Don’t force allocations on the callers of your API

by Dave Cheney

This is a post about performance. Most of the time when worrying about the performance of a piece of code the overwhelming advice should be (with apologies to Brendan Gregg) *don’t worry about it, yet.* However there is one area where I counsel developers to think about the performance implications of a design, and that is API design.

Because of the high cost of retrofitting a change to an API’s signature to address performance concerns, it’s worthwhile considering the performance implications of your API’s design on its caller.

A tale of two API designs

Consider these two `Read` methods:

```
func (r *Reader) Read(buf []byte) (int, error)
func (r *Reader) Read() ([]byte, error)
```

The first method takes a `[]byte` buffer and returns the number of bytes read into that buffer and possibly an `error` that occurred while reading. The second takes no arguments and returns some data as a `[]byte` or an `error`.

This first method should be familiar to any Go programmer, it’s `io.Reader.Read`. As ubiquitous as `io.Reader` is, it’s not the most convenient API to use. Consider for a moment that `io.Reader` is the only Go interface in widespread use that returns *both* a result *and* an error. Meditate on this for a moment. The standard Go idiom, checking the error and iff it is `nil` is it safe to consult the other return values, does not apply to `Read`. In fact the caller must do the opposite. First they must record the number of bytes read into the buffer, reslice the buffer, process that data, and only then, consult the error. This is an unusual API for such a common operation and one that frequently catches out newcomers.

A trap for young players?

Why is it so? Why is one of the central APIs in Go’s standard library written like this? A superficial answer might be `io.Reader`‘s signature is a reflection of the underlying `read(2)` syscall, which is indeed true, but misses the point of this post.

If we compare the API of `io.Reader` to our alternative, `func Read() ([]byte, error)`, this API seems easier to use. Each call to `Read()` will return the data that was read, no need to reslice buffers, no need to remember the special case to do this before checking the error. Yet this is not the signature of

`io.Reader.Read`. Why would one of Go's most pervasive interfaces choose such an awkward API? The answer, I believe, lies in the performance implications of the APIs signature on the *caller*.

Consider again our alternative `Read` function, `func Read() ([]byte, error)`. On each call `Read` will read some data into a buffer[1] and return the buffer to the caller. Where does this buffer come from? Who allocates it? The answer is the buffer is allocated *inside* `Read`. Therefore each call to `Read` is guaranteed to allocate a buffer which would escape to the heap. The more the program reads, the faster it reads data, the more streams of data it reads concurrently, the more pressure it places on the garbage collector.

The standard libraries' `io.Reader.Read` forces the caller to supply a buffer because if the caller is concerned with the number of allocations their program is making this is precisely the kind of thing they want to control. Passing a buffer into `Read` puts the control of the allocations into the caller's hands. If they aren't concerned about allocations they can use higher level helpers like `ioutil.ReadAll` to read the contents into a `[]byte`, or `bufio.Scanner` to stream the contents instead.

The opposite, starting with a method like our alternative `func Read() ([]byte, error)` API, prevents callers from pooling or reusing allocations–no amount of helper methods can fix this. As an API author, if the API cannot be changed you'll be forced to add a second form to your API taking a supplied buffer and reimplementing your original API in terms of the newer form. Consider, for example, `io.CopyBuffer`. Other examples of retrofitting APIs for performance reasons are the `fmt` package and the `net/http` package which drove the introduction of the `sync.Pool` type precisely because the Go 1 guarantee prevented the APIs of those packages from changing.

If you want to commit to an API for the long run, consider how its design will impact the size and frequency of allocations the caller will have to make to use it.

This API has other problems, such as, *how much data should be read?* or *should it try to read as much as possible, or return promptly if the read would block?*