FILIPPO.IO

Filippo Valsorda, 18 Jul 2019 on Go

EFFICIENT GO APIS WITH THE MID-STACK INLINER

A common task in Go API design is returning a byte slice. In this post I will explore some old techniques and a new one that became possible in Go 1.12 with the introduction of the mid-stack inliner.

Returning a fresh slice

The most natural approach is to return a fresh byte slice, like `crypto/ed25519.Sign`.

```
package ed25519 // import "crypto/ed25519"

func Sign(privateKey PrivateKey, message []byte) []byte
func Sign(privateKey PrivateKey, message []byte) []byte
    Sign signs the message with privateKey and returns a sign
```

The unfortunate issue is that such an API forces a heap allocation for the returned slice. Since the slice's memory must survive the function's lifespan, the escape analysis has to move it to the heap, where allocations are expensive and put pressure on the garbage collector.

```
$ benchstat <(gotip test -bench Sign -benchmem -count 10 cryp

name       time/op
Signing-4  53.5µs ± 0%

name       alloc/op
Signing-4    512B ± 0%

name       allocs/op
Signing-4    6.00 ± 0%
```

Of those six allocations, five are due to `hash.Hash` usage (more on this in the conclusion), but one is the return value. A single allocation might not matter for most applications, but when it does become significant, there is no way for the caller to mitigate it.

Passing the destination

A straightforward solution is to let the caller pass the destination slice, when the output size is known. An example is the current `golang.org/x/crypto/curve25519.ScalarBaseMult` API.

```
package curve25519 // import "golang.org/x/crypto/curve25519"

func ScalarBaseMult(dst, in *[32]byte)
    ScalarBaseMult sets dst to the product in*base where dst
    coordinates of group points, base is the standard generat
    are in little-endian form.
```

Besides looking a lot like C, this API is not at all ergonomic: using it requires pre-allocating the destination even if the caller doesn't really care about performance.

```
var dst [32]byte
curve25519.ScalarBaseMult(&dst, &peerShare)
```

Append-like APIs

A great compromise are append-like APIs. An append-like API appends the result to a passed slice and returns the extended (and possibly reallocated) slice, like `append()`. An example is `hash.Hash.Sum`.

```
h := sha256.New()
h.Write([]byte("hello world\n"))
fmt.Printf("%x", h.Sum(nil))
```

If the caller is unconcerned with performance, they can just pass `nil`, and a new slice will be allocated for them. If they want to save the allocation, though, they just need to pass a slice with enough spare capacity to hold the result.

```
out := make([]byte, 0, 32)
out = h.Sum(out)
```

It also works very well with `sync.Pool`, since the used buffer can just be sliced to zero (`out[:0]`) and returned to the pool. The buffers in the pool will naturally grow to the necessary size.

Still, passing `nil` to all APIs is awkward, and I've seen multiple people confused that `h.Sum(my32BytesSlice)` doesn't just fill the existing slice length. The API that returns a fresh slice would definitely be the most intuitive one, if only it didn't preclude the caller from optimizing away the allocation.

Using the inliner

Enter the mid-stack inliner! Since Go 1.12, the inliner learned how to inline functions that call other functions. We can use this capability to make our allocating APIs as efficient as any other.

All we need to do is make the exposed function a very thin wrapper around the actual implementation that just allocates the output and makes the call. For example, this is the new proposed curve25519 API.

```
func X25519(scalar, point []byte) ([]byte, error) {
        var dst [32]byte
        return x25519(&dst, scalar, point)
}

func x25519(dst *[32]byte, scalar, point []byte) ([]byte, err
        // ...
        return dst[:], nil
}
```

While `dst` normally escapes to the heap, in practice the `X25519` body will be inlined in the caller along with the `dst` allocation, and if the caller is careful not to let it escape, it will stay on the caller's stack. It will be as if the caller were using the hidden, less ergonomic, and more efficient `x25519` API.

We can verify that it works by looking at the inliner and escape analysis debug output (which I trimmed to the relevant lines). Note that the output in Go 1.13 is a little different.

```
$ cat x25519.go
package main

func main() {
        scalar, point := make([]byte, 32), make([]byte, 32)
        res, err := X25519(scalar, point)
        if err != nil {
                panic(err)
        }
        println(res)
}

func X25519(scalar, point []byte) ([]byte, error) {
        var dst [32]byte
        return x25519(&dst, scalar, point)
}

func x25519(dst *[32]byte, scalar, point []byte) ([]byte, err
        // [ actual crypto code omitted ]
        return dst[:], nil
}

$ go build -gcflags -m x25519.go
./x25519.go:??:6: can inline X25519
./x25519.go:??:20: inlining call to X25519
./x25519.go:??:13: leaking param: dst to result ~r3 level=0
./x25519.go:??:20: main &dst does not escape
./x25519.go:??:16: &dst escapes to heap
./x25519.go:??:6: moved to heap: dst
```

What this is telling us is that `dst` escapes to the heap when allocated in `X25519()`, but when `X25519()` is inlined in `main()`, the `dst` instance that got inlined doesn't escape!

This technique gets us the best of both worlds: we can make an intuitive API that's easy to use in the common case, but that still allows performance sensitive callers to avoid the heap allocation by ensuring they don't let the result escape.

Constructors

It's not just slice APIs that can benefit from it: we can use the inliner to allow saving any return value allocation. A very common case is `func NewFoo() *Foo` constructors.

For example, here's how I plan to use it in the new `golang.org/x/crypto/chacha20` package.

```
func NewUnauthenticatedCipher(key, nonce []byte) (*Cipher, er
    var c Cipher
    return newCipher(&c, key, nonce)
}
```

Looks like this will realize a lot of the benefit of a combined ChaCha20-Poly1305 implementation without requiring all that extra assembly! :partyparrot:

Further optimizations

Like any good idea, this is not new, and apparently goes by the general name of *function outlining*: it's moving parts of functions into the parent to enable other optimizations. In this case we are doing so manually, and with the specific objective of enabling more efficient escape analysis.

Russ Cox also pointed out that this will enable another optimization when interfaces are involved: inlining can help the compiler understand the concrete type of an interface return value, and that information can help the escape analysis ensure that arguments of the methods don't escape.

Take for example the `crypto/sha256` API. Unfortunately, `sha256.New` returns a `hash.Hash` interface value.

```
package sha256 // import "crypto/sha256"

func New() hash.Hash
    New returns a new hash.Hash computing the SHA256 checksum
    implements encoding.BinaryMarshaler and encoding.BinaryUn
    marshal and unmarshal the internal state of the hash.
```

The result is that any slice passed to the `Write([]byte)` method escapes, because the escape analysis doesn't actually know what the implementation of that virtual call is. The interface might be implemented by something that retains a reference to the argument!

```
s := make([]byte, 128)
h := sha256.New()
h.Write(s)
h.Sum(nil)
```

Once `New` is inlined though, it's possible that it will become clear in the body of the caller that the concrete type of `h` is always `sha256.digest`, and the escape analysis would be able to prove `s` can stay on the caller's stack. This is called devirtualization, and it's coming in Go 1.13, although it doesn't work yet with escape analysis.

Of course, this is yet another reason to always return concrete types from public APIs, but there's no changing `sha256.New` now.

Conclusion

It's great when different optimizations interact to enable the most idiomatic code to be also the fastest. In this case we've seen that APIs that allocate a return value, including classic constructors, can be made efficient by leveraging the inliner and escape analysis.

For more Go API design, you can follow me on Twitter.

> *Thanks to the new-ish mid-stack inliner, you can make Go APIs that return a new value without forcing a heap allocation!*
>
> *func X25519(scalar, point []byte) ([]byte, error) {*
> *var dst [32]byte*
> *return x25519(&dst, scalar, point)*
> *}https://t.co/shR7wc0mzm*
>
> — Filippo Valsorda (@FiloSottile) July 18, 2019

Filippo Valsorda

Cryptogopher on the Go team at Google. RC F'13, F2'17. You might know me as @FiloSottile.

Sign up to my newsletter—Cryptography Dispatches—for more frequent, lightly edited writings on cryptography.

https://buttondown.email/cryptography-dispatches?tag=blog&as_embed=true