GO-02 07: Основы языка. Panic и их обработка

Описание:

В этом разделе поговорим о критических ошибках приложения, которые приводят к аварийной ситуации.
В терминах языка Go аварийная ситуация очень часто называется паникой. Несмотря на смысл этого слова, аварийная ситуация является вполне штатным состоянием приложения. Когда случается критическая ошибка, функция, выполнение которой привело к панике, прекращает свою работу и вызываются запланированные отложенные вызовы. Далее управление передается вызывающей функции, где также выполняются отложенные вызовы и управление передается выше до функции main(), после чего работа приложения завершается с выводом значения, с которым была вызвана паника, и стека вызовов. В Go есть встроенная функция panic(), которая принимает в качестве параметра значение любого типа и запускает процесс обработки аварийной ситуации, описанный выше.

```
package main

import "fmt"

func main()  {
  defer fmt.Println("main: defer")

  someFunc()
  fmt.Println("main: after someFunc call")
}

func someFunc()  {
  defer fmt.Println("someFunc: defer")

  panic("something get wrong")
}
```

В примере выше мы объявили отложенные вызовы в main() и someFunc(), после чего в функции someFunc() мы смоделировали возникновение аварийной ситуации вызовом panic(). В результате запуска этого кода мы увидим следующий вывод:

```
someFunc: defer
main: defer
panic: something get wrong
```

```
goroutine 1 [running]:
main.someFunc()
        /dev/go/src/rebrain/panic_2_7.go:13 +0x95
main.main()
        /dev/go/src/rebrain/panic_2_7.go:7 +0x96
exit status 2
```

В выводе программы видна последовательность отложенных вызовов, а в распечатанном стеке вызовов также указан номер строки в файле, на которой он произошел. Если паника — это штатная ситуация и не теряется контроль над управлением, значит, должен быть способ обрабатывать такие ситуации. Об этом поговорим далее в разделе.

Обработка критических ситуаций и восстановление

Обработка аварийной ситуации осуществляется в отложенных вызовах. Как упоминалось выше, функция panic() принимает на вход любое значение. В нашем примере это была всего лишь произвольная строка, но вместо нее могут быть любые данные, которые помогут разработчику однозначно понять, что произошло.

Определить, произошла паника или отложенный вызов был штатным, позволяет функция recover(). Вызов функции recover() в отложенном вызове останавливает последовательность обработки аварийной ситуации и возвращает значение, с которым была вызвана последняя случившаяся паника. В противном случае recover() вернет nil. Если функция recover() вызывается в любом другом месте, то она не останавливает обработку аварийной ситуации и тоже возвращает nil. Немного доработаем наш пример.


```
package main

import (
  "fmt"
)

func main() {
  err := someFunc()
  if err != nil {
     fmt.Println(err)
  }

  fmt.Println("after someFunc")
}

func someFunc() (err error) {
  defer func() {
     if panicErr := recover(); panicErr != nil {
```

```
            switch panicErr {
            case "regular error":
                err = fmt.Errorf("application error")
            default:
                panic("critical")
            }
        }
    }()

    panic("regular error")
}
```

Пример показывает, как можно обрабатывать панику. В функции someFunc() мы добавили возвращаемое значение типа error. Для создания новой ошибки применили функцию fmt.Errorf(). И вызвали панику с ожидаемым значением. Вывод работы этой программы будет таким:

```
application error
after someFunc
```

То есть, наш код в состоянии обработать какие-то ожидаемые аварийные ситуации и продолжить работу дальше. В противном случае в операторе switch предусмотрен вызов новой паники, которая уже инициирует возникновение и обработку новой аварийной ситуации. Но вызов паники в процессе восстановления паники является плохой практикой. Некоторые программисты, знакомые с механизмом исключений в других языках программирования, на этом моменте могли бы провести аналогию между ними и паникой. Но нет, паника и отложенный вызов - это не реализация try … catch и исключений, поэтому не стоит пытаться применять механизм аварийных ситуаций по такому назначению. Все потому что в момент возникновения аварийной ситуации не гарантируется ожидаемое состояние структур данных, например: могут быть необновленные значения переменных, соединение с базой данных может иметь не актуальный статус и т.д. Но в то же время гарантируется последовательность вызовов. Именно поэтому хорошей практикой является вызов паники только в случае возникновения каких-то действительно критических ситуаций, при которых дальнейшая работа приложения не может быть корректной. Например, в момент запуска приложение не смогло подключиться к базе данных или к сервису очередей — это правильный пример вызова паники, а ситуация, когда в запросе от пользователя пришли неправильные данные, все-таки является ожидаемой и может быть корректно обработана.

Полезные ссылки:

- Handling Panics in Go
- Обработка ошибок в Golang с помощью Panic, Defer и Recover

Задание:

1. Создайте в проекте module02 новую ветку 07_task.
2. Создайте новую директорию с файлом main.go.
3. Скачайте файл in.txt и скопируйте его в директорию data рядом с файлом main.go.
4. Напишите программу, которая считывает данные из файла, проверяет, что все поля заполнены и записывает считанные данные в файл data/data_out.txt в формате Row: %d\nName: %s\nAddress: %s\nCity: %s\n\n\n.
5. Если какое-то поле не заполнено, то программа должна вызвать панику, передав строку вида parse error: empty field on string %d.
6. Объявите необходимые отложенные вызовы.
7. Обработайте панику таким образом, чтобы после обработки на экран вывелось содержимое файла data_out.txt, которое было записано до возникновения паники.
8. Зафиксируйте изменения в ветке и отправьте их в удаленный репозиторий проекта.
9. В качестве ответа пришлите ссылку на merge request в ветку master вашего проекта ветки 07_task.