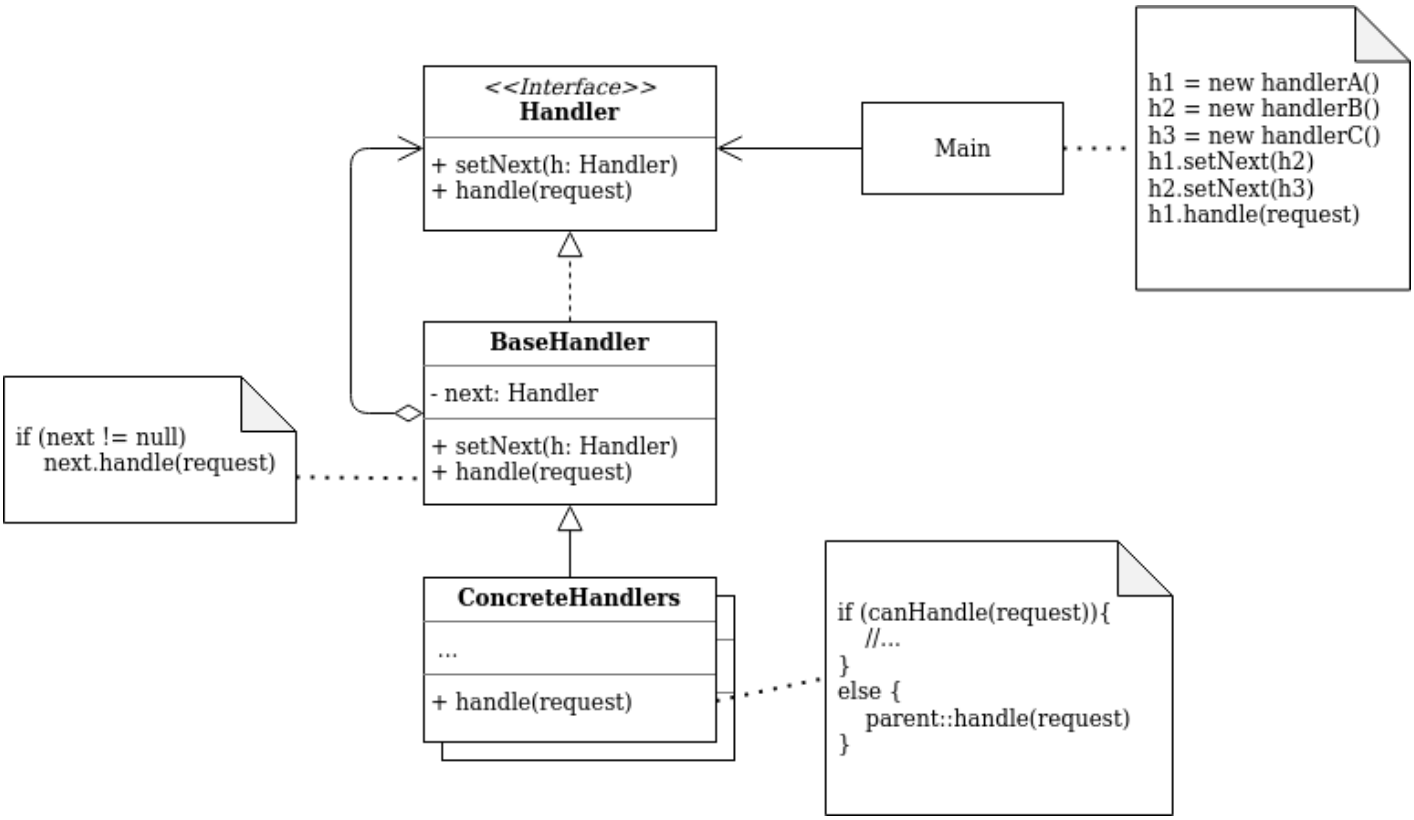
- Worst of all, when you tried to reuse the checks to protect other components of the system, you had to duplicate some of the code since those components required some of the checks, but not all of them.

Solution



- CoR relies on transforming particular behaviors into stand-alone objects called handlers
- The pattern suggests that you link these handlers into a chain
- Each linked handler has a field for storing a reference to the next handler in the chain. In addition to processing a request, handlers pass the request further along the chain
- The request travels along the chain until all handlers have had a chance to process it.

Diagram



Handler

- declares the interface common to all concrete handlers.
- usually contains just 1 method for handling requests and another method to set the next handler in the chain

Base handler

- optional class to put the boilerplate code that’s common to all handler class
- this class defines a field for storing a reference to the next handler
- the main class build a chain by passing a handler to the constructor or setter of the previous handler

Main

- compose chains just once or compose them dynamically depending on the app logic

Concrete Handlers

- contain actual logic for processing requests
- upon receiving a request, each handler decide whether to process it and whether to pass it along the chain

- usually self contained and immutable, accepting all necessary data just one via the constructor

Pros and Cons Pros

- control the order of request handling
- Fulfills *Single Responsibility Princple*
- Fulfills *Open/Closed Principle*

Cons

- some request may end up unhandled

How to code a simple Chain of Responsibility in Golang?

Section

```
package main

type section interface {
execute(*task)
setNext(section)
}
```

Material

```
package main

import (
```

```
	"fmt"
)

type material struct {
	next section
}

func (m *material) execute(t *task) {
	if t.materialCollected {
		fmt.Println("Material already collected")
		m.next.execute(t)
		return
	}
	fmt.Println("Material section gathering materials")
	t.materialCollected = true
	m.next.execute(t)
}

func (m *material) setNext(next section) {
	m.next = next
}
```

Assembly

```
package main

import (
	"fmt"
)

type assembly struct {
	next section
}

func (a *assembly) execute(t *task) {
	if t.assemblyExecuted {
		fmt.Println("Assembly already done")
```

```
□□a.next.execute(t)
□□return
□}
□fmt.Println("Assembly section assembling...")
□t.assemblyExecuted = true
□a.next.execute(t)
}

func (a *assembly) setNext(next section) {
□a.next = next
}
view raw
```

Packaging

```
package main

import (
□"fmt"
)

type packaging struct {
□next section
}

func (p *packaging) execute(t *task) {
□if t.packagingExecuted {
□□fmt.Println("Packaging already done")
□□p.next.execute(t)
□□return
□}
□fmt.Println("Packaging section doing packaging")
}

func (p *packaging) setNext(next section) {
□p.next = next
```

```
}
```

Task

```
package main

type task struct {
	name    string
	materialCollected bool
	assemblyExecuted  bool
	packagingExecuted bool
}
```

Main

```
package main

func main() {
	packaging := &packaging{}

	// set next for assembly section
	assembly := &assembly{}
	assembly.setNext(packaging)

	material := &material{}
	material.setNext(assembly)

	task := &task{name: "truck_toy"}
	material.execute(task)
}
```

Explanation

`section.go` is a Handler interface. This interface handling requests and sets the next handler on the chain.

`material.go` is a Concrete handler. It decides if the request to collect material should be processed and can move up the chain.

`assembly.go` is a Concrete handler. It decides if the request to perform the assembly work should be processed and can move up the chain.

`packaging.go` is a Concrete handler. It decides if the request to perform the packaging work should be processed and can move up the chain.

`main.go` is a client. Initializes up the chain of handlers.

How to run

Command:

```
go run .

Material section gathering materials
Assembly section assembling…
Packaging section doing packaging
```

Takeaways

I hope you understand how to code a simple abstract factory in Golang and more importantly understand how a chain of responsibility design pattern can help you to design your code better and to ensure maintainability.

Here the link to my github page for all source code on chain of responsibility design pattern in Golang: https://github.com/leonardyeoxl/go-patterns/tree/master/behavioral/chain_of_responsibility

---

Revision #1
Created 2021-09-09 06:22:35 UTC by gasick

Updated 2023-04-16 19:30:04 UTC by gasick