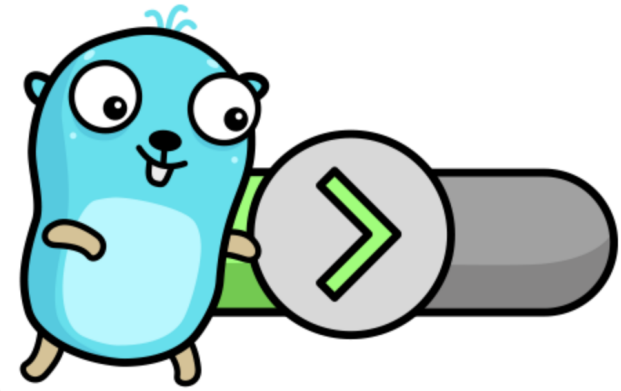
Концепция ошибок в Go

В этом уроке мы:

- познакомимся с интерфейсом `error`;
- сделаем первые шаги по созданию собственных ошибок;
- немного поговорим об идиоматичной обработке чужих ошибок.

Не переживайте, что в этом и следующем уроке мы не будем создавать ошибки через стандартную библиотеку – всё впереди!



type error interface

В Go для ошибок есть специальный тип – интерфейс `error`.

Интерфейс error – это встроенный (built-in) интерфейс с одним единственным методом `Error() string`. Задача этого метода – вывести сообщение об ошибке, чтобы конечный пользователь мог понять, что же пошло не так:

```
// The error built-in interface type is the conventional interface
for
// representing an error condition, with the nil value representing
no error.
type error interface {
    Error() string
```

```
}
```

Рассмотрим процесс создания и использования собственных ошибок на примере байтового буфера.

У нас есть некий тип `ByteBuffer` (исходник примера):

```
type ByteBuffer struct {
    // buffer представляет собой непосредственно буфер: содержит
какие-то данные.
    buffer []byte
    // offset представляет собой смещение, указывающее на первый
непрочитанный байт.
    offset int

}
```

Из него мы будем делать подобие правильного буфера с возможностью записи и чтения. Пока что он умеет вот так:

```
func (b *ByteBuffer) Write(p []byte) int {
    b.buffer = append(b.buffer, p...)
    return len(p)
}

func (b *ByteBuffer) Read(p []byte) int {
    if b.offset >= len(b.buffer) {
        return 0
    }

    n := copy(p, b.buffer[b.offset:])
    b.offset += n
    return n

}
```

Методы `Write` и `Read` возвращают единственный аргумент – число записанных или прочитанных байт. В случае чего ошибки тоже придётся передавать через этот аргумент. Пока что сильно смахивает на Си и его **in-band error indication**, только у нас нет глобальной переменной `errno`.

Но мы всё-таки хотим сообщать вызывающей стороне о том, что что-то пошло не так, и при этом **in-band** ошибки нас не устраивают. Как мы прекрасно знаем, Go поддерживает возврат из функции нескольких значений.

Без труда добавляем в сигнатуру методов `Write` и `Read` возвращаемое значение типа `error`:

```
func (b *ByteBuffer) Write(p []byte) (int, error) {
    // ...
}

func (b *ByteBuffer) Read(p []byte) (int, error) {
    // ...

}
```

Что может пойти не так, когда мы пишем в буфер или читаем из него? Например, при записи мы не хотим, чтобы размер буфера превышал заданное максимальное значение. А во время чтения пользователь может полностью вычитать буфер так, что в нём не останется данных для возврата — и об этом тоже хочется явно сообщить наружу.

Заведем два типа `MaxSizeExceededError` и `EndOfBufferError`, реализующих интерфейс `error`:

```
const bufferMaxSize = 1024

type MaxSizeExceededError struct {
    desiredLen int
}

func (e *MaxSizeExceededError) Error() string {
    return fmt.Sprintf("buffer max size exceeded: %d > %d",
e.desiredLen, bufferMaxSize)

}

type EndOfBufferError struct{}

func (b *EndOfBufferError) Error() string {
    return "end of buffer"
```

```
}
```

И доработаем методы `Write` и `Read` так, чтобы они возвращали необходимые ошибки (исходник примера) :

```
func (b *ByteBuffer) Write(p []byte) (int, error) {
    if len(b.buffer)+len(p) > bufferMaxSize {
        return 0, &MaxSizeExceededError{desiredLen: len(b.buffer) +
len(p)}
    }

    b.buffer = append(b.buffer, p...)
    return len(p), nil
}

func (b *ByteBuffer) Read(p []byte) (int, error) {
    if b.offset >= len(b.buffer) {
        return 0, new(EndOfBufferError)
    }

    n := copy(p, b.buffer[b.offset:])
    b.offset += n
    return n, nil

}
```

Таким образом, мы научили буфер возвращать ошибки и предоставили пользователю нашего типа возможность реагировать на них (обрабатывать, логировать и т.д.).

Итого мы узнали, что ошибки в Go представлены специальным интерфейсом `error`.

Для того, чтобы сообщить о том, что конкретно пошло не так, можно завести тип, реализующий интерфейс `error`, и пользоваться экземплярами этого типа.

Какие ещё есть варианты создания ошибок и как их обрабатывать по другую сторону баррикад (со стороны пользователя функции) – мы узнаем в следующих уроках.

Задача "Побайтовое чтение и запись"

Ссылка на заготовку.

Необходимо сделать так, чтобы тип `*ByteBuffer` из предыдущего шага реализовывал интерфейсы `io.ByteWriter` и `io.ByteReader`:

```
const bufferMaxSize = 1024

type ByteBuffer struct {
    // buffer представляет собой непосредственно буфер: содержит
какие-то данные.
    buffer []byte
    // offset представляет собой смещение, указывающее на первый
непрочитанный байт.
    offset int

}
```

```
package io

// ByteReader is the interface that wraps the ReadByte method.
//
// ReadByte reads and returns the next byte from the input or
// any error encountered. If ReadByte returns an error, no input
// byte was consumed, and the returned byte value is undefined.
type ByteReader interface {
    ReadByte() (byte, error)
}

// ByteWriter is the interface that wraps the WriteByte method.
type ByteWriter interface {
    WriteByte(c byte) error
}
```

- Метод `WriteByte` должен возвращать ошибку `*MaxSizeExceededError` при попытке записи в буфер, если в нём уже `bufferMaxSize` байт.

- Метод `ReadByte` должен возвращать ошибку `*EndOfBufferError` при попытке чтения из буфера, если ранее буфер уже был вычитан полностью.

Задача "Аллокатор на Go"

Ссылка на заготовку.

Системный программист решил зарефакторить свой аллокатор и переписать его на Go.

Алгоритм работы функции остаётся прежним:

- функция принимает ID пользователя и количество требуемых байт памяти;
- если пользователь не является администратором (ID не совпадает с `Admin`), то функция возвращает ошибку с текстом "**operation not permitted**".
- если пользователь запрашивает менее чем `MinMemoryBlock` байтов, то функция возвращает ошибку с текстом "**numerical argument out of domain of func**".
- в остальных случаях функция возвращает слайс байт заданного размера.

Идиоматичная проверка ошибок

Обилие конструкций `if err != nil` в Go – нормальное явление.

С одной стороны количество кода увеличивается, с другой стороны – всё прозрачно и понятно.

Так что сишный код из этого урока

```
errno_t major_func() // Сигнатура функции даёт понять, что возвращается ошибка.
{
    errno_t err = 0;

    if (err = minor_func1())
        return err;

    int val = 0;
    if (err = minor_func2(&val))
        return err;

    if (err = minor_func3(val))
        return err;

    return 0;

}
```

легким движением руки превращается в идеологически правильный Go-код:

```
func majorFunc() error { // Сигнатура функции даёт понять, что возвращается ошибка.
    var err error

    if err = minorFunc1(); err != nil {
        return err
    }

    val, err := minorFunc2();
    if err != nil {
        return err
    }

    if err = minorFunc3(val); err != nil {
        return err
    }

    return nil;

}
```

Indent Error Flow

В соответствии с [лучшими практиками] старайтесь бороться с излишними `else`-конструкциями.

Например, так делать не стоит:

```
if err != nil {
    // Обрабатываем ошибку.
} else {
    // Штатный флоу.

}
```

А так стоит:

```
if err != nil {
    // Обрабатываем ошибку.
    return err
}

// Штатный флоу.
```

```
x, err := doSomething()
if err != nil {
    // Обрабатываем ошибку.
    return err
}

// Штатный флоу, где используем x.
```

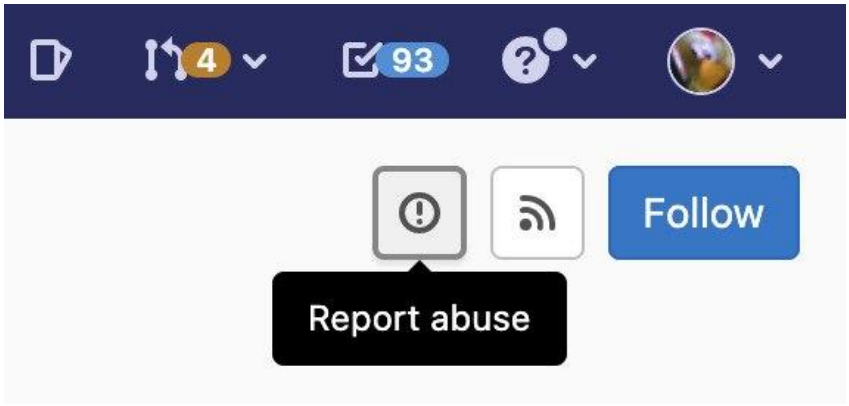
Так тоже лучше не делать:

```
if x, err := doSomething(); err != nil {
    // Обрабатываем ошибку.
} else {
    // Штатный флоу, где используем x.

}
```

А так делать следует:

```
x, err := doSomething()
if err != nil {
    // Обрабатываем ошибку.
    return err
}


// Штатный флоу, где используем x.
```

Тест "Токсичный ревьюер"



Выберите варианты, корректные **с точки зрения построения error flow** (на основе рекомендаций из предыдущих шагов).

A:

```
if err != nil {
    fmt.Println(err.Error)
    return err
} else {
    fmt.Println("OK")

}
```

B:

```
info, err := r.client.GetInfo(ctx, request)
if err != nil {
    return 0, fmt.Errorf("error getting info: %v", err)
}


return info.ID, nil
```

C:

```
info, err := r.client.GetInfo(ctx, request)
if err == nil {
    return 0, fmt.Errorf("error getting info: %v", err)
}
```

```
return info.ID, nil
```

D:

```
info, err := r.client.GetInfo(ctx, request)
if err != nil {
    return 0, fmt.Errorf("error getting info: %v", err)
} else {
    return info.ID, nil

}
```

E:

```
if info, err := r.client.GetInfo(ctx, request); err != nil {
    return 0, fmt.Errorf("error getting info: %v", err)
} else {
    return info.ID, nil

}
```

F:

```
rows, err := db.QueryContext(ctx, query, args...)
if err != nil {
    logger.WithError(err).Error("query error")
    return err
}

defer rows.Close()
```

G:

```
rows, err := db.QueryContext(ctx, query, args...)
if err != nil {
    logger.WithError(err).Error("query error")
}

defer rows.Close()
```

Меняем местами

Приём, которым поделился с нами @Anonymous 34072085.

Гошный код пестрит проверками вида `err != nil`, поэтому, если нам нужно, наоборот, проверить на равенство `nil`, то эта проверка просто теряется в серой массе других проверок:

```
if err != nil {
    // ...
}

if errUser == nil {
   // ...
}

if err != nil {
    // ...

}
```

Чтобы это исправить, следует поменять переменную и `nil` местами:

```
if err != nil {
    // ...
}

if nil == errUser {
   // ...
}

if err != nil {
    // ...

}
```

Это решает, как минимум, две задачи (при чтении кода):

- *Подтверждает намерения автора кода – он не ошибся и действительно хотел поставить `==`, а не `!=`.*

- *Обращает внимание читателя кода ("режет глаз"), что проверяем равенство на `nil`. Иначе, опять же, на автоматизме можно подумать, что это идиоматичная проверка на неравенство.*

В общем, очень люблю и уважаю этот приём :)

Подведём итоги

Интерфейс в Go – это набор методов с определённой сигнатурой. Тип неявно реализует интерфейс, если у него есть все методы интерфейса. При этом имена методов в интерфейсе не должны повторяться, а сам интерфейс может быть пустым – такому интерфейсу (`interface{}`) удовлетворяет любой тип.

Ошибки в Go – это значения типов, реализующих встроенный интерфейс `error`:

```
type error interface {
    Error() string

}
```

Соответственно ошибки следует воспринимать как обычные переменные со всеми вытекающими из этого последствиями.

Желательно не просто проверять наличие ошибок, а осознанно их обрабатывать (хотя бы на самом верхнем уровне).

В большинстве случаев игнорировать ошибки нельзя.