sync

Package sync go1.15.2 Latest

Published: **Sep 9, 2020** | License: BSD-3-Clause | Standard library

Overview

Package sync provides basic synchronization primitives such as mutual exclusion locks. Other than the Once and WaitGroup types, most are intended for use by low-level library routines. Higher-level synchronization is better done via channels and communication.

Values containing the types defined in this package should not be copied.

type Cond

```
type Cond struct {

    // L is held while observing or changing the condition
    L Locker
    // contains filtered or unexported fields
}
```

Cond implements a condition variable, a rendezvous point for goroutines waiting for or announcing the occurrence of an event.

Each Cond has an associated Locker L (often a *Mutex or *RWMutex), which must be held when changing the condition and when calling the Wait method.

A Cond must not be copied after first use.

func NewCond

```
func NewCond(l Locker) *Cond
```

NewCond returns a new Cond with Locker l.

func (*Cond) Broadcast

```
func (c *Cond) Broadcast()
```

Broadcast wakes all goroutines waiting on c.

It is allowed but not required for the caller to hold c.L during the call.

func (*Cond) Signal

```
func (c *Cond) Signal()
```

Signal wakes one goroutine waiting on c, if there is any.

It is allowed but not required for the caller to hold c.L during the call.

func (*Cond) Wait

```
func (c *Cond) Wait()
```

Wait atomically unlocks c.L and suspends execution of the calling goroutine. After later resuming execution, Wait locks c.L before returning. Unlike in other systems, Wait cannot return unless awoken by Broadcast or Signal.

Because c.L is not locked when Wait first resumes, the caller typically cannot assume that the condition is true when Wait returns. Instead, the caller should Wait in a loop:

```
c.L.Lock()
for !condition() {
    c.Wait()
}
... make use of condition ...
c.L.Unlock()
```

type Locker

```
type Locker interface {
    Lock()
    Unlock()
}
```

A Locker represents an object that can be locked and unlocked.

type Map

```
type Map struct {
    // contains filtered or unexported fields
}
```

Map is like a Go map[interface{}]interface{} but is safe for concurrent use by multiple goroutines without additional locking or coordination. Loads, stores, and deletes run in amortized constant time.

The Map type is specialized. Most code should use a plain Go map instead, with separate locking or coordination, for better type safety and to make it easier to maintain other invariants along with the map content.

The Map type is optimized for two common use cases: (1) when the entry for a given key is only ever written once but read many times, as in caches that only grow, or (2) when multiple goroutines read, write, and overwrite entries for disjoint sets of keys. In these two cases, use of a Map may significantly reduce lock contention compared to a Go map paired with a separate Mutex or RWMutex.

The zero Map is empty and ready for use. A Map must not be copied after first use.

func (*Map) Delete

```
func (m *Map) Delete(key interface{})
```

Delete deletes the value for a key.

func (*Map) Load

```
func (m *Map) Load(key interface{}) (value interface{}, ok bool)
```

Load returns the value stored in the map for a key, or nil if no value is present. The ok result indicates whether value was found in the map.

func (*Map) LoadAndDelete

```
func (m *Map) LoadAndDelete(key interface{}) (value interface{}, loaded bool)
```

LoadAndDelete deletes the value for a key, returning the previous value if any. The loaded result reports whether the key was present.

func (*Map) LoadOrStore

```
func (m *Map) LoadOrStore(key, value interface{}) (actual interface{}, loaded bool)
```

LoadOrStore returns the existing value for the key if present. Otherwise, it stores and returns the given value. The loaded result is true if the value was loaded, false if stored.

func (*Map) Range

```
func (m *Map) Range(f func(key, value interface{}) bool)
```

Range calls f sequentially for each key and value present in the map. If f returns false, range stops the iteration.

Range does not necessarily correspond to any consistent snapshot of the Map's contents: no key will be visited more than once, but if the value for any key is stored or deleted concurrently, Range may reflect any mapping for that key from any point during the Range call.

Range may be O(N) with the number of elements in the map even if f returns false after a constant number of calls.

func (*Map) Store

```
func (m *Map) Store(key, value interface{})
```

Store sets the value for a key.

type Mutex

```
type Mutex struct {
    // contains filtered or unexported fields
}
```

A Mutex is a mutual exclusion lock. The zero value for a Mutex is an unlocked mutex.

A Mutex must not be copied after first use.

func (*Mutex) Lock

```
func (m *Mutex) Lock()
```

Lock locks m. If the lock is already in use, the calling goroutine blocks until the mutex is available.

func (*Mutex) Unlock

```
func (m *Mutex) Unlock()
```

Unlock unlocks m. It is a run-time error if m is not locked on entry to Unlock.

A locked Mutex is not associated with a particular goroutine. It is allowed for one goroutine to lock a Mutex and then arrange for another goroutine to unlock it.

type Once

```
type Once struct {
    // contains filtered or unexported fields
}
```

Once is an object that will perform exactly one action.

func (*Once) Do

```
func (o *Once) Do(f func())
```

Do calls the function f if and only if Do is being called for the first time for this instance of Once. In other words, given

```
var once Once
```

if once.Do(f) is called multiple times, only the first call will invoke f, even if f has a different value in each invocation. A new instance of Once is required for each function to execute.

Do is intended for initialization that must be run exactly once. Since f is niladic, it may be necessary to use a function literal to capture the arguments to a function to be invoked by Do:

```
config.once.Do(func() { config.init(filename) })
```

Because no call to Do returns until the one call to f returns, if f causes Do to be called, it will deadlock.

If f panics, Do considers it to have returned; future calls of Do return without calling f.

type Pool

```
type Pool struct {

    // New optionally specifies a function to generate
    // a value when Get would otherwise return nil.
    // It may not be changed concurrently with calls to Get.
    New func() interface{}
    // contains filtered or unexported fields
}
```

A Pool is a set of temporary objects that may be individually saved and retrieved.

Any item stored in the Pool may be removed automatically at any time without notification. If the Pool holds the only reference when this happens, the item might be deallocated.

A Pool is safe for use by multiple goroutines simultaneously.

Pool's purpose is to cache allocated but unused items for later reuse, relieving pressure on the garbage collector. That is, it makes it easy to build efficient, thread-safe free lists. However, it is not suitable for all free lists.

An appropriate use of a Pool is to manage a group of temporary items silently shared among and potentially reused by concurrent independent clients of a package. Pool provides a way to amortize allocation overhead across many clients.

An example of good use of a Pool is in the fmt package, which maintains a dynamically-sized store of temporary output buffers. The store scales under load (when many goroutines are actively printing) and shrinks when quiescent.

On the other hand, a free list maintained as part of a short-lived object is not a suitable use for a Pool, since the overhead does not amortize well in that scenario. It is more efficient to have such objects implement their own free list.

A Pool must not be copied after first use.

func (*Pool) Get

```
func (p *Pool) Get() interface{}
```

Get selects an arbitrary item from the Pool, removes it from the Pool, and returns it to the caller. Get may choose to ignore the pool and treat it as empty. Callers should not assume any relation between values passed to Put and the values returned by Get.

If Get would otherwise return nil and p.New is non-nil, Get returns the result of calling p.New.

func (*Pool) Put

```
func (p *Pool) Put(x interface{})
```

Put adds x to the pool.

type RWMutex

```
type RWMutex struct {
    // contains filtered or unexported fields
}
```

A RWMutex is a reader/writer mutual exclusion lock. The lock can be held by an arbitrary number of readers or a single writer. The zero value for a RWMutex is an unlocked mutex.

A RWMutex must not be copied after first use.

If a goroutine holds a RWMutex for reading and another goroutine might call Lock, no goroutine should expect to be able to acquire a read lock until the initial read lock is released. In particular, this prohibits recursive read locking. This is to ensure that the lock eventually becomes available; a blocked Lock call excludes new readers from acquiring the lock.

func (*RWMutex) Lock

```
func (rw *RWMutex) Lock()
```

Lock locks rw for writing. If the lock is already locked for reading or writing, Lock blocks until the lock is available.

func (*RWMutex) RLock

```
func (rw *RWMutex) RLock()
```

RLock locks rw for reading.

It should not be used for recursive read locking; a blocked Lock call excludes new readers from acquiring the lock. See the documentation on the RWMutex type.

func (*RWMutex) RLocker

```
func (rw *RWMutex) RLocker() Locker
```

RLocker returns a Locker interface that implements the Lock and Unlock methods by calling rw.RLock and rw.RUnlock.

func (*RWMutex) RUnlock

```
func (rw *RWMutex) RUnlock()
```

RUnlock undoes a single RLock call; it does not affect other simultaneous readers. It is a run-time error if rw is not locked for reading on entry to RUnlock.

func (*RWMutex) Unlock

```
func (rw *RWMutex) Unlock()
```

Unlock unlocks rw for writing. It is a run-time error if rw is not locked for writing on entry to Unlock.

As with Mutexes, a locked RWMutex is not associated with a particular goroutine. One goroutine may RLock (Lock) a RWMutex and then arrange for another goroutine to RUnlock (Unlock) it.

type WaitGroup

```
type WaitGroup struct {
    // contains filtered or unexported fields
}
```

A WaitGroup waits for a collection of goroutines to finish. The main goroutine calls Add to set the number of goroutines to wait for. Then each of the goroutines runs and calls Done when finished. At the same time, Wait can be used to block until all goroutines have finished.

A WaitGroup must not be copied after first use.

func (*WaitGroup) Add

```
func (wg *WaitGroup) Add(delta int)
```

Add adds delta, which may be negative, to the WaitGroup counter. If the counter becomes zero, all goroutines blocked on Wait are released. If the counter goes negative, Add panics.

Note that calls with a positive delta that occur when the counter is zero must happen before a Wait. Calls with a negative delta, or calls with a positive delta that start when the counter is greater than zero, may happen at any time. Typically this means the calls to Add should execute before the statement creating the goroutine or other event to be waited for. If a WaitGroup is reused to wait for several independent sets of events, new Add calls must happen after all previous Wait calls have returned. See the WaitGroup example.

func (*WaitGroup) Done

```
func (wg *WaitGroup) Done()
```

Done decrements the WaitGroup counter by one.

func (*WaitGroup) Wait

```
func (wg *WaitGroup) Wait()
```

Wait blocks until the WaitGroup counter is zero.