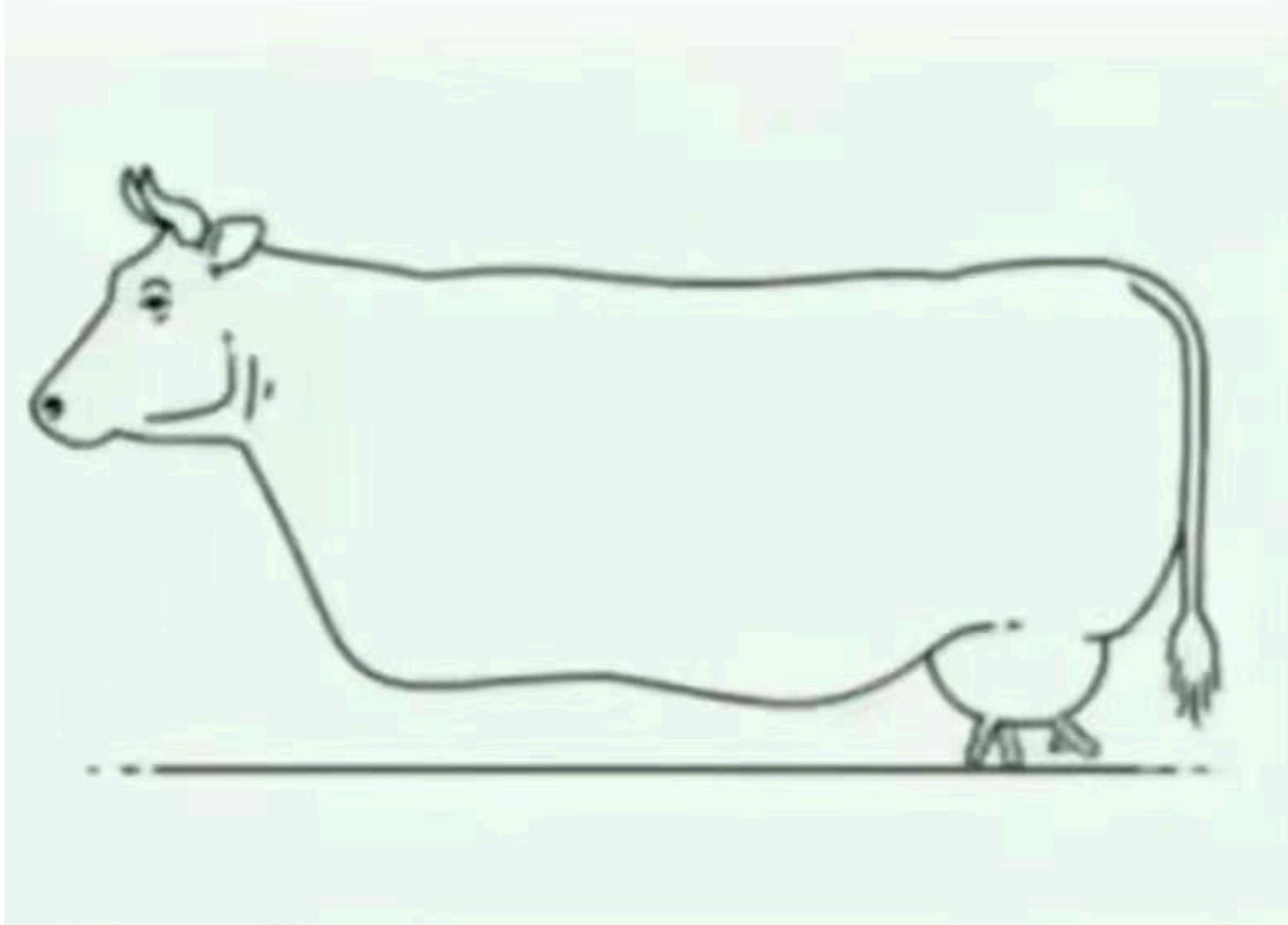


Go工程化实践

Engineering of Go Programming Language

当我们接手历史代码时，常常感叹：这代码我改不动！

编程的第一法则：如果您的代码以某种莫名其妙方式跑起来了，就不要再碰它了。



而在实际项目中

又要马儿跑 迭代新需求 vs 导致不稳定

又要马儿不吃草 保障稳定性 vs 不迭代

对人 - 代码可读性

Programs must be written for people to read, and only incidentally for machines to execute.

- Structure and Interpretation of Computer Programs 《计算机程序的构造和解释》

Software engineering is what happens to programming when you add time and other programmers.

- Russ Cox

对项目 - 保证质量

原则尽可能简单、流程尽可能自动化

难点 - 拉齐成员主观认知

入门 - 学习标准与经验

熟练 - 用工具提升研发效能

精通 - 控制炫技、沉淀方法论

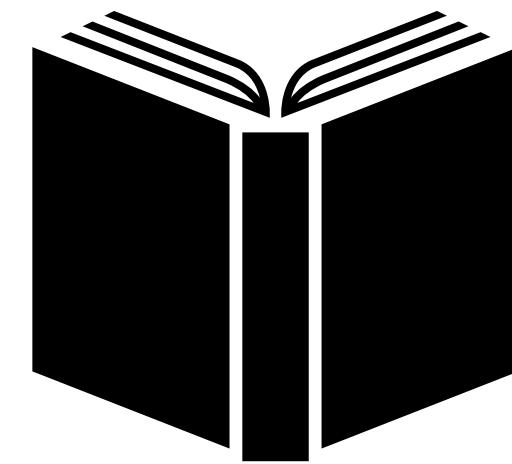
实践 - 可自动化的约束

语言 - 强类型、编译检查

库 - 标准库/Gorm等三方库

CICD - 静态检测、测试覆盖率

...



引入《Peopleware》中
一个很有意思的观点

**The major problems of our work
are not so much **technological** as
sociological in nature.**

本质上，我们工作中的主要问题，与其说是技术问题，不如说是**社会学问题**。

工程化并没有绝对正确，但在具体的场景下会有Best Practice

本次分享将围绕4个相对争议较小的case展开，而有更多的、具有争议性的内容放在ppt最后，希望引起大家的思考，欢迎在群里讨论。

CONTENTS

课程目录

1. 处理错误的三种方式
2. 分层下的Error Handling
3. 类型别名
4. 对象的部分更新
5. 小结

1.经典Go逻辑

处理错误的三种方式

```
type ZooTourl interface {
    Enter() error
    VisitPanda(panda *Panda) error
    VisitTiger(tiger *Tiger) error
    Leave() error
}

func Tourl(t ZooTourl, panda *Panda, tiger *Tiger) error {
    if err := t.Enter(); err != nil {
        return errors.WithMessage(err, "Enter failed")
    }

    if err := t.VisitPanda(panda); err != nil {
        return errors.WithMessagef(err, "VisitPanda failed, panda is %v", panda)
    }

    if err := t.VisitTiger(tiger); err != nil {
        return errors.WithMessagef(err, "VisitTiger failed, tiger is %v", tiger)
    }

    if err := t.Leave(); err != nil {
        return errors.WithMessage(err, "Leave failed")
    }

    return nil
}
```

接口定义：
直观地返回error

分步处理：
每个步骤可以针对具体
返回结果进行处理

处理错误的三种方式

```
type ZooTour2 interface {
    Enter()
    VisitPanda(panda *Panda)
    VisitTiger(tiger *Tiger)
    Leave()

    Err() error
}

func Tour2(t ZooTour2, panda *Panda, tiger *Tiger) error {
    t.Enter()
    t.VisitPanda(panda)
    t.VisitTiger(tiger)
    t.Leave()

    if err := t.Err(); err != nil {
        return errors.WithMessage(err, "ZooTour failed")
    }

    return nil
}
```

接口定义：
统一返回error

集中编写业务逻辑代码，
最后统一处理error

```
type myZooTour struct {
    err error
}

func (t *myZooTour) VisitPanda(panda *Panda) {
    if t.err != nil {
        return
    }
    // ...
}
```

将error保存到对象内部，处
理逻辑交给每个方法，本
质上仍是顺序执行

标准库中的bufio.Scanner就是用这种方式实现的

3.利用函数式编程的延迟运行

处理错误的三种方式

```
type MyFunc func(t ZooTour1) error

func NewEnterFunc() MyFunc {
    return func(t ZooTour1) error {
        return t.Enter()
    }
}
```

函数定义：
具体行为封装为
函数类型

```
type Walker interface {
    Next() MyFunc
}

// 一个简单的切片walker
type SliceWalker struct {
    index int
    fns   []MyFunc
}
```

分离关注点 - 遍历访问
用 **数据结构** 定义运行顺序，
根据场景选择，如顺序、逆
序、二叉树遍历等

```
func BreakOnError(t ZooTour1, walker Walker) error {
    for {
        f := walker.Next()
        if f == nil {
            break
        }
        if err := f(t); err != nil {
            // break
        }
    }
}
```

分离关注点 - 运行逻辑
将代码的 **控制流** 逻辑抽离，灵活调整

Kubernetes中的visitor对此就有很多种扩展方式，分离了数据和行为，有兴趣可以去扩展阅读

上面这三个例子，是Go项目处理错误使用频率最高的三种方式，也可以应用在error以外的处理逻辑。

你对这三种方式有什么个人理解，能否分享一下 你对go语言error处理心得呢？ 也可以聊聊，在特定的业务场景，你认为这三种代码风格的存在的利或弊。

Case 1

```
type ZooTour1 interface {
    Enter() error
    VisitPanda(panda *Panda) error
    VisitTiger(tiger *Tiger) error
    Leave() error
}
```

Case 2

```
type ZooTour2 interface {
    Enter()
    VisitPanda(panda *Panda)
    VisitTiger(tiger *Tiger)
    Leave()
    Err() error
}
```

Case 3

```
func BreakOnError(t ZooTour1, walker Walker) error {
    for {
        f := walker.Next()
        if f == nil {
            break
        }
        if err := f(t); err != nil {
            // break
        }
    }
}
```

CONTENTS

课程目录

1. 处理错误的三种方式
2. 分层下的Error Handling
3. 类型别名
4. 对象的部分更新
5. 小结

1.一个常见的三层调用

分层下的Error Handling

```
// controller
_, err := models.ListOneTiger("")
if err != nil {
    log.Trace.Errorf(ctx, legoTrace.DLTagUndefined, "method=Controller Method||%v", err)
    return nil, bizerror.PARAM_ERROR
}

// model
tigers, err := dao.ListTigers(name)
if err != nil {
    log.Trace.Errorf(ctx, legoTrace.DLTagUndefined, "method=Model Method||%v", err)
    return nil, fmt.Errorf("method=Model Method||error=%v", err )
}

// dao
if err != nil {
    log.Trace.Errorf(ctx, legoTrace.DLTagUndefined, "method=Dao Method||error=%v", err)
    return nil, fmt.Errorf("method=Dao Method||name=%s||error=%v", name, err)
}
```

1 – Controller

2 – Model

3 – Dao

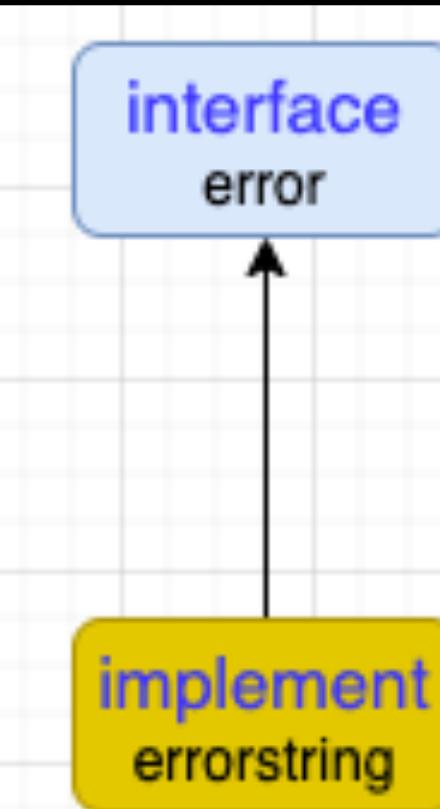
1. 分层带来的大量日志；
2. 拼接错误字符串很费力；

2. 标准库的error实现

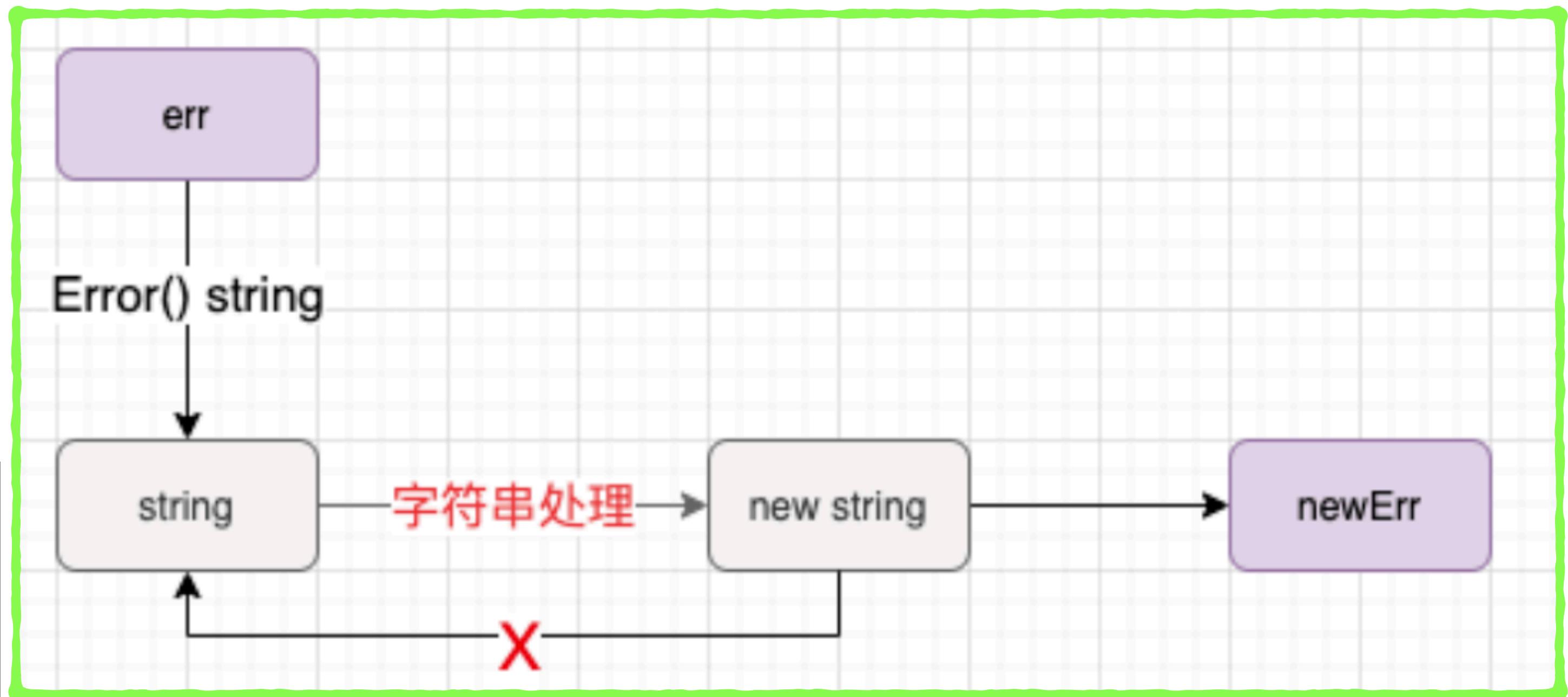
分层下的Error Handling

```
// 接口定义  
type error interface {  
    Error() string  
}
```

```
newErr := fmt.Errorf("Something has error %v", err)
```



```
// 实现  
type errorString struct {  
    s string  
}  
  
func (e *errorString) Error() string {  
    return e.s  
}
```



字符串处理fmt.Errorf是一个单向操作，破坏了原始err对象

Making errors more **informative** for both programs and people.

参考内容

Go Error Problem Overview

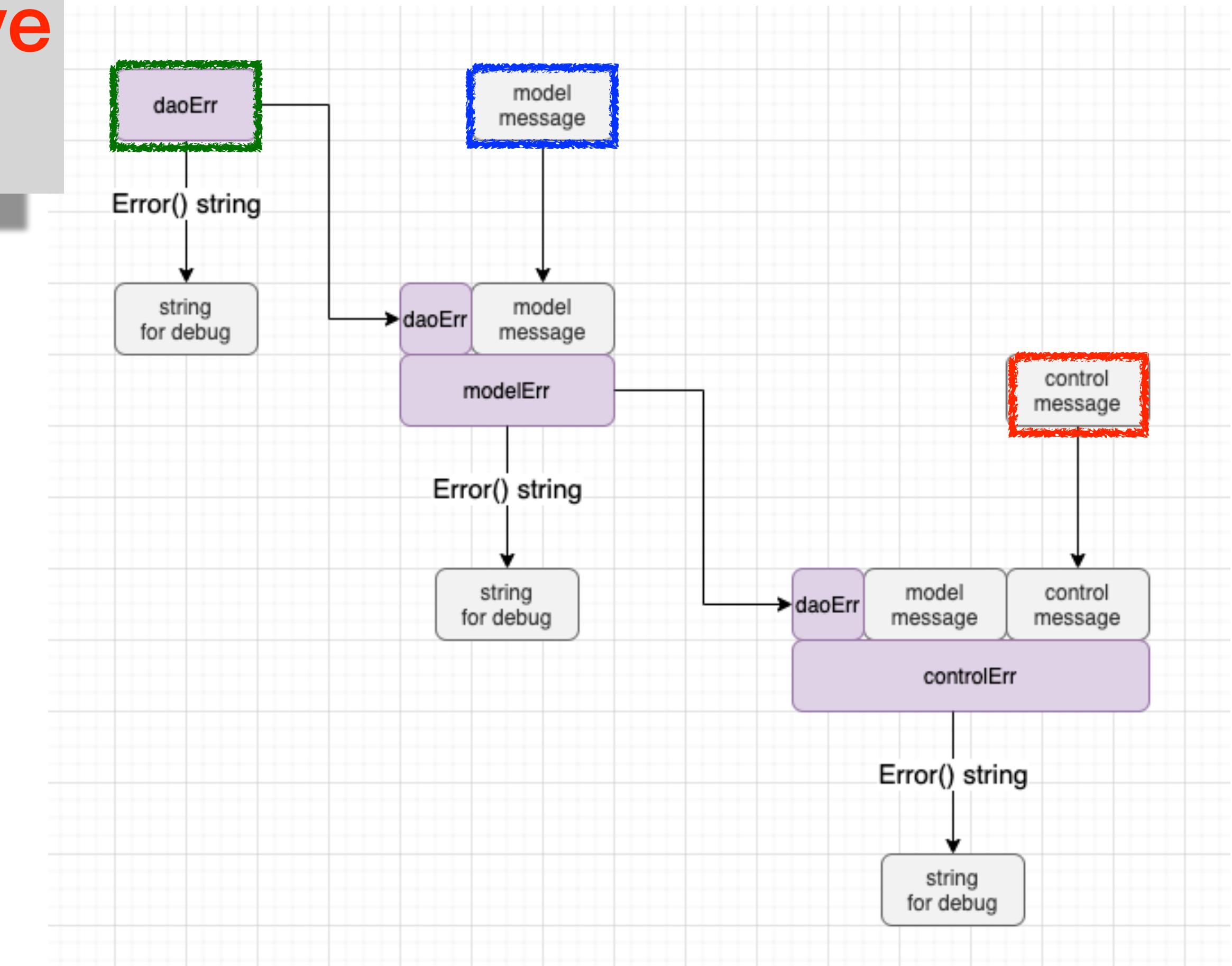
<https://go.googlesource.com/proposal/+/master/design/go2draft-error-values-overview.md>

Go 2 Error Proposal

<https://go.googlesource.com/proposal/+/master/design/29934-error-values.md>

Go 1.13 Errors

<https://blog.golang.org/go1.13-errors>



错误处理的思想: An error should be handled only once.

```
// controller
_, err := models.ListOneTiger("")
if err != nil {
    err = errors.WithMessagef(err, "Controller : req %+v", req)
    // 这里的打印方式要选择%+v才会变成堆栈格式, 而%v依旧调用的是 Error() 方法
    log.Trace.Errorf(ctx, legoTrace.DLTagUndefined, "%+v", err)
    return nil, getGrpcError(err)
}

// model
tigers, err := dao.ListTigers(name)
if err != nil {
    return nil, errors.WithMessagef(err, "Model : ListTigers parameters %s", name)
}

// dao
var err = gorm.ErrInvalidSQL
if err != nil {
    return nil, errors.Wrapf(err, "Dao : ListTigers name %s", name)
}
```

处理Error, 打印堆栈

WithMessage
追加信息

Wrap
初始化堆栈

```
invalid SQL
Dao : ListTigers name demo
git.xiaojukeji.com/zoo-project/zoo/dao.ListTigers
/Users/didi/GoProject/src/git.xiaojukeji.com/zoo-project/zoo/dao/tiger.go:54
git.xiaojukeji.com/zoo-project/zoo/models.ListOneTiger
/Users/didi/GoProject/src/git.xiaojukeji.com/zoo-project/zoo/models/tiger.go:20
git.xiaojukeji.com/zoo-project/zoo/controller.(*PingController).Ping
/Users/didi/GoProject/src/git.xiaojukeji.com/zoo-project/zoo/controller/controller-ping.go:21
git.xiaojukeji.com/zoo-project/zoo/controller.TestPingController_Ping.func1
/Users/didi/GoProject/src/git.xiaojukeji.com/zoo-project/zoo/controller/controller-ping_test.go:39
testing.tRunner
/usr/local/go/src/testing/testing.go:1123
runtime.goexit
/usr/local/go/src/runtime/asm amd64.s:1374
Model : ListTigers parameters demo
Controller : req name: "demo"
```

Dao – Wrap

Discussion-2

如果你的项目引入了堆栈化的错误，可能带来什么样的利与弊呢？

在RPC服务中，**错误码**是一个必不可少的信息，如何**将error对接到code呢？**

```
// 新增了一个controller的方法，直接调用一个已有的model层代码
_, err := models.ListOneTiger("")
if err != nil {
    // 怎么从err中提取错误码，作为rpc返回的错误码呢？
    return
}
```

常规方法：深入下层的代码实现，去挖掘具体的错误原因；或者对任何错误都返回统一的错误码
疑惑：有没有一种方式，可以让调用者**不再关心下层的实现细节**，能**直接从err提取出错误码**呢？

方案1：错误字符串中寻找关键词

```
_ , err := models.ListOneTiger( "" )  
if err != nil {  
    if strings.Index(err.Error(), "key word") >= 0 {  
        // 返回指定错误码  
    }  
    return  
}
```

无需改动底层代码

必须需要了解下层代码的实现，并随之不断调整key word

方案2：增加返回值

```
_ , code , err := models.ListOneTiger( "" )  
if err != nil {  
    // 错误码为code  
    return  
}
```

很直观的解决方案

大量的函数增加了一个返回值，大大提升了写个函数的成本

方案3：统一错误底层实现

```
type MyError struct {
    Message string
    Code int
}

func (e *MyError) Error() string {
    return e.s
}

func NewError(msg string, code int) error{
    return &MyError{Message:"", Code}
}
```

```
// controller
_, err := models.ListOneTiger("")
if err != nil {
    mErr, ok := err.(*MyError)
    if ok {
        // 错误码为 mErr.Code
    }
}

// model
return NewError("add your message", 10000)
```

用一个统一的类型判断提取出结构体

需要统一所有package的error实现底层都为MyError，很难兼容第三方库

方案4：从依赖结构体到依赖接口

```
// controller  
_, err := models.ListOneTiger("")  
if err != nil {  
    mErr, ok := err.(interface{GetCode() int})  
    if ok {  
        // 错误码为 mErr.GetCode()  
    }  
}
```

在多模块中不用依赖共同库中的具体结构体，意义尤为重大

要求error底层的结构体支持 `GetCode() int` 的方法，对第三方模块也有一定要求

3.关键实现

分层下的Error Handling - 错误码

```
type Status struct {
    state      protoimpl.MessageState
    sizeCache  protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields

    // The status code, which should be an enum value of [google.rpc.Code].
    Code int32 `protobuf:"varint,1,opt,name=code,proto3" json:"code,omitempty"`

    // A developer-facing error message, which should be in English. Any
    // user-facing error message should be localized and sent in the
    // [google.rpc.Status.details][google.rpc.Status.details] field, or localized by the client.
    Message string `protobuf:"bytes,2,opt,name=message,proto3" json:"message,omitempty"`

    // A list of messages that carry the error details. There is a common set of
    // message types for APIs to use.
    Details []*any.Any `protobuf:"bytes,3,rep,name=details,proto3" json:"details,omitempty"`
}
```

Code - 错误码
Message - 信息
Details - 详情

google.golang.org/grpc/status

nuwa idl背后的error结构体

github.com/pkg/errors 的errors.As
err不断Unwrap，提取为target

```
func As(err error, target interface{}) bool {
    // ignored
    for err != nil {
        if reflectlite.TypeOf(err).AssignableTo(targetType) {
            val.Elem().Set(reflectlite.ValueOf(err))
            return true
        }
        if x, ok := err.(interface{ As(interface{}) bool }); ok && x.As(target) {
            return true
        }
        err = Unwrap(err)
    }
    return false
}
```

1-循环中不断Unwrap

2-直到能赋值给目标

4.代码实现

分层下的Error Handling - 错误码

```
// dao  
return nil,errors.Wrapf(bizerror.GrpcDB_TIGER_SAVE_ERROR, "Dao : ListTigers name %s error %v",name,err)  
  
// model调用dao层的出现错误err后  
return errors.WithMessagef(err,"Model : ListTigers parameters %s", name)  
// model层生成一个全新的error  
return errors.Wrapf(bizerror.GrpcDB_TIGER_NUM_ERROR, "ListTigers number is %d, not 1", len(tigers))
```

1-调用第三方SDK或生成一个全新的error时，Wrap一个nuwa的error

```
// controller  
type grpcError interface {  
    error  
    GRPCStatus() *status.Status  
}
```

2-追加错误信息时，不覆盖原有error

```
// 提取错误的实现  
func getGrpcError(err error) (codes.Code, string) {  
    if err == nil {  
        return bizerror.ErrOk,bizerror.Msg[bizerror.ErrOk]  
    }  
    var unwrap grpcError  
    if errors.As(err, &unwrap) {  
        return unwrap.GRPCStatus().Code(),unwrap.GRPCStatus().Message()  
    }  
    return bizerror.ErrDefault,bizerror.Msg[bizerror.ErrDefault]  
}
```

4-1 没有错误，返回成功

4-3 提取成功，返回提取结果

4-2 提取不到，返回默认错误

3-定义一个接口，实现两个方法
1.error的方法
2.提取错误的GRPCStatus方法

4-提取错误，分情况返回

为什么把unwrap定义成结构体呢？是因为底层结构体被grpc放在internal目录中，无法引用

5.回顾github.com/pkg/errors的设计理念 分层下的Error Handling - 错误码

**Hide implementation details from programs
while displaying them for diagnosis**

隐藏实现

标准库

利用[error接口](#)隐藏底层实现

诊断问题

github.com/pkg/errors

利用底层保存的[错误堆栈](#)来协助排查问题

github.com/pkg/errors提供的另一块重要能力

从error接口中提取底层信息，如本例的错误码、错误信息

常用的函数有errors.As/Is/Unwrap

如果我们收到了一个来自公共库的error，它的风格有如下两种：
你觉得这两种实现有什么区别？你更倾向于哪种？

```

var (
    ErrRecordNotFound = errors.New("record not found")
    ErrInvalidSQL = errors.New("invalid SQL")
    ErrInvalidTransaction = errors.New("no valid transaction")
    ErrCantStartTransaction = errors.New("can't start transaction")
    ErrUnaddressable = errors.New("using unaddressable value")
)

func CallBack1() error{
    return ErrRecordNotFound
}

// 调用方
func usage1() {
    err := pkg.CallBack1()
    if err == pkg.ErrRecordNotFound {
        // handle certain here
    }
}

```

sentinel error

```

var (
    ErrRecordNotFound ="record not found"
)

func CallBack2() error{
    return errors.New(ErrRecordNotFound)
}

// 调用方
func usage2() {
    err := pkg.CallBack2()
    if err != nil && err.Error() == pkg.ErrRecordNotFound {
        // handle certain here
    }
}

```

在返回时创建error对象

在一个服务中，错误非常庞大，如何给错误分类呢？

```
// 服务降级: MySQL -> Redis
err := readMySQL()
if IsDegraded(err) {
    readRedis()
}
```

有了之前的经验，这里的实现并不难。最直观的想法，就是把需要降级的error维护到一个map里。

相比较于手工维护这个map，我们能不能给出一个更好的方案呢？

A computer program can write a computer program

发掘重复工作背后的一致性

```
/** 服务错误码，会同步至idl/error/error.go文件中 */
enum ErrCode {
    /** 成功 */
    ErrOk = 0
    /** 默认 */
    ErrDefault = 1
    /** 参数错误 */
    PARAM_ERROR = 8304001
    /** 保存TIGER失败 Degraded*/
    DB_TIGER_SAVE_ERROR = 8304002
    /** TIGER数量不匹配 Degraded*/
    DB_TIGER_NUM_ERROR = 8304003
}
```

```
// Msg 异常消息映射表
var Msg = map[codes.Code]string{
    ErrOk: "成功",
    ErrDefault: "默认",
    PARAM_ERROR: "参数错误",
    DB_TIGER_SAVE_ERROR: "保存TIGER失败 Degraded",
    DB_TIGER_NUM_ERROR: "TIGER数量不匹配 Degraded",
}
```

IDL引入降级定义字段：Degraded

脚本文件 gen.sh

```
nuwa gen
```

确保IDL生成的文件最新

```
(cat << EOF  
// Code generated by gen script. DO NOT EDIT.
```

提示不要手动编辑

```
var degradedMap = map[codes.Code]struct{}{  
    `grep "Degraded" idl/error/error.go | grep -v "Grpc" | grep -v "\/\\" | awk -F' ' '{print $1"{},"}'`  
}  
EOF  
) > $2
```

用linux的shell，每次都读取对应的源文件error.go

```
go fmt $2
```

格式化输出文件

3.go generate

分层下的Error Handling - 错误分类

```
//go:generate ./gen.sh idl/error/error.go idl/error/handler.go
```

```
var degradedMap = map[codes.Code]struct{}{  
    DB_TIGER_SAVE_ERROR: {},  
    DB_TIGER_NUM_ERROR:  {},  
}
```

```
type grpcError interface {  
    error  
    GRPCStatus() *status.Status  
}
```

```
func IsDegraded(err error) bool {  
    if err == nil {  
        return false  
    }  
  
    var unwrap grpcError  
    if errors.As(err, &unwrap) {  
        _, ok := degradedMap[unwrap.GRPCStatus().Code()]  
        return ok  
    }  
    return false  
}
```



降级的error map，也是每次go generate后动态变化部分



类似提取错误码，在gen.sh里写死，对外只暴露IsDegraded

Go Generate 的功能看似只是运行命令，但在工程上非常实用：它不仅能规避ctrl+c/ctrl+v带来的错误，也能反向推动我们形成一套“**代码模式**”。

Go语言最庞大、也是最成功的项目 – Kubernetes，就引用了大量的代码生成技术。

在大家的日常开发中，你觉得有什么场景适合go-generate吗？或者滥用这个功能可能带来的风险？

CONTENTS

课程目录

1. 分层下的Error Handling
2. 处理错误的三种方式
3. 类型别名
4. 对象的部分更新
5. 小结

1.一个常见的状态属性

类型别名

```
type Order struct {  
    Status int // 1: 正常, 2: 删除中, 3: 已删除  
}
```

依赖注释的实现

```
func (o *Order) SetStatus(targetStatus int) {  
    o.Status = targetStatus  
}
```

入参限制为int

```
func order1() {  
    var o = new(Order)  
    o.SetStatus(3)  
}
```

调用方的体验很差：

1. 必须跳转到具体实现看注释
2. 传了不合理的值，也不会有任何提示

不要把功能的正确实现，完全依赖于他人会主动阅读注释或实现细节

```
const (          将注释转化为更为具体的常量  
    Normal = iota + 1
```

```
    Deleting  
    Deleted  
)
```

调用方的体验改善：
明确了解参数含义是Deleted

```
func order2() {  
    var o = new(Order)  
    o.setStatus(Deleted)  
}
```

不要把公共变量定义在common/
consts包，而具体的调用方散落
在其余的package中。

The greater the distance between a
name's declaration and its uses, the
longer the name should be.

3.引入类型定义

类型别名

```
type OrderStatus int

const (
    Normal OrderStatus = iota + 1
    Deleting
    Deleted
)

type Order struct {
    Status OrderStatus
}

func (o *Order) SetStatusV2(targetStatus OrderStatus) {
    o.Status = targetStatus
}

func order2() {
    var o = new(Order)
    o.SetStatusV2(Deleted)
}
```

改造原有的实现

```
func fromApi(status int) {
    var o = new(Order)
    o.setStatusV2(status)
}
```

利用强类型语言的优势，在IDE或编译时，报传参的类型错误，尽早修正

调用方明确得知“状态”的类型是：OrderStatus
跳转过去后能看到具体支持的类型

3.引入类型定义

类型别名

1-无法限制常量

```
func constStatus() {  
    var o = new(Order)  
    o.setStatusV2(4)  
}
```

2-无法限制类型转换

```
func transferStatus(state int) {  
    var o = new(Order)  
    o.setStatusV2(OrderStatus(state))  
}
```

为什么Go语言的类型不能限制范围呢？

这里有2个比较有意思的讨论，有兴趣可以看看

<https://github.com/golang/go/issues/29649>

<https://github.com/golang/go/issues/36387>



bradfitz commented on 27 Feb 2019

What is the zero value of the MONTHS type from:

```
type MONTHS range 1 ... 12
```

If I write:

```
var m MONTHS
```

What is m ?

4.它山之石 - 标准库的http.NewRequest

类型别名

```
const (
    MethodGet      = "GET"
    MethodHead     = "HEAD"
    MethodPost     = "POST"
    MethodPut      = "PUT"
    MethodPatch    = "PATCH" // RFC 5789
    MethodDelete   = "DELETE"
    MethodConnect  = "CONNECT"
    MethodOptions  = "OPTIONS"
    MethodTrace    = "TRACE"
)
```

```
func NewRequest(method, url string, body io.Reader) (*Request, error) {
    return NewRequestWithContext(context.Background(), method, url, body)
}

if !validMethod(method) {
    return nil, fmt.Errorf("net/http: invalid method %q", method)
}
```

所有支持的method

传入的method参数不受限制，
在内部进行判断

约定 – 用具体定义调用
如 http.MethodTrace

这种方式有个好处：当库的版本不一致时，如果引用了一个不存在的具体定义，会直接在编译期报错，尤其适合在有公共依赖包的场景

校验 – 增加Valid方法

强制保证数据准确

- 1 – 返回值会增加一个error
- 2 – int类型可以结合iota+start/end判断
- 3 – 可以结合go generate每次自动生成

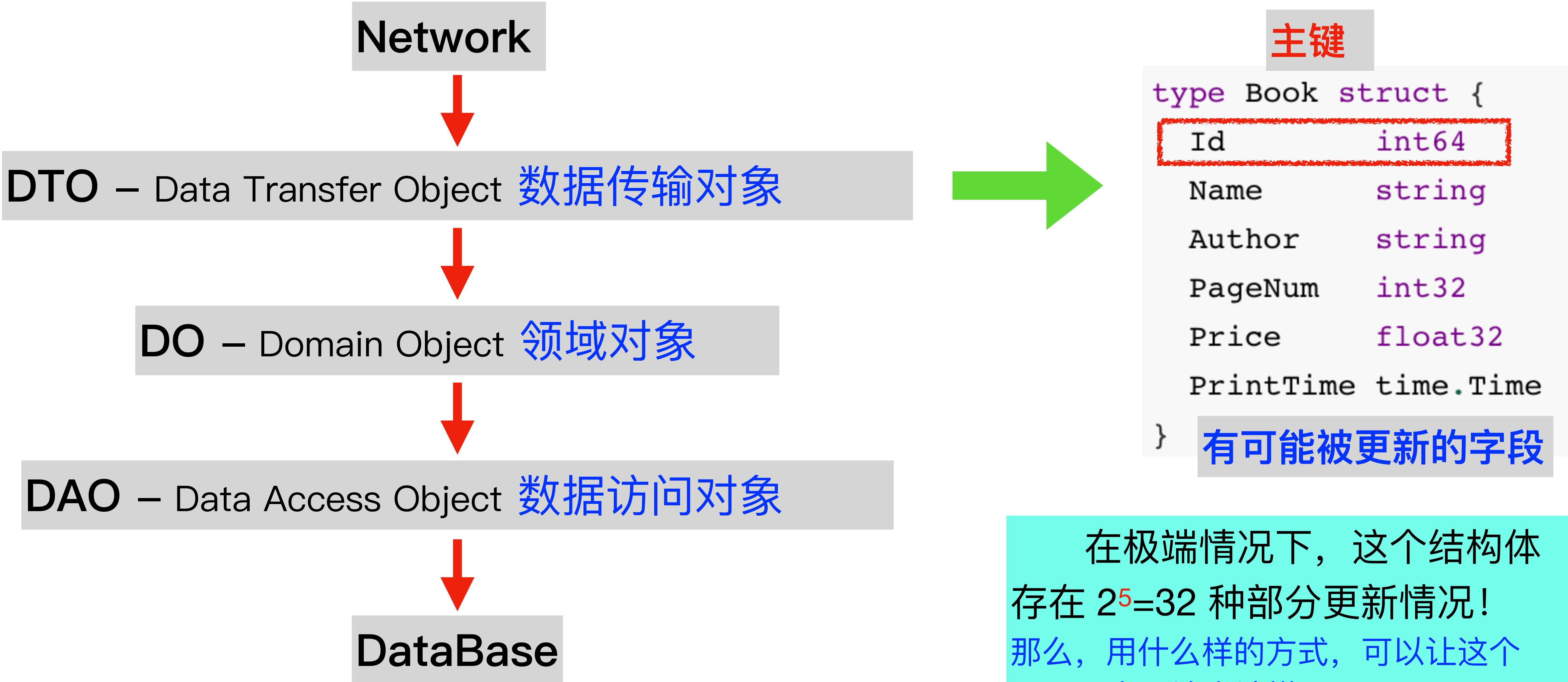
CONTENTS

课程目录

1. 分层下的Error Handling
2. 处理错误的三种方式
3. 类型别名
4. 对象的部分更新
5. 小结

1.更新部分字段的请求

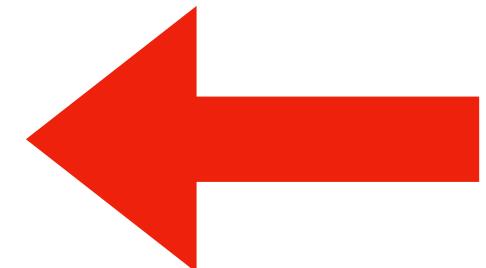
部分更新



2.方案A-定义专用的部分更新结构体

部分更新

```
type UpdateBook1 struct {  
    Id      int64  
    Name    string  
    Author  string  
}
```



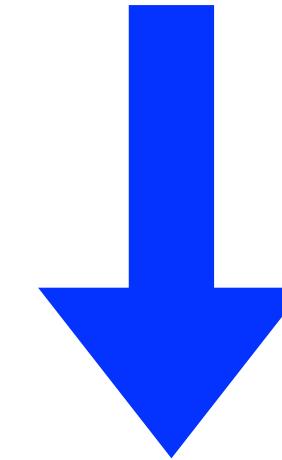
可读性大大提高

每个部分更新的请求，都需要定义：

1个DTO

1个服务端的handler

```
type UpdateBook2 struct {  
    Id      int64  
    Name    string  
    PageNum int32  
    Price   float32  
}
```



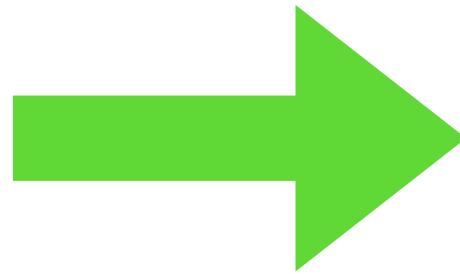
```
http.Handle("/book1", UpdateBook1)  
http.Handle("/book2", UpdateBook2)
```

3.方案B-引入标志位

部分更新

```
func update(b *Book) {  
    if b.Name == "" {  
        // update name  
    }  
    if b.PageNum == 0 {  
        // update page num  
    }  
}
```

改进



```
type BookWithFlags struct {  
    Id int64  
    Name string  
    NameUpdated bool  
    Author string  
    AuthorUpdated bool  
    PageNum int32  
    PageNumUpdated bool  
    Price float32  
    PriceUpdated bool  
    PrintTime time.Time  
    PrintTimeUpdated bool  
}
```

Field为默认值时不更新

如果希望将字段修改为默认值，这里就会出现问题

Flag位为true时才更新

一个DTO将手动增加大量的标志位，整个结构体复杂度大幅度提升

4. 方案C-两种方案的结合

部分更新

方案A – 新增结构体

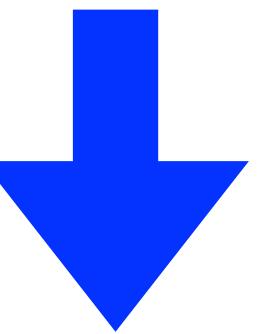
结构体简洁，可读性高
扩展性差



方案B – 新增Field

扩展性高，所有的更新复用一个结构体
可读性差

Tradeoff



方案C – 新增结构体+Field的组合

根据具体场景，在可读性和扩展性中进行平衡

Update的掩码字段

底层用一个`string`保存需要更新的Field

```
message DemoBook {
    int64 id = 1;
    string name = 2;
    int32 page_num = 3;
    google.protobuf.FieldMask update_mask = 5;
}

message FieldMask {
    repeated string paths = 1;
}
```



```
type GormBook struct {
    Id      int64 `gorm:"id"`
    Name   string `gorm:"name"`
    PageNum int32 `gorm:"page_num"`
}
```

```
var src = &DemoBook{
    Id:      1,
    Name:   "books",
    PageNum: 1,
    UpdateMask: &fieldmaskpb.FieldMask{
        Paths: []string{"id", "name"},
    },
}

var target = new(GormBook)
```

1. 获取src中的mask
2. 根据mask从src拷贝到target

```
mask, err := fieldmask_utils.MaskFromPaths(src.UpdateMask.Paths, generator.CamelCase)
// handle error here

fieldmask_utils.StructToStruct(mask, src, target)
// target &{Id:1 Name:books PageNum:0}
```

核心思想与方案二 – 标志位一致：

将多个字段的标志位合并到了一处，形成掩码字段

具体说明：<https://cloud.google.com/dialogflow/es/docs/how/field-mask?hl=zh-cn#go>

field_mask的一个示例工具库：<https://github.com/mennanov/fieldmask-utils>

更多关于设计API的技巧可参考：谷歌API设计指南

https://cloud.google.com/apis/design/standard_fields

https://cloud.google.com/apis/design/naming_convention

CONTENTS

课程目录

1. 分层下的Error Handling
2. 处理错误的三种方式
3. 类型别名
4. 字段的部分更新
5. 小结

At the end of the day, our job is to **keep agility and stability in balance** (维持灵活性和稳定性的平衡) in the system.

Essential complexity (必要复杂度) is the complexity inherent in a given situation that cannot be removed from a problem definition, whereas **accidental complexity** (意外复杂度) is more fluid and can be resolved with engineering effort.

<https://sre.google/sre-book/simplicity/>

Don't be clever.

<https://golang.org/ref/mem>

Thank you !

CONTENTS

课程目录

1. 分层下的Error Handling
2. 处理错误的三种方式
3. 类型别名
4. 字段的部分更新
5. 结束
6. 扩展

软件工程是一个包含大量“社会学”的问题，交流是达成共识的重要环节。

本次分享和基础原理课不同，很多问题背后并没有一个固定答案，也并不是非黑即白，更多的是向**当下Best Practice**共同摸索前进的过程。所以，我的分享将抛出不少主观看法，希望能引起大家的思考，也欢迎阶段性地打断交流。

同时，为了**不要过于主观而将问题发散**，我会在边角给出一些参考资料，尽量做到“有迹可循”。

1. 你更喜欢哪种调用方式?

Go工程化实践

github.com/elastic/go-elasticsearch

```
var buf bytes.Buffer
query := map[string]interface{}{
    "query": map[string]interface{}{
        "match": map[string]interface{}{
            "title": "test",
        },
    },
}
```

github.com/olivere/elastic

```
termQuery := elastic.NewTermQuery("user", "olivere")
searchResult, err := client.Search().
    Index("twitter").           // search in index "twitter"
    Query(termQuery).          // specify the query
    Sort("user", true).         // sort by "user" field, ascending
    From(0).Size(10).           // take documents 0-9
    Pretty(true).               // pretty print request and response JSON
    Do(context.Background()) // execute
```

自行填充关键词

一旦出现关键词的不匹配，会导致整块代码的不可用，但调用方只能通过运行时检查问题

(实现相对简单，不需要定义大量函数)

用方法名明确关键词

类似于GORM的调用方法，Coding时就能得知信息，可读性提升，也避免了关键词出错

(将工作量放在了sdk库的函数定义中)

2.对Go工程目录的思考

Go Standard Project Layout 通过 **对目录结构的功能约定，提高项目整体的可读性。**

我们可以适当借鉴，尝试对团队中的项目进行进一步的约定，例如：

- ◆ 团队内部怎么定义 **controller – model – dao** 三层的边界？并不断迭代对此的理解，让开发者有约定可遵循
- ◆ 哪些内容需要被放到internal目录中？防止服务特有的代码被外部项目调用
- ◆ common包的存在是否合理？如util迁移到专门的common-lib库，consts迁移
到业务代码最近的地方

抛开技术的局限性，**Go-Standards-Project-Layout** 这种 monolithic repository 的落地，能给开发人员带来什么帮助？

请尝试列举三点

Google的实践：<https://dl.acm.org/doi/fullHtml/10.1145/2854146>

4.几个Go语法的小特性

interface的浅封装

```
// private
var tigerPool sync.Pool

func AddTiger(tiger *Tiger) {
    tigerPool.Put(tiger)
}

func GetTiger() *Tiger {
    t := tigerPool.Get()
    tiger, _ := t.(*Tiger)
    return tiger
}
```

返回值的二元组

```
func safeMap(key int) string {
    var m = make(map[int]string)
    value, _ := m[key]
    return value
}

func notSafeMap(key int) string {
    var m = make(map[int]string)
    return m[key]
}
```

Val,ok := map[key]
 Val,ok := <- chan
 Val,ok := data.(type)

One-way channel

```
func caller() {
    var ch chan int
    go channelSend(ch)
    go channelReceive(ch)
}

// send only
func channelSend(ch chan<- int) {
    ch <- 1
    return
}

// receive only
func channelReceive(ch <-chan int) {
    val, ok := <-ch
    if !ok {
        os.Exit( code: 0)
    }
    fmt.Println(val)
    return
}
```

管理Goroutines

```
func SafeGo(f func()) {
    defer func() {
        if err := recover(); err != nil {
            stack := make([]byte, 4<<10)
            runtime.Stack(stack, all: true)
            // record log here
        }
    }()
    f()
}
```

防止Goroutine的panic非常重要，尽可能做一层封装

延伸思考：

1. 控制某些比较耗资源的Goroutine的并发数
2. 多并发情况下的error汇聚处理，参考<https://pkg.go.dev/golang.org/x/sync/errgroup>
3. 基于errgroup的二次扩展，具备业务特性，如服务降级、异常重试

6. interface的背后

```
switch t.Kind() {
case reflect.Bool:
    return boolEncoder
case reflect.Int, reflect.Int8, reflect.Int16:
    return intEncoder
case reflect.Uint, reflect.Uint8, reflect.Uint16:
    return uintEncoder
case reflect.Float32:
    return float32Encoder
case reflect.Float64:
    return float64Encoder
case reflect.String:
    return stringEncoder
case reflect.Interface:
    return interfaceEncoder
case reflect.Struct:
    return newStructEncoder(t)
case reflect.Map:
    return newMapEncoder(t)
case reflect.Slice:
    return newSliceEncoder(t)
case reflect.Array:
    return newArrayEncoder(t)
case reflect.Ptr:
```

```
func print(vals []interface{}) {
    for _, val := range vals {
        fmt.Println(val)
    }
}

func main() {
    names := []string{"stanley", "david", "oscar"}
    print(names)
}
```

Cannot use 'names' (type []string) as type []interface{} :

当前的Go语言中，不带任何方法的interface{}是最接近“泛型”概念的一个数据结构：类型为interface{}的参数，可以传递任意类型。但在这个背后，有两个有意思的特点：

- 分析Go标准库，interface{}相关的代码里大量使用了reflect（如encoding/json，见左图）
- 如上图所示，[]interface{}却无法被[]string等具体数据结构直接调用，而需要先转化为[]interface{}

那么，我很自然就有了一个念头：

interface{}往往是一个低效的实现？

7. embedded特性的思考

Go工程化实践

Embedded可以实现代码复用，但也很容易引发一些因Overwrite发生的奇怪现象

```
type Animal struct{}

func (a *Animal) Name() string {
    return "Animal"
}

type Cat struct {
    Animal
}

// Overwrite
func (a *Cat) Name() string {
    return "Cat"
}
```

```
type Animal struct {
    Name string `json:"name"`
}

type Dog struct {
    Name string `json:"name"`
    Animal
}

func main() {
    var dog = Dog{
        Name: "dog",
        Animal: Animal{
            Name: "animal",
        },
    }
    b, _ := json.Marshal(dog)
    fmt.Println(string(b))
}
```

如果出现比较多的Overwrite,
建议定义成**Named Field**, 显
式地进行调用

一个示例

```
// 一次旅游
type Tour struct {
    TourId int64
    StartTime time.Time
    EndTime time.Time
}
func (t *Tour) Duration() time.Duration {
    return t.EndTime.Sub(t.StartTime)
}
```

```
// 游客
type Tourist struct {
    name string
}
func (t *Tourist) Name() string {
    return t.name
}
```

```
// 一次动物园的旅游
type ZooTour struct {
    // Good
    // Tourist
    // Bad
    // Tourist
    ZooName string
    Animals []string
}
```

```
zt := new(ZooTour)
// 语义清晰, Duration函数的定义一致
zt.Duration()
// 语义混乱, 为什么旅游的name方法, 返回了游客的名字?
zt.Name()
```

Embed的对象应选择精简、通用的基础对象，**语义与当前对象保持一致**（或者说父类）

或者是类似于**sync.Mutex**这种用途明确的功能组件

https://golang.org/doc/effective_go#embedding

有些数据不适合放在业务对象中，更应该显示传递或作为全局变量。

你认同这个观点吗？

显示传递：context、error ...

全局变量：log、db、redis、httpClient ...

静态检查: **go vet**

```
func Copy(m sync.Mutex) {}  
  
func main() {  
    var m sync.Mutex  
  
    Copy(m)  
}
```

warning

```
func Copy(wg sync.WaitGroup) {}  
  
func main() {  
    var wg sync.WaitGroup  
  
    Copy(wg)  
}
```

warning

测试环境: **data race**

```
type noCopy struct{  
  
}  
  
func (*noCopy) Lock() {}  
func (*noCopy) Unlock() {}  
  
type Demo struct {  
    noCopy noCopy  
}  
  
func Copy(d Demo) {}  
  
func main() {  
    d := Demo{}  
  
    Copy(d)  
}
```

warning

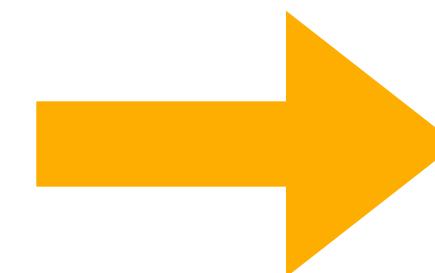
单元测试
go test -race

测试环境编译的二进制文件
go build -race

填充字段

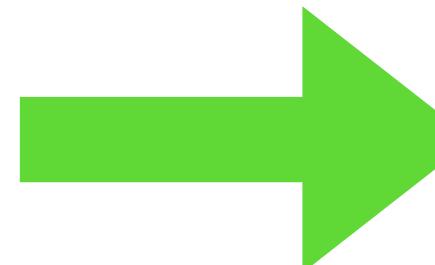
```
// 动物信息
type Animal struct {
    Id          int
    Name        string
    Age         int
    Color       string
    Foods       []string
    DeleteStatus int // 1 - 未删除, 2 已删除
    Owner       *OwnerInfo
}

// 主人信息
type OwnerInfo struct {
    Name string
}
```



```
// 一长段初始化
dog = new(Animal)
dog.Id = 1
dog.Name = ""
dog.Age = 0
dog.Color = "Yellow"

// panic
fmt.Println(dog.Owner.Name)
```



```
var dog = &Animal{
```

⋮

- Add parens to declaration ➤
- Fill all fields ➤
- Fill all fields recursively ➤**
- Specify type explicitly ➤
- Fill fields... ➤

Press ⌘Space to open preview

避免Owner为nil的panic

```
var dog = &Animal{
    Id:          0,
    Name:        "",
    Age:         0,
    Color:       "",
    Foods:       nil,
    DeleteStatus: 0,
    Owner: &OwnerInfo{
        Name: "",
    },
}
```

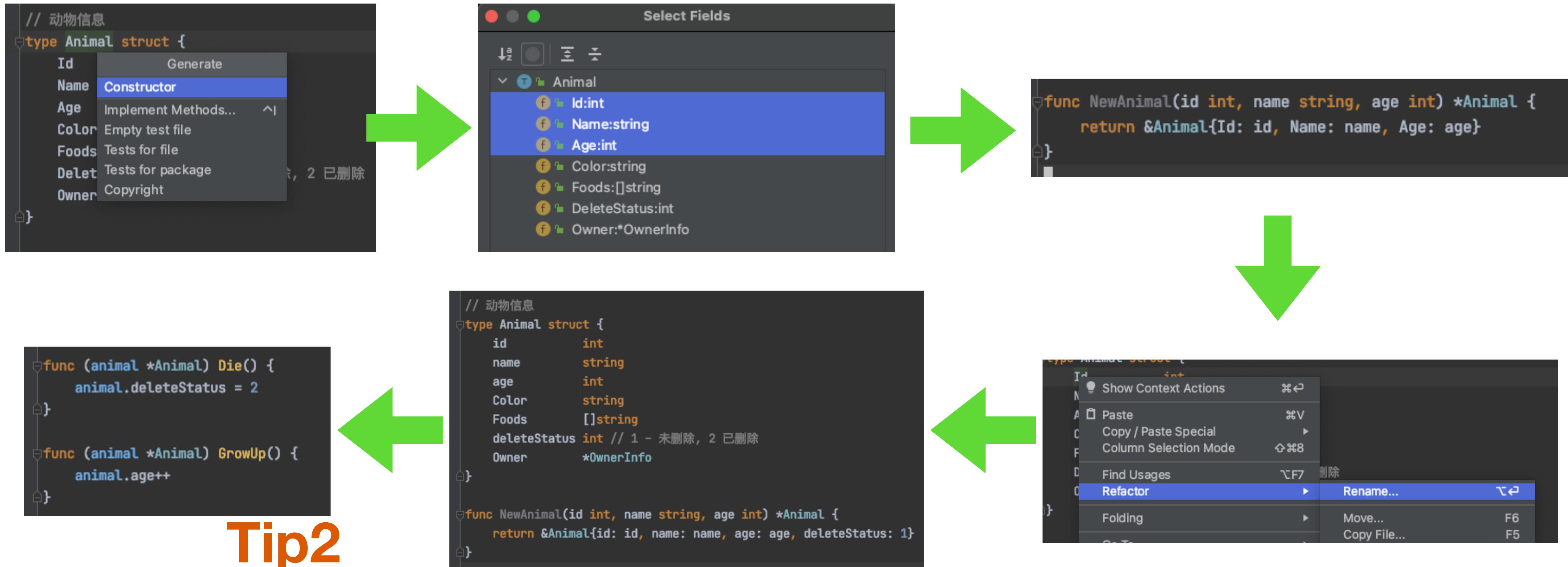
Tip1

Fill all fields 填充所有field

Fill all fields recursively 递归地填充所有field

在一个对象初始化时，声明的所有field的值（即便是默认值也进行赋值），能规避复杂结构体某个字段忘记初始化引入的各种问题，可读性也大大提高。

构造与Rename



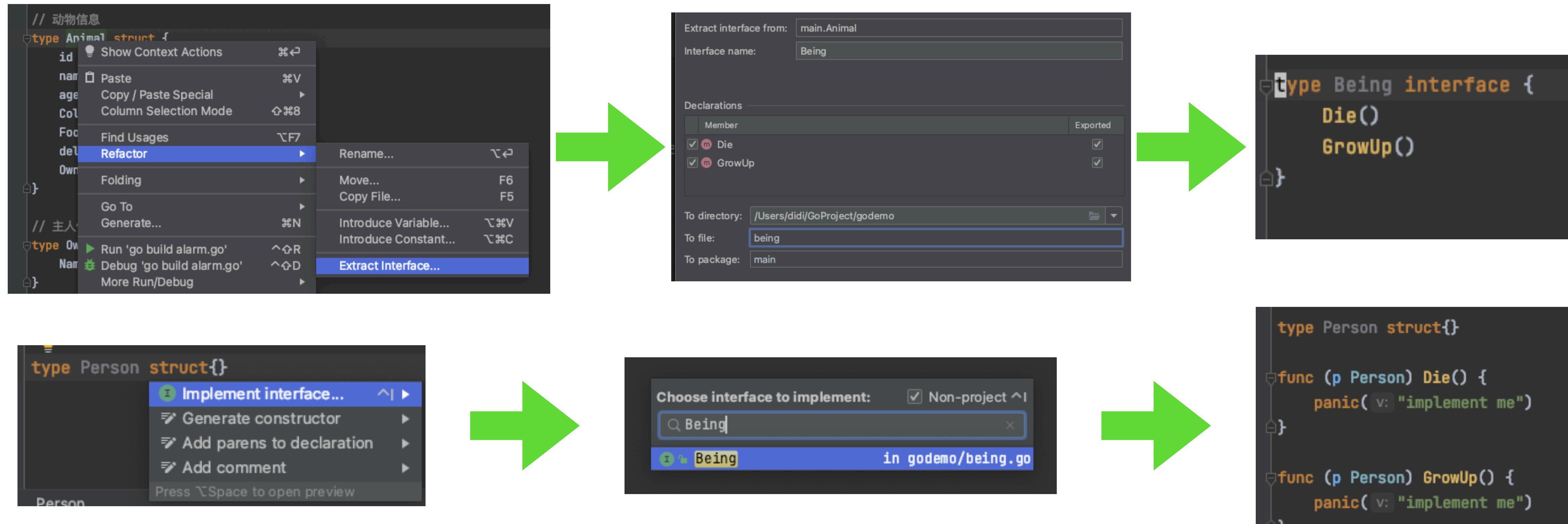
Tip2

Constructor 指定一个对象初始化的必填字段

Refactor - Rename 重命名，不仅提高变量的可读性，也通过大小写做好对象的封装

通过构造函数与关键Field的私有化，我们能有效地控制结构体的对外封装

Interface的抽象与实现



Tip3

Extract Interface 快速地从现有的一个实现中抽象出对外接口

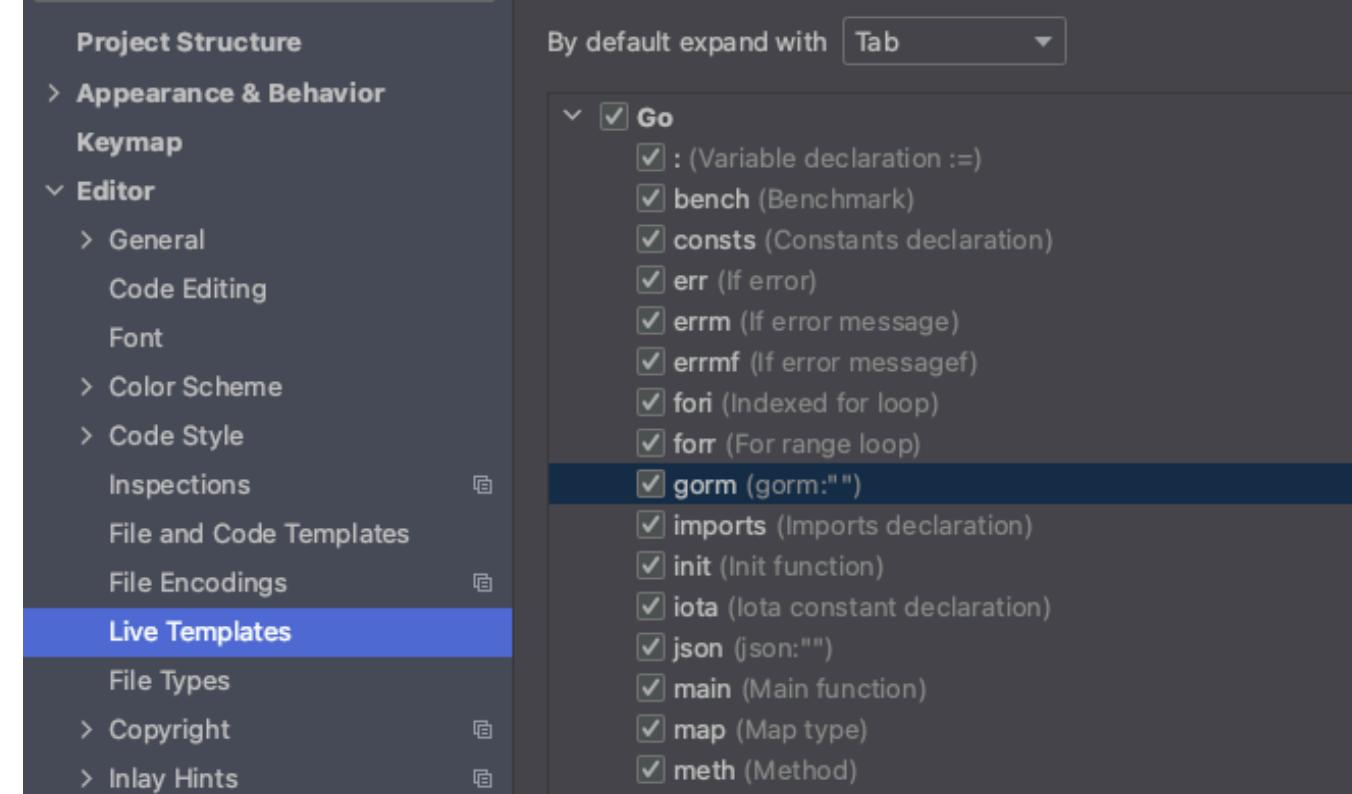
Implement Interface 让结构体实现某个Interface的所有方法

抽象Interface也是一个清洗Public方法的过程，将无需对外的方法尽量私有化

抽离变量到常量

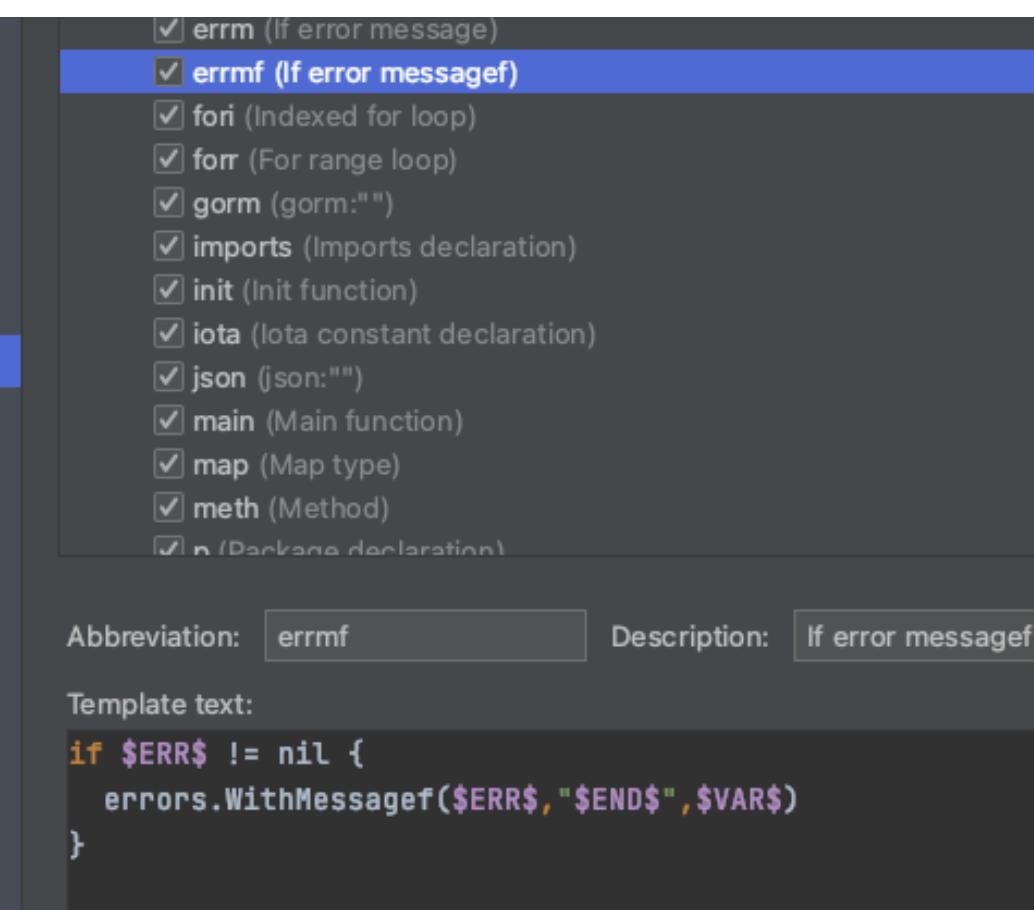


Live Templates



```
// 动物信息
type animal struct {
    id           int
    name         string
    age          int
    Color        string `gorm:"g"
    Foods        []string `gorm
    deleteStatus status   Press ⌘ to inser
    Owner        *OwnerInfo
}
```

```
// 动物信息
type animal struct {
    id           int
    name         string
    age          int
    Color        string `gorm:"color"
    Foods        []string
    deleteStatus status // 1 - 未删除, 2
    Owner        *OwnerInfo
}
```



Tip5

errmf
errmf
http://errMissingFile.net/http
...
if err != nil {
 errors.WithMessagef(err, format: "%")
}

Live Templates 高效的自动化Coding工具

将自己的开发经验沉淀成可自动化、可复用的形式，尽可能避免人工失误带来的问题

Go的标准库设计得很简洁，但相应的，有时候我们使用起来会觉得“不方便”

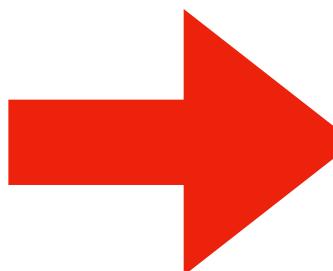
```
package util

func IsInArray(val string, arrays []string) bool {
    for _, v := range arrays {
        if val == v {
            return true
        }
    }
    return false
}
```

从[]string中查询string

```
exist := util.IsInArray( val: "dog", []string{"cat", "dog"})
fmt.Println(exist)
```

看过去一切都很美好，但是...

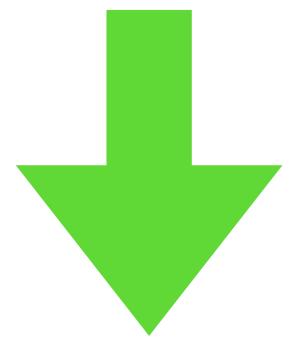


屏蔽了复杂度

```
var animals = []string{"dog1", "dog2", "dog3"}  
var allAnimals = []string{"cat", "dog"}  
  
// O(n) * O(m)  
for _, v := range animals {  
    util.IsAnyArray(v, allAnimals)  
}
```

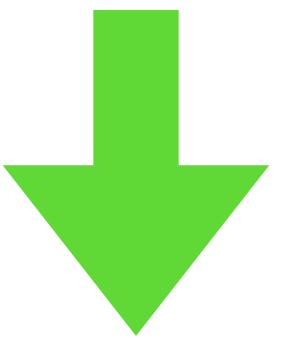
Reflect损失性能

```
func IsInArray(val interface{}, arrays []interface{}) bool {  
    for _, v := range arrays {  
        if reflect.DeepEqual(val, v) {  
            return true  
        }  
    }  
    return false  
}
```



不封装成函数反而更适合阅读，能提早发现问题

1. 坐等Go2的泛型
2. Go Generate <https://github.com/cheekybits/genny>



1. Go官方对error的未来初步设计中，下面哪项是正确的描述？
 - a. 用reflect提取底层实现，抽出想要的字段信息
 - b. 根据项目，定义并统一error对应的实现结构体
 - c. 用栈信息保存错误的上下文，方便排查问题
 - d. 维持现状，不作改动

2. 关于堆栈化的error，下列哪项的描述是不正确的？
- a. 多行的堆栈化打印，可能对日志采集的解析带来困扰
 - b. 堆栈化后的错误，在分层架构中逐层打印错误，能提高排查问题的效率
 - c. 堆栈化后的错误内容，很难解析成k-v格式
 - d. 堆栈化后的错误日志量暴增，对磁盘、日志工具是一个考验

3. go的interface包括两块：一个是面向对象中的接口(iface, 带方法)，另一个是数据类型中的“泛型”(eface, 不带方法)，下面说法不正确的是？
- a. iface和eface的底层数据结构不一致
 - b. iface是很好的编程实践，尤其在依赖注入的场景
 - c. eface作为函数的入参类型，对调用方来说非常灵活，但使用方不太方便，往往需要解析到具体类型后再使用
 - d. eface作为函数的返回类型，让函数能返回多种数据结构，对调用方和使用方都很方便

4. Google推荐的“巨型代码仓库”，也就是将多个应用的代码维护在一个代码仓库中。下面哪项不是其优点：
- a. 代码统一在一处，版本管理、依赖管理更简单
 - b. 方便管理用户权限
 - c. 更方便地进行大规模重构、升级
 - d. 有利于跨团队的代码复用

5. Go语言里面有一些很酷的特性，但在实际工程中，使用不当会引发一些奇怪的问题。下面四种情况中，哪个会破坏数据的一致性？
- a. 使用defer语句，保证对关键数据的处理，如锁的释放或文件的清理
 - b. buffered channel（即channel设置了容量）在程序重启前，保证处理完毕
 - c. 多goroutine处理数据，用mutex或atomic等机制保护临界区
 - d. 在数据处理过程中发生panic，保证recover，无需关注内部数据的变化