【翻译】如何处理 Go 语言中的错误

作者：JoeyGaojingxing
原文链接：https://ld246.com/article/1564036181869
来源网站：链滴

如何处理Go语言中的错误

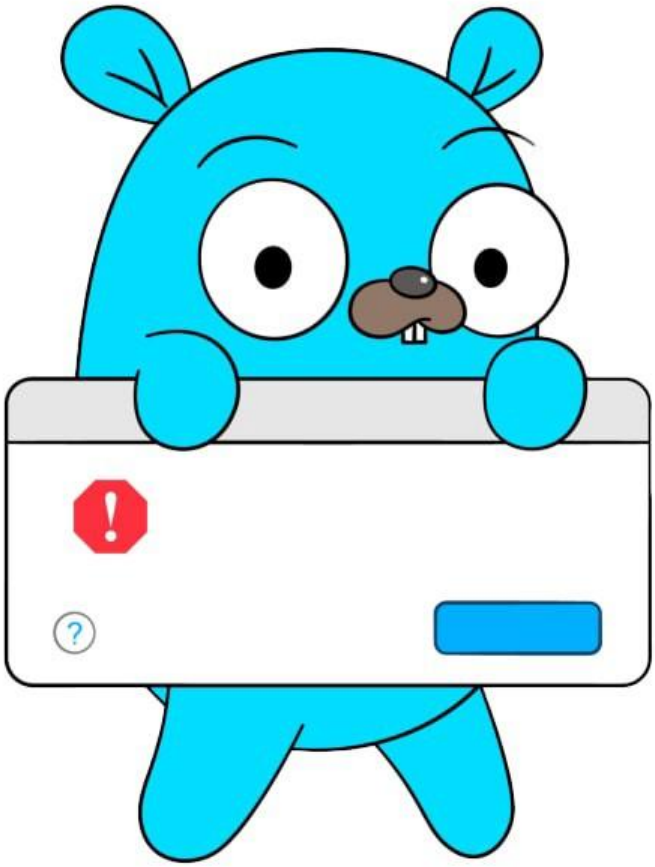
精通Go语言实用错误处理方式

这篇文章是"在你进入到Go语言的世界之前"系列中的一部分。在这里，我们可以一起探索 Golang 的界，让你了解用 Go 语言编程时应注意到的小技巧并领悟 Go 语言的特性，让你学习 Go 语言的过程再困难。

我假设你已经有了一些 Go 语言的基础，不过当你遇到文章中你不熟悉的知识点的时候，可以随时停来，查阅这些知识点之后，再回来继续读下去。

现在这些问题都讲清楚了，就让我们开始吧！

Go 语言的错误处理方法是一个一直都颇受争议或是被误用的特性。在这篇文章里，你将会学到 Go 如何处理错误的并理解他们的工作原理。你将会通过探索几种不同的方法、查看 Go 源码和一些标准的细节，去理解错误是如何产生 (**how errors work**) 的以及如何处理他们。你将会了解类型断言 (Ty e Assertions) 在处理这些错误时所扮演的重要角色,以及将会在 Go 2 中发布的一些重要的错误处理式的改变.



介绍

起始阶段(First thing's first)：Go 语言中的错误（Errors）**不**是异常（Exceptions），Dave Cheney 写了一个关于这个问题的epic blog post，我将在这里向你总结一下：在其它语言中，你无法确定一函数是否会向你抛出一个异常（Exceptions）。相比于抛出一个异常，Go 中的函数支持**返回多个值**有一个约定俗成的用法是返回这个函数的结果并伴随一个错误（error）变量。

```
func calculate(a, b int) (int, error) {
    // 一些代码
}
```

如果你的函数由于某些原因运行错误，你应当返回预先声明过的 error 类型。通常来讲，返回一个错是在向函数调用者发出信号表明发生了一个错误，如果没有错误，就返回 nil 值。这样，你就让调用知道发生了错误，并让调用者处理这个错误：函数的调用者应当在试图使用返回的值之前检查是否发了错误。如果 error 不是 nil ，调用者有责任去检查这个错误并处理它（日志、返回错误、serve、尝重新调用/清理机制等）。

```
result, err := calculate(a, b)
if err != nil {
```

```
    // 处理异常
}
// 继续
```

这些片段在 Go 语言中非常常见，有些人认为它们是一大堆死板的代码。编译器会将没有使用的变量为编译错误，所以当你不打算去检查错误的时候，应该给返回的错误变量分配一个空白标识符 _ 。但无论这个方式多方便，都不应该忽视错误。

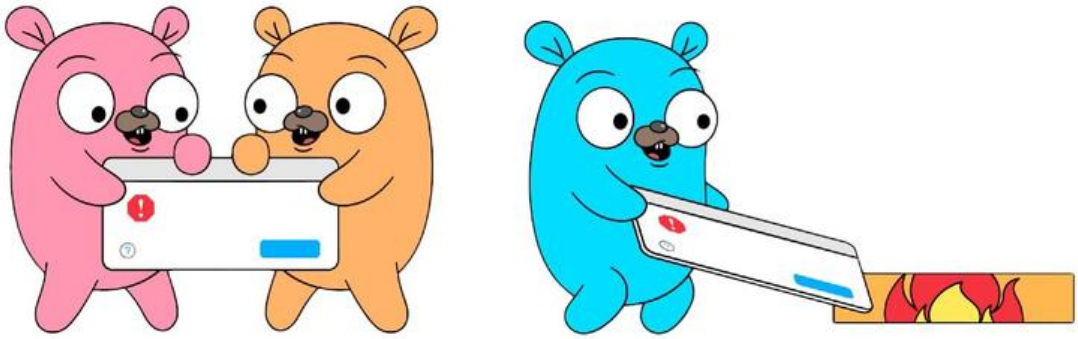
```
// 在检查错误之前，结果无法被信任

result, _ := caculate(a, b)

if result >0 {
    // 忽视错误是不安全的，
    // 理论上讲，在你检查是否有异常之前，
    // 是无法相信你接收到的结果的
}
```

在检查错误之前，结果不能被信任

在 Go 语言严格的检查机制下，让一个函数返回结果的同时返回错误，可以让你更难写出含有错误的法。你应当假设，函数的返回值是不正确的（损坏的）除非你检查了函数返回的错误值。如果将错误配给了空白标识符，说明你忽略了你的函数值可能已经损坏。



空白标识符是黑暗的，令人恐惧的。

Go 语言确实有一个 panic 和 recover 机制，这再另一篇Go博文中有详细的描述。但是这不意味着去仿异常。用 Dave 的话说就是：“当你在使用 Go 的时候产生 panic，你会被吓坏，这不是其他人的题，这是完蛋了，兄弟。”他们非常的致命，并且会导致你的程序崩溃。Rob Pike 创造了“不要恐慌的谚语，这是不言自明的：你应当避免它，并返回错误。

- “错误就是价值观。”
- “不要只是检查错误，优雅地处理它们”
- “不要惊慌失措”

Rob Pike 所有关于 Go 的谚语

深入理解

关于错误的接口

在底层实现中，error 类型是一个普通的单方法接口，如果你还对他不熟悉，我强烈建议你仔细的阅在 Go 官方博客中的这篇文章.

```
// error interface from the source code
type error interface {
    Error() string
}
```

错误接口的源码

实现你自己的错误类型非常容易，有非常多的方法能够让你构造实现Error() string方法的自定义结构。任何实现了这个方法的结构体都会被视为一个合法的错误值同时可以被返回。

接下来，就让我们一起去探索这些途径。

内置的错误字符串（errorString）结构体

错误接口中最常用同时也是最出名的就是 errorString 结构体。这是你能想到的最简洁的实现。

```
package errors

func New(text string) error {
    return &errorString{text}
}

type errorString struct {
    s string
}

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

来源：Go 语言源码

你可以在这里看到它的简单实现。它做的事情就是保存一个 string，同时，这个字符串是由 Error 方返回的。我们可以使用数据格式化这个错误信息，比如，fmt.Springf。但除此之外，它不包含任何其功能。如果你在使用内置的 errors.New 或者 fmt.Errorf，你就已经在使用他们了。

```
import (
    "errors"
    "fmt"
)

func main() {
    e1 := errors.New(fmt.Sprintf("Could not open file"))
    e2 := fmt.Errorf("Could not open file")
```

```
    fmt.Println(fmt.Sprintf("Type of error 1: %T", e1))
    fmt.Println(fmt.Sprintf("Type of error 2: %T", e2))

    // output:
    // Type of error 1: *errors.errorString
    // Type of error 2: *errors.errorString
}
```

尝试一下

github.com/pkg/errors

另一个简单的示例是 pkg/errors 包。不要与之前学到的内置 errors 包混淆这个包额外提供了一些重的功能，比如错误的封装（wrapping）、展开（unwrapping），格式化和堆栈跟踪记录。你可以通运行 go get github.com/pkg/errors 来安装这个包。

```
go get github.com/pkg/errors
```

如果需要将堆栈跟踪信息附加的错误中，或是附加必要的调试信息到错误中，可以使用此包的 New 者 Errorf 函数，他们已经记录下了你的堆栈记录。通过它的格式化能力，附加些简单的元数据。因为 rrorf 实现了 fmt.Formatter 接口，这意味着你可以使用 fmt 包的 runes(%s, %v, %+v etc) 来格式他们。

```
import "github.com/pkg/errors"

// ...

errors.New("error writing to file")
// or, alternatively
errors.Errorf("error writing to file %s", f.Path)
```

这个包还包含 errors.Wrap 和 errors.Wrapf 函。这些函数将上下文以及调用时的堆栈信息添加到 err r 中。这样,你就可以将其与其上下文和重要的调试数据封装在一起,而不是简单地返回错误。

```
if err != nil {
    return errors.Wrap(err, "could not open file")
}
```

经过封装的错误支持 Cause() error 方法，并且会返回它们的内部错误。通常，它们可以与 errors.Ca se(err error) error 函数一起使用，这将会检索这个错误中最底层的错误。

处理错误（Working with Errors)

类型断言

类型断言在处理错误的时候扮演者非常重要的角色。你需要使用它们来在接口值中断言信息，同时，于错误处理涉及到 error 接口的自定义实现，所以在对错误执行断言是非常方便的工具。

它的语法对于所有的目标（purposes）都是相同的——x.(T)，其中 x 是接口类型。x.(T) 断言 x 不为 n l，并且存储在 x 中的值类型为 T。在接下来的几节里面，你将会看到使用类型断言的两种方式——过使用具体类型 T 和使用接口类型 T。

```
var x interface{}
```

```
// short syntax, dropping the ok boolean
// panic: interface conversion: interface is nil, not string
s :+ x.(string)

// long syntax, with the ok boolean
if s ok := x.(string); ok {
    // does not panic, instead ok is set to false when assertion fails
    // we can now use s as string safely
}
```

playground: short syntax panic, safe long syntax

关于语法的附加说明:类型断言可以与短语法(当断言失败时，短语法会引发 panic)和长语法(使用 OK-boolean 表示成功或失败)一起使用。我总是建议选择长语法的而不是短语法，因为我更喜欢检查 OK 变而不是处理 panic 。

使用接口类型T进行断言

使用接口类型 T 进行类型断言能够断言 x 实现了接口 T 。通过断言，你能确保接口值是实现其接口义，只有在这个前提下，才去调用其方法。

```
type resolver interface {
    Resolve()
}

if v, ok := x.(resolver); ok {    // asserts x implements resolver
    v.Resolver() // here we can use this method safely
}
```

为了理解如何利用这一特性，让我们重新查看一下 pkg/errors。你已经知道了errors这个包，所以就我们直接进入 errors.Cause(err error) error 函数去看一下吧。

这个函数输入一个 error 并提取出它封装的最底层的错误（在这个错误内部没有再封装其它的错误）这看起来很简单，但是你可以从这个实现中学到很多很有用的东西：

```
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
}
```

来源：pkg/errors

这个函数获取一个错误值并且它不能假设 err 参数接收到的是一个封装过的错误（一个支持 Cause 法的错误）。所以，在调用 Cause 方法之前，有必要检查一下是否正在处理一个实现 Cause 方法的 e ror 变量。通过在每个 for 循环中进行类型断言，你可以保证变量 cause 支持 Cause 方法，并且可不断的从中提取出内部错误直到这个错误不再包含 cause。

通过创建一个只包含你需要的方法的精简的本地接口，并在其上执行断言，您的代码将与其他依赖项耦。你接收到的参数不需要是一个已知的结构体，只需要是一个错误就可以。任何实现 Error 和 Cause 方法的类型都可以在这里使用。所以，当你在你自定义的错误类型中实现 Cause 方法的时候，你可直接使用这个函数。

不过，你应该注意一个小问题：接口可能会发生变化。因此你应该小心的维护你的代码，这样你的断才不会崩溃。记住一点，要在使用它们的地方定义接口，保持它们的简洁，并小心维护它们，这样就容易出现问题。

最后，如果你只关心一个方法，那么在匿名接口上断言只包含您所依赖的方法有时会更方便，即 v, ok = x.(interface{ F() (int, error) })。使用匿名接口可以帮助你将代码从依赖项中分离出来，并且可以帮保护代码不受接口中可能发生的更改的影响。

使用具体类型T和类型转换进行断言

在本节开始之前，我将介绍两个类似的错误处理模式，它们都有一些缺点和陷阱。但这并不意味着它不常见。在小型项目中，这两种工具都非常方便，只是它们的伸缩性不太好。

首先，是第二种类型断言：使用具体类型 T 进行类型断言 x.(T)。它断言 x 的值是 T 类型，或者将它换为 T 类型。

```
if v, ok := err.(mypkg.SomeErrorType); ok {
    // we can use v as mypkg.SomeErrorType
}
```

另一个是类型转换模式。类型转换通过保留类型关键字 type 将 switch 语句与类型断言组合在一起。们在错误处理中特别常见。在错误处理中，了解错误变量的基本类型非常有用。

```
switch err.(type) {
case mypkg.SomeErrorType:
    // handle...
default:
    // handle...
}
```

这两种方法的最大缺点是，它们都会导致代码与其依赖项耦合。这两个示例都需要熟悉 SomeErrorTy e 结构(显然需要导出它)，并需要导入 mypkg 包。

在这两种方法中，当处理错误时，你必须熟悉这个类型并导入它的包。当您处理包装错误时，情况会得更糟，其中错误的原因可能是在你没有(也不应该)意识到的内部依赖项中创建的错误。

```
import "mypkg"

// ...

switch err := errors.Cause(err).(type) {
case mypkg.SomeErrorType:
```

```
    // handle...
default:
    // handle...
}
```

类型转换区分了 *MyStruct 和 MyStruct 。因此，如果不确定是在处理指针还是结构体的实例（actua instance），你必须同时提供这两种方法。而且，就像开关(译注：开关和转换的英文都是 switch)一，类型转换中的 case 不会顺延（fall through），但是与开关不同，类型转换禁止使用 fallthrough 所以您必须使用逗号并提供两个选项，这很容易被忘记。

```
if err != {
    // log the error once, log.Log(err)

    cause := errors.Cause(err)
    switch cause.(type) {
    case SomeErrorType, *SomeErrorType:
        // handle...
    default:
        // handle...
    }
}
```

总结

就是这样！现在，你已经熟悉了错误，并且应当准备好处理你的 Go 程序抛出（或实际返回）的任何误了！

这两个 errors 包都提供了在Go中处理错误的简单但重要的方法，如果它们满足了你的需求，那么它就是非常好的选择。你可以轻松地实现自己自定义的错误结构，并享受将它们与 pkg/errors 组合时得的好处。

当你扩展出简单的错误时，正确地使用类型断言可以成为处理不同错误的一个很好的工具。要么使用型转换，要么断言错误的行为并检查它实现的接口。

接下来该做什么

Go 的错误处理现在是一个非常热门的话题。现在你已经掌握了基本的知识，你可能会对 Go 错误处未来的发展趋势感兴趣!

在即将到来的 Go 2 版本，Go 错误处理获得了非常多的关注，你现在已经可以在设计草图中进行查。同时，在 dotGo 2019 期间，Marcel van Lohuizen就这个话题进行了一次非常棒的演讲，我极力荐大家去看一下——“Go 2 Error Values Today”。

很显然，还有很多方法、技巧和一些细节点，我不可能把它们都放在一篇文章中进行讲解！无论如何我希望你们喜欢这篇文章，我们将在“在你进入到Go语言的世界之前”系列中的下一期再见!

---

via: https://medium.com/gett-engineering/error-handling-in-go-53b8a7112d04

作者：Alon Abadi

译者：JoeyGaojingxing
校对：magichan

本文由 GCTT 原创编译，Go 中文网 荣誉推出