github.com/cockroachdb/errors (часть 2)

Этот урок посвящён таким интересным фичам **cockroachdb/errors**, как комбинирование ошибок и поддержка передачи ошибок по сети.

Приступим!

errors.WithSecondaryError, errors.CombineErrors

Иногда возникает ситуация, когда при обработке ошибки случается другая ошибка:

```
func foo() error {
    // ...

    if err := somepkg.DoSomething(); err != nil {
        if err := tx.Rollback(ctx); err != nil { // Обратите внимание на `:=`.
            // ...
        }
        return fmt.Errorf("do something: %w", err)
    }

}
```

При написании подобного кода возникают следующие вопросы:

1. Какая ошибка "главнее"? Наверное, первая, потому что из-за неё всё началось. Её и будем возвращать.
2. А что делать со второй ошибкой? Обычно:
    - ничего не делают;
    - или хотя бы логируют её;
    - или ищут способ вернуть обе ошибки (придумывая различные слайсы ошибок и т.д.).

cocroachdb/errors предлагает ещё один вариант (исходник примера): "прицепить" второстепенную ошибку к первой так, чтобы она отображалась при её логировании, но не влияла на анализ возникновения причины ошибок (грубо говоря, не реагировала на `errors.Is` и подобные методы):

```
package main

import (
    "context"
    "fmt"
    "io"
```

```
    "github.com/cockroachdb/errors"
)

func main() {
    err := io.EOF
    err = errors.WithSecondaryError(err, context.Canceled)

    fmt.Printf("%+v", err)
    /*
        EOF
        (1) secondary error attachment
          | context canceled
          | (1) context canceled
          | Error types: (1) *errors.errorString
        Wraps: (2) EOF
        Error types: (1) *secondary.withSecondaryError (2)
*errors.errorString
    */

    fmt.Println(errors.Is(err, io.EOF))            // true
    fmt.Println(errors.Is(err, context.Canceled)) // false
}
```

Как мы видим из примера выше, для реализации подобного нам поможет errors.WithSecondaryError и использующая её errors.CombineErrors:

```
package secondary

// WithSecondaryError enhances the error given as first argument with
// an annotation that carries the error given as second argument.
The
// second error does not participate in cause analysis (Is, etc) and
// is only revealed when printing out the error or collecting safe
// (PII-free) details for reporting.
//
// If additionalErr is nil, the first error is returned as-is.
//
// Tip: consider using CombineErrors() below in the general case.
func WithSecondaryError(err error, additionalErr error) error {
    if err == nil || additionalErr == nil {
```

```
        return err
    }
    return &withSecondaryError{cause: err, secondaryError:
additionalErr}

}

package secondary

// CombineErrors returns err, or, if err is nil, otherErr.
// if err is non-nil, otherErr is attached as secondary error.
// See the documentation of `WithSecondaryError()` for details.
func CombineErrors(err error, otherErr error) error {
    if err == nil {
        return otherErr
    }
    return WithSecondaryError(err, otherErr)

}
```

Пример использования (исходник):

```
func GetPage(ctx context.Context, url string) (data []byte, err
error) {
    // ...

    response, err := http.DefaultClient.Do(req)
    if err != nil {
        return nil, errors.WithMessage(err, "do request")
    }
    defer func() {
        // Дополняем основную ошибку второстепенной, если она
возникнет.
        // При этом, если есть только второстепенная ошибка, то она и
будет возвращена.
        //
        // ВАЖНО: использование именованного возвращаемого значения.
        err = errors.CombineErrors(err, response.Body.Close())
    }()

    // ...

}
```

- Если до `cleanup`-операции случается ошибка, то ошибка от операции будет присутствовать в сообщении первой.
- Если же до `cleanup`-операции ошибки не было, то ошибка от операции станет первостепенной и будет возвращена как полноценная ошибка.

Резюме

На вопрос, что делать с второстепенными ошибками, **cockroachdb/errors** отвечает так: возвращать, если основной ошибки нет, иначе прицеплять её к основной, чтобы хотя бы залогировать выше.

Подход интересный, но опять же – если нет желания затаскивать к себе целую библиотеку, то можно ограничиться реализацией только этих функций.

Задача "Комбинируем ошибки"

Ссылка на заготовку.

Вам необходимо реализовать функцию `Combine`:

```
// Combine "прицепляет" ошибки other к err так, что они начинают фигурировать при выводе
// её на экран через спецификатор `%+v`. Если err является nil, то первостепенной ошибкой
// становится первая из ошибок other.

func Combine(err error, other ...error) error
```

Как мы видим, это прокаченная версия `CombineErrors` из **github.com/cockorachdb/errors**.

Формат вывода на экран комбинированной ошибки следующий:

```
err := Combine(os.ErrExist,
    errors.New("ah shit, here we go again"),
    errors.New("another yet error"),
```

```
)
fmt.Println(err)
fmt.Println()
fmt.Printf("%+v", err)

/*
file already exists

file already exists
  - ah shit, here we go again
  - another yet error

*/
```

Убедитесь, что ваша реализация проходит тесты.

Go errors with network portability

Вообще складывается ощущение, что это одна из главных фичей библиотеки **cockroachdb/errors**, т.к. её реклама вынесена в самое начало описания либы:

> ### cockroachdb/errors: Go errors with network portability
>
> This library aims to be used as a drop-in replacement to `github.com/pkg/errors` and Go's standard `errors` package. It also provides *network portability* of error objects, in ways suitable for distributed systems with mixed-version software compatibility.

Вот оно, спасение всех и вся – казалось бы, используя эту библиотеку, можно легко и непринужденно передавать ошибки по сети, а библиотека сама красиво их сконвертирует в нативные Go-ошибки, которые уже можно вертеть разными `errors.Is`, `errors.As` и т.д.

Однако ~~как это обычно бывает~~ реальность не такая радужная. Судя по тестам "полноценно" данная фича работает только для gRPC, оставляя HTTP за бортом.

Чтобы разобраться в работе фичи, давайте рассмотрим тест на этот самый "network portability" для gRPC из самой библиотеки. Для удобства мы стянули его в наш репозиторий.

Там лежит ряд файлов, поглядим в которые в следующем порядке:

- echoer.proto;
- server_test.go;
- main_test.go;
- client_test.go.

echoer.proto

Собственно, здесь объявлен некий сервис `Echoer` с единственным методом `Echo`:

```
syntax = "proto3";

package grpc;

service Echoer {
  rpc Echo (EchoRequest) returns (EchoReply) {}
}

message EchoRequest {
  string text = 1;
}

message EchoReply {
  string reply = 1;
}
```

server_test.go

Приближаемся к интересному. В этом файле – имплементация сервиса выше, которая в зависимости от поступившей команды в `EchoRequest` возвращает или не возвращает какую-то ошибку:

```go
package cockroachgrpc

import (
    "context"

    "github.com/cockroachdb/errors"
    "github.com/cockroachdb/errors/grpc/status" // Обратите внимание, что это не гугловский grpc/status!
    "google.golang.org/grpc/codes"
)

var (
    ErrCantEcho = errors.New("unable to echo")
    ErrTooLong  = errors.New("text is too long")
    ErrInternal = errors.New("internal error")
)

type EchoServer struct{}

func (srv *EchoServer) Echo(ctx context.Context, req *EchoRequest) (*EchoReply, error) {
    msg := req.Text
    switch {
    case msg == "noecho":
        return nil, ErrCantEcho
    case len(msg) > 10:
        return nil, errors.WithMessage(ErrTooLong, msg+" is too long")
    case msg == "reverse":
        return nil, status.Error(codes.Unimplemented, "reverse is not implemented")
    case msg == "internal":
        return nil, status.WrapErr(codes.Internal, "there was a problem", ErrInternal)
    }
    return &EchoReply{
        Reply: "echoing: " + msg,
    }, nil

}
```

main_test.go

В этом файле – каноничный пример использования функции `TestMain`, в которой инициализируется и запускается сервер, а также инициализируется клиент к нему.

client_test.go

Наконец-то сами тесты, которые проверяют, что ошибки которые вернул сервер, действительно те самые Go-ошибки, которые мы ждем:

```
func TestGrpc(t *testing.T) {
    // ...

    // A sentinel error should be detectable across grpc boundaries.
    _, err = Client.Echo(context.Background(), &EchoRequest{Text:
"noecho"})
    tt.Assert(err != nil)
    tt.Assert(errors.Is(err, ErrCantEcho))
    tt.Assert(status.Code(err) == codes.Unknown)

    // A wrapped error should be unwrappable after crossing grpc
boundaries
    _, err = Client.Echo(context.Background(), &EchoRequest{Text:
"really_long_message"})
    tt.Assert(err != nil)
    tt.Assert(err.Error() == "really_long_message is too long: text
is too long")
    tt.Assert(errors.Is(err, ErrTooLong))
    tt.Assert(errors.UnwrapAll(err).Error() == "text is too long")

    // ...

}
```

При этом если распечатать одну из ошибок, мы увидим стек, который прикреплялся к ней ещё на стороне сервера:

```
client_test.go:52: spv:
there was a problem: internal error!
(1) gRPC code: Internal
Wraps: (2)
  | (opaque error wrapper)
```

```
  | type name:
github.com/cockroachdb/errors/withstack/*withstack.withStack
  | reportable 0:
  |
  | 04-non-standard-modules/cockroach-grpc.(*EchoServer).Echo
  |     /cockroach-grpc/server_test.go:29
  | 04-non-standard-modules/cockroach-grpc._Echoer_Echo_Handler.func1
  |     /cockroach-grpc/echoer.pb.go:192
  |
github.com/cockroachdb/errors/grpc/middleware.UnaryServerInterceptor
  |
/Users/anthony/golang_workspace/pkg/mod/github.com/cockroachdb/errors
@v1.8.5/grpc/middleware/server.go:19
  | 04-non-standard-modules/cockroach-grpc._Echoer_Echo_Handler
  |     /cockroach-grpc/echoer.pb.go:194
  | google.golang.org/grpc.(*Server).processUnaryRPC
  |
/Users/anthony/golang_workspace/pkg/mod/google.golang.org/grpc@v1.38.
0/server.go:1286
  | google.golang.org/grpc.(*Server).handleStream
  |
/Users/anthony/golang_workspace/pkg/mod/google.golang.org/grpc@v1.38.
0/server.go:1609
  | google.golang.org/grpc.(*Server).serveStreams.func1.2
  |
/Users/anthony/golang_workspace/pkg/mod/google.golang.org/grpc@v1.38.
0/server.go:934
  | runtime.goexit
  |     /usr/local/go/src/runtime/asm_arm64.s:1130
Wraps: (3) there was a problem
Wraps: (4)
  | (opaque error wrapper)
  | type name:
github.com/cockroachdb/errors/withstack/*withstack.withStack
  | reportable 0:
  |
  | 04-non-standard-modules/cockroach-grpc.init
  |     /cockroach-grpc/server_test.go:14
  | runtime.doInit
  |     /usr/local/go/src/runtime/proc.go:6309
  | runtime.doInit
  |     /usr/local/go/src/runtime/proc.go:6286
  | runtime.main
  |     /usr/local/go/src/runtime/proc.go:208
  | runtime.goexit
```

```
  |     /usr/local/go/src/runtime/asm_arm64.s:1130
Wraps: (5) internal error!

Error types: (1) *extgrpc.withGrpcCode (2) *errbase.opaqueWrapper (3)
*errutil.withPrefix (4) *errbase.opaqueWrapper (5) *errutil.leafError
```

Настоятельно рекомендуем поиграться со всем этим делом самостоятельно!

Промежуточный вывод

Мы поняли, что нас не обманывают, когда заявляют о "network portability" для gRPC – тесты честно проверяют, что с ошибками, прошедшими через сеть, можно работать как и с теми, что находятся в границах вашего приложения: сохраняются все детали, контекст, стек и т.д.

Остается главный вопрос: ~~нахрена это всё нужно~~ а как это всё работает "из коробки"?

Go errors with network portability: оставшиеся пазлы

На предыдущем шаге у внимательного читателя должен был возникнуть вопрос – а где собственно код, который пакует ошибки на серверной стороне и распаковывает на клиентской?

Ответ на этот вопрос лежит в тесте в cockroach-grpc/middleware, где реализованы интерсепторы, которые собственно используются сервером и клиентом:

```
// cockroach-grpc/main_test.go
package cockroachgrpc

func TestMain(m *testing.M) {
    // ...
    // Серверная middleware.
    grpcServer :=
grpc.NewServer(grpc.UnaryInterceptor(middleware.UnaryServerIntercepto
r))


    // ...
    dialOpts := []grpc.DialOption{
        grpc.WithDialer(func(target string, d time.Duration)
(net.Conn, error) { return lis.Dial("", "") }),
        grpc.WithInsecure(),
        grpc.WithUnaryInterceptor(middleware.UnaryClientInterceptor),
// Клиентская middleware.
    }
    clientConn, err := grpc.Dial("", dialOpts...)
    // ...

}
```

Наличие этих интерсепторов даёт нам понимание, что на самом деле не особо-то из коробки работает "network portability", и придётся пописать код, работающий с ошибками с обоих сторон. Посмотрим, что внутри.

TL;DR: серверный интерсептор берёт ошибку и пакует информацию о ней для клиента, пользуясь особенностями gRPC:

```
// cockroach-grpc/middleware/server.go
package middleware

import (
    "context"

    "github.com/cockroachdb/errors"
    "github.com/cockroachdb/errors/extgrpc"
    "github.com/gogo/status"
    "google.golang.org/grpc"
)
```

```
func UnaryServerInterceptor(
    ctx context.Context,
    req interface{},
    info *grpc.UnaryServerInfo,
    handler grpc.UnaryHandler,
) (interface{}, error) {
    resp, err := handler(ctx, req)
    if err == nil {
        // Если обработчик не вернул ошибку, то здесь делать нечего.
        return resp, nil
    }

    // Пробуем вытащить "статус" из вернувшейся ошибки.
    // "Статус" в терминах gRPC - это RPC-ответ с кодом, сообщением и
какими-то деталями.
    st, ok := status.FromError(err)
    if !ok {
        // Если ошибка не является сама по себе "статусом", то
сделаем её таковой.
        // Наполним информацией и т.д.
        code := extgrpc.GetGrpcCode(err)
        st = status.New(code, err.Error())

        // Кодируем ошибку и засовываем в детали получившегося
статуса
        // - это пригодится для клиентского интерсептора.
        enc := errors.EncodeError(ctx, err)
        st, err = st.WithDetails(&enc)
        if err != nil {
            // https://jbrandhorst.com/post/grpc-errors/
            // "If this errored, it will always error
            panic(err)
        }
    }

    return resp, st.Err()

}
```

Клиентский интерсептор делает всё наоборот – пытается распаковать полученную информацию в ошибку:

```
// cockroach-grpc/middleware/client.go
package middleware
```

```
import (
    "context"

    "github.com/cockroachdb/errors"
    "github.com/gogo/status"
    "google.golang.org/grpc"
)

func UnaryClientInterceptor(
    ctx context.Context,
    method string,
    req interface{},
    reply interface{},
    cc *grpc.ClientConn,
    invoker grpc.UnaryInvoker,
    opts ...grpc.CallOption,
) error {
    err := invoker(ctx, method, req, reply, cc, opts...)

    // Если случилась ошибка, то пробуем вытащить из неё что-то
полезное.
    st := status.Convert(err)
    var reconstituted error
    for _, det := range st.Details() {
        // Если в деталях оказалась запакованная ошибка, то
восстановим её.
        if t, ok := det.(*errors.EncodedError); ok {
            reconstituted = errors.DecodeError(ctx, *t)
        }
    }

    // Если есть восстановленная ошибка, то возвращаем её.
    if reconstituted != nil {
        err = reconstituted
    }

    return err

}
```

Поддержка `errors.EncodeError` и `errors.DecodeError` для ошибок, созданных тестовым gRPC-сервером, лежит в github.com/cockroachdb/errors/extgrpc/ext_grpc.go.

Вывод

Вот и всё, никакой особой магии не оказалось: на стороне сервера пакуем ошибку, на стороне клиента – распаковываем.

Подобный подход можно использовать и для протокола HTTP, но в таком случае нам придётся придумывать дополнительные соглашения о формате передаваемой ошибки.

P.S. Более того, в примере выше не хватает stream-интерсепторов.

Тест "Ошибки с поддержкой передачи по сети с помощью HTTP"

В качестве самостоятельной работы представьте, что вам на собеседовании задают вопрос: "как бы вы реализовали ошибки с поддержкой передачи по сети?"

Очевидные вопросы, которые приходят в голову:

- Какой формат хранения будет у ошибки?
- Если представить, что ошибка поддерживает сложные детали вроде стека – как передавать их?
- Какой код необходимо будет писать на стороне сервера и клиента, чтобы это всё работало?
- Как реализовать поддержку `errors.Is` и `errors.As` для переданных по сети ошибок?

При желании пишите свои мысли по этому поводу в комментарии.

Выберите один вариант из списка

- ☐ Интересно, я подумаю.
- ☐ Спасибо, как-нибудь в следующий раз.

Подведём итоги

Складывается ощущение, что **github.com/cockroachdb/errors** – это целый фреймворк по работе с ошибками с огромным API и местами непростой внутренней реализацией.

Кажется, что большинство его фишек нельзя назвать must have, поэтому мы рассмотрели только те, которые по нашему скромному мнению заслуживают внимания:

- Стектрейс;
- `errors.UnwrapOnce`, `errors.UnwrapAll` с поддержкой **errors** и **github.com/pkg/errors**;
- `errors.If`;
- `errors.IsAny`;
- `errors.Mark`;
- `errors.WithSecondaryError`, `errors.CombineErrors`;
- **network portability**, если вы используете gRPC.

Поддержка передачи ошибок по сети тянет за собой маркировки и как следствие сравнение ошибок работает иначе, чем в стандартной библиотеке – будьте бдительны!

Помните, что если какая-то концепция вам приглянулась, её всегда можно утащить себе в более скромном виде без зависимости на монструозную библиотеку.