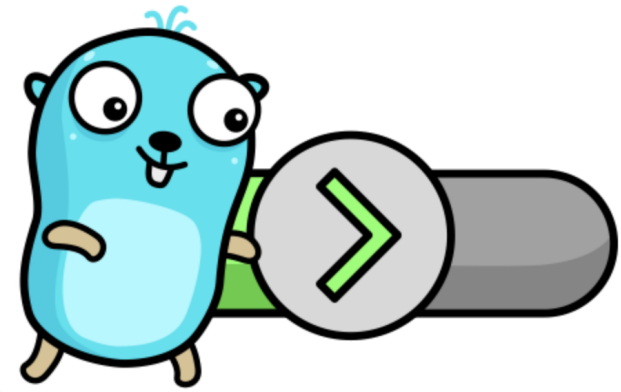
Логирование ошибок

В этом уроке мы узнаем, где обычно принято логировать ошибки, с помощью чего и как.

Рассмотрим несколько подходов, но конечный выбор, как всегда, за вами!



Или логируем или возвращаем ошибку

Either log or return an error.

Популярное правило, которое можно встретить в книгах, статьях и рекомендациях от Go разработчиков.

Взглянем на пример, в котором автор и логирует ошибку и пробрасывает её дальше:

```
// ...

import (
    "context"
    "fmt"
    "log"
    "time"
    // ...
)

const defaultTTL = time.Minute
```

```
var ErrKeyNotExist = errors.New("key doesn't exist")

func (c *Client) setTimeoutForKey(ctx context.Context, key string)
error {
    res, err = c.rdb.Expire(ctx, key, defaultTTL).Result()
    if err != nil {
        log.Printf("cannot set ttl for key %q: %v", key, err)
        return fmt.Errorf("set ttl for key %q: %v", key, err)
    }

    if res == "0" {
        log.Printf("key %q doesn't exist", key)
        return fmt.Errorf("key %q: %w", key, ErrKeyNotExist)
    }

    return nil
}
```

Считается, что так делать не очень хорошо по следующим причинам:

- Обычно ошибка пробрасывается наверх (неважно, обёрнутая или нет) или обрабатывается в одном месте, завершающем **error chain.** В данном случае мы обрабатываем ошибку через логирование, поэтому странно пробрасывать её наверх, ожидая дополнительных обработок. Более того после логирования ошибка перестаёт быть ошибкой, ведь мы уже её обработали! Т.е. по сути ожидается один из двух вариантов:

```
// ...
if err != nil {
    return fmt.Errorf("cannot set ttl for key %q: %v", key, err)
}
// ...
```

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

```
    handleErr(err) // == log.Printf(...)
    return nil     // Опционально.
}
// ...
```

- Явная дупликация кода в сообщении лога и во врапинге ошибки.

- Дупликация (в лучшем случае, в худшем – увеличение в N раз) логов, потому что выше в какой-нибудь мидлваре ошибка наверняка залогируется:

```
20:19:27 [ERROR] redis/client.go:50          key "unknown" doesn't exist
```

```
20:19:27 [ERROR] api/http/middlewares.go:10 handle event: set timeout: key "unknown": key doesn't exist
```

- Неочевидность – когда логировать ошибку в цепочке ошибок, а когда нет?

Поэтому рекомендуется не логировать ошибку вместе с её возвратом, а делать это только в месте её обработки (обычно на самом верху стека вызовов).

И всё это хорошо, пока мы не начинаем хотеть **структурированного логирования**.

Структурированное логирование

Structured logging is the practice of implementing a consistent, predetermined message format for application logs

that allows them to be treated as data sets rather than text. (тыц)

Нередко форматом для структурированного логирования выбирают JSON: такие логи удобно обрабатывать как подручными инструментами вроде jq, так и агрегировать сторонними системами вроде grafana/loki, ELK и т.д.

Стандартный пакет `log` не умеет в подобное логирование, он лишь умеет печатать форматированную строку на экран подобно `fmt.Printf`:

```
package log

// Printf calls Output to print to the standard logger.
// Arguments are handled in the manner of fmt.Printf.

func Printf(format string, v ...interface{})
```

```
20:19:27 [ERROR] api/http/middlewares.go:10 handle event: set
timeout: key "unknown": key doesn't exist
```

Понятно, что из подобных логов неудобно доставать данные (привет регуляркам), а поиск по ним будет куда медленнее, чем при поиске по конкретным индексируемым полям.

Лог выше в структурированном виде мог бы выглядеть, например, следующим образом:

```
{
    "time": "20:19:27",
    "level": "ERROR",
    "method": "api/http/middlewares.go:10",
    "redis_key": "unknown",
    "error": "handle event: set timeout: key doesn't exist",

}
```

Тест "Сопоставьте логер и его характеристики"

Обращаем ваше внимание, что логер может удовлетворять как одному, так и всем критериям разом.

Отметьте верные ячейки

	Позволяет создавать структурированные логи	Позволяет создавать "обычные" логи
log		
github.com/uber-go/zap		
github.com/francoispqt/onelog		
github.com/Sirupsen/logrus		

Потеря контекста ошибки

При использовании структурированных логов данный кусок кода

```
func (c *Client) setTimeoutForKey(ctx context.Context, key string)
error {
    res, err = c.rdb.Expire(ctx, key, defaultTTL).Result()
    if err != nil {
        log.Printf("cannot set ttl for key %q: %v", key, err)
        return fmt.Errorf("set ttl for key %q: %v", key, err)
    }

    if res == "0" {
        log.Printf("key %q doesn't exist", key)
        return fmt.Errorf("key %q: %w", key, ErrKeyNotExist)
    }

    return nil
}
```

выглядел бы скорее всего следующим образом

```
func (c *Client) setTimeoutForKey(ctx context.Context, key string)
error {
    res, err = c.rdb.Expire(ctx, key, defaultTTL).Result()
    if err != nil {
        c.log.Error("cannot set ttl",
            field.String("redis_key", key),
            field.Error(err),
        )
        return fmt.Errorf("set ttl: %v", err)
    }

    if res == "0" {
        c.log.Error("key doesn't exist",
            field.String("redis_key", key))
        return ErrKeyNotExist
    }

    return nil

}
```

или даже так

```
func (c *Client) setTimeoutForKey(ctx context.Context, key string)
error {
    logger := c.log.With(field.String("redis_key", key))

    res, err = c.rdb.Expire(ctx, key, defaultTTL).Result()
    if err != nil {
        logger.Error("cannot set ttl", field.Error(err))
        return fmt.Errorf("set ttl: %v", err)
    }

    if res == "0" {
        logger.Error("key doesn't exist")
        return ErrKeyNotExist
    }

    return nil

}
```

Но в начале данного урока мы выяснили, что логировать по месту не приветствуется.

И в таком случае мы получаем **проблему потери контекста ошибки**:

```
// Функция находится в N вызовах от setTimeoutForKey.
func middleware() {
    if err := handle(); err != nil {
        // Мы потеряли контекст ошибки в виде поля "redis_key".
        logger.Error("handle event", field.Error(err))
    }
}

func (c *Client) setTimeoutForKey(ctx context.Context, key string)
error {
    res, err = c.rdb.Expire(ctx, key, defaultTTL).Result()
    if err != nil {
        return fmt.Errorf("set ttl: %v", err) // Убрали логирование.
    }

    if res == "0" {
        return ErrKeyNotExist
    }

    return nil

}
```

В примере выше мы потеряли только `"redis_key"`, но на самом деле логер по пути может обогащаться б**о**льшим количеством полей, недоступным для верхнеуровневого логирования:

```
func main() {
    if err := a(logger); err != nil {
        logger.Error("cannot do a", field.Error(err)) // Потеряли все
поля ниже.
    }
}

func a(logger ILogger) error {
    // ...
    return b(logger.With(field.String("user_id", uid))
}

func b(logger ILogger) error {
```

```
    // ...
    return c(logger.With(field.String("file", fname))
}

func c(logger ILogger) error {
    // ...
    return d(logger.With(field.Int("pid", pid)))
}

func d(logger ILogger) error {
    // ...
    if err != nil {
        // Залогируем вместе с "user_id", "file" и "pid".
        logger.Error("operation", field.Error(err))
    }
    return err
}
```

Какие есть варианты?

- Забить на "лучшие" практики и логировать по месту с наибольшим для ошибки контекстом. При этом закрываем глаза на дублирование логов (выбираем из двух зол меньшее: лучше больше логов, чем меньше или чем логи, из которых сложнее понять, что происходит). Сохраняем баланс между возвратом ошибки и её логированием: скорее всего логирование будет происходить в месте возникновения новой ошибки (возврат sentinel, вызов базульки, поход в другой сервис и т.д.) и в мидлварах на входе приложения.

- Писать враперы над ошибками, которые позволят обогащать ошибки контекстом, а на самом верху работать с ними через `errors.As` или вспомогательные функции, позволяющие этот контекст доставать и добавлять к логу.

Какой стул выбрать – решать вам.

А если уже выбрали, то **делитесь опытом в комментариях**!

Задача "Error Context"

Ссылка на заготовку.

Вам необходимо реализовать две функции, позволяющие обогащать ошибку произвольным контекстом и доставать его из неё:

```
package errctx

type Fields map[string]any

func AppendTo(err error, fields Fields) error { /* ... */ }

func From(err error) Fields { /* ... */ }
```

Они пригодятся, если вы, например, хотите логировать всё в одном месте и при этом не желаете терять контекст ошибки:

```
import "github.com/sirupsen/logrus"

func main() {
    l := logrus.New()
    l.Formatter = new(logrus.JSONFormatter)

    if err := foo(); err != nil {
```

```
        // {"error":"EOF","level":"error","msg":"cannot do
foo","time":"...","uid":"f51be77e-f60e-11eb-b47f"}
        l.WithFields(errctx.From(err)).WithError(err).Error("cannot
do foo")
    }
}

func foo() error {
    return errctx.AppendTo(io.EOF, errctx.Fields{
        "uid": "f51be77e-f60e-11eb-b47f",
    })

}
```