



下载APP



## 19 | 错误处理（下）：如何设计错误包？

2021-07-08 孔令飞

《Go 语言项目开发实战》

课程介绍 >



讲述：孔令飞

时长 17:28 大小 16.00M



你好，我是孔令飞。

在 Go 项目开发中，错误是我们必须要处理的一个事项。除了我们上一讲学习过的错误码，处理错误也离不开错误包。

业界有很多优秀的、开源的错误包可供选择，例如 Go 标准库自带的errors包、github.com/pkg/errors包。但是这些包目前还不支持业务错误码，很难满足生产级应用的需求。所以，在实际开发中，我们有必要开发出适合自己错误码设计的错误包。当然，我们也没必要自己从 0 开发，可以基于一些优秀的包来进行二次封装。



这一讲里，我们就来一起看看，如何设计一个错误包来适配上一讲我们设计的错误码，以及一个错误码的具体实现。

## 错误包需要具有哪些功能？

要想设计一个优秀的错误包，我们首先得知道一个优秀的错误包需要具备哪些功能。在我看来，至少需要有下面这六个功能：

**首先，应该能支持错误堆栈。**我们来看下面一段代码，假设保存在 [bad.go](#) 文件中：

[复制代码](#)


```
1 package main
2
3 import (
4     "fmt"
5     "log"
6 )
7
8 func main() {
9     if err := funcA(); err != nil {
10         log.Fatalf("call func got failed: %v", err)
11         return
12     }
13
14     log.Println("call func success")
15 }
16
17 func funcA() error {
18     if err := funcB(); err != nil {
19         return err
20     }
21
22     return fmt.Errorf("func called error")
23 }
24
25 func funcB() error {
26     return fmt.Errorf("func called error")
27 }
```

执行上面的代码：

[复制代码](#)

```
1 $ go run bad.go
2 2021/07/02 08:06:55 call func got failed: func called error
3 exit status 1
```

这时我们想定位问题，但不知道具体是哪行代码报的错误，只能靠猜，还不一定能猜到。为了解决这个问题，我们可以加一些 Debug 信息，来协助我们定位问题。这样做在测试环境是没问题的，但是在线上环境，一方面修改、发布都比较麻烦，另一方面问题可能比较难重现。这时候我们会想，要是能打印错误的堆栈就好了。例如：


 复制代码

```
1 2021/07/02 14:17:03 call func got failed: func called error
2 main.funcB
3 /home/colin/workspace/golang/src/github.com/marmotedu/gopractise-demo/errors
4 main.funcA
5 /home/colin/workspace/golang/src/github.com/marmotedu/gopractise-demo/errors
6 main.main
7 /home/colin/workspace/golang/src/github.com/marmotedu/gopractise-demo/errors
8 runtime.main
9 /home/colin/go/go1.16.2/src/runtime/proc.go:225
10 runtime.goexit
11 /home/colin/go/go1.16.2/src/runtime/asm_amd64.s:1371
12 exit status 1
```

通过上面的错误输出，我们可以很容易地知道是哪行代码报的错，从而极大提高问题定位的效率，降低定位的难度。所以，在我看来，一个优秀的 errors 包，首先需要支持错误堆栈。

**其次，能够支持不同的打印格式。**例如%+v、%v、%s等格式，可以根据需要打印不同丰富度的错误信息。

**再次，能支持 Wrap/Unwrap 功能，也就是在已有的错误上，追加一些新的信息。**例如 errors.Wrap(err, "open file failed")。Wrap 通常用在调用函数中，调用函数可以基于被调函数报错时的错误 Wrap 一些自己的信息，丰富报错信息，方便后期的错误定位，例如：

 复制代码

```
1 func funcA() error {
2     if err := funcB(); err != nil {
3         return errors.Wrap(err, "call funcB failed")
4     }
5
6     return errors.New("func called error")
7 }
8
```

```
9 func funcB() error {
10     return errors.New("func called error")
11 }
```

这里要注意，不同的错误类型，Wrap 函数的逻辑也可以不同。另外，在调用 Wrap 时，也会生成一个错误堆栈节点。我们既然能够嵌套 error，那有时候还可能需要获取被嵌套的 error，这时就需要错误包提供Unwrap函数。

**还有，错误包应该有Is方法。**在实际开发中，我们经常需要判断某个 error 是否是指定的 error。在 Go 1.13 之前，也就是没有 wrapping error 的时候，我们要判断 error 是不是同一个，可以使用如下方法：

```
1 if err == os.ErrNotExist {
2     // normal code
3 }
```

[复制代码](#)

但是现在，因为有了 wrapping error，这样判断就会有问题。因为你根本不知道返回的 err 是不是一个嵌套的 error，嵌套了几层。这种情况下，我们的错误包就需要提供Is函数：

```
1 func Is(err, target error) bool
```

[复制代码](#)

当 err 和 target 是同一个，或者 err 是一个 wrapping error 的时候，如果 target 也包含在这个嵌套 error 链中，返回 true，否则返回 false。

**另外，错误包应该支持 As 函数。**


在 Go 1.13 之前，没有 wrapping error 的时候，我们要把 error 转为另外一个 error，一般都是使用 type assertion 或者 type switch，也就是类型断言。例如：

```
1 if perr, ok := err.(*os.PathError); ok {
2     fmt.Println(perr.Path)
```

[复制代码](#)

```
3 }
```


但是现在，返回的 `err` 可能是嵌套的 `error`，甚至好几层嵌套，这种方式就不能用了。所以，我们可以通过实现 `As` 函数来完成这种功能。现在我们把上面的例子，用 `As` 函数实现一下：

 复制代码

```
1 var perr *os.PathError
2 if errors.As(err, &perr) {
3     fmt.Println(perr.Path)
4 }
```

这样就可以完全实现类型断言的功能，而且还更强大，因为它可以处理 `wrapping error`。

**最后，能够支持两种错误创建方式：非格式化创建和格式化创建。**例如：

 复制代码


```
1 errors.New("file not found")
2 errors.Errorf("file %s not found", "iam-apiserver")
```

上面，我们介绍了一个优秀的错误包应该具备的功能。一个好消息是，Github 上有不少实现了这些功能的错误包，其中 [github.com/pkg/errors](https://github.com/pkg/errors) 包最受欢迎。所以，我基于 [github.com/pkg/errors](https://github.com/pkg/errors) 包进行了二次封装，用来支持上一讲所介绍的错误码。

## 错误包实现

明确优秀的错误包应该具备的功能后，我们来看下错误包的实现。实现的源码存放在 [github.com/marmotedu/errors](https://github.com/marmotedu/errors)。


我通过在文件 [github.com/pkg/errors/errors.go](https://github.com/pkg/errors/errors.go) 中增加新的 `withCode` 结构体，来引入一种新的错误类型，该错误类型可以记录错误码、`stack`、`cause` 和具体的错误信息。

 复制代码

```
1 type withCode struct {
2     err    error // error 错误
3     code   int  // 业务错误码
```

```
4     cause error // cause error
5     *stack // 错误堆栈
6 }
```

下面，我们通过一个示例，来了解下github.com/marmotedu/errors所提供的功能。假设下述代码保存在errors.go文件中：

 复制代码

```
1 package main
2
3 import (
4     "fmt"
5
6     "github.com/marmotedu/errors"
7     code "github.com/marmotedu/sample-code"
8 )
9
10 func main() {
11     if err := bindUser(); err != nil {
12         // %s: Returns the user-safe error string mapped to the error code or the
13         fmt.Println("=====> %s <=====")
14         fmt.Printf("%s\n\n", err)
15
16         // %v: Alias for %s.
17         fmt.Println("=====> %v <=====")
18         fmt.Printf("%v\n\n", err)
19
20         // %-v: Output caller details, useful for troubleshooting.
21         fmt.Println("=====> %-v <=====")
22         fmt.Printf("%-v\n\n", err)
23
24         // %+v: Output full error stack details, useful for debugging.
25         fmt.Println("=====> %+v <=====")
26         fmt.Printf("%+v\n\n", err)
27
28         // %#-v: Output caller details, useful for troubleshooting with JSON forma
29         fmt.Println("=====> %#-v <=====")
30         fmt.Printf("%#-v\n\n", err)
31
32         // %#+v: Output full error stack details, useful for debugging with JSON f
33         fmt.Println("=====> %#+v <=====")
34         fmt.Printf("%#+v\n\n", err)
35
36         // do some business process based on the error type
37         if errors.IsCode(err, code.ErrEncodingFailed) {
38             fmt.Println("this is a ErrEncodingFailed error")
39         }
40     }
```

```
41     if errors.IsCode(err, code.ErrDatabase) {
42         fmt.Println("this is a ErrDatabase error")
43     }
44
45     // we can also find the cause error
46     fmt.Println(errors.Cause(err))
47 }
48 }
49
50 func bindUser() error {
51     if err := getUser(); err != nil {
52         // Step3: Wrap the error with a new error message and a new error code if
53         return errors.WrapC(err, code.ErrEncodingFailed, "encoding user 'Lingfei K
54     }
55
56     return nil
57 }
58
59 func getUser() error {
60     if err := queryDatabase(); err != nil {
61         // Step2: Wrap the error with a new error message.
62         return errors.Wrap(err, "get user failed.")
63     }
64
65     return nil
66 }
67
68 func queryDatabase() error {
69     // Step1. Create error with specified error code.
70     return errors.WithCode(code.ErrDatabase, "user 'Lingfei Kong' not found.")
71 }
```

上述代码中，通过 [WrapC](#) 函数来创建新的 withCode 类型的错误；通过 [WrapC](#) 来将一个 error 封装成一个 withCode 类型的错误；通过 [IsCode](#) 来判断一个 error 链中是否包含指定的 code。

withCode 错误实现了一个 `func (w *withCode) Format(state fmt.State, verb rune)` 方法，该方法用来打印不同格式的错误信息，见下表：



格式占位符	格式描述
%s	返回可以直接展示给用户的错误信息
%v	alias for %s
%-v	打印出调用栈、错误码、展示给用户的错误信息、展示给研发的错误信息（只展示错误链中最后一个错误）
%+v	打印出调用栈、错误码、展示给用户的错误信息、展示给研发的错误信息（展示错误链中的所有错误）
%#-v	JSON格式打印出调用栈、错误码、展示给用户的错误信息、展示给研发的错误信息（只展示错误链中最后一个错误）
%#+v	JSON格式打印出调用栈、错误码、展示给用户的错误信息、展示给研发的错误信息（展示错误链中的所有错误）

例如，%+v会打印以下错误信息：

复制代码

```
1 get user failed. - #1 [/home/colin/workspace/golang/src/github.com/marmotedu/g
```

那么你可能会问，这些错误信息中的100101错误码，还有Database error这种对外展示的报错信息等等，是从哪里获取的？这里我简单解释一下。

首先，withCode 中包含了int 类型的错误码，例如100101。


其次，当使用github.com/marmotedu/errors包的时候，需要调用Register或者MustRegister，将一个Coder 注册到github.com/marmotedu/errors开辟的内存中，数据结构为：

复制代码

```
1 var codes = map[int]Coder{}
```



Coder 是一个接口，定义为：

 复制代码

```
1 type Coder interface {  
2     // HTTP status that should be used for the associated error code.  
3     HTTPStatus() int  
4  
5     // External (user) facing error text.  
6     String() string  
7  
8     // Reference returns the detail documents for user.  
9     Reference() string  
10  
11    // Code returns the code of the coder  
12    Code() int  
13 }
```

这样 withCode 的 Format 方法，就能够通过 withCode 中的 code 字段获取到对应的 Coder，并通过 Coder 提供的 HTTPStatus、String、Reference、Code 函数，来获取 withCode 中 code 的详细信息，最后格式化打印。


这里要注意，我们实现了两个注册函数：Register 和 MustRegister，二者唯一区别是：当重复定义同一个错误 Code 时，MustRegister 会 panic，这样可以防止后面注册的错误覆盖掉之前注册的错误。在实际开发中，建议使用 MustRegister。

XXX() 和 MustXXX() 的函数命名方式，是一种 Go 代码设计技巧，在 Go 代码中经常使用，例如 Go 标准库中 regexp 包提供的 Compile 和 MustCompile 函数。和 XXX 相比，MustXXX 会在某种情况不满足时 panic。因此使用 MustXXX 的开发者看到函数名就会有一个心理预期：使用不当，会造成程序 panic。

最后，我还有一个建议：在实际的生产环境中，我们可以使用 JSON 格式打印日志，JSON 格式的日志可以非常方便的供日志系统解析。我们可以根据需要，选择 %#-v 或 %#+v 两种格式。

错误包在代码中，经常被调用，所以我们要保证错误包一定要是高性能的，否则很可能会影响接口的性能。这里，我们再来看下 [github.com/marmotedu/errors](https://github.com/marmotedu/errors) 包的性能。

在这里，我们把这个错误包跟 go 标准库的 errors 包，以及 [github.com/pkg/errors](https://github.com/pkg/errors) 包进行对比，来看看它们的性能：

 复制代码

```
1 $ go test -test.bench=BenchmarkErrors -benchtime="3s"
2 goos: linux
3 goarch: amd64
4 pkg: github.com/marmotedu/errors
5 BenchmarkErrors/errors-stack-10-8          57658672          61.8 ns/op
6 BenchmarkErrors/pkg/errors-stack-10-8      2265558          1547 ns/op      3
7 BenchmarkErrors/marmot/errors-stack-10-8   1903532          1772 ns/op      3
8 BenchmarkErrors/errors-stack-100-8         4883659          734 ns/op
9 BenchmarkErrors/pkg/errors-stack-100-8     1202797          2881 ns/op      3
10 BenchmarkErrors/marmot/errors-stack-100-8  1000000          3116 ns/op      3
11 BenchmarkErrors/errors-stack-1000-8       505636          7159 ns/op
12 BenchmarkErrors/pkg/errors-stack-1000-8   327681          10646 ns/op     3
13 BenchmarkErrors/marmot/errors-stack-1000-8 304160          11896 ns/o
14 PASS
15 ok      github.com/marmotedu/errors 39.200s
```

可以看到 [github.com/marmotedu/errors](https://github.com/marmotedu/errors) 和 [github.com/pkg/errors](https://github.com/pkg/errors) 包的性能基本持平。在对比性能时，重点关注 **ns/op**，也即每次 error 操作耗费的纳秒数。另外，我们还需要测试不同 error 嵌套深度下的 error 操作性能，嵌套越深，性能越差。例如：在嵌套深度为 10 的时候，[github.com/pkg/errors](https://github.com/pkg/errors) 包 ns/op 值为 1547，[github.com/marmotedu/errors](https://github.com/marmotedu/errors) 包 ns/op 值为 1772。可以看到，二者性能基本保持一致。

具体性能数据对比见下表：

package	depth	ns/op
github.com/pkg/errors	10	1547
github.com/marmotedu/errors	10	1772
github.com/pkg/errors	100	2881
github.com/marmotedu/errors	100	3116
github.com/pkg/errors	1000	10646
github.com/marmotedu/errors	1000	11896

我们是通过 [BenchmarkErrors](#) 测试函数来测试 error 包性能的，你感兴趣可以打开链接看看。


## 如何记录错误？

上面，我们一起看了怎么设计一个优秀的错误包，那如何用我们设计的错误包来记录错误呢？

根据我的开发经验，我推荐两种记录错误的方式，可以帮你快速定位问题。

方式一：通过 `github.com/marmotedu/errors` 包提供的错误堆栈能力，来跟踪错误。


具体你可以看看下面的代码示例。以下代码保存在 [errortrack\\_errors.go](#) 中。

 复制代码

```
1 package main
2
3 import (
4     "fmt"
5
6     "github.com/marmotedu/errors"
```

```
7     code "github.com/marmotedu/sample-code"
8 )
9
10
11 func main() {
12     if err := getUser(); err != nil {
13         fmt.Printf("%+v\n", err)
14     }
15 }
16
17 func getUser() error {
18     if err := queryDatabase(); err != nil {
19         return errors.Wrap(err, "get user failed.")
20     }
21
22     return nil
23 }
24
25 func queryDatabase() error {
26     return errors.WithCode(code.ErrDatabase, "user 'Lingfei Kong' not found.")
27 }
```

执行上述的代码：

 复制代码

```
1 $ go run errortrack_errors.go
2 get user failed. - #1 [/home/colin/workspace/golang/src/github.com/marmotedu/g
```


可以看到，打印的日志中打印出了详细的错误堆栈，包括错误发生的函数、文件名、行号和错误信息，通过这些错误堆栈，我们可以很方便地定位问题。

你使用这种方法时，我推荐的用法是，在错误最开始处使用 `errors.WithCode()` 创建一个 `withCode` 类型的错误。上层在处理底层返回的错误时，可以根据需要，使用 `Wrap` 函数基于该错误封装新的错误信息。如果要包装的 `error` 不是用 `github.com/marmotedu/errors` 包创建的，建议用 `errors.WithCode()` 新建一个 `error`。

方式二：在错误产生的最原始位置调用日志包记录函数，打印错误信息，其他位置直接返回（当然，也可以选择性的追加一些错误信息，方便故障定位）。示例代码（保存在 [errortrack\\_log.go](#)）如下：

```
1 package main
2
3 import (
4     "fmt"
5
6     "github.com/marmotedu/errors"
7     "github.com/marmotedu/log"
8
9     code "github.com/marmotedu/sample-code"
10 )
11
12 func main() {
13     if err := getUser(); err != nil {
14         fmt.Printf("%v\n", err)
15     }
16 }
17
18 func getUser() error {
19     if err := queryDatabase(); err != nil {
20         return err
21     }
22
23     return nil
24 }
25
26 func queryDatabase() error {
27     opts := &log.Options{
28         Level:      "info",
29         Format:      "console",
30         EnableColor: true,
31         EnableCaller: true,
32         OutputPaths: []string{"test.log", "stdout"},
33         ErrorOutputPaths: []string{},
34     }
35
36     log.Init(opts)
37     defer log.Flush()
38
39     err := errors.WithCode(code.ErrDatabase, "user 'Lingfei Kong' not found.")
40     if err != nil {
41         log.Errorf("%v", err)
42     }
43     return err
44 }
```

执行以上代码：


 复制代码

```
1 $ go run errortrack_log.go
2 2021-07-03 14:37:31.597 ERROR errors/errortrack_log.go:41 Database error
3 Database error
```

当错误发生时，调用 log 包打印错误。通过 log 包的 caller 功能，可以定位到 log 语句的位置，也就是定位到错误发生的位置。你使用这种方式来打印日志时，我有两个建议。

只在错误产生的最初位置打印日志，其他地方直接返回错误，一般不需要再对错误进行封装。

当代码调用第三方包的函数时，第三方包函数出错时打印错误信息。比如：

 复制代码


```
1 if err := os.Chdir("/root"); err != nil {
2     log.Errorf("change dir failed: %v", err)
3 }
```

## 一个错误码的具体实现

接下来，我们看一个依据上一讲介绍的错误码规范的具体错误码实现 [github.com/marmotedu/sample-code](https://github.com/marmotedu/sample-code)。

sample-code 实现了两类错误码，分别是通用错误码（[@sample-code/base.go](https://github.com/marmotedu/sample-code/blob/main/base.go)）和业务模块相关的错误码（[@sample-code/apiserver.go](https://github.com/marmotedu/sample-code/blob/main/apiserver.go)）。

首先，我们来看通用错误码的定义：


 复制代码

```
1 // 通用：基本错误
2 // Code must start with 1xxxxx
3 const (
4     // ErrSuccess - 200: OK.
5     ErrSuccess int = iota + 100001
6
7     // ErrUnknown - 500: Internal server error.
8     ErrUnknown
9
10    // ErrBind - 400: Error occurred while binding the request body to the str
11    ErrBind
```

```
12     // ErrValidation - 400: Validation failed.
13     ErrValidation
14
15     // ErrTokenInvalid - 401: Token invalid.
16     ErrTokenInvalid
17 )
18 )
```

在代码中，我们通常使用整型常量（`ErrSuccess`）来代替整型错误码（100001），因为使用 `ErrSuccess` 时，一看就知道它代表的错误类型，可以方便开发者使用。

错误码用来指代一个错误类型，该错误类型需要包含一些有用的信息，例如对应的 HTTP Status Code、对外展示的 Message，以及跟该错误匹配的帮助文档。所以，我们还需要实现一个 `Coder` 来承载这些信息。这里，我们定义了一个实现了 [github.com/marmotedu/errors](https://github.com/marmotedu/errors) 接口的 `ErrCode` 结构体：

 复制代码

```
1 // ErrCode implements `github.com/marmotedu/errors`.Coder interface.
2 type ErrCode struct {
3     // C refers to the code of the ErrCode.
4     C int
5
6     // HTTP status that should be used for the associated error code.
7     HTTP int
8
9     // External (user) facing error text.
10    Ext string
11
12    // Ref specify the reference document.
13    Ref string
14 }
```

可以看到 `ErrCode` 结构体包含了以下信息：

`int` 类型的业务码。

对应的 HTTP Status Code。

暴露给外部用户的消息。

错误的参考文档。



下面是一个具体的 Coder 示例：

[复制代码](#)

```
1 coder := &ErrCode{
2     C:    100001,
3     HTTP: 200,
4     Ext:   "OK",
5     Ref:   "https://github.com/marmotedu/sample-code/blob/master/README.md",
6 }
```

接下来，我们就可以调用github.com/marmotedu/errors包提供的Register或者MustRegister函数，将Coder注册到github.com/marmotedu/errors包维护的内存中。

一个项目有很多个错误码，如果每个错误码都手动调用MustRegister函数会很麻烦，这里我们通过代码自动生成的方法，来生成register函数调用：

[复制代码](#)

```
1 //go:generate codegen -type=int
2 //go:generate codegen -type=int -doc -output ./error_code_generated.md
```

//go:generate codegen -type=int 会调用 [codegen](#) 工具，生成 [sample\\_code\\_generated.go](#) 源码文件：

[复制代码](#)


```
1 func init() {
2     register(ErrSuccess, 200, "OK")
3     register(ErrUnknown, 500, "Internal server error")
4     register(ErrBind, 400, "Error occurred while binding the request body to the")
5     register(ErrValidation, 400, "Validation failed")
6     // other register function call
7 }
```

这些 [register](#) 调用放在 init 函数中，在加载程序的时候被初始化。

这里要注意，在注册的时候，我们会检查 HTTP Status Code，只允许定义 200、400、401、403、404、500 这 6 个 HTTP 错误码。这里通过程序保证了错误码是符合 HTTP Status Code 使用要求的。

`//go:generate codegen -type=int -doc -output`  
`./error_code_generated.md`会生成错误码描述文档 [error\\_code\\_generated.md](#)。当我们提供 API 文档时，也需要记着提供一份错误码描述文档，这样客户端才可以根据错误码，知道请求是否成功，以及具体发生哪类错误，好针对性地做一些逻辑处理。


`codegen`工具会根据错误码注释生成`sample_code_generated.go`和`error_code_generated.md`文件：

 复制代码

```
1 // ErrSuccess - 200: OK.  
2 ErrSuccess int = iota + 100001
```

`codegen` 工具之所以能够生成`sample_code_generated.go`和`error_code_generated.md`，是因为我们的错误码注释是有规定格式的：`// <错误码整型常量> - <对应的HTTP Status Code>: <External Message>..`

`codegen` 工具可以在 IAM 项目根目录下，执行以下命令来安装：

 复制代码


```
1 $ make tools.install.codegen
```

安装完 `codegen` 工具后，可以在 `github.com/marmotedu/sample-code` 包根目录下执行 `go generate` 命令，来生成`sample_code_generated.go`和`error_code_generated.md`。这里有个技巧需要你注意：生成的文件建议统一用`xxxx_generated.xx`来命名，这样通过 `generated`，我们就知道这个文件是代码自动生成的，有助于我们理解和使用。

在实际的开发中，我们可以将错误码独立成一个包，放在 `internal/pkg/code/` 目录下，这样可以方便整个应用调用。例如 IAM 的错误码就放在 IAM 项目根目录下的

[internal/pkg/code/](#)目录下。

我们的错误码是分服务和模块的，所以这里建议你把相同的服务放在同一个 Go 源文件中，例如 IAM 的错误码存放文件：

 复制代码

```
1 $ ls base.go apiserver.go authzserver.go
2 apiserver.go authzserver.go base.go
```

一个应用中会有多个服务，例如 IAM 应用中，就包含了 iam-apiserver、iam-authz-server、iam-pump 三个服务。这些服务有一些通用的错误码，为了便于维护，可以将这些通用的错误码统一放在 base.go 源码文件中。其他的错误码，我们可以按服务分别放在不同的文件中：iam-apiserver 服务的错误码统一放在 apiserver.go 文件中；iam-authz-server 的错误码统一存放在 authzserver.go 文件中。其他服务以此类推。

另外，同一个服务中不同模块的错误码，可以按以下格式来组织：相同模块的错误码放在同一个 const 代码块中，不同模块的错误码放在不同的 const 代码块中。每个 const 代码块的开头注释就是该模块的错误码定义。例如：


 复制代码

```
1 // iam-apiserver: user errors.
2 const (
3     // ErrUserNotFound - 404: User not found.
4     ErrUserNotFound int = iota + 110001
5
6     // ErrUserAlreadyExist - 400: User already exist.
7     ErrUserAlreadyExist
8 )
9
10 // iam-apiserver: secret errors.
11 const (
12     // ErrEncrypt - 400: Secret reach the max count.
13     ErrReachMaxCount int = iota + 110101
14
15     // ErrSecretNotFound - 404: Secret not found.
16     ErrSecretNotFound
17 )
```

最后，我们还需要将错误码定义记录在项目的文件中，供开发者查阅、遵守和使用，例如 IAM 项目的错误码定义记录文档为 [code\\_specification.md](#)。这个文档中记录了错误码说明、错误描述规范和错误记录规范等。

## 错误码实际使用方法示例

上面，我讲解了错误包和错误码的实现方式，那你一定想知道在实际开发中我们是如何使用的。这里，我就举一个在 gin web 框架中使用该错误码的例子：

 复制代码

```
1 // Response defines project response format which in marmotedu organization.
2 type Response struct {
3     Code      errors.Code `json:"code,omitempty"`
4     Message   string    `json:"message,omitempty"`
5     Reference string    `json:"reference,omitempty"`
6     Data      interface{} `json:"data,omitempty"`
7 }
8
9 // WriteResponse used to write an error and JSON data into response.
10 func WriteResponse(c *gin.Context, err error, data interface{}) {
11     if err != nil {
12         coder := errors.ParseCoder(err)
13
14         c.JSON(coder.HTTPStatus(), Response{
15             Code:      coder.Code(),
16             Message:   coder.String(),
17             Reference: coder.Reference(),
18             Data:      data,
19         })
20     }
21
22     c.JSON(http.StatusOK, Response{Data: data})
23 }
24
25 func GetUser(c *gin.Context) {
26     log.Info("get user function called.", "X-Request-Id", requestid.Get(c))
27     // Get the user by the `username` from the database.
28     user, err := store.Client().Users().Get(c.Param("username"), metav1.GetOpt
29     if err != nil {
30         core.WriteResponse(c, code.ErrUserNotFound.Error(), nil)
31         return
32     }
33
34     core.WriteResponse(c, nil, user)
35 }
```

上述代码中，通过WriteResponse统一处理错误。在 WriteResponse 函数中，如果 `err != nil`，则从 `error` 中解析出 `Coder`，并调用 `Coder` 提供的方法，获取错误相关的 `Http Status Code`、`int` 类型的业务码、暴露给用户的信息、错误的参考文档链接，并返回 `JSON` 格式的信息。如果 `err == nil` 则返回 `200` 和数据。

## 总结

记录错误是应用程序必须要做的一件事情，在实际开发中，我们通常会封装自己的错误包。一个优秀的错误包，应该能够支持错误堆栈、不同的打印格式、`Wrap/Unwrap/Is/As` 等函数，并能够支持格式化创建 `error`。

根据这些错误包设计要点，我基于 `github.com/pkg/errors` 包设计了 `IAM` 项目的错误包 `github.com/marmotedu/errors`，该包符合我们上一讲设计的错误码规范。

另外，本讲也给出了一个具体的错误码实现 `sample-code`，`sample-code` 支持业务 `Code` 码、`HTTP Status Code`、错误参考文档、可以对内对外展示不同的错误信息。

最后，因为错误码注释是有固定格式的，所以我们可以通过 `codegen` 工具解析错误码的注释，生成 `register` 函数调用和错误码文档。这种做法也体现了我一直强调的 `low code` 思想，可以提高开发效率，减少人为失误。

## 课后练习

1. 在这门课里，我们定义了 `base`、`iam-apiserver` 服务的错误码，请试着定义 `iam-authz-server` 服务的错误码，并生成错误码文档。
2. 思考下，这门课的错误包和错误码设计能否满足你当前的项目需求，如果觉得不能满足，可以在留言区分享你的看法。

欢迎你在留言区与我交流讨论，我们下一讲见。

分享给需要的人，Ta订阅后你可得 **24** 元现金奖励

👍 赞 0

💡 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 18 | 错误处理（上）：如何设计一套科学的错误码？

下一篇 20 | 日志处理（上）：如何设计日志包并记录日志？

## 更多课程推荐

# 容器实战高手课

在实战中深入理解容器技术的本质

李程远

eBay 总监级工程师  
云平台架构师



涨价倒计时 🕒

今日订阅 **¥69**，7月20日涨价至 **¥129**

## 精选留言 (7)

写留言



pedro

2021-07-08

以推荐老师的项目给组里其它小伙伴，一起学习沉淀打磨为自己的实用标准，多谢老师

作者回复: 感谢pedro支持



1



你赖东东不错嘛

2021-07-16

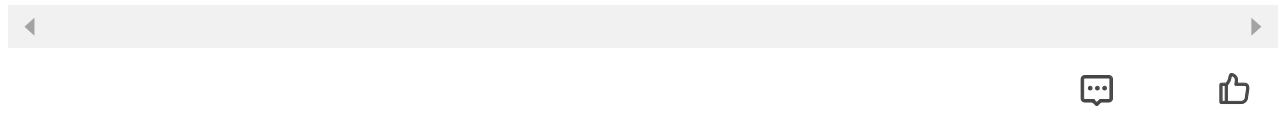
记录错误方式二：在err发生处打印了log，却依旧把err上抛，而最外层又对err进行了一次

处理，这样可能导致日志里写了两份重复的err信息。望解惑！

展开 ▾

作者回复: 这个地方可以根据需要选择是否在上层直接返回err，还是再添加一些信息。

一般情况下直接返回err即可。但可能也有些业务逻辑需要在上层追加一些额外的信息，来帮助排障。



**Realm**

2021-07-13

很强大！很实用！

展开 ▾



**8.13.3.27.30**

2021-07-12

非常实用的东西、老师讲的东西在实际应用中、真真切切的用到了、可惜当时老师没有出这个教程、现在只能下次重构的时候再使用了

展开 ▾



**Daiver** 

2021-07-12

还是不清楚，codegen是怎么生成注册代码的

展开 ▾

作者回复: codegen是我写的一个解析程序。主要是通过解析ast语法树，来获取其中的注释，然后根据注释自动生成文档和Go源文件，具体实现方式你就可以参考codegen源代码



**happychap**

2021-07-08

老师，您好，学习了您的专栏受益良多，感谢您的付出！有个问题还不太明白，烦请解答一下，就是：接口请问返回给error处理ref链接内容来源是怎么形成的或该如何维护管理它们呢？



作者回复: 内容来源，是产研总结的Q&A。链接中的文档可以放在docs/guide/zh-CN/faq/目录下。

1



helloworld



2021-07-08

优秀

展开

