20 | 错误处理（下）

2018-09-26 郝林

20 | 错误处理（下）

朗读人：郝林　07’57’’ | 4.56M

你好，我是郝林，今天我们继续来分享错误处理。

在上一篇文章中，我们主要讨论的是从使用者的角度看“怎样处理好错误值”。那么，接下来我们需要关注的，就是站在建造者的角度，去关心“怎样才能给予使用者恰当的错误值”的问题了。

知识扩展

问题：怎样根据实际情况给予恰当的错误值？

我们已经知道，构建错误值体系的基本方式有两种，即：创建立体的错误类型体系和创建扁平的错误值列表。

先说错误类型体系。由于在 Go 语言中实现接口是非侵入式的，所以我们可以做得很灵活。比如，在标准库的net代码包中，有一个名为Error的接口类型。它算是内建接口类型error的一个扩展接口，因为error是net.Error的嵌入接口。

net.Error接口除了拥有error接口的Error方法之外，还有两个自己声明的方法：Timeout和Temporary。

net包中有很多错误类型都实现了net.Error接口，比如：

1. *net.OpError；
2. *net.AddrError；
3. net.UnknownNetworkError等等。

你可以把这些错误类型想象成一棵树，内建接口error就是树的根，而net.Error接口就是一个在根上延伸的第一级非叶子节点。

同时，你也可以把这看做是一种多层分类的手段。当net包的使用者拿到一个错误值的时候，可以先判断它是否是net.Error类型的，也就是说该值是否代表了一个网络相关的错误。

如果是，那么我们还可以再进一步判断它的类型是哪一个更具体的错误类型，这样就能知道这个网络相关的错误具体是由于操作不当引起的，还是因为网络地址问题引起的，又或是由于网络协议不正确引起的。

当我们细看net包中的这些具体错误类型的实现时，还会发现，与os包中的一些错误类型类似，它们也都有一个名为Err、类型为error接口类型的字段，代表的也是当前错误的潜在错误。

所以说，这些错误类型的值之间还可以有另外一种关系，即：链式关系。比如说，使用者调用net.DialTCP之类的函数时，net包中的代码可能会返回给他一个*net.OpError类型的错误值，以表示由于他的操作不当造成了一个错误。

同时，这些代码还可能会把一个*net.AddrError或net.UnknownNetworkError类型的值赋给该错误值的Err字段，以表明导致这个错误的潜在原因。如果，此处的潜在错误值的Err字段也有非nil的值，那么将会指明更深层次的错误原因。如此一级又一级就像链条一样最终会指向问题的根源。

把以上这些内容总结成一句话就是，用类型建立起树形结构的错误体系，用统一字段建立起可追根溯源的链式错误关联。这是 Go 语言标准库给予我们的优秀范本，非常有借鉴意义。

不过要注意，如果你不想让包外代码改动你返回的错误值的话，一定要小写其中字段的名称首字母。你可以通过暴露某些方法让包外代码有进一步获取错误信息的权限，比如编写一个可以返回包级私有的err字段值的公开方法Err。

相比于立体的错误类型体系，扁平的错误值列表就要简单得多了。当我们只是想预先创建一些代表已知错误的错误值时候，用这种扁平化的方式就很恰当了。

不过，由于error是接口类型，所以通过errors.New函数生成的错误值只能被赋给变量，而不能赋给常量，又由于这些代表错误的变量需要给包外代码使用，所以其访问权限只能是公开的。

这就带来了一个问题，如果有恶意代码改变了这些公开变量的值，那么程序的功能就必然会受到影响。因为在这种情况下我们往往会通过判等操作来判断拿到的错误值具体是哪一个错误，如果这些公开变量的值被改变了，那么相应的判等操作的结果也会随之改变。

这里有两个解决方案。第一个方案是，先私有化此类变量，也就是说，让它们的名称首字母变成小写，然后编写公开的用于获取错误值以及用于判等错误值的函数。

比如，对于错误值os.ErrClosed，先改写它的名称，让其变成os.errClosed，然后再编写ErrClosed函数和IsErrClosed函数。

当然了，这不是说让你去改动标准库中已有的代码，这样做的危害会很大，甚至是致命的。我只能说，对于你可控的代码，最好还是要尽量收紧访问权限。

再来说第二个方案，此方案存在于syscall包中。该包中有一个类型叫做Errno，该类型代表了系统调用时可能发生的底层错误。这个错误类型是error接口的实现类型，同时也是对内建类型uintptr的再定义类型。

由于uintptr可以作为常量的类型，所以syscall.Errno自然也可以。syscall包中声明有大量的Errno类型的常量，每个常量都对应一种系统调用错误。syscall包外的代码可以拿到这些代表错误的常量，但却无法改变它们。

我们可以仿照这种声明方式来构建我们自己的错误值列表，这样就可以保证错误值的只读特性了。

好了，总之，扁平的错误值列表虽然相对简单，但是你一定要知道其中的隐患以及有效的解决方案是什么。

总结

今天，我从两个视角为你总结了错误类型、错误值的处理技巧和设计方式。我们先一起看了一下 Go 语言中处理错误的最基本方式，这涉及了函数结果列表设计、errors.New函数、卫述语句以及使用打印函数输出错误值。

接下来，我提出的第一个问题是关于错误判断的。对于一个错误值来说，我们可以获取到它的类型、值以及它携带的错误信息。

如果我们可以确定其类型范围或者值的范围，那么就可以使用一些明确的手段获知具体的错误种类。否则，我们就只能通过匹配其携带的错误信息来大致区分它们的种类。

由于底层系统给予我们的错误信息还是很有规律可循的，所以用这种方式去判断效果还比较显著。但是第三方程序给出的错误信息很可能就没那么规整了，这种情况下靠错误信息去辨识种类就会比较困难。

有了以上阐释，当把视角从使用者换位到建造者，我们往往就会去自觉地仔细思考程序错误体系的设计了。我在这里提出了两个在 Go 语言标准库中使用很广泛的方案，即：立体的错误类型体系和扁平的错误值列表。

之所以说错误类型体系是立体的，是因为从整体上看它往往呈现出树形的结构。通过接口间的嵌套以及接口的实现，我们就可以构建出一棵错误类型树。

通过这棵树，使用者就可以一步步地确定错误值的种类了。另外，为了追根溯源的需要，我们还可以在错误类型中，统一安放一个可以代表潜在错误的字段。这叫做链式的错误关联，可以帮助使用者找到错误的根源。

相比之下，错误值列表就比较简单了。它其实就是若干个名称不同但类型相同的错误值集合。

不过需要注意的是，如果它们是公开的，那就应该尽量让它们成为常量而不是变量，或者编写私有的错误值以及公开的获取和判等函数，否则就很难避免恶意的篡改。

这其实是“最小化访问权限”这个程序设计原则的一个具体体现。无论怎样设计程序错误体系，我们都应该把这一点考虑在内。

思考题

请列举出你经常用到或者看到的 3 个错误值，它们分别在哪个错误值列表里？这些错误值列表分别包含的是哪个种类的错误？