GO-04 06: Структуры и интерфейсы. Продвинутая обработка ошибок

Описание:

В этом задании, мы углубимся в работу с ошибками в Go и посмотрим на еще некоторые хитрые приемы для работы с ними. Для начала давайте вспомним как обычно мы работаем с ошибками на следующем примере

```
package main

import "errors"

func someFunc() error {
    return errors.New("some error")
}

func main() {
    if err := someFunc(); err != nil {
        panic(err)
    }
}
```

В этом примере, мы просто проверяем наличие ошибки и это вполне go way обработка, но иногда есть ситуации когда нам важно знать, что за ошибка нам вернулась а не просто знать о ее наличии, например.

```
package main

import (
      "errors"
      "log"
      "strconv"
)

var (
      errNotFound = errors.New("not found")
      values      = []string{"aaa", "bbb", "1"}
)
```

```
func searchAndConvert(expectedValue string) (int, error) {
	for _, v := range values {
		if v == expectedValue {
			convertedValue, err := strconv.Atoi(v)
			if err != nil {
				return 0, err
			}

			return convertedValue, nil
		}
	}

	return 0, errNotFound
}

func main() {
	res, err := searchAndConvert("2")
	if err != nil {
		if err == errNotFound {
			log.Println("value not found")
			return
		}

		log.Fatal(err)
	}

	log.Println(res)
}
```

Итак, давайте разберем пример по порядку. Во первых, сразу в импортах мы видим новый импортируемый пакет из стандартной библиотеки - errors, в котором есть множество полезных методов, для работы с ошибками. Ну и самый первый метод пакета, который мы в свами встречаем - это метод New.

```
errNotFound = errors.New("not found")
```

Этот метод, позволяет нам создавать собственные простые ошибки, в которые мы передаем обычное сообщение и дальше можем использовать эту ошибку как угодно (можем как возвращаемое значение из функции, можем как переменную, в общем как угодно). Метод New возвращает нам ошибку (тоесть то, что подчиняется стандартному go интерфейсу error).

Дальше в примере, мы возвращаем эту ошибку если в функции searchAndConvert не находим того, что ищем. С другой стороны, если мы нашли значение, которое ищем но не смогли привести его к int, тогда мы отбрасываем ошибку, которая приходит к нам из метода strconv.Atoi ну и что там могло пойти не так мы не знаем поэтому контролировать эту ошибку мы не в состоянии. Из этого следует, что когда мы обрабатываем ошибку из функции searchAndConvert у нас может случится два кейса, один из них нам не известен, а второй как раз таки известен это наша ошибка errNotFound, которую мы создали самостоятельно!
Так как же нам обработать тот кейс, который нам известен? Все очень просто! Раз пришедшая ошибка и наша созданная errNotFound имеют одинаковый тип, тогда мы можем попытаться их просто сравнить.

```
if err == errNotFound {
    log.Println("value not found")
    return
}
```

И такая запись, вполне корректно отработает. Теперь давайте чуть усложним нашу ошибку, ведь ошибкой может быть что угодно, что имплементирует интерфейс error, кстати этот простой интерфейс выглядит следующим образом.

```
type error interface {
    Error() string
}
```

Давайте перепишем нашу ошибку, теперь мы не будем пользоваться errors.New, а сделаем собственную структуру, которая будет имплементировать интерфейс error.

```
type errNotFound struct {
     message string
}

func (e *errNotFound) Error() string {
     return e.message
}

var (
     errNotFoundMessage = "not found"
     values             = []string{"aaa", "bbb", "1"}
)
```

Теперь в функции searchAndConvert мы изменим только одну строку в конце функции, которая меняет нашу переменную на инициализацию структуры.

```
func searchAndConvert(expectedValue string) (int, error) {
    // ... some code
```

```
    return 0, &errNotFound{message: errNotFoundMessage}
}
```

И сама функция при этом ругатся не должна потому что типы не перестали соответствовать (мы должны были вернуть из функции интерфейс error и наша структура errNotFound имплементирует этот интерфейс). Но при запуске такой программы, мы все таки встретим ошибку.

```
./main.go:39:10: type errNotFound is not an expression
```

Это случилось вот в этом куске нашей программы.

```
if err == errNotFound {
    log.Println("value not found")
    return
}
```

Потому что система типов не пытается в данном случае сравнить интерфейсы, она пытается сравнить error со структурой errNotFound и естественно такое сравнение в go не возможно, поэтому мы видим ошибку, но мы можем это исправить при помощи такого механизма как type assertion. Давайте попробуем исправить проверку.

```
if _, ok := err.(*errNotFound); ok {
    log.Println("value not found")
    return
}
```

Таким образом, мы проверяем что ошибка имеет тип errNotFound, причем не сам тип а ссылку на тип. После этого преобразования наша программа снова должна заработать корректно.

Но у нас есть еще один механизм и называется он wrapping, суть его состоит в том чтобы обернуть одну ошибку в другую и например добавить к ошибки дополнительное описание. Это можно сделать при помощи пакета fmt и его метода Errorf давайте попробуем.

```
func searchAndConvert(expectedValue string) (int, error) {
    // ... some code

    return 0, fmt.Errorf("searchAndConvert error: %w",
&errNotFound{message: errNotFoundMessage})
}
```

Здесь мы обогатили нашу ошибкой новой информацией (например о том из какой функции эта ошибка прилетела). Но теперь не сравнение не type assertion не будут проверять ошибку и наша программа снова поломана.

```
2009/11/10 23:00:00 some err: not found
```

Все потому что ошибка уже не имеет тип errNotFound, а снова имеет тип интерфейса error. Обычное сравнение также не подойдем (точнее будет работать не корректно) потому что это обернутая ошибка и сравнивать ее надо с такой же обернутой ошибкой. Есть несколько вариантов решения данной проблемы.
Первое решения состоит в том, что мы можем не только обернуть нашу ошибку, но также и извлечь обернутую ошибку. Сделать это можно через другой метод пакета errors, который называется Unwrap. Давайте попробуем.

```
if _, ok := errors.Unwrap(err).(*errNotFound); ok {
    log.Println("value not found")
    return
}
```

Таким образом мы извлекли ошибку errNotFound из ее обернутого варианта fmt.Errorf("searchAndConvert error: %w", &errNotFound{message: errNotFoundMessage}) и дальше по стандартному сценарию проверяем ошибку через type assertion.
Наверное может показаться, что мы делаем очень много телодвижений для обработки ошибок и вы будете правы, но с этим приходилось мирится до появления пакета errors в go 1.13, теперь помимо упомянутых методов New и Unwrap у нас еще есть два крутых метода проверки метод Is и метод As.
Метод As делает тоже самое что и мы делаем руками, а именно type assertion над ошибками, но также он отлично работает с обернутыми ошибками, то есть этот метод пытается сначала привести к типу, а вот если у него это не получается, тогда он пытается сделать Unwrap и если у него это получилось тогда снова пробует делать type assertion и так пока либо type assertion не даст положительного результата, либо пока не дойдет до изначальной ошибки. Интерфейс этого метода выглядит так

```
func As(err error, target interface{}) bool
```

Давайте исправим нашу программу с использованием метода As.

```
if err != nil {
    var enf *errNotFound
    if errors.As(err, &enf) {
        log.Println("value not found")
        return
    }

    log.Fatal(err)
}
```

Мы сначала инициализировали переменную типа errNotFound для того чтобы иметь тип с которым нужно сравнивать (в нашем случае тип - это ссылка на нашу структуру

errNotFound), а затем мы просто передаем полученную ошибку и ссылку на тип с которым ошибку надо сравнить. Таким образом значительно упрощаем обработку ошибок.
Метод Is работает тогда, когда нам не нужно делать type assertion, то есть в обычных ситуациях, когда мы могли бы воспользоваться обычным сравнением (например if err == someErrType), мы вместо этого можем воспользоваться методом Is. Плюсом этого метода несомненно является тот факт, что он как и метод As умеет разворачивать ошибки, которые были обернуты. Интерфейс метода точно такой же как и у метода As.

```
func Is(err error, target interface{}) bool
```

Так же до go1.13 был отдельный пакетик errors pkg который делел все, что сейчас делает стандартный пакет errors, поэтому в старых проектах можно встретить данный пакет в обиходе.

Полезные ссылки:

- Advanced error handling in go
- Golang — изящная обработка ошибок
- Creating Custom Errors in Go
- Golang Error Handling — Best Practice in 2020

Задание:

1. Перейдите в проект с заданиями второго модуля module02 и создайте там новую ветку 04_06_task из ветки 08_task.
2. В последнем задании 2-го модуля нужно было обработать ошибку io.EOF теперь попробуйте обработать ее с помощью пакета errors.
    - Основная логика должна быть вынесена в отдельную функцию, которая возвращает результат и ошибку. Это нужно для того чтобы в случае ошибки мы могли проверить ее тип.
3. Создайте переменную limit укажите ее в произвольное значение и если counter строк превысит лимит вы должны выбросить ошибку.
4. Ошибка при превышении лимита, должна быть структурой, которая имплементирует интерфейс error и содержит необходимые свойства для вывода: fmt.Sprintf("%s, limit: %d, last string: %s", e.message, e.limit, e.lastString)
5. В функции main, обработайте кейс с превышением лимита, при помощи пакета errors. Кейс не должен создавать панику, лишь печатать сообщение с ошибкой fmt.Println("string count exceed limit, please read another file =) err: ", err.Error()) в случае, если получена наша имплементация интерфейса error.
6. Зафиксируйте изменения в ветке и отправьте их в удаленный репозиторий проекта.
7. В качестве ответа пришлите ссылку на merge request в ветку master вашего проекта ветки 04_06_task.