Errors are values

Rob Pike
12 January 2015

A common point of discussion among Go programmers, especially those new to the language, is how to handle errors. The conversation often turns into a lament at the number of times the sequence

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

shows up. We recently scanned all the open source projects we could find and discovered that this snippet occurs only once per page or two, less often than some would have you believe. Still, if the perception persists that one must type

```
if err != nil
```

all the time, something must be wrong, and the obvious target is Go itself.

This is unfortunate, misleading, and easily corrected. Perhaps what is happening is that programmers new to Go ask, "How does one handle errors?", learn this pattern, and stop there. In other languages, one might use a try-catch block or other such mechanism to handle errors. Therefore, the programmer thinks, when I would have used a try-catch in my old language, I will just type `if err != nil` in Go. Over time the Go code collects many such snippets, and the result feels clumsy.

Regardless of whether this explanation fits, it is clear that these Go programmers miss a fundamental point about errors: *Errors are values*.

Values can be programmed, and since errors are values, errors can be programmed.

Of course a common statement involving an error value is to test whether it is nil, but there are countless other things one can do with an error value, and application of some of those other things can make your program better, eliminating much of the boilerplate that arises if every error is checked with a rote if statement.

Here's a simple example from the `bufio` package's `Scanner` type. Its `Scan` method performs the underlying I/O, which can of course lead to an error. Yet the `Scan` method does not expose an error at all. Instead, it returns a boolean, and a separate method, to be run at the end of the scan, reports whether an error occurred. Client code looks like this:

```
scanner := bufio.NewScanner(input)
for scanner.Scan() {
    token := scanner.Text()
    // process token
}
if err := scanner.Err(); err != nil {
    // process the error
}
```

Sure, there is a nil check for an error, but it appears and executes only once. The `Scan` method could instead have been defined as

```
func (s *Scanner) Scan() (token []byte, error)
```

and then the example user code might be (depending on how the token is retrieved),

```
scanner := bufio.NewScanner(input)
for {
    token, err := scanner.Scan()
    if err != nil {
        return err // or maybe break
    }
    // process token
}
```

This isn't very different, but there is one important distinction. In this code, the client must check for an error on every iteration, but in the real `Scanner` API, the error handling is abstracted away from the key API element, which is iterating over tokens. With the real API, the client's code therefore feels more natural: loop until done, then worry about errors. Error handling does not obscure the flow of control.

Under the covers what's happening, of course, is that as soon as `Scan` encounters an I/O error, it records it and returns `false`. A separate method, `Err`, reports the error value when the client asks. Trivial though this is, it's not the same as putting

```
if err != nil
```

everywhere or asking the client to check for an error after every token. It's programming with error values. Simple programming, yes, but programming nonetheless.

It's worth stressing that whatever the design, it's critical that the program check the errors however they are exposed. The discussion here is not about how to avoid checking errors, it's about using the language to handle errors with grace.

The topic of repetitive error-checking code arose when I attended the autumn 2014 GoCon in Tokyo. An enthusiastic gopher, who goes by `@jxck_` on Twitter, echoed the

familiar lament about error checking. He had some code that looked schematically like this:

```
_, err = fd.Write(p0[a:b])
if err != nil {
    return err
}
_, err = fd.Write(p1[c:d])
if err != nil {
    return err
}
_, err = fd.Write(p2[e:f])
if err != nil {
    return err
}
// and so on
```

It is very repetitive. In the real code, which was longer, there is more going on so it's not easy to just refactor this using a helper function, but in this idealized form, a function literal closing over the error variable would help:

```
var err error
write := func(buf []byte) {
    if err != nil {
        return
    }
    _, err = w.Write(buf)
}
write(p0[a:b])
write(p1[c:d])
write(p2[e:f])
// and so on
if err != nil {
    return err
}
```

This pattern works well, but requires a closure in each function doing the writes; a separate helper function is clumsier to use because the `err` variable needs to be maintained across calls (try it).

We can make this cleaner, more general, and reusable by borrowing the idea from the `Scan` method above. I mentioned this technique in our discussion but `@jxck_` didn't see how to apply it. After a long exchange, hampered somewhat by a language barrier, I asked if I could just borrow his laptop and show him by typing some code.

I defined an object called an `errWriter`, something like this:

```
type errWriter struct {
    w   io.Writer
```

```
    err error
}
```

and gave it one method, `write`. It doesn't need to have the standard `Write` signature, and it's lower-cased in part to highlight the distinction. The `write` method calls the `Write` method of the underlying `Writer` and records the first error for future reference:

```
func (ew *errWriter) write(buf []byte) {
    if ew.err != nil {
        return
    }
    _, ew.err = ew.w.Write(buf)
}
```

As soon as an error occurs, the `write` method becomes a no-op but the error value is saved.

Given the `errWriter` type and its `write` method, the code above can be refactored:

```
ew := &errWriter{w: fd}
ew.write(p0[a:b])
ew.write(p1[c:d])
ew.write(p2[e:f])
// and so on
if ew.err != nil {
    return ew.err
}
```

This is cleaner, even compared to the use of a closure, and also makes the actual sequence of writes being done easier to see on the page. There is no clutter anymore. Programming with error values (and interfaces) has made the code nicer.

It's likely that some other piece of code in the same package can build on this idea, or even use `errWriter` directly.

Also, once `errWriter` exists, there's more it could do to help, especially in less artificial examples. It could accumulate the byte count. It could coalesce writes into a single buffer that can then be transmitted atomically. And much more.

In fact, this pattern appears often in the standard library. The `archive/zip` and `net/http` packages use it. More salient to this discussion, the `bufio` package's `Writer` is actually an implementation of the `errWriter` idea. Although `bufio.Writer.Write` returns an error, that is mostly about honoring the `io.Writer` interface. The `Write` method of `bufio.Writer` behaves just like our `errWriter.write` method above, with `Flush` reporting the error, so our example could be written like this:

```
b := bufio.NewWriter(fd)
```

```
b.Write(p0[a:b])
b.Write(p1[c:d])
b.Write(p2[e:f])
// and so on
if b.Flush() != nil {
    return b.Flush()
}
```

There is one significant drawback to this approach, at least for some applications: there is no way to know how much of the processing completed before the error occurred. If that information is important, a more fine-grained approach is necessary. Often, though, an all-or-nothing check at the end is sufficient.

We've looked at just one technique for avoiding repetitive error handling code. Keep in mind that the use of `errWriter` or `bufio.Writer` isn't the only way to simplify error handling, and this approach is not suitable for all situations. The key lesson, however, is that errors are values and the full power of the Go programming language is available for processing them.

Use the language to simplify your error handling.

But remember: Whatever you do, always check your errors!

Finally, for the full story of my interaction with @jxck_, including a little video he recorded, visit his blog.

Next article: Package names
Previous article: GothamGo: gophers in the big apple
Blog Index