Exceptional Go

medium.com/extreme-programming/exceptional-go-3943c2230cf8

You'll often hear Go developers claiming that it's incorrect (not idiomatic) to panic in Go; instead we should return errors. After some research, I believe this is a misunderstanding of what the Go community has said in posts like this.

Dave Cheney, project member for the Go programming language, has a great article about the classic distinction of errors vs. exceptions.

Money quote:

> panics are always fatal to your program. In panicing you never assume that your caller can solve the problem. Hence panic is only used in exceptional circumstances, ones where it is not possible for your code, or anyone integrating your code to continue.

So the point the Go error handling mechanism is getting at is simply that you should differentiate between "errors" (which the client may reasonable be expected be able to recover from) from exceptions (things that are truly exceptional, like a file you expect to be checked in just mysteriously not being there, or your configuration not successfully loading).

It's Not a Go Thing

Note, this *isn't specific to Go!* These are the exact same guidelines laid out in the book *Exceptional Ruby*, which I posted notes from in this blog previously. The only difference is that in Ruby and most other languages, there's no built in error construct. Developers have come up with various ad hoc solutions to this problem. For instance, Ruby doesn't have multiple return values, so if you want to return a separate error object in the vein of Go errors, you can do something like this.

```
def get_config
  # lines of code which attempt to load a file...
  return config, error # This just returns an array [config, error]
end
```

```
config, error = get_config # This just splits the array in two
```

Here’s another approach, from the `ActiveRecord` ORM. This function is from the comment creation endpoint on a Rails web application. The comment model is allowed to be invalid (because it was assigned from user input via POST params), so of course we want to gracefully handle this rather than just blow up.

```
def create
   = Comment.new(params[:comment])
  if .save
    flash_success_message('comment created')
    redirect_to(comment_path(@comment)) # show the comment
  else
    render(:edit)
  end
end
```

The `save` method returns false if the record isn’t valid. If we then want to see the error information, we can just call `@comment.errors`. But let’s suppose we’re in a context where we have no interaction with a user, like a queued job where the parameters should have been correctly set by our own code.

```
def create
   = Comment.new(params[:comment])
  .save! # may raise ActiveRecord::RecordInvalid
end
```

People often abuse exceptions in Ruby, raising and even rescuing their own errors as a means of control flow, even for things that aren’t “exceptional”. Funny story. When I was new to Rails, I once wrote a function that used the exception-throwing `save!` instead of `save`, and then I rescued and dealt with the invalid record case, giving the user a chance to fix and resubmit the form. This is *exactly* the same kind of anti-pattern the Go community is talking about when they say, “return errors not exceptions”. They don’t mean “nothing should panic or exit”, they mean “differentiate exceptions from errors”. When I recovered from that invalid record error, that was a clear indication that I should have been using the non-exceptional `save` over the exceptional `save!`.

So back to Go. If you know you're going to panic anyway, there's nothing "anti-Go-standards" about doing it immediately rather than passing up an error and letting the panic happen in `main()`. A good example might be config loading at program start. Unless you have at least one caller of the `LoadConfig()` function that wants to (and can) recover, there's no reason to pass back an error. Rule of thumb: if you're always going to panic or exit from a returned error, it's perfectly reasonable to just do it immediately.

`panic` vs. `os.Exit()`

One last thing. If you fail via a call like `log.Fatalf()`, it actually directly calls `os.Exit()`, so you're not even panicking.