標準パッケージ初のgenerics利用事例 "sync/atomic.Pointer"

Money Forward, Inc.

uji

Money Forward

- Software engineer
  @Money Forward, Inc.(Osaka)
- Kyoto.go

uji

@uji_rb

Money Forward

今日のゴール

- sync/atomicのGo 1.19のアップデート
  について、背景と内容を理解してもらう

- Goの標準パッケージ初の
  generics利用事例に触れてもらう

sync/atomicとは

- Goの標準パッケージのうちの一つ
- システムの他からは単体に見え、
  割り込みができない操作を扱う
  (「不可分操作」や「アトミック操作」と呼ばれる)
- goroutine間でデータを同期するための
  最もプリミティブな仕組み
  - mutexなどの他の排他制御と比べると
    競合発生頻度が低い場合、余分なロックが少なく性能が良い

Go1.18までのsync/atomic

Money Forward

従来のAPI (int64, unsafe.Pointerのみ抜粋)

```
// Int64
func AddInt64(addr *int64, delta int64) (new int64)
func CompareAndSwapInt64(addr *int64, old, new int64) (swapped bool)
func LoadInt64(addr *int64) (val int64)
func StoreInt64(addr *int64, val int64)
func SwapInt64(addr *int64, new int64) (old int64)

// Pointer
func CompareAndSwapPointer(addr *unsafe.Pointer, old, new unsafe.Pointer) (swapped bool)
func LoadPointer(addr *unsafe.Pointer) (val unsafe.Pointer)
func StorePointer(addr *unsafe.Pointer, val unsafe.Pointer)
func SwapPointer(addr *unsafe.Pointer, new unsafe.Pointer) (old unsafe.Pointer)
```

Go1.18までのsync/atomic

Money Forward

従来のAPI (int64, unsafe.Pointerのみ抜粋)

```
// Int64
func AddInt64(addr *int64, delta int64) (new int64)
func CompareAndSwapInt64(addr *int64, old, new int64) (swapped bool)
func LoadInt64(addr *int64) (val int64)
func StoreInt64(addr *int64, val int64)
func SwapInt64(addr *int64, new int64) (old int64)

// Pointer
func CompareAndSwapPointer(addr *unsafe.Pointer, old, new unsafe.Pointer) (swapped bool)
func LoadPointer(addr *unsafe.Pointer) (val unsafe.Pointer)
func StorePointer(addr *unsafe.Pointer, val unsafe.Pointer)
func SwapPointer(addr *unsafe.Pointer, new unsafe.Pointer) (old unsafe.Pointer)
```

操作したい値を
第一引数に渡す関数API

Go1.19のアップデート

- 型が追加され、メソッド経由で操作が行えるようになった
- Pointer型は型パラメータで型を指定できる
  (標準パッケージでgenericsが使われた初めての事例)

```
type Int64 struct{/* unexported fields */}
  func (x *Int64) Add(delta int64) (new int64)
  func (x *Int64) CompareAndSwap(old, new int64) (swapped bool)
  func (x *Int64) Load() int64
  func (x *Int64) Store(val int64)
  func (x *Int64) Swap(new int64) (old int64)

type Pointer[T any] struct{/* unexported fields */}
  func (x *Pointer[T]) CompareAndSwap(old, new *T) (swapped bool)
  func (x *Pointer[T]) Load() *T
  func (x *Pointer[T]) Store(val *T)
  func (x *Pointer[T]) Swap(new *T) (old *T)
```

Go1.19のアップデート

Money Forward

- 型が追加され、メソッド経由で操作が行えるようになった
- Pointer型は型パラメータで型を指定できる
  (標準パッケージでgenericsが使われた初めての事例)

```
type Int64 struct{/* unexported fields */}
  func (x *Int64) Add(delta int64) (new int64)
  func (x *Int64) CompareAndSwap(old, new int64) (swapped bool)
  func (x *Int64) Load() int64
  func (x *Int64) Store(val int64)
  func (x *Int64) Swap(new int64) (old int64)

type Pointer[T any] struct{/* unexported fields */}
  func (x *Pointer[T]) CompareAndSwap(old, new *T) (swapped bool)
  func (x *Pointer[T]) Load() *T
  func (x *Pointer[T]) Store(val *T)
  func (x *Pointer[T]) Swap(new *T) (old *T)
```

アップデートの背景

- Go1.19でメモリモデルが定義が更新された
  - データ競合を避けるために適切な同期を使用することがより強く推奨されるようになった
  - アトミック操作は有効な手段の一つ

- 現状のAPIは、**通常の変数をアトミック関数に渡す**シグネチャになっており、どこがアトミック操作されるべきかがわかりにくい課題がある

https://go.dev/ref/mem#overview
https://research.swtch.com/gomm#maybe

追加された型を使うメリット

例) int32型の変数を並列処理でカウントアップする

~go1.18

```
var v int32
for i := 0; i < 100; i++ {
  wg.Add(1)
  go func() {
    atomic.AddInt32(&v, 1)
    wg.Done()
  }()
}
```

go1.19

```
var v atomic.Int32
for i := 0; i < 100; i++ {
  wg.Add(1)
  go func() {
    v.Add(1)
    wg.Done()
  }()
}
```

追加された型を使うメリット

例) int32型の変数を並列処理でカウントアップする

~go1.18

```
var v int32
for i := 0; i < 100; i++ {
  wg.Add(1)
  go func() {
    atomic.AddInt32(&v, 1)
    wg.Done()
  }()
}
```

go1.19

```
var v atomic.Int32
for i := 0; i < 100; i++ {
  wg.Add(1)
  go func() {
    v.Add(1)
    wg.Done()
  }()
}
```

アトミック操作の対象がatomic.Xxxになるため
どの変数がアトミック操作されるべきかがわかりやすくなった

追加された型を使うメリット

例) math/big.Int型の変数を並列処理でカウントアップする(~go1.18)

```
func main() {
  v := big.NewInt(0)
  delta := big.NewInt(1)
  var wg sync.WaitGroup

  for i := 0; i < 100; i++ {
    wg.Add(1)
    go func() {
      for {
        newVal := big.NewInt(0)
        oldVal := atomic.LoadPointer((*unsafe.Pointer)(unsafe.Pointer(&v)))
        newVal.Add((*big.Int)(oldVal), delta)
        if atomic.CompareAndSwapPointer((*unsafe.Pointer)(unsafe.Pointer(&v)), oldVal, unsafe.Pointer(newVal)) {
          break
        }
      }
      wg.Done()
    }()
  }

  wg.Wait()
  fmt.Println((*big.Int)(atomic.LoadPointer((*unsafe.Pointer)(unsafe.Pointer(&v))))) // 100
}
```

型アサーションが多く登場し、難解…

追加された型を使うメリット

Money Forward

例) math/big.Int型の変数を並列処理でカウントアップする(go1.19)

```
func main() {
  var v atomic.Pointer[big.Int]
  // Storeによる初期化が必要
  v.Store(big.NewInt(0))
  var wg sync.WaitGroup
  for i := 0; i < 100; i++ {
    wg.Add(1)
    go func() {
      for {
        newVal := big.NewInt(0)
        oldVal := v.Load() // 値の読み取り
        newVal.Add(oldVal, big.NewInt(1))
        // 他の並列操作で置換が行われていない場合のみ
        // 値を置換しループを終了する
        if v.CompareAndSwap(oldVal, newVal) {
          break
        }
      }
      wg.Done()
    }()
  }
  wg.Wait()
  fmt.Println(v.Load())
}
```

https://go.dev/play/p/L7I7RdSxgnE

追加された型を使うメリット

Money Forward

例) math/big.Int型の変数を並列処理でカウントアップする(go1.19)

```
func main() {
  var v atomic.Pointer[big.Int]
  // Storeによる初期化が必要
  v.Store(big.NewInt(0))
  var wg sync.WaitGroup
  for i := 0; i < 100; i++ {
    wg.Add(1)
    go func() {
      for {
        newVal := big.NewInt(0)
        oldVal := v.Load() // 値の読み取り
        newVal.Add(oldVal, big.NewInt(1))
        // 他の並列操作で置換が行われていない場合のみ
        // 値を置換しループを終了する
        if v.CompareAndSwap(oldVal, newVal) {
          break
        }
      }
      wg.Done()
    }()
  }
  wg.Wait()
  fmt.Println(v.Load())
}
```

操作する際に型アサーションをせずに
指定した型の値を使うことができる

atomic.Valueを使う実装方法もあるが
Load時にany型で値が返されるため
型アサーションする必要がある

https://go.dev/play/p/L7I7RdSxgnE

なぜ Pointer[T]？

Money Forward

boolやint32なども含めて
atomic.Value[T] などでも良かったのでは？

- Bool や Pointer は数値では無いので Add メソッドを持たない
  しかし数値型は Add を持つべき

- atomic.Xxx[T] のような単一の型で
  これらの型の制約を表現する方法が現状ない

- 無理に単一の型に統合するより、atomic.Bool や atomic.Int32 などに分けて
  提供した方がユーザーフレンドリーとGoチームは考えた
  これらは使われる頻度も高く、genericsを考えなくて良い場面が増える

https://github.com/golang/go/issues/50860

古いAPIとの使い分け

古いAPIでやれることは追加された型で全て実現できる
→deprecatedにして良いのでは？

- 古いAPIは警告するほど壊れてはいないため
  deprecated にはならず残る
  (deprecatedにするproposalを出したところdeclineになった)

- 新しく実装する際は利便性の高い
  追加された型のAPIを使うのが良い

https://github.com/golang/go/issues/55302

まとめ

- メモリモデルの定義が更新され、
  どこがアトミック操作されるべきかを
  わかりやすくする必要性が出てきた

- 追加された型を利用することで、
  どこがアトミック操作されるべきかがわかりやすくなった

- 新しく実装する際は利便性の高い
  追加された型のAPIを使うのが良い

- 古いAPIはdeprecated にはならず残る

参考文献

- sync/atomic: add typed atomic values
  https://github.com/golang/go/issues/50860
- The Go Memory Model
  https://go.dev/ref/mem
- Updating the Go Memory Model
  https://research.swtch.com/gomm
- Go1.19 Memory Modelを読む【入門編】
  https://docs.google.com/presentation/d/1jJvL__7VYHs4Qv-mGsuAThXpWiSpek27wXIn9K2PVJU
- Goならわかるシステムプログラミング 第2版
  https://www.lambdanote.com/products/go-2
- Go 1.19のメモリ周りの更新
  https://future-architect.github.io/articles/20220808a
- proposal: sync/atomic: deprecate AddXxx, CompareAndSwapXxx, LoadXxx, StoreXxx, SwapXxx
  https://github.com/golang/go/issues/55302