Go 业务开发中 Error & Context

毛剑

bilibili
iammao@vip.qq.com

探探 Gopher China 2019

Agenda

- Error
  - Background
  - Handle Error
  - Best Pratice
- Context
  - Background
  - Context With API
  - Best Pratice
- Conclusion

Error - Background

- 错误检查和打印
  - 分层开发导致的处处打印日志
  - 难以获取详细的堆栈关联
  - 根因丢失
- 业务错误处理
  - API中逻辑标识处理
  - API中错误消息展示
  - API中业务的Hint数据

```
if err != nil {
    return err
}
```

Errors are values By Rob Pike

Error - Handle Error

- 追加上下文
  - WithStack/Warp 来保存堆栈
  - WithMessage 来自定义消息

```go
// WithStack annotates err with a stack trace at the point WithStack was called.
// If err is nil, WithStack returns nil.
func WithStack(err error) error {
        if err == nil {
                return nil
        }
        return &withStack{
                err,
                callers(),
        }
}

type withStack struct {
        error
        *stack
}
```

```go
_, err := ioutil.ReadAll(r)
if err != nil {
        return errors.Wrap(err, "read failed")
}
```

Error types
github.com/pkg/errors
By Dave Cheney

Error - Handle Error

- 根因追踪
  - Cause获取根因，用于 Sentinel errors 逻辑处理

```
// If the error does not implement Cause, the original error will
// be returned. If the error is nil, nil will be returned without further
// investigation.
func Cause(err error) error {
        type causer interface {
                Cause() error
        }

        for err != nil {
                cause, ok := err.(causer)
                if !ok {
                        break
                }
                err = cause.Cause()
        }
        return err
}
```

```
if err == ErrSomething { … }
```

Sentinel errors
like io.EOF or low level errors
like the constants in the syscall package, like syscall.ENOENT

Error - Best Pratice

Error - Best Pratice

- 库内的代码使用 pkg/errors 的 New/Errorf 返回错误
- 库内的代码接受来自其他库的返回，直接透传
- 和标准库/第三方库交互时使用 WithStack/Wrap来携带上下文（不破坏）
- 把错误抛给调用者，而不是到处打日志
- 在最顶部的调用者或者是worker goroutine统一打印

```
// DefaultServer returns an Engine instance with the Recovery, Lo
func DefaultServer(conf *ServerConfig) *Engine {
    engine := NewServer(conf)
    engine.Use(Recovery(), Trace(), Logger(), CSRF(), Mobile())
    return engine
}
```

```
opt = append(opt, keepParam, grpc.UnaryInterceptor(s.interceptor))
s.server = grpc.NewServer(opt...)
s.Use(s.recovery(), s.handle(), serverLogging(), s.stats(), s.validate())
return
```


Error - Best Pratice

- 集中处理错误：errorgroup
  - 容易用错返回的context
  - 扇出没有控制
  - 业务代码容易panic

type Group
- func WithContext(ctx context.Context) (*Group, context.Context)
- func (g *Group) Go(f func() error)
- func (g *Group) Wait() error

```
[methods]
+GOMAXPROCS(n int)
+Go(f func(ctx context.Context) error)
+Wait() : error
-do(f func(ctx context.Context) error)
[functions]
+WithCancel(ctx context.Context) : *Group
+WithContext(ctx context.Context) : *Group
```

```go
var g errgroup.Group
var urls = []string{
    "http://www.golang.org/",
    "http://www.google.com/",
    "http://www.somestupidname.com/",
}
for _, url := range urls {
    // Launch a goroutine to fetch the URL.
    url := url // https://golang.org/doc/faq#closures_and_goroutines
    g.Go(func() error {
        // Fetch the URL.
        resp, err := http.Get(url)
        if err == nil {
            resp.Body.Close()
        }
        return err
    })
}
// Wait for all HTTP fetches to complete.
if err := g.Wait(); err == nil {
    fmt.Println("Successfully fetched all URLs.")
}
```


Error - Background

- 错误检查和打印
  - 分层开发导致的处处打印日志
  - 难以获取详细的堆栈关联
  - 根因丢失
- 业务错误处理
  - API中逻辑标识处理
  - API中错误消息展示
  - API中业务的Hint数据

```
if err != nil {
    return err
}
```

Errors are values By Rob Pike

Error - Handle Error

- 定义了适配业务的 Error Type
  - Code: 为业务错误码
  - Message: 返回标准库错误消息
  - Details: 返回业务Hint
- Code包级别工具方法
  - Cause 适配 pkg/errors，用于根因获取后的error检测
  - Equal，检测两个Codes接口实现者的Code码是否一致

```
// Codes ecode error interface which h
type Codes interface {
    // sometimes Error return Code in
    // NOTE: don't use Error in monitc
    Error() string
    // Code get error code.
    Code() int
    // Message get code message.
    Message() string
    //Detail get error detail,it may b
    Details() []interface{}
    // Equal for compatible.
    // Deprecated: please use ecode.Ec
    Equal(error) bool
}
```


Error - Handle Error

- 固定错误码类型: Code
  - 业务作为强标识返回
  - 约定变量名称结合错误码
  - 返回统一的错误Message
- 自定义错误码类型: Status
  - 适配gRPC框架返回的Status
  - 提供自定义Message
  - 提供自定义Details

```
// A Code is an int error code spec.
type Code int
```

```
// favorite
FavNameTooLong          = New(11001) // 收藏夹名称过长
FavMaxFolderCount       = New(11002) // 达到最大收藏夹数
FavCanNotDelDefault     = New(11003) // 不能删除默认收藏夹
```

```
message Status {
  // The error code see ecode.Code
  int32 code = 1;

  // A developer-facing error message, whic
  string message = 2;

  // A list of messages that carry the erro
  // message types for APIs to use.
  repeated google.protobuf.Any details = 3;
}
```


Error - Handle Error

ecode Method 优化

- ecode.Error interface 重命名为 ecode.Codes 避免混淆
- ecode.Codes 增加方法Details []interface{}
- ecode.ecode 改为 type Code int，对外暴露

添加以下 Method

```
// Error 覆盖 ecode message.
Error(code Code, message string) *Status

// Errorf 覆盖 ecode message, 使用自定义format
Errorf(code Code, format string, args ...interface{}) *Status
```

gRPC error 传递

封装的 gRPC 框架有义务对ecode 进行传达而不是单纯的返回 Error message，理论上服务端不需要主动返回 gRPC status

对于服务端返回的 ecode 客户端应该收到相同的 ecode

Ecode -> gRPC status -> network -> gRPC status -> Ecode

Error - Best Pratice

- 公共错误码: 标准库状态码
- 业务错误码: 业务命名空间
- API: 明确的错误码返回列表
- API: 微服务间传递的错误，立即消化转化，不随意透传
- 避免全局依赖：公共错误码统一维护、业务错误码自维护

```
message Status {
  // 一个容易被客户端进行处理的错误码。
  // 实际的错误码在 google.rpc.Code 里面定义
  int32 code = 1;

  // 面向开发人员的可读性高的英文错误信息
  // 这个错误信息应该同时说明错误的原因以及提供一个可操作的处理错误的方法
  string message = 2;

  // 额外的错误信息，这些错误信息可以被客户端代码用来处理这个错误，
  // 例如告诉客户端隔多长时间再次尝试或者提供一个帮助链接
  repeated google.protobuf.Any details = 3;
}
```

https://google-cloud.gitbook.io/api-design-guide/errors

Agenda

- Error
  - Background
  - Handle Error
  - Best Pratice
- Context
  - Background
  - Context With API
  - Best Pratice
- Conclusion

Context - Background

- 超时 & 取消
  - 业务基础库从无到有
  - 可用性：超时
  - 可用性：取消
- 元数据传递
  - 同进程：同步生命周期传递
  - 同进程：异步生命周期传递
  - 跨进程：网络传递

Use context values only for request-scoped data that transits processes and API boundaries, not for passing optional parameters to functions.

```
func DoSomething(ctx context.Context, arg Arg) error {
        // ... use ctx ...
}
```


Context - Context With API

- 覆盖业务库
  - Log、Sync、Cache、Database、Queue
  - gRPC、HTTP、Infra
- 显示传递 大于 隐式传递
  - 脚手架把DAO、Service等代码生成首参Context的Stub
  - 框架代码的Context不下沉，统一使用context.Context传递

```go
// Conn represents a connection to a Redis server.
type Conn interface {
    // Close closes the connection.
    Close() error

    // Err returns a non-nil value if the connection is broken. The returned
    // value is either the first non-nil value returned from the underlying
    // network connection or a protocol parsing error. Applications should
    // close broken connections.
    Err() error

    // Do sends a command to the server and returns the received reply.
    Do(commandName string, args ...interface{}) (reply interface{}, err error)

    // Send writes the command to the client's output buffer.
    Send(commandName string, args ...interface{}) error

    // Flush flushes the output buffer to the Redis server.
    Flush() error

    // Receive receives a single reply from the Redis server
    Receive() (reply interface{}, err error)

    // WithContext
    WithContext(ctx context.Context) Conn
}
```


Context - Context With API

- 通常在流量入口，比如HTTP、gRPC设置超时控制，在基础库内统统继承超时（尽可能规避OOM）

```
// get derived timeout from grpc context,
// compare with the warden configured,
// and use the minimum one
timeout := time.Duration(conf.Timeout)
if dl, ok := ctx.Deadline(); ok {
    ctimeout := time.Until(dl)
    if ctimeout-time.Millisecond*20 > 0 {
        ctimeout = ctimeout - time.Millisecond*20
    }
    if timeout > ctimeout {
        timeout = ctimeout
    }
}
ctx, cancel = context.WithTimeout(ctx, timeout)
defer cancel()
```

```
// Context is the most important part. It allows us
// middleware, manage the flow, validate the JSON of
// JSON response for example.
type Context struct {
    context.Context

    Request *http.Request
    Writer  http.ResponseWriter

    // flow control
    index    int8
    handlers []HandlerFunc

    // Keys is a key/value pair exclusively for the
    Keys map[string]interface{}

    Error error

    method string
    engine *Engine
}
```


Context - Context With API

- 超时传递不仅仅是考虑进程内全链路覆盖，更应该是跨服务级别的全链路覆盖
  - 利用gRPC、HTTP 的 Metadata 传递，框架层面支持或者自定义Header传递。顶层定义超时，逐层传递，逐层踢去网络衰减。（如边缘接入GFE）

Context - Background

- 超时 & 取消
  - 业务基础库从无到有
  - 可用性：超时
  - 可用性：取消
- 元数据传递
  - 同进程：同步生命周期传递
  - 同进程：异步生命周期传递
  - 跨进程：网络传递

Use context values only for request-scoped data that transits processes and API boundaries, not for passing optional parameters to functions.

```
func DoSomething(ctx context.Context, arg Arg) error {
        // ... use ctx ...
}
```


Context - Context With API

- 业务请求级数据传递
  - 鉴权信息
- 框架元数据传递
  - 信息: Caller、Address、Device、Trace
  - 路由: Color、Mirror
  - 控制： Timeout
  - 调度: gRPC中的Score

```
// get grpc metadata(trace & remote_ip & color)
var t trace.Trace
cmd := nmd.MD{}
if gmd, ok := metadata.FromIncomingContext(ctx); ok {
    for key, vals := range gmd {
        if nmd.IsIncomingKey(key) {
            cmd[key] = vals[0]
        }
    }
}
if t == nil {
    t = trace.New(args.FullMethod)
} else {
    t.SetTitle(args.FullMethod)
}

if pr, ok := peer.FromContext(ctx); ok {
    addr = pr.Addr.String()
    t.SetTag(trace.String(trace.TagAddress, addr))
}
defer t.Finish(&err)

// use common meta data context instead of grpc context
ctx = nmd.NewContext(ctx, cmd)
```


Context - Context With API

- Incoming: 跨进程传入
  - 接受来自外部 HTTP、gRPC的元数据挂载Provier的context
- Outgoing: 跨进程传出
  - 作为Consumer请求外部服务时候，将元数据传递出去
- 框架自动枚举元数据处理扇出和挂载，避免硬编码

```go
var outgoingKey = map[string]struct{}{
    Color:      struct{}{},
    RemoteIP:   struct{}{},
    RemotePort: struct{}{},
    Mirror:     struct{}{},
}

var incomingKey = map[string]struct{}{
    Caller: struct{}{},
}

// IsOutgoingKey represent this key should propagate
func IsOutgoingKey(key string) bool {
    _, ok := outgoingKey[key]
    return ok
}

// IsIncomingKey represent this key should extract f
func IsIncomingKey(key string) (ok bool) {
    _, ok = outgoingKey[key]
    if ok {
        return
    }
    _, ok = incomingKey[key]
    return
}
```


Context - Context With API

- Goroutine传递
  - 将原始meta进行Copy，产生新的context，减少手动传递漏、错等问题
  - Metadata的Key为: type mdKey struct{} 私有类型

```
// WithContext return no deadline context and retain m
func WithContext(c context.Context) context.Context {
    md, ok := FromContext(c)
    if ok {
        nmd := md.Copy()
        return NewContext(context.Background(), nmd)
    }
    return context.Background()
}
```

packages should define keys as an unexported type to avoid collisions.

Context - Best Pratice

- Context 非Owner情况下，传错context
  - 收敛通用的异步任务模型，做成基础库

```go
// Do save a callback func.
func (c *Fanout) Do(ctx context.Context, f func(ctx context.Context)) (err error) {
    if f == nil || c.ctx.Err() != nil {
        return c.ctx.Err()
    }
    nakeCtx := metadata.WithContext(ctx)
    if tr, ok := trace.FromContext(ctx); ok {
        tr = tr.Fork("", "Fanout:Do").SetTag(traceTags...)
        nakeCtx = trace.NewContext(nakeCtx, tr)
    }
    select {
    case c.ch <- item{f: f, ctx: nakeCtx}:
    default:
        err = ErrFull
    }
    stats.State(c.name+"_channel", int64(len(c.ch)))
    return
}
```


Context - Best Pratice

- context.Background 只应用在最高等级，作为所有派生 context 的根
- context.TODO 应用在不确定要使用什么的地方，或者以后会更新会使用
- context 取消是建议性的，这些函数可能需要一些时间来清理和退出
- context.Value 不应该被用来传递可选参数。这使得 API 隐式的并且可以引起错误。取而代之的是，这些值应该作为参数传递
- 不要将 context 存储在结构中，显式传递它们，最好是作为第一个参数
- 永远不要传递不存在的 context 。不确定就用TODO
- Context 结构没有取消方法，因为只有派生 context 的函数才应该取消 context

Agenda

- Error
  - Background
  - Handle Error
  - Best Pratice
- Context
  - Background
  - Context With API
  - Best Pratice
- Conclusion

Conclusion

- 业务基础库没想象中那么简单
  - 简单才可靠
  - 让每个人正确的使用
  - 去大师化编程
  - 鲁棒以及健壮性
- 常常思考和进步
  - 看看你的消费者怎么使用的
  - CaseStudy中记录大家容易出错的点
  - 不断质疑自己的设计，参考优秀的设计