Go 1.24 updates

Patryk Orwat

Lots of changes in different areas

Go 1.24 Release Notes

Table of Contents

More interactive changes: https://antonz.org/go-1-24/

Comment 1: Weak pointers

grahaman27 · 1mo ago

Weak pointers seems like it could cause a ton of bugs if people swapped to it hoping for memory improvements

22 Reply Award Share ...

Weak pointers - Java good analogy?

From Java tutorial:

WeakReferences can be used, for example, to store some information related to an object until the object get finalised. To do this you can implement a Map in which the ***keys*** *are wrapped in a WeakReference. As soon as GC will reclaim the key object, you can remove the value as well.*

Of course it can be done also using some notification mechanism, but using GC will be more robust and efficient. As example you can look at java.util.WeakHashMap but it is not thread-safe.

Source: https://medium.com/@ramtop/weak-soft-and-phantom-references-in-java-and-why-they-matter-c04bfc9dc792

Weak pointers - good for cache? NO!

From Java tutorial:

Why the WeakHashMap doesn't work for caching? First of all it wouldn't work anyway because it uses soft references for the keys and not for the map values. But additional to that, the garbage collector aggressively reclaims the memory that is referenced only by weak references. It means that once you lose the last strong reference to an object that is working as a key in a WeakHashMap, the garbage collector will soon reclaim that map entry.

Source: https://web.archive.org/web/20150403082405/http://www.codeinstructions.com/2008/09/weakhashmap-is-not-cache-understanding.html

Weak pointers - 3D rendering story time

Also, good story

https://stackoverflow.com/a/48048620

Weak pointers

And another interesting case - gauses:

All of the different forms of creating a gauge maintain only a weak reference to the object being observed, so as not to prevent garbage collection of the object.

Source: https://docs.micrometer.io/micrometer/reference/concepts/gauges.html

Weak pointers - summary

So when to use weak pointers?

In principle, where you want to store additional information to object that you don’t own.

In other cases, it depends but usual answer is NO - weak pointers are only for corner cases.

Weak pointers - good for additional data cache? Maybe..

Yes, you can walk around although cache keys cleanup is just bad…

Maybe addCleanup would help? If yes, what's the point of weak pointers…

A scheduled job that would iterate over map and delete entries to nil pointers.

That can allow to remove data quickly once strong references are removed.

```go
// Cache represents a thread-safe cache with weak pointers.
type Cache[K comparable, V any] struct {
    mu    sync.Mutex
    items map[K]weak.Pointer[V] // Weak pointers to cached objects
}

// NewCache creates a new generic Cache instance.
func NewCache[K comparable, V any]() *Cache[K, V] {
    return &Cache[K, V]{
        items: make(map[K]weak.Pointer[V]),
    }
}

// Get retrieves an item from the cache, if it's still alive.
func (c *Cache[K, V]) Get(key K) (*V, bool) {
    c.mu.Lock()
    defer c.mu.Unlock()

    // Retrieve the weak pointer for the given key
    ptr, exists := c.items[key]
    if !exists {
        return nil, false
    }

    // Attempt to dereference the weak pointer
    val := ptr.Value()
    if val == nil {
        // Object has been reclaimed by the garbage collector
        delete(c.items, key)
        return nil, false
    }

    return val, true
}
```

```go
// Set adds an item to the cache.
func (c *Cache[K, V]) Set(key K, value V) {
    c.mu.Lock()
    defer c.mu.Unlock()

    // Create a weak pointer to the value
    c.items[key] = weak.Make(&value)
}

func main() {
    // Create a cache with string keys and string values
    cache := NewCache[string, string]()

    // Add an object to the cache
    data := "cached data"
    cache.Set("key1", data)

    // Retrieve it
    if val, ok := cache.Get("key1"); ok {
        fmt.Println("Cache hit:", *val)
    } else {
        fmt.Println("Cache miss")
    }

    // Simulate losing the strong reference
    data = ""
    runtime.GC() // Force garbage collection

    // Try to retrieve it again
    time.Sleep(1 * time.Second)
    if val, ok := cache.Get("key1"); ok {
        fmt.Println("Cache hit:", *val)
    } else {
        fmt.Println("Cache miss")
    }
}
```

Ref https://dev.to/colindickson_78/weak-pointers-coming-in-go-124-imf

Comment 2: improved finalizer addCleanup

```go
type Blob []byte  3 usages  new *

func (b Blob) String() string {  new *
    return fmt.Sprintf( format: "Blob(%d KB)", len(b)/1024)
}

func newBlob(size int) *Blob {  1 usage  new *
    b := make([]byte, size*1024)
    for i := range size {
        b[i] = byte(i) % 255
    }
    return (*Blob)(&b)
}
```

```go
func main() {  Patryk Orwat *
    b := newBlob( size: 1000)
    now := time.Now()
    // Register a cleanup function to run
    // when the object is no longer reachable.
    runtime.AddCleanup(b, cleanup, now)

    time.Sleep(10 * time.Millisecond)
    b = nil
    runtime.GC()
    time.Sleep(10 * time.Millisecond)
}

func cleanup(created time.Time) {  1 usage  new *
    fmt.Printf(
        format: "object is cleaned up! lifetime = %dms\n",
        time.Since(created)/time.Millisecond,
    )
}
```

Source: https://antonz.org/go-1-24/#improved-finalizers

Comment 4: FIPS-140-3 compliance

The Go Cryptographic Module is a collection of standard library Go packages under crypto/internal/fips140/... that **implement FIPS 140-3 approved algorithms**.

Public API packages such as crypto/ecdsa and crypto/rand transparently use the Go Cryptographic Module to implement FIPS 140-3 algorithms.

Go Cryptographic Module version v1.0.0 **is currently under test** with a CMVP-accredited laboratory.

Source: https://go.dev/doc/security/fips140

Comment 5: Swiss table

Go 1.23:

```
Lookup time: 318.447458ms
Insertion time: 103.009625ms
Deletion time: 36.222416ms
```

Go 1.24

```
Lookup time: 237.979625ms
Insertion time: 60.243833ms
Deletion time: 58.681917ms
```

Source: https://www.bytesizego.com/blog/go-124-swiss-table-maps

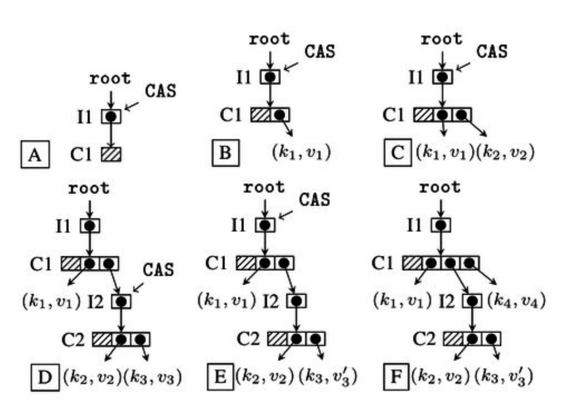
```go
func main() {
	// Create a large map
	m := make(map[int]int, 1_000_000)
	for i := 0; i < 1_000_000; i++ {
		m[i] = i
	}

	// Measure lookup performance
	start := time.Now()
	for i := 0; i < 10_000_000; i++ {
		_ = m[i%1_000_000]
	}
	fmt.Printf("Lookup time: %v\n", time.Since(start))

	// Measure insertion performance
	start = time.Now()
	for i := 1_000_000; i < 2_000_000; i++ {
		m[i] = i
	}
	fmt.Printf("Insertion time: %v\n", time.Since(start))

	// Measure deletion performance
	start = time.Now()
	for i := 0; i < 1_000_000; i++ {
		delete(m, i)
	}
	fmt.Printf("Deletion time: %v\n", time.Since(start))
}
```

Comment 6: sync.map is Ctrie



Source: https://en.wikipedia.org/wiki/Ctrie

Before Go 1.24

Underlying Structure

clike Copy

```
type Map struct {
   mu     Mutex
   read   atomic.Value // readOnly
   dirty  map[interface{}]*entry
   misses int
}

type readOnly struct {
   m       map[interface{}]*entry
   amended bool
}
```

- `mu` : A mutex used to protect access to `read` and `dirty` .
- `read` : A read-only data structure supporting concurrent reads using atomic operations. It stores a `readOnly` structure, which is a native map. The `amended` attribute marks whether the `read` and `dirty` data are consistent.
- `dirty` : A native map for reading and writing data, requiring locking to ensure data security.
- `misses` : A counter tracking how many times the read operation fails.

Entry Structure

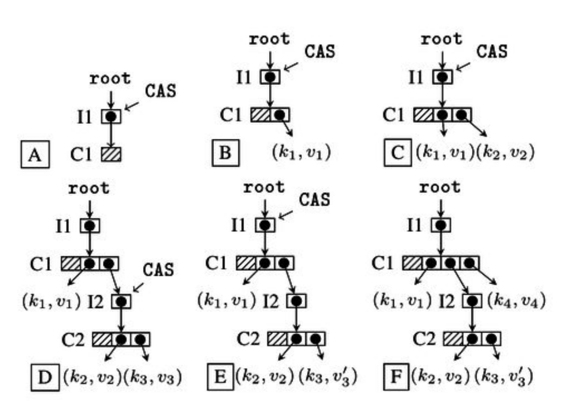
clike Copy

```
type entry struct {
   p unsafe.Pointer // *interface{}
}
```

- It contains a pointer `p` that points to the value stored for the element (key).

Source: https://reliasoftware.com/blog/go-sync-map

Comment 6:
sync.map is Ctrie



Source: https://en.wikipedia.org/wiki/Ctrie

	before sec/op	after sec/op	after vs base
MapLoadMostlyHits	7.870n ± 1%	8.415n ± 3%	+6.93%
MapLoadMostlyMisses	7.210n ± 1%	5.314n ± 2%	-26.28%
MapLoadOrStoreBalanced	360.10n ± 18%	71.78n ± 2%	-80.07%
MapLoadOrStoreUnique	707.2n ± 18%	135.2n ± 4%	-80.88%
MapLoadOrStoreCollision	5.089n ± 201%	3.963n ± 1%	-22.11%
MapLoadAndDeleteBalanced	17.045n ± 64%	5.280n ± 1%	-69.02%
MapLoadAndDeleteUnique	14.250n ± 57%	6.452n ± 1%	~
MapLoadAndDeleteCollision	19.34n ± 39%	23.31n ± 27%	~
MapRange	3.055µ ± 3%	1.918µ ± 2%	-37.23%
MapAdversarialAlloc	245.30n ± 6%	14.90n ± 23%	-93.92%
MapAdversarialDelete	143.550n ± 2%	8.184n ± 1%	-94.30%
MapDeleteCollision	9.199n ± 65%	3.165n ± 1%	-65.59%
MapSwapCollision	164.7n ± 7%	108.7n ± 36%	-34.01%
MapSwapMostlyHits	33.12n ± 15%	35.79n ± 9%	~
MapSwapMostlyMisses	604.5n ± 5%	280.2n ± 7%	-53.64%
MapCompareAndSwapCollision	96.02n ± 40%	69.93n ± 24%	-27.17%
MapCompareAndSwapNoExistingKey	6.345n ± 1%	6.202n ± 1%	-2.24%
MapCompareAndSwapValueNotEqual	6.121n ± 3%	5.564n ± 4%	-9.09%
MapCompareAndSwapMostlyHits	44.21n ± 13%	43.46n ± 11%	~
MapCompareAndSwapMostlyMisses	33.51n ± 6%	13.51n ± 5%	-59.70%
MapCompareAndDeleteCollision	27.85n ± 104%	31.02n ± 26%	~
MapCompareAndDeleteMostlyHits	50.43n ± 33%	109.45n ± 8%	+117.03%
MapCompareAndDeleteMostlyMisses	27.17n ± 7%	11.37n ± 3%	-58.14%
MapClear	300.2n ± 5%	124.2n ± 8%	-58.64%
geomean	50.38n	25.79n	-48.81%

The load-hit case (MapLoadMostlyHits) is slightly slower due to Swiss Tables improving the performance of the old sync.Map. Some benchmarks show a seemingly large slowdown, but that's mainly due to the fact that the new implementation shrinks promptly (as elements are deleted from the map), whereas the old one shrinks in generations (the dirty map needs to be promoted).

Source: https://antonz.org/go-1-24/#concurrent-hash-trie-map

Demo

Feat 1: [Tool dependencies]

Feat 2: [main module version]

Feat 3&4: working dir in test & test context