Error Values: Frequently Asked Questions

The Go 2 error values proposal adds functionality to the `errors` and `fmt` packages of the standard library for Go 1.13. There is also a compatibility package, `golang.org/x/xerrors`, for earlier Go versions.

We suggest using the `xerrors` package for backwards compatibility. When you no longer wish to support Go versions before 1.13, use the corresponding standard library functions. This FAQ uses the `errors` and `fmt` packages from Go 1.13.

How should I change my error-handling code to work with the new features?

You need to be prepared that errors you get may be wrapped.

- If you currently compare errors using `==`, use `errors.Is` instead. Example:

  ```
  if err == io.ErrUnexpectedEOF
  ```

  becomes

  ```
  if errors.Is(err, io.ErrUnexpectedEOF)
  ```

  - Checks of the form `if err != nil` need not be changed.
  - Comparisons to `io.EOF` need not be changed, because `io.EOF` should never be wrapped.

- If you check for an error type using a type assertion or type switch, use `errors.As` instead. Example:

  ```
  if e, ok := err.(*os.PathError); ok
  ```

  becomes

  ```
  var e *os.PathError
  if errors.As(err, &e)
  ```

  - Also use this pattern to check whether an error implements an interface. (This is one of those rare cases when a pointer to an interface is appropriate.)
  - Rewrite a type switch as a sequence of if-elses.

I am already using `fmt.Errorf` with `%v` or `%s` to provide context for an error. When should I switch to `%w`?

It's common to see code like

```
if err := frob(thing); err != nil {
    return fmt.Errorf("while frobbing: %v", err)
}
```

With the new error features, that code continues to work exactly as before, constructing a string that includes the text of `err`. Changing from `%v` to `%w` doesn't change that string, but it does wrap `err`, allowing the caller to access it using `errors.Unwrap`, `errors.Is` or `errors.As`.

So use `%w` if you want to expose the underlying error to your callers. Keep in mind that doing so may be exposing implementation detail that can constrain the evolution of your code. Callers can depend on the type and value of the error you're wrapping, so changing that error can now break them. For example, if the `AccessDatabase` function of your package `pkg` uses Go's `database/sql` package, then it may encounter a `sql.ErrTxDone` error. If you return that error with `fmt.Errorf("accessing DB: %v", err)` then callers won't see that `sql.ErrTxtDone` is part of the error you return. But if you instead return `fmt.Errorf("accessing DB: %w", err)`, then a caller could reasonably write

```
err := pkg.AccessDatabase(...)
if errors.Is(err, sql.ErrTxDone) ...
```

At that point, you must always return `sql.ErrTxDone` if you don't want to break your clients, even if you switch to a different database package.

How can I add context to an error I'm already returning without breaking clients?

Say your code now looks like

```
return err
```

and you decide that you want to add more information to `err` before you return it. If you write

```
return fmt.Errorf("more info: %v", err)
```

then you might break your clients, because the identity of `err` is lost; only its message remains.

You could instead wrap the error by using `%w`, writing

```
return fmt.Errorf("more info: %w", err)
```

This will still break clients who use `==` or type assertion to test errors. But as we discussed in the first question of this FAQ, consumers of errors should migrate to the `errors.Is` and `errors.As` functions. If you can be sure that your clients have done so, then it is not a breaking change to switch from

```
return err
```

to

```
return fmt.Errorf("more info: %w", err)
```

I'm writing new code, with no clients. Should I wrap returned errors or not?

Since you have no clients, you aren't constrained by backwards compatibility. But you still need to balance two opposing considerations:

- Giving client code access to underlying errors can help it make decisions, which can lead to better software.
- Every error you expose becomes part of your API: your clients may come to rely on it, so you can't change it.

For each error you return, you have to weigh the choice between helping your clients and locking yourself in. Of course, this choice is not unique to errors; as a package author, you make many decisions about whether a feature of your code is important for clients to know or an implementation detail.

With errors, though, there is an intermediate choice: you can expose error details to people reading your code's error messages without exposing the errors themselves to client code. One way to do that is to put the details in a string using `fmt.Errorf` with `%s` or `%v`. Another is to write a custom error type, add the details to the string returned by its `Error` method, and avoid defining an `Unwrap` method.

I maintain a package that exports an error-checking predicate function. How should I adapt to the new features?

Your package has a function or method `IsX(error) bool` that reports whether an error has some property. A natural thought would be to modify `IsX` to unwrap the error it is passed, checking the property for each error in the chain of wrapped errors. We advise against doing this: the change in behavior could break your users.

Your situation is like that of the standard `os` package, which has several such functions. We recommend the approach we took there. The `os` package has several predicates, but we treated most of them the same. For concreteness, we'll look at `os.IsExist`.

Instead of changing `os.IsExist`, we made `errors.Is(err, os.ErrExist)` behave like it, except that `Is` unwraps. (We did this by having `syscall.Errno` implement an `Is` method, as described in the documentation for `errors.Is`.) Using `errors.Is` will always work correctly, because it will exist only in Go versions 1.13 and higher. For older versions of Go, you should recursively unwrap the error yourself, calling `os.IsExist` on each underlying error.

This technique only works if you have control of the errors being wrapped, so you can add `Is` methods to them. In that case, we recommend:

- Don't change your `IsX(error) bool` function; do change its documentation to clarify that it does not unwrap.
- If you don't already have one, add a global variable whose type implements `error` that represents the condition that your function tests:

  ```
  var ErrX = errors.New("has property X")
  ```

- Add an `Is` method to the types for which `IsX` returns true. The `Is` method should return true if its argument equals `ErrX`.

If you don't have control of all the errors that can have property X, you should instead consider adding another function that tests for the property while unwrapping, perhaps

```
func IsXUnwrap(err error) bool {
    for e := err; e != nil; e = errors.Unwrap(e) {
        if IsX(e) {
            return true
        }
    }
    return false
}
```

Or you could leave things as they are, and let your users do the unwrapping themselves. Either way, you should still change the documentation of `IsX` to clarify that it does not unwrap.

I have a type that implements `error` and holds a nested error. How should I adapt it to the new features?

If your type already exposes the error, write an `Unwrap` method.

For example, perhaps your type looks like

```
type MyError struct {
    Err error
    // other fields
}

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

Then you should add

```
func (e *MyError) Unwrap() error { return e.Err }
```

Your type will then work correctly with the `Is` and `As` functions of `errors` and `xerrors`.

We've done that for `os.PathError` and other, similar types in the standard library.

It's clear that writing an `Unwrap` method is the right choice if the nested error is exported, or otherwise visible to code outside your package, such as via a method like `Unwrap`. But if the nested error is not exposed to outside code, you probably should keep it that way. Making the error visible by returning it from `Unwrap` will enable your clients to depend on the type of the nested error, which can expose implementation details and constrain the evolution of your package. See the discussion of `%w` above for more.