An Overview of sync.Map

Bryan C. Mills
bcmills@google.com
GopherCon, 15 Jul 2017

Hi, my name is Bryan Mills. I'm here to talk about the Map type I implemented for the sync package in Go 1.9.

New in Go 1.9

Package sync

```
…
type Map
    func (m *Map) Delete(key interface{})
    func (m *Map) Load(key interface{}) (value interface{}, ok bool)
    func (m *Map) LoadOrStore(key, value interface{}) (actual interface{}, loaded bool)
    func (m *Map) Range(f func(key, value interface{}) bool)
    func (m *Map) Store(key, value interface{})
…
```

sync.Map provides atomic versions of *most* of the usual map operations, plus a LoadOrStore method. For detailed usage, read the godoc page at tip.golang.org.

Why?

The Go FAQ says "[m]ost new code should live [*outside*] of the standard library", so why did sync.Map make the cut? What problems does it solve?

Diagnosis

On a machine with a lot of CPU cores, proto.Unmarshal is slow

...because reflect.New is slow

...because sync.RWMutex is slow

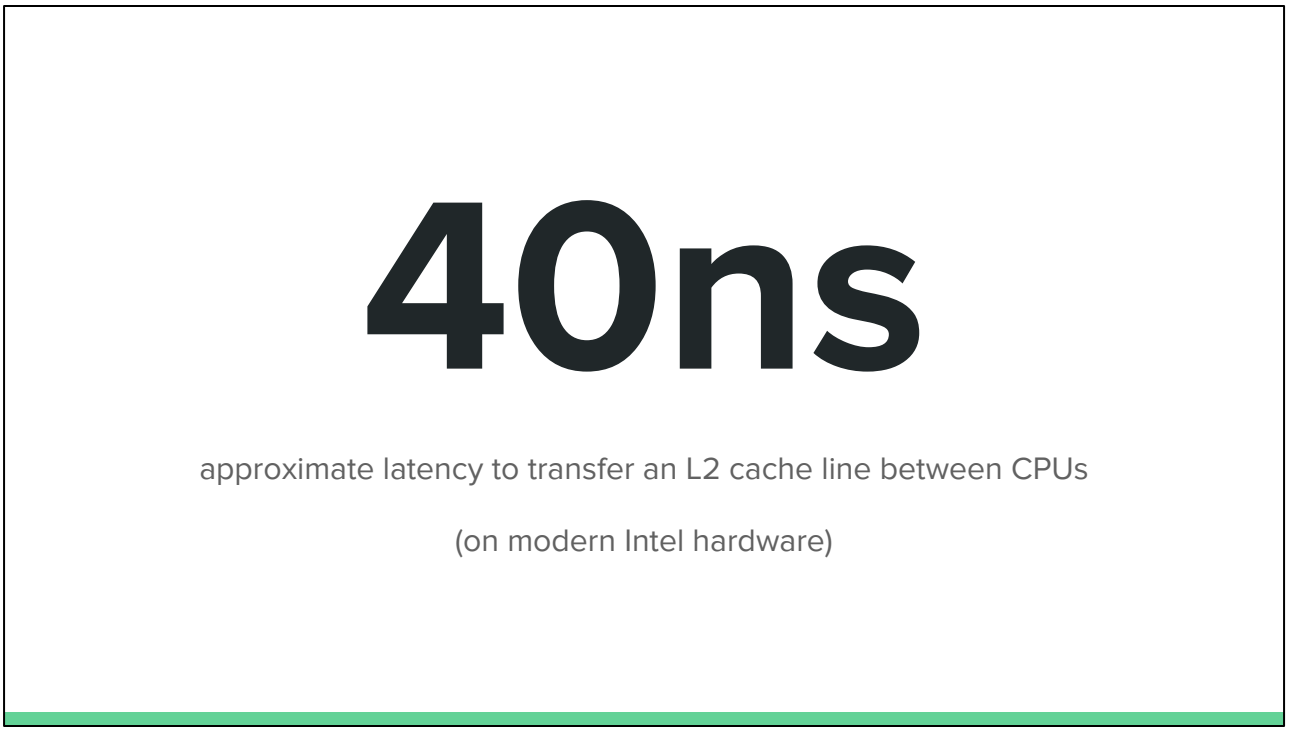
...because atomic.AddUint32 is slow

...because all of the cores are reading and writing the same address.

We traced a user-reported bottleneck in Google's Protocol Buffer package to a read-write mutex in the "reflect" package guarding a lazily-populated map. Threads running on different CPU cores were all trying to read-lock the mutex at the same time. To read-lock the mutex, each thread needs to update the reader count.

cache contention

This is a problem known as *cache contention*. When each core updates the count, it invalidates the local cache entries for that address in all the other cores, and marks *itself* as the owner of the up-to-date value.



The next core to update the count must fetch the value that the previous core wrote to its cache. On modern hardware, that takes about 40 nanoseconds. By itself, that's not going to make or break most applications…

O(N)

asymptotic latency of (*sync.RWMutex).RLock and RUnlock on N cores

(as of Go 1.9)

…but, only one core can update the counter at once. When multiple cores try to update it simultaneously, they have to wait in line. Operations that look like they should run in constant time instead become proportional to the number of CPU cores, and "concurrent" programs become sequential.

sync.Map is **in the standard library** so that we can **use it *in the standard library*.**

Coming back to my earlier question: why is sync.Map in the standard library? Because the problem we want to solve — cache contention — is *itself* in the standard library, in package-level mutexes.

The *only* special thing about sync.Map is its use in the standard library. It's not the *best possible* concurrent map for all use-cases. Fortunately, it should never appear in exported APIs, so if it isn't the best fit for your use-case, by all means *use something different*. Experimentation is a good thing!

What?

So, what are the use-cases sync.Map handles well, and what are the downsides?

stable keys

The Go 1.9 implementation of sync.Map is optimized for keys that are accessed repeatedly over time. *Loads* with *new* keys don't cause contention on their *own*. *Stores* with new keys *not only* contend with *each other*, but *also* temporarily cause subsequent Loads of *all* new keys to contend.

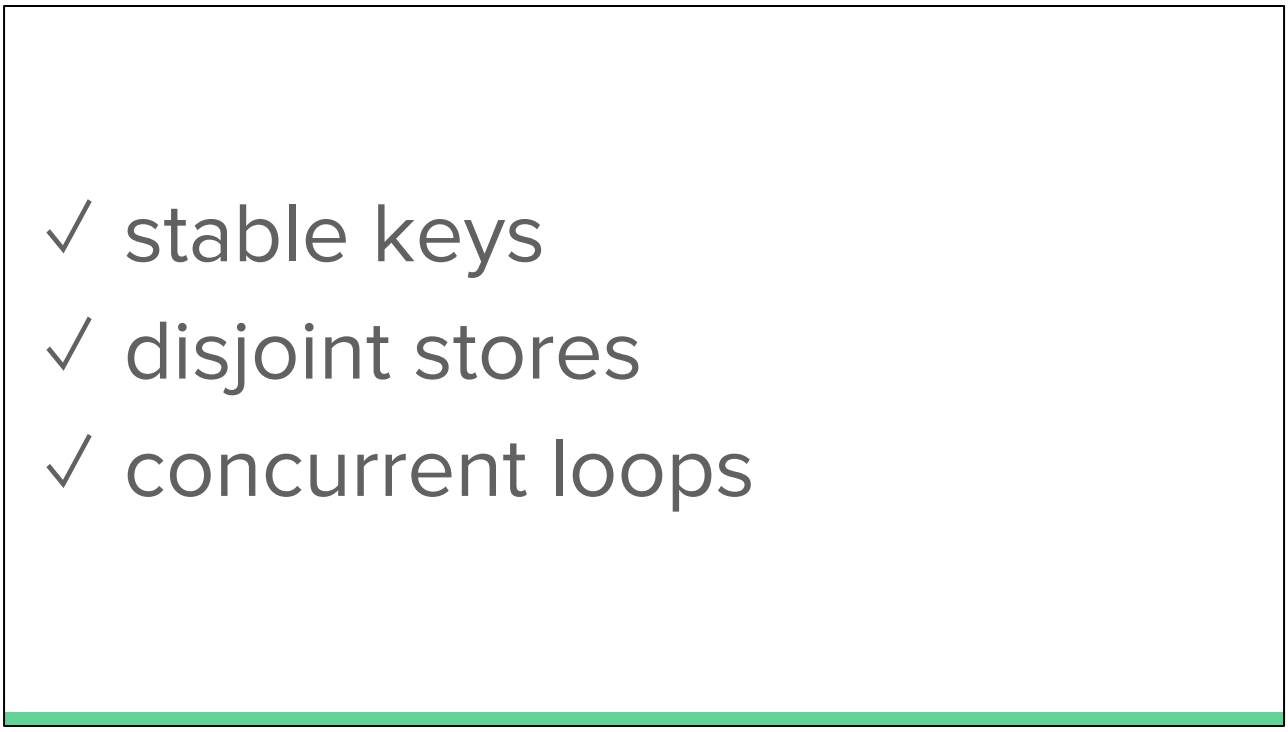
disjoint stores

Stores to the **same** key from **different** cores *always cause contention*, even if the key is not new. Keys should be write-once, read-many; or else the Stores for each key should be isolated to a single core at a time.

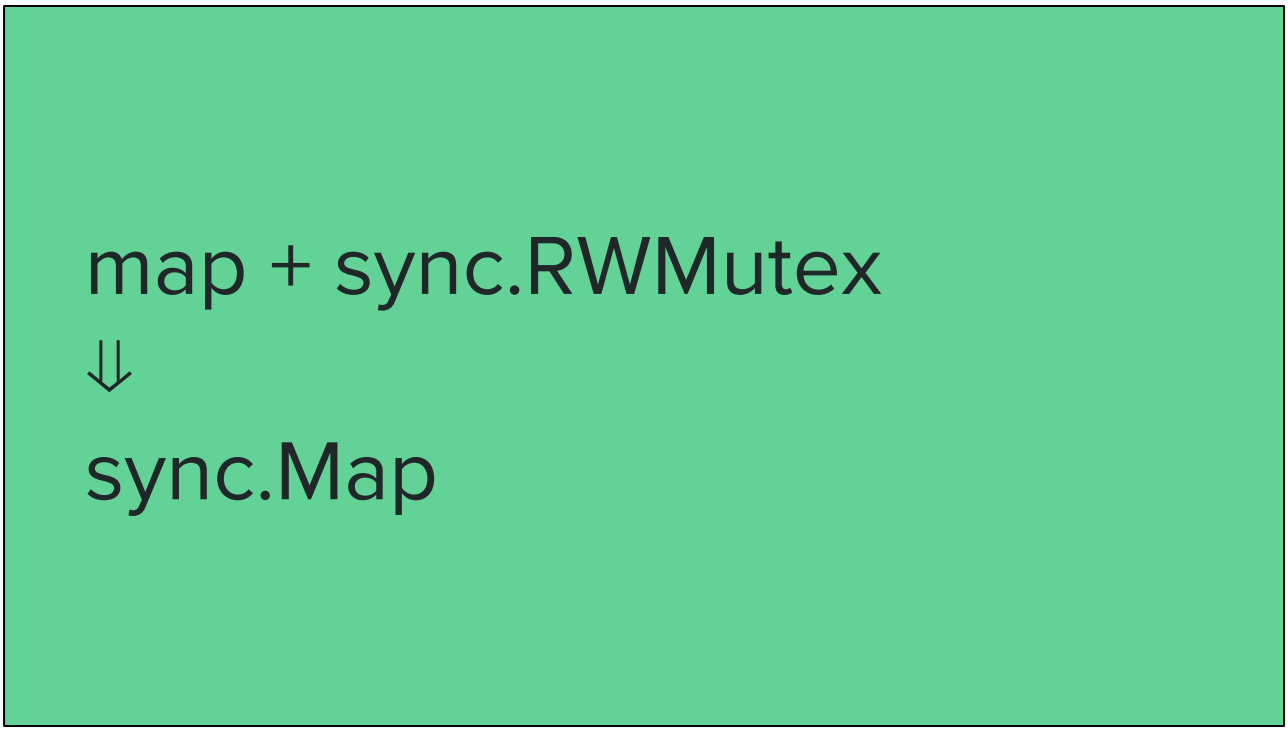
concurrent loops

Cache contention only matters if you have a lot of cores doing a lot of writes. If you aren't in a loop, you don't have "a lot of writes", and if you aren't using concurrency, you don't have "a lot of cores". For *either* of those cases, a read-write mutex will provide acceptable performance and better type-safety.

If you publish a *library*, you might not actually know whether it's used in concurrent loops. Base your decision on user feedback.



To summarize: the three ingredients for a sync.Map use-case are: stable keys, disjoint stores, and concurrent loops. So let's look at some cases in the standard library that illustrate these factors.



For each of these cases, we're using sync.Map to replace an ordinary map that was guarded by a read-write mutex. The read-write mutex is a good hint that we have *stable keys* and *relatively few stores*, so we just need to find the cases that may *also* have concurrent loops.

Lazy construction

- encoding/json.Encoder
  - type ⇒ encoder
- encoding/xml.Marshal
  - type ⇒ field metadata
- reflect
  - type* ⇒ type
    - ArrayOf, ChanOf, FuncOf, MapOf, PtrTo, SliceOf, StructOf

Our first use-cases include the original motivating example: lazily-populated maps, such as the ones in the "reflect" package. We know the keys are stable because the map entries persist forever: if the keys *weren't* stable, we would run out of memory.

Stores to these maps stop entirely in the steady state, but we might see a "warm-up" effect at the start of the program.

Registries

- encoding/gob.RegisterName
  - name ⇔ type
- mime
  - extensions ⇔ type
    - ExtensionsByType, TypeByExtension

The second group of use-cases in the standard library are registries populated during package init or early during execution.

There may be frequent *lookups* in these maps with unrecognized keys, but the set of keys that have been *registered* remains stable.

Disjoint counters

- expvar.Map

The final use-case for sync.Map is the "Map" type in the "expvar" package, which is used to count a variety of events for external monitoring.

The benefit of sync.Map for expvar.Map is less clear than for the previous use-cases, because *user* code — not the library — determines whether keys are stable, whether stores are disjoint, and whether accesses occur in concurrent loops.

If cache contention is
not the problem,
sync.Map is probably
not the solution.

If you have a "concurrent map" problem that doesn't suffer from cache contention, sync.Map probably won't help. It's just not optimized for anything else yet.

Costs

- overhead
  - pointer indirection (space and time)
  - binary bloat (instruction cache)
  - subtle interaction with escape analysis
- reduced type-safety
  - map[K]V ⇒ sync.Map (with type-assertions)
- limited API
  - no "len" or "swap"

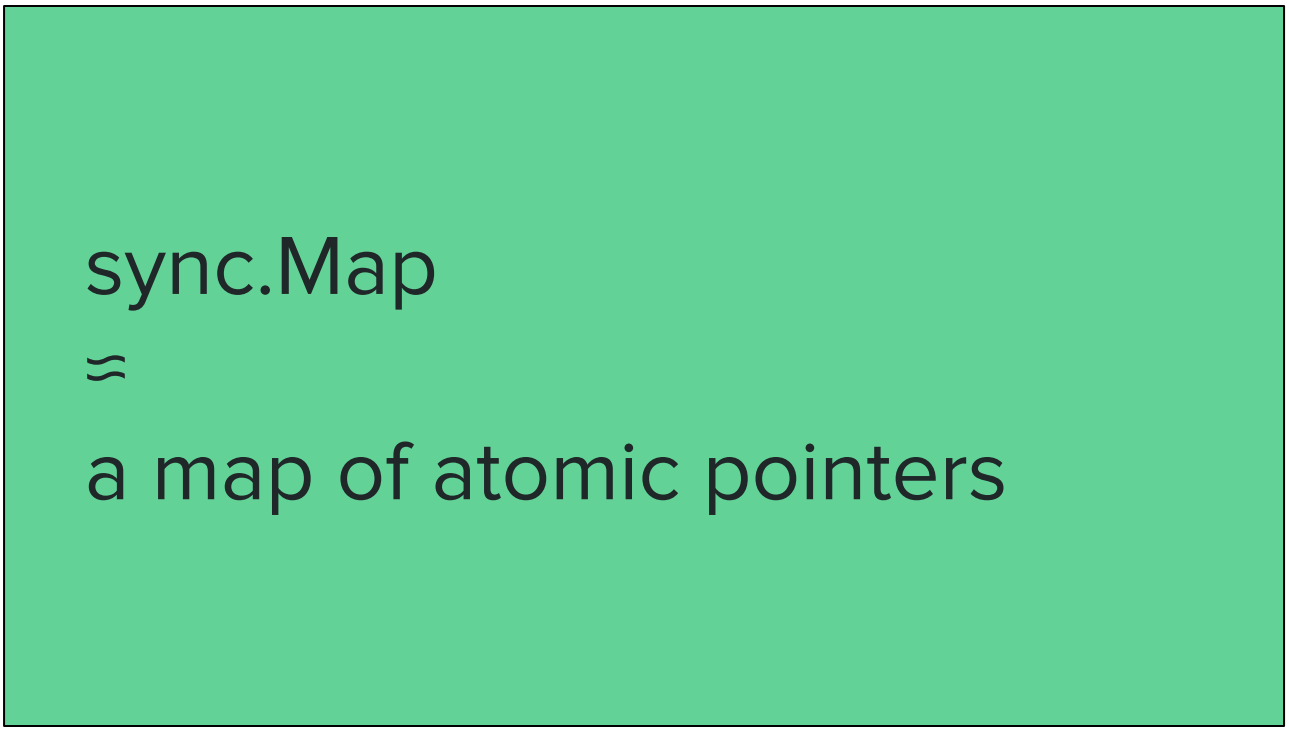
And sync.Map has some downsides. It may be slower than a read-write mutex for single-core access, and the generated code tends to be larger. It's designed to be compatible with the compiler's escape analysis, but the interaction is fairly subtle compared to a regular map.

Since the keys and values are stored as empty interfaces, converting a regular map to a sync.Map moves type-checking from compile time to run time.

And finally, the current API is very limited and it's driven by the use-cases in the standard library. We're being cautious about expanding the API, because that might also limit future optimization. If you have other use-cases that you think sync.Map should handle, please share them to help us understand the tradeoffs.

How?

How is sync.Map implemented in Go 1.9?



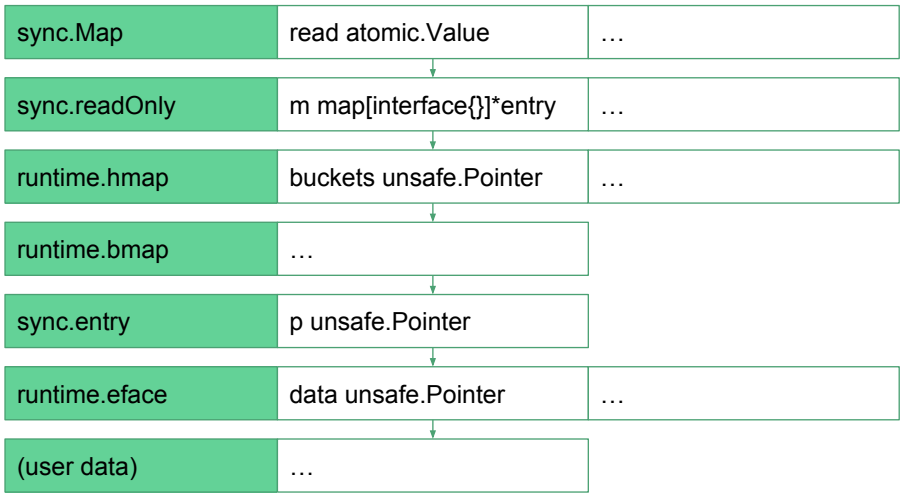
At a fundamental level, sync.Map is just a map of pointers that are each updated atomically.

actually two maps (one read-only, one read-write)

The implementation of sync.Map in 1.9 uses two maps: one read-only map stored in an atomic.Value, and one read-write map guarded by a Mutex. A boolean stored alongside the read-only map indicates whether the Mutex-guarded one contains any additional keys, so the fast path can often skip the Mutex entirely if the key isn't found.

Load operations that hit the Mutex path increment a counter. When there have been enough Loads to cover the cost of copying the map, the read-write map is promoted to read-only; at the next Store call, it's copied into a new read-write map.

Pointers all the way down



sync.Map methods receive pointers.

When we call them, we're passing a *pointer* to a struct
…with an atomic.Value, that *points* to a struct
…with a map, that points to a map header
…that points to a hash table
…whose elements are *themselves* pointers
…to atomic pointers
…to interface{} values that point to concrete values
(which may themselves be pointers).

That's a lot of pointers.

The internal maps store pointers to atomic pointers — rather than the atomic pointers directly — so that the sync.Map implementation doesn't need to know about the internals of the built-in map. That makes the implementation less efficient than it could be, but it allowed us to develop and publish a runtime-independent prototype in x/sync. Remember, the point is to turn an O(N) operation into O(1): there can be a lot of slop in the constant factors.

Room for improvement

Now that sync.Map is in the standard library,
we can peek inside the runtime.

- Flatten pointers.
- Optimize short-lived keys.
  - Use a Bloom filter instead of a boolean.
  - Shard the read-write map by key.
- ...and perhaps other optimizations.

Now that sync.Map is in the standard library, we can make use of *internal* knowledge of runtime data structures, including maps, to eliminate unnecessary pointer indirection, improving both speed and memory-efficiency.

In the runtime, we can also hash arbitrary keys efficiently. That may allow us to better optimize for short-lived keys, such as by applying Bloom filters or sharding across multiple locks.

By "us", I really mean "you": the Go team is hosting a Contributor Workshop today, and the Go 1.10 tree will be open for changes in early August. If you have ideas, please contribute!